K. R. Chowdhary

# Fundamentals of Artificial Intelligence

# Fundamentals of Artificial Intelligence

K. R. Chowdhary

# Fundamentals of Artificial Intelligence

K. R. Chowdhary
Department of Computer Science
and Engineering
Jodhpur Institute of Engineering
and Technology
Jodhpur, Rajasthan, India

*To my wife*
*Suman Lata*

# Preface

The course of Artificial Intelligence (AI), is one of the standard course in all computer science curricula at undergraduate programs, in many of masters programs, as well as taught a course in many of multi-disciplinary majors like Masters in Mathematics, Masters in Information Technology, and in non-CS Programs like Electrical Engineering, Electronics and Communication Engineering, and in some Institutions it is taught by the name Intelligent Systems. The concepts and theories of AI are often applied in many Engineering disciplines, like automation, production, packing, optimizations, planning; in routing of information and goods (in transporting); in message communications and message filtering; in natural language translation, speech synthesis, speech recognition, search engines, information retrieval, information extraction, information summarization, information filtering; in robotics, computer/machine vision systems; expert systems in medical diagnosis, fault diagnosis in machines, surveillance, remote sensing, automated vehicles; in computer games, combinatorial games; in recent use in on-line sales, on-line education, and many more.

ACM Curricula 2013 defines the scope of Artificial Intelligence as follows:[1]
*Intelligent Systems* (IS): Artificial intelligence (AI) is the study of solutions for problems that are difficult or impractical to solve with traditional methods. It is used pervasively in support of everyday applications such as email, word-processing and search, as well as in the design and analysis of autonomous agents that perceive their environment and interact rationally with the environment. The solutions rely on a broad set of general and specialized knowledge representation schemes, problem solving mechanisms, and learning techniques. They deal with sensing (e.g., speech recognition, natural language understanding, computer vision), problem-solving (e.g., search, planning), and acting (e.g., robotics) and the architectures needed to support them (e.g., agents, multi-agents). The study of Artificial

---

[1] https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf.

Intelligence prepares the student to determine when an AI approach is appropriate for a given problem, identify the appropriate representation and reasoning mechanism, and implement and evaluate it.

## Contents

This book covers in enough details all the topics recommended in the latest ACM "Computer Science Curricula 2013" (available at the time of writing this manuscript). The topics as per the curricula 2013 and covered in this text are as follows:

*Fundamental Issues*
*Basic Search Strategies*
*Basic Knowledge Representation and Reasoning*
*Basic Machine Learning*
*Advanced Search*
*Advanced Representation and Reasoning*
*Reasoning under Uncertainty*
*Agents*
*Natural Language Processing*
*Advanced Machine Learning*
*Perception and Computer Vision*

In addition, looking to their importance and scope in AI, this text covers following additional topics.

*Logic Programming and Prolog*
*Constraint Satisfaction Problems*
*Game Theory*
*Data Mining*
*Information Retrieval*
*Speech Recognition*

This book is written, keeping in mind the diversity of its readership—the concepts and fundamental principles are given highest significance, more and more explanation is based on figures and solved examples. This is supplemented with exhaustive list of exercises at the end of each chapter, which are aimed to cover wide range of problem solving skills for a learner. Each chapter begins with introduction to the topic of chapter, its scope, and relation to other fields of AI, supported with list of applications to keep constant motivation of the learner from chapter to chapter, followed with learning outcome of that chapter. To maintain a consistent interest of the student as well for ease of understanding, each chapter is organized to gradually pick up—from simple concepts to advanced theory and applications. For curious and interested readers, each chapter is supplemented with exhaustive list of original source references, in the form of recent articles published in Journals, Transactions, and Proceedings of Conferences of AI and CS.

The text is designed to be self taught, concept driven, simple language, with large number of exercises both solved and unsolved. The simplicity of English language used helps in easy grasping by an average student, and equally suited for non-English speaking students having medium of instruction as English. It covers all major areas of AI presently in focus and that belong to recent developments, including machine learning, machine vision, IR, agents, etc.

The contents of this text are based on teaching this course by the author for many years at M.B.M. Engineering College, Jodhpur (India), as well as based on the course of AI taught to undergraduate students at Indian Institute of Technology Jodhpur (IITJ) from the session 2011–12 to 2015–16 as a visiting faculty, the time during which the book came into this shape. The slides of the topics taught at IITJ can be referred by reader in the author's website (http://www.krchowdhary.com/). The contents of this text is class room tested over the years, and has been refined and improved again and again over these years of teaching; all the exercises are also class room tested.

## Intended Audience and Prerequisites

The book is intended primarily for students majoring in computer science at under-graduate and graduate level, but will also be of interest as a foundation to researchers in any of the discipline of AI, either for self study or reference. In general, it is suitable for all AI curricula of undergraduate courses in Engineering, particularly, Computer Science and Engineering, Information Technology, Electrical Engineering, and non-engineering curricula, like Master in Computer Applications, Master of Science. Some of the chapters, particularly, the Chaps. 13–21 may form course contents of graduate curricula. The chapters can be read in the order they appear in this text. Readers may correspond to the author (at mail-ids: kr.chowdhary@iitj.ac.in or kr.chowdhary@gmail.com) to point out the errors, suggestions/criticism, so as to make improvements in the script for future editions.

The prerequisites for this course are: mathematical logic, discrete mathematical structures, and data structures with basic level of programming. This course can be taught at third or final year of undergraduate program by name as AI or Intelligent Systems.

Since all the solved, as well as many unsolved problems in the book are classroom tested, these can go in parallel to the respective chapter's delivery. A keen learner is recommended to refer to the exhaustive list of references provided at the end of each chapter for more details.

## Examples, Exercises, and Learning Curve

The exercises are designed to continuously develop and strengthen the ideas presented in the book's many proofs and examples. More than 250 exercises and solved examples have been provided in the text to rigorously practice the material presented, and to consolidate the principles presented to strengthen the over all understanding of topics.

## Supplementary Reading Material

Lecture notes, classroom slides, problems—both solved and unsolved—quizzes, exercises, questions papers, and solutions of mid-semesters and end-semesters of the AI courses taught at IITJ, and list of semester projects along with their guidelines, is available in the author's website http://www.krchowdhary.com, can be used as supplementary reading material by the learner. In due course, and time to time, more help and learner's material shall be added by the author for this text on this URL. The students may send their feedback to the author for suggestions/errors/improvements in the contents of AI-related text on this website, their efforts shall be an invaluable contribution to improve the quality of this book.

Jodhpur, India                                                            Prof. K. R. Chowdhary

# Acknowledgements

# Contents

# About the Author

**Prof. K. R. Chowdhary** is currently a Professor of Computer Science at JGI, Jodhpur. He is the former Professor and Head, Department of Computer Science & Engineering, M.B.M. Engineering College, Jodhpur, India. He completed his Ph.D. from the same university in 2004. Prof. Chowdhary has over 35 years of teaching and research experience. He had also taught at IIT Jodhpur (2010–2017), and served as the Director at JIET Group of Institutions, Jodhpur from 2014 to 2019. He was honored with senior membership award of Association for Computing Machinery in 2008 and Eminent Computer Engineer's Award from the Institute of Engineers, India in 2011. He has authored several papers published in national and international journals and conference proceedings. His areas of specialization include: Discrete Mathematical Structures, Theory of Computation, AI, Machine Learning, Natural Language and Speech Processing.

# Acronyms

| | |
|---|---|
| ABL | Argument-based Learning |
| ABOX | Assertion Box |
| ACL | Agent Communication Language |
| ADJ | Adjective |
| ADV | Adverb |
| AI | Artificial Intelligence |
| ANNs | Artificial Neural networks |
| AQE | Automatic Query Expansion |
| ASR | Automatic Speech Recognition |
| ATR | Automatic Target Recognition |
| AUX | Auxiliary verb |
| BFS | Breadth-first Search |
| BOW | Bag-of-words |
| CBR | Case-based Reasoning |
| CD | Conceptual Dependency |
| CF | Cluster Feature |
| CFG | Context-free Grammar |
| CG | Conceptual Graph |
| CGT | Combinatorial Game Theory |
| CLIR | Cross-language IR |
| CNF | Conjunctive Normal Forms |
| CSP | Constraint Satisfaction Problem |
| CYC | enCYClopedia ontology |
| DAG | Directed Acyclic Graph |
| DET | Determiner |
| DFID | Depth-First Iterative Deepening |
| DFS | Depth-first Search |
| DL | Description Logic |
| DNF | Disjunctive Normal Forms |
| DST | Dempster-Shafer Theory |

| | |
|---|---|
| FIR | Fuzzy Information Retrieval |
| FOL | First Order Logic |
| FOPL | First Order Predicate Logic |
| FST | Finite State Transducer |
| GAs | Genetic Algorithms |
| GB | Gödel-Bernays |
| HMM | Hidden Markov Model |
| IBL | Instance-based Learning |
| IE | Information Extraction |
| IR | Information Retrieval |
| KB | Knowledge Base |
| KIF | Knowledge Interchange Format |
| k-NN | k-Nearest Neighbor |
| KR | Knowledge Representation |
| LM | Language Modeling |
| LOOM | Lexical OWL Ontology Matcher |
| MAS | Multi-agent System |
| MDP | Markov Decision Process |
| NLP | Natural Language Processing |
| NLTK | Natural Language Tool-Kit |
| NN | Nearest Neighbor |
| NP | Nondeterministic Polynomial |
| OCR | Optical Character Recognition |
| OWL | Web Ontology Language |
| PCA | Principle Component Analysis |
| PD | Prisoner's Dilemma |
| PSSH | Physical Symbol System Hypothesis |
| QA | Question Answering |
| RBS | Rule-based System |
| RGB | Red Green Blue |
| RL | Reinforcement Learning |
| SRC | Search Results Clustering |
| STRIPS | STanford Research Institute Problem Solver |
| SVM | Support Vector Machine |
| SVN | Support Vector Network |
| TBOX | Terminology Box |
| TREC | Text REtrieval Conference Series |
| TSP | Traveling Salesman Problem |
| WAM | Warren Abstract Machine |
| WSD | Word Sense Disambiguation |
| XCON | Expert Configurator |
| ZF | Zermilo-Fraenkel |

# Chapter 1
# Introducing Artificial Intelligence

**Abstract** The field of Artificial Intelligence (AI) has interfaces with almost every discipline of engineering, and beyond, and all those can be benefited by the use of AI. This chapter presents the introduction to AI, its roots, sub-domains, Turing test to judge if the given program is intelligent, what are the goals of AI for Engineers and Scientists, what are the basic requirements for AI, symbol system, what are the requirements for knowledge representation, and concludes with a chapter summary, and an exhaustive list of exercises. In addition raises many questions to ponder over, like, consciousness, and whether it is possible to create artificial consciousness.

**Keywords** Artificial intelligence · Turing test · Imitation game · Philosophy · Logic · Computation · Cognitive science · Evolution · Artificial consciousness · Speech processing · Expert systems · Physical symbol system hypothesis · Formal system · Knowledge representation

## 1.1 Introduction

The Artificial Intelligence (AI) is a branch of Computer Science, which is mainly concerned with *automation of Intelligent behavior*. This behavior we may consider from all domains—the human, animal world, and vegetation. A compact definition of Intelligence is:

$$Intelligence = Perceive + Analyze + React.$$

The following are often quoted definitions, all expressing this notion of intelligence but with different emphasis in each case:

- "The capacity to learn or to profit by experience."
- "Ability to adapt oneself adequately to relatively new situations in life."
- "A person possesses intelligence insofar as he has learned, or can learn, to adjust himself to his environment."
- "The ability of an organism to solve new problems."

- "A global concept that involves an individual's ability to act purposefully, think rationally, and deal effectively with the environment."
- "Intelligence is a very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience."

The foundation materials of AI comprises—data structures, knowledge representation techniques, algorithms to apply the knowledge and language, and the programming techniques to implement all these.

To get an idea of *Intelligence* it requires answering these and many similar questions:

- Is intelligence due to a *single faculty* or it is a name for a collection of distinct unrelated faculties?
- Is it a priori existence or it can be learned? What does exactly happen when we learn some thing, that is, in terms of information and storage structures.
- What is truly the process of *creativity* and *intuition* in human? These again, in terms of knowledge and its structures.
- Does the intelligence require an internal mechanism, or it can be concluded from the behavior observed?
- What is the mechanism for representing the knowledge in the living cells?
- Are the machines self aware like humans? What are the basic requirements for creating the facility of self-awareness in machines?
- Is it that computer intelligence can be defined only when we know the intelligence in reference to human beings?
- Would it ever be possible to achieve the intelligence in computers? Or, is it true that achievement of intelligence is possible only when or is it that an intelligent entity requires the richness of *sensation* and *experience* which might be found only in a biological existence?

Partly, the aim of Artificial Intelligence (AI) is to find the answer to these questions, through the tools provided by AI. These tools are because, the AI offers the medium, as well as the test-bed for theories of intelligence, which can be expressed in the form of computer programs, and can be tested as well as verified by running these programs on computers.

Unlike Physics and Chemistry, AI is still a premature field, hence, its structure, objectives, and procedures are less clearly defined, and not clear like those in physics and chemistry. The AI has been more concerned about expanding limits of computers, apart from defining itself.

**Learning Outcomes of this Chapter**[1]:

1. Defining AI. [Familiarity]
2. Describe Turing test thought experiment. [Familiarity]

---

[1]https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf (pp. 121–129).

3. Differentiate between the concepts of optimal reasoning/behavior and humanlike reasoning/behavior. [Familiarity]
4. Sub-fields of AI. [Familiarity]
5. Determine the characteristics of a given problem that an intelligent system (i.e., AI-based system) must solve. [Assessment]

## 1.2 The Turing Test

In 1950, in an article "Computing Machinery and Intelligence," *Alan M. Turing* proposed an *empirical test* for machine intelligence, now called *Turing Test* (see Fig. 1.1). It is designed to measure the performance of an intelligent machine against humans, for its intelligent behavior. Turing called it *imitation game*, where machine and human counter-part are put in different rooms, separate from a third person, called *interrogator*. The interrogator is not able to see or speak directly to any of the other two, and does not know which entity is a machine, and communicates to these two solely by textual devices like a dumb terminal [11, 12].

The interrogator is supposed to distinguish the machine from the human solely based on the answers received for the questions asked over the interface device, which is a keyboard (or teletype). Even after having asked the number of questions, if the interrogator is not able to distinguish the machine from the human, then as per the argument of Alan Turing, the machine can be considered intelligent. Interrogator may ask highly computation oriented questions to identify the machine, and other questions related to general awareness, poetry, etc., to identify the human [6].

The game (with the "player machine" omitted) is often in practice under the name of viva-voce to discover whether someone really understands something or has "learned it parrot fashion".

Many researchers argue that the Turing test is not sufficient to establish the presence of intelligence. Some of the arguments *for* and *against* the above test can be as follows:
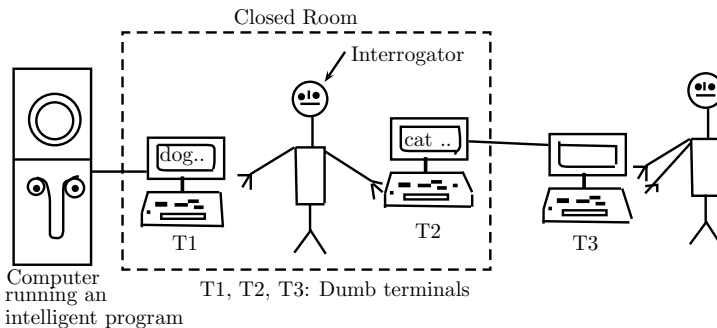


**Fig. 1.1** Turing test (imitation game)

1. It takes human being as a reference for intelligent behavior, rather than debating over the true nature of intelligence: *against*.
2. The unmeasurable things are not considered, e.g., whether a computer uses *internal structures*, or for example, whether the machine is *conscious* of its actions, which are currently not answerable: *against*.
3. Eliminates any bias to human oriented interaction mechanisms, since a computer terminal is used as a communication device: *for*.
4. Biased towards only symbolic problem solving: *against*.
5. Perceptual skills or dexterity cannot be checked: *against*.
6. Unnecessarily constrains the machine intelligence to human intelligence: *against*.

Though, the number of counts of *against* are far more than *for*, there is yet no known test which is considered better than the Turing Test.

In the part success of the Turing Test, a powerful computer has deceived humans in the thinking process, where this machine modeled intelligence of a young boy, to become the first machine to pass the Turing test, conducted in June 2014. In this experiment, five machines were tested at the Royal Society in central London to check if these machines could fool the people into thinking. The machines behaved like humans, and the conversation was in the form of text. A computer program, by the name "Eugene Goostman" was developed to simulate a young boy, which came out to convince the one-third of the judges that it was human [5].

## 1.3   Goals of AI

AI is the area of computer science aiming for the design of intelligent computer systems, i.e., systems that have characteristics of intelligence, like, we observe in human behavior, for example, to understand language(s), and have abilities of learning, reasoning, and problem solving [9].

For many researchers, the goal of AI is to emulate human cognition, while to some researchers, it is the creation of intelligence without considering any human characteristics. To many other researchers, AI is aimed to create useful artifacts for the comforts and needs of human, without any criteria of an abstract notion of intelligence.

The above variation in aims is not necessarily a wrong, as each approach uncovers new ideas and provides a base for pursuing the research in AI. However, there is a convincing argument that due to the absence of a proper definition of AI, it is difficult to establish as what can and what cannot be done through AI.

One of the goals for studying AI is to create intelligence in machines as a general property—not necessarily based on any attribute of humans. When this is the goal, it also includes the objective of creation of artifacts of human comforts and needs, which can be the driving force of technological development. However, this goal also requires a notion of intelligence, to start with.

Further, the problem gets compounded, since artifacts manufacturer may say their product is better in terms of saving labor and money, while cognitive scientists may say that their system correctly predicted human behavior. Apart from this dispute, without proper theoretical base of AI, it is not a wise idea to build complex system—on which one has confidence, can be analyzed for performance and error analysis can be carried out.

The definition of AI be such that for any system, it covers input, output, and their relationship based on the structure of the system. There is a need of such definition to be as general as possible so that it can be uniformly applicable. In the absence of such a definition, one takes the AI as that exists in Chess playing, the one in automated vehicle driving, and the one existing in medical expert system for diagnosis—all these approaches for the definition of AI are varying from case to case.

The *scientific goal* of AI is to determine theories about knowledge representation, learning, rule-based systems, and search that explain various sorts of intelligence.

The *engineering goal* of AI is to acquire ability in the machine so that it can solve the real-life problems. The basic techniques used by AI for this purpose are knowledge representation, machine learning, rule systems, and state space search.

In the past, computer scientists and engineers used to be more concerned with the engineering goals, but the psychologists, philosophers, and cognitive scientists were more keen on the scientific goals. In spite of these opposite concerns, there are common techniques that the two approaches can feed to each other. Hence, we will proceed with both the goals in mind.

## 1.4 Roots of AI

The field of AI does not live in isolation, and has significant roots in number of older disciplines, particularly,

Philosophy,
Logic/Mathematics,
Computing,
Psychology/Cognitive Science,
Biology/Neuroscience, and
Evolution.

In the above domains, there is significant overlap, for example, between philosophy and logic, and between mathematics and computation. By looking at each of these in turn, we get a better understanding about their role in AI, and how these fields have developed to play that role in AI [6, 10].

### 1.4.1  Philosophy

The evidence of philosophy goes as back as Socrates times (∼400 BC) where he asks for an algorithm to distinguish between *piety* (a reverence of supreme being) from *non-piety*. In around 300 BC, *Aristotle* formulated various types of *deductive* reasoning approaches, to mechanically generate conclusions using initial premises. One approach of deductive reasoning he used was *modus ponens*, now also a standard technique of inference in predicate and propositional logic. It is stated as

> If "A is True" → "B is True", and "A holds True" then conclude that "B holds True".

As an example of inference, "If it's raining then you get wet, and it's raining, then you should have got wet".

The philosopher *Rene Descartes* (1596–1650) introduced the concept of mind-body dualism, which says that part of the mind is exempted from following the physical laws. The conclusion of which he has drawn as *free will*. In the present time also, when the AI, machine learning, and data science are dominantly used, but it is argued that machine cannot supersede human in intelligence, as they do not have free will, and they need to be assigned the goal by a human.

*Gottfried Wilhelm Leibnitz*, a German philosopher and mathematician (1646–1716), who supported the materialist nature of mind, said that mind operates by ordinary physical processes. In the present context, this means that mental processes can be performed by the machines.

### 1.4.2  Logic and Mathematics

The logic, has history of development from the time of Greece philosophers—Plato and Aristotle (∼300–400 BC), however, there has been more recent development at a rapid pace. These were due to the following

- *Earl Stanhope's* Logic (1777), using which Earl demonstrated a machine capable of solving the problem using the inference rule, called *syllogisms*, and the numerical problems of logical nature, and elementary questions based on the theory of probability.
- *George Boole* (1815–1864) introduced the language-based formal logic for a drawing logical inference in 1847, later became popular by name *Boolean Algebra*.
- *Gottlob Frege* (1848–1925) introduced the *first-order logic* that today forms the most common knowledge representation system, called as FOPL (first-order predicate logic).
- *Kurt* Gödel (1906–1978), in 1931, demonstrated that there are limits of logic. Through his *incompleteness theorem* he showed that in any formal logic, which is powerful enough to describe the properties of natural numbers, there exist true statements, but their truth cannot be proved using any algorithm.

- *Roger Penrose* in 1995 tried to prove that human mind has non-computable capabilities.

### 1.4.3  Computation

In the nineteenth and twentieth-century, many scientists defined the formalism of what is *computation*, basic theory of it, and that there are things that are not computable irrespective of whatever are the computing resources and time provided.

In the year 1869, *William Jevon* constructed a *Logic Machine* capable of handling Boolean Algebra and Venn Diagrams, and could solve logical problems faster than human beings.

*Alan M. Turing* (1912–1954) tried to characterize exactly what are the functions that can be computed. He used, what is now called as Turing Machine. Unfortunately, it is difficult to give the notion of computation as a formal definition, however, the *Church-Turing thesis*, due to Alonzo Church and Turing, states that a Turing machine is capable of computing any computable function, which is now, accepted as a sufficient definition of computability. Turing also showed that there are some functions which no Turing machine can compute (e.g., *Halting Problem*)—these are non-computable functions.

*John von Neumann* (1903–1957) gave, now what is called as, von Neumann architecture—a description of a logical model of computation and computer, without any physical realization of a computer.

In 1960s, two important concepts emerged—*Intractability* (the solution time of a problem grows at least exponentially) and *Reduction* of complex problems into simpler problems.

### 1.4.4  Psychology and Cognitive Science

Cognitive Psychology or Cognitive Science is the study about the functioning of the mind, human behavior, and the processing of information about the human brain. An important consequence of human intelligence is human languages. The early work on knowledge representation in AI was about human language, and was produced through research in linguistics.

It is humans' quest to understand as to how our and other animals' brains lead to intelligent behavior, with the aim to ultimately build AI systems. On the other hand, it is also aimed to explore the properties of artificial systems, like, computer models/simulations to test our hypotheses concerning human systems.

Many people working in sub-fields of AI are in the process of building models of how the human system operates, and use artificial systems for solving real-world problems.

### 1.4.5   Biology and Neuroscience

The field of neuroscience says that human brains, which provide intelligence, are made up of tens of billions of neurons, and each neuron is connected to hundreds or thousands of other neurons. A neuron is an elementary processing unit, performing a function called *firing*, depending on the total amount of activity feeding into it. When a large number of neurons are connected together, it gives rise to a very powerful computational device that can compute, as well as learn how to compute.

The concept of human brains, having the capability to compute, as well as learn, is used to build artificial neurons in the form of electronics circuits, and connect them as circuits (called ANN—artificial neural networks) in large quantities, to build powerful AI systems. In addition, the ANN are used to model various human abilities.

The major difference between the functions of neurons and the process of human reasoning is that neurons work at *sub-symbolic level*, whereas much of conscious human reasoning appears to operate at a symbolic level, for example, we do most of the reasoning in the form of thoughts, which are manipulations of sentences.

The collection of neurons in the form of programs called Artificial Neural Networks (ANN), perform well in executing simple tasks, and provide good models of many human abilities. However, there are many tasks of AI that they are not so good at ANN, and other approaches are more promising in those areas, compared to ANN. For example, for natural language processing (NLP) and reasoning, the symbolic logic, called predicate logic is better suited.

### 1.4.6   Evolution

Unlike the machines, the humans (intelligence) has a very long history of evolution, of millions of years, compared to less than hundred years for electronic machines and computers. The first exhaustive document of human evolution, the evolution by *natural selection* is due to Charles Darwin (1809–1882). The idea is that fitter individuals will naturally tend to live longer and produce more children (may not be truly valid in the modern world). Hence, after many generations, a population will automatically emerge with good innate properties [3].

Due to this evolution, the structure of the human brain, and even the knowledge, are to a sufficient extent built-in at the time of the birth. This is an advantage over ANNs, which have no pre-stored knowledge, hence they need to acquire the entire knowledge by learning only. However, the present-day computers are powerful enough that even the evolution can be simulated using them, and can evolve the AI systems. It has now become possible to evolve the neural networks to some extent so that they are efficient at learning. But, may still be challenging to recreate the long history of the evolution of humans in the ANNs.

A closely related field to ANNs is *genetic programming*, which is concerned with writing the programs that evolve over time, and do not need to modify them as it is done in usual programs when the system requirement changes [4].

## 1.5  Artificial Consciousness

Right from the time, automated machines like computers came into existence, it was the quest of researchers to build machines that can compete in intelligence with humans. Looking at the current progress rate of smart machines, like smartphones, it is believed that in the not very far future, it may be possible to build machines which may be, if not more, but have comparable intelligence to that of humans. Using such machines, it may be possible to produce human like consciousness in machines—called as artificial consciousness.

On the contrary, even a far of realization of artificial consciousness gives rise to several philosophical nature of questions:

- Can the computers be made to think or they will just calculate?
- Is consciousness a human prerogative only, or it can be created in machines also?
- Is the consciousness due to the material comprised in the human brain, or it can be created in silicon also (the computer hardware)?

To provide the answers to these questions is difficult as of now, mainly because it requires combining the knowledge from the fields of computer science, neurophysiology, and philosophy.

On the other hand, the very talk of artificial consciousness—a possible product of the human imagination, express human desires, and fears about future technologies—may influence the course of progress.

At a social level, the science fiction stories simulate future scenarios that can help prepare us for crucial transitions by predicting the consequences of such technological advances [1].

## 1.6  Techniques Used in AI

The AI systems have a lot of variations, for example, the rule-based systems are based on symbolic representations, and work on inferences. There are other extremes, the ANN-based system work on the interface with other neurons, and connection weights. In spite of all these, there are four common features among all of them.

*Representation*

All AI systems have an important feature of knowledge representation. The rule-based systems, frame-based systems, and semantic networks make use of a sequence of *if-then rules*, while the artificial neural networks make use of connections along with connection weights.

*Learning*

All AI systems have capability of learning, using which they automatically build up the knowledge from the environment, e.g., acquiring the rules for a rule-based expert system, or determining the appropriate connection weights in an artificial neural network.

*Rules*

The rules of an AI-based system can be implicit or explicit. When explicit, the rules are created by a knowledge engineer, say, for an expert system, and when implicit, they can be, for example, in the form of connection weights in a neural network.

*Search*

The search can be in many forms, for example, searching the sequence of states that lead to solution faster, or searching for an optimum set of connection weights in an ANN by minimizing the fitness function.

## 1.7   Sub-fields of AI

Considering AI as replication of human intelligence may be misleading, and primitive. The later is because the true process of human intelligence and its sources are still under debate. However, if AI is taken to mean the advanced computing, it means more justified. In the past two decades, particularly after the year 2k, the AI applications have evolved and expanded, in the commercial, industrial, medicines and drug decide, medical science, consumer products, manufacturing processes, and even in management, to list only a few of its total domain. The use of AI techniques in every organization has become necessary to maintain competitiveness in the market. Many organizations keep secret of the true AI techniques they use.

AI now consists of many sub-fields, using a variety of techniques, such as the following:

*Speech Processing*: To understand speech, speech generation, machine dialog, machine user-interface.
*Natural Language Processing*: Information retrieval, Machine translation, Question/Answering, summarization.
*Planning*: Scheduling, game playing.
*Engineering and Expert Systems*: Troubleshooting medical diagnosis, Decision support systems, teaching systems.
*Fuzzy Systems*: For fuzzy controls.
*Models of Brain and Evolutionary*: Genetic algorithms, genetic programming, Brain modeling, time series prediction, classification.
*Machine Vision and Robotics*: Object recognition, image understanding, Intelligent control, autonomous exploration.
*Machine Learning*: Decision tree learning, version space learning.

Most of these have both engineering and scientific aspects. Many of these are going to be discussed in this text. Following is brief Introduction to some of these areas.
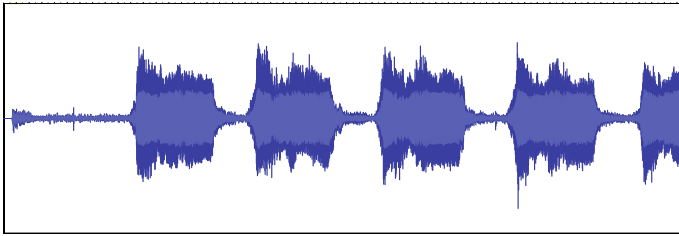
**Fig. 1.2**  Sound waves for the text "Hello" repeated five times by a five year child

### 1.7.1  Speech Processing

The speech processing and understanding of human speech has number of applications, some of them we come across quite often, like, speech recognition for dictation systems, speech production for automated announcements, voice-activated control, human–computer interface (HCI), and voice-activated transactions are a few examples.

One of the primary goals is, how do we get from sound waves to text streams and vice-versa? The Fig. 1.2 is an example, showing the sound wave pattern for the text "Hello" repeated five times.

To be precise, how should we go about segmenting the stream into words? How can we distinguish between "Recognize speech" and "Wreck a nice beach"?

### 1.7.2  Natural Language Processing

Consider the machine understanding and translation of simple sentences given below.

*Ram saw the boy in the park with a telescope.*
*Ram saw the boy in the park with a dog.*

In the parse-tree in Fig. 1.3a, the sentence structure is "Ram saw, the boy in the park, with a Telescope." Whereas in Fig. 1.3b, it is "Ram saw, the boy in the park with a dog." In first, it shows the association of verb *saw* and *telescope*, i.e., someone is seeing using telescope. In Fig. 1.3b the association of *boy*, and *dog* is shown, and all are in the park. The further deeper contexts help in resolving this ambiguity.

Though the sentences appear simple, finding out the meaning of each using the machine is difficult, as the parse-tree of each need to be analyzed for the meaning associated in each, in addition, the context knowledge is important.

**Fig. 1.3** Parse-trees with different semantics



(a)                              (b)
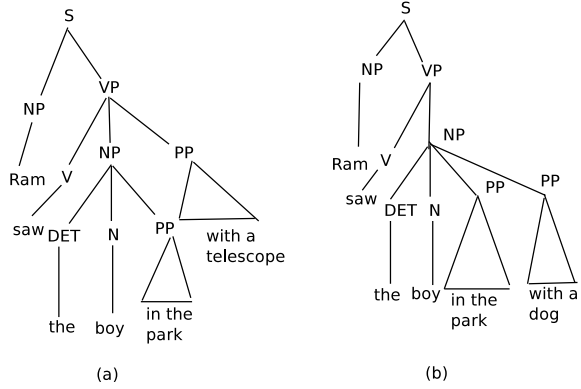
Some of the common applications of Natural Language Processing (NLP) are [2]:

1. Word Processing and Desktop Publishing
2. Spell check and Correction
3. Information Retrieval
4. Information Extraction
5. Information Categorization
6. Question-answering
7. Information summarization
8. Machine Translation.

### 1.7.3  Planning

Any planning, or specifically the robotic planning, is concerned with choosing (computing) the correct sequence of actions to complete a specific task. This requires a convenient and efficient representation of the problem domain. The plan steps are called states defined in formal language, like, in predicate language, or in the form of rules, depending on what type of planning has been used. A plan may be taken as a sequence of operations that transform the initial state into the goal state ultimately, the later is the solution. The best planning seeks to explore the best path of states for reaching the goal state. Hence, the best path or an optimum path seeks exploration or searching, to find out the best possible path efficiently, in terms of time and space required for running the planning algorithm.

### 1.7.4  Engineering and Expert Systems

These are primarily based on symbolic processing—which is the main stream of AI. The fundamental problems such as, how to represent knowledge, is the con-

tent of the following chapters. Various representation schemes are classified into two categories: *Symbolic* representation and *Graphical* representation. The first is *propositional* and *predicate logic* based, while the other is graph-based, like—*semantic networks, frames, ontologies* and *conceptual dependencies*.

### 1.7.5 Fuzzy Systems

The primary aim of fuzzy or *soft-computing* is to exploit the tolerance for *imprecision* and *uncertainty* to achieve tractability, robustness, and low cost applications. Fuzzy systems can be integrated with other techniques such as neural networks, and *probabilistic reasoning*. In fuzzy systems, the membership are partial (fuzzy/unclear). Examples of fuzzy sets are: old, cloudy, high speed, rainy, etc. This is in contrast to the classical-based systems, where boundaries are *crisp*, like, member of computer science class, an Indian national, chair (member of the set of chair), where all three are representing the full membership of each category of the given set.

### 1.7.6 Models of Brain and Evolution

The models of the human brain and evolution correspond to two major approaches to AI. The AI as a model of the brain corresponds to *Symbolic AI*, and has remained the major field of AI. This has the property of high level of mathematical abstractions, and considered as the macroscopic view of AI. The human psychology operates at a symbolic level, as well as the AI programming languages, and early engineered systems fall in this type of AI.

The other approach of AI is like human evolution, which is based on low level biological and genetic models of living beings. The neural-based computing (artificial neural networks), and genetic algorithms derived from these concepts from life. These biological models of AI may not necessarily be resembling verbatim to their living being counterparts, but the AI techniques that are based on GA (genetic algorithms) evolve the solution like the populations of human or like other forms of the life evolves [7].

Neural networks, similar to expert systems, are modeled on the human brain and learn by themselves from patterns. Thus, learning can then be applied to classification, prediction, or control applications. The GAs, are computer models based on genetic and evolution. Their basic idea is that genetic programs work towards finding better and better solutions to problems, just as species evolve to better adapt to their environments. The GAs comprises three basic processes: *reproduction* of solution based on their fitness, *crossover* of genes, and *mutation* for random change of genes. A broader definition GA is *evolutionary computing*, which includes not only GAs but classifier systems, genetic programming in which each solution is a computer program, and a part of *artificial life*.

## 1.8  Perception, Understanding, and Action

These fields are concerned with vision, speech processing, and robotics. The basic theme is applications that make machine sense (e.g., to see, hear, or touch), then understand and think, and finally take action.

For example, the basic objective of machine vision may be to make the machine "understand" the input consisting of reflected brightness values. Once this understanding is achieved, the results can be used for interpretation of patterns, inspecting parts, action of robots, and so forth. Developing an understanding presents the same difficulty in all areas of AI, including knowledge based systems—people understand what they see by integrating an optical image with complex background knowledge. Such background knowledge has been built over years of experiencing perceptions. Creating this type of information processing in the machine is a challenging task; however, some interesting applications have already appeared as the evidence to support future progress.

The speech processing uses two major technologies. One of these areas focus on input or speech recognition where acoustic input, like optical input in machine vision, is difficult task to automate. People understand what they hear with complex background knowledge. Speech recognition technology includes signal detection, pattern recognition, and possibly semantics—a feature closely related to Natural Language understanding. The other technology concerns to the creation of output or text-to-speech (tts) synthesis. Speech synthesis is easier than recognition, and its commercialization has been well established.

The field of Robotics integrate many techniques of sensing, and, is one of the AI areas in which industrial applications have the longest and widest successful records. The abilities of these robots are relatively limited. For example, only in a limited task such as welding seams and installing windshields, etc [7].

## 1.9  Physical Symbol System Hypothesis

The symbols are the basic requirements of intelligent activity, e.g., by human the symbols are basic number systems, alphabet of our languages, sign language, etc. Similar is the case with entire computer science, the languages, commands, computations, all have symbols as the base. When the information is processed by computers, on the completion of the task, we measure the progress, as well as the quality of results and efficiency of computations, only based on its symbols' contents in the end results [8].

### *1.9.1 Formal System*

Basic requirement of achieving AI is *formal system*, which is based on *physical symbol system hypothesis*. The term was coined by *Allen Newell and Herbert Simon*, which states that a "physical symbol system" is a necessary requirement for AI to function. As per this, the physical patterns, called symbols, are combined to produce structures (i.e., expressions), and the processes act on these to manipulate to produce new expressions [8].

This symbol system hypothesis claims that human intelligence is due to this symbol system, which comprises all the alphabets, numerals, and other punctuation symbols. Thus, the symbol system is a "necessary" requirement for achieving the intelligence. Based on this argument, one can say that, if the machines are provided with symbol system with symbol manipulation capabilities, it is "sufficient" for achieving the intelligence in the machines.

As per the Physical Symbol System Hypothesis (PSSH), the capabilities of symbol manipulation are the essence for human's, as well as machines' intelligence. Hence, it is a necessary and sufficient tool for achieving intelligence in both the machines and humans. There is also experimental evidence, that, in various problem solving, like, in mathematical puzzles, planning of activities, and execution, the symbol system is the key requirement. By the term "necessary" here means, that the system possessing general intelligence, on analysis, will prove to be based on a physical symbol system. And, the term "sufficient" means that the physical symbol system can be organized to be exhibiting the general intelligence. When the problem-solving process of humans were simulated step by step on computers by the researchers, it was found to be simply the process of symbols' manipulations.

Of course, various researchers have criticized this hypothesis strongly, but still, it forms the central part of AI research. The critics argue that the symbol systems work only for high level processes like chess, games, and puzzles, but not suitable for low level systems like vision and speech recognition. This distinction is based on the fact that high-level symbols directly correspond to objects, like ⟨*cat*⟩, ⟨*house*⟩, ⟨*hill*⟩, etc, but not to the low-level symbols that are present in the machinelike neural networks (or ANN).

### *1.9.2 Symbols and Physical Symbol Systems*

If we look at the entire knowledge of computer science, it is the symbols, which have been used to explain this knowledge at the most fundamental level. The explanation is nothing but the scientific proposition of nature, which is empirically derived over a long period of time, through a graduate development. Hence, the symbols are at the root of artificial intelligence, and are also the primary topic of artificial intelligence.

For all the information processed by computers in the service of finding the end goals, the intelligence of the system (computers) is their ability to reach goals in

the face of difficulties and complexity of the solution, as well as the complexity introduced by the environment. The fundamental requirements to achieve artificial intelligence is to store and manipulate the symbols, however, there is no uniformity and specific requirement of storage structures, as the structures vary from method to method used for implementation of AI, which are mostly the variants of network-based representation and predicate-based representations.

The "physical systems" used have two important characteristics: 1. Operation of systems is governed by the laws of physics, when they are realized as engineered systems, and made of engineered components; and 2. The "symbols" are not limited only to the symbols used by human beings.

### 1.9.3   Formal Logic

The "physical symbol system" hypothesis has its root to Russel's *formalizing logic*, which states that one need to capture the basic conceptual notion of mathematics in logic and put that notion to proof and deduction as sound base. This notion, with the effort ultimately grew in the form of mathematical logic—the propositional logic, predicate logic, and their variants [13].

### 1.9.4   The Stored Program Concept

The second generation of computers brought the concept of stored program concept in the mid-forties, after the *Eniac* computer. The arrival of these computers was considered as a milestone in terms of conceptual progress, as well as practical availability of systems. In such systems, the programs are treated as data, which, in turn, was processed by other programs, like a compiler program processing another program as data to generate object code. Interestingly, this capability was already verified and existed in Turing machine, which came as early as 1936. The Turing machine is a model of computing given by Alan M. Turing, where, in a universal Turing machine, an algorithm (another Turing machine) and data are on the very same tape. This idea was realized practically when machines were built with enough memory to make it practicable to store actual programs in some internal place, along with the data on which the program will act, as well as the data which will be produced as a result of the execution of the programs.

## 1.10   Considerations for Knowledge Representation

As far as AI is concerned, the following are the aspects of knowledge representation:

- What is the meaning of Knowledge?
- How the Knowledge can be represented in the machine?
- What are the requirements of representation of knowledge, e.g., structures, methods, size, etc.
- How the practical and theoretical aspects differ for knowledge representation?
- Can it be? Or, if yes, how to represent the knowledge using Natural Language?
- Can we call the databases as a form of knowledge representation?
- What are the semantic networks, and what are the frames? How the knowledge can be represented using these approaches?
- How the knowledge can be represented using the First-Order Predicate Logic (FOPL)?
- What is a Rule-Based Systems?
- What is an expert system?
- Out of the many techniques, which is the best technique for knowledge representation?

### 1.10.1  Defining the Knowledge

As per the Webster English language dictionary, the following are the meanings of knowledge:

1. The act or state of knowing; clear perception of fact, truth, or duty; certain apprehension; familiar cognizance; cognition. [1913 Webster]
   Knowledge, which is the highest degree of the speculative faculties, consists in the perception of the truth of affirmative or negative propositions—Locke. [1913 Webster]
2. That which is or may be known; the object of an act of knowing; a cognition—Chiefly used in the plural. [1913 Webster]
3. That which is gained and preserved by knowing; instruction; acquaintance; enlightenment; learning; scholarship; erudition. [1913 Webster]

### 1.10.2  Objective of Knowledge Representation

The objective of knowledge representation is to express the knowledge in computer so that the AI programs can use it to perform reasoning and inferences using this in an efficient way. The knowledge is represented using certain representation language,

for example, a predicate like language. The language has two important components in it.

*Syntax*

The system of a language defines the methods using which we or the machine can distinguish the correct structures from incorrect, i.e., it makes possible to identify the structurally valid sentences.

*Semantics*

The semantics of a language defines the world, or facts in the world of the concerned domain. And, hence defines the meaning of the sentence in reference, to the world.

## 1.10.3   Requirements of a Knowledge Representation

A good knowledge representation system for any particular domain should possess the following properties.

*Adequacy of representation*

The representation system should be able to represent all kind of knowledge needed in the concerned AI-based system.

*Adequacy of Inference*

The representation should be such that all that can be inferable by manipulating the given knowledge structures should be inferred by the system, when needed.

*Inference Efficiency*

The knowledge structures in the representation are so organized that the attention of the system, in the form of deductions, navigates in such a direction that it can reach the goal quickly.

*Efficient acquisition*

It should be able to acquire the new information automatically and efficiently, as and when needed, and also to update the knowledge regularly. In addition, there should be a provision that knowledge engineer can update the information in the system.

## 1.10.4   Practical Aspects of Representations

We are aware of good and bad knowledge representation, when we consider the knowledge representation in English or any other natural language. These are due to factors, like, syntax, semantics, partial versus full knowledge on any subject, depth and breadth of knowledge, etc.

There are many theoretical requirements for good knowledge representations, which can be met by dealing with a number of practical aspects, as follows:

- The representations should be *complete*, so that everything needs to be represented, can easily be represented.
- The representations should be simple and clear, so that one can easily understand what is being communicated by the representation,
- The important objects and their relations should be explicit and accessible, so that it becomes easy to see what is going on, and how the components of knowledge interact with each other.
- The irrelevant detail of the knowledge should be suppressed in the representation, so that they do not introduce complications. However, when needed, these are still available.
- The representation should be concise, so that information can be stored, retrieved and manipulated rapidly.
- The representation should be such that the overall system is fast.
- They must be *computable* and implementable with standard computing procedures.

To realize the above, a lot depends on algorithms used, representation structures, hardware, as well as dissemination of knowledge before it is represented.

### *1.10.5  Components of a Representation*

To carry out the analysis of any representation system, it is useful to break the entire representation into smaller (smallest) components, which are in the most fundamental form. Accordingly, the components of an AI representation are divided into the four fundamental components:

*Lexical components*

The lexical components of knowledge representation are the symbols and words of the vocabulary used for representation.

*Syntactic/Structural components*

It describes how the symbols can be arranged systematically to create meaningful sentences. These structures are the grammar of the language used for representation.

*Semantic components*

They help is associating real-world meaning to objects and entities.

*Procedural components*

These procedures are used for creating and modifying the representations, and also for answering the questions using these procedures.

## 1.11   Knowledge Representation Using Natural Language

We humans are intelligent beings, who make use of the knowledge represented in the form of natural language (like, English, Hindi, Chinese, etc.), we update that knowledge (i.e., acquisition), and do the reasoning and inferences using this representation. Of course, there are many other types of knowledge representation and inferencing with the humans, which are not symbolic-based, like those acquired through smell, touch, hearing, and taste (through tongue). Hence, why not use the natural language for knowledge representation for machines also? The following are the trade-offs for representation using natural language.

Advantages

There are very strong advantages in favor of using natural language for knowledge representation.

- The natural language is strong at expressiveness, using which we can represent almost everything (real-world situations, pictures, symbols, ideas, emotions) and can carry out the reasoning using that.
- It is the most abundantly used source for knowledge representation by humans, for example, can we list the name of textbooks not written in natural language? It is hard to reply!

Disadvantages

In spite of strong points in favor of natural language-based representation, there are serious difficulties in realizing such representation for machines, due to the following reasons:

- The syntax and semantics of the natural language are very complex, which are not so easily understood. Hence, it becomes challenging and risky if solely depended on machines.
- The uniformity in representation is lacking—the sentences carrying the identical meaning can be represented in many different syntax (structures).
- There is a lot of ambiguity in the natural language, a sentence/word may have many different meanings, and the meanings are context-dependent. Hence, it is overly risky to try these for machines, unless the machines are having intelligence at par with the human.

## 1.12   Summary

Intelligence is defined as:

$$Intelligence = Perceive + Analyze + React$$

AI has its inter-related goals for scientific, as well as engineering areas. Its roots are in several historical disciplines, which include, philosophy, logic, computation, psychology, cognitive science, neuroscience, biology, and evolution.

The major sub-fields of AI now include: neural networks, machine learning, evolutionary computation, speech recognition, text-to-speech translation, fuzzy logic, genetic algorithms, vision systems and robotics, expert systems, natural language processing, and planning. Many of these domains have dependency and are inter-related, for example, neural network is one of the techniques for machine learning. The common techniques used across these sub-fields are: knowledge representation, search, and information manipulations.

Human brain and evolution are also the areas of AI modeling.

The study of logic and computers have demonstrated that intelligence lies in the *physical symbol system* (PSS)—a collection of patterns and processes. The PSS needs the capability to manipulate the patterns, i.e., it should be able to create the patterns, modify the patterns, and should be able to destroy the patterns. The patterns have important properties, that they can designate objects, processes, and other patterns. When the patterns designate processes, the later can be interpreted, i.e., to perform the process steps. The two significant classes of symbol systems we are familiar with those that are used by human beings, and those by computers. The later uses binary strings or patterns.

The PSSH (physical symbol system hypothesis) says that to achieve the intelligence, it is sufficient to have three things,

  i.  a representation system, using which anything can be represented,
  ii.  a manipulation system, using which the symbols can be manipulated, and
iii.  search using which the solution can be searched.

For the above, it is in fact, not important as whether the medium of storage is the human brain (neurons) or the electronic memory of computer systems.

Various approaches for knowledge representations (KR) are:

  i.  Natural languages versus databases.
  ii.  Frame versus semantic network-based representation.
iii.  Propositional and predicate logic-based representation.
 iv.  Rule-based representation.

Knowledge representation helps to know the object or the concerned concept. Various characteristics of KR are:

  i.  KR has syntax and semantics.
  ii.  Requirements for knowledge representation are: adequacy of representation and inferencing, and efficiency of inference and acquisition.
iii.  Its practical aspects are: complete, computable, and suppression of irrelevant data.
 iv.  Components for KR are: lexical, structural, semantic, and procedural.

## Exercises

1. Try to analyze your own learning behavior, and list the goals of learning, in the order of their difficulty level of learning.
2. List the living beings—human, dog, cow, camel, elephant, cat, birds, insects, in the order of their intelligence levels. Also, justify your argument.
3. Suggest some method to combine large number of human beings, in a group, and formulate a method/algorithm to perform the pipeline or any fast computing work.
4. Write an essay, describing the various anticipated theories as to how the human processes the information?
5. Consider a task requiring knowledge like baking a cake. Out of your imagination, suggest what are the knowledge requirements to complete this task.
6. Out of your reasoning, explain the distinction between knowledge and belief.
7. What is the basic difference between neural network level processing and processing carried out for human reasoning?
8. What are the major advantages of humans over modern computers?
9. List the examples where PSSH is not sufficient or not the basis of achieving Intelligence. Justify your claims.
10. Write an essay, describing how the things (objects/activities) are memorized by

    a. Plants.
    b. Birds.
    c. Sea animals.
    d. Land animals.
    e. Human.

    What can be the size of memories in each of the above cases?
11. Discuss the potential uses of AI in the following applications:

    a. Word processing systems.
    b. Smartphones.
    c. Web-based auction sites.
    d. Scanner machines.
    e. Facebook.
    f. Twitter.
    g. Linkedin.
    h. Amazon and Flipkart.

12. How the artificial neural networks (ANN) and genetic algorithms differ from each other in respect of leaning to be used for problem solution? (Note: You need to explore the AI resources to answer this question).

13. It is an accepted scientific base that physical characteristics of life are genetically transferred. Do you believe that information and knowledge are also genetically transferred? Justify for yes/no?

14. Are the beliefs of rebirth and reincarnation are also the goals of AI research? How and how not?

## References

1. Buttazzo G (2001) Artificial consciousness: utopia or real possibility? IEEE Comput 34(7):24–30
2. Church KW, Lisa FR (1995) Commercial applications of natural language processing. Commun ACM 38(11)
3. Forrest S (1993) Genetic algorithms: principles of natural selection applied to computation. Science 261(13):872–878
4. Goldberg DE (1994) Genetic and evolutionary algorithms coming of age. Commun ACM 37(3):113–119
5. http://www.theguardian.com/technology/2014/jun/08/super-computer-simulates-13-year-old-boy-passes-turing-test. Accessed 19 Dec 2017
6. Ludger GF (2009) Artificial intelligence - structures and strategies for complex problem solving, 5th edn. Pearson, New Delhi
7. Munakata T (1994) Commercial and industrial applications of AI. Commun ACM 37(03)
8. Newell A, Herbert AS (1976) Computer science as an empirical inquiry: symbols and search. Commun ACM 19(3):113–126
9. Russell SJ (1997) Rationality and intelligence. Artif Intell 94:57–77
10. Russell SJ, Norvig P (2005) Artificial intelligence, a modern approach, 2nd edn. Pearson, New Delhi
11. Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/entries/turing-test/. Accessed 19 Dec 2017
12. Turing AM (1950) Computing machinery and intelligence. MIND - Q Rev Psychol Philos Lix(236):433
13. Whitehead AN, Russell B (1910) Principia mathematica, vol 1 (Part I. Mathematical logic). Merchant Books. ISBN 978-1-60386-182-3

# Chapter 2
# Logic and Reasoning Patterns

**Abstract** Logic is the foundation of AI, and the majority of AI's principles are based on logical or deductive reasoning. The chapter presents: contributions of pioneers of logic, the argumentation theory, which is based on logic and with its roots in propositional logic, the process of validating the propositional formulas, their syntax and semantics, interpretation of a logical expression through semantic tableau, followed with presents the basic reasoning patterns used by human, and their formal notations. In addition, presents the normal forms of propositional formulas and application of resolution principle on these for inference. The nonmonotonic reasoning and its significance is briefly described. At the end, the chapter presents the axiomatic system due to Hilbert and its limitations, and concludes with chapter summary.

## 2.1 Introduction

The ancient *Greeks* are the source of modern logic, their education system emphasized the competence in rhetoric (proficient in language) and philosophy; the words *axioms* and *theorem* are from Greek. The logic was used to formalize the deductions—the derivation of true *conclusions*—from true *premises*. Later it was formalized as a set theory by the mathematician *George Boole*. Till the arrival of the nineteenth century, the logic remained more of a philosophical nature, rather than a mathematical and scientific tool. Later, since complex things could not be reasoned through logic, the logic became part of mathematics, where mathematical deduction became justifiable through formalizing a system of logic, and resulted in one very important breakthrough. This was, about the set of true statements, stated as "the set of provable statements are only those that are true statements." This is because some proof exists for those due to some other true statements.

At the beginning of nineteenth century, the mathematician *David Hilbert* introduced the logic, as well as theories of the nature of logic-a far more generalization

of the logic. But, this generalization received a blow when another mathematician *Kurt Gödel* showed in 1931 that there are true statements of arithmetics that are not provable, through his *incompleteness* theorem.

Now, though mathematical logic remains the branch of pure mathematics, it is extensively applied to computer science and artificial intelligence in the form of propositional logic and predicate logic (first-order predicate logic (FOPL)).

As per the Newell's and Simons's Physical Symbol System Hypothesis (PSSH), discussed in the previous chapter, the knowledge representation is the first requirement of achieving intelligence. This chapter presents the knowledge representation using *propositional logic*, introduces first-order predicate logic (FOPL), and drawing of inferences using propositional logic.

Logic is a formal method for reasoning, using its concepts can be translated into symbolic representation, which closely approximate the meaning of these concepts. The symbolic structures can be manipulated using computer programs to deduce facts to carry out the form of automated reasoning [9].

The *aim* of logic is to learn principles of valid reasoning as well as to discern good reasoning from bad reasoning, identifying invalid arguments, distinguishing *inductive* versus *deductive* arguments, identifying *fallacies* as well as avoiding the fallacies.

The *Objective* of logic is to equip oneself with various tools and techniques, i.e., *decision procedures* for validating given arguments, detecting and avoiding fallacies of a given deductive or inductive argument.

We study the logic because of the following reasons:

- Logic deals with what follows from what? For example, Logical consequence, inference pattern, and validating such patterns,
- We want the computer to understand our language and does some intelligent tasks for us (Knowledge representation),
- To engage in debates, solving puzzles, game like situation,
- Identify which one is a fallacious argument and what is a type of fallacy?
- Proving theorems through *deduction*. To find out whether whatever proved is correct, or whatever obviously true has a proof, and
- To solve some problems concerning the foundations of mathematics.

**Learning Outcomes of This Chapter**:

1. Convert logical statements from informal language to propositional logic expressions. [Usage]
2. Apply formal methods of symbolic propositional such as calculating the validity of formula and computing normal forms. [Usage]
3. Use the rules of inference to construct proofs in propositional. [Usage]
4. Describe how symbolic logic can be used to model real-life situations or applications, including those arising in computing contexts such as software analysis (e.g., program correctness), database queries, and algorithms. [Usage]

5. Apply formal logic proofs and/or informal, but rigorous, logical reasoning to real problems, such as predicting the behavior of software or solving problems such as puzzles. [Usage]
6. Explain the difference between rule-based and model-based reasoning techniques. [Familiarity]
7. Describe the strengths and limitations of propositional logic. [Familiarity]

## 2.2 Argumentation Theory

The *Argumentation theory* is the study of how conclusions can be reached through logical reasoning, that is, whether the claims are soundly based on premises or not (Fig. 2.1). It includes the arts and sciences of civil debate, dialog, conversation, and persuasion. It includes the studies of rules of inference, logic, and procedural rules in both artificial and real world settings.

An argumentation system comprises debate and negotiations, aimed at reaching to a mutually agreeable conclusion. The argumentation may also consist of erroneous dialogs, where victory over an opponent is the only goal, without consideration of the truth. The argumentation theory is an art, as well as science, using these people protect their self-interests and beliefs using rational dialogs at commonplaces of their meeting points, and during the process of argumentation.

People make use of argumentation theory in law also, for example, in preparing an argument to be presented before the court of law, in debate in the court of law, in trials, and in testing the validity of certain kinds of evidence. Scholars of Argumentation theory study the post-hoc rationalizations by which organizational actors try to justify decisions even made irrationally.

The simple block diagram for logical reasoning shown in Fig. 2.1 has internal structure, comprising the following:

1. a set of assumptions or premises (or antecedents),
2. a method of reasoning or deduction, and
3. a conclusion or consequence.

If the premises are $P_1$, $P_2$, ..., $P_n$, then they are conjuncted and their conjunction imply the conclusion $C$, i.e., $P_1 \wedge P_2 \wedge \cdots \wedge P_n \rightarrow C$.

An argument must have at least one premise and one conclusion. Often, classical logic is used as the method of reasoning so that the conclusion follows logically from the assumptions or support. One challenge is that if the set of assumptions is

**Fig. 2.1** Inference process



Set of premises → Method of reasoning → Conclusion
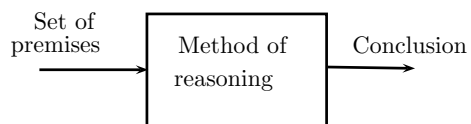
inconsistent then anything can follow logically from inconsistency. Therefore, it is common to insist that the set of assumptions be consistent. It is the practice to have a minimal set. Such argumentation has been applied to the field of medicine also.

The second school of argumentation investigates abstract arguments, where "argument" is considered a primitive term, so no internal structure of arguments is taken on the account.

## 2.3   Role of Knowledge

We have discussed in the above section about knowledge and logic. The Logic needs a base of knowledge to infer or conclude new knowledge. The knowledge is also used for *learning*, *retrieval* and *reasoning*. The learning is not only adding new facts into an existing knowledge base, but before the new data are put into the storage, they need to be classified for ease of retrieval. The interaction and inference with existing facts avoid the redundancy and duplication of knowledge in the knowledge base. In addition, the learning updates the existing facts.

Having stored the knowledge in the process of learning, one important objective of that is, retrieval. The representation scheme used in the knowledge base has critical effect on the efficiency of the retrieval system. As humans, we are very good in retrieval from our knowledge (memories), and many AI systems have used that for modeling AI learnings.

The knowledge is also used for reasoning process, i.e., to infer new facts from the existing facts in the knowledge. For example, observing many birds flying, to infer that all the birds fly, as well for solving a complex problem, say, based on sufficient facts, to infer that a customer financed by a bank, will be able to repay the loan the bank has financed to him.

## 2.4   Propositional Logic

The propositional logic deals with individual Propositions, which are viewed as *atoms*, i.e., these cannot be further broken down into smaller constituents. For building propositional logic, first we describe the logic with the help of a formula called *Well-Formed Formulas* (wff, read as woofs). A formula is a *syntactic* concept, which means whether or not a string of symbols is a formula not. It can be determined solely based on its formal construction, i.e., whether it can be built according to its construction rules. Therefore, we are in a position to verify that a sequence of symbols is a formula or not, as per the specified rules. This function of verification, in a compiler, is done by a *parser*—to verify whether the formula belongs to the particular programming language or not. A parser also constructs a parse-tree of the given formula through which it tells how the formula is constructed [2].

The meaning (*semantics*) is associated with each formula by defining its *interpretation*, which assign a value *true* (*T*) or *false* (*F*) to every formula. The syntax is also used to define the concept of *proof*—the symbolic manipulations of formulas to deduce the given theorem. The important thing we should note is that provable formulas are only those which are always true.

We start the propositional logic with the individual propositional variables. These variables themselves are formulas, which cannot be further analyzed. We represent these by English alphabets and subscripted alphabets $p, q, r, s, t, p_1, p_2, q_1, q_2, \ldots$, etc. These formulas may have smaller constituents but it is not the role of propositional logic to go into the details of their constructions. The use of letters to represent propositions is not in true sense variables, they simply represent the propositions or statements in a symbolic form, and they are not the variables in the sense used in *predicate logic* (to be discussed later), or in *high-level languages* like *C* or *Fortran*, where a variable stands for a domain of values. For example, an integer variable in a Fortran program stands for any integer number as per the specifications of the language.

The other symbols of propositional logic are *operators* as follows:

$\wedge$ conjunction operator,
$\vee$ disjunction operator,
$\neg$ not or inverting operator,
$\rightarrow$ implication, i.e., if ... than ... rule, and
$\perp$ contradiction (false).

Let following be the propositions:

$p =$ Sun is star.
$q =$ Moon is satellite.

We can construct the following formulas using the above propositions:

$p \wedge q =$ Sun is star *and* Moon is satellite.
$p \vee q =$ Sun is star *or* Moon is satellite tennis.
$\neg p \vee q =$ Sun is *not* star or Moon is satellite.
$\neg p \rightarrow q = if$ Sun is *not* star *then* Moon is satellite.

A formula in propositional logic can be recursively defined as follows:

 (i) Each propositional variable and null are formulas, therefore, $p, q, \phi$ are formulas,
 (ii) If $p, q$ are formulas, then $p \wedge q$, $p \vee q$, $\neg p$, $p \rightarrow q$, $(p)$, are also formulas,
(iii) A string of symbols is a formula only as determined by finitely many applications of above (i) and (ii), and
(iv) nothing else is propositional formula.

This recursive form of the definition can be expressed using $BNF$ (Backups-Naur Form) notation as follows:

1. $formula := atomicformula \mid formula \wedge formula \mid formula \vee formula$
$\mid formula \to formula \mid \neg formula \mid (formula)$
2. $atomicformula := \bot \mid p \mid q \mid r \mid p_0 \mid p_1 \mid p_2 \mid \ldots$

$$(2.1)$$

In the above notation, the symbols—*formula* and *atomicformula*, that appears to the left-hand are called *non-terminals* and represent grammatical classes. The $p, q, r, \bot, p_1$, etc, that appear only to the right-hand side, are called *terminals*, and represent the symbols of the language.

A sentence in the propositional language is obtained through a derivation that starts with a non-terminal, and repeatedly applied the substitution rules from the BNF notations, until the terminals are reached [8].

**Example 2.1** Derivation for $p \wedge q \to r$.

The sequence of substitutions rules to derive this formula, i.e., to establish that it is syntactically correct, are as follows:

$$\begin{aligned}
formula &\Rightarrow formula \to formula \\
&\Rightarrow formula \wedge formula \to formula \\
&\Rightarrow atomic \wedge formula \to formula \\
&\Rightarrow p \wedge formula \to formula \\
&\Rightarrow p \wedge atomic \to formula \\
&\Rightarrow p \wedge q \to formula \\
&\Rightarrow p \wedge q \to atomic \\
&\Rightarrow p \wedge q \to r.
\end{aligned}$$

The symbol *atomic* stands for atomic formula and the symbol "$\Rightarrow$" stands for "implies", i.e., the expression to right to this is implied by the expression to left of "$\Rightarrow$".

The derivation can also be represented by a *derivation-tree* (*parse-tree*), shown in Fig. 2.2. From the derivation-tree, we can obtain another tree shown in Fig. 2.3, called *syntax-tree* or *formation-tree*, by replacing each non-terminal by the child that is an operator under that. There is always unique syntax-tree for every formula. $\square$

Considering two propositions $p, q$, the interpretation (semantics) of the formulas constructed when they are joined using binary operators ($\vee, \wedge, \to$) are shown in the truth-table Table 2.1.

The *Material conditional* '$\to$' joins two simpler propositions, e.g., $p \to q$, read as "if $p$ then $q$". The proposition to the left of the arrow is called the *antecedent* and to the right is *consequent*. There is no such designation for conjunction or disjunction operators because they are commutative operations. The $p \to q$ expresses that $q$ is true whenever $p$ is true. Thus it is true in every case in Table 2.1, except in row three, because this is the only case when $p$ is true but $q$ is not. Using "if $p$ then $q$",

**Fig. 2.2**  Parse-tree for the
expression $p \wedge q \rightarrow r$
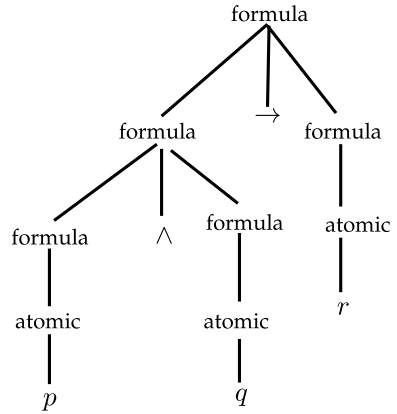


**Fig. 2.3**  Syntax-tree for the
expression $p \wedge q \rightarrow r$
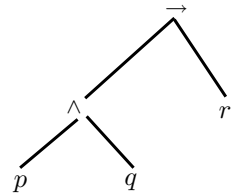


**Table 2.1**  Interpretation of propositional formulas

| $p$ | $q$ | $p \vee q$ | $p \wedge q$ | $p \rightarrow q$ |
|---|---|---|---|---|
| $F$ | $F$ | $F$ | $F$ | $T$ |
| $F$ | $T$ | $T$ | $F$ | $T$ |
| $T$ | $F$ | $T$ | $F$ | $F$ |
| $T$ | $T$ | $T$ | $T$ | $T$ |

we can express that "if it is raining outside then there is a cold over Kashmir".
The material conditional is often confused with *physical causation*. The material
conditional, however, only relates two propositions by their truth values—which is
not the relation of *cause* and *effect*. It is contentious in the literature whether the
material implication represents logical causation.

## 2.4.1   Interpretation of Formulas

The interpretation of formula is assigning truth value to that formula. As discussed
earlier, a formula can be atomic or in may be complex, i.e., joining or atomic formulas.
The following are some definitions related to the interpretation of formulas [1].

**Definition 2.1** (*Satisfied, model, valid, and tautology*) A propositional formula $A$ is *satisfied* iff $\mathscr{I}(A) = True$ for some interpretation $\mathscr{I}$. A *satisfying interpretation* is called model for $A$. The formula $A$ is called valid, denoted by $\models A$, iff $\mathscr{I}(A) = True$ for all interpretations $\mathscr{I}$. A valid propositional formula is also called tautology.

A propositional formula is *unsatisfiable* (also called *contradiction, $\perp$*), *iff* it is not satisfiable, i.e., $\mathscr{I}(A) = False$, for all interpretations $\mathscr{I}$. If $\mathscr{I}(A) = False$ for some interpretation $\mathscr{I}$, then $A$ is called non-valid or falsifiable, and denoted by $\not\models A$.

**Definition 2.2** (*Simultaneously satisfiable*) A set of formulas $S = \{A_1, A_2, \ldots, A_n\}$ is *simultaneously satisfiable iff* there exists an interpretation $\mathscr{I}$ such that $\mathscr{I}(A_i) = True$ for all $i$. The $S$ is unsatisfiable *iff* for every interpretation $\mathscr{I}$ there exits an $i$ such that $\mathscr{I}(A_i) = False$.

### 2.4.2   Logical Consequence

The *logical consequence* or *logically follows* is the central concept in the foundations of logic. It is much more interesting to assume that a set of formulas is true and then to investigates the consequences of these assumptions [1].

Assume that $\theta$ and $\psi$ are formulas (sentences) of a set $\mathscr{P}$, and $\mathscr{I}$ is an interpretation of $\mathscr{P}$. The sentence $\theta$ of propositional logic is true under an interpretation $\mathscr{I}$ iff $\mathscr{I}$ assigns the truth value $T$ to that sentence. The $\theta$ is false under an interpretation $\mathscr{I}$ iff $\theta$ is not true under $\mathscr{I}$.

**Definition 2.3** (*Logical consequence*) A sentence $\psi$ of propositional logic is a *logical consequence* of a sentence (or set of sentences) $\theta$, represented as $\theta \models \psi$, if every interpretation $\mathscr{I}$ that satisfy $\theta$ also satisfy $\psi$.

In fact, $\psi$ need not be true in every possible interpretation, only in those interpretations which satisfy $\theta$, i.e, those interpretations which satisfy every formula in $\theta$. In the formula $((p \rightarrow q) \wedge p) \vdash q$, the $q$ is logical consequence of $((p \rightarrow q) \wedge p)$. The sign '$\vdash$', is sign of *deduction*, and $S \vdash q$ is read as $S$ deduces $q$, where $S$ is a set of formulas and $q$ is the formula.

A sentence of propositional logic is *consistent iff* it is true under at least one interpretation. It is *inconsistent* if it is not consistent.

**Example 2.2** Determine the logical consequence of $\psi = (p \vee r) \wedge (\neg q \vee \neg r)$ from $\theta = \{p, \neg q\}$, i.e., find $\theta \models \psi$, and validity for $\psi$.

Here $\psi$ is logical consequence of $\theta$, denoted by $\theta \models \psi$, because $\psi$ is true under all the interpretations such that $\mathscr{I}(p) = True$, and $\mathscr{I}(q) = False$, is the interpretation, for which $\theta$ is satisfied.

However, $\psi$ is not valid, since it is not true under the interpretation $\mathscr{I}(p) = F$, $\mathscr{I}(q) = T$, $\mathscr{I}(r) = T$.

Further note that $\theta \vdash \psi$ is a valid statement because the expression $\theta \vdash \psi$ is always true.                                                                              □

### 2.4.3 Syntax and Semantics of an Expression

*Syntax* is name given to a correct *structure* of a statement. It is the meaning associated with the expression. It is mapping to the real-world situation is *semantics*. The semantics of a language defines the truth of each sentence with respect to each possible *world*. For example, the usual semantics for interpretation of the statement $(p \lor q) \land r$ is true in a *world* where either $p$ or $q$ or both are *true* and $r$ is *true*. Different worlds can be all the possible sets of *truth* values of $p, q, r$, which is total 8. The truth values are simply the assignment to these variables, and not necessarily the values which are only *true*. For example, $\mathscr{I}(p) = F, \mathscr{I}(q) = F, \mathscr{I}(r) = T$; and $\mathscr{I}(p) = T, \mathscr{I}(q) = F, \mathscr{I}(r) = T$ are the possible worlds for the expression $(p \lor q) \land r$.

### 2.4.4 Semantic Tableau

Semantic tableau is relatively efficient method for deciding satisfiability for the formula of propositional calculus. The method (or algorithm) *systematically* searches for a model for a formula. If it is found, the formula is satisfiable, else not satisfiable. We start with the definition of some terms, and then analyze some formulas to motivate us for the construction of semantic tableau [1].

**Definition 2.4** (*Literal and complementary pair*) A literal is an atom or negation of an atom. For any atom $p$, the set $\{p, \neg p\}$ is called complementary pair of literals. For any formula $A$, $\{A, \neg A\}$ is complementary pair of formulas.

**Example 2.3** Analysis of the satisfiability of a formula.

Consider that a formula $A = p \land (\neg q \lor \neg p)$, has an arbitrary interpretation $\mathscr{I}$. Given this, $\mathscr{I}(A) = T$ iff $\mathscr{I}(p) = T$ and $\mathscr{I}(\neg q \lor \neg p) = T$. Hence, $\mathscr{I}(A) = T$ iff either,

1. $\mathscr{I}(p) = T$ and $\mathscr{I}(\neg q) = T$, or
2. $\mathscr{I}(p) = T$ and $\mathscr{I}(\neg p) = T$.

Hence $A$ is satisfiable if either (1) interpretation holds or (2) holds. But (2) is not feasible. So, $A$ is satisfiable when the interpretation of (1) holds true. Note that the satisfiability of a formula is reduced to the satisfiability of literals.

It is clear that a set of literals is satisfied if and only if it does not contain complementary pair of literals. In the above case, the pair of literals $\{p, \neg p\}$ in case (2) is complementary pair, hence the formula is unsatisfied for this interpretation. However, the first set $\{p, \neg q\}$ is not the complementary pair, hence it is satisfiable.

From the above discussion, we have trivially constructed a model for the formula $A$ by assigning *True* to positive literals and *False* to negative literals. Hence, $p =$

**Fig. 2.4**  Tree for semantic tableau



$$p \wedge (\neg q \vee \neg p)$$

$$p, (\neg q \vee \neg p)$$

$$p, \neg q \qquad p, \neg p$$
$$\odot \qquad\qquad \times$$
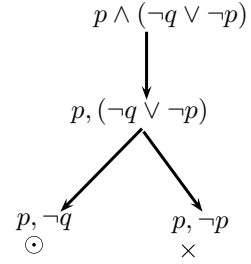
*True*, and q = *False* makes the set in (1) true, hence $\{p = T, q = F\}$ is a model for formula A.

The above is a *search process*, and can be represented by a tree shown in Fig. 2.4. The leaves in the tree represent a set of literals that must be satisfied. A leaf containing complementary pair of literals is marked *closed* by ×, while the satisfying leaf is marked as *open* by ⊙.

The construction process of the tree can be represented as an algorithm, to find out if some model exists for a formula, and what is that model.                       □

**Definition 2.5**  (*Semantic Tableau*) Semantic Tableau is a tree, each node of which will be labeled with a set of formulas, and these formulas are inductively expanded to leaves such that each leaf is marked as open by ⊙ or closed by ×.

**Definition 2.6**  (*Completed tableau*) A semantic tableau whose construction is terminated is called completed tableau . A completed tableau is closed if all the leaves are marked closed. Otherwise, it is open i.e., some leaves are open.

**Definition 2.7**  (*Unsatisfiable formula*) Any formula A is unsatisfiable if its completed tableau $\mathscr{T}$ is closed.

**Corollary 2.1**  (Method for semantic tableau) *A formula A is satisfied if its tableau $\mathscr{T}$ is open. Thus a method for semantic tableau is an algorithm for the validity of a propositional calculus formula.*

**Example 2.4**  Find out whether $(p \vee q) \wedge (\neg p \wedge \neg q)$ is satisfiable, using tableau method.

Let $A = (p \vee q) \wedge (\neg p \wedge \neg q)$. For the satisfaction of A, $\mathscr{I}(A) = True$ for some assignments. That is, $\mathscr{I}(p \vee q) = True$ and $\mathscr{I}(\neg p \wedge \neg q) = True$. Thus, $\mathscr{I}(A)$ is *True* if either,

- $\mathscr{I}(p) = T, \mathscr{I}(\neg p) = True, \mathscr{I}(\neg q) = True$, or
- $\mathscr{I}(q) = True, \mathscr{I}(\neg p) = True, \mathscr{I}(\neg q) = True$.

So that, two sets of literals are,

$(p, \neg p, \neg q)$ and $(q, \neg p, \neg q)$.

Since both contain complementary pairs, hence neither of the literals is satisfiable. So it is impossible to find a model for A, and A is unsatisfiable.

## 2.5 Reasoning Patterns

How can we reason about solving any problem? To a certain extent, it depends on the chosen knowledge representation. The followings are the methods in broad about how the reasoning is performed by humans [2].

Deductive Reasoning

It is a process by which general premises are used to obtain the inferences, which are specific. For example, we may have the following premises and conclusion:

*Premise-I*: I do shopping when the weather is good on weekends.
*Premise-II*: Today is Saturday and the sky is clear.
*Conclusion*: Therefore, I will go for shopping Today.

To perform the deductive reasoning, the problem is first formulated in the way as we did in the above example. Having done this, the conclusions must be valid when the premises are true. Beginning with a small set of *axioms*, *postulates*, and *definitions*, the Greek mathematician Euclid proved a total of 465 geometric propositions as the logical consequences of the input assumptions.

One of the most fundamental rules of inference is *modus ponens* rule. We have the following example for modus ponens.

*Premise-I*: All the men are mortal.
*Premise-II*: Socrates is man.
*Conclusion*: Therefore, Socrates is mortal.

The new knowledge, "Socrates is mortal" has been deduced from the earlier two sentences.

The enumeration table of all possible worlds for *modus ponens* are shown in Table 2.2. We note that it is a valid inference, as the sentence $((p \rightarrow q) \wedge p) \rightarrow q$, with $q$ as the inference implied, is true in all the rows.

Other deductive reasoning approaches are : *modus tollens* and *syllogism*, and *abduction*. The Table 2.3 shows the formulas for these rules.

*Abduction* is deductive type logic, which provides only a "plausible inference." For example, given that: "smoking causes lung cancer" and "Sam died due to lung cancer", through abduction one would infer that "Sam was smoker". However, this

**Table 2.2** Modus ponens is valid inference

| $p$ | $q$ | $p \rightarrow q$ | $(p \rightarrow q) \wedge p$ | $((p \rightarrow q) \wedge p) \rightarrow q$ |
|---|---|---|---|---|
| $F$ | $F$ | $T$ | $F$ | $T$ |
| $F$ | $T$ | $T$ | $F$ | $T$ |
| $T$ | $F$ | $F$ | $F$ | $T$ |
| $T$ | $T$ | $T$ | $T$ | $T$ |

**Table 2.3** Inference rules

| Rule | Formula | Description |
|------|---------|-------------|
| Modus ponens | $((p \rightarrow q) \wedge p) \vdash q$ | If $p$ then $q$; $p$; therefore $q$ |
| Modus tollens | $((p \rightarrow q) \wedge \neg q) \vdash \neg p$ | If $p$ then $q$; not $q$; therefore not $p$ |
| Abduction | $(p \rightarrow q) \wedge q \vdash p$ | if $p$ then $q$; $q$; therefore $q$ |
| Hypothetical syllogism | $((p \rightarrow q) \wedge (q \rightarrow r))$ | if $p$ then $q$; $\vdash (p \rightarrow r)$ if $q$ then $r$; therefore, if $p$ then $r$ |
| Disjunctive syllogism | $((p \vee q) \wedge \neg p) \vdash q$ | Either $p$ or $q$, or both; not $p$; therefore, $q$ |

conclusion is not necessarily true, because there are other reasons also for lung cancer, which are not due to smoking. When statistics and probability theory are used along with abduction, it may result in most probable inferences out of the many likely inferences. To illustrate how the abduction based reasoning works, we consider a logical system comprising a general rule and one specific proposition.

> All successful enterprising industrialists are rich (general rule). Rajan is a rich person (specific proposition). Therefore, a plausible inference can be that Rajan is a successful, enterprising industrialist.

However, this conclusion can be false also, because there are many other paths to richness, such as a lottery, inherited property, coming across a treasure, and so on. If we have a table of all the riches and how they became rich, we may draw the probability of abduction for richness to be true in this case.

Inductive Reasoning

The inductive reasoning arrives at a conclusion about all members of a class. It is based on examination of only a few members of the class and based on that it generalizes for the entire class. It is broadly reasoning from a *specific* to the *general*. For example, the traffic police comes to know about following situation on a particular day about nature of road accidents:

> 1st accident was due to wrong side drive,
> 2nd accident was due to wrong side drive,
> 3rd accident was due to wrong side drive.

> One would logically infer that all the accidents are due to wrong side driving.
> Another example is about the birds for their flying attribute.

> Crow fly,
> peacock fly,
> pigeon fly.

> Thus, we conclude that all the birds fly.
> Another example is about the progressive sum of 1st $n$ odd integers:

$$1 = 1^2$$
$$1 + 3 = 2^2$$
$$1 + 3 + 5 = 3^2$$
$$1 + 3 + 5 + 7 = 4^2$$

Thus, by induction we prove that, the sum of $n$ successive odd integers is $n^2$.

The outcome of the inductive reasoning process will frequently contain some measures of uncertainty because including all possible facts in the premises are usually impossible.

We know that the inference of an accident's example is not always true, and also of "all birds fly" is not true, because, ostrich and penguins do not fly. However, for 1st odd integers sum, it is true.

The deductive or inductive approaches are used in logic, rule-based systems, and in frames.

### Analogical Reasoning

The analogical reasoning assumes that when question is asked, the answer can be derived by analogy, as in the case of following example.

*Premise*: All the 100 m racers get 5% additional in their merit score.
*Question*: How much one 400 m racer will get additional in academic score?
*Conclusion*: Because, 400 m is a race, and an sports activity like 100 m, so it will also benefit one with 5% in final scores.

Analogical reasoning is a type of verbalization of an internalized learning process. An individual uses processes that require the ability to recognize previously encountered experiences. This approach is not very common in AI, however, the case-based reasoning, semantic networks, and frames use this analogical reasoning approach.

### Formal Reasoning

It uses the process of syntactic manipulation of data structures to deduce new facts. A typical example is the mathematical logic used in proving theorems in geometry. For example, proof by *resolution*.

### Procedural and Numeric Reasoning

It uses mathematical models or simulation to solve the problems. The model-based reasoning is an example of this approach.

### Generalization and Abstraction

The approaches of generalization and abstraction, both can be used with the logical and semantic representation of knowledge.

### Meta-level Reasoning

The meta-level reasoning involves the knowledge about what you, how much you know about so and so. Also, which approach to use, how successful the inference will

be, depends on a great extent on which knowledge representation method is used. For example, reasoning by analogy can be more successful with semantic networks than with frames.

### 2.5.1   Rule-Based Reasoning

The rule-based reasoning is also called *pattern matching*, and uses forward and backward chaining. The implementation of rule-based system makes use of modus ponens and other approaches. Consider the rule:

**Rule 1**: If export rises the prosperity increases.

Using the modus ponens, if the premises, e.g., "The export rises" is *true*, the conclusion of the rule is accepted as *true*. We call this accepting the rule as "rule fires". The firing of a rule occurs when all its premises are satisfied, whether all are true or some are false. On firing, the resulting conclusion is stored in the assertion base, to use for further firing of the rules and generate the assertions. When a premise is not available as an assertion, it can be obtained by querying the user, or by firing other rules. Testing of a rule premise or conclusion is as simple as matching a symbol pattern.

Every rule in the knowledge base can be checked to see if its premises or conclusion can be satisfied by previously made assertions. This process of matching, if done using forward chaining, i.e., premises to conclusions. If it is done from conclusions to premises, it is called *backward chaining*.

### 2.5.2   Model-Based Reasoning

A reasoning within a context is important in any reasoning system. In real-life situations, one often provides a lot of missing contexts or out of context information when answering certain queries. This situation can be correctly modeled by supplementing the existing knowledge about the world, with additional context-specific information. When it is supplemented by context information, reasoning within context becomes a deduction process.

The added information may act as constrain to the existing information in the system, as in the absence of this additional information the deduction process has more paths of freedom in the reasoning process. But, due to the availability of this added context information the reasoning task becomes easier because the domain in which reasoning takes place gets restricted (constrained) due to having lesser flexibility of deduction paths to be navigated. This task can be formalized as a task of varying contexts.

The knowledge that comprises the information for reasoning in the model-based system is in the form of a set of models of the world. These models satisfy the

assignments and examples of the world. This is, in contrast, to the use of only the formulas in the first-order predicate logic to describe the world. The other difference is that the model-based approach is motivated from a cognitive point of view – the forerunners of this approach of reasoning are cognitive psychologists who support the "reasoning by examples." When a model-based reasoning system is presented with a query, the reasoning is performed by evaluating the query on these models.

Let us suppose that model-based knowledge base representation $\Gamma$, and a query $\alpha$ are both given, and it is required to find out if $\Gamma$ implies $\alpha$ (i.e., $\Gamma \models \alpha$)? This we can determine in two steps: 1) evaluate $\alpha$ on all the models in the representation, 2) If there is a model of $\Gamma$ that does not satisfy $\alpha$, the $\Gamma$ does not model the alpha (i.e., $\Gamma \not\models \alpha$), otherwise we conclude that $\Gamma \models \alpha$. This means if the model-based representation contains all the models of $\Gamma$, then by definition, this approach verifies the implication correctly, and produces the correct deduction.

However, there is a problem—the representation of $\Gamma$, such that it explicitly holds all the models, is not a plausible solution. The model-based approach is feasible only if $\Gamma$ can be replaced by small model-based representation, and after that also it should correctly support the deduction.

Various topics in reasoning are as follows:

Monotonic versus nonmonotonic reasoning,
Reasoning with uncertainty,
Shallow and deep representation of knowledge,
Semantic networks,
Blackboard approach,
Inheritance approach,
Pattern matching,
Conflict resolution.

These are discussed in current, and the following chapters, in details.

## 2.6 Proof Methods

There are two different methods, one is through *model checking* and other is *deduction* based. The first comprises enumeration of truth-tables, and is always exponential in $n$, where $n$ is the size of the set of propositional symbols. The other, i.e., deduction based approach is repeated application of inference rules. The inference rules are used as operators in the standard search algorithm. In fact, the application of the inference approach to proof is called searching for solution. Proper selection of search directions is important here, as these will eliminate many unnecessary paths that are not likely to result in the goal. Consequently, the proof-based approach for reasoning is considered better and efficient compared to model enumeration/checking based method. The later is exhaustive and exponential in $n$, where $n$ is the size of the set of propositional symbols.

The property, the logical system follows, is the fundamental property of *monotonicity*. As per this, if $S \vdash \alpha$, and $\beta$ is additional assertion, then $S \wedge \beta \vdash \alpha$.

Thereby, the application of inference rules is legitimate (*sound*) rule, which helps in the generation of new knowledge from the existing. If a search algorithm like DFS (depth first search) is used, it will always be possible to find the proof, as it will search the goal, whatever the depth may it be. Hence, the inference method in this case is *complete* also [7].

Before the inference rules are applied on the knowledge base, the existing sentences in the knowledge base (KB) needs to be converted into some *normal form*.

### 2.6.1   Normal Forms

A logical expression can be represented as sum-of-product terms or product-of-sum terms. If a given logical expression is represented as sums of elementary products, then this form is called *disjunctive normal form* (DNF), and if it is represented as product of elementary sums, it is called *conjunctive normal form* (CNF). In DNF, the elementary product terms are called *minterms*, while in a CNF elementary sum terms are called *maxterms*. For a given formula, an equivalent disjunctive normal form with only disjunctions of minterms is called *principle disjunctive normal form* or *sum-of-products canonical form*. Similarly, an equivalent CNF with only conjunctions of maxterms is called *principle conjunctive normal form* or *product-of-sums canonical form* [2].

One technique to get a CNF expression for a given DNF expression, say, $\neg a \neg bc + \neg ab \neg c + \neg abc + a \neg bc$ is given in steps as follows:

1. Considering a DNF expression of three variable $a, b, c$, write down all the minterms: $\neg a \neg b \neg c$, $\neg a \neg bc$, $\neg ab \neg c$, $\neg abc$, $a \neg b \neg c$, $a \neg bc$, $ab \neg c$, $abc$.
2. Cross out all combinations in the original DNF. We are left with $\neg a \neg b \neg c$, $a \neg b \neg c$, $ab \neg c$, $abc$.
3. Next, write the expression in CNF by inverting each subset of three variables and ORing as $(a + b + c)(\neg a + b + c)(\neg a + \neg b + c)(\neg a + \neg b + \neg c)$ in the form of CNF.

Obtaining DNF from CNF is just the reverse process.

### 2.6.2   Resolution

The *resolution rule* is an inference which uses *deduction* approach. It is used in theorem proving. If two disjunctions have complementary literals, then a resultant inference of these is disjunction of these expressions, with complementary terms removed. If $p = p_1 \vee p_2 \vee c$ and $q = q_1 \vee \neg c$ are two formulas, then resolution of

$p$ and $q$ results to dropping of $c$ and $\neg c$ and disjunction is performed of the remaining propositions of $p$ and $q$, as follows:

$$\frac{(p_1 \vee p_2 \vee c), (q_1 \vee \neg c)}{p_1 \vee p_2 \vee q_1} \tag{2.2}$$

The necessary condition for the above is that $C$ should not be a function of any of the $p_1, p_2, q_1$.

**Example 2.5** Show by resolution that $(p \rightarrow q) \rightarrow [(r \wedge p) \rightarrow (r \wedge q)]$ is a tautology:

$$\Rightarrow \neg(\neg p \vee q) \vee [\neg(r \wedge p) \vee (r \wedge q)]$$
$$\Rightarrow (p \wedge \neg q) \vee [(\neg r \vee \neg p) \vee (r \wedge q)]$$
$$\Rightarrow (p \wedge \neg q) \vee [((\neg r \vee \neg p) \vee r) \wedge ((\neg r \vee \neg p) \vee q)]$$
$$\Rightarrow (p \wedge \neg q) \vee [(r \vee \neg r \vee \neg p) \wedge (q \vee \neg r \vee \neg p)]$$
$$\Rightarrow (p \wedge \neg q) \vee [(q \vee \neg r \vee \neg p)]$$
$$\Rightarrow (q \vee \neg r \vee \neg p \vee p) \wedge (q \vee \neg r \vee \neg p \vee \neg q)$$
$$\Rightarrow T \wedge T$$
$$\Rightarrow T.$$

### 2.6.3   Properties of Inference Rules

An inference rule is a mechanical process of producing new facts from the existing facts and rules. The semantics of predicate logic provides a basis for a formal theory of *logical inference*. It allows the creation of new facts from the existing facts and rules [5, 7].

An interpretation of a predicate statement means the assignment of true or false value to that statement. An interpretation that makes a sentence true is said to *satisfy* a sentence. An interpretation that satisfies every member of a set is said to satisfy the set.

**Definition 2.8** (*Logically follows*) If every interpretation that satisfies $S$ also satisfy $X$, then we say the expression $X$ *logically follows* from a set of expressions $S$ (the *knowledge base*). In other words, the knowledge base $S$ entails sentence $X$ if and only if $X$ is true in all *worlds* where knowledge base is true. If a sentence $X$ logically follows $S$, we represent it as $S \models X$.

The term logically follows simply means that $X$ is true for every, potentially infinite interpretations that satisfy $S$. However, it is not a practical way of interpretations. In fact, inference rules provide a computationally feasible way to determine the expression $X$, when it logically follows a set of premises $S$.

An example of an inference rule is *Modus Ponens*:

$$[(P \rightarrow Q) \wedge P] \rightarrow Q \tag{2.3}$$

which is a valid statement (a tautology). Here, the $Q$ also *logically follows* (entails) from $(P \rightarrow Q) \wedge P$. That is, $[(P \rightarrow Q) \wedge P] \models Q$.

**Definition 2.9**  'Sound' inference system.

When every inference $X$ deduced from $S$ also logically follows $S$, then the inference system is *sound*. This is expressed by,

$$S \vdash x \Rightarrow S \models x. \tag{2.4}$$

The '$\vdash$' is sign of 'deduction'.
*Soundness* means that you cannot prove anything that is wrong.                    □

**Definition 2.10**  (*A Complete inference system*) If every $X$ which logically follows $S$ can also be deducted (inferred), then the inference rule is *complete*. This is expressed by

$$S \models x \Rightarrow S \vdash x. \tag{2.5}$$

*Completeness* means that you can prove anything that is right.

Another rule of inference is *Modus tollens*, specified as,

$$[(P \rightarrow Q) \wedge \neg Q] \rightarrow \neg P \tag{2.6}$$

is sound and complete.
     The reader may verify whether the inference rule of modus tollens is sound or complete or both or none?

## 2.7   Nonmonotonic Reasoning

The classical logic or FOPL (first order predicate logic) discussed far, is not all time sufficient to model the real-world knowledge of the world we live in. The reason are: things become false to true or vice-versa over a time, addition of new knowledge in the knowledge base may contradict the existing knowledge (e.g., the statement "the surface of the earth is curved" becomes false on poles), things may be partially true instead either true or false, and some times there is a probability of being some thing true or false, and so on. Hence, there is a requirement of an all together different approach and method of inferencing for real world situations.
     The *Nonmonotonic logic* is the study of those ways of inferring from given knowledge that do not necessarily satisfy the *monotonicity* property, satisfied by all methods

**Fig. 2.5** Nonmonotonic reasoning
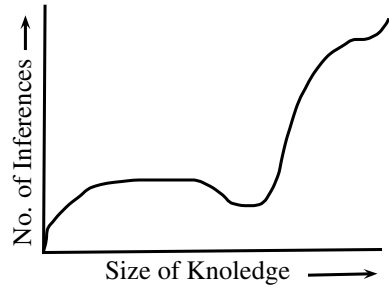


based on classical logic. In classical logic, if a conclusion is warranted on the basis of certain premises (knowledge), no additional premises will ever invalidate the conclusion.

In everyday life, however, it seems clear that we humans draw sensible conclusions from what we know and that, on the face of new information we often have to take back previous conclusions. This happens even when the new information we gathered in no compel us to take back our previous assumptions (see Fig. 2.5).

For example, we may hold the assumption that "most birds fly", but that "penguins are birds that do not fly". On learning that "Tweety is a bird", we infer that "Tweety flies." However, on learning that "Tweety is a penguin," will in no way make us change our mind about the fact that most birds fly, and also that penguins are birds that do not fly or the fact that Tweety is a bird. However, it should make us abandon our conclusion about Tweety's flying capabilities. It is desirable that intelligent automated systems will have to do the same kind of (nonmonotonic) inferences.

Considering that $\Gamma$ is a set of sentences of propositional logic, and $\alpha$ is inferred from it, i.e $\Gamma \vdash \alpha$. For any new propositional sentences $\beta$, if $\Gamma \cup \{\beta\} \vdash \alpha$ then it is *monotonic* reasoning. If it is not necessary that $\Gamma \cup \{\beta\} \vdash \alpha$, then it is *nonmonotonic* reasoning. We note from Fig. 2.5, that some times, even when we add into knowledge base, the number of inferences decreases instead of increasing; and, this is property of nonmonotonic reasoning.

Some of the systems that perform such nonmonotonic inferences are—*negation as failure*, *circumscription*, *modal system*, *default logic*, *autoepistemic logic*, and *inheritance systems*.

## 2.8 Hilbert and the Axiomatic Approach

An axiomatic system comprises a set of *axioms* and a set of *primitives*, where the primitives are object names but, these objects are left undefined. The axioms are the sentences that make assertions about the primitives. Further, these assertions are not provided with any justifications, so they are neither true nor false. The subsequent or new assertions about the primitives are called *theorems*, are rigorous logical consequences of axioms and previously proved theorems.

In 1899 the mathematician David Hilbert published his ground-breaking research in the form of a book. He provided a complex deductive system based on five *groups of axioms*, namely:

1. Axioms of *incidence*,
2. Axioms of *order*,
3. Axioms of *congruence*,
4. Axioms of *continuity*, and
5. an axiom of *parallels*.

As per Hilbert's approach, the basic concepts of geometry comprises *points*, *lines* and *planes* of Euclidean geometry. However, these concepts are never explicitly defined. Instead, they are implicitly defined by the axioms such that, points, lines, and planes are any family of *mathematical objects* that satisfy the given axioms of geometry.

Twenty years later Hilbert was considered as the chief promoter of a program intended to provide solid foundations to arithmetic, based on purely axiomatic methods—the mathematics that model all the computations. It was called *formalist* program, and Hilbert was identified as the champion of the formalist approach to mathematics as a whole [6].

### 2.8.1  Roots and Early Stages

The *formal definitions* in an axiomatic system serves the purpose to simplify the things as they can be used to create new objects made of complex combinations of primitives and previously defined terms (objects and theorems). If a definite meaning is assigned to a primitive of an axiomatic system, called as an *interpretation*, the theorems become meaningful assertions.

Following are some definitions of the axiomatic system.

**Definition 2.11** (*Model (for axiomatic system.)*) If all the axioms are true for a given interpretation, then everything asserted by the theorem is also true. Such an interpretation is called a model for the axiomatic system.

**Definition 2.12** (*Inconsistent (axiomatic system.)*) Since a contradiction can never be true, an axiomatic system using a contradiction can arrive at a logical deduction that it has no model. An axiomatic system with this property is called *inconsistent*.

**Definition 2.13** (*Consistent (axiomatic system.)*) If an abstract axiom system does have a model, then such system is consistent.

**Definition 2.14** (*Isomorphic*) If two models of the same axiom system can be proved as structurally equivalent, then they are isomorphic to each other.

An axiomatic system can have more than one model.

**Definition 2.15** (*Categorical Axioms*) If all models of an axiom system are *isomorphic* then the axiom system is *categorical*.

Thus, for a categorical axiom system, there exists a model—the one and only interpretation in which its theorems are all true.

The qualities—*truth, logical necessity, consistency*, and *uniqueness* were considered as the base of classical Euclidean geometry. Till recently, it was accepted that Euclidean geometry is the only way to think about space. Now, the axiomatic systems are taken as the basis of geometry, and later all of the mathematics including the computational mathematics and algorithms.

Hilbert's definition of an axiomatic system lays the foundation of *theory* and verifies that this system satisfies three main properties: *independence*, *consistency*, and *completeness*. He proposed that just as in geometry, this kind of axiomatic analysis should be applied to other fields of knowledge, and in particular to physical theories. When we study any system of axioms as per Hilbert's perspectives, the focus of interest remained always on the *disciplines* themselves rather than on the axioms. The axioms are just a means to improve our understanding of the discipline, and not aimed to turn mathematics into a formally axiomatized game. For example, in the case of geometry, a set of axioms were selected in such a way that they reflected the basic manifestations of the intuition of space [4].

### 2.8.2 Axiomatics and Formalism

To understand the role of axioms, we will discuss the axioms of the set, as they are useful in reasoning and inferences. By analyzing the mathematical arguments, logicians become convinced that the notion of "set" is the most fundamental concept of mathematics. For example, it can be shown that the notion of an integer can be derived from the abstract notion of a set. Thus, in our world all the objects are sets, and we do not postulate the existence of any more primitive objects. To support this intuition, we can think our universe as all sets which can be built by successive collecting processes, starting from the empty set, and we allow the formation of infinite sets.

The first set of axioms for a general set theory was given by E. Zermelo in 1908, and later developed by A. Fraenkel, hence usually referred to as Zermelo-Fraenkel (ZF) set theory, the one we are most concerned. Another systems of axioms, which has only finitely many axioms, but is less natural, was developed by von Neumann, Bernays, and Gödel. The later is usually referred to as Gödel-Bernays (GB) set theory.

Following are some of the important axioms of ZF set theory [3, 8].

1. *Axioms of Extensibility.*

$$\forall x \forall y (\forall z (z \in x \leftrightarrow z \in y) \rightarrow x = y) \tag{2.7}$$

The above says that set is determined by its members. We can define the subsets as follows:

$$x \subseteq y \leftrightarrow \forall z(z \in x \rightarrow z \in y). \tag{2.8}$$

Also,

$$x \subset y \leftrightarrow x \subseteq y \wedge \neg x = y. \tag{2.9}$$

2. *Axiom of the Null set.*

$$\exists x \forall y(\neg y \in x). \tag{2.10}$$

The set defined by this axiom is the empty or null set and we denote it by $\phi$.

3. *Axiom of Unordered Pairs.*

$$\forall x \forall y \exists z \forall w(w \in z \leftrightarrow w = x \vee w = y). \tag{2.11}$$

We represent the set $z$ by $\{x, y\}$. Also, $\{x\}$ is $\{x, x\}$ and we put $\langle x, y \rangle = \{\{x\}, \{x, y\}\}$. The set $\langle x, y \rangle$ is called *ordered pair* of $x$ and $y$.
Using the above we can define a function as follows: a function is a set $f$ of ordered pairs such that $\langle x, y \rangle, \langle x, z \rangle \in f \rightarrow y = z$. The set of $x$ such that $\langle x, y \rangle \in f$ is called *domain*, and set of $y$ is called *range*. We say, $f$ maps in set $u$ if the range of $f$ is in $u$.

4. *Axiom of set Union.* It can be expressed as:

$$\forall x \exists y \forall z(z \in y \leftrightarrow \exists t(z \in t \wedge t \in x)). \tag{2.12}$$

The above says that $y$ is union of all sets in $x$. Using the axiom Eq. 2.12, we can deduce that given $x$ and $y$, there exists $z$, such that $z = x \cup y$, that is, $t \in z \leftrightarrow t \in x \vee t \in y$.
To motivate for the next axiom being described, if $x$ is an integer, the successor of $x$ will be defined as $x \cup \{x\}$. Then the "axiom of infinity" generates a set that contains all the integers and thus infinite.

5. *Axiom of Infinity.* It can be expressed as follows, and we understand that it is the principle of Induction.

$$\exists x(\phi \in x \wedge \forall(y \in x \rightarrow y \cup \{y\} \in x). \tag{2.13}$$

6. *Axiom of Power Set.* This axiom states that there exists for each $x$ the set $y$ for all the subsets of $x$.

$$\forall x \exists y \forall z(z \in y \leftrightarrow z \subseteq x). \tag{2.14}$$

If the axiom of extensionality is dropped, the resulting system may contain atoms, i.e., sets $x$ such that $\forall y(\neg y \in x)$ yet the sets $x$ are different. Indeed, one possible view is that integers are atoms and should not be taken as sets.

The first interesting axiom is the Axiom of Infinity. If we drop it, then we can take a model for ZF set of all finite sets which can be built from $\phi$.

The axioms discussed above can be used to prove theorems, like, *mathematical induction*, *invertible functions*, and in fact another theorem of set theory, as well as the corollaries, but the same are not appropriate to cover here, and a curious reader is encouraged to refer the literature given in the bibliography.

## 2.9 Summary

Logic is used for valid deductions, and it avoids fallacy reasoning. Logic is also useful in *argumentation theory*—a study of how conclusions can be reached through logical reasoning, that is, whether the claims are soundly based on premises or not. Argumentation includes debate and negotiation, that are concerned with reaching mutually acceptable conclusions. The logic is used in proofs, games, and puzzles solutions. The arguments have the internal structure: comprising of premises, reasoning process, and consequence.

The most commonly used, Propositional logic, represents sentences using single symbols, called *atoms*, which are joined using the operators $\vee, \wedge, \neg, \rightarrow$ to create compound sentences. The sign of "$\rightarrow$" in $p \rightarrow q$ is *material implication*, also called *conditional join, if p then q*. Propositional logic expressions are called sentences/statements; these are interpreted as *true* or *false*. The sentences are called *wff*, and are defined recursively. A formula is a *syntactic* concept, which means whether or not a string of symbols is a formula.

The meaning (*semantics*) is associated with each formula by defining its *interpretation*, which assign a value *true* ($T$) or *false* ($F$) to every formula. Interpretation of a statement means the assignment of true values to its atoms. A set of truth values assigned to the atoms in a statement is called its *world*. Assignment of truth values to the atoms in a statement, which makes the statement true is called *model* of the statement.

The model checking is the process of truth-table enumeration, and is exponential on $n$, the number of atoms in a statement. The derivation can also be represented by a *derivation-tree (parse-tree)*.

A propositional formula $A$ is *satisfied iff* $v(A) = True$ for some interpretation $v$. A satisfying interpretation is called *model* for $A$. The formula $A$ is called *valid*, denoted by $\models A$, iff $v(A) = True$ for all interpretations $v$. A sentence is logically true (valid) iff it is true under every interpretation. $\models \theta$ means that $\theta$ is valid.

A reasoning, in which addition of new knowledge may produce inconsistency in the knowledge base, is called *nonmonotonic reasoning*. As per the property of *monotonicity*, if $S \vdash \alpha$, and $\beta$ is additional assertion, then $S \wedge \beta \vdash \alpha$. The *Nonmonotonic logic* is the study of those systems that do not satisfy the *monotonicity* property satisfied by all methods based on classical logic.

The reasoning pattern comprises *inference methods*: *modus ponens, modus tollens, syllogism*; and *Proof methods: resolution theorem, model checking, model checking,*

*Normal forms.* Deducing new knowledge from the existing set of the knowledge base is called *inferencing*. The *Modus ponens, modus tolens, syllogism* are inference rules, and *Sound* and *Complete* are good properties of inference systems.

Semantic tableau is a method for deciding satisfiability for the formula of propositional calculus, which *systematically* searches for a model for a formula. If it is found the formula is satisfiable, else not satisfiable. *Semantic Tableau* is a tree, each node of which will be labeled with a set of formulas, and these formulas are inductively expanded to leaves such that each leaf is marked as *open* by ⊙ or *closed* by ×.

The *resolution rule* is an inference which uses *deduction* approach. It is used in theorem proving.

If every interpretation that satisfies $S$ also satisfy $X$, then we say expression $X$ *logically follows* from a set of expressions $S$ (the *knowledge base*). The *Soundness* means that you cannot prove anything that is wrong, and *Completeness* means that you can prove anything that is right.

An axiomatic system comprises a set of *axioms* and *primitives*.

## Exercises

1. Prove the following assertions:

   a. $\alpha$ is valid only if $\mathbf{T} \models \alpha$, where $\mathbf{T}$ stands for True.
   b. For any (not necessarily valid) $\alpha$, $\mathbf{F} \models \alpha$, where $\mathbf{F}$ is logically False.
   c. $\Gamma \models \beta$ if and only if the sentence $(\Gamma \Rightarrow \beta)$ is valid. Here, *Gamma* is knowledge base and $\beta$ is sentence.
   d. $\alpha \equiv \beta$ if and only if the sentence $(\alpha \Leftrightarrow \beta)$ is valid. Here, $\alpha$ and $\beta$ are sentences.

2. Give your argument in favor and against, that the *material join* $p \rightarrow q$ is not same as the *cause effect* $p \rightarrow q$.

3. Determine, which of the following formulas are valid /satisfied /contradiction?

   a. $((p \rightarrow q) \wedge (\neg p \rightarrow r)) \rightarrow (q \vee r)$
   b. $(p \vee q) \rightarrow (p \wedge q)$
   c. $p \rightarrow \neg q$
   d. $(p \wedge q) \rightarrow (p \vee q)$.

4. Show that $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$ is valid (Hint: Construct truth-table, the interpretation of this formula shall be true for all the worlds).

5. Show that formula $(\neg p \wedge \neg q) \wedge (p \vee q)$ is unsatisfiable (Hint: Construct truth-table).

6. Assume a vocabulary with only four propositions, $A$, $B$, $C$, and $D$. Find out the number of *models* for each of the following sentences?

   a. $P \Leftrightarrow Q \Leftrightarrow R$.
   b. $P \wedge Q$,
   c. $(P \wedge Q) \vee (Q \wedge R)$,

7. If $S$ is a set of propositional formulas, then show that $S \models F$ iff $S \cup \{\neg F\}$ is contradiction (Hint: A set of propositional formulas is contradiction, iff every valuation of $S$ there is a formula $p$ in the set such that $S \not\models p$).
8. If $\theta \models \psi$ then prove that $\theta \cup \{A\} \models \psi$ for any formula $A$.
9. If $\theta \models \psi$ and $A$ is valid then prove that $\theta - \{A\} \models \psi$ for any formula $A$.
10. Find out the satisfiability of the following formulas using semantic Tableau methods:

   a. $p \wedge (\neg p \vee \neg q)$.
   b. $(p \wedge q) \vee (\neg p \wedge \neg q)$.

11. Show that if $S$ is unsatisfiable then $S - \{A_i\}$ is also unsatisfiable for every $i \leq i \leq n$.
12. Establish the consistency/inconsistency of the following statements using Tableau method.

   a. $(p \wedge q) \vee (p \rightarrow q)$,
   b. $(\neg q \rightarrow \neg p) \leftrightarrow (p \rightarrow q)$,
   c. $(\neg p \vee \neg q) \wedge (p \vee q)$.

13. Convert the following statements into CNF:

   a. $a\neg bc + bc + \neg ab\neg c$,
   b. $a + bc + \neg b\neg c$.

14. Convert the following statements into DNF:

   a. $(a + b)(a + \neg b + c)(a + \neg c)$,
   b. $(\neg a + b) \vee (a \rightarrow b \rightarrow c)$.

15. Find out the resolvent for $\{p \rightarrow q, \neg q \vee \neg r, r \rightarrow p\}$.
16. Write a recursive algorithm $TRUE(\mathbf{x}, \mathbf{M})$ that returns true if and only if propositional logic sentence $x$ is true in the model $\mathbf{M}$, where $\mathbf{M}$ assigns a truth value for every symbol in $x$. The algorithm should run in time linear in the size of the sentence.

# References

1. Ben-Ari M (2008) Mathematical logic for computer science, 2nd edn. Springer International, London. ISBN-978-81-8128-344-3
2. Chowdhary KR (2012) Fundamentals of discrete mathematical structures, 2nd edn. EEE PHI India. ISBN: 978-81-203-4506-5
3. Cohen PJ (2008) Set theory and the continuum hypothesis. Dover, New York
4. Davis M, Putnam H (1960) A computing procedure for quantification theory. J ACM 7(3):201–215. https://doi.org/10.1145/321033.321034
5. Gelder AV (1991) The well-founded semantics for general logic programs. J ACM 38(3):620–650
6. Hilbert D (2005) The foundations of geometry. Ebook # 17384

7. Kaushik S (2002) Logic and prolog programming. New Age International, New Delhi
8. Puppes P (1972) Axiomatic set theory. Dover, New York
9. Shankar N (2009) Automated deduction for verification. ACM Comput Surv 41(4):20:1. https://doi.org/10.1145/1592434.1592437

# Chapter 3
# First Order Predicate Logic

**Abstract**  The first order predicate logic (FOPL) is backbone of AI, as well a method of formal representation of Natural Language (NL) text. The Prolog language for AI programming has its foundations in FOPL. The chapter demonstrates how to translate NL to FOPL in the form of facts and rules, use of quantifiers and variables, syntax and semantics of FOPL, and conversion of predicate expressions to clause forms. This is followed with unification of predicate expressions using instantiations and substitutions, compositions of substitutions, unification algorithm and its analysis. The resolution principle is extended to FOPL, a simple algorithm of resolution is presented, and use of resolution is demonstrated for theorem proving. The interpretation and inferences of FOPL expressions are briefly discussed, along with the use of Herbrand's universe and Herbrand's theorem. At the end, the most general unifier (mgu) and its algorithms are presented, and chapter is concluded with summary.

## 3.1   Introduction

This chapter presents a formulation of first-order logic which is best suited as a basic theoretical instrument—a computer based theorem proving program. As per the requirements of theory, an inference method should be *sound*—allows only logical consequences of premises deducible from the premises. In addition, it should be *effective*—algorithmically decidable whether a claimed application of the inference principle is really an application of it. When the inference principle is performed by computer, the complexity of the inference principle is not an issue. However, for more powerful principles, usage of combinatorial information processing for single application may become dominant.

The system described in the following is an inference principle—the *resolution principle*, is a machine-oriented rather than human-oriented system. Resolution principle is quite powerful in psychological sense also, as it obeys a single type of

inference, which is often beyond the ability of the human to grasp. In theoretical sense, it is a single inference principle that forms the complete system of first-order logic. However, this latter property is not of much significance, but it is interesting in the sense that no any other *complete system* of first-order logic is based on just one inference principle, if ever one tries to realize a device of introducing a logical axioms, or by a schema as an inference principle. The principle advantage of using the resolution is due to its ability that allows us to avoid any major combinatorial obstacles to efficiency, which used to be a serious problem in earlier theorem-proving procedures.

**Learning Outcomes of this Chapter**:

1. Translate a natural language (e.g., English) sentence into predicate logic statement. [Usage]
2. Apply formal methods of symbolic predicate logic, such as calculating validity of formula and computing normal forms. [Usage]
3. Use the rules of inference to construct proofs in predicate logic. [Usage]
4. Convert a logic statement into clause form. [Usage]
5. Describe the strengths and limitations of predicate logic. [Familiarity]
6. Apply resolution to a set of logic statements to answer a query. [Usage]
7. Implement a unification-based type-inference algorithm for a simple language. [Usage]
8. Precisely specify the invariants preserved by a sound type system. [Familiarity]

## 3.2  Representation in Predicate Logic

The first Order Predicate Logic (FOPL) offers formal approach to reasoning that has sound theoretical foundations. This aspect is important to mechanize the automated reasoning process where inferences should be correct and *logically sound*.

The statements of FOPL are flexible enough to permit the accurate representation of *natural languages*. The words—*sentence* or *well formed formula* will be indicative of predicate statements. Following are some of the translations of English sentences into predicate logic:

- English sentence: Ram is man and Sita is women.
  Predicate form: $man(Ram) \land woman(Sita)$
- English sentence: Ram is married to Sita.
  Predicate form: $married(Ram, Sita)$
- English sentence: Every person has a mother.
  The above can be reorganized as: For all $x$, there exists a $y$, such that if $x$ is person then $x$'s mother is $y$.
  Predicate form: $\forall x \exists y[person(x) \Rightarrow hasmother(x, y)]$

- English sentence: If $x$ and $y$ are parents of a child $z$, and $x$ is man, then $y$ is not man.
  $\forall x \forall y [[parents(x, z) \wedge parents(y, z) \wedge man(x)] \Rightarrow \neg man(y)]$

We note that predicate language comprises constants {Ram, Sita}, variables {$x, y$}, operators {$\Rightarrow, \wedge, \vee, \neg$}, quantifiers {$\exists, \forall$} and functions/ predicates {$married(x, y), person(x)$}. Unless specifically mentioned, the letters $a, b, c, \ldots$ at the beginning of English alphabets shall be treated as constants to indicate names of *objects* and *entities*, and those at the end, i.e., $u, v, w, \ldots$ shall be used as variables or identifiers for objects and entities.

To indicate that an expression is universally true, we use the *universal quantifier* symbol $\forall$, meaning 'for all'. Consider the sentence "any object that has a feathers is a bird." Its predicate formula is: $\forall x[hasfeathers(x) \Rightarrow isbird(x)]$. Then certainly, $hasfeathers(parrot) \Rightarrow isbird(parrot)$ is true. Some expressions, although not always *True*, are *True* at least for some objects: in logic, this is indicted by 'there exists', and the *existential quantifier* symbol $\exists$ is used for this. For example, $\exists x[bird(x)]$, when *True*, this expression means that there is at least one possible object, that when substituted in the position of $x$, makes the expression inside the parenthesis as *True* [1].

Following are some examples of representations of knowledge FOPL.

**Example 3.1** Kinship Relations.

$$mother(namrata, priti).\text{(That is, Namrata is mother of Preeti.)}$$
$$mother(namrata, bharat).$$
$$father(rajan, priti).$$
$$father(rajan, bharat).$$
$$\forall x \forall y \forall z [father(y, x) \wedge mother(z, x) \Rightarrow spouse(y, z)].$$
$$\forall x \forall y \forall z [father(y, x) \wedge mother(z, x) \Rightarrow spouse(z, y)].$$
$$\forall x \forall y \forall z [mother(z, x) \wedge mother(z, y) \Rightarrow sibling(x, y)].$$

In above, the predicate $father(x, y)$ means $x$ is father of $y$; $spouse(y, z)$ means $y$ is spouse of $z$, and $sibling(x, z)$ means $x$ is sibling of $y$. $\square$

**Example 3.2** Family tree.

Suppose that we represent "Sam is Bill's father" by *father(sam, bill)* and "Harry is one of Bill's ancestors" by *ancestor(harry, bill)*. Write a wff to represent "Every ancestor of Bill is either his father, his mother, or one of their ancestors".

$$\forall x \forall y [ancester(y, bill) \Rightarrow (father(y, bill) \vee mother(y, bill))$$
$$\vee ((father(x, bill) \wedge ancester(y, x))$$
$$\vee ((mother(x, bill) \wedge (ancester(y, x))].$$

**Example 3.3**  Represent the following sentences by predicate calculus wffs.

1. A computer system is intelligent if it can perform a task which, if performed by
   a human, requires intelligence.

$$\exists x[[(perform(human, x) \rightarrow requires(human, intelligence))$$
$$\wedge (perform(computer, x)] \rightarrow intelligent(computer))]$$

2. A formula whose main connective is $\Rightarrow$ is equivalent to a formula whose main
   connective is $\vee$.

$$\forall x \forall y[(formula(x) \wedge mainconnective(x,' \Rightarrow'))$$
$$\wedge (formula(y) \wedge mainconnective(y, \vee))$$
$$\rightarrow x \equiv y].$$

3. If a program cannot be told a fact, then it cannot learn that fact. $\forall x[(program(x) \wedge \neg told(x, fact)) \rightarrow \neg learn(x, fact)]$ □

**Example 3.4**  Blocks World.

Consider that there are physical objects, like—*cuboid*, *cone*, *cylinder* placed on the
table-top, with some relative positions, as shown in Fig. 3.1. There are four blocks
on the table: $a, c, d$ are cuboid, and $b$ is a cone. Along with these there is a robot
arm, to lift one of the object having clear top.

**Fig. 3.1**  Blocks world
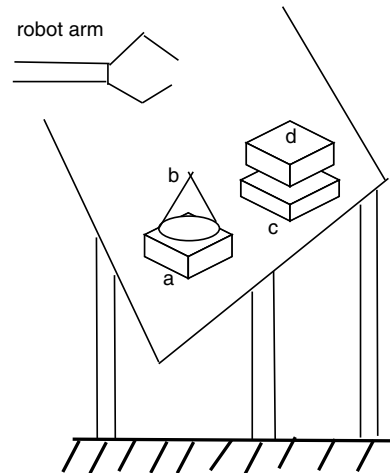
Following is the set of statements about the blocks world (called knowledge base):

*cuboid*(*a*).
*cone*(*b*).
*cuboid*(*c*).
*cuboid*(*d*).
*onground*(*a*).
*onground*(*c*).
*ontop*(*b*, *a*).
*ontop*(*d*, *c*).
*toplcear*(*b*).
*topclear*(*d*).
$\forall x \forall y [toplcear(x) \wedge toplcear(y) \wedge \neg cone(x) \Rightarrow puton(y, x)]$.

The knowledge specified in the blocks world indicate that objects $a, c, d$ are cuboid, $b$ is cone, $a$, $c$ are put on the ground, and $b$, $d$ are on top of $a$, $c$, respectively, and the top of $b$, $d$ are clear. These are called facts in knowledge representation. At the end, the rule says that there exists objects $x$ and $y$ such that both have their tops clear and $x$ is not a cone, then $y$ can be put on the object $x$. □

Bound and Free Variables

A variable in a wff is bound if it is within the scope of a quantifier naming the variable, otherwise the variable is free. For example, in $\forall x(p(x) \rightarrow q(x, y))$, $x$ is bound and $y$ is free; in $\forall x(p(x) \rightarrow q(x)) \rightarrow r(x)$, the $x$ in $r(x)$ is free variable. In the latter case it is better to rename the variable to remove the ambiguity, hence we rephrase this statement as $\forall x(p(x) \rightarrow q(x)) \rightarrow r(z)$. An expression can be evaluated only when all the variables in that are bound.

If $F_1, F_2, \ldots, F_n$ are wffs with $\wedge, \neg$ as operators in each of $F_i$, then $F_1 \vee F_2 \vee \cdots \vee F_n$ is called *DNF* (disjunctive normal form). Alternatively, if operators in $F_i$ are $\vee, \neg$ then $F_1 \wedge F_2 \wedge \cdots \wedge F_n$ is called *CNF* (conjunctive normal form). The expression $F_i$ called a *term*, consists only literals. We will prefer the *CNF* for predicate expression. Thus, for an inference to be carried out, it is necessary to convert a predicate expression to *CNF*. For example, $\exists x[p(x) \Rightarrow q(x)]$ can be converted to $\neg p(a) \vee q(a)$, where $a$ is an instance of variable $x$. The expression $\neg p(a) \vee q(a)$ is a term of *CNF*. A formula in *CNF*, comprising $\wedge, \vee, \neg$ along with constants, variables, and predicates, is called *clausal* or *clause* form [2].

## 3.3 Syntax and Semantics

Two types of semantics are defined for the programing languages: (1) *operational* semantics, and, (2) *fixpoint* semantics. The operational semantics defines input-output relation computed by a program in terms of the individual operations performed by the program inside a machine, like, basic logical and arithmetic operations. The meaning of a program is input-output relation obtained by executing the

program on a machine. The other semantics—fixpoint semantics, is machine independent. It defines the meaning of a program to be the input-output relation which is the minimal fixpoint of a transformation associated with the program. The Fixpoint semantics is used to justify existing methods for proving properties of programs, and to justify new methods of proof.

We know the distinction between the *syntax* and the *semantics* from previous chapter as well from the study of programming languages. The Syntax deals with the formal part of language in abstraction from its meaning. It concerns with the definition of *well-formed formulas*. Syntax in its narrow sense and also deals with the study of axioms, rules of reference and proofs, which constitute proof theory. Semantics is concerned with the interpretation of language and includes such notions as meaning, logical implication and truth.

It is convenient to restrict attention to predicate logic programs written in *clausal form*. Such programs have an especially simple syntax but retain all the expressive power of the full predicate logic [3].

- *Atomic formula*. A string of symbols consisting of a predicate symbol of degree $n \geq 0$ followed by $n$ terms is an *atomic formula*.
- *Clause*. A clause is a disjunction $L_1 \vee \cdots \vee L_n$ of literals $L_i$, each of which is *atomic formula* $P(t_1, \ldots, t_m)$ or the negation of atomic formulas, where $P$ is a predicate symbol and $t_i$, are *terms*. A finite set (possibly empty) of literals is called a clause. The empty clause is denoted by: []
- *Sentence*. A sentence is a finite set of clauses.
- *Literals*. An atomic formula is a literal; and if $A$ is an atomic formula then $\neg A$ is also literal.
  The atomic formulas are *positive* literals, and negations of atomic formulas are *negative* literals.
- *Term*. A term is either a variable or an expression like $f(t_i, \ldots, t_n)$, where $f$ is a function symbol, $t_i$ are terms, and constants are 0-*ary* function symbols. A variable is also a term, and a string of symbols comprising a function symbol of degree $n \geq 0$ followed by $n$ terms is again a term.
  A set of clauses $\{C_1 \ldots, C_n\}$ is interpreted as a conjunction of clauses $C_1 \ldots C_n$. A clause $C$ containing just the variables $x_1, \ldots, x_n$ is called universally quantified. For example,

$$\text{for all } x_1, \ldots, x_n C \tag{3.1}$$

  is universally quantified clause.
- *Ground Literals*. A literal having no variables is called Ground Literal.
- *Ground clauses*. A clause with every member of it as a ground literal, is called a Ground Clause. Empty clause—[] is a Ground Clause.
- *Well-formed expressions*. The Terms and Literals are (the only) Well-Formed expressions.
- *Lexical Order of Well-formed expressions*. This is set of all well formed expressions ordered in lexical order. The ordering is as follows: $A$ precedes $B$ if $A$ is shorter

than $B$. If $A$ and $B$ have same length, then $A$ has the alphabetically earlier symbol in the first symbol position, at which $A$ and $B$ have distinct symbols.

- *Herbrand Universe*. It is set of ground terms associated with any set of $S$ of clauses. Let $F$ be the set of all function symbols which occur in clause set $S$. If $F$ contains any function symbols of degree 0, then the functional vocabulary of $S$ is $F$, otherwise, it is $\{a\} \cup F$, where $a$ is a ground term. In this case, the Herbrand universe of $S$ is set of all ground terms with only symbols of the functional vocabulary of $S$.
- *Models*. It is a set of ground literals having no complementary pair. If $M$ is a Model and $S$ is a set of *ground clauses*, then $M$ is a model of $S$ if, for all $C \in S$, $C$ contains a member of $M$. In general, if $S$ is any set of clauses, and $H$ is the Herbrand Universe of $S$, then $M$ is model of $H(S)$.
- *Satisfiability*. A set $S$ is Satisfiable if there is a model of $S$, otherwise $S$ is Unsatisfiable.

For every sentence $S_1$ of first order predicate logic there is always a sentence $S_2$ in *Clausal Form* which is satisfiable if and only if $S_1$ is also satisfiable. In other words, for every non-clause form sentence there is a logically equivalent clause form sentence. Due to this, all questions concerning to the *validity* or *satisfiability* of sentences in FOPL can be addressed to sentences in clausal form.

Procedure for obtaining clausal-form for any well-formed formula (*wff*) are discussed later in this chapter. In the above we have defined part of the syntax of predicate logic, which is concerned with the specification of well-formed formulas. The formalism we are going to use in the next section is based on the notions of *unsatisfiability* and *refutation* rather than upon the notions of *validity* and *proof*.

To work on the criteria of refutation and unsatisfirability, it is necessary to convert the given *wff* into clausal form.

To determine whether a finite set of sentences $(S)$ of first-order predicate is satisfiable, it is sufficient to assume that each sentence in $S$ is in clause form, and there is no existential quantifiers as the prefix to $S$. In addition, the matrix of each sentence in $S$ is assumed to be a disjunction of formulas, each of which is either atomic formula or the negation of an atomic formula. Therefore, the syntax of $S$ is designed such that the syntactical unit is a finite set of sentences in this special form, called *clause form*. Towards the end of conversion process, the quantifier prefix is omitted from each sentence, since it is necessary that universal quantifiers bind each variable in the sentence. The matrix of each sentence is simply a set of disjuncts and the order and multiplicity of the disjuncts are not important.

## 3.4 Conversion to Clausal Form

Following are the steps to convert a predicate formula into clausal-form [2].

1. Eliminate all the implications symbols using the logical equivalence: $p \rightarrow q \equiv \neg p \vee q$.

2. Move the outer negative symbol into the atom, for example, replace $\neg\forall x\ p(x)$ by $\exists x\neg p(x)$.
3. In an expression of nested quantifiers, existentially quantified variables not in the scope of universal quantifiers are replaced by constants. Replace $\exists x\forall y(f(x)\to f(y))$ by $\forall y(f(a)\to f(y))$.
4. Rename the variables if necessary. For example, in $\forall x(p(x))\to q(x)$, rename second free variable $x$, as $\forall x(p(x)\to q(y))$.
5. Replace existentially quantified variables with *Skolem* functions; then eliminate corresponding quantifiers. For example, for $\forall x\exists y[\neg p(x)\vee q(y)]$, we obtain $\forall x[\neg p(x)\vee q(f(x))$. These newly created functions are celled *Skolem functions*, and the process is called *Skolemization*.
6. Move the universal quantifiers to the left of the equation. For example, substitute $\exists x[\neg p(x)\vee\forall y\ q(y)]$ by $\exists x\forall y[\neg p(x)\vee q(y)]$
7. Move the disjunctions down to the Literals, i.e., terms should be connected by conjunctions only, vertically.
8. Eliminate the conjunctions.
9. Rename the variables, if necessary.
10. Drop all the universal quantifiers, and write each term in a separate line.

The resulting sentence is a CNF, and suitable for inferencing using resolution.

**Example 3.5** Convert the expression $\exists x\forall y[\forall z\ p(f(x),y,z)\Rightarrow(\exists u\ q(x,u)\wedge\exists v\ r(y,v))]$ to clausal form.

The steps discussed above are applied precisely, to get the clausal form of the predicate formula.

1. Eliminate implication.

$$\exists x\forall y[\neg\forall z\ p(f(x),y,z)\vee(\exists u\ q(x,u)\wedge\exists v\ r(y,v))]$$

2. Move negative symbols to the atom.

$$\exists x\forall y[\exists z\neg p(f(x),y,z)\vee(\exists u\ q(x,u)\wedge\exists v\ r(y,v))]$$

3. Replace existentially quantified variables not in the scope of universal quantifier to constants.

$$\forall y[\exists z\neg p(f(a),y,z)\vee(\exists u\ q(a,u)\wedge\exists v\ r(y,v))]$$

4. Rename variables (not required in this example.)
5. Replace existentially quantified variables that are functions of universal quantified variables, by Skolem functions:

$$\forall y[\neg p(f(a),y,g(y))\vee(q(a,h(y)\wedge r(y,l(y)))]$$

6. Move $\forall$ to left is not required in this example.
7. Move disjunctions down to Literals.

$$\forall y[(\neg p(f(a), y, g(y)) \lor (q(a, h(y)))) \land (\neg p(f(a), y, g(y)) \lor r(y, l(y)))]$$

8. Eliminate conjunctions.

$$\forall y[\neg p(f(a), y, g(y)) \lor (q(a, h(y)), (\neg p(f(a), y, g(y)) \lor r(y, l(y)))]$$

9. Renaming variable is not required in this example.
10. Drop all universal quantifiers and write each term on separate line.

$$\neg p(f(a), y, g(y)) \lor (q(a, h(y)),$$
$$\neg p(f(a), y, g(y)) \lor r(y, l(y)).$$

**Example 3.6** Convert the following wff to clause form.

$$(\forall x)(\exists y)\{[p(x, y) \Rightarrow q(y, x)] \land [q(y, x) \Rightarrow s(x, y)]\}$$
$$\Rightarrow (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]$$

For $[p(x, y) \Rightarrow q(y, x)] \land [q(y, x) \Rightarrow s(x, y)]$ by application of syllogism, it can be reduced to $[p(x, y) \Rightarrow s(x, y)]$. Thus, original expression reduces to:

$$= (\forall x)(\exists y)[p(x, y) \Rightarrow s(x, y)] \Rightarrow (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]$$
$$= \neg(\forall x)(\exists y)[p(x, y) \Rightarrow s(x, y)] \lor (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]$$
$$= (\exists x)\neg(\exists y)[p(x, y) \Rightarrow s(x, y)] \lor (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]$$
$$= (\exists x)(\forall y)\neg[p(x, y) \Rightarrow s(x, y)] \lor (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]$$
$$= (\forall y)\neg[p(a, y) \Rightarrow s(a, y)] \lor [p(a, y) \Rightarrow s(a, y)]$$
$$= [p(a, y) \land \neg s(a, y)] \lor [\neg p(a, y) \lor s(a, y)]$$
$$= T$$

## 3.5 Substitutions and Unification

The following definitions are concerned with the operation of *instantiation*, i.e, substitutions of terms for variables in the well-formed expressions and in sets of well-formed expressions [8].

*Substitution Components*

A substitution component is any expression of the form $T/V$, where $V$ is any variable and $T$ is any term different from $V$. The $T$ can be any constant, variable, function, predicate, or expression.

*Substitutions*

A substitution is any finite set (possibly empty) of substitution components, none of the variables of which are same. If $P$ is any set of terms, and the terms of the components of the substitution $\theta$ are all in $P$, we say that $\theta$ is a substitution over $P$. We write the substitution where components are $T_1/V_1, \ldots, T_k/V_k$ as $\theta = \{T_1/V_1, \ldots, T_k/V_k\}$, with the understanding that order of components is immaterial. We will use lowercase Greek letters $\theta$, $\lambda$, $\mu$ denote substitutions.

*Instantiations*

If $E$ is any function string of symbols, and $\theta = \{T_1/V_1, \ldots, T_k/V_k\}$ is any substitution, then the instantiation of $E$ by $\theta$ is the operation of replacing each occurrence of variable $V_i$, $1 \le i \le k$, in $E$ by term $T_i$. The resulting string denoted by $E\theta$ is called an instance of $E$ by $\theta$. That is, if $E$ is the string $E_0 V_{i_1} E_1 \ldots V_{i_n} E_n$, $n \ge 0$, then $E\theta$ is the string $E_0 T_{i_1} E_1 \ldots T_{i_n} E_n$. Here, none of the substrings $E_j$ of $E$ contain occurrences of variables $V_1, \ldots, V_k$ after substitution. Some of $E_j$ are possibly *null*, and each $V_{i_j}$ is an occurrence of one of the variables $V_1, \ldots, V_k$.

### 3.5.1 Composition of Substitutions

If $\theta = \{T_1/V_1, \ldots, T_k/V_k\}$ and $\lambda$ are any two substitutions, then the composition of $\theta$ and $\lambda$ denoted by $\theta\lambda$ is union $\theta' \cup \lambda'$, defined as follows:

The $\theta'$ is set of all components $T_i\lambda/V_i$, $1 \le i \le k$, such that $T_i\lambda$ ($\lambda$ substituted in $\theta$) is different from $V_i$, and $\lambda'$ is set of all components of $\lambda$ whose variables are not among $V_1, \ldots, V_k$.

Within a given scope, once a variable is bound, it may not be given a new binding in future unifications and inferences. If $\theta$ and $\lambda$ are two substitution sets, then the composition of $\theta$ and $\lambda$, i.e., $\theta\lambda$, is obtained by applying $\lambda$ to the elements of $\theta$ and adding the result to $\lambda$.

Following examples illustrate two different scenario of composition of substitutions.

**Example 3.7** Find out the composition of $\{x/y, w/z\}$, $\{v/x\}$, and $\{A/v, f(B)/w\}$.

Let us assume that $\theta = \{x/y, w/z\}$, $\lambda = \{v/x\}$ and $\mu = \{A/v, f(B)/w\}$. Following are the steps:

1. To find the composition $\lambda\mu$, $A$ is is substituted for $v$, and $v$ is then substituted for $x$. Thus, $\lambda\mu = \{A/x, f(B)/w\}$.
2. When result of $\lambda\mu$ is substituted in $\theta$, we get composition $\theta\lambda\mu = \{A/y, f(B)/z\}$.  □

**Example 3.8** Find out the composition of $\theta = \{g(x, y)/z\}$, and $\lambda = \{A/x, B/y, C/w, D/z\}$.

By composition,

$$\theta\lambda = \{g(x, y)/z\} \circ \{A/x, B/y\}$$
$$= \{g(A, B)/z, A/x, B/y, C/w\}$$

The $\{D/z\}$ has not been included in the resultant substitution set, because otherwise, there will be two terms for the variable $z$, one $g(A, B)$ and other $D$. $\square$

One of the important property of substitution is that, if $E$ is any string, and $\sigma = \theta\lambda$, then $E\sigma = E\theta\lambda$. It is straight forward to verify that $\varepsilon\theta = \theta\varepsilon = \theta$ for any substitution $\theta$. Also, composition enjoys the associative property $(\theta\lambda)\mu = \theta(\lambda\mu)$, so we may omit the parentheses in writing multiple compositions of substitutions. The substitutions are not in general commutative; i.e., it is generally not the case that $\theta\lambda = \lambda\theta$, because for this $E\theta\lambda$ has to be equal to $E\lambda\theta$, which is not guaranteed. However, the composition has distributive property.

The point of the composition operation on substitution is that, when $E$ is any string, and $\sigma = \theta\lambda$, the string $E\sigma$ is just the string $E\theta\lambda$, i.e., the instance of $E\theta$ by $\lambda$.

### 3.5.2 Unification

If $E$ is any set of well-formed expressions and $\theta$ is a substitution, then $\theta$ is said to unify $E$, or to be a unifier of $E$, if $E\theta$ is a singleton. Any set of well-formed expressions which has a unifier is said to be unifiable [6].

In proving theorems using quantified variables, it is often necessary to "match" certain subexpressions. For example, to apply the combination of modus ponens and universal instantiation (Eq. 3.5) to produce "*mortal*(*socrates*)", it was necessary to find substitution {*socrtaes*/*x*} for *x* that makes *man(x)* and *man(socrates)* equal (singleton).

Unification *algorithm* determines the substitutions needed to make two predicate expressions match. For this, all the necessary condition is that variables must be universally quantified. Unless the variables in an expression are existentially quantified, they are assumed to be universally quantified. This criteria allows us full freedom choosing the substitutions. The existentially quantified variables can be eliminated by substituting them with constants or with Skolem functions that makes the sentence true. For example, in sentence,

$$\exists x \, mother(x, jill),$$

we can replace *x* with a constant designating jill's mother, *susan*, to get:

$$mother(susan, jill);$$

and write unifier as {*susan*/*x*}.

For perform unification, a variable can be replaced by any term, including other variable or function expressions of arbitrary complexity. This also includes function expressions that themselves contain variables. For example, the function expression, *mother(joe)*, may be substituted for *x* in *human(x)* to get *human(mother(joe))*.

**Example 3.9** Find out the substitution instances for *foo*(*x, a, zoo*(*y*)), given the similar predicates with literal arguments.

1. *foo*(*fred, a, zoo*(*z*)), where *fred* is substituted for *x* and *z* for *y*, i.e., $\lambda_1 = \{fred/x, z/y\}$. Thus,

$$foo(fred, a, zoo(z)) = foo(x, a, zoo(y))\lambda_1.$$

2. *foo*(*w, a, zoo*(*jack*)), where $\lambda_2 = \{w/x, jack/y\}$; hence

$$foo(w, a, zoo(jack)) = foo(x, a, zoo(y))\lambda_2.$$

3. *foo*(*z, a, zoo*(*moo*(*z*))), where $\lambda_3 = \{z/x, moo(z)/y\}$, hence;

$$foo(z, a, zoo(moo(z))) = foo(x, a, zoo(y))\lambda_3.$$

We use the notation $x/y$ to indicate that *x* is substituted for the variable *y*; we also call this as *bindings*, so *y* is bound to *x*. A variable cannot be unified with a term containing that variable. So $p(x)$ cannot be substituted for *x*, because this would create an infinite regression: $p(p(p(...x)...)$.

## 3.6  Resolution Principle

The resolution rule can be traced back to 1960, which was introduced by Davis and Putnam. However, this algorithm required all around ground instances for the given formula, which resulted to a combinatorial explosion. However, the source of combinatorial explosion was eliminated in 1965, when J. A. Robinson introduced an algorithm for unification. The unification allowed the instantiation of the formula during the proof "in demand", just as per the need through the newly introduced most general unifier (mgu) algorithm [8].

The resolution method for (propositional) logic due to J. A. Robinson (1965) is *sound and complete*, and a well-known procedure for checking the unsatisfiability of a set of clauses. The resolution is mathematical oriented rather than human oriented. It is quite powerful both in the psychological sense that it condones single inferences which are often beyond the ability of human to grasp, and in theoretical sense that it alone, as sole inference principle, forms a complete system of FOPL. Because of only one inference type, it allows to avoid the combinatorial obstacles to efficiency.

Let us refresh us with the terminology we discussed in the beginning of this chapter. We will designate a *literal* by symbol *L*, which is either a propositional

**Fig. 3.2** DAG for theorem proving

$$\{P, Q\} \quad \{P, \neg Q\} \quad \{\neg P, Q\} \quad \{\neg P, \neg Q\}$$

$$\{P\} \quad \{Q\}$$

$$\{\neg P\}$$

$$\{\}$$
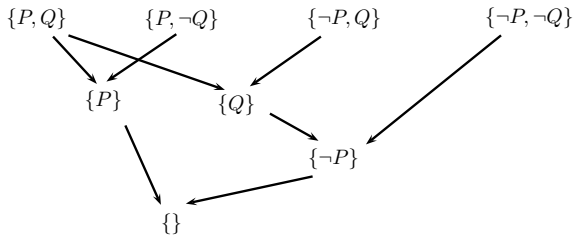
symbol, $P$, or the negation, $\neg P$. A finite set of literals $\{L_1, \ldots, L_k\}$ is a *clause*, which is interpreted as a disjunction of literals, $L_1 \vee \cdots \vee L_k$. With $k = 0$, it is an empty clause, denoted as []. A conjunction of a set of clauses $\Gamma = \{C_1, \ldots, C_n\}$ is interpreted as $C_1 \wedge \cdots \wedge C_n$. In short, we write set of clauses as $\Gamma = C_1, \ldots, C_n$.

The resolution method is a procedure for determining whether a set of clauses $\Gamma$, is *unsatisfiable*. To find out latter, the resolution method first builds a certain kind of labeled *DAG* (Directed Acyclic Graph) whose terminal nodes are labeled with clauses in $\Gamma$ and the interior nodes are labeled as per the resolution rule [5].

Consider that there are any two clauses $C = A \cup \{P\}$ and $D = B \cup \{\neg P\}$ (where $P$ is a propositional letter, $P \notin A$ and $\neg P \notin B$). The resolvent of $C$ and $D$ is the clause $R = A \cup B$ obtained by canceling out $P$ and $\neg P$, and making disjunction of the remaining part in each clause. A resolution DAG for $\Gamma$ is a DAG whose terminal nodes are labeled with clauses from $\Gamma$ and such that every interior node $n$ has exactly two predecessors, $n_1$ and $n_2$ so that $n$ is labeled with the resolvent of the clauses labeling $n_1$ and $n_2$. A resolution refutation for $\Gamma$ is a resolution DAG with a single root whose label is the empty clause. Note that the root will come at the end, as shown in Fig. 3.2.

**Example 3.10** Resolution refutation.

A resolution refutation for the set of clauses

$$\Gamma = \{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}.$$

is shown in Fig. 3.2. □

A recursive algorithm can be given for construction of a resolution DAG using any number of clauses, and to prove its correctness. That means, if the input set of clauses is unsatisfiable, the output resolution DAG is a resolution refutation. This confirms the completeness of propositional resolution constructively.

### 3.6.1  Theorem Proving Formalism

It is a syntactic inference procedure, when applied to clauses, determines, if the satisfied set is unsatisfiable. Proof is similar to proof by *contradiction* and deduce [] (i.e., null). If for example, we have set of clauses (axioms) $C_1, C_2, \ldots, C_n$, and we want to deduce $D$, i.e., $D$ is logical consequence of of $C_1, C_2, \ldots, C_n$. For this we add $\neg D$ to the set $\{C_1, C_2, \ldots, C_n\}$, then we show that set is unsatisfiable by deducing contradiction [7].

The process of deduction using resolution is given in Algorithm 3.1. Given two clauses $C_1, C_2$ with no variables in common, and if $l_1$ is a literal in $C_1$ and its complement literal $l_2$ is a literal in $C_2$, then $l_1, l_2$ can be dropped and disjunction $C$ is obtained from the remaining part of $C_1, C_2$. The $C$ is called resolvent of $C_1, C_2$.

Let $C_1 = \neg P \vee Q$, and $C_2 = \neg Q \vee R$, then following can be deduced through resolution,

$$\frac{P \Rightarrow Q, Q \Rightarrow R}{P \Rightarrow R} \qquad (3.2)$$

equivalently,

$$\frac{(\neg P \vee Q), (\neg Q \vee R)}{\therefore (\neg P \vee R)}. \qquad (3.3)$$

It can be easily verified that $(\neg P \vee Q) \wedge (\neg Q \vee R) \models (\neg P \vee R)$, hence $(\neg P \vee Q) \wedge (\neg Q \vee R) \Rightarrow (\neg P \vee R)$ is a valid statement. Thus, $\neg P \vee R$ is inference or the resolvent. Arriving to a proof by above is called proof by *refutation.*

Resolution says that if there are axioms of the form $\neg P \vee Q$ and there is another axiom of the form $\neg Q \vee R$, then $\neg P \vee R$ logically follows; called the *resolvent.* Let us see why it is so? When $\neg P \vee Q$ is True, then either $\neg P$ is True or $Q$ is True. For other expression, when $\neg Q \vee R$ is True, then either $\neg Q$ is True or $R$ is True. Then we can say that $\neg P \vee R$ is certainly True. This can be generalized to two expressions, when we have any number of expressions, but two must be of opposite signs.

### 3.6.2  Proof by Resolution

To prove a theorem, one obvious strategy is to search forward from the axioms, using sound rules of inference. We try to prove a theorem by refutation. It requires to show that negation of a theorem cannot be True. The steps for a proof by resolution are:

1. Assume that negation of the theorem is True.
2. Try to show that axioms and assumed negation of theorem, together are True, which cannot be True.
3. Conclude that above leads to contradiction.
4. Conclude that theorem is True because its negation cannot be True.

To apply the resolution rule,

1. Find two sentences that contain the same literal, one in its positive form and one in its negative form, like,

$$CNF : summer \lor winter, \neg winter \lor cold,$$

2. use the *resolution* rule to eliminate the complement literals from both sentences to get,

$$CNF : summer \lor cold.$$

The Algorithm 3.1 is an algorithm for theorem proving through resolution-refutation, where $\alpha$ is the theorem to be proved, and $\beta$ is set of axioms, both of these are input to the algorithm. All the inputs to algorithm are in the clause form. The algorithm returns "true" if the theorem is true, else returns "False".

---

**Algorithm 3.1** Algorithm-Resolve(Input: $\alpha$, $\beta$)

---

1: $\Gamma = \beta \cup \{\neg\alpha\}$
2: **while** there is a resolvable pair of clauses $C_i, C_j \in \Gamma$ **do**
3:     $C = resolve(C_i, C_j)$
4:     **if** $C = NIL$ **then**
5:         return "Theorem $\alpha$ is true"
6:     **end if**
7:     $\Gamma = \Gamma \cup \{C\}$
8: **end while**
9: Report that theorem is False

---

## 3.7 Complexity of Resolution Proof

The question is, how you can be so clever to pickup the right clauses to resolve? The answer is that you take advantage of two ideas:

1. You can be sure that every resolution involves the *negated theorem*, directly or indirectly.
2. You know where you are and where you are going, hence you can compute the difference to help you proceed with your intuition for selection of clauses.

Consider there are total $n$ clauses, $c_1 \ldots c_n$. We can try to match $c_1$ with $c_2 \ldots c_n$, and in next level $c_2$ is matched with $c_3 \ldots c_n$, and so on. This results to breadth first search (BFS). Consider that resolvents generated due to this matching are $c'_1 \ldots c'_m$. Next all the newly generated clauses are matched with the original, and then they are merged into the original. This process is repeated until contradiction is reached, showing that theorem is proved. Since, the entire set of clauses are compared, the

proof is bound to result, if it exists, at all. This gives completeness to the resolution proof.

The other alternative is, nodes which are farther and farther away are matched before those which are closer to the root. The $c_1$ is matched with first child $c_2$ out of $c_2 \ldots c_n$. Then $c_2$ is matched with its first child generated, and so on, resulting to the search process called DFS (depth first search).

However, the above both are brute-force algorithms, and are complex. The other methods are heuristic based. In fact, there is difficulty to express your concepts required in pure logic. One of the approaches is to use the clauses having smallest number of literals. Another, to use negated clauses.

The resolution search strategies are subject to the exponential-explosion problem. Due to this, those proofs which require long *chains of inferences*, will be exponentially expensive in time.

All resolution search strategies are subject to a version of *halting problem*, for search is not guaranteed to terminate unless there actually is a proof. In fact, all complete proof procedures for the first order predicate calculus are subject to halting problem. Complete proof procedures are said to be *semi-decidable*, because they are generated to tell you whether an expression is a theorem, only if the expression is indeed a theorem.

Theorem proving is suitable for certain problems, but not for all problems, due to the following reasons:

1. Complete theorem proving requires search, and search is inherently exponential,
2. Theorem provers may not help you to solve practical problems, even if they do their work instantaneously.

## 3.8 Interpretation and Inferences

A FOPL statement is made of predicates, arguments (constants or variables), functions, operators, and quantifiers. Interpretation is process of assignment of truth values (True/False) to subexpressions and atomic expressions, and computing the resultant value of any expression/statement. A statement or expression in predicate logic is also called *wwf* (well formed formula).

Consider the interpretation of predicate formula:

$$\forall x[bird(x) \rightarrow flies(x)]. \tag{3.4}$$

To find out the *satisfiability* of the formula (3.4), we need to substitute (*instantiate*) a value for $x$ (an instance of $x$) and check if $flies(x)$ is true. Until, that $x$ is found, it may require instantiation with large number of values. Similarly, to check if the Eq. (3.4) is valid, it may require infinitely large number of values in the domain of

*x* to be verified. If any one of that makes the formula false, the formula is not valid. Thus, checking of satisfiability as well as validity of a formula is predicate logic are complex process. The approach of *truth table* and *tableau* method we discussed in the previous chapter are applicable here also.

Given a predicate sentences of *m* number of predicates each having one argument, and domain size of all the arguments is *n*, in the worst case it will require total $n^m$ substitutions to test for satisfiability, as well as for validity checking. However, a sentence of *m* propositions will require in the worst case only $2^m$ substitutions. Hence, satisfiability checking in predicate sentences is much more complex than that in proposition logic. It equally applies with expressions having existential quantifiers, like, $\exists x[bird(x) \rightarrow flies(x)]$.

Thus, it is only the proof methods, using which *logical deductions* can be carried out in realistic times.

**Example 3.11**  Given, "All men are mortal" and "Socrates is man", infer using predicate logic, that "Socrates is mortal".

The above statement can be written in predicate logic as:

$$\forall x[man(x) \Rightarrow mortal(x)],$$
$$man(socrates). \tag{3.5}$$

Using a rule called *universal instantiation*, a variable can be instantiated by a constant and universal quantifier can be dropped. Hence, from (3.5) we have,

$$man(socrates) \Rightarrow mortal(socrates),$$
$$man(socrates). \tag{3.6}$$

Using the rule of *modus ponens* on (3.6) we deduce "*mortal(socrates)*". It is also *logical consequence*. If $\Gamma = \{[man(socrates) \Rightarrow mortal(socrates)] \wedge man (socrates)\}$, and $\alpha = mortal(socrates)$, then we can say that $\Gamma \vdash \alpha$.

The set of formulas $\Gamma$ is called knowledge base. To find out the result for the query "Who is man?", we must give the query

$$?man(X).$$

in Prolog (to be discussed later), which will match (called *unify* or *substitute*) $man(X)$ with $man(socrates)$ with a unification set, say, $\theta = \{socrates/X\}$. The substitution which returns $man(socrates)$ is represented by $man(X)\theta$. □

**Example 3.12**  Prolog Program.

The sentence in Eq. (3.5) will appear in prolog as,

$$mortal(socrates) :- man(socrates).$$
$$man(socrates).$$

Here, the sign ':-'is read as 'if'. The subexpression before the sign ':-'is called 'head'or *procedure name* and the part after ':-'is called *body* of the *rule*. The sentence (3.6) can also be written in a *clause form* (to be precise, in *Horn clause* form) as,

$$mortal(socrates) \lor \neg man(socrates).$$
$$man(socrates). \tag{3.7}$$

## *3.8.1  Herbrand's Universe*

Defining an operational semantics for a programming language is nothing but to define an implementation independent interpreter for it. In case of predicate logic, the proof procedure itself behaves like an interpreter. The Herbrand's Universe and Herbrand's base play an important role in interpretation of predicate language. In the following, we define the Herbrand's Universe and Herbrand's Base.

**Definition 3.1**  (*Herbrand's Universe*) In a predicate logic program, a Herbrand Universe **H**, is a set of ground terms that use only function symbols and constants.

**Definition 3.2**  (*Herbrand's Base*) A set of atomic formulas formed by predicate symbols in a program, is called Herbrand's base. The additional condition is that, arguments of these predicate symbols are in the Herbrand Universe.

For a predicate program, the Herbrand universe and Herbrand base are countably infinite if the predicate program contains a function symbol of positive arity. If the arity of function symbols is zero, then both the haerbrand's universe and base are finite [3].

In special cases, when resolution proof is used on FOPL, it reduces the expressions to propositional form. If the set of clauses is **A**, its Harbrand's universe is set of all the ground terms formed using only the function symbols and constants in **A**. For example, if **A** has constants $a$, $b$, and a unary function symbol $f$, then the Herbrand universe is the infinite set:

$$\{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), \dots \}.$$

The Herbrand's base of **A** is the set of all ground clauses $c\theta$ where $c \in \mathbf{A}$ and $\theta$ is a substitution that assigns the variables in $c$ to terms in the Herbrand's universe.

Horn clauses

The *Horn* clauses are *terms* without variables, these are constructed using constants and function symbols that occur in the set of clauses **A**. These terms form the *data structures*, which are manipulated by the program built-in in the clauses **A**. The collection of all such *terms*, determined by **A**, is called *Herbrand universe*. Every *n*-ary predicate symbol *P* occurring in **A** denotes an *n*-ary relation over the Herbrand universe of **A**. We call the *n*-tuples which belong to such relations as *input-output* tuples and the relations themselves as *input-output relations*.

In an inference system, the operational semantics determine a unique denotation for a formula *P* such that the *n*-tuple $(t_1, \ldots, t_m)$ belongs to the denotation of $P \in \mathbf{A}$, iff $\mathbf{A} \vdash P(t_1, \ldots, t_n)$. That is,

$$D(P) = \{(t_1, \ldots, t_n) : \mathbf{A} \vdash P(t_1, \ldots, t_n)\}. \tag{3.8}$$

We first pick an arbitrary constant, say, *a*, and then construct the variable-free terms. Formally, $D(P)$ is inductively defined as follows:

1. All constants occurring in *P* belong to $D(P)$; if no constant occurs in *P*, then $a \in D(P)$.
2. For every *n*-ary functional symbol *p* occurring in *P*, if $t_1, t_2, \ldots, t_n \in D(P)$ then $p(t_1, t_2, \ldots, t_n) \in D(P)$.

Here $X \vdash Y$ means $X$ derives $Y$. For resolutions systems, if $X \vdash Y$ then there exists a refutation of the sentence in clausal form with atoms as $X$ and $\overline{Y}$.

In goal oriented inference systems, the procedure calls are replaced by procedure bodies. Such inference systems correspond to standard notion of operational semantics. In theoretical sense, any inference system, based on predicate logic represents an abstract machine, that generates only those derivatives which are determined by this inference system.

For predicate logic, the corresponding programs compute the relations represented by predicate symbols in the set of clauses **A**. These relations may be in the form of predicates or functions. However, these function or predicate symbols are not treated as functions computed by the program, but they result into some *data structures*, and these data structures are actually the input and output objects of the relations / functions being computed.

**Definition 3.3** Herbrand's Structure.

Let *P* be a formula in Skolem form (when a constant, say, *a* is substituted for *x* in $\exists x \, p(f(x), b)$ results to Skolem form $p(f(a), b)$)). A structure $\mathscr{A} = (U_{\mathscr{A}}, I_{\mathscr{A}})$ suitable for *p* is a *Herbrand structure* for *P* if it satisfies the following conditions:

1. $U_{\mathscr{A}} = D(P)$, and
2. for every *n*-ry function symbol *p* occurring in *P* and every $t_1, t_2, \ldots, t_n \in D(P)$ : $f^{\mathscr{A}}(t_1, t_2, \ldots, t_n) = f(t_1, t_2, \ldots, t_n)$.

In above, $\mathscr{A} = (U_\mathscr{A}, I_\mathscr{A})$ is model with $U_\mathscr{A}$ as formula and $I_\mathscr{A}$ as its interpretation.                                                                 □

**Definition 3.4** (*General logic program*) It is a finite set of general rules, with both positive and negative subgoals.

A general logic program comprises rules and facts. A general rule's format is: its *head*, or *conclusion* is to the left of the symbol "←," (read "if"), and its subgoals (called body) is right of the symbol "←". Following is an example of rule, where $p(X)$ is head, $q(X)$ is positive subgoal, and $r(X)$ is a negative subgoal.

$$p(X) \leftarrow q(X), \; \neg \, r(X). \tag{3.9}$$

This rule may be read as "$p(X)$ *if* $q(X)$ *and not* $r(X)$." A *Horn rule* has no negative subgoals, and a *Horn logic program* is made of only Horn rules.

We will follow the conventions of *Prolog* for naming the objects: the logical variables begin with an uppercase letter, while the constants, functions, and predicates begin with a lowercase letter. For both the predicate and its relation, we will use the same symbol, for example $p$.

Followings may be the arguments of a predicate:

1. a constant/variable is a term;
2. a function symbol with terms as arguments, is a term.

The terms may be viewed as data structures of the program, with function symbols serving as record names. Often a constant is treated as a function symbol of arity zero.

Following are the definitions of some important terms.

**Definition 3.5** (*Herbrand Instantiation*) Herbrand instantiation of a general logic program is the set of rules obtained by substituting terms in the Herbrand universe for variables, in every possible way.

**Definition 3.6** An **instantiated rule** is one only, whereas "uninstantiated" logic programs are assumed to be a finite set of rules, and instantiated logic programs may be infinite in number.

**Definition 3.7** (*Complement of a set*) For a set of literals $L$ its complement is a set formed by complementing of each literal in $L$, represented by $\neg L$.

Further,

– $p$ is said to be *inconsistent* with $L$ if $p \in \neg L$,
– Sets of literals $R$ and $L$ are *inconsistent* if at least one literal in $R$ is inconsistent with $L$, i.e., when $R \cap \neg L \neq \phi$,
– A set of literal is *inconsistent* if it is inconsistent with itself; otherwise it is *consistent*.

### 3.8.2 Herbrand's Theorem

Herbrands theorem is a fundamental theorem based on mathematical logic, that permits a certain type in reduction from FOPL to propositional logic [4].

In its simplest form, the Herbrand's theorem states that a formula of first-order predicate logic $\exists x\, A$, where $A$ is quantifier free, is provable if and only if there exist ground terms $M_1, \ldots, M_n$ such that,

$$\models A[x := M_1] \vee \cdots \vee A[x := M_n]. \tag{3.10}$$

When using the classical formulation, the Herbrand's theorem relates the validity of a first-order formula in *Skolem prenex form*[1] to the validity of one of its Herbrand extensions. That means, the formula $\forall x_1 \ldots \forall x_n \psi(x_1 \ldots, x_n)$ is valid if, and only if, $\bigwedge_i^m \psi(t_{i,1}, \ldots, t_{i,n})$ is valid for some $m \geq 1$ and some collection of ground Herbrand terms $t_{i,j}$.

Since it is possible that every classical first-order formula can be reduced to this Skolem prenex form through the Skolemization while preserving its satisfiability, the Herbrand's theorem provides a way to reduce the question of validity of first-order formulas to propositional logic formula.

However, the required Herbrand's extension and the terms $t_{i,j}$ cannot be computed recursively (for otherwise first-order logic would be decidable), this result is highly useful for the automated reasoning as it gives a way to some highly efficient proof methods such as *resolution* and the *resolution refutation*.

**Theorem 3.1** *A closed formula $F$ in Skolem form is satisfiable if and only if it has a Herbrand model.*

**Proof** If the formula has a Herbrand model then it is satisfiable. For the other direction let $\mathscr{A} = (U_{\mathscr{A}}, I_{\mathscr{A}})$ be an arbitrary model of $F$. We define a Herbrand structure $\mathscr{B} = (U_{\mathscr{B}}, I_{\mathscr{B}})$ as follows:

Universe: $U_{\mathscr{B}} = D(F)$

Functional Symbols: $f^{\mathscr{B}}(t_1, t_2, \ldots, t_n) = f(t_1, t_2, \ldots, t_n)$

Predicate Symbols: $(t_1, \ldots, t_n) \in P^{\mathscr{B}}$ iff $\mathscr{A}(t_1), \ldots, \mathscr{A}(t_n) \in P^{\mathscr{A}}$.

*Claim:* $\mathscr{B}$ is also a model of $F$.

We prove a stronger assertion: For every closed form $G$ in Skolem form such that $G^*$ only contains atomic formulas of $F^*$ : *if* $\mathscr{A} \models G$ *then* $\mathscr{B} \models G$.

By induction on the number $n$ of universal quantifiers of $G$.

*Basis* ($n = 0$). Then $G$ has no quantifiers at all.

It follows $\mathscr{A}(G) = \mathscr{B}(G)$, this proves the theorem. $\qquad \square$

To perform reasoning with the Herbrand base, the unifiers are not required, and we have a *sound* and *complete* reasoning procedure, which is guaranteed to terminate. The idea used in this approach is: Herbrand's base will typically be an infinite set of propositional clauses, but it will be finite when Herbrand's universe is finite (there

---

[1] A string of quantifiers followed by a quantifier-free part, e.g., $\forall x_1 \ldots \forall x_n \psi(x_1 \ldots, x_n)$.

is no function symbols and only finitely many constants appear in it). Sometimes we can keep the universe finite by considering the type of the arguments (say $t$) and values of functions ($f$), and include a term like $f(t)$ in the universe only if the type of $t$ is appropriate for the function $f$. For example, $f(t)$ may be, *birthday*(*john*), which produces a date.

### 3.8.3  The Procedural Interpretation

It is easy to procedurally interpret the sets of clauses, say, **A**, which contain at most one positive literal per clause. However, along with this any number of negative literals can also exist. Such sets of clauses are called *Horn sentences* or *Horn Clauses* or simply clauses. We distinguish three kinds of *Horn clauses* [3].

1. '[]'the *empty clause*, containing no literals and denoting the truth value *false*, is interpreted as a *halt statement*.
2. $\bar{B}_1 \vee \cdots \vee \bar{B}_n$, a clause consisting of no positive literals and $n \geq 1$ negative literals, is interpreted as a *goal statement*. Note that goal statement is negated and added into the knowledge base to obtain the proof through *resolution refutation*.
3. $A \vee \bar{B}_1 \vee \cdots \vee \bar{B}_n$, a clause consisting of exactly one positive literal and $n \geq 0$ negative literals is interpreted as a *procedure declaration* (i.e., rule in Prolog program). The positive literal $A$ is the *procedure name* and the collective negative literals are the *procedure body*. Each negative literal $B_i$, in the procedure body is interpreted as a *procedure call*. When $n = 0$ the procedure declaration has an empty body and interpreted as an unqualified assertion of fact.

In the procedural interpretation, a set of procedure declarations is a program. Computation is initiated by an *initial goal* statement, which proceeds by using declared procedures to derive new goal statements (*subgoals*) $B_i$s from old goal statements, and terminates on the derivation of the halt statement. Such derivation of goal statements is accomplished by *resolution*, which is interpreted as *procedural invocation*.

Consider that, a selected procedure call $\bar{A}_1$ inside the body of a goal statement as,

$$\bar{A}_1 \vee \cdots \vee \bar{A}_{i-1} \vee \bar{A}_i \vee \bar{A}_{i+1} \vee \cdots \vee \bar{A}_n \tag{3.11}$$

and a procedure declaration is given as,

$$A' \vee \bar{B}_1 \vee \cdots \vee \bar{B}_m, m \geq 0. \tag{3.12}$$

Suppose, the name of procedure $A'$ matches with the procedure call $A_i$, i,e., some substitution $\theta$ of terms for variables makes $A_i$ and $A'$ identical. In such a case, the resolution derives a new goal statement by disjunction formulas (3.11) and (3.12) as given below, subject to substitution $\theta$.

$$(\bar{A}_1 \vee \cdots \vee \bar{A}_{i-1} \vee \bar{B}_1 \vee \cdots \vee \bar{B}_m \vee \bar{A}_{i+1} \vee \cdots \vee \bar{A}_n)\theta. \qquad (3.13)$$

In general, any *derivation* can be regarded as a computation, and any *refutation* (i.e. derivation of []) can be regarded as a successfully terminating computation. It is to be noted that, only goal oriented resolution derivations correspond to the standard notion of computation.

Thus, a goal-oriented derivation, from an initial set of Horn clauses **A** and from an initial goal statement (computation) $C_1 \in \mathbf{A}$, is a sequence of goal statements $C_1, \ldots, C_n$. So that each $C_i$ contains a single selected procedure call and $C_{i+1}$, obtained from $C_i$ by procedure invocation relative to the selected procedure call in $C_i$, using a procedure declaration in **A**.

For the implementation of above, one method is *model elimination*. Using this, the selection of procedure calls is governed by the last-in/first-out rule: a goal statement is treated as a stack of procedure calls. The selected procedure call must be at the top of the stack. The new procedure calls which by procedure invocation replace the selected procedure call are inserted at the top of the stack. This would result to a *depth-first search* procedure.

The Predicate logic is a *nondeterministic* programming language. Consequently, given a single goal statement, several procedure declarations can have a name which matches the selected procedure call. Each declaration gives rise to a new subgoal statement. A *proof procedure* which sequences the generation of derivations in the search for a refutation behaves as an *interpreter* for the program incorporated in the initial set of clauses.

The following example explains how to use procedural interpretation to append two given lists.

**Example 3.13**   Appending two lists [3].

Let a term *cons*$(x, y)$ is interpreted as a list whose first element, the *head*, is $x$ and whose *tail* $y$ is the rest of the list. The constant *nil* denotes the empty list. The terms $u, x, y$, and $z$ are variables. The predicate *append(x,y,z)* denotes the relationship: $z$ is obtained by appending $y$ to $x$.

The following two clauses constitute a program for appending two lists.

$$append(nil, x, x). \qquad (3.14)$$

$$append(cons(x, y), z, cons(x, u)) \vee \overline{append}(y, z, u). \qquad (3.15)$$

The clause in statement (3.14) represents halt statement. In (3.15) there is a positive literal for procedure name, and negative literal(s) for the procedure body, both together it is procedure declaration. The positive literal means, if *cons*$(x, y)$ is appended with $z$, it results to $x$ appended with $u$ such that $u$ is, $y$ appended with $z$. The later part is indicated by the complementary (negative) term. Note that clausal expression (3.15) is logically equivalent to the expression $append(y, z, u) \rightarrow append(cons(x, y), z, cons(x, u))$.

Suppose it is required to compute the result of appending list $cons(b, nil)$ to the list $cons(a, nil)$. Therefore, the goal statement is,

$$append(cons(a, nil), cons(b, nil), v), \qquad (3.16)$$

where $v$ (a variable) and $a, b$ (constants), are the "atoms" of the lists. To prove using resolution, we add the negation of the goal,

$$\overline{append}(cons(a, nil), cons(b, nil), v), \qquad (3.17)$$

into the set of clauses. The program is activated by this *goal statement* to carry out the append operation. With this goal statement the program is deterministic, because only one choice is available for matching. The following computation follows with a goal directed theorem prover as interpreter: The goal statement,

$$C_1 = \overline{append}(cons(a, nil), cons(b, nil), v). \qquad (3.18)$$

matches with the clause statement (3.15) with matchings: $x = a, y = nil, z = cons(b, nil)$. Also, $v = cons(x, u) = cons(a, u)$, i.e., there exists a unifier $\theta_1 = \{cons(a, w)/v\}$. The variable $u$ has been renamed as $w$. On unifying clauses (3.18) and (3.15), the next computation $C_2$ is:

$$C_2 = \overline{append}(nil, cons(b, nil), w)\theta_1. \qquad (3.19)$$

Keeping $\theta_1$ accompanying the predicate in above is for the purpose that if $C_2$ is to be unified with some other predicate, the matching of the two shall be subject to the same unifier $\theta_1$.

As next matching, $C_2$ can be unified with (3.14) using a new unifier $\theta_2 = \{cons(b, nil)/w\}$ to get next computation,

$$C_3 = []\theta_2. \qquad (3.20)$$

The result of the computation is value of $v$ in the substitution, i.e.,

$$\begin{aligned} v &= cons(a, u) \\ &= cons(a, w) \\ &= cons(a, cons(b, nil)). \end{aligned}$$

The above result is equal to goal: $append(cons(a, nil), cons(b, nil), v)$.          □

**Example 3.14**  Theorem proving using resolution-refutation.

Following axioms are about the observed block relationship shown in Fig. 3.3, which are already in clausal form.

**Fig. 3.3**  Objects on table
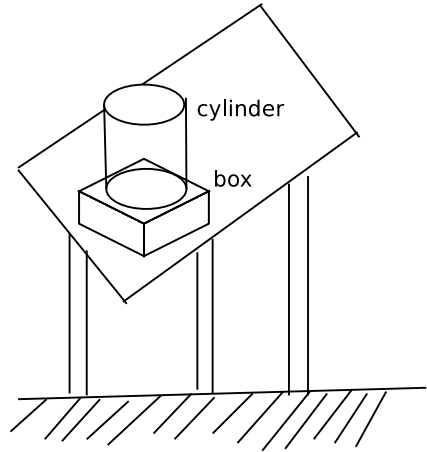


$$on(cylinder, box).$$

$$on(box, table).$$

It is required to be shown that object *cylinder* is above table, i.e., *above*(*cylinder*, *table*), given the the following rules:

$\forall x \forall y[on(x, y) \rightarrow above(x, y)]$, and
$\forall x \forall y \forall z[above(x, y) \wedge above(y, z) \rightarrow above(x, z)]$.

After we have gone through the procedure for conversion to clausal form, the above axioms are transformed into clause forms.

$\neg on(u, v) \vee above(u, v)$.
$\neg above(x, y) \vee \neg above(y, z) \vee above(x, z)$.

The expression to be proved is "above(cylinder, table)"; its negation is $\neg above(cylinder, table)$. Let us list all the clauses systematically.

(1)  $\neg on(u, v) \vee above(u, v)$.
(2)  $\neg above(x, y) \vee \neg above(y, z) \vee above(x, z)$.
(3)  $on(cylinder, box)$.
(4)  $on(box, table)$.
(5)  $\neg above(cylinder, table)$.

Now, we manually run the Algorithm 3.1 on the clauses (1)–(5), as well as those which would created new, to unify them according to unification Algorithm 3.2, until we reach to a null resolvent.

First we resolve clauses (2) and (5) and bind $x$ to '*cylinder*'and $z$ to '*table*'. Applying the resolution, we get resolvent (6). Unifier for this is {*cylinder*/$x$, *table*/$z$}.

(2)  ¬*above*(*cylinder*, *y*) ∨ ¬*above*(*y*, *table*) ∨ *above*(*cylinder*, *table*).
(5)  ¬*above*(*cylinder*, *table*).
(6)  ¬*above*(*cylinder*, *y*) ∨ ¬*above*(*y*, *table*).

Next, resolve clauses (1) with (6), binding *u* with *y* and *v* with '*table*', we get (7). Unifier for this is {*y*/*u*, *table*/*v*}.

(1)  ¬*on*(*y*, *table*) ∨ *above*(*y*, *table*).
(6)  ¬*above*(*cylinder*, *y*) ∨ ¬*above*(*y*, *table*).
(7)  ¬*on*(*y*, *table*) ∨ ¬*above*(*cylinder*, *y*).

We use (1) again with (7) with *u* bound to *cylinder* and *v* to *y*. Unifier for this is {*cylinder*/*u*, *y*/*v*}. On resolving we get (8).

(1)  ¬*on*(*cylinder*, *y*) ∨ *above*(*cylinder*, *y*).
(7)  ¬*on*(*y*, *table*) ∨ ¬*above*(*cylinder*, *y*).
(8)  ¬*on*(*cylinder*, *y*) ∨ ¬*on*(*y*, *table*).

Next, use clause (3) and (8), binding *y* to *box*, with unifier {*box*/*y*}. We get (8) as resolvent.

(3)  *on*(*cylinder*, *box*).
(8)  ¬*on*(*cylinder*, *box*) ∨ ¬*on*(*box*, *table*).
(9)  ¬*on*(*box*, *table*).

Finally, the clauses (4) and (9) are resolved to get '[]':

 (4)  *on*(*box*, *table*).
 (9)  ¬*on*(*box*, *table*).
(10)  [].

  Since we have arrived at the contradiction, it shows that negation of the theorem: ¬*above*(*cylinder*, *table*) must be False. Hence the theorem *above*(*cylinder*, *table*) must be True.                                                                                    □

## 3.9  Most General Unifiers

The simple approach to avoid needless search in a first-order derivation is to keep the search procedure as general as possible. Consider, for example the following two clauses, each as a literal only.

$$c_1 = p(g(x), f(x), z),$$

and

$$c_2 = \neg p(y, f(w), a).$$

They are unified by the substitution $\theta_1$,

$$\theta_1 = \{b/x, g(b)/y, a/z, b/w\},$$

and also by $\theta_2$,

$$\theta_2 = \{f(z)/x, g(f(z))/y, a/z, f(z)/w\}.$$

Note that a constant, or variable, or a function substitutes for a variable, and not the other way.

We may very well be able to derive the empty clause using $c_1$, $c_2$ with substitution of $\theta_1$, followed with application of resolution. But if we cannot, we will need to consider other substitutions like $\theta_2$.

The trouble is that both of these substitutions are overly specific. We can see that any unifier must give $w$ the same value as $x$, and to $y$ the same as $g(x)$, but we do not need to commit yet to a value for $x$. The substitution,

$$\theta_3 = \{g(x)/y, a/z, x/w\}$$

unifies the two literals without making an arbitrary choice that might preclude a path to the empty clause. The $\theta_3$ is a *most general unifier* (mgu).

More precisely, an mgu $\theta$ of literals $\rho_1$ and $\rho_2$ is a unifier that has the property that for any other unifier $\theta'$, there is a further substitution $\theta^*$ such that $\theta' = \theta\theta^*$. So starting with $\theta$, you can always get to any other unifier by applying additional substitutions. For example, given $\theta_3$, we can get to $\theta_1$ by further applying $\lambda = \{b/x\}$ so that $\theta_1 = \theta_3\lambda$. And, we can get to $\theta_2$ by $\mu = \{f(z)/x\}$ so that $\theta_2 = \theta_3\mu$. Note that an mgu need not be unique. For example, $\theta_4 = \{g(w)/y, a/z, w/x\}$ is also an *mgu* for $c_1$ and $c_2$.

The key fact about mgus is that we can limits the resolution rule to mgus without loss of completeness. This helps immensely in the search since it dramatically reduces the number of resolvents that can be inferred from these two input clauses.

**Example 3.15** Given a unifier, obtain a more general unifier.

Suppose you have two expressions $p(x)$ an $p(y)$. One way to unify these is to substitute any constant expression for $x$ and $y$: $S = \{fred/x, fred/y\}$. But this is not the most general unifier, because if we substitute any variable for $x$ and $y$, we get a more general unifier: $G = \{z/x, z/y\}$. The first unifier is a valid unifier, but it would lessen the generality of inferences that we might want to make.

$$\text{Let } E = \{p(x), p(y)\},$$
$$S = \{fred/x, fred/y\},$$
$$G = \{z/x, z/y\}.$$
$$\text{Now let } S' = \{fred/z\}$$

$$\text{Then } ES = \{p(\mathit{fred}), p(\mathit{fred})\}$$
$$\text{and } GS^{'} = \{\mathit{fred}/x, \mathit{fred}/y\}$$
$$\text{and therefore } EGS^{'} = \{p(\mathit{fred}), p(\mathit{fred})\} = ES.$$

So, given a unifier, you can always create a more general unifier. When both of these unifiers are composed and instantiate the original expression $E$, you get the same instance as it was obtained with the earlier unifier.

### 3.9.1  Lifting

It is necessary to show that the general resolution principle is *sound* and *complete*. However, a technical difficulty is the completeness of the proof. Using the Herbrand's theorem and semantic trees, we can prove that there is a ground *resolution refutation* of an unsatisfiable set of clauses. But, this cannot be generalized as a proof for general resolution, because the concept of semantic trees cannot be generalized. Why it cannot be generalized, is due to the variables, which give rise to potentially infinite number of elements in the Herbrand's base, as we will show it shortly.

Fortunately, there is a technique, called, "Lifting", to prove completeness of a theorem. Following are the steps for lifting:

1. first prove the *completeness* of the system for a set of *ground* classes, then,
2. as a second step, lift the proof to non-ground case.

**Example 3.16** Infinite inferences.

Let us assume that there are two non-ground clauses: 1. $p(u, a) \vee q_1(u)$ and, 2. $\neg p(v, w) \vee q_2(v, w)$. If the signature pattern contains function symbols, then these clauses have infinite *set* of instances, as follows:

$\{p(r, a) \vee q_1(r) \mid r \text{ is ground}\}$.
$\{\neg p(s, t) \vee q_2(s, t) \mid s, t \text{ are ground}\}$.

We can resolve above instances if and only if $r = s$ and $t = a$. Then we can apply the resolution refutation and obtain the inference given in the denominator of Eq. (3.21), which are infinite, due to variable $s$.

$$\frac{p(s, a) \vee q_1(s), \neg p(s, a) \vee q_2(s, a)}{q_1(s) \vee q_2(s, a)} \tag{3.21}$$

$\square$

The above difficulty can be overcome by taking a ground resolution refutation and "lifting" it to a more abstract general form.

The lifting is an idea to represent infinite number of ground inferences of the form given in Eq. (3.21) by a single non-ground inferences:

$$\frac{p(u, a) \lor q_1(u), \neg p(v, w) \lor q_2(v, w)}{q_1(v) \lor q_2(v, a)}$$

This lifting can be done using most general unifier, we will be discussing shortly.

**Example 3.17** Find out the Lifting for following clauses:

$C_1 = p(u) \lor p(f(v)) \lor p(f(w)) \lor q(u)$
$C_2 = \neg p(f(x)) \lor \neg p(z) \lor r(x)$

Using the substitution $\theta = \{f(a)/u, a/v, a/w, a/x, f(a)/z\}$, the above clauses become $C_1' = p(f(a)) \lor q(f(a))$, and $C_2' = \neg p(f(a)) \lor r(a)$. Using $C_1'$ and $C_2'$, it resolves to $C' = q(f(a)) \lor r(a)$. The lifting claims that there is a clause $C = q(f(x)) \lor r(x)$ which is resolvent for clauses $C_1$ and $C_2$, such that clause $C'$ is ground instance of $C$. This can be realized using the *unification algorithm* to obtain a most general unifier (mgu) of clauses $C_1$ and $C_2$, the latter two clauses resolves to $C$, as
$\{f(x)/u, x/v, x/w, f(x)/z\}$.

### 3.9.2 Unification Algorithm

A unification algorithm is central to most of the theorem-proving systems. This algorithm receives as input a pair of expressions, and returns as output a set of substitutions (assignments) that make the two expressions look identical.

The unification algorithm recursively compares the structures of the clauses to be matched, working across element by element. The criteria is that,

1. the matching individuals, functions, and predicates must have the same names,
2. the matching functions and predicates must have the same number of arguments, and
3. all bindings of variables to values must be consistent throughout the whole match.

To unify two atomic formulas in an expression **A**, we need to understand the *disagreement* set.

**Definition 3.8** Disagreement Set.

If **A** is any set of well-formed expressions, we call the set $D$ the disagreement set of **A**, whenever $D$ is the set of all well-formed subexpressions of the well-formed expressions in **A**, which begin at the first symbol position at which not all well-formed expressions in **A** have the same symbol.  □

**Example 3.18** Find out the disagreement set for given set of atoms.

Let the string is, $\mathbf{A} = \{p(x, h(x, y), y), p(x, k(y), y), p(x, a, b)\}$, having three predicate expressions. The disagreement set for **A** is,

$$D = \{h(x, y), k(y), a\}. \tag{3.22}$$

Once the disagreement is resolved through unification for this this symbol position, there is no disagreement at this position. The process is repeated for the new first symbol position at which all wffs in **A** do not have same symbol, and so on, until **A** becomes a singleton.

Evidently, if **A** is nonempty and is not a *singleton* (a set with exactly one element), then the disagreement set of **A** is not a singleton and nonempty. Also, if $\theta$ unifies **A**, and **A** is not singleton, the $\theta$ unifies the disagreement set **A**.                                    □

For **A** to be a finite nonempty set of well-formed expressions for which the substitution Algorithm 3.2 terminates with "return $\sigma_{\mathbf{A}}$", the substitution $\sigma_{\mathbf{A}}$ available as output of the unification algorithm is called the most general unifier (mgu) of **A**, and **A** is said to be most generally unifiable [8, 9].

---

**Algorithm 3.2** Unification-Algorithm (Input: **A**, Output: $\sigma_{\mathbf{A}}$)

---

1: Set $\sigma_0 = \varepsilon, k = 0$
2: **while** true **do**
3:     **if** $A\sigma_k$ is a singleton **then**
4:         Set $\sigma_{\mathbf{A}} = \sigma_k$
5:         terminate
6:     **end if**
7:     Let $U_k$ be the earliest and $V_k$ be the next earliest element in the disagreement set $D_k$ of $A\sigma_k$ (see Eq. 3.22)
8:     **if** $V_k$ is a variable, and does not occur in $U_k$ **then**
9:         set $\sigma_{k+1} = \sigma_k\{U_k/V_k\}$,
10:        $k = k + 1$
11:    **else**
12:        (**A** is not unifiable)
13:        exit.
14:    **end if**
15: **end while**

---

Through manually running the Algorithm 3.2 for the disagreement set in (3.22), stepwise computation for $\sigma_k$ is as follows:

For $k = 0$, and $\sigma_0 = \varepsilon$,

$$\sigma_{k+1} = \sigma_k\{k(y)/h(x, y)\}$$
$$\Rightarrow \sigma_1 = \{k(y)/h(x, y)\}.$$

which, in the next iteration becomes,

$$\sigma_2 = \sigma_1\{a/k(y)\}$$
$$= \{k(y)/h(x, y)\}\{a/k(y)\}.$$

The same process is repeated for the disagreement set of 3rd argument in **A**, which results to substitution set as $\{b/y\}$.

$$\sigma_3 = \sigma_2\{b/y\}$$
$$= \{k(y)/k(x, y)\}\{a/k(y)\}\{b/y\}.$$

On substituting these, we have,

$$\mathbf{A} = \{p(x, a, b), p(x, a, b), p(x, a, b)\}.$$

which is a singleton, and $\sigma_3$ is mgu.

For obtaining the unifier $\sigma_k$, the necessary relation required between $U_k$ and $V_k$ is, $V_k$ has to be a variable, and $U_k$ can be a constant, variable, function, or predicate. $V_k$ may even be a predicate or function with variable.

The Algorithm 3.2 always terminates for finite nonempty set of well-formed expressions, otherwise it would generate an infinite sequence of $A, A\sigma_1, A\sigma_2, \ldots$, each of which is a finite nonempty sets of well-formed expressions, with the property that each successive set contains one less variable than its predecessor. However, this is impossible because $A$ contains only finitely many distinct variables.

The Algorithm 3.2 runs in $O(n^2)$ time on the length of the terms, and an even better. However, there exists more complex, but linear time algorithms for same. Because, most general unifiers (mgus) greatly reduce the search, and can be calculated efficiently, almost all Resolution-based systems implementations are based on the concept of mgus.

## 3.10 Unfounded Sets

In the well-founded semantics, the unfounded sets provide the basis for negative conclusions. Let there is a program **P** (set of rules and facts in FOPL), its associated Herbrand base is $H$, and suppose its partial interpretation is $I$. Then, some $A \subseteq H$ is called an *unfounded set* of **P** with respect to the interpretation $I$, with following condition: for each instantiated rule $R \in \mathbf{P}$, at least one of the following holds: (In the rules **P** , we assume that $p$ is a head, and $q_i$ are the corresponding subgoals.)

1. Some positive / negative subgoal $q_i$ of the body of the rule is false in the interpretation $I$,
2. Some positive subgoals $q_i$ of the body occurs in the unfounded set $A$.

For rule $R$ with respect to $I$, a literal that makes conditions 1 or 2 above true is called *witness of unusability*.

Intuitively, the interpretation $I$ is intended model of **P**. The rules that satisfy condition 1 cannot be used for further derivations because their hypotheses are already known to be false.

The condition 2 in above, called *unfoundedness condition*, states that all the rules which might still be usable to derive something in $A$, should have an atom (i.e., a fact) in $A$ as true. In other words, there is no single atom in $A$, that can be established to be true by the rules of **P** (as per the knowledge of interpretation $I$). Therefore, if we infer that some or all atoms in $A$ are false, there is no way available later, using that we could infer that an atom is true [4].

Hence, the well-founded semantics uses conditions 1 and 2 to draw negative conclusions, and simultaneously infers all atoms in $A$ to be false. The following example demonstrates the construction of unfounded set from the set of rules and facts.

**Example 3.19** Unfounded set.

Assume that we have a program in predicate logic with instantiated atoms.

$$p(c).$$
$$q(a) \leftarrow p(d).$$
$$p(a) \leftarrow p(c), \neg\, p(b).$$
$$q(b) \leftarrow q(a).$$
$$p(b) \leftarrow \neg\, p(a).$$
$$p(d) \leftarrow q(a), \neg\, q(b).$$
$$p(d) \leftarrow q(b), \neg\, q(c).$$
$$p(e) \leftarrow \neg\, p(d).$$

From above rules, we see that $A = \{p(d), q(a), q(b), q(c)\}$ is an unfounded set with respect to $\phi$ (null set). Since $A$ is unfounded, its subsets are also unfounded. The component, $\{q(c)\}$ is unfounded due to condition (1), because there is no rule available to establish its truth. The set $\{p(d), q(a), q(b)\}$ is unfounded due to condition (2) (their subgoals or body appear in unfounded set.

There is no way available to establish $p(d)$ without first establishing $q(a)$ or $q(b)$. In other words, whether we can establish $\neg q(b)$ to support the first rule for $p(d)$ is irrelevant as far as determination of unfoundedness is the concern.

Interestingly, there is no way available to establish $q(a)$ in the absence of first establishing $p(d)$, and also there is no way available to establish $q(b)$ without first establishing $q(a)$. Further, $q(c)$ can never be proven. We note that among $p(d)$, $q(a)$, and $q(b)$ as goals, none can be proved without the other two or their negation as subgoals.

The pair $p(a)$, $p(b)$, even though they depend on each other, but does not form an unfounded set due to the reason that the only dependence is through negation. Hence, it can be concluded that the any attempt for proof of $p(a)$ and $p(b)$ will fail, but this claim is faulty.

The difference between sets $\{p(d), q(a), q(b)\}$ and $\{p(a), p(b)\}$ is as follows: declaring any of $p(d)$, $q(a)$, or $q(b)$ false (unfounded), does not create a proof that any other element of the set is true.

Finally, consider the set $\{p(a), p(b)\}$: If any of the elements $p(a)$ or $p(b)$ is taken false, it becomes possible to prove that the other is true. And, if both are declared false together, there is an inconsistency.                                                          □

## 3.11   Summary

First-order logic is best suited as a basic theoretical instrument of a computer theorem proving program. From the theoretical point of view, an inference principle need only be *sound* (i.e., allow only logical consequences of premises to be deduced from them) and *effective* (i.e., it must be algorithmically decidable whether an alleged application of the inference principle is indeed an application of it). The resolution principle satisfies both.

Two types of semantics, namely, *operational* and *fixpoint*, have been defined for programing languages. The operational semantics defines input-output relation computed by a program in terms of the individual operations performed by the program inside the machine. Meaning of a program is nothing but the input-output relation obtained due to executing it in a machine.

A machine independent alternative to semantics, called *fixpoint semantics*, defines the meaning of a program as input-output relation which is the minimal fixpoint of a transformation associated with the program.

A FOPL statement is made of predicates, arguments (constants or variables), functions, operators, and quantifiers. Interpretation is process of assignment of truth values (True/False) to subexpressions and atomic expressions, and computing the resultant value of any expression/ statement.

It is easy to procedurally interpret the sets of clauses which contain at most one positive literal per clause. However, along with this any number of negative literals can also exist. Such sets of clauses are called *Horn sentences* or *Horn Clauses* or simply clauses.

The Predicate logic is a *nondeterministic* programming language. Consequently, given a single goal statement, several procedure declarations can have a name which matches the selected procedure call. Each declaration gives rise to a new subgoal statement.

A *proof procedure* which sequences the generation of derivations in the search for a refutation behaves as an *interpreter* for the program incorporated in the initial set of clauses. Defining an operational semantics for a programming language means to define an implementation independent *interpreter* for it. For predicate logic, the proof procedure behaves as such an interpreter.

The *Herbrand universe* is the set of ground terms which use the function symbols and constants that appear in the predicate logic program. The *Herbrand base* is defined as the set of atomic formulas formed by predicate symbols in the program, whose arguments are in the Herbrand universe.

Herbrands theorem is a fundamental theorem of mathematical logic, which allows a certain type of reduction of first-order logic to propositional logic.

A substitution component is any expression of the form $T/V$, where $V$ is any variable and $T$ is any term different from $V$, is called unifier. The $V$ is called variable of component $T/V$, and $T$ is called term of the component. A most general unifier (mgu) (i. e., simplest one) $\theta$ of literals $\rho_1$ and $\rho_2$ is a unifier that has the property that for any other unifier $\theta'$, there is a further substitution $\theta^*$ such that $\theta' = \theta\theta^*$.

The backbone of most theorem-proving systems is a unification algorithm. This algorithm returns a set of substitutions for a pair of input expressions. These substitutions may be assignments to variables or expressions, which make the two expressions (or variables or functions) identical or equivalent. To prove a theorem, one obvious strategy is to search forward from the axioms, using sound rules of inference. We try to prove a theorem by refutation. It requires to show that negation of a theorem cannot be True.

## Exercises

1. Apply the Resolution theorem to prove:

   "Socrates is mortal", given that
   All men are mortal, and
   Socrates is man.

2. What are the other methods for automated theorem proving? Explain any three in brief.
3. Convert the following into clause form:

$$\forall x[p(x) \wedge q(x)] \Rightarrow [R(x, I) \wedge \exists y(\exists z \; r(y, z)$$

$$\Rightarrow S(x, y))] \vee \forall x \; T(x).$$

4. Show that a formula in CNF is valid if and only if each of its disjunctions contains a pair of complementary literals $P$ and $\neg P$.
5. Prove or disprove the followings:

   a. If $S$ is a first-order formula, then $S$ is valid iff $S \rightarrow \perp$ is contradiction.
   b. If $S$ is a first-order formula and $x$ is a variable, then $S$ is contradiction iff $\exists x S$ is a contradiction.

6. Using the resolution principle prove the validity of following formula:

$$\forall x \exists y (p(f(f(x)), y) \wedge \forall z(p(f(x), z)$$
$$\rightarrow p(x, g(x, z)))) \rightarrow \forall x \forall y \; p(x, y).$$

7. Is the predicate logic deterministic or nondetermnistic programming language? justify for yes / no.

8. Consider a set of statements of FOPL that uses two 1-place predicates: *Large* and *Small*. The set of object constants are $a, b$. Find out all possible models for this program. For each of the following sentences find out the models in which each of the sentence becomes true.

    a. $\forall x \, Large(x)$.
    b. $\forall x \, \neg Large(x)$.
    c. $\exists x \, Large(x)$.
    d. $\exists x \, \neg Large(x)$.
    e. $Large(a) \wedge Large(b)$.
    f. $Large(a) \vee Large(b)$.
    g. $\forall x \, [Large(x) \wedge Small(x)]$.
    h. $\forall x \, [Large(x) \vee Small(x)]$.
    i. $\forall x \, [Large(x) \Rightarrow \neg Small(x)]$.

9. Find out the clauses for the following FOPL formulas.

    a. $\exists x \forall y \exists z (P(x) \Rightarrow (Q(y) \Rightarrow R(z)))$.
    b. $\forall x \forall y ((P(x) \wedge Q(y)) \Rightarrow \exists z R(x, y, z))$.

10. Define the required predicates and represent the following sentences in FOPL.

    a. Some students opted Sanskrit in fall 2015.
    b. Every student who opts Sanskrit passes it.
    c. Only one student opted Tamil in fall 2015.
    d. The best score in Sanskrit is always higher than the best score in Tamil.
    e. There is a barber in a village who shaves every one in the village who does not shave himself / herself.
    f. A person born in country $X$, each of whose parents is a citizen of $X$ or a resident of $X$, is also a resident of $X$.

11. Determine whether the expression $p$ and $q$ unify with each other in each of the following cases. If so, give the *mgu*, if not justify it. The lowercase letters are variables, and upper are predicate, functions, and literals.

    a. $p = f(x_1, g(x_2, x_3), x_2, b); \quad q = f(g(h(a, x_5), x_2), x_1, h(a, x_4), x_4)$.
    b. $p = f(x, f(u, x)); \quad q = f(f(y, a), f(z, f(b, z)))$.
    c. $p = f(g(v), h(u, v)); \quad q = f(w, j(x, y))$.

12. What can be the strategies for combination of clauses in resolution proof? For example, if there are $N$ clauses, in how many ways they can be combined?

13. Why resolution based inference is more efficient compared modus-ponens?

14. Let $\Gamma$ is knowledge base and $\alpha$ is inference from $\Gamma$. Give a comparison among the following inferences, in terms of their performances:

    a. Proof by Resolution, i.e., $\Gamma \vdash \alpha$,
    b. Proof by Modus poenes, i.e., $\Gamma \vdash \alpha$,
    c. Proof by Resolution Refutation, i.e., $\Gamma \cup \{\neg \alpha\} \vdash \phi$.

15. Given $n$ number of clauses, draw a resolution proof tree to demonstrate combining them. Suggest any two strategies.
16. Given the knowledge base in clausal form, is it possible to extract answers from that making use of resolution principle? For example, finding an answer like, "Where is Tajmahal located?"
17. Represent the following set of statements in predicate logic, convert them to clause from, then apply the resolution proof to answer the question : Did Ranjana kill Lekhi?

    "Rajan owns a pat. Every pat owner is an animal lover. No animal lover ever kills an animal. Either Rajan or Ranjana killed a pat, called Lekhi."
18. Explain:

    a. Unification
    b. Skolemization
    c. Resolution principle versus resolution theorem proving.

19. Use resolution to show that the following set of clauses is unsatisfiable.

    $$\{p(a, z), \neg p(f(f(a)), a), \neg p(x, g(y)) \vee p(f(x), y)\}.$$

20. Derive $\perp$ from the following set of clauses using the resolution principle.

    $$\{p(a) \vee p(b), \neg p(a) \vee p(b), p(a) \vee \neg p(b), \neg p(a) \vee \neg p(b)\}.$$

21. Give resolution proofs for the inconsistency $\forall x \, shaves(Barber, x) \rightarrow \neg shaves(x, x)$, where $Barber$ is a constant.
22. Consider ab locks-world described by facts and rules:

    Facts:

    $ontable(a), ontable(c), on(d, c), on(b, a), heavy(b),$
    $cleartop(e), cleartop(d), heavy(d), wooden(b), on(e, b).$

    Rules:

    All blocks with clear top are black.
    All wooden blocks are black.
    Every heavy and wooden block is big.
    Every big and black block is on a green block.

    Making use of resolution theorem find out the block that is on the green block.
23. Given the following knowledge base:

    If $x$ is on top of $y$ then $y$ supports $x$.
    If $x$ is above $y$ and they are touching each other then $x$ is on top of $y$.
    A phone is above a book.
    A phone is touching a book.

Translate the above knowledge base into clause form, and use resolution to show that the predicate "supports(book, phone)" is true.

24. How resolution can be used to show that a sentence is:

    a. Valid?
    b. Unsatisfiable?

25. "The application of resolution principle for theorem proving is a non-deterministic approach." justify this statement.

26. a. Use Herbrand's method to show that formula,

$$\forall x \ shaves(barber, x) \rightarrow \neg shaves(x, x)$$

   is unsatisfiable?
   b. What is Herband's universe for $S = \{P(a), \neg P(f(x)) \lor P(g(x))\}$?

27. Prove that $\forall x \neg p(x)$ and $\neg \exists x \ p(x)$ are equivalent statements.

28. Let $S$ and $T$ be unification problems. Also, let $\sigma$ be a most general unifier for $S$ and $\theta$ be a most general unifier for $\sigma(T)$. Show that $\theta\sigma$ is a most general unifier for $S \cup T$.

29. Write the axioms describing predicates: *grandchild, grandfather, grandmother, soninlaw, fatherinlaw, brother, daughter, aunt, uncle, brotherinlaw*, and *first-cousin*.

30. For each pair of atomic sentences in the following, find out the most general unifier.

    a. *knows(father(y), y)* and *knows(x, x)*.
    b. $\{f(x, g(x)) = y, h(y) = h(v), v = f(g(z), w)\}$.
    c. $p(a, b, b)$ and $p(x, y, z)$.
    d. $q(y, g(a, b))$ and $q(g(x, x), y)$.
    e. *older(father(y), y)* and *older(father(x), ram)*.

31. Explain what is wrong with the below given definition of set membership predicate $\in$:

$$\forall x, s : x \in \{x \mid s\}$$
$$\forall x, s : x \in s \Rightarrow \forall y : x \in \{y \mid s\}.$$

32. Consider the following riddle: "Brothers and sisters have I none, but that man's father is my father's son". Use the rules of kinship relations to show who that man is?

33. Let the following be a set of facts and rules:
    Rita, Sat, Bill, and Eden are the only members of a club.
    Rita is married to Sat.
    Bill is Eden's brother.
    Spouse of every married person in the club is also in the club.

   a. Represent the above facts and rules using predicate logic.
   b. Show that they do not conclude "Eden is not married."
   c. Add some some more facts, and show that now the augmented set conclude that Eden is not married.

# References

1. Chowdhary KR (2015) Fundamentals of discrete mathematical structures, 3rd edn. EEE, PHI India
2. Davis M, Putnam H (1960) A computing procedure for quantification theory. J ACM 7(3):201–215. https://doi.org/10.1145/321033.321034
3. Emden V, Kowalki RA (1976) The semantics of predicate logic as a programming language. J ACM 23(4):733–742
4. Av Gelder et al (1991) The well-founded semantics for general logic programs. J ACM 38(3):620–650
5. Luckham D, Nilsson NJ (1971) Extracting information from resolution trees. Artif Intell 2:27–54
6. Nilsson NJ (1980) Principles of artificial intelligence, 3rd edn. Narosa, New Delhi
7. Robinson JA (1963) Theorem-proving on the computer. J ACM 10(2):163–174
8. Robinson JA (1965) A machine-oriented logic, based on the resolution principle. J ACM 12(1):23–41
9. Stickel ME (1981) A unification algorithm for associative-commutative functions. J ACM 28(3):423–434

# Chapter 4
# Rule Based Reasoning

**Abstract** The popularity of rules-based systems (RBSs) is due to their naturalness. This chapter presents the potential applications of RBSs, the working of RBS, forward and backward chaining RBSs, their Algorithms, and inferencing using these systems. The analysis of complexity of preconditions, and efficiency of rule selection are introduced to sufficient depth, as well the cofmparison between the two types of RBSs are presented. A typical RBS, and other methods—model-based and case-based approaches are also discussed. In addition, number of solved, as well exhaustive list of exercises are provided at the end of the chapter for practice. The chapter concludes with its summary.

**Keywords** Rule-based systems (RBSs) · Forward chaining · Backward chaining · Forward chaining Algorithm · Backward chaining Algorithm · Model-based reasoning · Case-based reasoning · Conflict resolution

## 4.1 Introduction

Symbolic rules are popular for knowledge representation and reasoning. Their popularity stems mainly from their naturalness, which facilitates comprehension of the represented knowledge. The basic form of a rule is,

$$if \ <conditions> \ then \ <conclusion>$$

where $<conditions>$ represent the *conditions* or *premises* of a rule, and the $<conclusion>$ represent its conclusion or consequence. The conditions of a rule are connected between each other with logical connectives such as *AND/OR* thus forming a logical function. When sufficient conditions of a rule are satisfied, the conclusion is derived and the rule is said to *fire* (or *trigger*). Rules represent general knowledge regarding a domain.

**Table 4.1** Application areas of rule based systems

| Problem | System functions |
|---|---|
| Troubleshooting | Analyzing situations, suggesting measures, and prescribing preventative actions |
| Process control | Identifying problematic data and remedies of inequalities |
| Quality assurance | Assessment of tasks, proposing the practices, and enforcing the requirements |
| Equipment maintenance | Diagnosing various faults and recommending the repairs |
| Component selection | Specifying requirements and matching parts from an electronics catalog |
| Computer operation | Analyzing requirements to be fulfilled, selecting and operating a software |
| Product configuration | Specifying preferences, and identifying parts that satisfy constraints |

In this chapter we will discuss the use of easily stated *if-then* rule to solve problems. In particular you will learn the *forward-chaining* from the assertions and *backward* chaining from hypotheses. The illustrative examples will explain the rule firing sequences and graph search. The complexities of Algorithms for both forward and backward rule chaining are also analyzed and presented. In addition, the complexities of preconditions of rules, and other popular types of *expert systems*, like—*Case based reasoning*, and *model based reasoning* are presented.

Rule-based systems (RBSs) constitute the one of the most common and simple to implement means for codifying the problem-solving know-how of human experts. Experts tend to express most of their problem-solving techniques in terms of a set of *situation-action* rules, and this suggests that RBSs should be the method of choice for building knowledge-intensive expert systems. The RBSs share following key properties [3]:

1. They incorporate practical human knowledge in conditional *if-then* rules,
2. Their skill increases at a rate proportional to the enlargement of their knowledge bases,
3. By selecting the relevant rules and combining them appropriately, it is possible to solve a large range of problems, of varying complexities.
4. They can be designed to adaptively decide the best sequence of rules to execute, and
5. The RBS can retrace the reasoning steps and explain the justification for the conclusions drawn. In this retracing, it can show each rule executed, using natural language.

Table 4.1 lists the major application areas addressed by RBS.

**Learning Outcomes of this Chapter**:

1. Explain the difference between rule-based, case-based and model-based reasoning techniques. [Familiarity]
2. Inferences in rule-based systems. [Usage]
3. Complexities of rule-based systems. [Assessment]

## 4.2  An Overview of RBS

Roughly speaking, a Rule Based System comprises a *knowledge base* and an *inference engine* (see Fig. 4.1). The knowledge base is collection of facts and rules. Facts are simple statements with constant or variable object lists, while the Rules always express a conditional sentence, with one or more premises and a consequent component. The rules needs to be interpreted, which means, if a premise can be satisfied the consequent also be satisfied. The consequent can be a conclusion or an action. When it is an action, the effect of satisfying the premises is to schedule that action for execution, and when the consequent defines a conclusion, the effect is to infer the conclusion [2].

Several key techniques for organizing RBSs have emerged. Rules can be used to express *deductive knowledge*, such as logical relationships, and thereby to support inference, verification, or evaluation tasks. Conversely, rules can be used to express goal-oriented knowledge that an RBS can apply in seeking problem solutions and cite in justifying its own goal-seeking behavior. The rules can also be used to express *causal* relationships, which an RBS can use to answer "what-if" questions, or to determine possible causes for specified events.

**Fig. 4.1** Inference cycle of a forward-chaining RBS
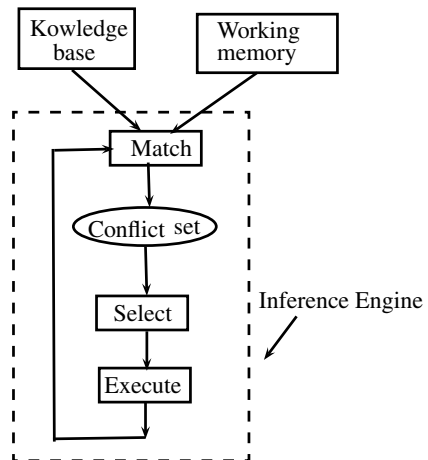
In a rule based system, each *if* pattern may match to one or more of assertions in a collections of assertions. The collections of assertions is called *working-memory* (Fig. 4.1). The assertions collectively may match to premises of one or more rules in the knowledge base. This is done by "match" block of the inference-engine. All the rules matching with the assertions in the working memory are put in the *conflict set*, from where one of the rule is "selected" based on some conflict resolving criteria set for, and then the selected rule is executed. The resulting consequences/conclusions (the then patterns) are put in the working memory to form new assertions, and the process continues till desired result (goal) is not reached.

A system like this is called *deduction system*, as it deduces new inferences from the rules and assertions. A realistic example of a rule is:

```
R1: if 1. stiff neck,
       2. high temperature, and
       3. impairment of conciousness occur togetehr,
    then
          meningitis is suspected.
```

In the above, meningitis is a disease related to "Inflation of membrane of spinal chord and brain" will be suspected by this rule if the "if" patterns 1, 2, 3 are found true.

RBS can be applied in judiciary systems also. Following is an example of a more complex a rule.

```
If the plaintiff received an eye injury
 and it was one eye injured
 and treatment for eye required surgery
 and recovery from the injury was almost complete
 and visual acuity was slightly reduced by the injury
 and there is fixed condition,
then increase the injury trauma factor by $5,000.
```

*Facts* is the kind of data in a knowledge base that express assertions about properties, relations, propositions, etc. In contrast to rules, which the RBS interprets as imperatives, facts are usually static and inactive implicitly. Also, a fact is silent regarding the pragmatic value and dynamic utilization of its knowledge. Although in many contexts facts and rules are logically interchangeable, in the RBSs they are quite distinct.

In addition to its static memory for facts and rules, an RBS uses a *working-memory* to store temporary assertions. These assertions record earlier rule-based inferences. We can describe the contents of working memory as problem-solving state information. Ordinarily, the data in working memory adhere to the syntactic conventions of facts. Temporary assertions thus correspond to dynamic facts.

The basic function of an RBS is to produce results. The primary output may be— a problem solution, an answer to a question, or result of an analysis of some data. Whatever the case, an RBS employs several key processing determining its overall

activity. A *world* manager maintains information in working memory, and a built-in control procedure defines the basic high-level loop; if the built-in control provides for programmable specialized control, an additional process manages branches to and returns from special control blocks.

Some times, the patterns specify the *actions* rather than *assertions*, e.g., "to put them on the table". In such case the rule based system is called *reaction system*.

In both the deduction systems and reaction systems, forward chaining is the process of moving from the *if* patterns to *then* patterns, where *if* patterns identifies the appropriate situation for deductions of new assertion, and performance of an action in the case of *reaction system*.

## 4.3   Forward Chaining

In a forward chaining system, we start with the initial facts, and use the rules to draw new conclusions (or take certain actions), given those facts. Forward chaining systems are primarily *data-driven*. Whenever an if pattern is observed to match an assertion, the antecedent is *satisfied*. When all the *if* patterns of a rule are satisfied, the rule is *triggered*. When a triggered rule establishes a new assertion or performs an action, it is *fired*. This procedure is repeated until no more rules are applicable (Fig. 4.1).

The selection process is carried out in two steps:

1. *Pre-selection*: Determining the set of all the matching rules, also called the *conflict-set*.
2. *Selection*: Selection of a rule from the conflict set by means of a conflict resolving strategy.

### 4.3.1   Forward Chaining Algorithm

A simple forward chaining Algorithm 4.1 starts from the known facts in the knowledge base, and triggers all the rules whose premises are the known facts, then adds the consequent of each into the knowledge base. This process is repeated until the query is answered or until there is no conclusion generated to be added into the knowledge base. We will use symbols $\theta, \lambda, \gamma$ to represent substitutions. The *unify* is a function unifies the newly generated assertion $q'$ and the query $\alpha$, and returns a unifying substitution $\lambda$ if they are unified, else returns null.

Let $\alpha$ be the goal. The forward-chaining Algorithm 4.1 picks up any sentence $s \in \Gamma$, where $\Gamma$ is knowledge base, and checks all possible substitutions $\theta$ for $s$. Let, the predicate form of $s$ is $p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q$. On substituting $\theta$, say, it results to $(p_1 \wedge p_2 \wedge \ldots p_n)\theta = (p'_1 \wedge p'_2 \wedge \ldots p'_n)$, such that $p'_i$'s $\in \Gamma$. Let $(p'_1 \wedge p'_2 \wedge \ldots p'_n) \rightarrow q'$, such that $q' = q\theta$. This $q'$ is added into *new* inference. If the

inference $q'$ and goal $\alpha$ unify, then their unifier $\lambda$ is returned as the solution, else the Algorithm continues.

---

**Algorithm 4.1** Forward-chaining(Input: $\Gamma, \alpha$) // $\alpha$ is a query, $\Gamma$ is knowledge base

1: **while** True **do**
2:    $new = \{\}$
3:    **for** each sentence $s \in \Gamma$ **do**
4:        Convert $s$ into the format $p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q$
5:        **for** each substitution $\theta$ such that $(p'_1 \wedge p'_2 \wedge \ldots, \wedge p'_n) \leftarrow (p_1 \wedge p_2 \wedge \ldots, \wedge p_n)\theta$ for some $p'_i s \in \Gamma$ **do**
6:            $q' \leftarrow q\theta$
7:            $new \leftarrow new \cup \{q'\}$
8:            $\lambda \leftarrow unify(q', \alpha)$
9:            **if** $\lambda$ is not null **then**
10:                return $\lambda$
11:            **end if**
12:        **end for**
13:        $\Gamma \leftarrow \Gamma \cup new$ // add new inferences in knowledge base
14:    **end for**
15: **end while**
16: Return Fail

---

The Algorithm may fail to terminate in the case when the raised query has no answer. For example, every natural number can be generated by recursively applying successor operation on a natural number, and assuming that 0 is natural number. This will lead to indefinite loop for very large numbers.

$$naturalnum(0).$$
$$\forall x[naturalnum(x) \rightarrow naturalnum(succ(x)).]$$

The following example demonstrates the manual run of forward chaining Algorithm.

**Example 4.1** Produce the inference, given the knowledge base $\Gamma = \{man(x) \rightarrow mortal(x), man(socrates)\}$ and query $\alpha = mortal(w)$, i.e., "Who is mortal?"

We follow the Algorithm 4.1 manually and note that $p_1 \wedge \cdots \wedge p_n = p_1 = man(x)$; $p'_1 = man(socrates)$, substitution $\theta = \{socrates/x\}$, and $q = mortal(x)$. Also,

$$q' = q\theta$$
$$= mortal(x)\{socrates/x\}$$
$$= mortal(socrates).$$

Also, $new = \{\} \cup \{q'\} = mortal(socrates)$.
On unification of $\alpha$ (i.e, $mortal(w)$), and $q'$, the unifier $\lambda$ obtained is,

$$\lambda = unify(q', \alpha)$$
$$= unify(mortal(socrates), mortal(w))$$
$$= \{socrates/w\}.$$

Hence, the answer for query is $w = socrates$.  □

Complexity Issues

The complexity of the forward chaining Algorithm is determined by the inner loop in the Algorithm 4.1. It finds all the possible premises such that the premises unify with certain set of facts in the knowledge base $\Gamma$. The process is called *pattern matching*, and tried for every rule, for every substitution $\theta$. Thus, if set of rules are $P$, there are $n^k$ substitutions for each rule, assuming that in worst case $k$ arguments and $n$ number of literals' assignments for each argument in the $P$. This makes the complexity of above Algorithm as,

$$|P|n^k \tag{4.1}$$

which is exponential. However, since the size and arities are constant in the real-world, the complexity of expression (4.1) is a polynomial in nature. To search the predicates for matching, they are indexed and a hash function is generated for quick search.

## 4.3.2 Conflict Resolution

When we are doing data-directed reasoning, we may not like to fire all of the rules in case more than one rule is applicable. In cases where we want to eliminate some applicable rules, some kind of conflict resolution is necessary for arriving at the most appropriate rule(s) to fire. In a deduction system all rules generally fire, but in a reaction system, when more than one rule are triggered at the same time, only one of the possible actions is desired [4].

The conflict resolving strategy is used to avoid superfluous rules. The most obvious strategy is to choose a rule at random, out of those that are applicable. Following are some common approaches for deciding the preferences for the rule to fire.

Selection by order

Selection by order may either on the order in which the rule appears in the knowledge base, or the order may be based on how recent the rule was used.

*Order*

Pick the first applicable rule in the order they have been presented. This is the type of strategy used by *Prolog*, and is one of the most common ones. Production system programmers would take this strategy into account when formulating rule sets.

*Recency*

Select an applicable rule based on how recently it has been used. There are different versions of this strategy, ranging from firing the rule that matches on the most recently created (or modified) wff, to firing the rule that has been least recently used. The former could be used to make sure that a problem solver stays focused on what it was just doing (typical of depth-first search); the latter would ensure that every rule gets a fair chance to influence the outcome (typical of breadth-first search).
Selection according to syntactic structure of the rule
There are many ways one can consider the syntax, e.g., it may be conditions and their specificity, or the size of the rule: largest, smallest, or in increasing order of size.

*Specificity Criteria*

Select the applicable rule whose conditions are most specific. A set of conditions is taken as more specific than other if the set of wffs that satisfy it is a subset of those that satisfy the other. For example, consider the following three rules:

i)   *if bird($x$) then add(canfly($x$))*
ii)  *if bird($x$) ∧ weight($y$, $x$) ∧ gt($y$, 5kg) then*
     *add(cannotfly($x$))*
iii) *if bird($x$) ∧ penguin($x$) then add(cannotfly($x$))*

Here, the second and third rules are both more specific than the first, because the condition in (ii) and (iii) are more specific than in (i) (condition of (i) is very general). If we have a bird that has weight greater than 5 kg, or it is a penguin, then the first rule applies, because the condition of "bird" is satisfied. However, in this case the other two rules should take precedence, as they are more specific. Note that if the bird is a penguin and heavy, neither second nor third is applicable, and another conflict resolution criteria must be applied to decide between the second and third rules.

*Largest Rule First*

Apply the syntactically largest rule, i.e., the rule which contains the most propositions.

*Incremental in Size*

Solutions to subproblems are constructed incrementally, and are cached to avoid the recompilation.

Selection by Means of Supplementary Knowledge

The supplementary knowledge about the rules may be in the form of priority of the rules, or as meta-knowledge.

*Highest Priority First*

For this purpose each rule must be given a priority, which may be represented, e.g., by a numeric value.

Apart from the priority criteria, there are additional rules, called *meta-rules*, control the selection.

The simple way of implementing the forward-chained interpreter is to check the preconditions of all the rules, sort the applicable rules in an agenda (i.e., an ordered list) according to the criteria given by the conflict-resolving strategy, and then perform the action of the leading rule in the agenda, followed with this previous agenda is deleted and the process begin as a new.

### 4.3.3   *Efficiency in Rule Selection*

Following are the criteria for selection of rules for implementation of reasoning process.

Similarity Between Rules

The first improvement over the brute-force approach is based on the similarity between preconditions of different rules. Using this, rules with a common propositions may be eliminated all together if these propositions do not hold. For achieving this, the rules are structured in a tangled hierarchy by constructing a network of interconnecting trees in which the internal decision nodes are formed by the propositions and the leaves of the rules. The path from the root of the tree to a rule includes all the propositions of the precondition of this rule.

Check Applicability of New Rules

This improvement is based on the fact that a complete new calculation of an agenda requires far more effort than a modification on the basis of changes. From one cycle to next, the knowledge base is altered only by the action of the selected rule, which may add or delete portions of knowledge. Thus, it needs to be tested that whether the deleted knowledge destroy the applicability of any rule in the old agenda and whether the added knowledge make any new rules applicable. The later test is performed by running through the rule network with new knowledge as starting point.

Indexing

One way of improving over the brute-force Algorithm is to index all the rules so that each parameter of the predicate produces a reference to its rule. When a parameter takes on a new value or changes its value, the reference enables the rules concerned to be found and checked efficiently.

Consider the following rule, for which many conjunct ordering may be possible,

$$\forall x \forall y [p(x)q(x, y) \rightarrow r(x, a)]. \tag{4.2}$$

Here, all the $p(x)$ may be ordered, next find $q(x, y)$ for each $x$ and $y$. Alternatively, for each $x$ find $p(x)$, then find all $q(x, y)$ for different $y$. Which approach is better? Finding this is solution of a *conjective-ordering problem*, which is *NP-hard*. A heuristic can use most constrained, the one having fewest values. This shows

a close relationship between *CSP* (constraint satisfaction problem), discussed in next chapters, and a pattern matching problem. Due to these complexity issues the forward-chaining lgorithm 4.1 is *NP-hard* in its inner loop.

### 4.3.4   Complexity of Preconditions

Different formalisms have differing expressive power of describing the preconditions and the rules. The differences are those between propositional and predicate logic. The count of number of rules is not a sufficient criteria for the indication of the size of an expert system. Ideally each relevant situation of the domain should be covered by just one rule. Description of a situation by different rules will impair the modularity of the knowledge base since these rules should be meaningful only as group. On the other hand, more expressive power of the rule formalism require a more complex and often more inefficient rule interpreter so that one must check how much complexity is necessary for the domain [4].

The simplest method of evaluating preconditions is lookup in the knowledge base. For example, in the case of query: "Is the pain intensifying synchronous with respiration?", it is only necessary to check whether the value—"synchronous with respiration", is stored under the object - "pain intensifying". With the logical connectives, *and*, *or*, *not*, this situation can be described. However, in the case of numerical or temporal relationships, the looking alone is not sufficient, and additional, calculations are required. For example, for the query "Did shortening of breath occur before throat pain". here, the preposition "before" has reference to time. Does it mean 1 second or a minute, or a day, because all these satisfy the criteria of before ! There is need of comparison, using the relations operators, like, less than, greater than, equal, which are needed to be applied along with the time information. Also, depending on the situation, magnitude of precede be available in the knowledge base.

One of the drawback of forward chaining is that it makes all allowable inferencing based on the facts and rules available, irrespective of whether they are relevant to deriving goals or not. This makes the forward chaining, not a very efficient approach for reasoning. To avoid this problem, we work for selected subclass of rules. The other approach, the backward chaining, arrives to only those premises which certainly lead to goal, hence the system is called goal driven.

## 4.4   Backward Chaining

While forward chaining allows conclusions to be drawn only from a given knowledge base, a backward-chained rule interpreter is suitable for requesting still unknown facts. In a backward chaining system, you start with some hypothesis (goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new sub-goals to prove as you go.

For this, the goal expression is initially placed in the working memory, followed with this, the system performs two operations:

1. matches the rule's "consequent" with the goal. For example, $q$ in $p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q$,
2. selects the matched rule $p_1 \wedge p_2 \wedge \ldots p_n \rightarrow q$ and places its premises ($p_1 \wedge \cdots \wedge p_n$) in the working memory.

The second step in above corresponds to the transformation of the problem into subgoals. The process continues in the next iteration, with these premises becoming the new goals to match against the consequent of other rules. Thus, the backward chaining system, in the human sense are *hypothesis testing*.

Backward chaining uses *stack*. First, the goal is put in the stack, then all the rules which results to this goal are searched and put on the stack. These becomes the subgoals, and the process goes on till all the facts are proved or are available as literals (ground clauses). Every time the goal and premises are decided, the unification is performed.

### 4.4.1 Backward Chaining Algorithm

The Algorithm for backward chaining returns the set of substitutions (*unifier*) which makes the goal true. These are initially empty. The input to the Algorithm is knowledge base $\Gamma$, goals $\alpha$, and current substitution $\theta$ (initially empty). The Algorithm returns the substitution set $\lambda$ for which the goal is inferred. The Algorithm 4.2 is the backward-chaining Algorithm.

---

**Algorithm 4.2** Backward-chaining(Input: $\Gamma, \alpha, \theta$) // $\alpha$ is a query, $\Gamma$ is knowledge base, $\theta$ current substitution (initially empty), $\lambda$ represent substitution set for the query to be satisfied (initially empty)

---

1: $\theta = \{\}, \lambda = \{\}$
2: $q' \leftarrow \alpha\theta$
3: **for** each sentence $s \in \Gamma$, where $s = p_1 \wedge p_2 \wedge \ldots, \wedge p_n \rightarrow q$ and $\gamma \leftarrow unify(q, q') \neq null$ **do**
4:    $\alpha_{new} \leftarrow (p_1 \wedge p_2 \wedge \cdots \wedge p_n)$
5:    $\theta \leftarrow \theta\gamma$
6:    $\lambda \leftarrow$ backward-chaining($\Gamma, \alpha_{new}, \theta) \cup \lambda$
7: **end for**
8: return $\lambda$

---

**Example 4.2** Apply the backward-chaining Algorithm for inferencing from a given knowledge base.

Let $\Gamma = \{man(x) \rightarrow mortal(x), man(socrates)\}$. Assume that it is required to infer answer for "Who is mortal?" That is, goal $\alpha = mortal(w)$, find $w$. The loop iterations in the backward-chaining Algorithm 4.2 are as follows:

1st Iteration: Initially $\theta$ is empty, hence, $q' = \alpha\theta = mortal(w)$. From Algorithm and knowledge base $\Gamma$, the sentence is, $s = (man(x) \rightarrow mortal(x))$. Next, $\gamma = unify(mortal(x), mortal(w)) = \{w/x\}$. Also, the new goal, $\alpha_{new} = man(x)$. The new value of current substitution is, $\theta \leftarrow \theta\gamma = \{\}\{w/x\} = \{w/x\}$. Next, compute $\lambda = backward\text{-}chaining(\Gamma, man(x), \{w/x\}) \cup \lambda$, as a recursive call.

Recursive call: We apply the Algorithm in a recursive mode, and get $q' = man(x)$ $\{w/x\} = man(w)$. Next, the subgoal $man(w)$ (i.e., $q'$) matches with other subgoal (it is a *fact*) $man(socrates) \in \Gamma$. Hence, $\gamma = unify(man(socrates), man(w)) = \{socrates/w\}$. Next, the new goal is, $\alpha_{new} = man(socrates)$ and new substitution is:

$$\begin{aligned}
\theta &= \theta\gamma \\
&= \{w/x\}\{socrtaes/w\} \\
&= \{socrates/x\}.
\end{aligned}$$

In the next call of recursion at step 5, $q' = \alpha\theta = man(socrates) \{socrates/x\}$, the substitution fails, as there is no $x$ where "socrates" can be substituted. Hence, $q' = null$. Since $\gamma$ is null, the *for loop* at step 3 terminates, and returned value of $\lambda$ is $\{socrates/w\}$, i.e., $w = socrates$, or $mortal(socrtaes)$.                 $\square$

### 4.4.2  Goal Determination

If  the goal is not known in the knowledge base, the rule interpreter first decides whether it can be derived or must be requested from user. For the derivation of goal, all the rules which includes the goal in their consequent part are processed. These rules can be identified efficiently if they are indexed according to their consequent parts.

Backward chaining therefore contains implicitly a dialogue control, where order of the questions decides the order of the rules for deriving a parameter. This dependency on the order reduces the modularity of the rule-based system. The efficiency of the backward-chained rule interpreter is determined by the formulation of the goal: the more precise the goal, the smaller is the search tree of rules to be checked and questions to be asked. For example, in medicine, a query may be: "What is name of disease?" as against an alternate query of "Is $X$ the name of disease?".

The examples of back-ward chained rule interpreter are MYCIN, and PROLOG.

## 4.5  Forward Versus Backward Chaining

Whenever the rules are such that typical set of facts relate to large number of con-clusions, i.e., the *fan-out* is larger than *fan-in*, it is better to use backward chaining. This is to stop in explosive growth in search space. On the other hand, when the rules

**Table 4.2** Knowledge base for Animal-Kingdom

| S. no. | Rule |
|---|---|
| 1 | $sponge(x) \rightarrow animal(x)$ |
| 2 | $arthopod(x) \rightarrow animal(x)$ |
| 3 | $vertebrate(x) \rightarrow animal(x)$ |
| 4 | $fish(x) \rightarrow vertebrate(x)$ |
| 5 | $mammal(x) \rightarrow vertebrate(x)$ |
| 6 | $carnivore(x) \rightarrow mammal(x)$ |
| 7 | $dog(x) \rightarrow carnivore(x)$ |
| 8 | $cat(x) \rightarrow carnivore(x)$ |

are such that a typical hypothesis can lead to many facts, i.e., *fan-in* is larger than *fan-out*, it is better to use forward chaining. The reason being that if we go backward in a case where fan-in is high, it would lead to exponentially growing search space, so we prefer forward chaining in such knowledge.

If there is a situation that all the required facts are known, and we want to get all the possible conclusions from these, the forward chaining is better. However, if the rules are such that facts are incompletely known, we cannot proceed from the facts to conclusions, and thus goal driven strategy should be used.

Forward chaining is often preferable in cases when there are many rules with the same conclusions, because we choose the satisfying premises. In backward chaining it may lead to non-satisfying premises. The following examples demonstrate this.

**Example 4.3** Given a knowledge base for an animal kingdom, infer *animal(bruno)* after adding of *dog(bruno)*. The taxonomy of the animal kingdom includes such rules as shown in Table 4.2.

It is required to show that, $KB + dog(bruno) \rightarrow animal(bruno)$.

1. *Forward Chaining.* We start with rule 7, and unify $dog(bruno)$ with $dog(x)$. Next, we will successively infer and add *carnivore(bruno)*, *mammal(bruno)*, *vertebrate(bruno)*, and *animal(bruno)*. The query will then succeed immediately. The total work is proportional to the height of the hierarchy of this taxonomy, which is 4.
2. *Backward-chaining.* Alternatively, if we use backward chaining, the query *animal(bruno)* will unify with the first rule above and generate the sub-query *sponge (bruno)*, which will initiate a search for *bruno* through all the subdivisions of sponges. Not finding, it tries with 2nd rule, but orthopod is not in consequent. Next, with rule 3, the goal driven chain is: "animal $\rightarrow$ vertebrate $\rightarrow$ fish", which fails. The successful invocation rule sequence is "3 $\rightarrow$ 5 $\rightarrow$ 6 $\rightarrow$ 7." Thus, it searches the entire taxonomy of animals looking for $dog(bruno)$. We note that work done in the background chaining much more than forward chaining. However, this is not necessarily true always. $\square$

In some cases, it is desirable to combine both forward and backward reasoning, and due to the merits of individual rules they are identified as forward / backward. However, this process results to a far more complex mechanism.

## 4.6 Typical RB System

The $R_1$ (later called as *XCON*—for e**x**pert **con**figurator) was production rule-based expert system, which was written in *OPS5* language by *John P. McDermtt* in 1978 to help in ordering DEC's VAX computer system by automatically selecting the computer systems components, based on the requirements of customers. When fully developed it had 2500 rules, which provided over 95% accuracy in configuration of systems as per customers needs [5].

The XCON differed from other systems mainly in its use of "match" rather than "generate-and-test" as its central problem solving strategy. It exploits knowledge of its task domain to generate a single acceptable solution. The input to XCON is customer's order and output is a set of diagrams showing spatial relationships among the components of the order. These diagrams act as guidance to the technician who assembles the system. For example two inter-dependent activities needs to be performed for configuring a VAX mini-computer system:

1. It is determined that the customer's order is complete; if it is not, whatever are the components in shortfall as per the standard order, must be added.
2. Next, the spatial relationships among all of the components is to be determined (including those that are added).

The criterion of success whether a configuration is complete or not, cannot be determined by any simple test, but involves instead particular knowledge about all the individual components and their relationships. The criterion of successful spatial arrangement is that it should be geometrically compact, and in addition, it involves particular knowledge on a component by component basis. Thus, the task accomplishment is defined by a large set of constraints comprising a large amount of knowledge.

## 4.7 Other Systems of Reasoning

In addition to the methods discussed above, there exists reasoning methods, like, *model based reasoning*, *case based reasoning*, and some *hybrid* combinations of these.

### 4.7.1 Model-Based Systems

Consider a situation of applying the goal driven strategy of trouble-shooting electronic circuits, like, TV, or electron-microscope. On confirming that the system does not function (i.e., given the goal), we try to find out systems modules (i.e., sub-goals) and check which out of these are working. For the unit which is not working, we try to find out its sub units, and carry on the check like this, till we reach to component levels, like resisters, capacitors, etc. In addition, we also need to look at the history of failure rate of the components. This, in fact becomes a very exhaustive work.

Instead, it would be better if we take the physical model of the system, with mathematical equations describing the interactions and relationships between them. It would then base the diagnosis on the signal reading at various points in the circuit.

A *Model based* reasoning comprises following details:

1. A description of each component in the device. These description can simulate the behavior of the component.
2. A description of the device's internal structure: A representation of components and their interconnections, with ability of simulation of interactions.
3. Information about the input output relationships and magnitudes of values at input and outputs.

To have a glimpse of a model based reasoning, consider a model shown in Fig. 4.2. The input and output relations are functionally defined, expected and actual values are shown in the output. Given this, we can determine the faulty component.

This model comprises *adders* and *multiplications* units which deliver the output as per the functions they perform. In the multiplier-2 the actual output is 56, but expected is 63. Hence the possibility is, either the Mult-2 is faulty or its inputs are faulty. Since the actual output of Mult-1 and its expected are matching, that confirms that, Mult-1 and its two inputs are correct. Thus possibility left is, either first input to Mult-2 is wrongly delivered, or add-2 is malfunctioning, or input $D$ is incorrect. Assuming that connections are not faulty, then either Mult-2 is faulty, or Add-3 is faulty, or the input $D$ is faulty, which can be systematically checked to locate the fault.



**Fig. 4.2** Model to be troubleshooted
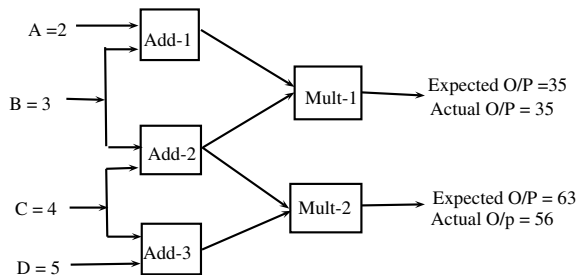
### 4.7.2  Case-Based Reasoning

The case-based reasoning (CBR) uses specific database for problem solving. Consider the case of medical practitioner, who uses medical history of failed as well as successful "cases" for diagnosis of future patients. If a new patient's case appears similar to one of the earlier patient's successful case treatment case, the knowledge history of old case is useful. At the same time if some old case has failed, that will not be tried again due to available history.

Figure 4.3 shows the functional block diagram of case based reasoning. Often, the researchers use the combination of above cases for reasoning due to their advantages.

The similar is the situation in the case of legal practitioners. Lawyers, are allowed by law, to quote the old case histories in their legal proceedings, if that case matches in the present situations. They can also give a different interpretation for the old case to deal with new situations, even if they do not fully match [1].

The above are the examples of *case based reasoning*.

Many other cases can be counted in this category: programmers often reuse the old code to fit in the new situation with partial modification, architects reuse the design of older successful architectures, the wars winners learn from the history of wars won, all of us often follow the success stories of others in life, and so on. Many a times, the case where we need to apply the reasoning is not identical to the previous, hence some transformations are required to fit into the current situation. Thus, this reasoning is a a kind of learning, from the analogy, called *analogical leaning*.

For each new problem, the case based reasoning researchers share the following common operations:

1. Retrieve appropriate cases from the storage (case database).
2. Modify the retrieved case to fit into the current situations.
3. Apply the transferred rule.
4. Save the solution along with the case details, for success or failure for future use.
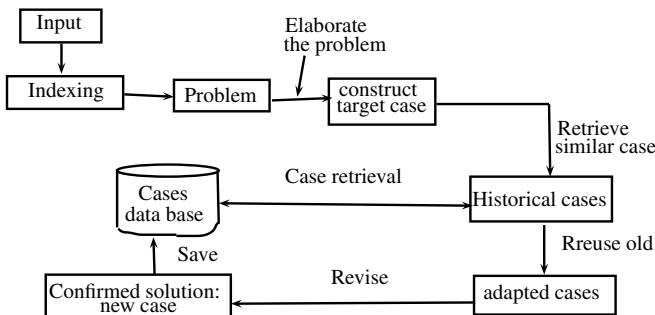


**Fig. 4.3**  Case-based Reasoning

## 4.8 Summary

Some of the applications of rule based expert systems are: component selection, equipment maintenance, product configuration, computer operation & troubleshooting, process control, and quality assurance. The basic form of a rule is,

$$if <conditions> then <conclusion>$$

where <conditions> represent the conditions or premises of a rule, and <conclusion> represent its conclusion or consequence.

A Rule Based System (RBS) consists of a knowledge base and an inference engine. In addition to its static memory for facts and rules, an RBS uses a *working-memory* to store temporary assertions. These assertions record earlier rule-based inferences. Some times, the patterns specify the *actions* rather than *assertions*, e.g., "to put them on the table". In such case the rule based system is called *reaction system*.

The rule based systems are two types: *forward chaining* (data driven) and *backward chaining* (goal) driven. In gaol driven system, one starts from the hypothesis (goal) and tries to prove it by finding the subgoals, then these subgoal becomes the goals, and so on. In data driven system, we search from the data and search the goal. In both the cases one needs to perform the rule chaining. While forward chaining allows conclusions to be drawn only from a given knowledge base, a backward-chained rule interpreter is suitable for requesting still unknown facts. The other types of reasoning systems are: Case based system (CBS) and Model based systems (MBS).

When we are doing data-directed reasoning, we may not like to fire all of the rules in case more than one rule is applicable. For this, some kind of conflict resolution is necessary for arriving at the most appropriate rule(s) to fire. Some conflict resolving strategies may be: *Order of listing*, *recency*, *specificity*, *syntactic structures*, etc.

The criteria for reasoning process can be: similarity between rules, applicability of new rules, and Indexing.

Whenever the rules are such that typical set of facts relate to large number of conclusions, i.e., the *fan-out* is larger than *fan-in*, it is better to use backward chaining. On the other hand, when the rules are such that when a typical hypothesis can lead to many facts, i.e., *fan-in* is larger than *fan-out*, it is better to use forward chaining.

The *case based reasoning* uses specific database for problem solving. A medical practitioner, who uses medical history of failed as well as successful cases for diagnosis of future patients. If a new patient's case appears similar to one of the earlier patient's successful case treatment case, the knowledge history of old case is useful. At the same time if some old case has failed, that will not be tried again due to available history.

## Exercises

1. List the various components of a Production System, and explain each.
2. In the above context, explain the importance of "binding", "matching" and "conflict resolution".
3. Suppose there is a production system with four initial facts: $A$, $B$, $C$, $D$ and following three rules:

$$R_1 : \; if \; A \; then \; E$$
$$R_2 : \; if \; B \wedge F \; then \; G$$
$$R_3 : \; if \; C \wedge E \; then \; F$$

   a. Using these rules and facts, explain what is meant by "backward chaining" and show explicitly how it can be used to determine the truth of $G$?
   b. Explain what is meant by "forward chaining", and show explicitly how it can be used in this case to determine new facts.

4. Consider the following possibilities, suggest the solution strategy to be adopted if the system is implemented as a rule based system:

   a. A subgoal literal is generated such that the higher goal is a subset of the subgoal.
   b. A subgoal literal is generated whose negation unifies with the higher-goal.
   c. A subgoal $S$ literal is generated that is equal to another goal $G$, and $G$ is neither higher nor lower to $S$.

5. Translate the following knowledge base into predicate form.

   If $x$ is on top of $y$ then $y$ supports $x$.
   If $x$ is above $y$ and they are touching each other then $x$ is on top of $y$.
   A phone is above a book.
   A phone is touching a book.

   a. Use forward-chaining to show that the predicate "supports(book, phone)" is true.
   b. Use forward-chaining to show that the predicate "supports(book, phone)" is true.

   Count the number of triggering and rule firing in each of the above cases.

6. Given the propositions $A$, $B$, $C$, $D$ and rules $R_1$ to $R_4$,

$$R_1 = if \; A \wedge X \wedge Y \; then \; Z$$
$$R_2 = if \; B \wedge V \; then \; Y$$
$$R_3 = if \; C \wedge V \; then \; X$$
$$R_4 = if \; D \; then \; C$$

   a. Use the forward-chaining to determine if $Z$ can be inferred from the above knowledge base.
   b. Use the backward-chaining to determine if goal $Z$ succeeds from the above knowledge base.

7. What are the conflict resolving strategies in case more than one rule matches with the assertions? Discuss the merits and demerits of each strategy for selection of a rule to fire.

8. Briefly justify each of the following for conflict resolution. Also, give examples in each case.

   a. Specificity criteria.
   b. Syntactically longest rule first.
   c. Ordering rules.
   d. Most recent first.
   e. In order of increasing size of the rule.

9. List the following formulas in oder of specificity, and construct a tree such that those having same specificity will stand at the same level in the tree.

   a. $b \vee c$
   b. $b \wedge c$
   c. $a \vee c$
   d. $a \wedge c$
   e. $a \wedge b$
   f. $a \wedge b \wedge c$
   g. $b$
   h. $c$

10. How the assignment of priority to rules can be implemented? Consider the Table 4.2 to demonstrate the assignment of suitable priorities to rules listed in this table. Suggest the structure as how the priorities are to be stored and rules be invoked based on these priorities.

11. Explain what structures can be used and how they will operate, for knowledge based systems, for the following functionalities?

   a. If a precondition of one or more rules is false, the system should exclude these rules from participation in the inference process.
   b. If firing of some rules infer new knowledge, this should be added into the existing knowledge.
   c. To maintain and use index file for rules.

12. For the rule (4.2), if domain size of $p$ is $|p| = m$, domain size of variable $x$ is $n$, and of $y$ is $l$,

   a. Find out the worst-case time complexity of the rule to be selected,
   b. In how many ways ordering can be done?
   c. Justify that it is *NP-hard* problem as a general case.

13. Which type of rule chaining you will prefer in the following situations? Also, identify, what is the goal in each case.

    a. To diagnose the case of malaria infection, based on the symptoms of cough, soar throat, regular fever, shivering.
    b. Identify a thief, based on the nature of theft, finger prints, and goods stolen.

14. ] Suggest an approach to combine the forward and backward chaining for the knowledge base shown in Table 4.2 (Hint: Try backward chaining to begin with, if it is not doing efficiently, then switch over to forward chaining.)

15. To prove a theorem of geometry using rule-based systems, represent the following statements as production rules:

    a. Corresponding sides of two congruent triangles are also congruent.
    b. Corresponding angles of two congruent triangles are also congruent.
    c. If corresponding sides of two triangles are congruent then the triangles are congruent.
    d. If corresponding sides and the angle covered by them are equal then the triangles are congruent.
    e. Base angles of an isosceles triangle are congruent.

16. In what order the rules should be fired for inference efficiency? Discuss the merits and demerits of choosing a particular order of rule firing.

17. What are the criteria of selection of premises for firing the rules? Discuss this based on specificity and generality of the premises.

18. Consider a network of applicability of rules. Imagine that due to firing of a rule, a new inference is generated. This inference may contradict some fact or goal. Explain, what should be the structure of dependency network so that any inconsistency caused due to new inference is taken care of.

19. Suggest the strategy, as how a tree like rules structure can be used for reasoning in forward direction. Explain or suggest your strategy for following:

    a. Whether the breadth-first or depth-first search is better for rule searching?
    b. Whether the top-down or bottom-up search is better for firing of rule sequences?
    c. How to eliminate one or both the rules, say $R_1$ and $R_2$, for consideration, if one of the precondition is false, which is common in both $R_1$ and $R_2$?
    d. In the begin of a reasoning process whether you would prefer to choose to fire a rule with large number of preconditions or small?

20. Analyze the Algorithm 4.1 and justify that it is NP-hard as a general case, however, polynomial in real world situation.

21. Analyze the complexity of preconditions in the forward reasoning process.

22. How the following situations are implemented in knowledge base for rule-chaining? Explain.

    a. "Is the pain intensifying with respirations?"

      b. "Did shortening of breadth occur before throat pain?"

23. Given the knowledge base of rule-chaining of Table 4.2, show an analysis as, in general, which type, forward or backward search strategy is better? Also, justify your claim.

24. Why, a rule-based system having combination of forward and backward rule-chaining is more complex than either of these both?

# References

1. Allen BP (1994) Case-based reasoning: business applications. Commun ACM 37(3):40–42
2. Patterson DW (2001) Introduction to artificial intelligence and expert systems. PHI India
3. Hayes-Roth F (1985) Rule-based systems. Commun ACM 28(9):921–922
4. Puppe F (1993) Systematic introduction to expert systems. Springer-Verlag
5. McDermott JP (1980) R1: A rule-based configurator of computer systems—technical report CMU-CS-80-119, 1–56

# Chapter 5
# Logic Programming and Prolog

**Abstract** Prolog is logic programming languages for AI, based on predicate logic. This chapter discusses the structure, syntax, and semantics of Prolog language, provides comparison with procedural language like C, interpretation of predicate logic and that of Prolog, both formally as well through worked out examples, and explain how the recursion is definition as well solution of a problem, and explains with simple examples as how the control sequencing takes place in Prolog. Use of two open source compilers of prolog using simple worked out examples is demonstrated. Each concept of Prolog and logic programming is explained with the help of worked out examples. At the end, a brief summary gives glimpse of the chapter, followed with number of exercises to strengthen the learning.

**Keywords** Logic programming · Prolog · Predicate logic · Prolog compiler · Control sequencing · Knowledge base · Query · Horn clause · Recursion · Rule-chaining · Backward chaining · Forward chaining · List · Cut · Fail

## 5.1 Introduction

This chapter presents the basic concepts of logic programming, and Prolog language. Prolog is a logic programming language, implemented in two parts:

1. *Logic*, which describes the problem, and
2. *Control*, provides the solution method.

This is in contrast to procedural programming languages, where description and solution go together, and are hardly distinguishable. This, feature of prolog helps in separate developments for each part, one by the programmer and other by implementer.

PROLOG is a simple, yet powerful programming language, based on the principles of first-order predicate logic. The name of the language is an acronym for the French 'PROgrammation en LOGique'(*pro*gramming in *log*ic). PROLOG was designed by A. Colmerauer and P. Roussel at the University of Marseille (Canada), around 1970. The PROLOG has remained connected with a new programming style, known as

*logic programming*. Prolog is useful in problem areas, such as artificial intelligence, natural language processing, databases, etc., but pretty useless in others, such as graphics or numerical computations. The main applications of the language can be found in the area of Artificial Intelligence; but PROLOG is being used in other areas in which symbol manipulation is of prime importance. Following are the application areas.

- Computer algebra.
- Design of parallel computer architectures.
- Database systems.
- Natural-language processing.
- Compiler construction.
- Design of expert systems.

The popular compilers of prolog are *swi-prolog* and *gnu prolog*; both are available in open source, and runs on Windows and Linux platforms.

**Learning Outcomes of this Chapter**:

1. Prolog versus procedure-oriented languages. [Assessment]
2. Working with Prolog, using gprolog and swi-prolog compilers. [Usage]
3. Writing small prolog programs and running. [Usage]
4. Translating predicate logic into Prolog. [Usage]
5. Prolog Syntax and semantics. [Familiarity]
6. Forward-chaining versus back-ward chaining. [Assessment]
7. Using backward-chaining for reasoning and inference in Prolog. [Assessment]

## 5.2   Logic Programming

In conventional languages, like, C, C++, and Java, a program is a description of sequence of instructions to be executed one-by-one by a target machine to solve the given problem. The *description* of the problem is *implicitly* incorporated in this instruction sequence, where usually it is not possible to clearly distinguish between the *description* (i.e., logic) of the problem, and the *method* (i.e., control) used for its solution.

In logic programming language, like Prolog, the description of the problem and the method for its solution are explicitly separated from each other. Hence, in an algorithm of a Prolog program, these two parts are distinctly visible, and can be expressed by [6]:

$$\boxed{Algorithm = Logic + Control} \tag{5.1}$$

In the above equation, term 'logic' represents the descriptive part of an algorithm, and 'control' represents the solution part, which takes the description as the point

**Fig. 5.1** Logic
programming
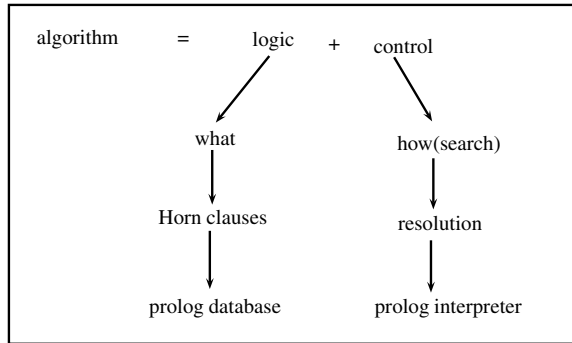


of departure. In other words, the logic component defines what the algorithm is
supposed to do, and the control component indicates how it should be done.

For solution through logic program, a problem is always described in terms of
relevant objects and relations between them. These relations are represented in a
*clausal* form of logic—a restricted form of first-order predicate logic. The logic
component for a problem is called a *logic program*, while the control component
comprises method for logical deduction (reasoning) for deriving new facts from the
logic program. This results to solving a problem through deduction. The deduction
method is designed as a general program, such that it is capable of dealing with any
logic program that follows the clausal form of syntax.

There are number of advantages of splitting of an algorithm into a logic component
and a control component:

- We can develop these two components of the program independent of each other.
  For example, when writing the description of the problem as logic, we do not have
  to be familiar with how the control component operates for that problem; thus
  knowledge of the declarative reading of the problem specification suffices.
- A logic component may be developed using a method of stepwise refinement; we
  have only to watch over the correctness of the specification.
- Changes to the control component affect (under certain conditions) only the effi-
  ciency of the algorithm; they do not influence the solutions produced.

The implementation of Logic programming is explained in Fig. 5.1.

When all facts and rules have been identified, then a specific problem may be
looked upon as a query concerning the objects and their interrelationships. In sum-
mary, to provide specification of a logic program amounts to following jobs:

- specify the *facts* about the objects and relations between them for the problem
  being solved;
- specify the *rules* about the objects and their interrelationships;
- specify the *queries* to be posed concerning the objects and relations.

An algorithm for logic program can be shown to be decomposed into the compo-
nents shown in Fig. 5.2.

**Fig. 5.2**  Components of
logic program
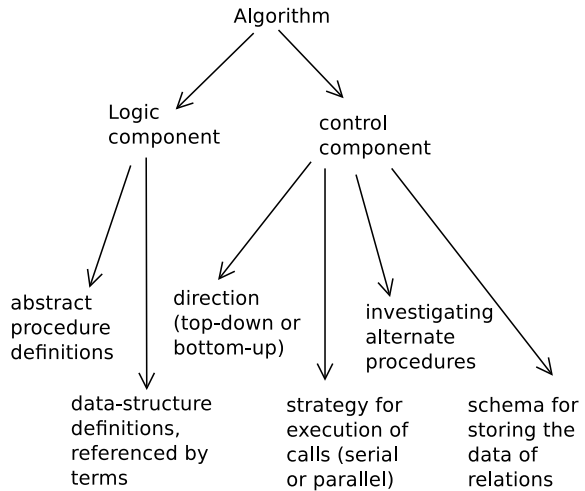


## 5.3  Interpretation of Horn Clauses in Rule-Chaining

Horn clause is restricted form of a predicate logic sentence. A typical representation
of a problem in Horn clause form is:

1. a set of clauses defining a problem domain and,
2. a theorem consisting of: (a) hypotheses represented by assertions $A_1 \leftarrow, \ldots,$
   $A_n \leftarrow$ and (b) a conclusion in negated form and represented by a denial $\leftarrow$
   $B_1, \ldots, B_m$.

The reasoning process can be carried out as back-ward reasoning, or forward rea-
soning. In backward-chaining, reasoning is performed backwards from the conclu-
sion, which repeatedly reduces the goals to subgoals until ultimately all the subgoals
are solved directly in the form of original assertions.

In the case of problem-solving using forward-chaining approach, we reason for-
wards from the hypotheses, and repeatedly derive new assertions from old ones until
eventually the original goal is solved directly by derived assertions [6].

For our reasoning using forward and backward chaining, we consider the family-
tree of *Mauryan Dynasty* (India) as shown in Fig. 5.3 [5].

The problem of showing that *chandragupta* is a grandparent of *ashoka* can be
solved either backward-chaining or forward-chaining. In forward-chaining, we start
with the following assertions:

*father(bindusara, asoka)* $\leftarrow$
*father(chandragupta, bindusara)* $\leftarrow$

Also, we shall use the clause $parent(x, y) \leftarrow father(x, y)$ to derive new
assertions,

**Fig. 5.3**  Mauryan dynasty
family-tree



Chandragupta
(Grandfather)

Durdhara
(Grandmother)

Bindusara
(father)
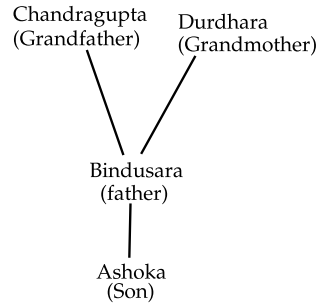
Ashoka
(Son)

$parent(chandragupta, bindusara) \leftarrow$
$parent(bindusara, ashoka) \leftarrow$

Continuing forward-chaining we derive, from the definition of grandparent, the new assertion,

$grandparent(chandragupta, ashoka) \leftarrow$

which matches the original goal.

Reasoning using backward-chaining, we start with the original goal, which shows that chandragupta is a grandparent of ashoka,

$\leftarrow grandparent(chandragupta, ashoka)$

and use the definition of grandparent to derive two new subgoals,

$\leftarrow parent(chandragupta, z), parent(z, ashoka),$

by denying that any $z$ is both a child of chandragupta and a parent of ashoka. Continuing backward-chaining and considering both subgoals (either one at a time or both simultaneously), we use the clause,

$$parent(x, y) \leftarrow father(x, y)$$

to replace the subproblem $parent(chandragupta, z)$ by $father(chandragupta, z)$ and the subproblem $parent(z, ashoka)$ by $father(z, ashoka)$. The symbol "$\leftarrow$" is read as "if". The newly derived subproblems are solved compatibly by assertions which determine "bindusara" as the desired value of $z$.

In both the backward-chaining and forward-chaining solutions of the grandparent problem, we have mentioned the derivation of only those clauses which directly contribute to the ultimate solution. In addition to the derivation of relevant clauses, it is often unavoidable, during the course of searching for a solution, to derive assertions or subgoals which do not contribute to the solution. For example, in the forward-chaining search for a solution to the grandparent problem, it is possible to derive the irrelevant assertions as,

$parent(durdhara, bindusara) \leftarrow$
$male(chandragupta) \leftarrow$

Also, in backward-chaining search it is possible to replace the subproblem,

$parent(chandragupta, z)$

by the unsolvable and irrelevant subproblem,

$mother(chandragupta, z).$

There are proof procedures which understand logic in backward-chaining, e.g., model elimination, resolution, and interconnectivity graphs. These proof procedures operate with the clausal form of predicate logic and deal with both Horn clauses and non-Horn clauses. Among clausal proof procedures, the connection graph procedure is able to mix backward and forward reasoning.

The terminology we used here—the backward-chaining, is also called "top-down". Given a grammar formulated in clausal form, top-down parsing algorithm generates a sentence to its original form, i.e., the assertions. The forward-chaining is also called "bottom-up", where we start from the assertions and try to reach to the goals.

## 5.4  Logic Versus Control

Different control strategies for the same logical representation generate different behaviors. Also, the information about a problem-domain can be represented using logic in different ways. Alternative representations can have a more significant impact on the efficiency of an algorithm compared to alternative *control strategies* for the same representation.

Consider the problem of sorting a list $x$ and obtaining the list $y$. In our representation, we can have a definition with an assertion consisting of two arguments: "$y$ is permutation of $x$", and "$y$ is ordered", i.e.,

*sorting x gives y $\leftarrow$ y is a permutation of x, y is ordered.*

Here "$\leftarrow$" is read as 'if' and ',' is logical *AND* operator. The first argument generates permutations of $x$ and then it is tested whether they are ordered. Executing procedure calls as coroutines, the procedure generates permutations, one element at a time. Whenever a new element is generated, the generation of other elements is suspended while it is determined whether the new element preserves the orderedness of the existing partial permutation.

A program consists of a logic component and a control component. The logic component defines the part of the algorithm that is problem specific, which not only determines the meaning of algorithm but also decides the way the algorithm behaves. For systematic development of well-structured programs using successive refinements, the logic component needs to be defined before the control component.

Due to these two components of a logic program, the efficiency of an algorithm can be improved through two different approaches: 1. by improving the efficiency of logic component, 2. through the control component. Note, that both improvements are additive, and not as alternative choices.

In a logic programs, the specification of control component is subordinate to logic component. The control part can be explicitly specified by the programmer through a separate control specification language, or the system itself can determine the control. When logic is used like in relational calculus to specify queries (i.e., higher level of language) to knowledge base, the control component is entirely determined by the system. Hence, for higher level of programming language, like queries, lesser effort is required for programming the control part, because in that case the system assumes more responsibility about efficiency, as well as to exercise control over the use of given information.

Usually, a separate control-specifying language is preferred by advanced programmers to exercise the control with higher precision. A higher system efficiency is possible if programmer can communicate to system a more precise information to have finer control. Such information may be a relation, for example, $F(x, y)$, where $y$ can also be a function of $x$. This function could be used by a *backtracking interpreter* to avoid looking for another solution to the first goal in the goal statement. This can be expressed, for example by,

$$\leftarrow F(A, y), G(y),$$

when the second goal fails. Other example of such information can be that one procedure,

$$S \leftarrow T$$

may be more likely to solve problem $S$ than another procedure,

$$S \leftarrow R.$$

This kind of information of cause-effect relation is common in fault diagnosis where, on the basis of past experience, it might be possible to estimate that symptom $S$ is more likely have been caused by $T$ rather than by $R$.

In the above examples, the control information is problem-specific. However, if the control information is correct, and the interpreter is correctly implemented, then the control information should not affect the meaning of the algorithm, which is decided by the corresponding logic component of the program.

### 5.4.1  Data Structures

In a well-structured program it is desirable to keep data structures separate from procedure that interact with the data structure. Such separation means data structures can be manipulated without altering the procedure. Usually, there is need of alteration of data structures—some times due to the need of change of requirements and other

times with an objective to to improve the algorithm by replacing data structure by a more efficient one. In large and complex programs, often the information demand made on the data structures are only determinable in the final stages of the program design. If data structures are separated from the procedures, the higher level procedures can be written before the data structures are finalized, and these procedures can be altered conveniently later any time without effecting the data structure [6].

In Prolog, the data structures of a program are already included in the logic component of the program. Consider, for example, the data structure "Lists", which can be represented by following terms:

- *nil*; empty list,
- *cons*(*x*, *y*); list with first element *x*, and *y* is another list.

Hence, the following example names a three-element list consisting of individuals as 2, 1, 3 in that order.

$$cons(2, cons(1, cons(3, nil)))$$

**Example 5.1**  A logic program for quick-sort.

In *quick-sort*, the predicates *empty*, *first*, *rest*, *partitioning*, and *appending*, can be defined independently from the definition of *sorting* (see Eqs. 5.2 and 5.3). For this definition we assume that, partitioning of a list $x_2$ by element $x_1$ gives a list *u* comprising the elements of $x_2$ that are less or equal to $x_1$, and a list *v* of elements of $x_2$ that are greater than $x_1$.

$$sorting\ x\ gives\ y\ \leftarrow x\ is\ empty,\ \ y\ is\ empty \qquad (5.2)$$

$$
\begin{aligned}
sorting\ x\ gives\ y \leftarrow &first\ element\ of\ x\ is\ x_1,\ rest\ is\ x_2,\\
&partitioning\ x_2\ by\ x_1\ gives\ u\ and\ v,\\
&sorting\ u\ gives\ u',\\
&sorting\ v\ gives\ v',\\
&appending\ w\ to\ u'gives\ y,\\
&first\ element\ of\ w\ is\ x_1,\\
&rest\ of\ w\ is\ v'. \qquad (5.3)
\end{aligned}
$$

The *data-structure-free* definition of *quicksort* interacts with the data structure of lists through the following definitions:

*nil is empty* ←
*first element of cons*(*x*, *y*)*is x* ←
*rest of cons*(*x*, *y*)*is y* ←

If the predicates: *empty*, *first*, *rest*, are dropped from the definition of *quick-sort*, and instead the preliminary forward-chaining/forward-chaining deduction is used, then the original data-structure-free definition of quick-sort can be replaced by a definition that mixes the data structures with the procedures,

$$sorting \; nil \; gives \; nil \leftarrow$$

$$sorting \; cons(x_1, x_2) \; gives \; y \leftarrow partitioning \; x_2 \; by \; x_1 \; gives \; u, v,$$
$$sorting \; u \; gives \; u',$$
$$sorting \; v \; gives \; v',$$
$$appending \; list \; cons(x_1, v') \; to \; u' gives \; y.$$

□

There is another advantage of data-structure-independent definition: with well-chosen interface procedure names, data-structure-independent programs are virtually self-documenting. In conventional programs that mix procedures and data structures, the programmer needs to provide separate documentation for data structures.

On the other hand, in-spite of the arguments in support for separating procedures and data structures, the programmers usually mix them together for the sake of run-time efficiency.

## 5.4.2 Procedure-Call Execution

In a simplest backward reasoning based execution, the procedure calls are executed one at a time, in the order they have been written. Typically, an algorithm can be made to run faster by executing the same procedure calls in the form of *coroutines* or as *communicating parallel-processes*. Consider an algorithm $A_1$,

$$A_1 = L + C_1 \tag{5.4}$$

where logic component is $L$ and control component is $C_1$. Assume that from $A_1$ we have obtained a new algorithm $A_2$, having logic component $L$ and control component $C_2$,

$$A_2 = L + C_2 \tag{5.5}$$

where we replaced control strategy $C_1$ by new control strategy $C_2$, and the logic $L$ of the algorithm remains unchanged. For example, executing procedure calls in sequence, the procedure,

$$sorting \; x \; gives \; y \leftarrow y \; is \; a \; permutation \; of \; x, \; y \; is \; ordered, \tag{5.6}$$

first generates permutations of $x$ and then tests whether they are ordered. By executing procedure calls as *coroutines*, the procedure generates permutations, one element at a time. Whenever a new element is generated, the generation of other elements is suspended while it is determined whether the new element preserves the orderedness of the existing partial permutation.

In the Similar way to procedure (5.6), procedure calls in the body of *quick-sort* can be executed (either as coroutines or as parallel/sequential processes). When they are executed in parallel, partitioning the rest of $x$ can be initiated as soon as the first element of the rest are generated (see procedure (5.3) on p. 118). Note that the sorting for $u$ and $v$ can take place in parallel, the moment first elements of $u$ and $v$ are generated. And, the appending of lists can start soon after the first elements of $u'$, and the sorted output of $u$, are made available.

The high level language SIMULA provides the facility of writing both the usual sequential algorithms, as well as algorithms with coroutines. Here, the programmer can provide the controls about: When the coroutines should be suspended and resumed? However, in this language, the logic and control are inextricably intertwined in the program text, like in other procedure oriented languages. Hence, the control of an algorithm cannot be modified without rewriting the entire program.

In one way, the concept of separating logic and controls is like separating data structures and procedures. When a *procedure* is kept separate from a *data structure*, we are able to distinguish clearly as what functions are fulfilled by which data structures. On the other hand, when *logic* is separated from *control*, it becomes possible to distinguish, what the program (i.e., *logic*) does, and how it does it (i.e., controlling takes place). In both conditions it becomes obvious as what the program does, and hence it can be more easily determined whether it correctly does the job it was intended for.

### 5.4.3  Backward Versus Forward Reasoning

Recursive definitions are common in mathematics, where they are more likely to be understood as forward-chaining rather than backward-chaining. Consider, for example, the definition of factorial, given below.

$$factorial \ of \ 0 \ is \ 1 \leftarrow \tag{5.7}$$

$$
\begin{aligned}
factorial \ of \ x \ is \ u \leftarrow &y \ plus \ 1 \ is \ x, \\
&factorial \ of \ y \ is \ v, \\
&x \ times \ v \ is \ u.
\end{aligned}
\tag{5.8}
$$

The mathematician is likely to understand such a definition forward-chaining, generating the sequence of assertions as follows:

$$
\begin{aligned}
&factorial \ of \ 0 \ is \ 1 \leftarrow \\
&factorial \ of \ 1 \ is \ 1 \leftarrow \\
&factorial \ of \ 2 \ is \ 2 \leftarrow \\
&factorial \ of \ 3 \ is \ 6 \leftarrow \\
&\qquad\qquad and \ so \ on.
\end{aligned}
$$

Conventional programming language implementations understand recursions as backward-chaining. Programmers, accordingly, tend to identify recursive definitions with backward-chaining execution. However, there is one exception, and that is Fibonacci series, which is efficient when interpreted as forward reasoning. It is left as an exercises for the students to verify the same.

### 5.4.4  Path Finding Algorithm

Consider an algorithm $A$, with $C_1$, $C_2$ control components, and $L_1$, $L_2$ as logic components, which can often be analyzed in different ways [6].

$$A = L_1 + C_1 = L_2 + C_2. \tag{5.9}$$

Some of the behavior determined by $C_1$ in one analysis might be determined by the logic component $L_2$ in another analysis. This has significance for understanding the relationship between programming style and execution facilities. In the short term sophisticated behavior can be obtained by employing simple execution strategies and by writing complicated programs. In the longer term the same behavior may be obtained from simpler programs by using more sophisticated execution strategies.

A path-finding problem illustrates a situation in which the same algorithm can be analyzed in different ways. Consider the problem of finding a path from vertex $a$ to vertex $z$ in the directed graph shown in Fig. 5.4.

In one analysis, we can employ a predicate $go(x)$ which states that it is possible to go to $x$. The problem of going from $a$ to $z$ is then represented by two clauses. One asserts that it is possible to go to $a$. The other denies that it is possible to go to $z$. The arc directed from $a$ to $b$ is represented by a clause which states that it is possible to go to $b$ if it is possible to go to $a$. Different control strategies determine different path-finding algorithms. Forward search from the initial node $a$ is forward-chaining based reasoning from the initial assertion $go(a) \leftarrow$ (see Table 5.1). Backward search from the goal node $z$ is backward reasoning from the initial goal statement $\leftarrow go(z)$ (see Table 5.2).

**Fig. 5.4**  Directed graph path finding
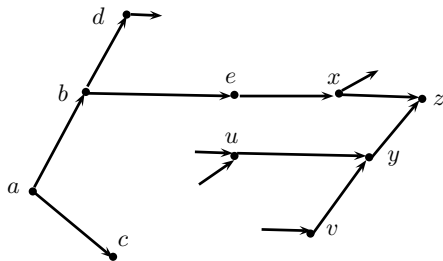
**Table 5.1** Forward-chaining

$$
\begin{aligned}
&go(a) \leftarrow \\
&go(b) \leftarrow go(a) \\
&go(c) \leftarrow go(a) \\
&go(d) \leftarrow go(b) \\
&go(e) \leftarrow go(b) \\
&go(x) \leftarrow go(e) \\
&go(z) \leftarrow go(x) \\
&\ldots
\end{aligned}
$$

**Table 5.2** Backward-chaining

$$
\begin{aligned}
&\leftarrow go(z) \\
&go(z) \leftarrow go(x) \\
&go(z) \leftarrow go(y) \\
&go(x) \leftarrow go(e) \\
&go(y) \leftarrow go(u) \\
&go(y) \leftarrow go(v) \\
&go(e) \leftarrow go(b) \\
&go(b) \leftarrow go(a) \\
&\ldots
\end{aligned}
$$

Carrying out a bidirectional search from both the start node and the goal node results to a combination of backward and forward reasoning. In that case, whether a path-finding algorithm investigates one path at a time (in depth-first) or develops all paths together (in breadth-first) will depend on search strategy used.

## 5.5   Expressing Control Information

The distinction between backward-chaining and forward-chaining based execution can be expressed in a graphical notation using arrows to indicate the flow of control. The same notation can be used to represent different combinations of backward-chaining and forward-chaining based execution. The notation does not, however, aim to provide a complete language for expressing useful control information.

The arrows are attached to atoms in clauses to indicate the direction of transmission of processing activity from one clause to other clause. For every pair of matching atoms in the initial set of clauses (i.e., one atom in the conclusion of a clause and the other among the conditions of a clause), and there is an arrow directed from one atom to the other. For backward-chained reasoning, arrows are directed from goals to assertions. For the grandparent problem, we have the graph shown in Fig. 5.5.

$$\leftarrow grandparent(chandragupta, ashoka)$$

$$grandparent(x, y) \quad \leftarrow \quad parent(x, z), parent(z, y)$$

$$parent(x, y) \quad \leftarrow \quad father(x, y) \qquad parent(x, y) \quad \leftarrow \quad mother(x, y)$$

$$father(chandragupta, bindusara) \quad \leftarrow$$

$$father(bindusara, ashoka) \quad \leftarrow$$

$$mother(durdhara, bindusara) \leftarrow$$
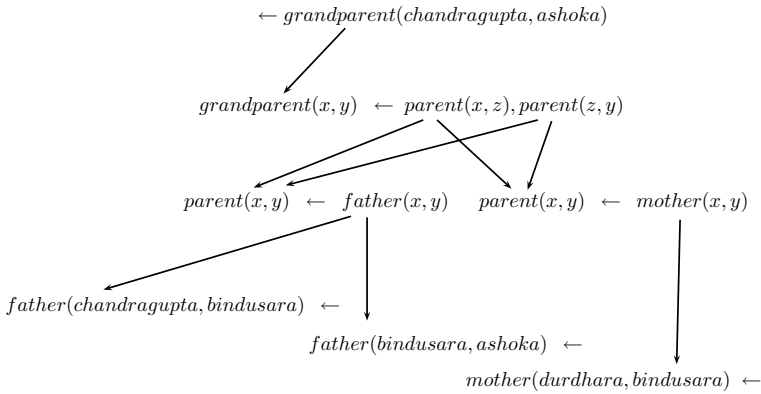
**Fig. 5.5**  Control-flow for backward-chaining

$$\leftarrow grandparent(chandragupta, ashoka)$$

$$grandparent(x, y) \leftarrow \quad parent(x, z), parent(z, y)$$

$$parent(x, y) \quad \leftarrow \quad father(x, y) \qquad parent(x, y) \quad \leftarrow \quad mother(x, y)$$

$$father(chandragupta, bindusara) \leftarrow$$

$$father(bindusara, ashoka) \leftarrow$$

$$mother(durdhara, bindusara) \leftarrow$$
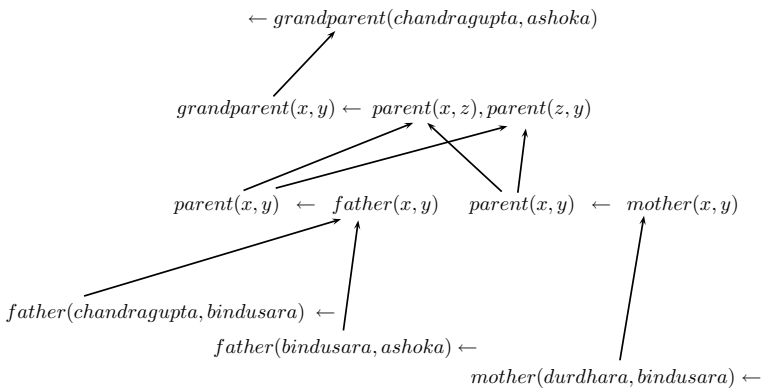
**Fig. 5.6**  Control-flow for forward-chaining

A processing activity in backward-chaining in this figure is shown as starting with the initial goal statement, and transmits activity to the body of the *grandparent* procedure, whose procedure calls, in turn, activate the parenthood definitions. At the end, the assertions provided in the knowledge base passively accepts processing activity, and does not further transmit it to other clauses.

For forward-chaining execution, arrows are directed from assertions to goals (see Fig. 5.6). The processing activity originates with the database of initial assertions (i.e., bottom of the graph in this figure). They transmit activity to the parenthood definitions, which, in turn, activate the grandparent definition. Processing terminates when it reaches all the conditions in the passive initial goal statement, at the top of the graph [6].

The grandparent definition can be used in a combination of backward-chaining and forward-chaining methods. Using numbers to indicate sequencing, we can

**Fig. 5.7** Combination of
logic and control-I

$$3$$
$$\uparrow$$
$$grandparent(x,y) \; \leftarrow \; parent(x,z), parent(z,y)$$
$$\uparrow \qquad \qquad \downarrow$$
$$1 \qquad \qquad 2$$
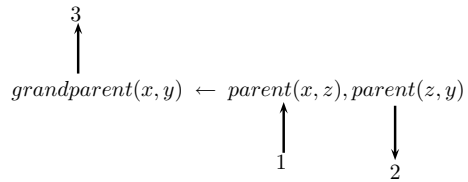
**Fig. 5.8** Combination of
logic and control-II

$$1$$
$$\downarrow$$
$$grandparent(x,y) \; \leftarrow \; parent(x,z), parent(z,y)$$
$$\uparrow \qquad \qquad \downarrow$$
$$2 \qquad \qquad 3$$
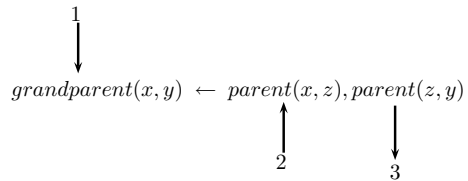
represent different combinations of backward-chaining and forward-chaining reasoning. For the sake of simplicity, we only show the control notation associated with the grandparent definition.

The combination of logic and control indicated in Fig. 5.7 represents the algorithm, with markers 1–3, having following sequence of operation:

1. indicates that the algorithm waits until $x$ is asserted to be the parent of $z$, then
2. indicates that the algorithm finds a child $y$ of $z$, and finally
3. indicates that the algorithm asserts that $x$ is grandparent of $y$.

The combination indicted by Fig. 5.8, which represents the algorithm, has marking 1–3 as events, with following meanings:

1. this event waits until $x$ is asserted to be parent of $z$, then
2. this event waits until it is given the problem of showing that $x$ is grandparent of $y$,
3. this event attempts to solve by showing that $z$ is parent of $y$.

In the similar way the the algorithms takes care of rest of the controls of backward-chaining and forward-chaining.

## 5.6  Running Simple Programs

Program files can be compiled using the predicate *consult*. The argument has to be a Prolog atom denoting the particular program file. For example, to compile the file socrates.pl, submit the following query to swi-Prolog [8]:

```
?- consult(socrates.pl).
```

If the compilation is successful, Prolog will reply with 'Yes'. Otherwise a list of errors is displayed. The 'swi-prolog' can also run in GUI environment in Windows.

The gnu-prolog running on linux can run a prolog program 'socrates.pl' as follows [3]:

```
$ gprolog <enter>
GNU Prolog 1.3.0
Copyright ....
| ?-[socrates]. % filename without extensions.
```

Let us demonstrate to run a prolog program and perform interpretation of the clauses by backward-chaining. For this, we consider our single old problem of "socrates" and application of inference rule of *modus ponens*.

**Example 5.2**  Demonstrating Backtracking.

> All men are mortal.
> Socrates is a man.
> ————————————
> Therefore, Socrates is mortal.

In terms of Prolog, the first statement corresponds to the rule: $X$ is mortal, if $X$ is a man (i.e., for every $X$). The second statement corresponds to the fact: 'Socrates is a man'. Note that 'socrates' is constant (literal), and $X$ is a variable. The above rule and fact can be written in Prolog language syntax as,

```
mortal(X) :- man(X).
man(socrates).
```

The conclusion of the argument is: "Socrates is mortal," which can be expressed in predicate as 'mortal(socrates)'. After we have compiled, we run the above program, and query it, as follows:

```
?- mortal(socrates).
Yes
```

We notice that Prolog agrees with our own logical reasoning. But how did it come to its conclusion? Let's follow the goal execution step-by-step [7].

1. The query $mortal(socrates)$ is designated the initial goal.
2. Scanning through the clauses of this program, Prolog tries to match $mortal(socrates)$ with the first possible fact or head of rule. It finds $mortal(X)$—head of the first (and only) rule. When matching the two 'socrates'is bound to $X$, with unifier $\{socrates/X\}$.
3. The variable binding is extended to the body of the rule, i.e. $man(X)$ becomes $man(socrates)$.
4. The newly instantiated body becomes our new subgoal: $man(socrates)$.

5. Prolog executes the new subgoal by again trying to match it with a rule-head or a fact.
6. Obviously, new subgoal $man(socrates)$ matches the fact $man(socrates)$, and current sub-goal succeeds.
7. This, means that the initial goal succeeds, and prolog responds with 'YES'.    □

We can observe the trace of sequences operations of interpretations by running it.

Prolog is a declarative (i.e., descriptive) language. Programming in Prolog means describing the *world*. Using such programs means asking questions about the previously described world. The simplest way of describing the world is by stating facts, like "train is bigger than bus", as,

```
bigger(train, bus).
```

The following example demonstrates this [2].

**Example 5.3** Knowledge base about sizes of transports.

Let's add a few more facts to vehicles of transport as:

```
bigger(train, bus).
bigger(bus, car).
bigger(car, bicycle).
bigger(car, motorbike).
```

This is a syntactically correct program, and after having compiled it, we can ask the Prolog system questions (or *queries*) about it.

```
?- bigger(car, bicycle). <enter>
Yes
```

The query 'bigger(car, bicycle)' (i.e. the question "Is a car bigger than a bicycle?") succeeds, because the fact 'bigger(car, bicycle)' was previously communicated to the Prolog system. Our next query is, "is a motorbike bigger than an train?"

```
?- bigger(train, motorbike).
No
```

The reply by Prolog is "No". The reason being that, the program says nothing about the relationship between train and motorbike. However, we note that, the program says—'trains are bigger than bus', and 'buses are bigger than cars', which in turn are bigger than motorbike. Thus, trains are also to be bigger than motorbikes. In mathematical terms, the bigger-relation is *transitive*. But it also not been defined in our program. The correct interpretation of the negative answer Prolog is that: "from the information communicated to the program it cannot be proved that an

train is bigger than a motorbike". As an exercise, we can try the proof by resolution refutation, but it cannot be proved because it is not possible to verify the statements.

Solution would be to define a new relation, which we will call *isbigger*, as the *transitive closure*. Animal *X* is bigger than *Y*, if this has been stated as a *fact*. Otherwise, there is an animal *Z*, for which it has been stated as a fact that animal *X* is bigger than animal *Z*, and it can be shown that animal *Z* is bigger than animal *Y*. In Prolog such statements are called *rules* and are implemented as follows:

```
isbigger(X, Y) :- bigger(X, Y).                %rule1
isbigger(X, Y) :- bigger(X, Z), isbigger(Z, Y). %rule2
```

where ':-' stands for 'if' and comma (,) between 'bigger(X, Z)' and 'isbigger(Z,Y)' stands for 'AND', and a semicolon (;) for 'OR'. If from now on we use 'isbigger' instead of 'bigger' in our queries, the program will work as intended.

```
?- isbigger(train, motorbike).
Yes
```

In the rule1 above, the predicate 'isbigger(X, Y)' is called *goal*, and 'bigger(X, Y)' is called *sub-goal*. In the rule2 'isbigger(X,Y)' is goal and the expressions after the sign ':-' are called sub-goals. The goal is also called *head* of the rule, and the expressions after sign ':-' is called *body* of the rule statement.

In fact, the rule1 above corresponds to the predicate,

$$if\ bigger(X, Y)\ then\ isbigger(X, Y),$$

or

$$bigger(X, Y) \rightarrow isbigger(X, Y).$$

Similarly, predicate expression for rule2 is

$$bigger(X, Z) \wedge isbigger(Z, Y) \rightarrow isbigger(X, Y).$$

The prolog expressions which are not conditionals, i.e., like,

```
bigger(train, bus).
bigger(bus, car).
bigger(car, bicycle).
bigger(car, motorbike).
```

are called *facts*(or assertions). The facts and rules, together, make the knowledge base (KB) in a program.

For the query 'isbigger(train, motorbike)' the Prolog still cannot find the fact 'bigger(train, motorbike)' in its database, so it tries to use the second rule instead. This is done by matching the query with the head of the rule, which is 'isbigger

(X, Y)'. When doing so, the two variables get bound: $X$ = train, and $Y$ = motorbike. The rule says that in order to prove the goal 'isbigger(X,Y)' (with the variable bindings that's equivalent to isbigger(train, motorbike)), Prolog needs to prove the two subgoals 'bigger(X, Z)' and 'isbigger(Z, Y)', with the same variable bindings. Hence, the rule2 gets transformed to:

$$isbigger(train, motorbike) : -bigger(train, Z),$$
$$isbigger(Z, motorbike).$$

By repeating the process *recursively*, the facts that make up the chain between *train* and *motorbike* are found and the query ultimately succeeds. Our earlier Fig. 5.5 demonstrated the similar chain of actions.

Of course, we can do slightly more exciting job than just asking yes/no-questions. Suppose we want to know, what animals are bigger than a car? The corresponding query would be:

```
?- isbigger(X, car).
```

We could also have chosen any other name in place of $X$ for it as long as it starts with an uppercase letter, which makes it a variable. The Prolog interpreter replies as follows:

```
X = bus;     % press here ';' to get another match
X = train ;                              if exists.
No
```

There are many more ways of querying the Prolog system about the contents of its database.

For example, try to find out the answer for:

```
?- bigger(Who, Whom).
```

You will get many answers! The Prolog treats the arguments *Who* and *Whom* as variables.

As an example we ask whether there is an animal $X$ that is both smaller than a car and bigger than a motorbike:

```
?- isbigger(car, X), isbigger(X, motorbike).
No
```

The following example explains the execution sequence of prolog statements.

**Example 5.4** Give the trace of execution of query "isbigger(bus, motorbike)", submitted to the animal world.

The trace of the above query is shown below.

```
 ? isbigger(bus, motorbike).
    1. Call: isbigger(bus, motorbike)?
    2. Call: bigger(bus, motorbike)?
    2. Fail: bigger(bus, motorbike)?
    2. Call: bigger(bus, _80)?
    2. Exit: bigger(bus, car)?
    2. Call: isbigger(car, motorbike)?
    3. Call: bigger(car, motorbike)?
    3. Exit: bigger(car, motorbike)?
    2. Exit: isbigger(car, motorbike)?
    1. Exit: isbigger(bus, motorbike)?
 True?
 Yes
 {Trace}
```

The trace can be verified to be performing as per the Rule1 and Rule2 discussed above.                                                                  □

Many a times, when started from goal, it may not be possible to reach to facts available. This shows that prolog is *incomplete* in theorem proving even for definite clauses, as it fails to prove facts that can be concluded from knowledge base.

## 5.7  Some Built-In Predicates

The built-ins can be used in a similar way as user-defined predicates. The important difference between the two is that a built-in predicate is not allowed to appear as the principal function in a *fact* or in the *head of a rule*. This must be so, because using them in such a position would effectively mean changing their definition [1].

*Equality.* We write $X = Y$. Such a goal succeeds, if the terms $X$ and $Y$ can be matched.

*Output.* Besides Prolog's replies to queries, if you wish your program to have further output, you can use the *write* predicate. The argument can be any valid Prolog term. In the case of a variable argument, its value will be printed. Execution of the predicate causes the system to skip a line, as in the following cases.

```
?- write(Hello World!), nl.
    Hello World!
    Yes

    ?- X = train, write(X), nl.
    train
    X = train
    Yes
```

```
read(N).
write('the number is'), write(N), nl.
the number is 5
N = 5
Yes
```

*Matchings.* Following are the examples for matchings. If two expressions matches, the output is 'Yes' otherwise it is 'No'. The query, to prolog shows that the two expressions cannot be matched.

```
?- p(X, 2, 2) = p(1, Y, X).
No
```

Sometimes there is more than one way of satisfying the current goal. Prolog chooses the first possibility (as determined by the order of clauses in a program), but the fact that there exists alternatives, is recorded. If at some point, Prolog fails to prove a certain subgoal, the system can go back and try an alternative left behind in the of executing of the goal. This process is known as *backtracking*. The following example demonstrates backtracking.

## 5.8  Recursive Programming

Using recursive programs, we can provide recursive definition of functions. We know that the factorial $n!$ of a natural number $n$ is defined as the product of all natural numbers from 1 to $n$. Here's a more formal, recursive definition (also known as an inductive definition), and the code in prolog [4].

**Example 5.5**  Factorial Program.

Recall the definition of factorial in Eqs. (5.7) and (5.8) in Sect. 5.4.3.
$0! = 1$, (base case) $n! = (n1)! * n$, for $n \geq 1$ (Recursion rule)

```
%finding factorial.
fact(0, 1).                    % base case

fact(N, R) :-  N >= 1,      % recursion step
               N1 is N - 1,
               fact(N1, R1),
               R is R1 * N.
```

☐

For a recursive program to test the membership of an element in a set, if the element is not as *head* of the list, then it is in the *tail*. The process is recursively called. The membership algorithm is built-in feature of prolog, as well as it can be user-defined.

**Fig. 5.9** Towers of Hanoi problem
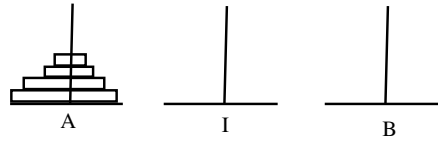


A recursive algorithm for GCD (greatest common divisor) based on Euclid's Algorithm can be constructed as follows.

**Example 5.6** Program for Greatest Common Divisor (GCD).

```
%gcd
gcd(X, X, X).
gcd(X, Y, Z) :- X > Y, D is X - Y, gcd(D, Y, Z).
gcd(X, Y, Z) :- X < Y, D is Y - X, gcd(X, D, Z).
```
□

**Example 5.7** Towers of Hanoi Problem.

Given three stacks *A* (source), *B* (destination), and *I* (intermediate), the towers of Hanoi problem is to move *N* number of disks from stack *A* to *B* using *I* as temporary stack. The disks are originally on stack *A* such that larger diameter disks are below the smaller diameter disks, and no two disks have equal diameters. The movement is to be done following the rules of this game, which states that only one disk is to be moved at a time, and at no time the bigger diameter disk shall come over a smaller diameter disk (Fig. 5.9).

```
move(A,B):- nl,
            write('move top from '),
            write(A),
            write(' to '),
            write(B).
transfer(1,A,B,I) :- move(A,B).
transfer(N, A, B, I):- N > 1,
                       M is N -1,
                       transfer(M, A, I, B),
                       move(A, B),
                       transfer(M, I, B, A).
```

The algorithm uses the strategy: move $n - 1$ disks from *A* to *I*, then move a single disk from *A* to *B*, finally move $n - 1$ disks from *I* to *B*. For $n - 1$, it recursively calls the algorithm. The predicate *nl* stands for new-line. □

## 5.9  List Manipulation

Prolog represents the lists contained in square brackets with the elements being separated by commas. Here is an example:

```
[train, bus, car, bicycle]
```

Elements of lists could be any valid Prolog terms, i.e. atoms, numbers, variables, or compound terms. A term may also be other list. The empty list is denoted by '[]'. The following is another example for a (slightly more complex) list:

```
[train, [], X, parent(X, tom), [a, b, c], f(22)]
```

Internally, lists are represented as compound terms using the function. (dot). The empty list '[]' is an atom and elements are added one by one. The list $[a, b, c]$, for example, corresponds to the following term:

```
.(a, .(b, .(c, []))) 
```

We discussed in Sect. 5.4.1 about lists. A list is a recursive definition, consisting of a head and a tail. The tail also comprises of head and rest of the elements as tail, and so on, until the tail is empty list.

**Example 5.8**  Membership Program.

```
% membership built-in
?-member(2, [a, b, c, 2, 4, 900]).
Yes.

% membership program
ismember(X, [X|R]).    % matches with 1st element
ismember(X, [Y|R]) :- ismember(X, R). % try for
                                       % next element
```

□

The built-in program *append*, appends two lists.

```
?append([1, 2, 3], [a, b, c], X). % This is buit-in
X=[1, 2, 3, a, b, c]
```

**Example 5.9**  Appending of lists.

$append([], L, L).$
$append([X|L1], L2, [X|L3]) : -append(L1, L2, L3).$

$$append([1, 2], [3, 4], Z)$$

$\theta_1 = \{1/X, [2]/L1, [3, 4]/L2,$    $append([X|L1], L2, [X|L3])$    ;Unification
$[1|L3]/Z\}$

call subgoal: $append(L1, L2, L3)$
$= append([2], [3, 4], L3)$

$\theta_2 = \{2/Y, [3, 4]/L5, [2, L4]/L3\}$   $append([Y|[]], [3, 4], [Y|L4])$    ;Unification

calls subgoal: $append([], [3, 4], L4)$
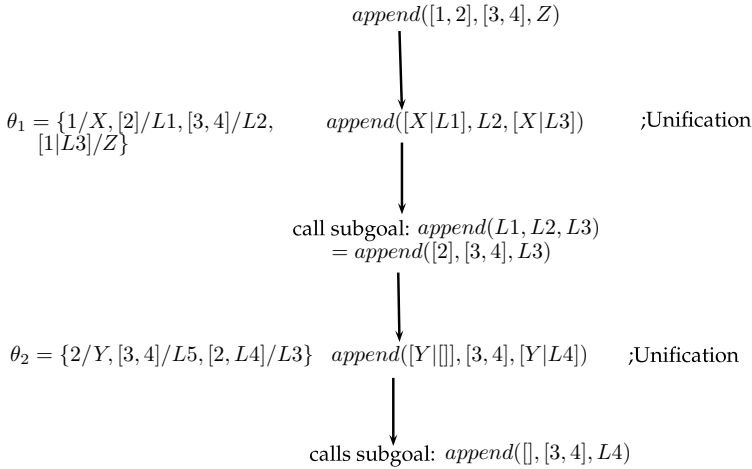
**Fig. 5.10**  Prolog search for appending two lists

For a query, we write,

```
? append([1, 2], [3, 4], Z)
Z = [1, 2, 3, 4]
Yes
?-
```

The search, along with unifications for appending two lists is shown in Fig. 5.10. The goal search shows alternate cycles of *unification* and calling of sub-goals. As a result of the recursion, the *append* operation can be realized as follows. The terminal node is matched with the *fact*: $append([], L, L)$. Consequently, $L4 = L5 = [3, 4]$. On back substitution,

$$L3 = [2|L4]$$
$$= [2|[3, 4]]$$
$$= [2, 3, 4].$$

$$Z = [1|L3]$$
$$= [1|[2, 3, 4]]$$
$$= [1, 2, 3, 4].$$

$\square$

Head and Tail

The first element of a list is called its *head* and the remaining list is called the *tail*. An empty list does not have a head. A list containing a single element has a head (namely that particular single element) and its tail is the empty list. A variant of the

list notation allows for convenient addressing of both head and tail of a list. This is
done by using the separator | (bar) [1].

```
?- [11, 12, 13, 14, 15] = [Head | Tail].
Head = 11
Tail = [12, 13, 14, 15]
Yes
```

Notice that the *Head* and *Tail* are just names for variables.[1] We could have used
*X* and *Y* or something else instead with the same result. We can access 2nd element
as well.

```
?- [bovi, jovi, kv, licet, quod, non,   ] = [_, X | _].
X = jovi
```

The more examples are as follows, which are self explanatory.

```
?- append([1, 2, 3], [d, e, f, g], X).
X = [1, 2, 3, d, e, f, g]
Yes

?- append(U, V, [a, b, c, d]).
U = []
V = [a, b, c, d] ;
U = [a]
V = [b, c, d] ;
U = [a, b]
V = [c, d] ;
U = [a, b, c]
V = [d] ;
U = [a, b, c, d]
V = [] ;
No

?- length([train, [], [1, 2, 3, 4]], Length).
Length = 3
Yes

?- member(bicycle, [train, bus, car, bicycle, motorbike]).
Yes

?- reverse([1, 2, 3, 4, 5], X).
X = [5, 4, 3, 2, 1]
Yes
```

---

[1] A Prolog variable starts with uppercase letter.

## 5.10   Arithmetic Expressions

Prolog is not designed to handle arithmetics efficiently. Hence, it handles expressions and assignment operations in some different way [1].

```
?3 + 5 = 8.
No

?X is 3 + 5.
X = 8
Yes
```

The terms $3 + 5$ and 8 do not match as the former is a compound term, whereas the latter is a number.

The following are arithmetical relational predicates:

```
X > Y
X < Y
X >= Y
X =< Y
X \= Y
X = Y
```

The last two predicates express inequality and equality, respectively.

## 5.11   Backtracking, Cuts and Negation

The Prolog language has number of predicates to explicitly control the backtracking behavior of its interpreter. This way the Prolog deviates from the logic programming idea. For example, the predicate *True* takes no arguments, and it always succeeds. Some of the other explicit predicates of Prolog as discussed below.

Fail Predicate

The predicate *Fail* also has no arguments, the condition *fail* never succeeds. The general application of the predicate fail is to enforce *backtracking*, as shown in the following clause:

$$a(X) \; : \text{-} \; b(X), \; fail.$$

When the query $a(X)$ is entered, the PROLOG interpreter first tries to find a match for $b(X)$. Let us suppose that such a match is found, and that the variable $X$ is *instantiated* to some term. Then, in the next step $fail$, as a consequence of its failure, enforces the interpreter to look for an alternative instantiation to $X$. If it succeeds

in finding another instantiation for $X$, then again $fail$ will be executed. This entire process is repeated until no further instantiations can be found. This way all possible instantiations for $X$ will be found. Note that if no side-effects are employed to record the instantiations of $X$ in some way, the successive instantiations leave no trace. It will be evident that in the end the query $a(X)$ will be answered by $no$. But, we have been successful in backtracking, i.e., going back and trying all possible instantiations for $X$, which helps in searching all the values.

The negation in prolog is taken as failure as shown in the following program.

**Example 5.10** Negation as failure.

```
bachelor(P) :- male(P), not(married(P)).
male(rajan).
male(rajam).
married(dicken).
```

When run, the queries responded as obvious. In the third case, married(Who) succeeds, so the negation of goal fails.

```
?bachelor(rajan).
 yes
?bachelor(dicken).
no
bachelor(Who).
Who = dicken
no
```

Cut Predicate

Some times it is desirable to selectively turn off backtracking. This is done by *cut* (!). The cut, denoted by !, is a predicate without any arguments. It is used as a condition which can be confirmed only once by the PROLOG interpreter: on backtracking it is not possible to confirm a cut for the second time. Moreover, the cut has a significant side effect on the remainder of the backtracking process: it enforces the interpreter to reject the clause containing the cut, and also to ignore all other alternatives for the procedure call which led to the execution of the particular clause.

**Example 5.11** Backtracking.

```
a :- b,c,d.
c :- p,q,!,r,s.
c.
```

Suppose that upon executing the call $a$, the successive procedure calls $b$, $p$, $q$, the *cut* and $r$ have succeeded (the *cut* by definition always succeeds on first encounter). Furthermore, assume that no match can be found for the procedure call $s$. Then as usual, the interpreter tries to find an alternative match for the procedure call $r$. For

each alternative match for $r$, it again tries to find a match for condition $s$. If no alternatives for $r$ can be found, or similarly if all alternative matches have been tried, the interpreter normally would try to find an alternative match for $q$. However, since we have specified a cut between the procedure calls $q$ and $r$, the interpreter will not look for alternative matches for the procedure calls preceding $r$ in the specific clause. In addition, the interpreter will not try any alternatives for the procedure call $c$; so, clause 3 is ignored. Its first action after encountering the cut during backtracking is to look for alternative matches for the condition preceding the call $c$, i.e., for $b$. $\square$

## 5.12   Efficiency Considerations for Prolog Programs

For a given goal, prolog explores the premises for rules in the knowledge base, making the goal true. If there are premises $p_1 \wedge p_2 \wedge \cdots \wedge p_n$, it fully explores the premise (called choice point) $p_i$ before proceeding to $p_{i+1}$.

The solution through prolog is unification, and binding of variables, pushing and retrieving the stack, associated with backtracking. When a search fails, prolog will backtrack to the previous choice point, followed with possibly unbounding of some of the variables. It always keeps track of all the bound variables at any moment, and those kept in the stack. In addition, it has to manage the index for fast searching of predicates. This is called trail. Accordingly, even the most efficient prolog interpreters consume thousands of machine instructions for even the simple unifications and matching.

For the huge task, and due to nature of computing, required, the normal processors give very poor performance to prolog programs. Hence, the prolog programs are compiled into intermediate programs, called WAM (Warren Abstract Machine). WAM helps prolog running faster as well as making it parallel.

Prolog may some times lead to incomplete loops.

The true version of prolog is called *pure Prolog*. It is obtained from a variation of the backward chaining algorithm that allows Horn clauses with the following rules and conventions:

- The Selection Rule is to select the leftmost literals in the goal.
- The Search Rule is to consider the clauses in the order they appear in the current list of clauses, from top to bottom.
- Negation as Failure, that is, Prolog assumes that a literal $L$ is proven if it is unable to prove $\neg L$.
- Terms can be set equal to variables but not in general to other terms. For example, we can say that $x = A$, $x = F(B)$ but we cannot say that $A = F(B)$.

These rules makes fast processing. But, unfortunately, the Pure Prolog inference Procedure is *Sound* but *not Complete*. This can be seen by the following example. Using this we are unable to derive in Prolog $P(a, c)$ because we get caught in an ever deepening depth-first search.

$$P(a, b).$$
$$P(b, c).$$
$$P(Y, X) : - P(X, Y).$$
$$P(X, Z) : - P(X, Y), P(Y, Z).$$

Actual Prolog

Actual Prolog differs from pure Prolog in three major respects:

- There are additional functionalities besides theorem proving, such as functions to assert statements, functions to do arithmetic, functions to do I/O.
- The "cut" operator allows the user to prune branches of the search tree.
- The unification routine is not quite correct, in that it does not check for circular bindings e.g. $X \rightarrow Y, Y \rightarrow f(X)$.

## 5.13   Summary

Prolog is a logic programming language, implemented in two parts:

1. *Logic*, which describes the problem, and
2. *Control*, provides the solution method.

This is in contrast to other programming languages, where description and solution go together, and they are hardly distinguishable. This, feature of prolog helps in separate developments for each part, one by the programmer and other by implementer.

The Prolog is being used in many areas where symbol manipulation is of prime importance; however, the main applications of this language is in the area of Artificial Intelligence.

In a well-structured program it is desirable to have data structures separate from the procedures which interrogate and manipulate them. This separation means that the data structures can be altered without altering the higher level procedures. Typically, an algorithm, whcih is separate from data structures, can be made more efficient by executing the same procedure calls either as coroutines or as communicating parallel processes.

A Prolog program is declaration of facts and rules, called knowledge base, a searching of this knowledge base in prolog is DFS (depth first search).

Running a prolog program is querying that program. Inferencing process in prolog is goal driven.

Due to long processing of unification, binding, searching, use of stack, a prolog program runs very slow. To run it faster, a prolog program is converted into a virtual machine code, which is executed by the WAM (warren abstract Machine). Efficiency of a Prolog program can be improved through two different approaches, either by improving the logic component or by leaving the logic component unchanged and improving the control over its use.

## Exercises

1. Determine, in which of the following lists cases the unification succeeds and where it fails? If it succeeds, write down the unifier. (Note: Uppercase are variables.)

   $[a, d, z, c]$ and $[H|T]$
   $[apple, pear, grape]$ and $[A, pear|Rest]$
   $[a|Rest]$ and $[a, b, c]$
   $[a, []]$ and $[A, B|Rest]$
   $[One]$ and $[two|[]]$
   $[one]$ and $[Two]$
   $[a, b, X]$ and $[a, b, c, d]$

2. Give Prolog predicates for *natural-number* and *plus* that implement the Peano axioms. Using *plus* give a Prolog program *times* for multiplication. Further, using *times* give a Prolog program *exp* for exponentiation.

3. Given the following knowledge base for prolog, find a female descendant of 'george', by manually running the program.

   ```
   parent(george,sam).
   parent(george,andy).
   parent(andy,mary).
   male(george).
   male(sam).
   male(andy).
   female(mary).
   ancestor(X,Z) :- parent(X, Z).
   ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
   ```

4. What is response of Prolog interpreter for following queries?

   ```
   ?[a,b,c,d]  =  [a,b,c,d|[]].
   ?[a,b,c,d]  =  [a,b,c,[d]].
   ?[a,b,c,d]  =  [a,b,[c,d]].
   ?[a,b,c,d]  =  [a|[b,c,d]].
   ?[a,b,c,d]  =  [a,b|[c,d]].
   ?[a,b,c,d]  =  [a,b,c,d,[]].
   ?[a,b,c,d]  =  [a,b,c|[d]].
   ?[a,b,c,d]  =  [a,[b,c,d]].
   ```

5. Which of the following lists are syntactically correct for Prolog language? Find out the number of elements in the lists that are correct.

   [1, 2, 3, 4| []]
   [1| [2| [3| [4]]]]
   [1, 2, 3| []]
   [[] | []]
   [[1, 2] |4]
   [1| [2, 3, 4]]
   [[1, 2] , [3, 4] | [5, 6, 7]] [1|2, 3, 4]

6. Write a predicate $second(S, List)$, that checks whether $S$ is the second element of List.
7. Consider the knowledge base comprising the the following facts:

   $tran(ek, one)$.
   $tran(do, two)$.
   $tran(teen, three)$.
   $tran(char, four)$.
   $tran(panch, five)$.
   $tran(cha, six)$.
   $tran(saat, seven)$.
   $tran(aat, eight)$.
   $tran(no, nine)$.

   Write a predicate $listtran(H, E)$ that translates a list of Hindi number words to the corresponding list of English number words. For example, for a list $X$, $listtran([ek, teen, chaar], X)$, should give response:

   $X = [one, three, four]$.

8. Draw the search trees for the following prolog queries:

   a. ?- $member(x, [a, b, c])$.
   b. ?- $member(a, [c, b, a, y])$.
   c. ?- $member(X, [a, b, c])$.

9. Write a program that takes a grammar represented as a list of rules and given a query of as sentence, and returns whether the sentence is grammatically correct.
10. Run the following programs in trace mode with single step, and describe the observed behavior, as why it is so?

    a. Factorial Program (Example 5.5).
    b. GCD Program (Example 5.6).
    c. Mortal men program (Example 5.2).

11. Given the following facts and rules about a blocks world, represent them in rules forms, then translate the rules into prolog, and find out "what block is on black block?"

**Facts**:
*A* is on table.
*B* is on table.
*E* is on *B*.
*C* is on *A*.
*C* is heavy.
*D* has top clear.
*E* has top clear.
*E* is heavy.
*C* is iron made.
*D* is on *C*.

**Rules**:
Every big, black block is on a red block.
Every heavy, iron block is big.
All blocks with top clear are black.
All iron made blocks are black.

# References

1. Clocksin WF, Mellish CS (2009) Programming in polog, 3rd edn. Narosa, New Delhi
2. http://www.dtic.upf.edu/~rramirez/lc/Prolog.pdf. Cited 19 Dec 2017
3. http://www.gprolog.org/. Cited 19 Dec 2017
4. Ivan B (2007) PROLOG programming for artificial intelligence, 3rd edn. Pearson Education
5. Strong John S (1989) The Legend of King Asoka—a study and translation of Asokavadana. Princeton University Press, Princeton, N.J.
6. Kowalski RA (1979) Algorithm = Logic + Control. Commun ACM 22(7):424–436
7. Van Emden MH, Kowalski RA (1976) The semantics of predicate logic as a programming language. J ACM 23(4):733–742
8. http://www.swi-prolog.org/. Cited 19 Dec 2017

# Chapter 6
# Real-World Knowledge Representation and Reasoning

**Abstract**  Apart from its theoretical significance, the AI must represent real-world knowledge, and produce reasoning using that. The real-world things are collections of entities in different classes. This chapter presents the representations structures for such knowledge, e.g., taxonomies and reasoning based on that. Other phenomena in real-world, that are presented are, action and change, commonsense reasoning, ontology structures for different domains, like, language, and world. The Sowa's ontology for objects, and processes, both concrete and abstract, is explained. The situation calculus is presented in its formal details, along with worked exercises. The more prevalent real-world reasoning, like nonmonotonic and default reasoning are also treated in sufficient details, along with supporting worked exercises. This is followed, with summary of the chapter, and exhaustive list of practice exercises.

**Keywords**  Real-world knowledge · Ontologies · Ontological reasoning · Sowa's ontologies · Situation calculus · Nonmonotonics reasoning · Default reasoning

## 6.1  Introduction

Who is taller, the father or his son held in his hands by the father? Can you make tea out of salt? If you push a needle into a potato, does the needle make a hole in the potato or in the needle? We may find these questions as absurd, but many tasks, such as machine vision, natural language processing, planning, and reasoning in general, requires the same kind of real-world knowledge and capabilities of reasoning. As another example, if a six feet tall person is holding a two feet baby in his arms, and it is also given that they are father and son, we take no time to conclude which one is father and which one is son.

The use of real-world knowledge for natural language processing (NLP), particularly for all kinds of sense disambiguation, and for machine translation is a challenging task. Some of the ambiguities are resolved using the simple rules, while the others can be resolved using rich understanding of the world. Consider the following two examples:

"The Municipal council of the city refused permission to demonstrators because they feared violence," versus,

"The Municipal council of the city refused permission to demonstrators because they advocated violence."

In the first sentence it is required to determine that "they" refers to "Municipal council of the city" and not to "demonstrators." But, in the second sentence it is required to find out that, "they" refers to "demonstrators." The above determination requires the knowledge about the characteristic relations of "Municipal council of city" and "demonstrators" with the verbs "fear" and "advocated." In NLP, such ambiguities in the language can be resolved by true understanding of the text, which requires bringing of real-world knowledge in the system.

Considering computer vision, we are able to recognize the objects because of context. For example, the *bowl*, *banana*, *knife* in kitchen are for container, for eating, and to cut vegetables, respectively. But, the same items in a departmental store are for sale. So, it is relation with other objects/environment, like kitchen and shop, explain the meaning of objects [1].

**Learning Outcomes of this Chapter**:

1. Compare and contrast the most common models used for structured knowledge representation, highlighting their strengths and weaknesses. [Assessment]
2. Identify the components of non-monotonic reasoning and its usefulness as a representational mechanism for belief systems. [Familiarity]
3. Compare and contrast the basic techniques for representing uncertainty. [Assessment]
4. Compare and contrast the basic techniques for qualitative representation. [Assessment]
5. Apply situation and event calculus to problems of action and change. [Usage]
6. Explain the distinction between temporal and spatial reasoning, and how they interrelate. [Familiarity]

## 6.2 Taxonomic Reasoning

Taxonomy is a name given to collection of individuals, categories, and the relations between their pairs. The taxonomies are also referred as *semantic networks*. As an example, Fig. 6.1 shows a taxonomy of some categories of animals, individuals, and the relations between categories and individuals, as well between category and its subcategories.

There are three basic relations types [2]:

• A category is a subset of other category. For example, the categories *dog* and *cat* are subsets of another category *mammal*;
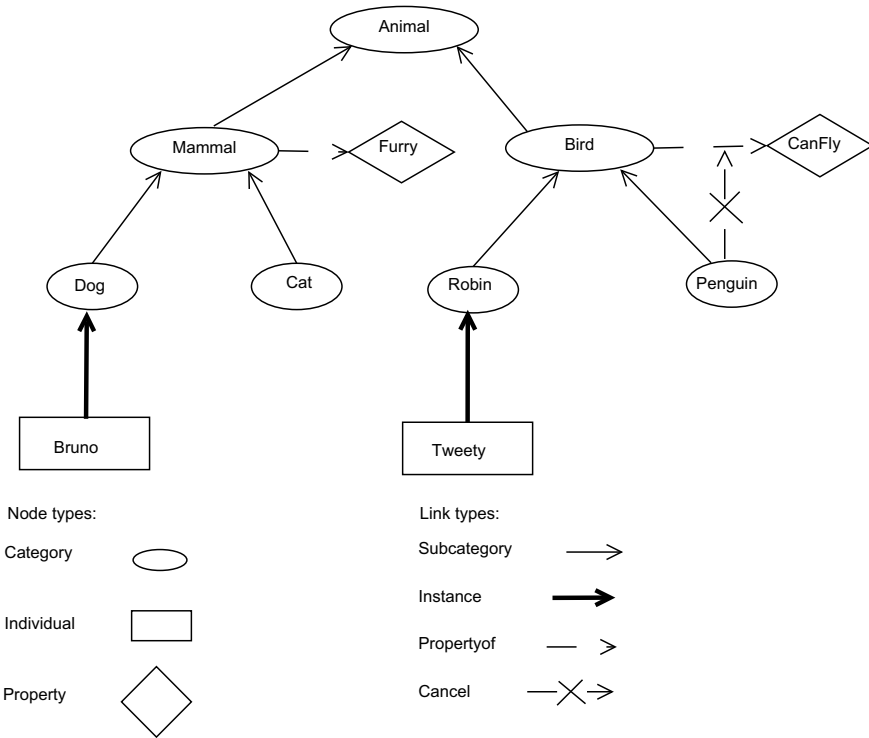
**Fig. 6.1** Taxonomy of animal world

- An individual can be an instance of a category. For example, an individual called *Bruno* is an instance of a category *dog*; *robin* and *penguin* are subsets (subcategories) of *bird*;
- Two categories are always disjoint. For example, *dog* and *cat* are disjoint sets.

A property can be used a tag for a category. For example a category *Mammal* can be tagged with property *furry*, and another category *bird* can be tagged with property *Canfly*.

One form of inference from taxonomical structure is through the relation of *transitivity*. In the taxonomy shown in Fig. 6.1, Bruno is shown as an instance of dog, and dog is a subset of mammals, then it can be reasoned that Bruno is an instance of mammal.

The other form of inference—the *inheritance*, works as follows:

Bruno is an instance of dog,
Dog is subset of mammal,
Mammal has property *furry*,
Therefore, it can be inferred through inheritance that dog and Bruno have property *furry*.

A variant of this inheritance is *default inheritance*, which have following characteristics:

1. A category can be marked with a characteristic but cannot be marked as a universal property, and
2. A subcategory or instance will always inherit property from higher order category unless otherwise it is specifically negated.

For instance, there is a category *bird*, with property *Canfly*. And, there are subcategories Robin and Penguin. The property *Canfly* should be inherited by sub-category Robin, but not by Penguin. This is implemented by negating the Canfly property for Penguin, as shown in the simple and standard taxonomy of the animal world in Fig. 6.1, where every two categories are related in such a way that, they are either disjoint or one is subcategory of other.

Other taxonomies structures are less straight forward. Consider a semantic network for categories of people, the individual say *Dr. C. V. Raman*, is having following relations with other categories, as an instance of those categories. He is simultaneously a *Physicist*, a *Nobel-laureate*, a *ScientistIndian*, and *NativeofMadras*. Also, there is an overlap, and it is some times not clear as which are the properties, and which are taxonomic categories. Hence, in taxonomizing more abstract categories, choosing and delimiting the categories become a challenging task.

Number of specialized taxonomies have been developed in the domains such as medicine, geonomics, languages, and other fields. The more sophisticated taxonomies are based on *description logic*, which provides the tractable constructs for describing concepts and the relation between them. These have been applied in semantic OWL (*Web Ontology Language*).

Temporal Reasoning

It provides the representation of knowledge and automating reasoning about *time*, *duration*, and *time intervals*. For example, in the sentence, "After landing at Frankfurt, it took a long time, before I could board the connecting flight to New Delhi" versus "It for the two countries a long time before they could arrive to trade treaty." In the first sentence, "long time" may mean few hours, while in the second same phrase may mean many years, or even decades. Integrating such reasoning, with natural language (NL) interpretation, has been problematic. Many temporal relations are not explicitly stated in text, and they are complex and text dependent.

Action and Change

Theory of *action and change* is other area of commonsense reasoning, and has been well understood. Following are the constraints of representation and reasoning for the domains of *a*ction and *change*:

- Events are *atomic*, that is, at a time only one event occurs. The reasoning will take place based on the state of the world at the beginning, and at the end of the event. The intermediate states while the event is in execution are not accounted for.
- Each and every change of the world is due to occurrence of some event.

- All the events are deterministic in nature, i.e., state of the world at the end of every event can be fully determined using the state before that event, plus this event itself.

The problem of representation, and the form of reasoning such as prediction and planning, have been largely understood for the domain that satisfy the above mentioned three constraints. However, there are other extreme domains: *continuous domains, simultaneous events, probabilistic events*, and *imperfect knowledge domains*, where, the representation and reasoning are not so well understood yet. Some of these domains will be discussed in the following text.

## 6.3   Techniques for Commonsense Reasoning

The field of commonsense reasoning can be largely divided into two areas:

- *knowledge based* approaches, and
- *machine learning* approaches that cover large data corpora, which are almost always of the text corpora types.

However, there is a very limited interaction between these approaches. There are other approaches, e.g., *crowd-sourcing* based approaches, which try to construct knowledge base as combination of other knowledge bases and participation of many non-experts [2].

The knowledge based approaches are further divided into categories that are based on following domains (see Fig. 6.2).

1. Mathematical logic or some other mathematical formalism,
2. Informal approaches, which are in contrary to mathematical formalism,
3. Based on theories from cognition, and
4. There are large-scale other approaches, which are usually mathematical or informal, but mainly targeted at collecting a large amount of knowledge.
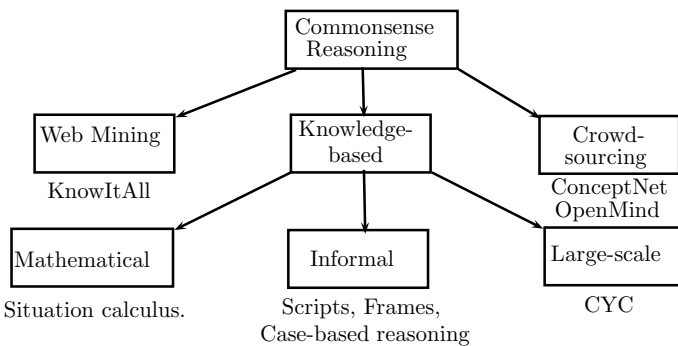


**Fig. 6.2**  Taxonomy of approaches to commonsense reasoning

One of the successful form of a commonsense reasoning, which has mathematical base is *qualitative reasoning* (QR). The QR helps in automating the reasoning and problem solving about the physical world around us. It has also generated techniques and systems that are being commonly used in application domains, like autonomous support for spacecraft, on-board diagnosis of vehicular systems and their failure analysis, automated generation of control software for photocopiers, and intelligent aids for learning about thermodynamic cycles. There are number of prominent features that are typical for QR systems: ontologies, causality, and inference of behavior from structure.

The other type of reasoning is *quantitative reasoning*, which deals with quantities, and reasons on the criteria of high/low, big/small, long/short etc.

The commonsense reasoning targets a number of different objectives, some of these are as follows:

- *Plausible inference*. It is used for drawing provisional/uncertain conclusions.
- *Reasoning architecture*. The reasoning architecture is a general-purpose data structures to encode knowledge and algorithms, which are helpful for carrying out the reasoning. For example, to represent the meaning/senses of natural language sentences.
- *Basic domains*. While performing the commonsense reasoning, human can do complex reasoning about basic domains like space, time, basic physics, and basic psychology. The knowledge we as human draw from these domains is largely not fully documented and the reasoning processes are largely unavailable for introspection. An automated reasoning should comprise all these reasoning capabilities.
- *Reasoning modes*. The commonsense reasoning system should incorporate a variety of modes of inference, like, explanation-based reasoning, generalization, abstraction, analogical-based reasoning, as well be able to perform the simulation.
- *Independence of experts*. Hand-coding of a large knowledge base for commonsense reasoning is expensive and slow process. Hence, assembling it either automatically or building it by drawing the knowledge of non-experts, e.g., through crowd-sourcing, is considered as better choice.
- *Breadth*. To perform a powerful commonsense reasoning through machines, will require a large body of knowledge.
- *Cognitive modeling*. These are the theories of automated commonsense reasoning that describe the commonsense reasoning in people.
- *Applications*. To be useful, a commonsense reasoner must interface with applications smoothly, and must serve the needs of applications.

## 6.4   Ontologies

Ontology is a particular theory about the nature of being or the kinds of existent. The task of intelligent systems in computer science is to formally represent these existents. A body of formally represented knowledge is based on conceptualization.

A concept is some thing we can form a mental image of. Conceptualization consists of a set of objects, concepts, and other entities about which knowledge is being expressed and of relationships that hold among them. Every knowledge model is committed to some conceptualization, implicitly or explicitly [3].

The major applications of ontologies are in natural language processing, information retrieval, modeling and simulation. The CYC (enCYClopedia) ontology, for example, is used for a CYC natural language system, built for the purpose of translating natural language text into cycL [4].

An ontology is a systematic arrangement of all of the important categories of objects or concepts that exist in some field of discourse, that shows the relations between these concepts/objects. In a completed form, an ontology is a categorization of all of the concepts in some field of knowledge, that includes all the objects, their properties, relations, and functions needed to define the objects, and specify their actions. A simplified ontology may contain only a hierarchical classification of objects and processes (called taxonomy), which shows the *subsumption* relations between concepts in the specific domain of knowledge [5].

An ontology may be visualized as an abstract graph with nodes representing the objects and labeled arcs representing the relations between them, as shown in Fig. 6.3, which is the upper level of the CYC (from en*CYC*lopedia) project hierarchy.

The CYC Project is an example of ontology. CYC contains more than 10,000 concept types used in the rules and facts encoded in the knowledge base. Its goal was to build a commonsense knowledge base containing millions of facts and their interrelationships. It was the efforts to build the knowledge in it that is seldom written down—the knowledge for example, the reader of an encyclopedia is assumed to possess to understand the contents of encyclopedia. This kind of knowledge is required as built-in in a system, which are required to read the restricted natural language.

A concept is some thing we can form a mental image of. At the top of the hierarchy is the *Thing* "concept", which does not have any properties of its own. The hierarchy under *Thing* is quite tangled. Not all the subcategories are exclusive. In general, Thing is partitioned in three ways: First is *Represented Thing* versus *Internal Machine Thing*. Every CYC category must be an instance of one and only one of these sets. Internal-Machine-Thing is anything that is local to the platform CYC is running on (strings, numbers, and so on). *Represented-Thing* is everything else.

The concepts used in an ontology and hierarchical ordering are to some extent arbitrary, and largely depends on the purpose the ontology is created. This is because, same objects are of varying importance when used for different purposes, and one or other set of properties of objects are chosen as the criteria for classification of these objects. Apart from this, different degrees of aggregation of concepts may be used, and the importance for one purpose may be of no concern for a other purpose. For example, *class* is collection of *students*, and when all of them are playing in ground we call it a *team*. As other example, when both parents are teaching they are called teachers, when traveling together they are passengers, and at home they are spouse.
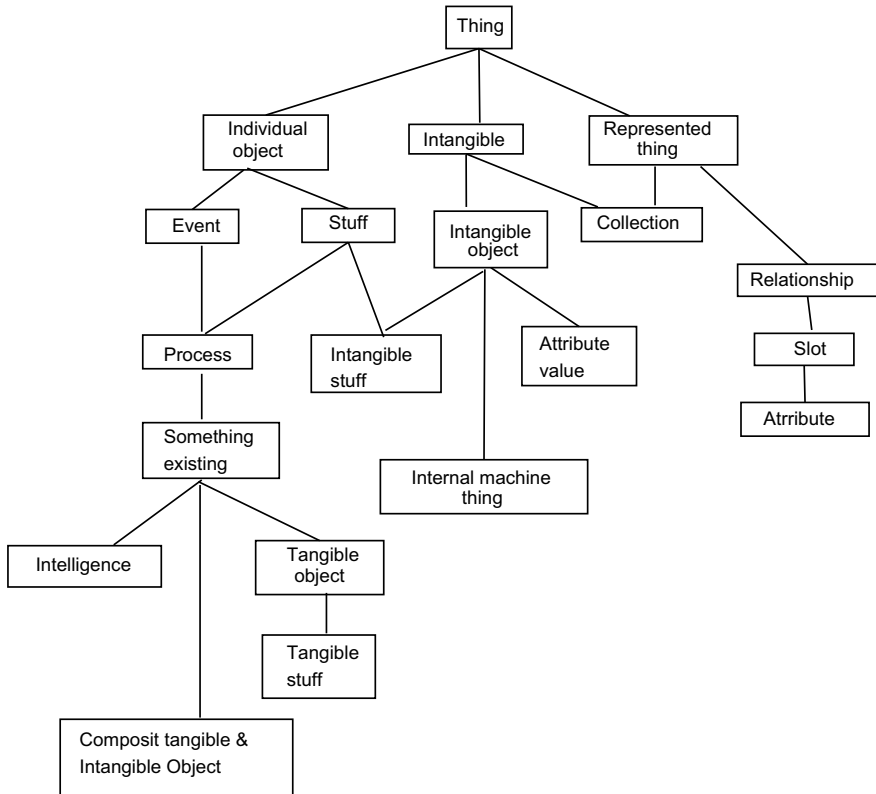
**Fig. 6.3** World ontology

## 6.5   Ontology Structures

When compared with First-Order Predicate Logic (FOPL) for knowledge representation and reasoning, in the ontologies we are particular for knowledge *organization* as well as *contents*. This approach has generalization, combined with exceptions. For example, "all birds fly" can be a rule. But, there are some birds which do not fly, for example, *penguin*. This is an exception. We should be able to add this concepts of knowledge, not as exception, but as extension. This requires categorization of objects.

Ontologies provide explicit representations of domain concepts busing a structure around which knowledge base is built. Each ontology is a structure of concepts and relations between them, such that all concepts are defined as well interpreted using a declarative approach. The ontology system also defines the vocabulary of the problem domain, and a set of constructs as how the terms can be combined to model the domain. The Fig. 6.3 shows a general structure of ontology.

**Fig. 6.4** Language, logic, and ontology
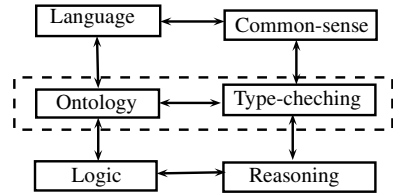


### 6.5.1 Language and Reasoning

The Fig. 6.4 shows the interrelationship of language and logic with ontology, and how each of these are related to commonsense, reasoning, and type checking. The language, ontology, and logic are representations, where as the corresponding blocks, i.e., common-sense, type-checking, reasoning, are respectively, the applications. Consider the object "ball", we identify it as a member in the collection of balls by its type checking; we say "plants have life", they belong to the category of all the living beings (type-check). At lowest level in Fig. 6.4 is *logic*, making use of binary operators. This logic helps in reasoning process. The common-sense is context level knowledge, which is essential for language understanding. For example, the phrase—"a good play", while in a cricket ground is understood as good shot, while in a theater, it is taken as performance by an artist. We resolve the phrase "a good play" by context.

There is a close relationship between natural language understanding (NLU) and knowledge representation and reasoning. In other words, the relationship between *language* and *knowledge*. In fact, understanding is reasoning paradigm, as it has become quite clear that understanding natural language is, for the most part, a commonsense reasoning process at the pragmatic level. As an example, to illustrate this strong interplay between language understanding and commonsense reasoning, consider the following:

(a) "Jack defended a jailed activist in every state."
(b) "Jack knows a jailed activist in every state."

Through commonsense, we have no difficulty in inferring for (a) that Jack supports for the same activist in every state. Perhaps jack might have traveled to different states for campaign. Where as in (b), we hardly conceive of the same. It may mean, "there is a jailed activist in every state", or an activist being jailed in every state. Such inferences lie beyond syntactic and semantic explanations, and are in fact depend on our commonsense knowledge of the world (where we actually live).

As another example, consider resolving of the noun He in the following:

(c) Rajan shot a policeman. He instantly

   (i) ran away.
  (ii) collapsed.

It is obvious that the above references can only be be resolved through common-sense knowledge. For example, typically, when $shot(x, y)$ holds between some $x$ and some $y$, the $x$ is the more likely a "subject" which would run away. So, in (c(i)), most likely its is Rajan who ran away. In c(ii), $y$ is the more likely to collapse, that is the policeman. Note, however, that such inferences must always be considered defeasible, since quite often additional information might result in the retraction of previously made inferences. For example, (c(ii)) might be describing a situation in which Rajan, a ten-year old who was shooting for practice, fell down. Similarly, and (c(i)) might actually be describing a situation in which the policeman, upon being slightly injured, tried to run away, may be to escape further injuries!

There are number of challenges, in term of computational complexities, in reasoning with uncommitted (or underspecified) logical forms. However, even bigger challenge is to make available of large body of commonsense knowledge, with computationally effective reasoning engines.

### 6.5.2   Levels of Ontologies

In comparing various ontologies, they can be viewed at three different levels:

1. *is-a* taxonomy of concepts,
2. the *internal concept* structure and relation between concepts, and
3. the presence or absence of *explicit axioms*.

The Fig. 6.5 shows various concepts related by *is-a* hierarchy relation (a member of relation), also subset relation. A member can be an instance of a category. Taxonomy is central part of most ontologies, and its organization can vary greatly. For example, all concepts can be in one large taxonomy, or there can be number of smaller hierarchies, or there can be no explicit taxonomy at all.

Although all general-purpose ontologies try to categorize the same world, they are very different at the top level. They also differ in their treatment of basic parts: *things*, *processes*, and *relations* (see Fig. 6.3).

The next level of comparison is internal concept structure, which can be realized by *properties* and *roles*. The internal concept relations are the relations, for example, between *bird* versus *canfly, wings, and feathers* (see also Fig.6.5). Concepts in some ontologies are atomic (axioms) and might not have any properties or roles or any other internal structure associated with them.

An important test for any ontology is the practical applications it is used for. These can be the applications in natural language processing, information retrieval, simulation and modeling, and so on, that use knowledge represented in the ontology.
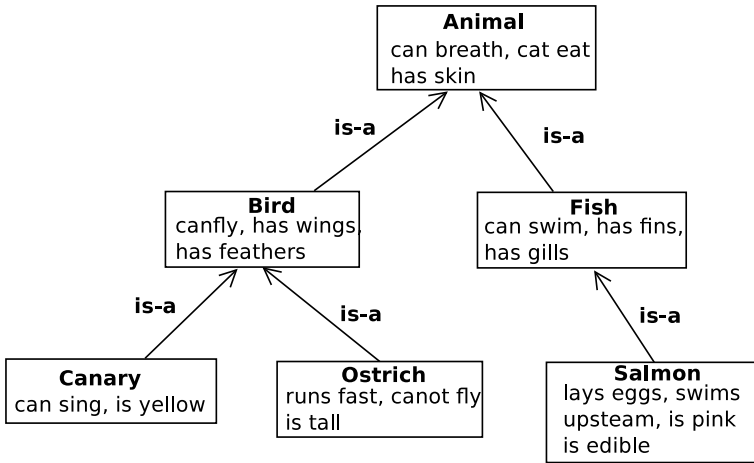
**Fig. 6.5** Top level ontology for animal

### 6.5.3 WordNet

The WordNet is a well developed lexical ontology—a manually constructed online lexical database system, provided with open access and semantically organized lexical objects. Its structure is provided with capability to distinguish between various parts-of-speech: like nouns, adjectives, verbs and adverbs. The basic object of WordNet is *synset*, which is a set of synonyms. Hence, if a word has more than one *senses* (e.g., the word "bank" for "money", and "bank" for "river bank"), it will appear in more than one synset. The synsets are organized in a hierarchy as super-class (hypernyms) and subclass (hyponyms). The Fig. 6.6 shows part of wordnet hierarchy of tangible things, with braces enclosing concepts in same synset. Note that in the synset {living thing, organization}, each of its subordinate item has two parts, e.g., {plant, flora}, but in case of {nonliving thing, object} there is only one item (the object name) in each of the subordinate, because the non-living thing is not classified thing [5, 6].

The Wordnet is a taxonomical architecture, which does not have more elementary concepts or axioms. For each concept (i.e., synset) provided in the Wordnet, there is a pointer to nouns representing its parts. For example, the parts of concept "bird" may be feathers, beak, and tail. There are provisions in the Wordnet for other types of pointers, for example, from noun to verb, to represent functions (actions), or from noun to adjective to represent properties of nouns.
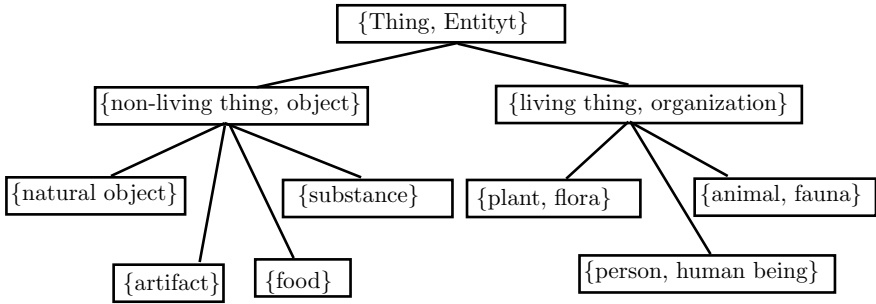
**Fig. 6.6** WordNet ontology representing subclass relation among synsets

### 6.5.4  Axioms and First-Order Logic

Structurally, an ontology is collection of terms, their definitions, and axiom relating them. The terms are typically organized as taxonomy. In some cases axioms are central to ontology design, while in other cases the axiomatic language is associated with central concepts.

Apart from the taxonomy and structure of concepts, axioms are the base for representing more information about categories and their relations. In addition, the axioms specify constraints on properties and role values for each category. Some times, axioms are specifically specified, while other times ontology consists of categories and corresponding frames, and every thing else in hidden in the application code. There is a fine-line difference between internal concept structure and axioms.

The category is represented using a frame formalism, and roles and properties are slot of the frame. The same facts can also be represented using axioms, i.e., a taxonomy can also be represented using axiomatic notations. In this notation, the axioms are represented using first-order predicate logic. For example, "all persons are living being and all living being are things", is represented Fig. 6.6, as well by the following Eq. 6.1.

$$\forall x \forall y [(person(x) \rightarrow livingbeing(x)) \wedge (livingbeing(y) \rightarrow thing(y))] \quad (6.1)$$

The CYC project, for example, comprised among the largest number of axioms, i.e., in the order of $10^6$.

### 6.5.5  Sowa's Ontology

John Sowa (1997, 1995) stated his fundamental principles for ontology design as "distinctions, combinations, and constraints" (see Fig. 6.7). He uses philosophical motivation as the basis for his categorization. There are three top-level distinctions:

**Fig. 6.7** Sowa's ontology
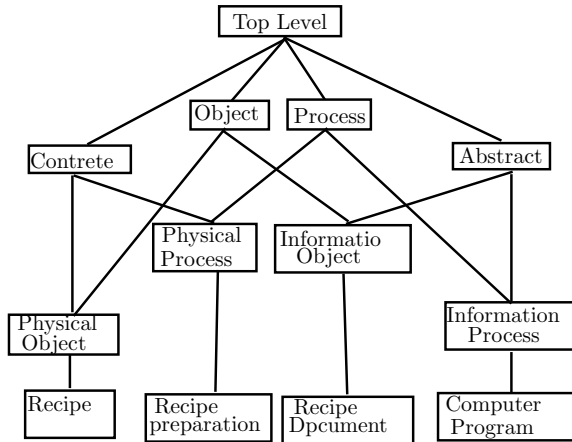


First is *Physical* versus *Information*, or *Concrete* versus *Abstract*. This is a disjoint partition of all the categories in the ontology [5].

The second principle for ontology design is based on combinations. The combinations classify the objects into *firstness* versus *secondness* versus *thirdness*, or *Form* versus *Role* versus *Mediation*. These categories are not mutually exclusive. For example, *Woman* is considered to be a form (firstness) because it can be defined without considering anything outside a person. As a mother, a teacher, or an employee, the same individual would be an example of a role (secondness). These roles represent an individual in relation to another type (a child, a student, an employer). Marriage is a mediation (thirdness) category because it relates several (in this case, two) types together, lecture class of AI is mediation as it relates many students as a type together.

The third principle is based on constraints. It is, *continuant* versus *occurrent*, or *object versus Process*. Continuants are objects that retain their identity over some period of time; occurrents are processes "whose form is in the state of flux". For example, *Avalanche* is a process, and *Glacier* is an object. Note that this distinction depends on the time scale. On a grand time scale of centuries, Glacier is also a process. These distinctions are combined to generate new categories (Fig. 6.7). At a lower level, for example, Script (a computer program, or a baking recipe) is a form that represents sequences and is thus defined as *Abstract*, *Form*, *Process*. Also, *History* (an execution of a computer program), which is a proposition that describes a sequence of processes, is then Abstract, Form, or object [7].

In the John Sowa's ontology, top level category is for every possible combination of *distinctions*. Constraints are used at the lower levels to rule out categories that cannot exist, as well it avoids too many combinations of categories. For example, *logical constraints*, would rule out triangles of more than three sides, and *empirical constraints* would rule out birds that talk or think. Representation of constraints is in the form of axioms, inherited through the ontology hierarchy to the lower levels.

## 6.6 Reasoning Using Ontologies

We have discussed in the above that, ontologies are taxonomical structures of categories, subcategories, and members of concepts. As human, we identify the objects, whether physical or abstract, due to this classification. For example, we can distinguish an oranges from cricket balls, because in the first the surface property belong to softness, and spongy, though their size and shape are quite similar. The Structures, e.g., a University System, a building, a computer network, operating system, or say, even office chair, are decomposed into parts, so that we identify the object as a whole as well the relationship of the parts remain explicit in this representation. Hence, it is due to these associated categorizations, and sub-categorizations, along with restrictions we are able to identify the real world objects. The reasoning can be done using categories and objects, using physical decomposition of categories, using measuring criteria, or using object oriented analysis.

### 6.6.1 Categories and Objects

The taxonomical hierarchies are common in government, military, and other organizations. Classifying objects into categories is important for knowledge organization. Of course interactions take place at individuals levels but relations are created at the level of categories. For example, the sentence, "Basketball team has won", does not refer to a single ball and nor a single player, but a relationship among team members as a group.

The Category also helps in prediction of objects once they are classified. One refers the objects from perceptual inputs, infers category of membership from perceived properties of the objects, then uses category information to predict about object. For example, from its yellow colour, size, shape, we identify that an object is a 'Mango'.

In FOPL there are two choices for categories: *predicates* and *objects*. Following are the examples:

$basketball(b)$;
$member(b, basketballs)$;
$b \in basketballs$; (the object $b$ is a member of category basketball)
$subset(basketballs, balls)$; (category is subclass of another category)
$basketballs \subset balls$.
$dogs \in domesticated animals$. (categories are members)

The category is defined by *members* and *subset* relation. We also note that categories organize the knowledge as well as they help to inherit. Thus, 'mango' ∈ 'mangoes', and 'mangoes'⊂ 'fruits'. Thus, mango inherits the properties of 'taste' from fruits (fruits have 'tastes'). Similarly, we have categories: animals, birds, foods, institutions. A "set of institutions" ⊂ "institutions", and MBM ∈ "institutions". The subclass organize the categories into a *taxonomic* hierarchy.

Following are the examples of reasoning through taxonomies:

$(x \in basketballs) \Rightarrow round(x)$; (all members of a category have same property of roundedness)
$color(x, brown) \wedge round(x) \wedge dia(x) = 9" \wedge x \in balls \Rightarrow x \in basketballs$; (the member category can be recognized by some property)

Having represented in taxonomy, when a top element is identified having some property, those its subordinates can be easily identified, as they possess similar properties, unless there is an exception.

### 6.6.2  Physical Decomposition of Categories

An object can be part of another, for example, nose, eyes, hands, are part of body; steering is part of car, wheels are part of wheel-assembly, and wheel-assembly is part of car. This can be represented by relations of physical decompositions,

$partof(wheel, wheelassembly)$.
$partof(wheelassembly, car)$.
$partof(x, y) \wedge partof(y, z) \Rightarrow partof(x, z)$.

Note that taxonomies are *transitive relations*. The relation of reflexivity also holds on individual objects, e.g., $partof(x, x)$, as $x$ has one part, and that part is itself. The categories of composite objects is defined by structural relations between parts and assemblies.

### 6.6.3  Measurements

The physical objects have height, weights, mass, length, and other physical measurements. To characterize a physical object, values need to be assigned to the objects in the form of their properties. Following are the examples:

$length(ipad) = inches(5)$ or $length(ipad) = centimeters(12.5)$.
$listprice(basketball) = rupees(500)$.
$height(x) > height(y) \Rightarrow taller(x, y)$.

### 6.6.4  Object-Oriented Analysis

The ontology development processes are similar to that of object-oriented design and analysis. In both, domain vocabulary is collected in the beginning, which is often out of domain's generic *nouns*, *verbs*, and *adjectives*. The result of object-oriented

analysis is in the form of a draft document for domain ontology and relevant to the application. However, analysts do not call the result as ontology. A designer in an object-oriented system defines the things like, objects, classes, interface functions, hierarchies, and the system behavior. But, an ontological engineer make use of intermediate representations such as graphs, tables, and semantic networks, to design hierarchies and other concept relationships, which hold true for object oriented design also.

In object oriented analysis, the classes act as templates. Both types of specialists make use of templates to specify product details. The classes are then merged or refined, as with ontologies.

## 6.7   Ontological Engineering

The *knowledge engineering* is process of knowledge representation—identification of task, assemble relevant knowledge, decide vocabulary of predicates, functions and constructs, encode knowledge about domain, encode description of specific problem instance, pose queries to procedures and get answers and debug knowledge base.

The ontological engineering can be defined on the same line of knowledge engineering. It is related to the creating representation of general concepts—actions, time, physical objects and beliefs.

*Ontological engineering* comprise a set of activities conducted during the time of: conceptualization, design, implementation and deployment of ontologies. Ontological engineering covers topics of philosophy, metaphysics, knowledge representation formalisms, development methodology, knowledge sharing and reuse, knowledge management, business process modeling, common-sense knowledge, systematization of domain knowledge, information retrieval from the Internet, standardization, and evaluation. It also gives us design rationale of a knowledge base, helps define the essential concepts of the world of interest, allows for a more disciplined design of a knowledge base, and enables knowledge accumulation. In practice, knowledge of above mentioned disciplines helps to:

- Organize the knowledge acquisition process;
- Specify the ontology's primary objective, purpose, granularity, and scope; and
- Build its initial vocabulary and organize taxonomy in an informal or semi-formal way, possibly using an intermediate representation.

Special-purpose languages/tools used for implementing ontologies, such as *Ontolingua*, *CycL*, and *LOOM* (lexical OWL ontology matcher), use a frame-based formalism, a logic-based formalism, or combination. Ontolingua is a frame-based language that uses KIF (Knowledge Interchange Format). It is a language for publication and knowledge communication, with notation and semantics like some extended form of first-order predicate logic. Ontolingua enables writing knowledge-level specifications independent of particular data or programming languages, and can translate a knowledge base from one representation language into another.

The LOOM takes two ontologies represented in OWL and produces a pair of relatedconcepts from the ontologies. In order to identify the corresponding concepts, LOOM compares the preferred names and symbols of the concepts in both ontologies. It identifies these concepts as similar, if and only if, their preferred names or synonyms are equivalent based on modified string functions. The string comparison removes the delimiters from both strings, then uses approximate matching techniques to compare them, allowing for mismatch of at most one character in a specified length.

Some languages/tools used in building ontologies are based on translation approaches. Using these approaches it is possible to build ontologies directly at knowledge level, and knowledge level is translated into the implementation level, thus eliminating the need to master the implementation languages [8].

## 6.8   Situation Calculus

The concept of *action* arises in at least two major subareas of artificial intelligence, (1) natural language processing and (2) problem solving. For the most part, the formalisms that have been suggested in each sub-area are independent of each other and difficult to compare. However, so far there does not exists a computational theory of actions, which is powerful enough to capture the range of the meanings and distinctions expressible in any natural language. The primary goal of situation calculus is to provide a formalism which is expressive enough for representation of actions and to explore its use in defining the meanings of English language sentences that describe actions and events [9].

The requirement on the formalism for representation of actions is that it should be a useful representation for *action reasoning* (i.e., problem solving). It has a very useful application, i.e., to describe, how this representation could be used for planning (of actions) or for plan recognition system. This is essential to the natural language processing as well, because the natural language understanding is aimed to ultimately result to problem-solving capability.

The situation calculus is also the language of choice for investigations of various technical problems that arise in theorizing about *actions* and their *effects*. It is being taken as a foundation for practical work in planning, control, simulation, database updates, agent programming and robotics. In parallel with these developments of its applications, there have emerged axiomatizations for the situation calculus, and explorations of some of their mathematical and computational properties [10].

### 6.8.1   *Action, Situation, and Objects*

The situation calculus is a logic formalism for representing and reasoning about dynamical domains. The concepts in the situation calculus are *situations*, *actions* and *fluents*. Number of objects are typically involved in the description of the world.

The situation calculus is based on a *sorted domain* with three sorts: actions, situations, and objects, where the objects include everything that is not an action or a situation. Hence, domain $D$ is,

$$D = \langle A, S, O \rangle \tag{6.2}$$

where $A$ is set of actions, $S$ is set of situations, and $O$ is set of objects, with variables of all sort can be used. The actions, situations, and objects are elements of the domain, but fluents are modeled either as predicates or as functions. The phrase, e.g., sorted domain of actions means, that set of actions are to be carried out in a certain (sorted), and similar meaning exists for sorted situations, and sorted fluents.

A situation is a kind of state (*ako*), however it is a result of chain of earlier situations. Actions are what make the dynamic world change from one situation to another when performed by agents. A fluent is a condition that can change over time. Fluents are situation-dependent functions used to describe the effects of actions. There are two kinds of them: *relational fluents* and *functional fluents*. The former have only two values: *True* or *False*, while the latter can take a range of values. For instance, one may have a relational fluent called *handempty* which is true in a situation if the robot's hand is not holding anything. We may need a relation like this in a robot domain. One may also have a functional fluent, e.g., *battery-level*, whose value in a situation is an integer of value between 0 and 100 denoting the total battery power remaining in one's notebook computer.

The set of actions form sort of domain. The action can also use variables, and they can be quantified, for example in a robot world, possible action terms would be $move(x, y)$ to model a robot moving to a new location $(x, y)$, and $pickup(o)$ to model the robot picking up an object $o$, and so on.

## 6.8.2  Formalism

A formalism used in situation calculus is aimed to implement the commonsense reasoning, such that the information used by humans can be expressed in sentences in logical order, then it is made part of the database (knowledge base). As next step, a goal oriented program will consult these databases for the required facts to ultimately achieve the goal. The database stores facts about what are the effects for any given actions, for example, a set of facts and effects concerning to a robot which moves objects from one location to another location. The situation calculus is designed to derive the actions (effects) for a given set of facts, independent of any specific problem.

The major part of the database are facts about effect which resulted due to actions, e.g., action of a robotic arm. These consequences of actions in situation calculus are of general nature, and not bound to specific problem, hence they can be used for solution of variety of problems, without creating new databases for new instances of problems.

In the situation calculus, a dynamic world is modeled to progress through a series of situations as a result of various actions being performed within this world. A situation is a consequence of sequence of action's occurrences. The situation available before any actions are performed is generally denoted by $S_0$, called *initial situation*. A new situation resulting from the performance of an action is denoted using the function symbol *result* or *do*. This function symbol has a 'situation' and 'action' as arguments, and a new situation as a result due to performing the given action in the given situation. We shall make use of *do*, called binary function symbol expressed as,

$$do : action \times situation \rightarrow situation. \tag{6.3}$$

The intended interpretation is that $do(a, s)$ (or $result(a, s)$) denotes the successor situation resulting from performing action $a$ in situation $s$. Accordingly, the basic formalism of the situation calculus is represented by:

$$s' = do(e, s) \tag{6.4}$$

which asserts that $s'$ is the resulting situation when event $e$ (action) occurs in situation $s$.

The basic ontology of situation calculus consists of:

1. situations, which corresponds to snapshots of universe or an instant of time, and,
2. actions or events, which change the world from one state to another.

It is a sorted language with sorting (order) for situation and actions.

The fluent "$iscarrying(o, s)$" can be used to indicate that the robot is carrying a particular object '$o$' in a particular situation '$s$'. If the robot initially carries nothing, "$iscarrying(Ball, S_0)$" is false while, the fluent

$$iscarrying(Ball, do(pickup(Ball), S_0))$$

is true. The location of the robot can be modeled using a functional fluent location, which returns the location coordinates '$(x, y)$' of the robot in a particular situation.

To describe a dynamic domain using the situation calculus, one has to decide on the set of actions available for the agents to perform, and the set of fluents needed to describe the changes these actions will have on the world. For example, consider the blocks world where some blocks of equal size can be arranged into some set of towers on a table. The set of actions in this domain depends on what the imaginary agent can do. If we imagine an agent to be a robot-arm that can be directed to grasp any block that is on the top of a tower, and either add it to the top of another tower or put it down on the table to make a new tower, then we can have the following actions:

$stack(x, y)$—put block $x$ on block $y$, provided that robot is holding $x$, and $y$'s top is clear. Being action, it is read "stack $x$ on $y$", and shall not be confused with predicate.

*unstack*(*x*, *y*)—pick up block *x* from block *y*, provided that robot's hand is empty, *x* is on *y*, and *x* has top clear.
*putdown*(*x*)—put block *x* down on the table, provided that robot is holding *x*.
*pickup*(*x*)—pick up block *x* from the table, provided the robot's hand is empty, *x* is on the table and top is clear.
*move*(*x*, *y*)—move block *x* to position *y*.

To describe the effects of these actions, we can use the following relational fluents:

*handempty*—True in a situation if the robot's hand is empty.
*holding*(*x*)—True in a situation if the robot's hand is holding the block *x*.
*on*(*x*, *y*)—True in a situation if block *x* is on block *y*.
*ontable*(*x*)—True in a situation if block *x* is on the table.
*clear*(*x*)—True in a situation if block *x* has top clear.

For action *stack*(*x*, *y*) to be performed in a situation, the fluent,

– *holding*(*x*) must be true, and
– *clear*(*y*) must be true.

Also, after *stack*(*x*, *y*) is performed and it results to a new situation, the fluent,

– *on*(*x*, *y*) will be true,
– *handempty* will be true,
– *holding*(*x*) will be false, and
– *clear*(*y*) will be false.

The action *stack*(*x*, *y*), along with present value of fluents, and resulting action with new value of fluents can be formally expressed as an axiom,

$$\forall x \forall y[[((holding(x) \wedge clear(y)) \rightarrow (stack(x, y)] \rightarrow on(x, y)$$
$$\wedge\ handempty \wedge \neg holding(x) \wedge \neg clear(y)).] \qquad (6.5)$$

Now, for example, we can say that for action *stack*(*x*, *y*) to be performed in a situation, *holding*(*x*) and *clear*(*y*) must be true, and that after *stack*(*x*, *y*) is performed, in the resulting new situation, *on*(*x*, *y*) and *handempty* both will be True, and *holding*(*x*) and *clear*(*y*) will no longer be True.

We only need the action *move*(*x*, *y*) to take place, to move block *x* to position *y*. This can happen if the agent (robot) in its world moves a block from a clear position to another clear position. The block *x* is clear to move means that *x* is on the table with top clear, or it is top most position in a stake of blocks. Accordingly, *move*(*x*, *y*) action can be expressed by axiom,

$$\forall x \exists y[(clear(x) \wedge clear(y)) \rightarrow move(x, y)] \rightarrow on(x, y) \wedge clear(x)]. \qquad (6.6)$$

For describing the effects of this action two fluents are sufficient: *clear*(*x*), and *on*(*x*, *y*). The action *move*(*x*, *y*) can be performed only when the situation "*clear*(*x*), *clear*(*y*), *x* ≠ *y*" is *True*. The consequence of this action is that *x* is no longer at the place where it was earlier, but instead at location *y* in the resulting new *situation*.

### 6.8.3   *Formalizing the Notions of Context*

The axiomatic system we discussed above is justified only in some specific context. With a little creative thought it is possible to construct a more general context in an axiomatic system. However, in such a generalized context the exact form of axioms will not hold. We can appreciate this argument if we think of human reasoning apparent in natural language, for example, consider axiomatizing of the adjective (concept) "on", to draw correct inferences from the information presented in a English language sentence, in a world of space-craft on a Mars mission (as context), where flight crew answers a query of other crew as,

<div align="center">"The book is on the table."</div>

As critic may propose to suggest about the precise meaning of 'on', causing difficulties about what can be between the book and the table or about how much gravity there has to be in a spacecraft in order to use the word "on" and whether centrifugal force counts. If the space-raft is orbiting the planet Mars! Thus, we encounter a Socratic puzzles over the issue of what the concepts mean in complete generality and come across examples that never arise in reality—"there simply is not a most general context"!

On the other hand, if a system is axiomatized at sufficiently higher level (i.e., more general), the axioms will be too long to be suitable even in special situations. Thus, we find it convenient to say, "The book is on the table," and omit the reference to exact time when it is on the table (global or local), its precise location on the table (on which part or corner it is located), and what is location of table in reference to global (GPS) coordinates, or universal coordinates, etc. Hence, we conclude that position of book in above example is sufficiently general. How much general the specification be? This depends on whether that general/commonsense knowledge has been expressed in the logic used for reasoning, in the corresponding program or in other other formalism.

A possible solution to the question—"how much general?", is, formalize the idea of context, and combine it with circumscription procedure of *monotonic* reasoning. This is done by adding context parameters to predicates and functions in the axioms, because each axiom makes an assertion about certain context. Apart from this, the axioms should express that, facts are inherited using more restricted context unless exceptions are specified. Each assertion is assumed to apply nonmonotonically in any particular more general context, but for that also there are exceptions. For example, there is rule that says, birds fly. In this rule, we implicitly assume that there is an atmosphere available. However, in a more general context, this (existence of atmosphere) is not required to be assumed. Further, it still remains to be determined as how the inheritance to more general contexts differs from that of more specific contexts.

There are a some predicates about contexts as well as dealing with time. One of the most important predicate is *holds*, which asserts that a property holds (i.e., is true) during a time interval. Thus, the sentence,

$holds(p, t)$

is true if and only if property $p$ holds during time $t$. A subsequent axiom will state, this is intended to mean that $p$ holds at every subinterval of $t$ as well. Note that if we had introduced *holds* as a "modal operator" we would not need to introduce properties into our ontology.

Suppose that, whenever a sentence $p$ is present in the memory of a computer, we consider it as in a particular context and as an abbreviation for the sentence,

$holds(p, C)$

where $C$ is the name of a context. We create a relation about the generality of a context as follows: a relation ($\leq$), e.g., $c_1 \leq c_2$ means context $c_2$ is more general than context $c_1$. Using this relation, we allow sentences like,

$holds(c_1 \leq c_2, C)$

which means that statements relating contexts ($c_1 \leq c_2$) can have contexts $C$.

A logical system using contexts might provide operations of *entering* and *leaving* a context yielding what we might call *unnatural deduction* allowing a sequence of reasoning as given in the followings.

$$holds(p, C)$$
$$ENTER\ C$$
$$p$$
$$...$$
$$...$$
$$q$$
$$LEAVE\ C$$

This resembles the usual logical natural deduction systems.

**Example 6.1** Represent the following sentences in the formalism of situation calculus.

"A man called Rex is standing at the gate of Ghana Bird Sanctuary wearing a black T-shirt. Rex loads his toy gun, waits for few seconds, and shoots at a migratory crane."

The fluents, which are either true or false, in this sentence are:

*standing(place)*: whether Rex is standing at a given place or not?
*black*: whether Rex is wearing black T-shirt or not?
*loaded*: whether the gun is loaded or not?

Following are the actions in above sentence:

*load*: Rex is loading the gun.
*wait*: Rex waits for few seconds.
*shoot*: Rex shoots his gun.

Now, let us find out what holds at initial situation $S_0$.

$holds(standing(gate), S_0)$
$holds(black, S_0)$
$holds(alive, S_0)$
$\neg holds(loaded, S_0)$

Now we try to relate actions with situations. In other words, which fluents will hold after performing a certain action in a given situation?

1. If Rex shoots the gun and gun is loaded, crane is not alive.
2. If Rex shoots the gun, gun will become unloaded.
3. If Rex loads the gun, gun will be loaded.
4. If Rex waits on an loaded gun, the gun remain loaded.
5. If Rex waits on an unloaded gun, the gun remain unloaded.

Our first method for writing above sentences in formal way is as follows. Consider a general sentence "$f_2$ (fluent) will be true by performing action $a_2$ in state $s$ if (provided that) $f_1$ is true in $s$."

$$\forall s[holds(f_1, s) \rightarrow holds(f_2, do(a_2, s))]$$

Now consider the first sentence: "if Rex shoots the gun and the gun is loaded, crane is not alive," which is written as:

1. $\forall s[holds(loaded, s) \rightarrow \neg holds(alive, do(shoots, s))]$
2. The remaining four sentences can be expressed in situation calculus as follows:
3. $\forall s[\neg holds(loaded, do(shoot, s))]$
4. $\forall s[holds(loaded, do(load, s))]$
5. $\forall s[holds(loaded, s) \rightarrow holds(loaded, do(wait, s))]$
6. $\forall s[\neg holds(loaded, s) \rightarrow \neg holds(loaded, do(wait, s))]$

Having represented in this form of FOPL, the reasoning is done using conventional methods used for predicate logic reasoning, e.g., resolution based theorem proving.

## 6.9  Nonmonotonic Reasoning

The most logic we have studied so far falls in the category of *monotonic logic*, and the corresponding reasoning as *monotonic reasoning*. The property of *monotonicity* is satisfied by all methods that are based on the classical (mathematical) logic. As per this property, if a conclusion is warranted on the basis of certain premises, no

additional premises should ever invalidate the conclusion. The Nonmonotonic logic is those ways of reasoning to infer additional information, that do not satisfy the monotonicity property of classical logic, that is, on the face of additional information, some of the earlier conclusions drawn may even become invalid!

In everyday life, however, we often draw sensible conclusions from what we know/information available to us. On receiving new/updated information, many a times we take back our previous conclusions, even when the new information we have collected has in no way indicated that it is contradicting the previous conclusions. Consider the following example: we have assumption that most birds fly, and that penguin are the birds which do not fly. On learning or receiving a new information that "Tweety is a bird", we infer that it flies. Further learning that "Tweety is a penguin", still does not effect our old inference that most birds fly. But, it (addition of new information) will require to abandon on inference of "Tweety flies." It is quite appropriate to say that intelligent automated systems must have the capabilities of this kind of inference, call as non-monotonic inference.

Many systems perform such nonmonotonic inferences. The most common are : negation as failure, circumscription, the modal system, default logic, autoepistemic logic and inheritance systems. Each of those systems is worth studying by itself, but a general framework in which those many examples could be compared and classified is necessary. The following section presents a general framework, concentrating on properties, which are important families of nonmonotonic reasoning systems.

## 6.10   Default Reasoning

One of the features of commonsense reasoning that makes it different from traditional mathematical proofs is the use of *defaults*. A default is a proposition that is postulated to be true in the absence of information to the contrary. For instance, an intelligent agent may assume by default that his observation correctly reflects the state of the world, and be prepared to retract this assumption in the face of evidence that be is in error [11].

The logic systems we have often come across, like, *classical logic*, *intuitionistic logic*, and *modal logic* are monotonic in nature. The name monotonic indicates that on adding a new fact in these systems never results in retraction of a conclusion derived before addition of that new fact. In fact, after addition of new knowledge in these logic systems, the inference ability of these systems monotonically increases. The Default logic is called nonmonotonic, because the new fact may be an exception to one of the defaults included in the set, and on the face of that the inference system should withdraw the conclusions drawn already.

In an intelligent system (either computer-based or human), when it tries to solve a problem, it may rely on complete information about the problem, and its main task will be to draw correct conclusions using classical reasoning. The classical (predicate) logic may be sufficient in such cases. But, in many situations, the system may have only incomplete information at hand, and it is required to draw inferences

with only these available information. The nonavailability of the information may be because of the need to respond quickly, and acquiring these more information will take some time, but the system cannot afford to wait for the collecting of all the relevant data. For example, the case evacuation required in a nuclear disaster, where we cannot wait to gather the complete information, and then decide to evacuate or not, as otherwise it would have caused havoc. Some times, the information available is unreliable/inaccurate, and it needs to be authenticated, which will require more time, and we cannot prolong the decision.

The classical logic possess the quality to represent and reason with certain aspects of incomplete information. In special conditions, additional information needs to be "filled in" to overcome the incompleteness, as it is important to make certain decisions. In such special circumstances, the system need to make some plausible conjectures, which are based on *rules of thumb*, called *defaults*, and this modified logic is called *default logic*, and the reasoning is called *default reasoning*. As an example, a doctor has to make some conjectures during a situation of emergency, about some most probable causes of the symptoms observed. Obviously, in such conditions it would be not appropriate to wait for the results of laboratory tests, as possibly extensive and time-consuming tests results may arrive too late to complete the diagnosis based on those and begin the treatment.

However, when a medical diagnosis (i.e., decision) is based on assumptions, it may turn out to be wrong on the face of availability of new information, and may require a modified diagnosis! The phenomenon of having to take back some previous conclusions is called *nonmonotonic* reasoning or *non-monotonicity*, which can be stated as: if a statement $\varphi$ follows from a set of premises $M$, i.e., $M \models \varphi$ and $M \subseteq M'$, then $\varphi$ does not necessarily follow from $M'$ (see Example 6.2). Default Logic provides formal methods to support this kind of reasoning.

**Example 6.2** Let

$$M = \{\forall x(bird(x) \Rightarrow fly(x)), \forall y(penguin(y) \Rightarrow bird(y)),$$
$$\forall z(penguin(z) \Rightarrow \neg fly(z)), bird(tweety)\}.$$

Given these, we have $M \models fly(tweety)$, and $M' = M \cup \{penguin(tweety)\}$ is inconsistent. We would expect the inference

$$M \cup \{penguin(tweety)\} \models \neg fly(tweety), \tag{6.7}$$

making $fly(tweety)$ a defeasible consequent. □

The default Logic is the most commonly used method for nonmonotonic reasoning, due to the simplicity of the notion of default used in this, and because the defaults occur quite common in real-life situations.

### 6.10.1   Notion of a Default

A rule used by football organizers in Kashmir valley might be: "A football game shall take place, unless there is snow in the stadium." This rule of thumb is represented by the default

$$\frac{football : \neg snow}{takes Place} \tag{6.8}$$

Consider the following example to understand the default reasoning: In the absence of information that there is going to be snowfall in the stadium, it is reasonable to assume $\neg snow$, and also to conclude that game will take place. Accordingly, the preparations for the game can go on. But actually if there is a heavy snowfall during the night before the game is scheduled, then this assumption is wrong. This is because, when we are certain, based on definite information that there is snow, we cannot assume $\neg snow$, and therefore the default cannot be applied. In this case we need to refrain from the previous conclusion (i.e., the game will take place), so the reasoning is nonmonotonic, as we are going to withdraw the previous conclusion [11].

Before proceeding with more examples, let us first explain why classical logic is not appropriate to model this situation. Of course, we could use the rule:

$$football \wedge \neg snow \rightarrow takes Place. \tag{6.9}$$

The problem with this rule is that we have to definitively establish that there will be no snow in the stadium before applying the rule. But that would mean that no game could be scheduled in the winter, as it would not be possible to comment about "no snow" on previous night of game, for in advance to decide whether to proceed for preparations or not! And, if we wait for the previous night of match to make decision for match, then there is no time left to do preparations. It is necessary to analyze the difference between two statements: having "to know that it will not snow", and being able to "assume that it will not snow." The defaults works on the second, and it supports drawing conclusions based on assumptions.

The defaults can be used to model *prototypical* reasoning, i.e., the most instances of a concept carry some property. One example is the statement "Typically, children have (living) parents"; this statement can be expressed using the default logic as,

$$\frac{child(X) : has Parents(X)}{has Parents(X)} \tag{6.10}$$

A no-risk reasoning is another form of default logic based reasoning, which is concerned to situations where we need to draw a conclusion even if it is not the most probable one, as another decision may be more disastrous. One such example is the

commonly used principle of awarding justice in the courts of Law: "In the absence of evidence to the contrary, assume that the accused is innocent." In default form we write this as:

$$\frac{accused(X) : innocent(X)}{innocent(X)} \tag{6.11}$$

The interpretation of rule (6.11) is that if $accused(X)$ is known, and there is no evidence that $innocent(X)$ (at numerator) is false, then $innocent(X)$ (at denominator) can be inferred.

### 6.10.2 The Syntax of Default Logic

A *default theory* $T$ is a pair $\langle W, D \rangle$, where $W$ is set of first-order predicate logic formulas (called the facts/axioms or belief set of $T$) and a countable set $D$ of default rules. A default rule $\delta$ is of the form

$$\delta = \frac{\varphi : \psi_1, \ldots, \psi_n}{\chi} \tag{6.12}$$

here $\varphi, \psi_1, \ldots, \psi_n, \chi$, are one or more *closed formulas* in predicate logic. The formula $\varphi$ is called *prerequisite* of the inference rule (6.12), $\psi_1, \ldots, \psi_n$ are called *justifications*, and $\chi$ is *consequent* or *inference* of this rule $\delta$.

It is important to note that the formulae in a default must be a *ground clause*. For example the formula,

$$\frac{bird(X) : flies(X)}{flies(X)} \tag{6.13}$$

is not a default according to the definition: it should have all clauses as ground clauses. Let us call this rule of inference as *open defaults* rule. An open default is interpreted as a default schema, and it may represents a set of defaults, which may be infinitely large in numbers.

A default schema is very much like a default, with the difference that $\varphi, \psi_1, \ldots, \psi_n, \chi$ in default are arbitrary predicate formulas (i.e., they may contain free variables). Where as, a default schema defines a set of defaults as,

$$\frac{\varphi\sigma : \psi_1\sigma, \ldots, \psi_n\sigma}{\chi\sigma} \tag{6.14}$$

for all *ground substitutions* $\sigma$ that assign values to all free variables occurring in the schema. That means free variables are interpreted as being universally quantified over the whole default schema. Given a default schema

$$\frac{bird(X) : flies(X)}{flies(X)} \qquad (6.15)$$

and the facts $bird(tweety)$ and $bird(sam)$, the *default theory* is represented as

$$
\begin{aligned}
T =& \langle W, D \rangle \\
=& \langle \{bird(tweety), bird(sam)\}, \\
& \left\{ \frac{bird(tweety) : flies(tweety)}{flies(tweety)}, \right. \\
& \left. \frac{bird(sam) : flies(sam)}{flies(sam)} \right\} \rangle.
\end{aligned}
$$

The Default Logic is a simple and commonly used method of knowledge representation and reasoning in real-world situations. It has following characteristics:

- The most important aspect is that it supports reasoning with incomplete information.
- Defaults inferences are found naturally in many applications, like in, medical diagnostics, reasoning related to legal issues, information retrieval, preparing specifications of systems, and as a logic in software.
- Default Logic is commonly used to model the reasoning with incomplete information, which was the original motivation, as well as a formalism that enables compact representation of information.
- Important prerequisites for the development of successful applications in these domains are,

  - basic concepts' understanding, and
  - there exists a powerful implementation of default logic.

### 6.10.3  Algorithm for Default Reasoning

The Algorithm 6.1 for default reasoning provides extensions to the formula set $W$. Let $T = \langle W, D \rangle$ be a closed default theory (unspecified variables are false) with a finite set of default rules $D$ and formula $W$. Let $P$ be the set of all permutations of elements of default set $D$. If $P = \{\}$, i.e., $D = \{\}$, or if $W$ is inconsistent, then return default theory $Th(W)$ as the only extension of $T$. The set of justifications and beliefs are indicated by variables *JUST* and *BELIEF*, respectively.

---

**Algorithm 6.1** Algorithm for Default Reasoning

---

1: $P$ = All permutations of elements of $D$
2: **while** $P \neq \{\}$ **do**
3:    Take a permutation, $perm = \{d_1, \ldots, d_n\} \in P$
4:    $P = P - \{perm\}$
5:    ;Initialization
6:    $BELIEF = W, JUST = \{\}$
7:    ;Application of defaults and consistency test
8:    **for** $i = 1$ to $n$ **do**
9:       ;[assume $d_i = \frac{A_i : B_i}{C_i}$]
10:       **if** $(BELIEF \vdash A_i) \wedge (BELIEF \nvdash \neg B_i)$ **then**
11:          $BELIEF = BELIEF \cup \{C_i\}$
12:          $JUST = JUST \cup \{B_i\}$
13:          **if** $\exists A (A \in JUST$ and $BELIEF \vdash \neg A)$ **then**
14:             exit the algorithm
15:          **end if**
16:       **end if**
17:    **end for**
18: **end while**
19: Return Th(BELIEF)
20: End

---

**Example 6.3** Find extensions of following default theory:

$$T = \langle W, D \rangle = \langle \{R(n) \wedge Q(n)\}, \left\{ \frac{R(x) : \neg P(x)}{\neg P(x)}, \frac{Q(x) : P(x)}{P(x)} \right\} \rangle$$

First consider the permutation of $D$ as,

$$(d_1, d_2) = \left\{ \frac{R(x) : \neg P(x)}{\neg P(x)}, \frac{Q(x) : P(x)}{P(x)} \right\}.$$

At the begin, we initialize $BELIEF = \{R(n) \wedge Q(n)\}, JUST = \{\}$. As per Algorithm 6.1, $d_1 = \frac{A_1 : B_1}{C_1} = \frac{R(x) : \neg P(x)}{\neg P(x)}$. From algorithm, we note that $BELIEF \vdash R(n)$, and $BELIEF \nvdash \neg(\neg P(n))$. Thus, we add $C_i$, i.e., $\neg P(n)$ in the $BELIEF$. Also, add $\neg P(n)$ into $JUST$. We also note that, the justification does not negate the $BELIEF$, hence it is consistent.

Next, we repeat the loop of Algorithm 6.1, for $i = 2$: $(d_2) = \frac{A_2 : B_2}{C_2} = \frac{Q(x) : P(x)}{P(x)}$. Before this, the updated $BELIEF = \{R(n) \wedge Q(n), \neg P(n)\}$. We note that $BELIEF \vdash A_2$ (i.e., $Q(n)$), but $BELIEF \nvdash \neg B_2$ (i.e., $\neg P(n)$) does not hold. Also, $Q(n) \in JUST$ holds, but $BELIEF \vdash \neg Q(n)$ does not hold, so algorithm continues. This conclude that belief is stable (see Table 6.1).

When above is repeated for other permutation, i.e., $(d_2, d_1)$, we have,

$$(d_2, d_1) = \left\{ \frac{Q(x) : P(x)}{P(x)}, \frac{R(x) : \neg P(x)}{\neg P(x)} \right\},$$

**Table 6.1** Belief computation-1

| Iteration | BELIEFS | JUSTIFICATIONS | Consistency test |
|---|---|---|---|
| Initial | $R(n) \wedge Q(n)$ | {} | |
| $i = 1, d_1$ | $\neg P(n)$ | $\neg P(n)$ | OK |
| $i = 2, d_2$ | Stable | | |

**Table 6.2** Belief computation-2

| Iteration | BELIEFS | JUSTIFICATIONS | Consistency test |
|---|---|---|---|
| Initial | $R(n) \wedge Q(n)$ | {} | |
| $i = 1, d_2$ | $P(n)$ | $P(n)$ | OK |
| $i = 2, d_1$ | Stable | | |

we get the extended belief set as shown in Table 6.2.

In conclusion, the $T$ has two different extensions: $Th(\{R(n) \wedge Q(n), \neg P(n)\})$, and $Th(\{R(n) \wedge Q(n), P(n)\})$, but obviously, not both at th same time.

## 6.11　Summary

*Ontology* is a theory about the nature of being or the kinds of existent. It is a systematic arrangement of all the important categories of *objects* or *concepts* that exist in some field of discourse, such that the arrangement shows the relations between them (objects/concepts). When compared with First-Order Predicate Logic (FOPL), in the ontologies we are particular for knowledge *organization* as well as *contents*, where as in predicate logic, the emphasis is on knowledge contents only.

The language, logic, ontology are closely related to commonsense, reasoning, and type checking, respectively. The language, ontology, and logic are representations, where as common-sense, type-checking, reasoning, are respectively, the applications. The natural language 'understanding is reasoning' paradigm. In comparing various ontologies, they can be viewed at three different levels: *is-a* taxonomy of concepts, (2) *internal concept* structure and relation between concepts, and (3) the presence or absence of *explicit axioms*.

John Sowa stated his fundamental top-level principles for ontology design as "distinctions, combinations, and constraints". The *distinctions* is Physical versus Information, *combinations* classify the objects into firstness versus secondness versus thirdness, or Form versus Role versus Mediation. The *constraint* is, continuant versus occurrent, or object versus Process.

The *knowledge engineering* is process of knowledge representation comprising of: identification of task, assemble relevant knowledge, decide vocabulary of predicates, functions and constructs, encode knowledge about domain, and deploy it. The *ontological engineering* is aimed to create representation of general concepts—actions, time, physical objects and beliefs.

Some of the languages and tools for implementing ontologies are: *Ontolingua*, *CycL*, and *LOOM*, which use either a frame-based formalism, or a logic-based formalism, or both.

Classifying objects into categories is important for knowledge organization and reasoning. For example, the sentence, "Basketball team has won", does not refer to a single ball and nor a single player, but a relationship among team members as a group. The Category helps in reasoning, e.g., prediction of objects once they are classified.

The concept of *action* arises in two major subareas of artificial intelligence, (1) natural language processing and (2) problem solving. However, the formalisms that have been suggested in each sub-area are independent of each other and difficult to compare. The requirement on the formalism for representation of actions is that it should be a useful representation for *action reasoning* (i.e., problem solving). It has an important application to describe how this representation could be used for planning (of actions) or for plan recognition system.

A *situation* is a consequence of sequence of action's occurrences. We shall make use of *do* (a binary function symbol) expressed as,

$$do : action \times situation \rightarrow situation.$$

which results to a new situation given an action and a present situation.

Nonmonotonic logic is the study of those ways of inferring additional information from given information that do not satisfy the monotonicity property. The following systems perform nonmonotonic reasoning:

- Negation as failure,
- Circumscription,
- Modal logic system,
- Default logic,
- Autoepistemic logic, and
- Inheritance systems.

One important features of commonsense reasoning that makes it different from traditional classical reasoning is the use of *defaults*. A default is a proposition that is assumed to be true in the absence of information to the contrary, i.e., unless the contrary information is made available, it is taken as true. In addition, the default logic is the most commonly used method for nonmonotonic reasoning due to its simplicity and due to the notion of a default.

## Exercises

1. Give the top level ontology of following structures, represent the concepts using relations, and attributes.

    a. University system—consisting of faculties, departments, teachers, classes, students, courses, etc.

  b. Organizational ontology of a manufacturing firm.
  c. Organizational ontology of a project based software company.
  d. Government system ontology with various bodies and their responsibilities.

2. Suggest some applications of Sowa's ontology.
3. Compare and contrast the Wordnet and Sowa's ontology, and explain the reasoning performed in both with small examples.
4. Represent the ontologies of the following worlds, and explain, how you will perform the question-answering using each of these ontology?

   a. Ontology of Shirt.
   b. Ontology of Dining table.
   c. Ontology of University system.

5. Suggest the approach, how you will generate e-learning exercises using the ontology.
6. What is unnatural deduction in reference to situation calculus? Give examples to justify your claims.
7. How the inheritance works in a world of contexts? For example, in space-craft, on earth, and when context changes from one to other?
8. Show some similarities between "contexts" and "properties" in expressing situation calculus axioms.
9. Consider a robotic-hand which can move between several bins, pickup an object from the bin if the hand is above the bin and the hand is empty. The hand can drop an object into a bin if the hand is holding an object and the hand is above the bin. Moving of hand from any bin to any other bin is always possible, it does not require any preconditions. The actions are:

   $drop(x, y)$ (drop object $x$ into bin $y$)
   $move(y)$ (move hand to be above the bin $y$)
   $grab(x, y)$ (pickup object $x$ from bin $y$).

   The fluents are:

   $holding(x, s)$ (the hand is holding $x$ in situation $s$)
   $over(y, s)$ (the hand is over bin $y$ in situation $s$)
   $in(x, y, s)$ (object $x$ is in the bin $y$ in a situation $s$).

   a. Write the axioms for $move$, $drop$ and $grab$ actions.
   b. Write the successor state axioms for all the $fluents$.

10. A robot is to pickup $n$ ($n = 10$) cuboid lying on the table and drop one-by-one in a bucket, available nearby the table. Write the statements for sequential calculus. What are the *situations, actions*, and *fluents* here?
11. Consider the eight puzzle shown in Fig. 6.8 with initial and goal states (situations).
    The objective is to go from a initial situation to the goal situation. We are allowed to move a tile into the empty space if that tile is adjacent to the empty space (e.g.

**Fig. 6.8** Initial and final
situation of 8-puzzle

Initial situation

| 1 | 2 | 3 |
|---|---|---|
| 4 | 5 | 6 |
| 7 | 8 |   |

Goal situation

| 1 | 2 | 3 |
|---|---|---|
| 8 |   | 4 |
| 7 | 6 | 5 |

in the initial situation tile 6 and 8 are adjacent to empty space). The locations are numbered as 1-9, as shown in initial situation, with number 9 as empty tile. The tiles are numbered 1-8. There is a single action $move(t, l)$ which indicates moving tile $t$ to location $l$. Assume a predicate $adjacent(l_1, l_2)$, which is true when location $l_2$ is one move from $l_1$. It is only possible to do a move action if a tile is in a location adjacent to an empty location. The only fluent is $location(t, l, s)$ meaning tile $t$ is in location $l$ in situation $s$. Given this, write down:

- initial conditions,
- effect axioms,
- precondition axioms, and
- from the effect axioms derive the successor state axioms.

12. Given the following set of facts and default rules:

    a. People typically live in the same city where they work (default: $d_1$)
    b. People typically live in the same city where their spouses are (default: $d_2$)
    c. John works in New Delhi (fact: $f_1$)
    d. John's spouse works in Mumbai (fact: $f_2$)

    Answer these questions:

    a. Where does John live according to default logic?
    b. Where does John live according to your intuition?

13. Formalize these set of facts and default rules:

    a. Bob usually speaks the truth ($d_1$).
    b. John usually speaks the truth ($d_2$).
    c. Bob says that the suspect stabbed the victim to death ($f_1$).
    d. John says that the suspect shot the victim to death ($f_2$).
    e. Nobody can be both stabbed and shot to death ($f_3$).
    f. Stabbing or shooting to death is killing ($f_4$).

    Answer these questions:

    a. Did the suspect kill the victim according to default logic?
    b. Did the suspect kill the victim according to your intuitions?

14. Is the formula (6.10) sufficient and correct form of default reasoning, with $X$ as variable? Justify your answer.

15. Translate the following into first-order predicate logic, and check whether the given conclusion follow from it?

Typically, the computer science students like computers. Female students who like computers are typically interested in cognitive science. The computer science students are typically female: for example Anita, Babita, Cathy; but Dorthy is an exception to this rule. Conclusion: Anita, Babita, Cathy are interested in cognitive science; Dorthy is not interested in cognitive science.

16. Compute the default extensions of following theories $T = (M, D)$:

a. $M = \{a\}$, $D = \{\frac{a:\neg b}{c}, \frac{:\neg c}{d}, \frac{:\neg d}{e}\}$

b. $M = \{a \rightarrow c, b \rightarrow c\}$, $D = \{\frac{:\neg b}{a}, \frac{:\neg a}{b}, \frac{:\neg d}{e}\}$

c. $M = \{\}$, $D = \{\frac{:\neg b}{a}, \frac{:\neg a}{b}, \frac{:\neg d}{d}\}$

d. $M = \{p \wedge q\}$, $D = \{\frac{b:a}{a}, \frac{:\neg a}{a}, \frac{:\neg a}{\neg c}, \frac{:\neg q}{b}, \frac{:\neg p}{q}\}$

17. Compute the default extensions of $T = \langle M, D \rangle$, where

$$M = \{\forall x[mynah(x) \rightarrow \neg nests(x)],$$
$$\forall x[penguin(x) \rightarrow \neg flies(x)],$$
$$\forall x[birds(x) \equiv mynah(x) \vee penguin(x) \vee canary(x)],$$
$$bird(Tweety)\},$$
$$D = \{\frac{bird(x) : nests(x)}{nest(x)}, \frac{bird(x) : flies(x)}{flies(x)}\}$$

18. Find the extensions of the following default theories:

a. $T = \langle \{\}, \{\frac{:\neg p}{p}, \frac{p \vee q:\neg p}{\neg p}\} \rangle$

b. $T = \langle \{\neg Sun\text{-}shining \wedge Summer\}, \{\frac{Summer:\neg Rain}{Sun\text{-}shining}\} \rangle$

c. $T = \langle \{\}, \{\frac{r:\exists x \ P(x)}{\exists x \ p(x)}, \frac{:r \wedge \neg p(x)}{r \wedge \neg p(x)}\} \rangle$

d. $T = \langle \{p \vee q\}, \{\frac{:\neg p}{p}, \frac{p \vee q:\neg p}{\neg p}\} \rangle$

19. Assume that $\langle D, W \rangle$ be a propositional default theory, and $D'$ be a set of normal defaults such that $D \subseteq D'$. If $E$ is an extension of $\langle D, W \rangle$, then show that there exists an extension $E'$ of $\langle D', W \rangle$ such that $E \subseteq E'$.

# References

1. Chowdhary KR (2004) Natural language processing for word-sense disambiguation and information extraction. PhD thesis, Department of Computer Science and Engineering, J.N.V. University, Jodhpur (India)
2. Davis E, Marcus G (2015) Commonsense reasoning and commonsense knowledge in artificial intelligence. Commun ACM 58(9):92–103
3. https://protege.stanford.edu/ . Cited 19 Dec 2017
4. Douglas BL (1995) CYC: a large-scale investment in knowledge infrastructure. Commun ACM 38(11):33–38

5. Fridman-Noy N, Hafner CD (1997) The state of the art in ontology design: a survey and comparative review. AI Mag 53–74
6. http://clarity.princeton.edu/pub/wordnet/ . Cited 19 Dec 2017
7. Sowa JF (1995) Distinctions, combinations, and constraints. In: Proceedings of the workshop on basic ontological issues in knowledge sharing. Montreal, Canada
8. Devedzic V (2002) Understanding ontological engineering. Commun ACM 45(4):136–144
9. Pinto JA (1994) Temporal reasoning in the situation calculus. PhD Dissertation, Submitted to the Graduate department of Computer Science, University of Toronto
10. Thielscher M (1999) From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. Artif Intell 111:277–299
11. Antoniou G (1999) A tutorial on default logics. ACM Comput Surv 31(3)

# Chapter 7
# Networks-Based Representation

**Abstract** Network-based method is another approach for knowledge representation and reasoning. They have particularly the advantage that, using the network one can navigate through the knowledge represented, and can perform the inferences. This chapter presents the semantic networks, conceptual graphs, frames, and conceptual dependencies, as well as their syntax and semantics. The $\mathscr{DL}$ (description logic)—a modified predicate logic for real-world applications is treated in detail, with examples of its language—the concept language for inferencing. Conceptual dependency (CD) is a language-independent representation and reasoning framework, such that whatever may be the natural language used, as long as its meaning is the same, the CD will be the same. The script language for representation and reasoning along with its syntax, semantics, and reasoning for CD is presented, followed with chapter summary, and an exhaustive list of exercises.

**Keywords** Network-based representation · Semantic networks · Conceptual graph · Frames · Description Logic (DL) · Conceptual dependencies · Scripts

## 7.1 Introduction

*Semantic Networks* were developed with the goal of characterizing the knowledge and the reasoning of a system by means of network-shaped cognitive structures. The similar goals were later achieved through frame-based systems, which depend on the notion of a "frame" as a prototype, and have the capability to express the relationship between the frames. The frames and semantic networks are quite different from each other, but both have cognitive features, and have the capability that allows navigation in the structures. Due to this, both of them can be classified as network-based structures, where the network is used for representing individuals and the relationship between them.

Due to their human-oriented origins, the network-based systems are more appealing and effective from the practical point of view than the logical systems, that are based on predicate logic and its variants. However, these network-based systems were not accepted as a complete solution, due to their lack of semantic properties.

Due to that, every system behaved differently from the others, despite their almost identical-looking components, as well as identical-looking relationships. Hence, the need was felt to represent semantic characteristics in these structures. The semantic characteristics in network-based systems could be achieved by introducing the notion of hierarchical structures. Using hierarchical structures in semantics networks and frames, one could gain both in terms of ease of representation, and also in terms of efficiency of reasoning.

**Learning Outcomes of this Chapter**:

1. Identify all of the data, information, and knowledge elements and related organizations, for a computational science application. [Assessment]
2. Describe how to represent data and information for processing. [Familiarity]
3. Compare and contrast the most common models used for structured knowledge representation, highlighting their strengths and weaknesses. [Assessment]
4. Identify the components of non-monotonic reasoning and its usefulness as a representational mechanism for belief systems. [Familiarity]
5. Compare and contrast the basic techniques for qualitative representation. [Assessment]

## 7.2   Semantic Networks

A network-based representation provides means of structuring and exhibiting the structure of knowledge. In a network, the pieces of knowledge are clustered together into coherent semantic groups. It provides a natural way of mapping knowledge between the natural language and these networks. In addition, the network representation provides a pictorial representation of knowledge objects, their attributes, and relationship between them [2].

The basic difference between ontologies we studied earlier and the semantic networks is that, ontologies are hierarchies, which may have multiple inheritances, providing knowledge organization of world. Whereas, semantic networks are not necessarily be hierarchy and they follow the lattice structure for knowledge representation.

There is a need of built-in feature of natural language understanding in knowledge representation, so that it becomes possible to carry out the inference through this representation. However, when we make use of general theorem proving framework, like resolution, these actually desired inferences are lost due to a wide range of inferences carried out using resolution-refutation method (see Example 3.14, p. 74).

The Semantic networks not only represent information but facilitate the retrieval of relevant facts. For instance, all the facts about an object "Rajan" are stored with a pointer directly to one node representing Rajan. Another advantage of semantic networks is about the inheritance of properties. If a semantic network represents the knowledge: "All canaries are of yellow color", and "Tweety is canary", the network would be able to infer that "Tweety is of yellow color." This inference is performed

**Fig. 7.1** A semantic
network
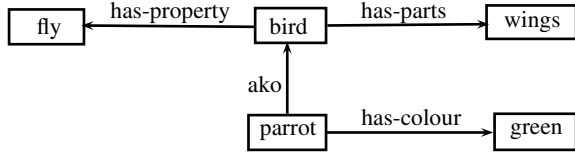


through *network mathcer* or *retriever*. The most advanced system of inference has *Inference Engine*, which can perform specialized inferences tailored to treat certain functions, predicates, and constant symbols differently than others. This is achieved by building into the inference engine certain true sentences, which involve these symbols, and control is provided to handle these sentences. The inference engine is able to recognize the special conditions, on which it makes use of specialized machinery. It becomes possible by coupling the specialized knowledge to the form of situations that it can deal with.

The semantic networks also called the *Associative Networks*, model the semantics and words of the English language. In a system developed by their inventor Quillian, evidenced that meanings were found between the words by the path connecting them. These models carry an intuitive appeal: the related information is always clustered and bound together through relational links, and the knowledge required to perform a certain task is typically contained in a narrow domain or in the vicinity of the concerned task. This type or knowledge organization, in some way, resembles the way knowledge is stored and retrieved in human brain [11].

Semantic networks includes a lattice of concept types [6]. Originally they included a different correlated nets, which were based on 56 relations, like: subtypes, instances, case-relations, part-whole, kinship relations, and various types of attributes. A simple example of semantic network is shown in Fig. 7.1, where *ako*(a-kind-of), *has-parts*, *color*, and *has-property*, are binary relations.

There are several benefits of using Semantic Networks for representing knowledge:

- Real-world meanings (semantics) are clearly identifiable.
- Reflects the structure of the part of the world being represented in the knowledge structuring.
- The representation due to "is-a" and "is-partof" relations help in organizing the inheritance based hierarchies, which are useful for inheritance-based inferences.
- Accommodates a hierarchy to be useful for default reasoning (e.g., we can assume the height of an adult as 175 cm, but if the person is a basketball player, then we take it as 190 cm).
- The semantic networks are useful in representing events and natural language sentences, whose meanings can be very precise. However, the concept of semantic networks is very general. This causes a problem, unless we are clear about the syntax and semantics in each case.

The Semantic networks have been used for knowledge representation in various applications like, natural language understanding, information retrieval, deductive databases, learning systems, computer visions, and speech generation systems.

### 7.2.1   Syntax and Semantics of Semantics Networks

Unlike the predicate logic, there is no well accepted syntax and semantic for semantic networks. A syntax for any given system is determined based on the objects and relation primitives chosen and the rules used for connection of the objects. However, there are some primitives which are quite established. We can define a Semantic Network by specifying its fundamental components:

1. *Lexical part*:

   a. *Nodes* denote the objects.
   b. *Edges* or *links* denote the relations between objects.
   c. *Labels* denoting the particular objects and relations between them.

2. *Structural part*: The nodes and the edges connecting them form directed graphs, and the labels are placed on the edges, which represent the relation between nodes.
3. *Semantic part*: Meanings (semantics) are associated with the edges and node labels, whose details depend on the application domains.
4. *Procedural part*: The *constructors* are part of the procedural part, they allow for the creation of new edges (links) and nodes. The *destructors* allow the deletion of edges and nodes, the *writers* allow the creation and alteration of labels, and the *readers* can extract answers to questions. Clearly, there is plenty of flexibility in creating these representations.

The word-symbols used for the representation are those which represent object constants and *n*-ary relation constants. The network nodes usually represent nouns (objects) and the arcs represent the relationships between objects. The direction of the arrow is taken from the first to the second objects, as they represent in the relations. The Fig. 7.2 shows a *is-a* hierarchy representing a semantic network. In set theory terms, *is-a* corresponds to the *sub-set* relation '⊆', and an *instance* corresponds to the membership relation '∈' (an object class relation) [3].

The commonly used relations are: *Member-of*, *Subset-of*, *ako* (a-kind of), *has-parts*, *instance-of*, *agent*, *attributes*, *shaped-like*, etc. The 'is-a' relationship occurs quite often, like, in sentences: "Rajan is a Professor", "Bill is a student", "cat is a pet animal", "Tree is a plant", "German shepherd is a dog", etc. The 'is-a' relation is most often used to state that an object is of a certain type, or to state that an object is a *subtype* of another, or an object is an *instance* of a class.

Figure 7.2 shows some important features of semantic networks. The representation makes it easy to retrieve the properties of any object efficiently due to hierarchy of relations. These networks implement property of inheritance (a form of *inference*).
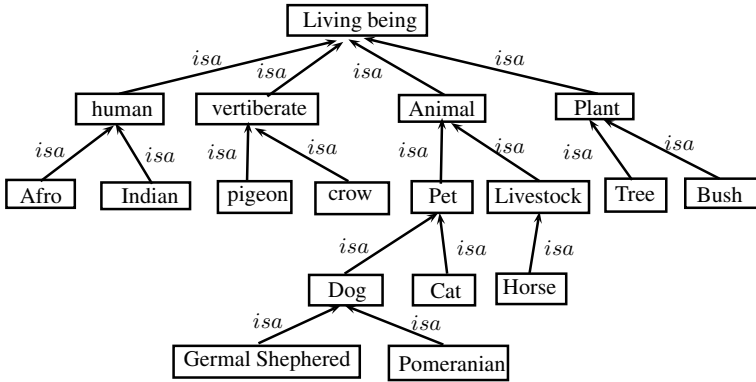
**Fig. 7.2** Semantic network showing "Is-a" Hierarchy

**Fig. 7.3** A semantic network showing contradiction to an inherited property
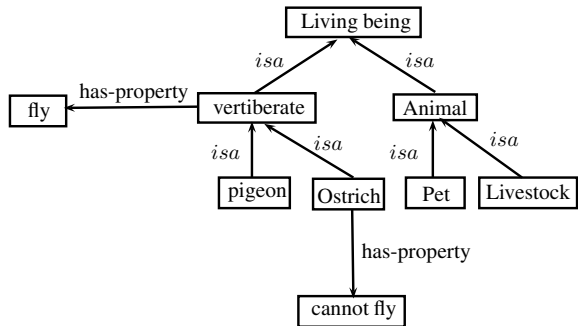


The nodes which are members or subsets of other nodes may inherit the properties from their higher level ancestor nodes. For example, we can infer from Fig. 7.2 that dogs are animals and pigeons are vertebrates, and both of them are living beings. The property inherited like this is recognized as *default reasoning.* It is assumed that unless there is an information to the contradictory, it is reasonable to inherit the information from the ancestor nodes. In Fig. 7.3, Pigeon inherits the property of "can fly" from the vertebrates, while Ostrich has a locally installed attribute of "cannot fly", hence the property 'fly' will not be inherited by it.

The inference procedures for semantic networks can also be in parallel to those in propositional and predicate logic. For example, if a class *A* of objects have property *P*, and '*a*' is a member of *A*, we can infer that '*a*' has property *P*. The inferences in these networks can be defined as per those in predicate logic, making use of unification, chaining, modus ponens, and resolution, however, the reasoning is *default* type [13].

### 7.2.2   Human Knowledge Creation

The semantic networks are based on the *associationist* theory, which defines the meaning of an object in terms of a network of associations with other objects. When human perceives and reasons about an object, that perception is first mapped into a concept. This concept is part of our entire knowledge about the world and is connected through appropriate relationships to other concepts. These relationships form an understanding of the properties and behavior of objects such as *snow*. For example, through associations, we associate the snow with other concepts like cold, white, snowman, slippery, and ice. Our understanding of snow and truth of statements such as "snow is white" and "the snowman is white" manifests out of this network of associations.

There is experimental evidence that, in addition to associate concepts, humans also organize their knowledge hierarchically, as per that information is kept at the highest levels, to be inherited by other concepts. The evidence have shown that if the concepts in the networks were far off in the hierarchy, it took a relatively longer time by humans to understand this relation compared to the concepts which were in proximity to each other. For example, with reference to Fig. 7.2, the human response will be faster to infer that "dog is an animal", compared to the statement "Pomeranian is a living being". This time difference is argued due to the fact that humans store the information in a hierarchical way. The fastest recall was for the cases where the traits were specific to the object. For example, the negative response for "Can Ostrich fly?", will be faster than the positive response for "Can pigeons fly?". This is because in the reasoning for the first path: "Ostrich → cannot fly", is faster that for "pigeon → vertebrate → fly", due to path length difference [1].

### 7.2.3   Semantic Nets and Natural Language Processing

The inheritance based system allows to store the knowledge at the highest level of abstraction. This results to reduction in size of the knowledge base, as well as it helps in updating the inconsistencies. The graphs can explicitly represent the relations using arcs and nodes, which helps in formalizing the knowledge of semantic networks. These networks can be used to answer various queries related to the knowledge base stored, using inferences.

Much of the research in network representation has been done in the field of Natural Language Understanding (NLU). Often, the natural language understanding requires the understanding of the common sense, the ways in which the physical objects behave, the interactions that occur between humans, and the ways in which the human institutions are organized. A natural language understanding the program must understand the intentions, beliefs, hypothetical reasoning, plans, and goals embedded in the natural language text. Due to these, the language understanding has been the driving force for knowledge representation.

The NLU programs define the words of the English language in terms of other words, like the English dictionary does, rather than defining in terms of primitive words or axioms. Thus, to understand the meaning of a word we traverse a network of words until we understand the meaning of the required word.

### *7.2.4   Performance*

The network structures used to provide intuitive and useful representations for modeling semantic knowledge. These models are required to fulfill the following four objectives:

1. Is is possible to organize the human semantic knowledge using the general structural principles which are characteristics of semantic networks?
2. Is it possible that the human performance of semantic processing can be emulated in terms of general processes operating on semantic networks?
3. Is it possible to emulate the human processes of semantic retrieval and search on general structures of semantic networks?
4. Can the semantic networks emulate the human processes of semantic acquisition and development of semantics in humans?

For all the above questions, the required answer is not only in terms of yes/no, but to what degree it is achievable. Also, the answer depends on the applications, which exploit the features of these networks, as well as there is yet lot to be found out.

The best example of a semantic network is *semantic web*, which enables the people to access the documents and services on the Internet. The interface to service is represented in web pages written in natural language, which must be understood and acted on by humans. The existing web is augmented by the semantic web with formalized knowledge and data, to be processed by computers.

## 7.3   Conceptual Graphs

Although there is no accepted standard for semantic network representations, but something which is very close to the goal is *Conceptual Graphs*. It is the portrayal of mental perception which consists of basic primitive concepts and relationships which exist between them. The conceptual graphs may be regarded as formal building blocks of Semantic networks. When they are linked together, they form a more complex and useful network [4].

Simmons [12] suggested primitives to represent standard relationships, by using the *case structure* of English verbs. In the verb oriented approach, links define the roles played by nouns and noun phrases inaction of the sentence. Case relationships includes: agent, object, instrument, location, and time [12].
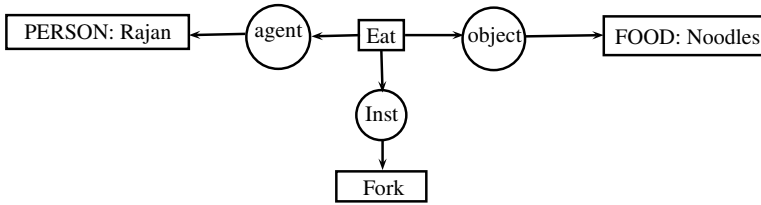
**Fig. 7.4**  Conceptual Graph-I

A sentence in the semantic network is represented with a *verb node*, and various case links to this node represent other participants in carrying out the action. The complete structure formed is called *case-frame*. While this sentence is parsed, the algorithm identifies the verb node, and retrieves the complete case-frame from the knowledge base. As a next step, the algorithm binds the values of an agent, object, etc., to appropriate nodes in the case-frame.

**Example 7.1**  Represent the sentence: "Rajan eats noodles with fork" as a conceptual graph.

The given sentence is represented by the conceptual graph shown in Fig. 7.4, and the corresponding predicate formula is given as Eq. 7.1.

$$\exists x \exists y (eat(x) \land person(Rajan) \land food(Noodles) \land fork(y) \land$$
$$agent(x, Rajan) \land object(x, Noodles) \land inst(x, y)) \quad (7.1)$$

The concept we have represented is called *typed* or *sorted* version of logic, each of the four concepts (i.e., Rajan, noodles, eat, and fork) have type labels, that represent the type of entity the concept refers. The two concepts, Rajan and Noodles, which have the names, identify the referent, e.g., the concept [PERSON:Rajan] has type PERSON and referent Rajan. Three concepts have type labels: Agent, Instrument, Object. The Conceptual Graph (CG) as a whole indicates the semantic that a person "Rajan" is an agent of some instance of eating the food, noodles are an object, and the fork is an instrument. Eat and Fork has no fields to refer them by name, as these are generic concepts.

$$[PERSON : Rajan] \leftarrow (AGENT) \leftarrow [EAT]-$$
$$\rightarrow (OBJECT) \rightarrow [FOOD : noodles]$$
$$\leftarrow (INSTRUMENT) \leftarrow [FORK] \quad (7.2)$$

The concept symbols may represent entities, actions, properties, or events. For the Fig. 7.4, a linear conceptual graph form, which is easier to represent as text is given in the Eq. 7.2. □

We note that the representation of an English language sentence in the language of CG in Eq. 7.2, captures much of the deep structures of the natural language, such as the relationship between the verb and its subject (called the agent relation), and that between verb and object. When this sentence is parsed, the built-in relationship indicates that "Rajan" is a person, who is eating, and the fork is used for eating noodles (as object). These linguistic relationships are stored independent of the actual sentence, and are independent of the language of the sentence.

In 1976, John Sowa developed the concept of the conceptual graph, as an intermediate language, that mapped natural language questions and assertions to a relational database. The concepts were represented using symbols of rectangles, and the circles represented as conceptual relations. An arc that pointed to a circle, marked the first argument of the relation, and an arc pointing away from the circle marked the last argument. The relation is expressed in mathematical form as, $(relation(arg_1, arg_2, \ldots, arg_n)$. If there is only one argument, the arrow-head is omitted, while for relation with two or more arguments, the arrow-heads are replaced by integers, $1, 2, \ldots, n$.

**Example 7.2** Represent the sentence "Rajan is going to Mumbai by bus", using a Conceptual Graph.

The Fig. 7.5 shows a CG for the sentence: "Rajan is going to Mumbai by bus."

In the CG all the four concepts, *Person*, *Go*, *Mumbai*, and *Bus*, have type label, for the type of the entity referred by the concept. Two concepts, Person and Destination have names. The verb "Go" is related to the remaining three concepts, by relations of agent, destination, and Instrument. The complete CG indicates that the person Rajan is an agent of some instance "Going", the city of Mumbai is the destination, and the bus is the instrument.

$$(\exists x)(\exists y)(Go(x)Person(Rajan)City(Mumbai)Bus(y)$$
$$Agent(x, Rajan)Dest(x, Mumbai)Inst(x, y)) \quad (7.3)$$

The Fig. 7.5 can be translated into the predicate formula (7.3). □
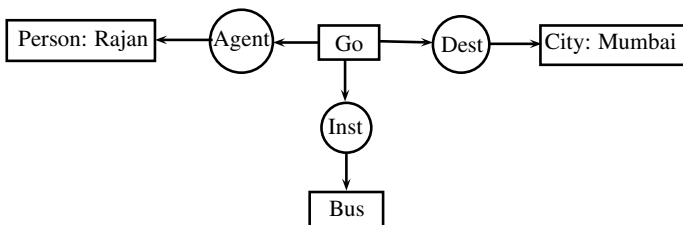


**Fig. 7.5** Conceptual Graph-II

The only logical operators used in Fig. 7.5 are conjunction and the existential quantifier, as expressed in the Eq. 7.3, which are the most common operators in translations from natural languages, to first-order predicate logic [14].

## 7.4  Frames and Reasoning

The frames are structure-based knowledge representation technique, and are similar to semantic networks. The latter is based on the concept of human associative memory, but may simply be thought of as data structures of nodes—"concepts"—and links—"associations"—between them [3].

The concept of a frame was proposed in the 1970s by Minsky. As per Minsky, when we encounter a new situation, or there is substantial change in the present context, we select from memory a new structure, called *Frame*. This is a previously remembered framework, which is adapted to fit into the current set of things as required for the new situation, with necessary changes in the frame. The frame is a *data structure* to represent a stereotype situation, like a certain kind of living room, or a frame of a picnic, or of the classroom, etc. Each frame is attached with several kinds of information, where some information may also be about how the frame is to be used, while other information may be about what one may expect to be happening next, and some information may be about what is to be done if these expectations are not met, and so on [8].

Consider representing the sentence: "Car #12 is red"

Approach 1: *red(car12)*. With this representation, it is easy to ask "what is red", but we cannot ask "what is the color of car12?"
Approach 2: *color(car12, red)*. In this approach, it is easy to ask "What is red?" Also, we can ask, "What is the color of car12?" But, we cannot ask "What property of car12 has value red?"
Approach 3: *property(car12, color, red)*. With this, it is easy to ask all the above questions.

We call this representation as, object-property-value representation, and have format *property(Object, Property, Value)*. To get the object-centered representation, we merge many properties of the object of the same type into one structure, as follows:
*property(Object, Property1, Value1)*
*property(Object, Property2, Value2)*
…
*property(Object, Property-n, Value-n)*

The representation is called Frame, as shown in Fig. 7.6.

It is important to note that objects enable grouping of procedures for determining the properties of objects, their parts, and interaction with parts. The first step in structuring is to collect together all propositions concerning a particular object in a data structure, like records in PASCAL, property lists in LISP, or relations in a

**Fig. 7.6** A frame object

Object

| Object |
|---|
| Property 1 |
| Property 2 |
| ... |
| Property n |
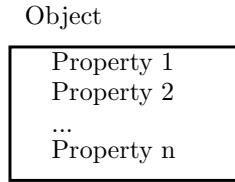
**Fig. 7.7** A frame of "Elephant"

| Object | Property | Values |
|---|---|---|
| Elephant | is-a: | mammal |
| | color: | grey |
| | has: | trunk |
| | size: | huze |
| | habitate: | India/Africa |

database. Figure 7.7 shows an example of a structured representation of facts by collecting together all the properties of an object in the data structure.

There are two types of frames: 1. The *Individual frames* represent a single object, e.g., a person, part of a trip; 2. Other type, the *Generic frames*, represent categories of objects, e.g., students. An example of a generic frame is, "Indian city", and individual frames are "Delhi", "Mumbai". An individual frame is a named list of *buckets*, also called *slots*. What goes in the bucket is called a *filler* of the slot.

```
(frame − name
< slot − name1 filler1 >
< slot − name2 filler2 > ...)
```

## 7.4.1 Inheritance Hierarchies

A frame is a network of nodes and relations, whose "top levels" are fixed, and represent the things that are always true concerning the supposed situation. The lower levels of this frame-based network comprise many terminals or "slots", which must be filled in by specific instances of the data. The conditions must also be specified for each terminal, under which the assignment be made. The assignments are usually smaller frames, called "sub-frames" (see Fig. 7.8). Simple conditions are indicated by markers, which might require a terminal assignment to be a person, an object of sufficient value, or a pointer to a sub-frame of a certain type. It is possible to specify relations among the things assigned to several terminals using more complex conditions.

The inheritance hierarchies serve for economic data conservation. Instead of storing all the properties of each object, all the objects are structured in a hierarchy, and only the individual properties are stored in the object itself, while the general properties are attached to the predecessors and inherited by all the successors.

**Fig. 7.8** A frame
representation of hotel room

| Frame |
|---|
| hotel room |
| is-a: room |
| location: hotel |
| contains:<br>{hotel chair,<br>hotel painting,<br>hotel mirror,<br>...} |
|  |

| sub-frame |
|---|
| hotel chair |
| is-a: chair |
| location: hotel room |
| ht: 3-5ft |
| type: swivel |
| use: for sitting |
| ... |
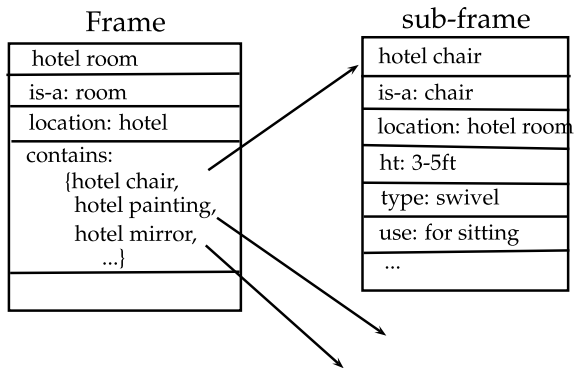
In object-centered representations, an object is a natural way to organize the knowledge about some physical objects, like "a desk has a surface-material, number of drawers, width, length, height, color, procedure for unlocking, etc." Some variations can be "no drawers, multi-level surface". Alternatively, an object may describe a situation, e.g., for a lecture-hall the complete set of situation is: hall, students, teacher, day, time, seating arrangement, lighting, grading, etc. Or, it can be about a trip with slot values as: origin (of trip), destination, procedures for buying ticket, transport, getting through customs, reserving a hotel room, locating a car rental, etc.

## 7.4.2  Slots Terminology

Every frame is identified by its individual name—the *frame name*; a frame consists of a set attributes associated with it (see Fig. 7.8). For example, in the frame "Person", slots may be: name, weight, height, and age; for the frame "Computer", the slots may be: model, processor, memory, and price. There is also a value attached to each attribute or slot. A frame-based approach provides a natural way of structured and compact knowledge representation. The knowledge is organized in slots that describe various attributes or properties of any object. This approach is appropriately suited for object-oriented programming of expert systems.

Following is the typical Information included in a Slot:

1. *Relationship*: A frame provides the relationship to the other frames. The frame *Hotel room* (Fig. 7.8) can be a member of other frame class *Room*, which in turn can belong to the class *Housing*, thus providing the relationship with these other room types.
2. *Slot value*: The value of a slot can be numeric, symbolic, or Boolean (True/False). For example, a slot identified as 'Person' is symbolic, with slot names as 'Age', and 'Height', both having float values. The slot's values can be dynamically assigned during a session with the expert system, or they can be static, or can be initialized in the beginning while the slot is created.

3. *Default value of slot*: A slot may contain default value when the true value is not available, and there is no evidence that the value chosen is in no way providing any contradiction. For example, in a frame named as *Car* when slot values are not provided, default values of the slots: *wheels-count* and *Engine-count* can be taken as 4 and 1, respectively.
4. *Slot value's range*: The range of a slot's value is useful in checking the bounds of the slot value—whether the provided value of a slot is within the prescribed limit? For example, a pressure range of a car tire may range 30–50 psi (pounds per inch).
5. *Slot Procedure*: A slot has a procedure attached to it; when this procedure is called it may read a value from the given slot, or it can update the value of the slot (write the value in it).
6. *Facets*: The facets provide an extension to slot value structure in a frame-based expert system. A facet provides extended knowledge about the attribute of a frame. It can be used for establishing the attributed value of a frame, it can control end-user queries, and can direct the inference engine as how to process the attributes.

### 7.4.3   Frame Languages

Frame-based languages are knowledge representation languages, where a frame designates a concept or a concrete entity. A concept is represented by a generic frame, called class, and concrete entity illustrates one or more classes. These classes are also represented by a specific frame, called *instance*. A frame, whether it is a class or instance, is a data structure composed of slots which carry the properties of the entity described by the frame or relations between frames.

The slots themselves are described using facets, which contribute to the description semantics of the slot. The facets are two types: 1. descriptive (passive), and 2. active (reflexes). The passive faces represent domain, vales, or defaults. The active facets (also called *daemons* or *procedural* facets) are concerned to actions. They start with keywords like, If-needed, If-created, which gets triggered after the value of a slot is manipulated. The slots are characterized either by vales introduced by the *value facet*, or default values, which reintroduced by the *default facets*.

The classes are organized as a hierarchy, and relations called structural links, that connect the frames. A commonly used link, ako (a-kind-of) connect the classes within the hierarchy, whereas "Is-a" link allows to connect instances to classes to which they belong. A class connected by a link ako to another class, called *mother-class*, inherits all the properties possessed by the mother-class. These properties are represented using slots. The presence of a value within a class slot means this value is true for this slot, as well as it is true for all the sub-classes and the instances of that class, where the value has been declared. Specifying a default value in a class means this value is generally true for that slot, also for the sub-classes, and the instances of that class where the default is declared. However, the default can be overridden by declaring an exception to the default, at any of the sub-class(es).

It is assumed that there is no multiple inheritance conflicts among these slots. The frames of the hierarchy communicate by means of messages. The frames make use of methods (processes), which gets executed when a message is received by the frame in which this message is declared or when the message is received by its sub-classes or instances.

A domain knowledge base builder uses the frame languages to describe the type of objects to be modeled by the system, and this representation can be provided efficiently by the frame languages. The object description of certain type may contain a prototype description of objects of that type. These objects are used for creating a default description of an object when its type becomes known in the model.

The frame-based languages are good for providing facilities for describing object attributes. For example, a frame representing a *car* might include descriptions for the length of the car, number of seats, size of the engine, engine's horse-power, etc. These attributes can be used to include partial descriptions of the attribute values, and are helpful in preserving the semantic integrity of a system's knowledge base by restricting the number and range of permitted attribute values. The frame languages do not provide a facility for describing the declarative behavior, however, they have facilities for attaching procedural information expressed in some other language, e.g., LISP. The procedural capability enables behavioral models of objects, and as an expert in an application domain. It also provides a powerful tool for object-oriented programming, where frames are treated as objects, and they respond to messages.

There are two standard forms for attaching the procedures: 1. *methods*, and 2. *active values*. In languages, like LISP and Prolog, the methods are used as procedures, that are attached to the frames, and respond to messages sent to the frames. Just like the attributes, the methods are also stored as values in slots, and they are recognized as responders to the messages. The messages sent to the frames carry the information about target message-responder slot, these messages contain the arguments needed by the methods stored in the slot. The "actives values" in the slots are either procedures or collection of production rules attached to the slot. These (procedure or rules) are invoked when the slot's values are accessed or new values are stored in the slots. Thus, these slots behave like "daemons", and monitor the change and usages of the values.

The active values can also be used to dynamically compute values on a "when-needed" basis. The methods and active values are so written that they apply to any member of a class of objects, and are included by the knowledge base designer in the class description as a part of the prototype description of a class member.

### 7.4.4  Case Study

Many systems make use of the approach discussed above to control the reasoning, where functions or rule classes behave like daemons, are attached to the slots of the frames. The attachments get invoked when the value of a slot changed. That way, they behave like sensors or alarms, or monitors. For example, an expert system is used for

**Fig. 7.9** A satellite diagnostic system
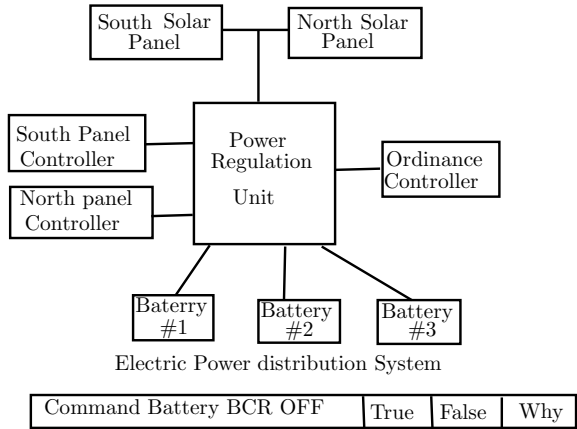
an intelligent alarm facility, and this calls a user supplied function only when a slot's value crosses the threshold. The user may establish an alarm by providing critical boundaries, and an alarm function. The system stores the boundaries and function as facets of the slots, and may attach generic active value for check of boundary crossing, whenever a value of the slot changes.

An example of the above phenomenon can be found in the knowledge system (see Fig. 7.9), which is a Satellite diagnostic system . The system is meant to serve as an intelligent facility for human operators to perform diagnostic and trouble-shooting of satellite malfunctions. Such systems can also be used as a simulator to train the operators and diagnostic experts. An important requirement of the diagnosis task is that it requires detailed analysis of the satellite system through a diverse set of domain experts. The software architecture of the prototype, which can be built using an expert system, will make use of daemons attached to the slots to respond to messages sent to objects. The prototypes of experts have the property that they can be instantiated, as well as deleted dynamically, as per the need, during the operation of the system.

Significant events can be controlled. This system makes elaborate and effective use of frames, including prototype expert frames, which also, like a daemon, can be instantiated and deleted dynamically as per the requirement, during operation of the system. The use of frame-based approach to build the system's model allowed the designers to organize the frames in such a way, that knowledge could easily be accessible and comprehensible to diagnostic experts, as well as the satellite operators [9].

A satellite diagnostic, e.g., STAR-PLAN, is designed with several requirements, that resulted in an architecture based on the integration of frames and production rules, with the following important features:

1. First, the system's knowledge is made accessible and comprehensible to the satellite operators as well as to diagnostic experts.
2. The system is incrementally and progressively built-up, as the descriptions of additional satellite modules become ready, and as the needed experts became

available. Accordingly, the system's knowledge is required to be partitioned into a limited number of experts, for example, knowledge about a particular type of malfunction, or about a particular module of the satellite.

3. In the case of STAR-PLAN, it was known to the designers that eventually, the system will become very large, and it would be operating in real-time. To meet the speed requirements, the system was so designed that only part of the system shall be awakened—the one which is required at a certain moment of time, and other shall be put to sleep.

Role of Frame Language

The designers of STAR-PLAN used frame language to build the taxonomy that described parts of a typical communication satellite. The daemons and methods were later associated with the prototype in the taxonomy. This maintained the relationship between various parts, and defined the behavior of each part, which resulted in a kind of object-oriented programming style, built for creating simulation behavior.

To model the diagnostic part, two separate taxonomies were constructed, such that each class in the first taxonomy represented experts who were assigned the task of "watching over" to some particular components of the satellite. The members of these classes were called *Guardians*. In the second taxonomy, each class represented experts responsible for responding to a particular type of problem that may occur in the satellite. The members of the class in this second taxonomy were called *Monitors*.

For each component of the satellite, guardians were created and initialized at the start of the satellite system. When an *initialize* message is sent to a guardian, it places an intelligent alarm in the system's model of the satellite. These alarms would wake-up their guardian by sending a message when a problem has occurred in the satellite. Hence, the guardian is active only when the satellite demands.

Alarms/Inferencing

The methods of *guardian* system respond to messages from the *daemons*. This is done by invoking a class of diagnostic rules that determine what kind of problem is occurring. For example, to find out all the possible consequences of an anomalous situation that might have tripped the alarms, the set of rules is applied in forward-chaining manner. In the process, all the rules are applied that have conditions matching some aspects of the anomalous situations, or it matches a conclusion of the already applied rule. This process of rules' application continuous until no match is left.

The total number of rules associated with this system are small in number, typically, 10–20, so that modularization provides a small expert system of closely related rules to be focused to develop a guardian.

As soon as a guardian comes to know about the occurrence of a problem, it creates a*monitor* and initializes it. This represents an expert for the problem. The monitor's task is to watch the problem right from its evolution, and to make the recommendations to the satellite operator. Once the initialization is done, the monitor may create its own daemons and put itself to sleep for a fixed duration. When the monitor wakes up itself or it is awakened by a message from one of its daemons, it invokes a set of rules to analyze the status of the satellite. If the monitor

is waking-up itself after a fixed amount of time, the rules will be invoked by a backward-chaining rule interpreter that tests a specific hypothesis (goal) about the problem. The backward-chaining system attempts to find a sequence of rule applications that should conclude the hypothesis. When such a sequence is found, the rules are applied so that the hypothesis is added into the knowledge base.

Based on the conclusions arrived-at making use of rules, the monitor will either put itself to sleep again or make recommendations to the operator. When the monitor concludes that the problem has been solved, it removes its daemons and then removes (deletes) itself, and frees the memory occupied.

Due to the creation and deletion of monitors in a dynamic way, the satellite problems by the STAR-PLAN system model are actually handled like by human operators and experts. The situation is like in real-world: when a problem is identified, a suitable expert is called in, which works with the team until the problem is resolved, and then the expert leaves! The monitor's rule sets are organized for problem specific knowledge base about the problems of the satellite. The module-based system and its organization structure also makes it easier for a (human) domain expert to create and debug the knowledge base of rules.

## 7.5 Description Logic

Approaches to knowledge representation are sometimes divided roughly into two categories: *logic-based* formalisms, which evolved out of the intuition, that predicate calculus could be used unambiguously to capture facts about the world; and other, non-logic-based representations. The latter was often developed by building on more *cognitive notions*—for example, network structures, and rule-based representations derived from experiments on *recall* from human memory and human execution of tasks like mathematical puzzle solving. Even though such approaches were often developed for specific representational chores, the resulting formalisms were usually expected to serve in general use [5].

Since first-order predicate logic (FOPL) provides very powerful and general machinery, logic-based approaches were more general purpose from the very start. In a logic-based approach, the representation language is usually a variant of the first-order predicate calculus, and reasoning amounts to verifying logical consequence. In the *non-logical* approaches, often based on the use of graphical interfaces, knowledge is represented by means of some ad hoc data structures, and reasoning is accomplished by similarly ad hoc procedures that manipulate the structures. Among these specialized representations we find semantic networks and frames. However, frames and semantic networks lack formal semantics. Description Logic ($\mathscr{DL}$) was first introduced into Knowledge Representation (KR) systems to overcome these deficiencies of semantic networks and frames. The $\mathscr{DL}$ makes it easier to describe definitions and properties of categories. The $\mathscr{DL}$ evolved from semantic networks to formalize the network representation while retaining the emphasis on taxonomic structures as an organizing principle.

A Description Logic models *concepts*, *roles* and *individuals*, and their relationships. The fundamental modeling concept of a $\mathscr{DL}$ is the axiom: "a logical statement relating roles and/or concepts". $\mathscr{DL}$ is a family of formal knowledge representation languages, which is more expressive than propositional logic and has more efficient decision properties than first-order predicate logic. It is used in formal reasoning on the concepts of an application domain (known as terminological knowledge). It is used for providing a logical formalism for ontologies and the Semantic Web. Modern ontology languages are based on $\mathscr{DL}$, such as OWL (Ontology Web Language).

## 7.5.1 Definitions and Sentence Structures

A $\mathscr{DL}$ describes the domain in term of the following:

- *Individuals*—are the things in the world that are being described. (For example a house, book, ram, john, rita, etc, all starting with lowercase letters).
- *Classes/Categories/Roles*—are sets of individuals. It is a *ako* (a kind of) concept. A class is a set of all real or potential things that would be in the class. For example, Hunter, Teenager, etc.
- *Properties/Relations*—are used to describe individuals. It is *ako* Roles or relational nouns, and used to describe objects that are parts or attributes or properties of other objects. Examples are: Child, Mother, Age, etc.

 Two different sets of symbols—*logical symbols* (with a fixed meaning) and *non-logical symbols* (domain-dependent) are used in the Description Logic.
 Following classes of *Logical symbols* are used in $\mathscr{DL}$:

Punctuation: (,), [,]
Positive integers
Concept-forming operators: $\forall$, $\exists$, *FILLs*, *AND*.
Connectives: $\sqsubseteq$, $\doteq$, $\rightarrow$, $\sqcup$, $\sqcap$.

*Non-logical symbols*:

- Constants: `john`, `rajanShaw` (camel casing, but starting with uncapitalized letter.
- Atomic concepts: `Person`, `FatherOfOnlyGirls`, `Hunter`, `Teenager` (camel casing, first letter capital).
- Roles: `:Height`, `:Age`, `:FatherOf`, `:Child`, `Mother` (same as concepts, but precede by colons).

Concepts

In terms of semantics, the concepts are given set-theoretic interpretation, where a concept is interpreted as a set of individuals, and roles are a set of pairs of individuals. An interpretation domain can be chosen arbitrarily, which can be infinite also. The infinite domain and the open-world assumptions are distinctive features of Description Logic.

### *7.5.2 Concept Language*

Atomic concepts are thus interpreted as subsets of the interpretation domain, while the semantics of the other constructs is then specified by defining the set of individuals denoted by each construct. For example, the concept $C \sqcap D$ is the set of individuals obtained by intersecting the sets of individuals denoted by $C$ and $D$, respectively. For example, *Female $\sqcap$ Teacher*. Similarly, the interpretation of $\forall R.C$ is the set of individuals that are in the relationship $R$ with individuals belonging to the set denoted by the concept $C$. For example, $\forall ResidentsOfJodhpur.Students$ represents all the students who are residents of Jodhpur.

There exists some ambiguity also, due to the natural language being the source, where many nouns can be used to refer as a category as well as relations. For example, `child` can be used as a *category*, i.e., a very young person, it can also be used to represent a relation, which stands for the inverse of `parent`.

An important feature of Description Logic is to define complex concepts in terms of simpler ones. This is achieved by means of *concept-forming operators*: $\exists$, $\forall$, *AND*, *FILLs*. A *complex concepts* is defined as [4, 10]:

Every atomic concept is a concept;
If $R$ is a role and $C$ is a concept, then $\forall R.C$ is a concept;
If $R$ is a role and $n$ is a positive integer, then $\exists n.R$ is a concept;
If $R$ is a role and $C$ is a constant, then *FILLs R.C* is a concept; and
If $C_1 \ldots C_n$ are concepts, then *AND* $C_1, \ldots, C_n$ is a concept.

The symbol $\exists$ stands for the class of individuals in the domain that are related by relation $R$ to at least $n$ other individuals. The following can be created as complex concepts:

$\exists$ 1.*Child*: All the individuals (the class of the individuals) that have at least one child.

$\exists$ 2.*HasCar*: All the individuals that have at least two cars.

$\exists$ 6.*HasWheels*: All the individuals that have at least six wheels.

The *FILLs R.C* stands for those individuals that are related (*R-related*) to the individual identified by $C$. For example, "All the individuals that have the car with plate *RJC12* is represented by *FILLs HasCar.RJC*12.

The $\forall$ *R.C* stands for those individuals that are *R-related* only to individuals of class $C$. For example, $\forall$ *BeingInThisRoom.PHDStudents* represents "All the individuals that are in this room and are Ph.D. students."

The full syntax of a concept in $\mathscr{DL}$ is:

$$Concept : -Thing|conceptName$$
$$|AND(concept_1, concept_2, ...)$$
$$|\forall RoleName.concept$$
$$| \leq ingtger.RoleName$$
$$| \geq integer.RoleName$$
$$|FillsRoleName.IndividualName$$
$$|SameAs(Path, Path)$$
$$|\exists IndividualName.concept|\top|\bot$$
$$Path : -[RoleName, ...]$$

The family of concept languages is called Attribute Language ($\mathscr{AL}$) , which is minimal language that is of practical importance. Given that *Person* and *Female* are both atomic concepts, then *Person* $\sqcap$ *Female* and *Person* $\sqcap \neg Female$ are $\mathscr{AL}$-concepts which intuitively describe, persons that are female, and those that are not female. Suppose that *hasChild* is atomic role, then *Person* $\sqcap \exists hasChild.\top$, and *Person* $\sqcap \forall hasChild.Female$ denote those persons that have a child, and all those whose children are female. Opposite to the *top*, there is a *bottom* concept ($\bot$), using which we can describe the persons without a child: *Person* $\sqcap \forall hasChild.\bot$.

Sentences

A knowledge base in a Description Logic is collection of sentences like,

If $C_1$ and $C_2$ are concepts, then ($C_1 \sqsubseteq C_2$) is a sentence;

If $C_1$ and $C_2$ are concepts, then ($C_1 \doteq C_2$) is a sentence;

If $C$ is a constant and $D$ is a concept, then ($C \rightarrow D$) is a sentence;

For example,

*PhDStudent* $\sqsubseteq$ *Student*, i.e., Every Ph.D. student is also a student (not vice versa).

$C_1 \doteq C_2$, i.e., concept $C_1$ is equivalent to concept $C_2$ , i.e. the individuals that satisfy $C_1$ are precisely those that satisfy $C_2$.

*PhDStudent* $\doteq AND(Student, Graduated, HasFunding)$, i.e., a Ph.D. student is a student that is already graduated, and that has some funding.

$C \rightarrow D$, i.e., the individual denoted by $C$ satisfies the description expressed by concept $d$. For example, *rajan* $\rightarrow$ *PostDoc*, i.e. "Rajan is a Post Doc."

When compared with FOPL, the FOPL focuses on sentences, and it does not help you with reasoning on complex categories. For example, we can say that $X$ is a hunter by a 1-ary predicate *Hunter*$(X)$. Similarly, there is 1-ry predicate, we can say *Shooter*$(X)$. What if we want to say is that $X$ is both a hunter and a shooter. In predicate logic, it is

$$Hunter(X) \wedge Shooter(X),$$

whereas in $\mathscr{DL}$ it is a 2-ary relation

$$Hunter\&Shooter(X).$$

or

$$AND(Hunter, Shooter).$$

In the $\mathscr{DL}$, intersection of concepts, which is denoted $C \sqcap D$, is used to restrict the set of individuals under consideration to those that belong to both $C$ and $D$. In the syntax of $\mathscr{DL}$, concept expressions are variable-free. In fact, a concept expression denotes the set of all individuals satisfying the properties specified in the expression. Therefore, $C \sqcap D$ can be regarded as the first-order logic sentence, $C(x) \wedge D(x)$, where the variable ranges over all individuals in the interpretation domain and $C(x)$ is true for those individuals that belong to the concept $C$.

We can represent the concept of "persons that are not female" and the concept of "individuals that are female or male" by the expressions:

$$Person \sqcap \neg Female$$

and

$$Female \sqcap Male.$$

The key characteristic features of $\mathscr{DL}$ lies in the constructs that are helpful for creating relationships between concepts. The most common and elementary is the *value restriction*, written as $\forall R.C$, which means all the individuals that are having relationship $R$ with the concept being described, belong to concept $C$. Similarly, $\exists R.C$ is a value restriction for some individuals.

**Example 7.3**  Represent the concepts of semantic network form in Fig. 7.10, using $\mathscr{DL}$.



**Fig. 7.10**  Semantic network hierarchy
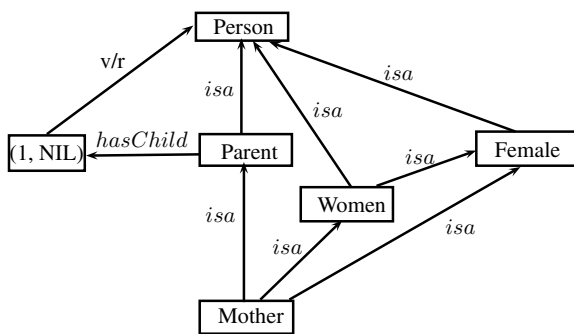
Assume that the atomic concepts are: Female, Person, and Woman. And, hasChild, hasFemaleRelative are atomic roles. They use the operators *intersection*, *union* and *complement* of concepts, interpreted as set operations.

The Description Logic has a characteristic feature as their ability to represent other kinds of relationships that can hold between concepts, this is beyond the IS-A relationships. For example, in Fig. 7.10, the concept of *Parent* has a property that is usually called a "role," and expressed by a link from the concept to a node for the role, labeled hasChild. The role has a "value restriction," denoted by the label $v/r$, which expresses a limitation on the range of types of objects that can fill that role. In addition, the node has a number of restrictions expressed as (1, NIL), where the first number is a lower bound on the number of children and the second element is the upper bound, and NIL denotes no restriction on the upper limit. Overall, the representation of the concept of *Parent* here can be read as "A parent is a person having at least one child, and all of his/her children are persons", thus the role link between Parent and Person in Fig. 7.10 can be expressed as a concept expression $\mathscr{DL}$ as,

$$\exists 1.\texttt{hasChild}.Parent \sqcap \forall\texttt{hasChild}.\textit{Person}.$$

Existential quantification and value restrictions are thus meant to characterize relationships between concepts. Such an expression therefore characterizes the concept of *Parent* as the set of individuals having at least one filler of the role hasChild belonging to the concept Person; moreover, every filler of the role hasChild must be a person.

Relationships are inherited from concepts to their subconcepts. For example, the concept Mother, i.e., a female parent, is a more specific descendant of both the concepts Female and Parent, and as a result inherits from Parent the link to Person through the role hasChild; in other words, Mother inherits the restriction on its hasChild role from Parent. The other relations are translated as:

$$\texttt{Woman} \doteq \texttt{Person} \sqcap \texttt{Female}.$$
$$\texttt{Mother} \doteq \texttt{Woman} \sqcap \texttt{Female} \sqcap \texttt{Parent}.$$
$$\forall\texttt{Female}.\texttt{Person}$$
$$\forall\texttt{Parent}.\texttt{Person}.$$

we observe that there may be implicit relationships between concepts. For example, if we define Woman as the concept of a *female person*, it is the case that every Mother is a Woman. It is the task of the knowledge representation system to find implicit relationships such as these (many are more complex than this one). Typically, such inferences have been characterized in terms of properties of the network. In this case, one might observe that both Mother and Woman are connected to both Female and Person, but the path from Mother to Person includes a node Parent,

which is more specific then Person, thus enabling us to conclude that Mother is more specific than Person. □

### 7.5.3 Architecture for $\mathscr{DL}$ Knowledge Representation

A knowledge representation system based on Description Logic provides facilities to set up knowledge base, to reason about their concepts, and manipulate them. The Fig. 7.11 shows the related blocks and their interactions for this purpose.

The KB comprises two components, the *TBOX* (terminology Box) introduces the terminology, i.e., the vocabulary of an application domain; and the *ABOX* (assertion box) contains assertions about the named individuals in terms of vocabulary.

The vocabulary consists of concepts, which denotes the individuals, and the roles which denote the binary relationship between the individuals. In addition to these, $\mathscr{DL}$ system allows the users to build a complex description of concepts and roles. The *TBOX* can be used to assign names to complex descriptions. The description language has model theoretic semantics. Consequently, the semantics in *ABOX* and *TBOX* are FOPL formulas or its extensions.

The $\mathscr{DL}$ system provides the services for reasoning using using KB, typically, to reason if the terminology is satisfiable. The reasoning process checks that the assertions are consistent. With subsumption testing, it is easy to organize the concepts of terminology in the hierarchy.

In any application, the KR system is embedded into a large environment. The other components interact system through queries to KB and by modifying it, i.e., by adding or retracting concepts, roles, and assertions.

The basic form of declaration in a *TBox* is a concept definition, that is, the definition of a new concept in terms of other previously defined concepts. For example, a woman can be defined as a female person by writing this declaration:

$$\text{Woman} \doteq \text{Person} \sqcap \text{Female}$$

**Fig. 7.11** $\mathscr{DL}$ based knowledge representation architecture
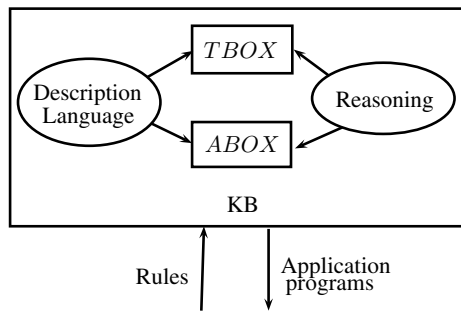
There are some important common assumptions usually made about $\mathscr{DL}$ terminologies:

- Only one definition for a concept name is allowed;
- Definitions are acyclic in the sense that concepts are neither defined in terms of themselves nor in terms of other concepts that indirectly refer to them.

The *ABox* comprises extended knowledge about the domain of interest, which are, assertions about individuals, called membership assertions. For example,

$$\texttt{Female} \sqcap \texttt{Person(sita)}$$

show that the individual named as `sita` is a female person. Given this definition of woman, one can derive from this assertion that `sita` is an instance of the concept Woman. Similarly,

$$\texttt{hasChild(sita, luv)}$$

indicates that `sita` has `luv` as a child. Assertions of the first category are also called *concept assertions*, while of the second is called *role assertions*.

### 7.5.4  Value Restrictions

Let us now turn our attention to role restrictions by looking first at the quantified role restrictions and, subsequently, at what we call "number restrictions." Most languages provide (full) existential quantification and value restriction that allows one to describe, for example, the concept of "individuals having a female child" as $\exists\texttt{hasChild.Female}$, and to describe the concept of "individuals all of whose children are female" by the concept expression $\forall\texttt{hasChild.Female}$. In order to distinguish the function of each concept in the relationship, the individual object that corresponds to the second argument of the role viewed as a binary predicate is called a role filler. In the above expressions, which describe the properties of `Parents` having female children, individual objects belonging to the concept `Female` are the fillers of the role `hasChild`.

Another important kind of role restriction is given by number restrictions, which restrict the cardinality of the sets of fillers of roles. For instance, the concept

$$(\geq 3\,\texttt{hasChild}) \sqcap (\leq 2\,\texttt{hasFemaleRelative})$$

represents the concept of "individuals having at least three children and at most two female relatives." Number restrictions are sometimes viewed as a distinguishing feature of Description Logics, although one can find some similar constructs in some database modeling languages (notably Entity-Relationship models).

Beyond the constructs to form concept expressions, Description Logics provide constructs for roles, which can, for example, establish role hierarchies. However,

the use of role expressions is generally limited to expressing relationships between concepts.

*Intersection* of roles is an example of a role-forming construct. Intuitively, `hasChild ⊓ hasFemaleRelative` yields the role "has-daughter," so that the concept expression

$$\text{Woman} \sqcap \leq 2 \, (\text{hasChild} \sqcap \text{hasFemaleRelative})$$

denotes the concept of "a woman having at most 2 daughters".

**Example 7.4** Represent the following statement in Description Logic: A cheese pizza is defined as a pizza having toping and having a pizza base. The topping is a cheese topping, while the base is a pizzabase. A cheese topping is a topping.

$$
\begin{aligned}
\text{CheesePizza} = {}& \text{Pizza} \\
& \sqcap (\exists \text{hasTopping.CheeseTopping}) \\
& \sqcap (\exists \text{hasPizzabase.PizzaBase}). \\
\text{cheeseTopping} \sqsubseteq {}& \text{Topping}. \qquad\qquad\qquad \square
\end{aligned}
$$

### 7.5.5  *Reasoning and Inferences*

The basic inference on concept expressions in Description Logics is *subsumption*, typically written as $C \sqsubseteq D$ (read as "C is subsumed by D"). Sometimes, this axiom type is also referred to as *is-a* relationship, inspired by the often chosen wording for this type of statement (e.g. "a cat is a mammal" would be a typical verbalization of `Cat ⊑ Mammal`). Determining subsumption is the problem of checking whether the concept denoted by $D$ (the *subsumer*) is considered more general than the one denoted by $C$ (the *subsumee*). In other words, subsumption checks whether the first concept always denotes a subset of the set denoted by the second one.

The principle inferences of $\mathscr{DL}$ are *subsumtion*—checking if one category is a subset of another category, and classification checking, whether an object belongs to a category.

For example, one might be interested in knowing whether `Woman ⊑ Mother`. In order to verify this kind of relationship, one has, in general, to take into account the relationships defined in the terminology in Fig. 7.10.

Given a knowledge base expressed as a set $S$ of sentences:

"Does a constant $c$ satisfies concept $d$?"
"Is a concept $c$ subsumed by a concept $d$?"

Answering to these questions amount to compute the entailment. For example, representation for "A Ph.D. student is, a student that already graduated, and that has some funding." is:

PhDStudent $\doteq$ AND(Student, Graduated, HasFunding).

As another example, to say that "Bachelors are unmarried adult males", we write in $\mathscr{DL}$ as

Bachelor $\doteq$ Unmarried $\sqcap$ Adult $\sqcap$ Male

The most important aspect of $\mathscr{DL}$ is its emphasis on tractability of inference. A problem instance is solved by designing it and then asking if it is subsumed by one of several possible solution categories. The complexity of $\mathscr{DL}$ is far simpler than FOPL. The $\mathscr{DL}$ usually also lacks the negation and disjunction operators.

The main application domains of description logic are: software engineering, configuration of large software, digital libraries, Web-based information systems, Planning, and Data Mining.

## 7.6  Conceptual Dependencies

The Conceptual Dependency (CD) framework is a simplified linguistic system, aimed to provide a computational theory of simulated performance. In Conceptual Dependency terms, the linguistic process is a mapping into and out of some mental representation. This mental representation consists of concepts related to each other by various meaning-contingent dependency links. Each concept in the interlingual network may be associated with some word that is its realization on a sentential level.

A representation using conceptual dependency is a linked network, which characterizes the conceptualization inherently present in a piece of language, with reference to a real-world scenarios. A simple rule for representing concepts as dependent on other concepts is, to check whether the dependent concept further explains its governor, and this concept cannot make sense without its governor. For example, in the sentence,

"The big man steals the red book from the girl.",

the analysis of the sentence is as follows: The article, 'The' stands for connecting sentences in paragraphs, i.e., 'The' also specifies that 'man' might have been used previously. The adjective, 'Big' refers to the concept 'big', which cannot stand independently. The concept 'man', however, can stand alone, but in the above sentence it is conceptually modified by 'big', and in the network, it is realized as *governor* with its dependent. The verb 'steals' is an *action*, which is dependent on the concept of doing the acting. A conceptualization (a statement about a conceptual actor) cannot be complete without a concept acting (or an attribute statement). Thus, to complete a two-way dependency, a link must exist between the 'man' and the 'steal'.

**Fig. 7.12** CD for "The big man steals the red book from the girl"
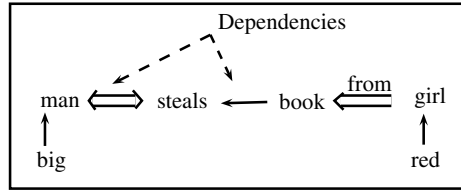
Which indicates that they are dependent on each other, and also govern each other as shown in Fig. 7.12.

It is mandatory that every conceptualization has a two-way dependency link. In the Fig. 7.12, the concept 'Book' governs the concept 'red' attributively (color is an attribute), and the whole entity is placed as objectively dependent on 'steals'. The construction "from the girl" is realized as being dependent on the action through the conceptual object. This is *prepositional* type of dependency (denoted by ⇐). There are different forms of this prepositional dependency, each of which is expressed by writing the preposition over the link, that indicates the prepositional relationship.

A language may use *inflections*, or nothing may be used instead of prepositions to indicate prepositional dependency. Here we will discuss a language-free system, which represents the relation of the parts conceptually.

The CDs are intended to be used in reasoning with natural language constructs, independent of any language or the phrases in the language. This has resulted in a small number of primitive actions, about 10–12, and a set of dependencies which connect the primitive actions with each other and with their actions, objects, instruments, etc.

The CDs have two main objectives:

1. If the meaning of any two sentences is the same, they should be represented the same CD, regardless of the particular words are used in these.
2. Any information, which may be present implicitly in the sentence, should be represented explicitly in the CD.

For item 1 above, the examples are 'get' and 'receive', both will have the same CD representation. For item 2, the machine must extract the implicit part from the sentence.

The CDs have:

1. a set of primitive actions,
2. a set of states for representation and result of the action,
3. a set of dependencies, or conceptual relationships, which could exist between primitives, states, and objects.

The representation of English sentences could be constructed by joining together the building blocks to form a CD graph. The CD provides four *conceptualization primitives* using which the world of meaning is built. These are:

ACTs: Actions
PPs: Picture producers(objects)
AAs: Action Aiders (they modify the actions)
PAs: Picture Aiders (they modify the objects)

All the actions are assumed to reduce to one or more of the primitive ACTs, out of the following actions only [7]:

PROPEL: Apply physical pressure to an object (push)
MOVE: Move body parts by owner
GRASP: Grab an object by actor(grasp)
ATRANS: Transfer of relationship (give)
PTRANS: Transfer of physical location of an object (go)
INGEST: Ingest an object by an animal (eat)
EXPEL: Expel from an animal's body (cry)
MTRANS: Transfer mental information(tell)
MBUILD: Mentally make a new information(decide)
ATTEND: Focus sense organ
CONC: Conceptualize or think about an idea (think)
SPEAK: Produce sound (say)

At the conceptual level, a CD framework is responsible for representing the meaning of a piece of written language in language-free terms. The conceptualization is written on a straight line, in a conceptual dependency analysis. The dependents written perpendicular to the line are attributes of their governor, except when they are part of another conceptualization line. The whole of conceptualizations can relate to other conceptualizations as actors or attributes.

Each primitive comprises a set of slots associated with it, from the set of conceptual dependencies. Associated with each slot are restrictions as to what sorts of objects could appear in that slot. For example, the following are slots for PTRANS.

ACTOR: It is either human or animate object, that initiates the PTRANS
OBJECT: It is a physical object, that is moved (PTRANSed)
FROM: The PTRANS begins at this location
TO: PTRANS ends at this location

Figure 7.13 shows the basic roles of conceptualization primitives of CD, and Fig. 7.14 shows a general case of primitives along with an example.

The inference rules are written based on these primitives to make explicit the information which is implicitly presented in the English or any other language text. For example, using the primitive PTRANS, it is possible to make inference about the OBJECT that was PTRANSed (physically transferred), which was initially at the FROM location, and after the PTRANS is carried out, it is at the TO location. The same inferences shall be made no matter what type of PTRANS was present, like, flying, driving, walking, falling, etc.
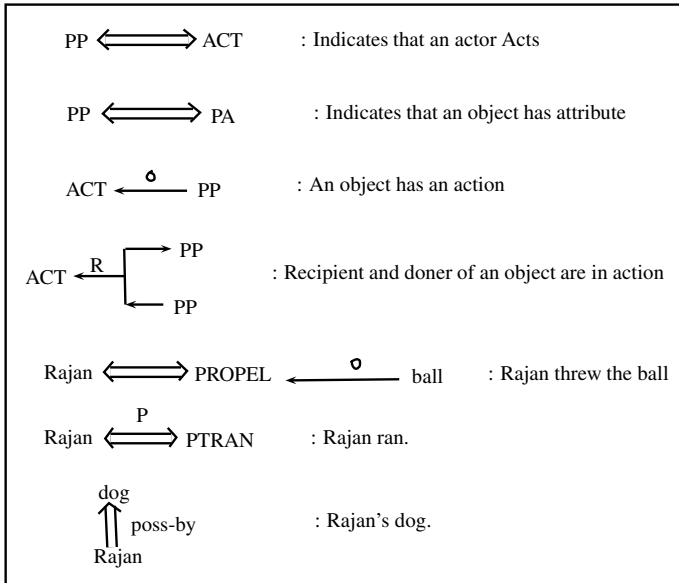
**Fig. 7.13** Conceptual dependency representations
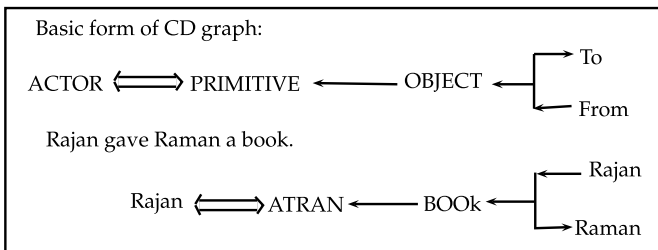


**Fig. 7.14** A CD graph

## 7.6.1 The Parser

The CD framework can be used for natural language parsing. A CD-based system analyzes sentences into their conceptual representation by operating on pieces of every sentence, and lookup for potential conceptual cases. All the conceptualizations are checked against a list of expressions to see if that part of the concept has occurred before. Those concepts which never occurred, are meaningless in the system. Consider the sentence: "tall boy went to the park with the dog." Part of the parser output is as shown in Fig. 7.15.

In this sentence, the problem is, where to attach the concept "with dog"? Is it to the "park" or to the "tall boy"? The problem can be solved by conceptual semantics. The semantics for 'go' contains a list of conceptual prepositions. Under the preposition

**Fig. 7.15** Parser output for "tall boy went to the park"
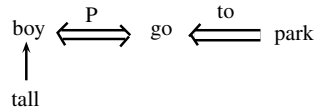


**Fig. 7.16** CD for parsing a sentence: tall boy went...
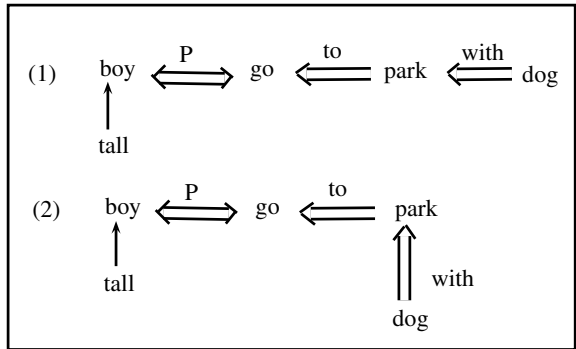


'with' there is a description: "any movable physical object", and since the dog is a physical object, the dependency is applicable. As per the sentence, the parse tree (1), in Fig. 7.16 is allowed, while (2) is rejected.

The parser tries to analyze a sentence in a way analogous to human method. It handles input one word at a time as it is encountered, checks potential linkages with its own knowledge of the world and past experience, and puts its output into a language-free formalism that can be acted on.

The CD parser is a conceptual analyzer rather than syntactic parser.

Consider a sentence: "big boy gives apples to pig". The input sentence is processed word by word, and parsed into CD as shown with steps in Fig. 7.17.
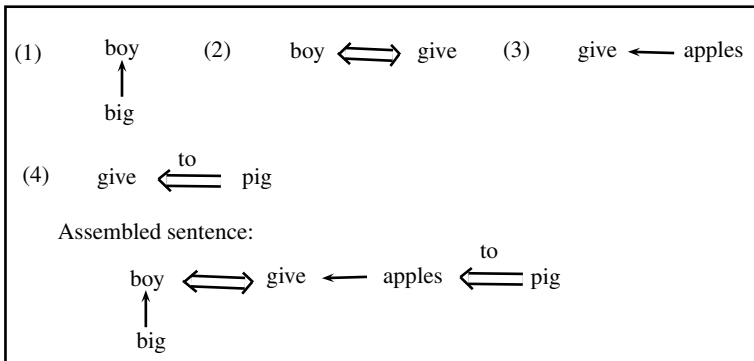


**Fig. 7.17** Steps for parsing using CD

## *7.6.2 Conceptual Dependency and Inferences*

The representation of text in canonical form has the benefit to perform inference using that text, because the canonical form allows to write inference rules as general as possible. If the representation does not capture the similarities in the meaning of the text, the rules about what can be inferred from a given text needs be duplicated by writing one rule for every form of the representation, even though they may have the same meaning relevant to the inference. To illustrate that inference is facilitated by CD, the inferences are used to build a "causal chain" to connect the events like in a story, as in the following sentence.

> Simon hit Anne. Anne's mother took Anne to the hospital. Anne's mother called Simon's mother. Simon's mother slapped Simon.

An inference system could draw many inferences through this story, e.g., 1. Simon's mother slapped Simon because she was angry at him for hitting Anne, 2. Anne was taken to the hospital because she was hurt, 3. Anne's mother called Simon's mother because she wanted to complain later. Inferences are based on a set of rules, organized around inference categories. The total number of categories in this case is 16, some of these are:

1. *Causative inferences*: These are hypothesized as possible causes or preconditions of actions.
2. *Specification inferences*: These fill in missing 'slots' in a CD primitive, such as the ACTOR or INSTRUMENT of an ACTion.
3. *Resultantive inferences*: These are inferred as likely results of the actions.
4. *Function inferences*: These are inferred as likely functions of objects.

The inference rules are applied in an undirected fashion. When a system reads a sentence, the inference rules for this are automatically applied without a goal, such as building a causal chain of events. So, it is fortuitous (having no cause or apparent cause): the inferences are applied to new representation in an undirected fashion, which some times result in the confirmation of another representation.

The undirected inferences are analogous to the spontaneous nature of inferences made by people, which some times seem uncontrollable for people. This is convincing as one cannot learn new facts without inferring things about that. However, this undirected behavior leads to problems: When processing a story, an expert system, which is based on CD, would not know which inferences are most likely to lead to building a coherent causal chain to represent the story. Hence, it would lead to a combinatorial explosion in the number of inferences that the system needs to consider to build the causal chain to represent the story. In other words, the expert system lacks the commonsense knowledge about what inferences are most likely to be relevant in a given situation. For another example,

> Simon picked up the menu. He decided on fish.

As an example, consider a situation, where one visits a restaurant, and decides to take a seat for eating (see Fig. 7.18).
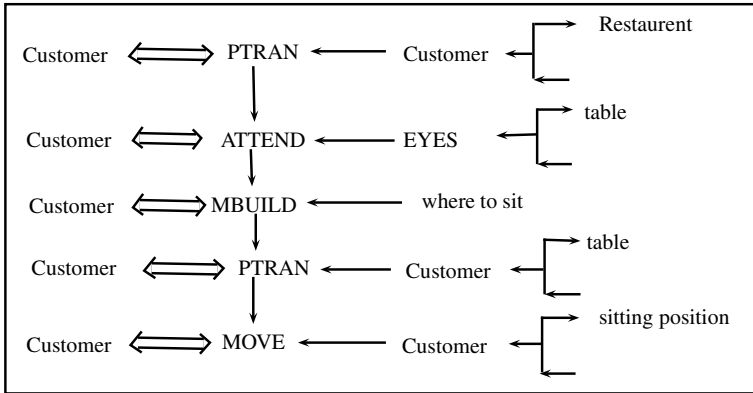
**Fig. 7.18** CD for "Customer moves to Restaurant"

Once a CD with a sequence of the graph is represented, it is possible to infer many things from the representation. Like, from the Fig. 7.18, it is possible to infer many things, like: "Who went to a restaurant?", "Why the customer searched some thing in a restaurant?", etc.

### 7.6.3 Scripts

The scripts represent stereotypical sequences of events, like going to a restaurant, buying from a store, etc. The theory of scripts has an emphasis to understand and quick access those events that always happen in a stereotypical event sequence, without worrying about those inferences which would most likely be irrelevant.

Scripts are larger knowledge structures, used to solve problems using undirected inference. The scripts are pre-compiled sets of likely inferences, with elements of each set packed together so that they can be searched more efficiently, and produces lesser number of irrelevant inferences.

A script is a set of *roles* (participants) involved in the script, as well as common objects used. A script comprises *scenes*, such that each scene describes typical events in a portion of a script. For example, in *restaurant*'s script, roles may be *waiter*, *customer*, *restaurant* itself, and *food*. Scene may be, ENTER, ORDER, EAT, PAY, LEAVE. Each scene's details are represented as a sequence of CD representations. For example, the ENTER scene in a restaurant may consist of a causal chain as shown in Fig. 7.18, and the sequence of CDs are:

**Scene-name: ENTER**
*C* PTRAN *C* into restaurant
*C* ATTEND eyes to tables
*C* MBUlLD where to sit

*C* PTRANS *C* to table
*C* MOVE *C* to siting position

In above, *C* is a customer.


### 7.6.4   *Conceptual Dependency Versus Semantic Nets*

The conceptual dependency networks and its derivatives are often grouped into the family of semantic net representations, because the CD is a *content theory*, whereas the semantic networks are a *structure theory*. The distinction between these two types of theories lies in their emphasis: the semantic nets theory is about how the knowledge should be organized—there are nodes with arcs connecting them. There is also some general notion about the structure's semantics, i.e., how to interpret a particular semantic network. In any representation, apart from the knowledge organization, there is also a general notion about inheritance. Both of these points are about structural information. The semantic networks say nothing about what will be represented, e.g., what labels should be used for nodes and arcs, and what arcs to use where? It is only up to the user to decide about these details, and what a semantic network says is about the (structural) form that the representation will take.

The CD theory was an attempt to enumerate the types of nodes and arcs which could be used to build representations. Instead of specifying the structure of representations, the CD theory specifies the contents. Note that the basic conventions used for drawing CD graphs also specified structures, but these graphs were not really the essence of the theory. The essence of the CD was in the *primitives*, and in the types (names) of dependencies which could be used to link the primitives together.

The above highlighted distinctions between CDs and semantic nets would become clear if one thinks of trying to implement the CDs and semantic nets in first-order predicate logic (FOPL). In CDs, it is not difficult to imagine that the primitives: ACTs, dependencies, and states would specify a set of predicates to be used when one writes the predicate calculus statements to represent sentences. There will be a need to adopt some translation conventions, in order to make all assertions of FOPL type, but these would be quite straightforward.

But, it does not make a sense to implement a semantic net in FOPL, as the two representations are in competition with each other: each representation provides a different syntax for distinguishing between predicates, arguments, and relations. There would be nothing left to semantic nets, if they are translated into predicate logic. Putting it another way, the semantic nets would not add anything to FOPL. This statement is in contrast to CD, which adds the CD primitives to the predicates used, as and when needed.

## 7.7   Summary

In a network-based representation, the pieces of knowledge are clustered together into coherent semantic groups. It provides a more natural way of mapping knowledge between the natural language and these networks. A semantic network has *ako* (a kind of), has-parts, color, and has-property are binary relations. Semantic networks have primitives, and inference in the semantic networks is provided through inheritance. The semantic networks are based on the associationist theory, which defines the meaning of an object in terms of a network of associations with other objects.

A natural language understanding program must understand the *intentions*, *beliefs*, *hypothetical reasoning*, *plans*, and *goals*. Conceptual Graphs (CG) portray mental perception which consists of basic primitive concepts and relationships which exist between them. In CG, a sentence is represented as a verb node, with various case links to the node representing other participants in the action.

A *frame* can be viewed as generalized *semantic network*. In a frame, there is stress on instances or classes, rather than nodes, and on slots and their values instead of links and connections. Inheritance moves the default values in frames from classes to instances through activation of the appropriate when-constructed procedure. Frames represent some *stereotypical situation*, like, a classroom, house, a machine with various parts, etc, and have slots, which may hold values, or procedures. Procedures in frames may be sleeping procedures (daemons) and may be awakened when required. Frames have the language to describe them. The Frames have applications in machine vision.

A Description Logic models *concepts*, *roles* and *individuals*, and their relationships. The fundamental modeling concept of a $\mathscr{DL}$ is the axiom: "a logical statement relating roles and/or concepts". It is a family of formal knowledge representation languages, which is more expressive than propositional logic and has more efficient decision properties than first-order predicate logic. It is used in formal reasoning on the concepts of an application domain.

The Conceptual Dependency (CD) framework is a stratified linguistic system that attempts to provide a computational theory of simulative performance. Every conceptualization must have a two-way dependency link. The CDs are intended to be used in reasoning with natural language constructs, independent of any language or the phrases in the language. The CDs have a set of primitive actions, a set of states for representation and result of the action, and a set of dependencies, or conceptual relationships, which could exist between primitives, and states. An English language sentence can be parsed into CDs. The inference using CDs is unambiguous.

*Scripts* represent stereotypical sequences of events, such as going to a restaurant. By using the script, the theory was that the understand had quick access to those events which always happen in a stereotypical event sequence.

## Exercises

1. Explain the difference between Ontologies and Semantic networks.
2. Describe the logical, structural, semantic, and procedural parts of semantic networks.
3. There are many words in the English language which can be used as noun and verb, for example, "book" in "Book my ticket" and "This is my book" have used the word "book" as verb and noun, respectively. In the following words, what are their different parts of speech?
   milk, house, liquid, airborne, group, set.
   Suggest a method in each case, as to how you will reason the true meaning.
4. Suggest a data structure for the implementation of semantic nets such that retrieval can be as fast as possible.
5. Represent the relationships between quadrangle, parallelogram, rhombus, rectangle, and square in the form of a semantic network. Is the semantic network unique, or are there many different forms it can take?
6. Represent the following statements using semantic networks:

   a. "Rajan teaches his students a lot of innovative things."
   b. "Raman tells Rajan's students a number of useful things."
   c. Mike and Mary's telephone number is the same.
   d. John believes that Mike and Mary's telephone number is the same.

7. Represent the following knowledge in a semantic network:

   | | |
   |---|---|
   | Dogs are Mammals | Birds have Wings |
   | Mammals are Animals | Bats have Wings |
   | Birds are Animals | Bats are Mammals |
   | Fish are Animals | Dogs chase Cats |
   | Worms are Animals | Cats eat Fish |
   | Cats are Mammals | Birds eat Worms |
   | Cats have Fur | Fish eat Worms |

8. Represent the following in partitioned semantic networks:

   a. Every player kicked a ball.
   b. All players like the referee.
   c. Andrew believes that there is a fish with lungs.

9. Represent the following statements using semantic networks:

   a. "John tells his students a lot of useful things."
   b. "Andrea tells John's students an enormous number of useful things."

   Suppose you wanted to build an AI system that was able to work out "who tells John's students the greatest number of useful things." How could you do that?
10. Suppose you learn that "Tom is a cat". What additional knowledge about Tom can be derived from your representation? Explain how.

11. Suppose Tom is unlike most cats and does not eat fish. How could one deal with this in the semantic network?

12. Formulate the solutions as to how the semantic networks can be used in the following cases?

    a. Natural language understanding
    b. Information retrieval
    c. Natural language translation
    d. Learning systems
    e. Computer vision
    f. Speech generation system

13. "The inferencing in semantic networks make use of unification, chaining, modus ponens, and resolution." Justify each, taking a suitable example.

14. Explain, using semantic networks, how we can map an object's perception to concepts, and identify these concepts. Give examples.

15. How semantic networks help in understanding the meaning of words in natural language sentences? Explain.

16. Represent the following as a series of frames:

    Dogs are Mammals       Birds have Wings
    Mammals are Animals Bats have Wings
    Birds are Animals       Bats are Mammals
    Fish are Animals        Dogs chase Cats
    Worms are Animals       Cats eat Fish
    Cats are Mammals        Birds eat Worms
    Cats have Fur           Fish eat Worms

17. Express the following sentences in Description Logic:

    a. All employees are humans.
    b. A mother is a female who has a child.
    c. A parent is a mother or a father.
    d. A grandmother is a mother who has a child who is a parent.
    e. Only humans have children that are humans.

18. Translate the logic expressed in Fig. 7.10 into $\mathscr{DL}$.

19. Select one or more answers from the following. Also, justify the answer(s) selected by you.

    a. What type of reasoning is performed using semantic networks?

    (A) Deductive    (B) Default
    (C) Inductive    (D) Abductive
    (E) Hierarchical

    b. In the Description Logic, the domain is always,

    (A) Open world                (B) Closed world
    (C) Depends on the domain used (D) None of above

# References

1. Collins AM, Quillian MR (1969) Retrieval time from semantic memory. J Verbal Learn Verbal Behav 8(2):240–247
2. Deliyanni A, Kowalski RA (1979) Logic and semantic networks. Commun ACM 22(3):184–192
3. Faucher C (2001) Easy definition of new facets in the frame-based language Objlog+. Data Knowl Eng 38:223–263
4. Harmelen FV et al (2008) Handbook of knowledge representation. Elsevier, pp 213–237
5. https://www.inf.unibz.it/~franconi/dl/course/dlhb/dlhb-02.pdf. Accessed 19 Dec 2017
6. http://www.jfsowa.com/pubs/semnet.htm. Accessed 12 Feb 2018
7. Lytinen SL (1992) Conceptual dependency and its descendants. Comput Math Appl 23(2–5):51–73 Pergamon Press
8. Minsky M (1974) A framework for representing knowledge. MIT-AI Laboratory Memo-306
9. Pike R, Kehler T (1968) The role of frame-based representation in reasoning. Commun ACM 28(9):904–920
10. Quillian MR (1967) Word concepts: a theory and simulation of some basic semantic capabilities. Behav Sci 12(5):410–443
11. Quillian MR (1968) Semantic information processing. Cambridge, Mass., MIT Press, pp 216–270
12. Simmons RF (1973) Semantic networks: their computation and use for understanding English sentences. In: Schank RC, Colby KM (eds) Computer models of thought and language. W.H. Freeman and Co, San Francisco, CA
13. Simmons RF, Chester D (1977) Inferences in quantified semantic networks. Proceedings of the fifth international joint conference on artificial intelligence. MIT, pp 267–273
14. Sowa J (1976) Conceptual graphs for a data base interface. IBM J Res Develop 336–355

# Chapter 8
# State Space Search

**Abstract** State space search is one of the three fundamental requirements to achieve AI. This chapter present the basic techniques, called uninformed search, of searching the goal in problem solution. A search requires the representation of state space in the forms of a directed graph. The basic methods—depth-first search (DFS), breadth-first search (BFS), along with their algorithms, analysis of these algorithms, and along with worked out exercises are presented. The improved techniques—iterative deepening DFS, and bidirectional search, followed with chapter summary, and then large number of practice exercises are provide at the chapter end

**Keywords** State-space · State-space search · Depth-first search · Breadth-first search · Complexities of search · Bidirectional search · Iterative deepening DFS · Search algorithms

## 8.1   Introduction

Next to knowledge representation, search is another important requirement for Artificial Intelligence. This chapter relates the problem solving to search of solution, where search process is a tree search, consisting of generating of new states, and ultimately leading to a goal state. The presented search is called *blind-search* or *exhaustive search*, and turns out to be an exponential search ultimately. The various search methods' analysis and complexities have been derived, and presented.

Search is one of the operational task that characterize AI programs best. Almost every program depends on search procedure to be performed to carry out its prescribed functions. Problems are generally specified in terms of states, and solution corresponds to the goal states. Solving a problem then amounts to searching through the different states, called *state-space*, until one of the goal state is reached. If goal state is not found, it concludes that solution does not exist for the problem or the goal state is not reachable.

The state space consists set of vertices $V$ and set of connections between them in the form of edges (links) from $E$. Thus, the search space is a graph $G = (V, E)$. The state space to be searched here is different than it is found in the information

searching in the conventional trees and graphs, in the sense that there, the space limits are fixed, i.e., the number of vertices, and edges connecting them are already known. However, in AI search problems, the alternate moves are generated from each vertex, like in a chess game, the search tree is to be generated and simultaneously be searched. Hence, you cannot determine the size of tree to be searched in advance.

**Learning Outcomes of this Chapter**:

1. Formulate an efficient problem space for a problem expressed in natural language (e.g., English) in terms of initial and goal states, and operators. [Usage]
2. Describe the problem of combinatorial explosion of search space, search time, and its consequences. [Familiarity]
3. Select and implement an appropriate uninformed search algorithm for a problem, and characterize its time and space complexities. [Usage]

## 8.2  Representation of Search

The search space is generally represented as a directed graph (in fact a tree), with vertices as states, and edges of the graph as transitions for moves to explore for the goal state. The initial configuration of the problem description is *start* state, at which we apply the specified rules to generate new states (vertices). Thus, it becomes similar to a tree with start configuration as *root* node, then there are interior nodes, and finally child nodes at the lowest level having no further children. To understand the state space search better, we consider the following example of puzzle game [5].

**Example 8.1**  8-Puzzle Game.

The 8-puzzle game is shown in Fig. 8.1, with *initial configuration*, *goal configuration*, and transitions (i.e., moves) possible from each state. We refer the configuration of the game equal to the collective status of tiles on the board. This configuration we call as *state*.

Using the allowed moves of a numbered tile (1–8), to tile left, right, up, down, to the destination blank-tile, we generate new states starting from the start state ($S$) as $A, B, C, \ldots,$. As a state is generated, it checked for the goal state. If yes, we terminate the process and declare the fact that goal is reached. If not, the generated states are further expanded by application of *generate rules* (moves), until the goal is reached. If we carry on the process of expanding all the states generated, the goal state shown in the figure with reverse order of tiles from first to last row, will ultimately be reached, of course after a large number of transitions have been carried out.

The transitions' diagram like this is called graph, with each configuration as a node, each move as an edge, and finding the solution, i.e., goal is nothing but searching the graph through traversing, for the goal state.                                                               □

There are many ways in what order we must generate the new child nodes. For example, generate children at a level $k + 1$ only after all the nodes at level $k$ have
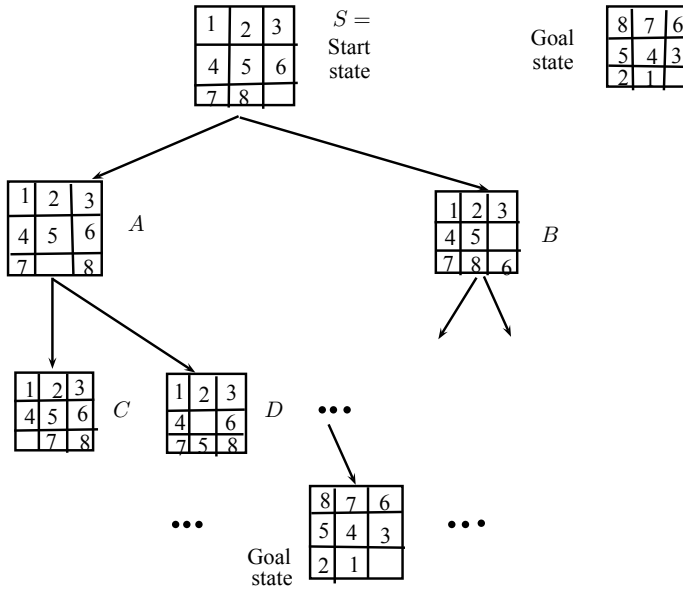
**Fig. 8.1** The 8-puzzle game

been expanded. This is called BFS (breadth first search). Alternatively, to expand a node to its full depth without touching its neighbors (siblings), this BFS (breadth first search), or there can be variants of DFS or BFS.

The Search Strategies are evaluated along the following dimensions:

- *Completeness*: Does it always find a solution if one exists?
- *Optimality*: Does it always find a least-cost solution, i.e., in terms of minimum number of transitions?
- *Time complexity*: What is to be number of nodes generated? The duplicate nodes generated due to multiple paths, are also counted.
- *Space complexity*: What is maximum number of nodes in memory at any time?

## 8.3 Graph Search Basics

Let there is a *directed* graph $G = (V, E)$, where $V$ is set of vertices, and $E$ is set of ordered pairs $(u, v)$, called edges, such that $u$ is head and $v$ is tail of an edge. In an *undirected* graph, the edges are unordered pairs of vertices, and there is no concept of fixed head and tails of edges. Graphs form abstractions for many problems, e.g., circuits in electrical engineering, molecular structures in chemistry, relations in social structures, and in modeling for example, in online sales. Due to their enormous applications, it is important that there should be efficient algorithms for answering graph-theoretical questions.

We should be aware of following terminologies, to when thinking of graph algorithms. For the graph $G = (V, E)$, a path $p = u \xrightarrow{*} v$ is a sequence of connected edges and vertices from vertex $u$ to $v$. A path is *simple* if all the vertices in that path are distinct, otherwise the path has one or more cycles. A path $p = u \xrightarrow{*} v$ is called a *closed path*, and a closed path $p = u \xrightarrow{*} u$ is a *cycle* if all its edges are distinct and the only vertex that occurs twice in $p$ is $u$. Two cycles which are cyclic permutations of each other are considered to be the same cycle.

An *undirected* version of a directed graph is the graph formed by converting each edge of the directed graph into an undirected edge and removing the duplicate edges, if any. An undirected graph is connected if there is a path between every pair of vertices.

Trees

A rooted tree $T$ is a directed graph, whose undirected version is a connected graph, with one special vertex (the root), which is head of no edges, but is tail of one or more edges. All the vertices other than the root are head of exactly one edge. An edge "$(u, v) \in T$" is a relation in tree $T$, and denoted by $u \rightarrow v$. A relation of, "there is a path from $u$ to $v$, in $T$" is denoted by $u \xrightarrow{*} v$. If $u \rightarrow v$ holds, then $u$ is called "parent" of $v$, and $v$ is called *son* of $u$. If $u \xrightarrow{*} v$ then $u$ is ancestor $v$, and $v$ is called descendant of $u$. Every vertex is also ancestor and descendant of itself.

If $u$ is a vertex in a tree $T$, then we call $T_u$ as subtree of $T$ comprising all vertices of all the descendants of $u$ in $T$.

For a directed graph $G$, the tree $T$ is spanning tree of $G$ if $T$ is subgraph of $G$ and $T$ contains all the vertices of $G$.

## 8.4   Complexities of State-Space Search

From the previous section, it is clear that the process of searching can be automated by applying the rules of moves to cause transitions from one state to next, until you reach to the goal state. However, in the process a state should not repeat in the path, for example, like $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_g$. Such repeated vertices form a loop and we never reach to goal state. Since the previously visited vertex $v_1$ is repeated, the path may instead be taken as $v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_g$.

Let us try to find out number of states to be visited in the worst case, to reach to goal state, in the search required in Fig. 8.1. Thus, all unique states generated (like shown in figure as $A, B, C, \ldots, goal$), in the path to $goal$, in the worst it is $9! = 362880$.

If we consider the case of one of the popular board game like *Chess*, for a particular strategy of opponent, we definitely have some winning strategy, represented by a configuration. Our aim is to reach to that configuration from the starting configuration, in step-by-step way, where each step is a move by us, followed by move by opponent. Each move transforms the chessboard from one configuration to

another configuration. Suppose we assign the job to a computer to play against us as opponent. So let us try to compute the number of total moves required in the worst case to move from start state to winning state, of course, going through all the states generated so far. Assuming on the average 20 alternate moves for a configuration, and assuming there are 100 different configurations, the total states which can be generated and searched are $20^{100}$. This figure is greater than $10^{130}$, a combinatorial explosion of states. This value is even greater than the number of pico-seconds passed since Big-bang occurred and also greater than the number of molecules in the known universe. Winning a game on computer, using an *exhaustive search method* like this, amounts to going through these sequence of states, which ultimately should lead to goal state. From above, we note that the search time is *exponential in time*.

**Example 8.2** Consider the graph shown in Fig. 8.2 for a small size problem, where it is required to reach to goal state $E$ from the start state $A$.

For the graph shown in Fig. 8.2a, when it is searched from the start state $A$, the states generated are shown in the tree in Fig. 8.2b. Since objective of search is to reach to the goal, a search process terminates the moment the goal state is reached through any path. The tree in Fig. 8.2b shows the total possibles states generated in the worst case, of course many repeated, but in a path no vertex is repeated. □

Thus, We can conclude that problem solution through search process is a tree search (even if it is a graph originally) with already specified start node and goal node. Tree search can end-up repeatedly visiting the same nodes, unless it keeps track of all nodes visited. But, this could take vast amounts of memory, so large that most computers do not have, even for small size graph of say 50 nodes.

Complexity of Travelling Salesman Problem

The travelling salesman problem's (TSP) solution requires to find a best path covering all the nodes in a graph with given start node and the destination node. Consider that number of nodes are $n$. Assuming that there is a path from every node to every other node. Thus, all the possible paths are the all permutations of these $n$ nodes. Having given the first node, traversing all the nodes and visiting back to the first, can be done in $n!$ number of ways. This gives worst case complexity of $O(n!)$. For example, for



(a) Graph with vertices $A$-$E$.    (b) Graph search is a tree.
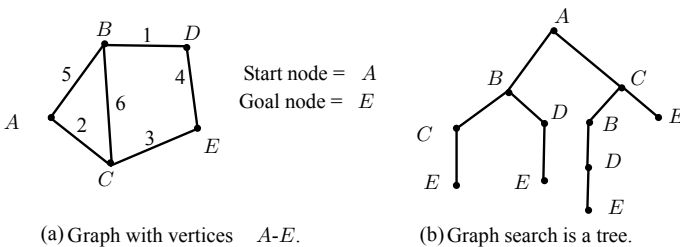
**Fig. 8.2** Undirected graph

the graph shown in Fig. 8.2b, the possible paths set for start $A$ and destination node $E$, is $\{ABCE, ABDE, ACBDE, ACE\}$.

The possible applications of *TSP* are distribution networks, *VLSI* design, and other optimization problems.

## 8.5   Uninformed Search

It is also called *blind search*, because we do not know, which moves ultimately will lead to the goal faster. Since all the nodes are required to be searched—the search is called *exhaustive* search. To search the entire state space, the two important approaches are—*breadth-first search* (BFS) and *depth-first search* (DFS). What others approaches exist are variants of DFS and BFS. For example, *depth-limited* search, and *iterative deepening DFS* are variants of DFS, while *bidirectional search* may be based on DFS as well BFS.

### 8.5.1   Breadth-First Search

To carry out the BFS for a graph or to solve a puzzle, like, shown in Fig. 8.1, first, root node is checked if it is goal, if yes then terminate the process after declaring that goal is found. If goal is not found, the children nodes are generated for root node and tested if any one of them is goal. Then further children are generated for each of them, and so on, until goal is reached or the search is terminated. In the latter, no new state can be generated in any of the search path. A BFS phenomena is indicated in Fig. 8.3 where dotted trace is showing order in which nodes are tested for the goal [2].

The *BFS* can be implemented using a *queue* type data structure, named here as **List**. The front node of the of the queue is represented by **List**.*Head*. The algorithm for this is shown as, Algorithm 8.1, which checks all the paths at a given length, before testing the paths of longer length. The inputs to this algorithm are: the graph **G**, start node **S**, and goal node **Goal**. If goal is reached it returns the *success*. After searching all the vertices, which will be indicated by a empty **List**, if goal is still not found, the algorithm returns fail and exits.

If the BFS Algorithm 8.1 is applied for generate-and-search, a tree like one shown in Fig. 8.3 gets constructed, and this entire tree needs to be searched in the worst case to find the goal state. The BFS order of this tree is $S, A, B, C, D, E, F, G, H$.

Though, we certainly locate the goal if it exists, the path to goal from start state is not remembered by the search Algorithm 8.1. This, however can be carried out in the following way. In the *queue* data structure used for implementing BFS, each entry is stored like $(x, y)$, where $x$ is next child node to be explored for goal, and $y$ is its parent node. The value $y = 0$ represent the root node. We call this queue as *open-list*, i.e, the nodes which have not been fully explored yet. When a node is deleted from front

---

**Algorithm 8.1** BFS(Input: **G**, **S**, **Goal**)

---
1: **List** = [S]
2: **repeat**
3:    **if List**.*Head* = **Goal then**
4:       return *success*
5:    **end if**
6:    generate children set **C** of **List**.*Head*
7:    **append C** to **List**
8:    delete **List**.*Head*
9: **until List** = []
10: return *fail*

---

**Table 8.1** Trace of state space search (BFS) for Fig. 8.3

| Open-list | Closed-list |
|---|---|
| $[(S, 0)]$ | [] |
| $[(A, S), (B, S), (C, S)]$ | $[(S, 0)]$ |
| $[(B, S), (C, S), (D, A), (E, A)]$ | $[(S, 0), (A, S)]$ |
| $[(C, S), (D, A), (E, A), (F, B), (G, B)]$ | $[(S, 0), (A, S), (B, S)]$ |
| $[(D, A), (E, A), (F, B), (G, B), (G, C), (H, C)]$ | $[(S, 0), (A, S), (B, S), (C, S)]$ |
| $[(E, A), (F, B), (G, B), (G, C), (H, C)]$ | $[(S, 0), (A, S), (B, S), (C, S), (D, A)]$ |
| $[(F, B), (G, B), (G, C), (H, C)]$ | $[(S, 0), (A, S), (B, S), (C, S), (D, A), (E, A)]$ |
| $[(G, B), (G, C), (H, C)]$ | $[(S, 0), (A, S), (B, S), (C, S), (D, A), (E, A), (F, B)]$ |

**Fig. 8.3** Breadth-first search
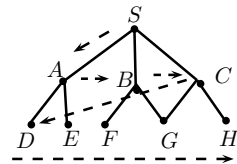


of list, we add this deleted node into a separate list, called *closed-list*. The entries in Table 8.1 show the search operations to search the goal node $G$. Each row in open-list is constructed by a deletion or append operation on the previous content of the queue data structure [4].

The search process is terminated when goal node $G$ is encountered as the next node in open-list. After reaching the goal node we can backtrack to find out the path from goal node to start as: "$(G, B), (B, S)$". Another path is "$(G, C), (C, S)$". We consider the first, or if path lengths are given then the one which is shorter.

Since BFS always explores the shallow nodes before the deeper ones, BFS finds the shallowest path leading to the goal state. A modified and improved search can be obtained by expanding only the lowest cost node ($n$). This is called *Uniform cost search*. This modifies BFS such that the cost of a path should remain low but not

decreasing hence the term *uniform*. Since, a BFS algorithm is bound to find a goal, if at all the goal exists, the BFS is *complete* inference system, as well as optimal (finds the shortest path, as the nearer nodes are searched before the farther).

### 8.5.2   Depth-First Search

The depth-first search (also called backtracking search), is a technique, which is widely used for finding solutions to problems in Artificial Intelligence, and Combinatorial Theory. In the following discussions we are trying to analyse it properties. Consider that $G$ is a graph we wish to explore. At start, all of its vertices are unexplored. To explore it, we start from some vertex of $G$ and choose an edge to traverse, which leads to new vertex. Continuing in this way, at each step we select an edge that leads from the vertex already visited, and reaches to a vertex that is either already visited or new. When there is no further edges available in sequence to explore, we choose an edge yet to be explored from most recently visited vertex, and continue the process. If there is no most recent edge available, we may try the parent node of this vertex, and so on. Whenever we run out of edges leading from previous visible vertices, we choose some unexplored vertex, if one exists, begin new exploration from this point. Eventually, we will traverse all the edges of $G$, each exactly once [6].

Consider the following choice rule: when selecting an edge to traverse, always choose an edge emanating from the vertex most recently reached which still has unexplored edges. A search which uses this rule is called a *depth-first search* (DFS). The set of old vertices with possibly unexplored edges may be stored on a stack type data structure. Thus a depth-first search is very easy to program either *iteratively* or *recursively*, provided we have a suitable computer representation of a graph.

To perform a DFS search, we generate all the next states for the root node, then pickup the left-most node, generate the children for this, check for goal, and repeat this, until we reach to goal or the dead end.

When reached to the dead end (the final child node), from then, back track, to the siblings of this node, apply the DFS, then reach to siblings of its parent, and so on. Applying the DFS Algorithm 8.2, generates a tree like one shown in Fig. 8.4. The order of nodes visited in DFS order are: $S, A, D, E, B, F, G, C, H$. For DFS also, a table of trace can be constructed as it was done for BFS, to find the path from root node to goal node.

**Fig. 8.4**  Depth-first search

---
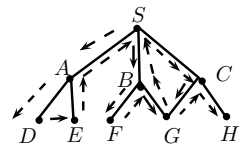
**Algorithm 8.2** DFS(Input: **G**, **S**, **Goal**)

---

1: **List** = [S]
2: **repeat**
3:    **if List**.*Head* = **Goal then**
4:        return *success*
5:    **end if**
6:    generate children set **C** of **List**.*Head*
7:    delete **List**.*Head*
8:    **insert C** at begin of **List**
9: **until List** = []
10: return *fail*

---

### 8.5.3 Analysis of BFS and DFS

Consider the Figs. 8.3, and 8.4. Assume that the goal node is *E*. Using the *DFS* method it needs only four steps to reach to node *E*. Where as it requires six steps to reach to *E* if *BFS* search method is used. Thus, for a deeper goal node, *DFS* is considered better. If the node to be searched was *C*, the *BFS* required four steps, and *DFS*, which first search deeper information, then shallow, needs total eight comparisons. Thus, which approach is best, depends on the position of goal node. Also, if branching factor *b* (number of branches per node) is large, the *DFS* is better suited, and *BFS* is worst. Thus, the efficiency of search depends on the structure of tree, the search method used, and branching factor [2].

Let us assume that in a tree constructed in a search has depth *d*, and for each node there are *b* nodes that gets generated, called *branching factor* of the tree. For a *BFS* tree, the worst-case time spent for any search is the maximum number of nodes visited to determine the goal. This is the worst-case *time-complexity* of the search. At a time how many nodes are in the *open-list*, determine the space, or *space-complexity* of the algorithm.

For a *BFS* search, total nodes visited for tree of depth *d* are:

$$1 + b + b^2 + \cdots + b^d = O(b^d), \tag{8.1}$$

which is worst-case *time complexity*, and the maximum number of nodes in the *Open-list* will exist at the lowest level of the tree. Thus, space-complexity is $O(b^d)$. Hence, in the case of *BFS*, both time and space complexity are $O(b^d)$.

For a *DFS* search also, in the worst-case, all the nodes are required to be visited. Hence, *time-complexity* is same as for *BFS*, and equal to $O(b^d)$. Since, *DFS* needs to store only *b* nodes per level for a depth of *d*, the total nodes to be stored are in memory are $b \times d$, hence *space-complexity* is $O(bd)$. Usually the branching factor *b* is much smaller than *d*, results to the space complexity as $O(d)$.

In the *BFS*, since all the nodes at a given depth are stored in order to generate the nodes at the next depth, the maximum number of nodes that must be stored to search to depth *d* is $b^d$, hence the *space complexity* is $O(b^d)$. As with time, the

**Fig. 8.5** Graph with nodes
in linear oder, but paths in
exponential order
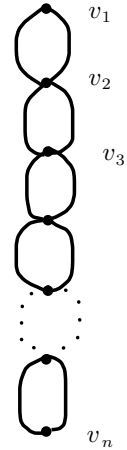


average-case space complexity is roughly one-half of this, which is also $O(b^d)$. This
space requirement of breadth-first search is its most critical drawback. As a practical
matter, a breadth-first search of most problems, spaces will exhaust the available
memory long before an appreciable amount of time is used.

The Depth-first search avoids the memory limitation of *BFS*. It works by always
generating a descendant of the most recently expanded node, until some depth cutoff
is reached, and then backtracking to the next most recently expanded node and
generating one of its descendants. Therefore, only the path of nodes from the initial
node to the current node must be stored in order to execute the algorithm. If the depth
cutoff is $d$, the space required by *DFS* is only $O(d)$.

Since the depth-first needs to store the current path at any given depth, it is bound
to search all the paths down to any specified cut-off depth. A new parameter, called
*edge branching factor* ($e$), is defined to analyse its time complexity.

$e$ = average number of different operators applicable to a given state (i.e., node).

In case of trees, the edge branching factor and node branching factors are equal.
This is because, the number of branches of a tail node $v$ of an edge $(u, v)$ are same
as that from the edge $(u, v)$. But, for a graphs, the edge branching factor may exceed
the node branching factor. For example, for the graph shown in Fig. 8.5 has an edge
branching factor of two, while its node branching factor is one only. Accordingly,
the worst case number of paths from node $v_1$ to reach to the node $v_n$ is $2^{n-1}$, i.e., in
the worst case in which you can traverse the tree of root $v_1$ is $O(2^{n-1})$, which is equal
to $O(2^n)$, an exponential time complexity!

We know that *BFS* traverses a graph by visiting the nearer nodes before the farther
nodes, hence when Fig. 8.5 is traversed in *BFS* it will be in linear order of all the nodes
in the graph. However, the *DFS*, which goes deeper and deeper by sequencing the
edges, hence in general may visit many nearer nodes far later than the farther nodes.
But, considering the Fig. 8.5 for *DFS*, since it joins the edges up to the maximum
depth, the worst-case paths will be $2^{n-1}$, with time complexity of $O(2^n)$, which is

exponential. In general, the time complexity of a depth-first search for depth $d$ and edge branching factor $e$ is $O(e^d)$. Note that, the space used by depth-first search grows only as the log of the time required, the algorithm is actually *time-bound* rather than space-bound. This is because, the time is $e^d$, while space is $\log e^d = d$.

Consequently, for a system in which state repetition is possible (see Fig. 8.5), every generated state must be stored in a table, and every new state generated must be looked upon if it is generated again. In principle, this lookup table can be expensive, but an efficient approach can be used to do it in a time logarithm of the total number of states or better. A simple way of doing this is *discrimination tree*, where previously seen states are stored in the leaves of the tree, whose non-terminal nodes are labeled with discriminations.

Another drawback of *DFS* is that it requires an arbitrary cutoff depth. During the search, if branches are not cutoff, and duplicates are not checked, the algorithm may run for ever. Generally, the depth at which the first goal may appear, is not known in advance, hence if cutoff is set too low, the algorithm may not find the goal, in spite of its existence, and may terminate. If the goal estimate is taken too deep, the algorithm may spend too much of a time, before it reaches to the goal or terminating without finding the goal.

It is to be noted that, *DFS* is not good at finding the goal, particularly if the goal is on the opposite side of the tree which is being searched, even though the goal is shallow. So, many a times, a *depth-limited DFS search* is performed, i.e., searching the space for predetermined depth only. A method which combines the advantages of both the *BFS* and *DFS* is *iterative deepening DFS*, discussed in the following part.

### *8.5.4  Depth-First Iterative Deepening Search*

The depth-first iterative deepening *DFS* (DFID) reaches to shallow goals much faster than the ordinary *DFS*. It first sets the tree depth $d = 0$, performs DFS, hence checks the root node for goal. Then discards all the nodes generated in the previous search, starts over and do a depth-first search for $d = 1$. Next, starts over and do a DFS for $d = 2$, and so on. In ordinary DFS, the goal node $C$ (Fig. 8.6) will get searched in 8 comparisons, where as using iterating deepening *DFS*, search is carried out for tree depth $d = 0$, next in $d = 1$ the node $C$ gets located, requiring total $1 + 4 = 5$ comparisons [3].

The standard algorithms of BFS and DFS, both have serious limitations, which are overcome by the algorithm DFID. The iterative-deepening algorithm, however, is a general algorithm, and can be applied to *uni-directional search*, *bi-directional search*, and *heuristic searches* like $A^*$, which we will discuss in Chap. 9.

Since, DFID expands all the nodes at given depth, it is guaranteed to find a solution in shortest-time. The disadvantage of DFID is that it performs wasted computation prior to reaching to goal depth. In fact, at first glance it seems very inefficient. However, this wasted computation does not effect the asymptotic growth of the run

**Fig. 8.6**  Search tree of
iterative deepening DFS
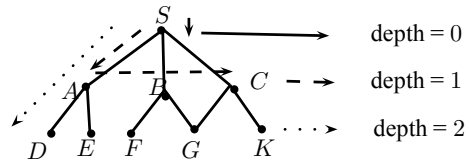


time for exponential search. The intuitive reason is that almost all the work is done
at the deepest level of the search.

The worst-case complexities for the this method remains the same as ordinary
*DFS*, however, the average case improves. This is because, in the worst case you
still need all nodes to be compared, taking time equal to $O(b^d)$, and worst case space
requirements is $O(d)$ only.

### 8.5.5   Bidirectional Search

If the search is carried out in both the directions, it can be sped-up. The bidirectional
search sacrifices the space for time, i.e., gains in time complexity at the cost of space
complexity. This technique search from the initial state in forward direction, and
from the goal state in backward direction, and both searches are carried out together.
The states generated are stored until the common state is found to both the searches.

Consider that a bidirectional search is carried out with the depth as $d$ and branching
factor $b$. If each side progresses with same depth, the search required from each, for
example, for iterative deepening *DFS*, has $O(b^{\frac{d}{2}})$ for time, and $O(b * \frac{d}{2})$ for for
space. When combined from both opposite sides, it becomes $2 * O(b^{\frac{d}{2}}) = O(b^{\frac{d}{2}})$
for *time*, and $2 * O(b \times \frac{d}{2}) = O(bd)$, which is $O(d)$, for *space*.

The depth first iterative deepening (*DFID*) search can be applied to bidirectional
search as follows: A single iteration consists of a DFS from one direction to a depth $k$,
and stores the states at depth $k$. The second DFS searches from the other direction, one
to depth $k$ and other to $k + 1$, which does not stores the states but simply matches
against the states stored from other direction. Note that the search to depth $k + 1$
is necessary to find odd-length solutions for the goal. The process is repeated from
$k = 0$ (solution at depth zero), to $k = d/2$. If it is assumed that some *hashing* scheme
is used to perform the matching in constant time for every node, the bidirectional
algorithm will find the optimal solution of total depth $d$ of the graph, in time $O(b^{d/2})$
and space $O(b^{d/2})$.

However, for bidirectional search to be implemented, it is necessary that *invertible
functions* must be available for generating nodes. Which, in fact, does not exists in
every problem. For example, to search a descendant in a tree of many generations,
one can generate children nodes for each parent, and if a path leads from a forefather
node $x$ to a descendant node $y$, we can say that $y$ is descendant of $x$. Similarly, we can
travel in the tree backward, from a descendant $y$ to its parents, and then their parents,

until, if it reaches to *x* through any path, we can declare that *x* is forefather of *y*, or *y* is descendant of *x*. Hence, there exists an invertible function. Similar phenomena exists in the case of board game, when we traverse from, say state *x* to state *y*, in case of chess game. The non-invertible functions are: *sin(x)*, *cos(x)*, and mapping from composite number to unique primes. For all these you cannot uniquely determine, the argument from the value of the function.

## 8.6 Memory Requirements for Search Algorithms

It is important, what amount of memory is used by the graph search algorithms. We will consider first DFS algorithms for this discussion, followed with BFS, and finally the *frontier search* algorithm. The search of a graph can be specified explicitly, by listing all its nodes and edges in a data structure, which should be large enough to hold the entire graph. When the search is specified implicitly, the graph is described by an initial node, and a set of operators that generate all the successor nodes for any given current node. In graph search, we are usually concerned with very large graphs, hence they need to be specified implicitly. To *generate* a successor node means, to create explicit data structure that represents the node, and to *expand* a node means to generate all its children [1, 2].

### 8.6.1 Depth-First Searches

We consider first the depth-first search algorithm, as it has the minimal space requirements. The DFS can be implemented using *last-in first out stack* data structure. The nodes are stored and taken off from the top of the stack. Every time a node is taken from the top of the stack, it is expanded, and the resulting children are are placed on the top of the stack. When recursive implementation is used, every time a node is generated, the DFS is called recursively, on each of the child nodes generated. The memory requirements of DFS, in both the explicit DFS and recursive implementation, is only linear in the maximum depth search, and it is generally not of much significance.

The DFS consists number of drawbacks, e.g., in a graph with cycles, a pure DFS may not terminate. Even in a tree, the first solution found by DFS is not necessarily optimal. Both of these deficiencies can be solved by depth-first iterative deepening (DFID) method. This method performs a series of depth first searches, to successively greater depths, until the goal state is found. At this point, the current node stack is solution, with top most node as goal, and bottom node to top node sequence as the path from start node to the goal node. If recursive procedure is used, then the recursive call stack contains the solution. The space complexity of DFID is linear, as in the case of DFS.

Another drawback of DFS is such that it cannot be easily rectified. In a graph having multiple paths to the same state, DFID as well all the DFS can generate far

more states than the number of actual states. It is due to the inability of DFS and DFID to detect the duplicate nodes which corresponds to the same states. As an example, consider a *grid graph*, where each node has four adjacent nodes (north, east, south, and west) as its neighbors.[1] An efficient DFS would keep track of the operator used to generate a node from its parent, and will not apply its inverse when generating its children. This will reduce the branching factor from four to three. Accordingly, when considering a search in a radius $r$ of this graph, the DFS would generate $O(3^r)$ nodes. The large majority of these nodes are duplicate, because, there are only $O(r^2)$ unique states with in a radius $r$. For example, applying, East followed by North, generates the same states, as applying the North followed by East. Hence, the DFS can generate exponentially more nodes than the actual states available in the problem space [1].

### 8.6.2   Breadth-First Searches

As a general solution to BFS, the search algorithm stores all the generated nodes and compares the newly generated nodes with the stored nodes to find out if there are any duplicates. The BFS maintains two nodes lists, a *Closed-list* which is a list of nodes that have been expanded, another list called *Open-list* that is generated but yet not expanded. The Open-list is managed as first-in fist-out (FIFO) queue. If a head node of the Open-list is not the goal state, it is removed from Open-list, expanded, marked as Closed, and its children generated due to expansion are added to the tail of the Open-list (i.e., appended).

Apart from the above operations, as each node is generated, it is also added to a *hash table* if it is already not in this table. This storing in hash table is needed for fast checking of the nodes in future as, whether the same node is reached again (through a different path). For the implementation, every generated node is typically stored in the hash table. The Open-list consists of pointers originated through the hash table, and each node of Closed-list is indicated by a flag in the hash table storage.

The main drawback of this method is the memory requirements to store the nodes, which is exponential.

## 8.7   Problem Formulation for Search

Given a problem for solution using AI search, we need to formulate its solution through a state space search. This requires discovering the start and goal states, which may be explicitly specified in the problem, else we need to discover from the problem specification. The next step is to find out the possible moves to generate children at each state. Consider the following example to formulate a search problem.

---

[1]See Fig. 8.10a, p. 235, in exercises at the end of this chapter.

**Example 8.3**  Farmer, goose, grains, fox problem.

A farmer (F) wants to move himself, a fox (X), a goose (G), and some grains (R), across a river. Unfortunately, his boat is so tiny that he can take only one of his possessions across on any trip. Worse yet, an unattended fox will eat a goose, and an unattended goose will eat grains, so the farmer must not leave the fox alone with the goose nor the goose alone with the grains. What is he to do?

We create a node for each safe position. Then draw the possible next moves by a link from the one to the other. The end result is a graph with 10 nodes shown in Fig. 8.7. The states are marked as 1–10. State 1 indicates that all the four objects are one side of the river bank, and other side (separated by symbol |) is empty. Only one move is possible from state 1, caused by transporting of goose and farmer by boat to other bank. This results to state 2. Then, the farmer leaves the goose and returns back, the new state is 3. From state 3 two alternate moves are possible, to states 4, 5. These states result to states 6, 7, respectively.

The states 6, 7 merges to a single state 8, which has one child state 9, and next state 10 is goal state, indicating that all objects are to the destination back of river.

What one needs to do is to find a shortest path from the state where all objects are on one side of the bank, to the goal state where all the object have been transported to other side of the river bank.

The general approach is to search the paths until either the goal is reached or no more path is left to be explored. We note that the goal state can be reached by several paths, but to avoid loops and cycles. We note that each state is *invertible* also. When, loops are not considered, there are two paths to goal as: 1–3, 4, 6, 8–10, and 1–3, 5, 7, 8–10.                                                                                       □



**Fig. 8.7**  Farmer-goose-grains-fox problem as graph search
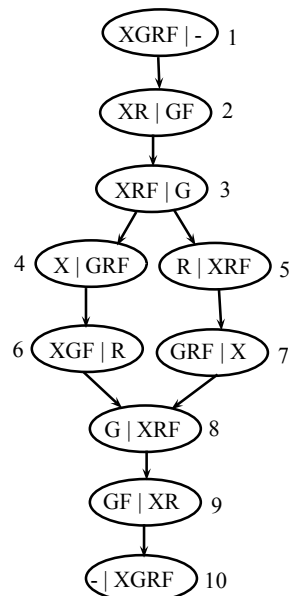
## 8.8   Summary

All the search strategies discussed in this chapter are called *blind search*. In the absence of any idea as what direction of search may lead to a solution faster, one need to carry out the *exhaustive* search. In all these searches, the time complexity is exponential.

A graph search for AI turns out to be a tree search. Unlike conventional data searching in a tree with tree's physical dimensions are given, in a AI search one need to *generate-and-search* simultaneously. The search space is called *state space*, and nodes of the tree are called *states*.

The *BFS* (breadth-first search) has time and space complexities both as $O(b^d)$, and in *DFS* (depth-first search) has these complexities as $O(b^d)$ and $O(d)$, respectively. The *Depth-first iterative-deepening* (DFID) provides better average case in space complexity, as it can locate the shallow nodes faster. However, the worst case time and space complexities are same as that in *DFS*. The *bidirectional* search reduces the time complexity to half exponential, i.e., $O(b^{\frac{d}{2}})$ in both the depth and breadth searches.

To analyse the time complexity of DFS, a new parameter, called *edge branching factor* (e) is important. It is "average number of different operators which are applicable to a given state." In case of trees, both the *edge branching factor* and the *node branching factor* are equal. However, for the graphs in general, the edge branching factor may exceed the node branching factor. In a graph with multiple paths to the same state, any depth-first search, including DFID, can generate far more nodes than there are states. This is due to the inability of the DFS algorithm to detect most duplicate nodes that represent the same state. Following table gives comparison in brief:

| Search algorithm | Properties |
|---|---|
| Breadth-first search | Uninformed, Non-optimal (Exception:Optimal only if you are counting total path length), Complete |
| Depth-first search | Uninformed, Non-optimal, Incomplete |

## Exercises

1. Describe a state space in which iterative deepening search performs worse than depth-first search.
2. Three missionaries ($m_1, m_2, m_3$) and three cannibals ($c_1, c_2, c_3$) are on one side of a river, along with a boat that can hold one or two people. Suggest a solution to get everyone to the other side, without ever leaving a group of missionaries on any side of the river outnumbered by the cannibals in that place. Formulate the solution for this, draw the graph, and demonstrate the goal search through *BFS*.

3. Given a 5-litre jug full of water, and an empty 2-litre jug. The jugs have no marking or level indicator. The goal is to fill 2-litre jug with exactly one litre of water. Give the corresponding state diagram. Assume that following are some example of moves: $p_{5,2}$ (pour 5 l into 2 l jug), $e_2$ (empty two litre jug), etc.

4. John has two jugs: one holds exactly 3 l and one exactly 4 l, but they have no subdivision marking. A recipe calls for exactly 5 l. He has a source of water but no other jug. Draw the state-space graph and determine the sequence of steps in getting exactly 5 l (into the recipe) that wastes the least amount of water. Suggest the state-space search approach to be followed.

5. Figure 8.8 shows a search tree with states $A$ to $O$ along with estimated distance from that state to the goal. The state with distance zero is goal itself.

   a. Make use of *BFS*, and show the steps along with the elements of Queue data structure as well the distance from goal, for each step.
   b. Make use of *DFS*, and show the steps along with the elements of Stack data structure as well the distance from goal, for each step.

6. Manually run the Algorithm 8.2 for the graph shown in Fig. 8.4 considering that start node is $S$ and goal node is $H$. While running this algorithm, show the progressive contents of the stack.

7. Given the blocks world indicated in Fig. 8.9, it is required to get to goal state from the initial start state. Construct the search tree for:

   a. BFS
   b. DFS

   Assume that following rules for moves will be followed by the robot arm for carrying out this job:

   • *stack*(*x*, *y*): stack block *x* on block *y*,
   • *lift*(*x*): lift-up the block *x*,
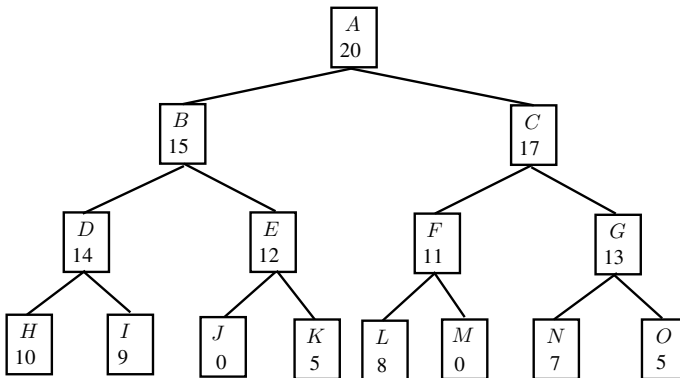


**Fig. 8.8** State—space tree
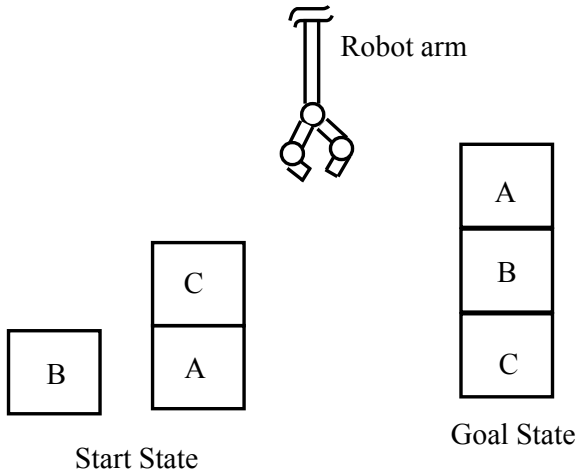
**Fig. 8.9**  Blocks world

- *putg*(*x*): put block *x* on ground,
- *unstack*(*x*, *y*): unstack block *x* from block *y*.

8. Find space and time complexity for DFS, for parsing a NL sentence of 50 words. Suggest the assumptions you would make to reduce the space and time complexity.
9. Give examples of graphs having different edge and node branching factors.
10. For the graphs shown in Fig. 8.10a–d, with start node *S* and goal node *G*, find out the average node branching factor and average edge branching factor.
11. For the following board games, find out the worst case path length to goal, total number of paths to goal, node branching factor, and edge branching factor.

   a. Rubics-cube
   b. Tic-tac-toe
   c. 8-puzzle
   d. Sudoku

12. Consider that a large number of websites are available, with their domain names in a domain-name-file. Each website has index-page (homepage) and other pages linked directly or indirectly from the homepage. In addition, there may be links from website pages to other website pages. Suggest a method and write an algorithm to crawl (scan) these websites to create an index table as follows: each row in table has one entry corresponding to each domain name, comprising domain name and five most frequent keywords: {domain-name, key-word1, …, key-word5}. Your algorithm must follow a suitable search approach.

   a. Give the justification for the type of search you have used.
   b. Give the time and and space complexity analysis.
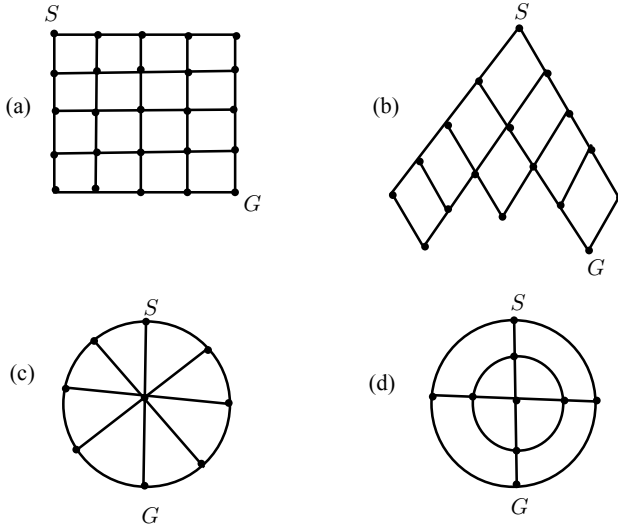
**Fig. 8.10** A graph with start node *S* and goal node *G*

   c. Is the search carried out by this algorithm *complete*?
   d. Is the search *optimal*?
13.  a. List four criteria that are generally used to evaluate the search.
     b. List two limitations, for each of the following search methods: DFS, BFS, Bidirectional.
     c. Give an example of problem where DFS will work better than BFS.
     d. Give an example of problem where BFS will work better than DFS.
14. There are 10 pages of power-point presentation slides, which are inter-linked through pointers, with each page having not more than 5-links. The "start" page is given, and "end" page is identified by "Thanks". Suppose only way to navigate through these pages is through links, and no idea available about the actual order of the pages.

   a. What is number of transitions in worst case to reach to the end page?
   b. What is the number of transitions in best case?
   c. What is time complexity (i.e., number of nodes visited) in DFS, BFS, and iterative deepening DFS?
   d. What is space complexity in DFS and BFS?

15. Discuss all the drawbacks of search in case the *Closed-list* is not used in search algorithms.
16. Solve the *Towers of Hanoi* problem using DFS, and find out the space and time complexity for this for *n* disks. Also answer the following:

   a. What is maximum branching factor for *n* disks?
   b. What is average branching factor for *n* = 4 disk?

17. Imagine there are 10 million websites in the entire Internet. Suppose that a search engine uses certain strategies to index these websites on regular basis, to maintain and update a very large database of indexes. Answer the following through your imaginations/logic as well as explore them on web.

    a. What strategy should search engine use to navigate through these websites through the links.
    b. What strategy should be used to create the index of the home-page of every website?
    c. What should be criteria to update the indexes?
    d. How to avoid the re-indexing of already indexed nodes?

18. Can every AI search be carried out using bidirectional search? What is necessary condition for this, if any?

19. For a search-tree of depth $d$ and branching-factor also $b$, carry-out the following analysis for DFS and BFS:

    a. Average case time complexity.
    b. Average case space complexity.
    c. Best case time complexity.
    d. Best case space complexity.

20. Suggest an architecture of hash-table for storage of nodes found in graph search, and quick searching of the same when it is found again next-time in graph search.

21. What is the memory requirement (in KB or MB) for data-structure for 8-puzzle for each of the following searches?

    a. BFS
    b. DFS

    Assume that only the open-list is maintained to store the nodes. Each node correspond a state of the 8-puzzle board, and is equal to 9-bytes (in each one byte, 4-bits are for square number, and 4-bits for integer number in that, including the space). Note that, when head nodes are deleted, the memory is freed.

22. Suggest an approach to combine the DFS and BFS in a single algorithm, so that at certain states you can choose DFS in place of BFS or vice-versa, based on meeting specific situation. For example, to check for shallow goals BDF can be used while for deep goals DFS can be tried. What data structures you will use to implement such type of search, as well the true strategies?

## References

1. Korf RE et al (2005) Frontier search. J ACM 52(5):715–748
2. Korf RE (2008) Linear-time disk-based implicit graph search. J ACM 55(6):26:1–26:40. https://doi.org/10.1145/1455248.1455250

3. Korf RE (1985) Depth-first iterative-deepening: an optimal admissible tree search. Artif Intell 27:97–109
4. Ludger GF (2009) Artificial intelligence—structures and strategies for complex problem solving, 5th edn. Pearson Education, New Delhi
5. Nilsson NJ (1980) Principles of artificial intelligence, 3rd edn. Narosa Publishing
6. Tarjan R (1972) Depth-first search and linear graph algorithms*. SIAM J Comput 1(2):146–160

# Chapter 9
# Heuristic Search

**Abstract** This chapter provides in depth study of heuristic search methods—the methods for searching the goal (solution) to problems, that are more like human, and do not follow the exhaustive search approach, making them far more efficient than the uninformed search methods. The introduction starts with formal definition of heuristic search, then follows Hill-climbing searches, their algorithm and analysis, best-first search, its algorithm and analysis, optimization, A-star search, and approaches to better heuristics. Finally, the search methods—simulated annealing (based on treatment of metals), Genetic Algorithm (GA)-based search method, along with their analyses are presented, followed with chapter summary, and at end an exhaustive list of practice exercises, along with multiple-choice questions are provided.

**Keywords** Heuristic search · Hill-climbing search · Best-First Search · A-star search · Simulated annealing · Genetic algorithm

## 9.1 Introduction

The subject of combinatorial optimization comprises a set problems that are central to the domain of computer science. The field of combinatorial optimization aims to develop efficient algorithms to find out minimum/maximum values of some function having large number of independent variables. The function is usually called *objective* function or *cost* function, and represents the quantitative measure of the "goodness" of some complex system. The cost function depends on the total configuration of the system which comprises many parts.

The best-quoted example of a combinatorial optimization problem is Traveling Salesman Problem (TSP). It is stated as: given a list of *n* cities, and distance between every two cities, it is required to plan a salesman's route which guarantees to pass through every city once only, covering all the cities, and finally returns to the starting point. The order of cities to be visited should be so planned, that the total distance, to cover all the cities, is minimum. Instead of distance, it can be cost of travel from city to city, or some other parameter. For the sake of generality, we call this as

*cost*. The problems of this type are common in the areas of *scheduling* and *design*. Two subsidiary problems in these areas are of general interest: 1. Predicting the estimated cost of a salesman's optimal route averaged over some arrangement of cities, having given the arrangement of cities and the distance between pairs of cities, 2. Estimating/obtaining the upper bounds of computing efforts necessary to determine the optimal route.

All the exact methods (also called exhaustive) known so far, for determining the optimal route of the salesman problem, requires the computing efforts, that grows exponentially with the total number of cities $n$. Hence, to carryout the solution in realistic times, the exact solution can be attempted only for a small number of cities $n$, may be less than a hundred. The TSP belongs to a class of large set of problems, called *NP-Complete* (NP for nondeterministic polynomial in time) problems. So far, no method of exact solution having computing effort bounded by a polynomial power of $n$ (say $n^k$, $k \geq 1$) has been found for any of these problems.[1]

However, if such a solution was found for any problem, say $A$, which is NP-complete, then it would be possible to map to $A$, all the remaining NP problems, as all NP problems are member problems of $A$ (an NP-complete). However, it is yet not known as what are the features of the individual problems that make it NP-complete, and that cause this level of difficulty of solving!

The problems of NP-complete class are at common place in many situations of practical interest, and hence the importance of their solutions. Fortunately, the solution methods, called, *heuristic methods* have been developed, which have computational requirements proportional to only a limited powers of $n$ (the size of the problem). Unfortunately, there is no universal heuristic method which can be applied to all types of problems. In fact, the heuristics are not general, but problem-specific, and there is no guarantee that one heuristic procedure for finding near-optimal solutions for one NP-complete problem will be effective for another problem also.

Fundamentally, there exist two basic approaches to heuristics: 1. "divide-and-conquer" and 2. "iterative improvement". The first approach is based on the concept of dividing the original problem into subproblems of manageable sizes, and then the subproblems are individually solved. At the end, the solutions to the subproblems are patched back together to get the complete solution. For this method to produce a good solution, it is necessary that subproblems are naturally disjoint, and the division of the original into subproblems is proper, in the sense that errors made in patching up the subproblems do not offset the gains obtained in applying more powerful methods to the subproblems.

Other approach to heuristics is based on iterative improvement, where one has to start with a *known configuration*. And, for this configuration, a standard rearrangement operation is applied to all parts of the system in turn, until a rearranged configuration which improves the cost function is discovered. As a next step, this rearranged configuration becomes the new configuration of the system, and the process is repeated until we reach to a configuration such that no further improvements

---

[1]The NP-Complete problems require exponential power of computing efforts, in terms of $n$, i.e., $k^n$.

can be found. The iterative improvement comprises a search space for rearrangement, so that there is a flat region in space, indicating that an optimized solution has reached—called global maxima.

Instead of settling to global maxima, the search quite often gets stuck-up in a local maxima. Due to this, it is usual to perform the search several times, starting from different randomly generated configurations, and save the best result, so as to reach the global maxima.

This chapter presents the heuristic methods for AI search problems. These methods are better informed, hence explore the state space in a more right directions. The analysis and complexities of these methods are also discussed.

**Learning Outcomes of this Chapter**:

1. Describe the role of heuristics and describe the trade-offs among completeness, optimality, time complexity, and space complexity. [Familiarity]
2. Select and implement an appropriate informed search algorithm for a problem by designing the necessary heuristic evaluation function. [Usage]
3. Evaluate whether a heuristic for a given problem is admissible (i.e., can guarantee optimal solution). [Assessment]
4. Design and implement a genetic algorithm solution to a problem. [Usage]
5. Design and implement a simulated annealing schedule to avoid local minima in a problem. [Usage]
6. Design and implement $A^*$ search to solve a problem. [Usage]
7. Compare and contrast genetic algorithms with classic search techniques. [Assessment]
8. Compare and contrast various heuristic searches vis-a-vis applicability to a given problem. [Assessment]

## 9.2  Heuristic Approach

The search efficiency can improve tremendously—reducing the search space, if there is a way to order the nodes to be visited in such that most promising nodes are explored first. These approaches are called *informed* methods, in contrast to the uninformed or blind methods discussed in the previous chapter. These methods depend on some heuristics determined by the nature of the problem. The heuristics is defined in the form of a function, say $f$, which some how represents the mapping to the total distance between start node and the goal node. For any given node $n$, the total distance between start and goal node is $f(n)$, such that

$$f(n) = g(n) + h(n), \tag{9.1}$$

where $g(n)$ is distance between start node and the node $n$, and $h(n)$ is the distance between node $n$ and the goal node. We note that $g(n)$ can be easily determined

and can be taken as shortest. However, the distance to goal is $f(n)$, which requires computation of $h(n)$, called heuristics, cannot be so easily determined. In deciding the next state every time, which is represented by node $n$, the state is chosen such that $f(n)$ is minimum.

Considering the case of the traveling salesman problem, which otherwise, is a combinatorially explosive problem, with exponential time complexity of $O(n!)$ for $n$ nodes, reduces to only $O(n^2)$ if every time the next node selected is the nearest neighbor, that is, the one having shortest distance from the current node.

Similarly, in the 8-puzzle problem, the next move is chosen the one having minimum *disagreement* from the goal, i.e., having minimum number of misplaced positions with respect to the goal.

The heuristic methods reduce the state space to be searched, and supposed to give the solution, but may fail also.

## 9.3   Hill-Climbing Methods

The name hill-climbing comes from the fact that to reach the top of a hill, one selects the steepest path at every node, out of the number of alternatives available. Naturally, one has to sort the slope values available, pick up the direction of move having highest angle, then reach to the next point (node) toward the hill top, then repeat the process. The hill-climbing Algorithm 9.1 is an improved variant of the depth-first search method. A Hill-climbing method is called *greedy local search*. Local, because it considers a node close to the current node, at a time; and greedy because it always selects the nearest neighbor without knowing its future consequences. The inputs to this algorithm are **G**, **S**, and **Goal**, which stand for—graph, start(root) node, and the goal node, respectively.

Consider the graph shown in Fig. 9.1a, where start node is $A$ and goal node is $G$. It is required to find out the shortest path from node $A$ to node $G$, using the method of hill-climbing. Figure 9.1b shows the search tree for reaching to goal node $G$ from start node $A$, with shortest path $A$, $B$, $D$, $E$, $G$, and path length 14. It can be easily worked out that this approach cannot lead to shortest path always.

Though simple, hill-climbing suffers from various problems. These problems are prevalent when hill-climbing is used to optimize the solution.

- **Foothill Problem**: This occurs when there are secondary peaks or *local maxima*. These are mistaken for the *global maxima*, as the user is left with false sense of achieving the goal.
- **Plateau Problem**: This occurs when there is a flat region separating the peaks.
- **Ridge Problem**: It is like a knife edge or an edge on top of a hill, both the sides are valleys. It again gives a false sense of top of the hill, as no slope change appears.

**Algorithm 9.1** Hill-Climb(Input: **G**, **S**, **Goal**)

1: **Open** = [S]
2: **Closed** = *nil*
3: **if Open** = *nil* **then**
4:     return *fail*
5: **end if**
6: **repeat**
7:     **if Open**.*Head* = **Goal then**
8:         return *success*
9:     **end if**
10:     expand **Open**.*Head* and generate children's set, call it **C**
11:     reject all paths in **C** having loops
12:     delete **Open**.*Head* and **insert** it into **Closed**
13:     sort **C** in order of heuristic, with best heuristic node in the front
14:     **insert C** at the front of **List**
15: **until Open** = *nil*
16: Return *fail*
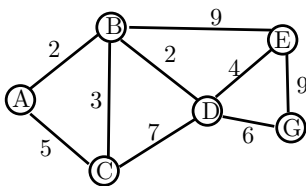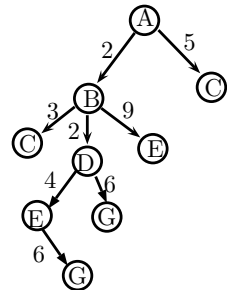


(a) Graph to be searched for goal 'G'.          (b) Search-Tree.

**Fig. 9.1**  Graph with Hill-climbing search

Due to above difficulties, a hill-climbing algorithm is not able to find a goal, even if the goal exist. Thus, hill-climbing algorithms are *not complete*.

Consider the following phenomenas:

1. Rotating the brightness knob in control panel of an analog TV does not improve the quality of picture,
2. While testing a program, running it again and again, with different data sets does not indicate new discovery of errors,
3. Participating in a sports again and again (without new ideas and training), does not improve further performance.

In all the above three cases, we strive for optimum performance. In case 1, the adjustable parameter is TV control, in second types of different input data, and in the third, adjustable parameter is more energy and preparedness. But it is appears that optimum has reached (a highest point in performance, from where no improvements take place, an indication of saturation point, the goal, but not the true goal).

The above scenarios are created due to either of the foothill problem, or plateau, or ridge.

Local Versus Global Search

The preference for local search where only one state is considered to further expand at a time, out of the newly explored states, is good for a memory efficiency, in comparison to expanding all the successor nodes, but appear to be a too extreme step. Thus, instead, $k$ best nodes out of successors generated are considered for further expansion. This gives rise to a new method, called *local beam search*.

However, the local beam search too will lead to concentration to a specific direction to those successors which generate more potential nodes. Consequently, this search also ultimately becomes a local search. A better solution is to elect these $k$ nodes such that they are not the best successors, but randomly selected, out of the next generation of successor nodes. This new method is called *stochastic beam search*. Thus, we grow a population (of nodes) and from that we generate further another population by random selection, as well as on some criteria of merit. Thus, we reach closely to the approach used in *genetic algorithms*-based search.

## 9.4  Best-First Search

If a problem that has a very large search space and can be solved by iteration (unlike theorem proving),[2] there is usually no alternative to using the iterative methods. In such problems, there is a serious issue in bounding the effort to make the search tractable. Due to this, the search should be kept limited in some way, e.g., in terms of total number of nodes to be expanded, or maximum depth to which it may reach. Since there is no guarantee that a goal node will be ultimately reached, an evaluation function is invoked to help us decide the approximate distance to the goal, for any given node at the periphery of the search. This or a similar function can also be used for deciding which tip node to sprout next. Hence, the effort for proper design of valuation functions limits the later stage difficulty in solving the problem. Note that heuristics-based methods are called *informed search* methods, in contrast to the *uninformed* methods discussed in the previous chapter.

Among all the problem-solving strategies based on search the heuristics, one of the most popular methods of exploiting heuristic information to cut down search time is *best-first* search strategy. This strategy possess general philosophy of using heuristic information to assess the "merit" latent in every candidate search avenue exposed during the search, and then continues the exploration along the direction of highest merit. There are no local maxima "traps" in best-first search like in hill-climbing methods. The best-first search strategy is usually seen in context of path-searching problems—a formulation that represents many combinatorial problems with practical applications, such as routing telephone traffic, layout of printed circuit

---

[2]In problems such as theorem proving, the search must continue until a proof is found.

board, scheduling, speech recognition, scene analysis, mechanical theorem proving, and problem-solving.

The heuristic approach typically uses special knowledge about the domain of the problem being represented by the graph to improve the computational efficiency of solution to particular graph-searching problem. However, the procedures developed via the heuristic approach generally have not been able to guarantee that minimum cost solution paths will always be found.

Given a weighted directional graph $G = (V, E, W)$ with a distinguished start node $S$ and a set of goal nodes $R$, the optimal path problem is to find a least cost path from $S$ to any member of $R$ where the cost of the path may, in general, be an arbitrary function of them, weights assigned to the nodes and branches along that path. A Generalized Best-First Search (GBFS) strategy will pursue this problem by constructing a tree $T$ of selected paths of $G$ using the elementary operation of node expansion, that is, generating all successors of a given node, Starting with $S$, the GBFS will select for expansion that leaf node of $T$ that features the highest "merit," and will maintain in $T$ all paths which have been encountered so far. And, that still appear as viable candidates for *sprouting* an optimal solution path. When no such candidate is available for further expansion, the search terminates. In that case the best solution path found so far is issued as a solution; if none has been found, a failure is declared. Due to its nature of search, the best-first search is also called *branch-and-bound method*, i.e., branching a search to other directions to which path cost is minimum, and bounding the cost to that minimum, until a better minimum is found after next expansion.

### 9.4.1 GBFS Algorithm

A best-first search algorithm maintains two lists of nodes, an "Open-list"and a "Closed-list". The Closed-list contains those nodes that have been expanded, by generating all their children, and the Open-list contains those nodes that have been generated, but not yet expanded. At each iteration of the algorithm, an Open node having smallest total cost from start node, is expanded, moved to the Closed-list and, its children are added into the Open-list [2].

The best-first search method takes the best node to be explored first, among the number of nodes in the open-list. When a node is selected as a candidate for expanding, its all children's distance is computed from the start node, which serve as heuristic value. This approach works because there is always a best node available for expanding, until the goal is reached or the entire search has taken place.

Since, best node is selected every time, it is guaranteed to give the best solution. The value of heuristic function $f(n)$ for a given node $n$, does not here include the distance from current node to the goal node, as required in Eq. , however, since the best path is chosen every time, it is likely to provide the optimum solution for the problem.

Algorithm 9.2 shows the steps for best-first search.

---

**Algorithm 9.2** Best-first Search(Input: **S**, **Goal**)

---
1:  **Open** = [*S*]
2:  **Closed** = []
3:  **repeat**
4:    **if Open**.*Head* = **Goal then**
5:      return *success*
6:    **end if**
7:    generate children's set **C** of **Open**.*Head*
8:    **if** *n* ∈ **C** already exists in **OPEN** and new *n* is reachable by shorter path **then**
9:      remove the old *n*
10:   **end if**
11:   **if** *n* ∈ **C** already exists in **Closed** and reachable by shorter path **then**
12:     replace *n* ∈ **C** by the same node from **Closed**, along with shorter distance from root
13:   **end if**
14:   **remove Open**.*Head* and **insert** into **Closed**
15:   **update** distance from root for all **C** nodes
16:   add all **C** to either side of **Open** and record their parents
17:   sort **Open** by path length so that least cost path node is at front
18: **until Open** = *nil*
19: return *fail*

---

**Example 9.1** Best-first search.

Fig. 9.2a shows the graph, and Fig. 9.2b shows the search tree using GBFS for reaching to goal node *G*.

Every node in best-first search shows the node identification along with its distance from the root node *S*. The order in which the nodes are explored is shown with dotted line. Since *G* is goal, its path from root is *S*, *A*, *C*, *G* with shortest path length 11. Note that any other path will be of longer or equal length.                                      □



(a) Graph
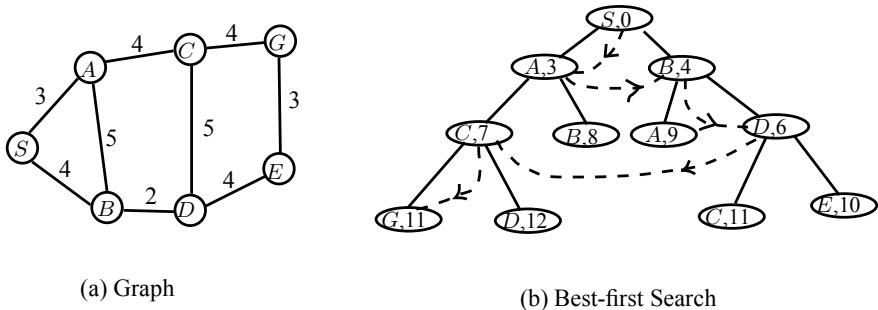
(b) Best-first Search

**Fig. 9.2** Best-First (Branch-and-Bound) search

### 9.4.2   Analysis of Best-First Search

As we have noted that, the best-first searches tend to put the searching effort into those subtrees that seem most promising (i.e., they are most likely of providing the best solution). However, the best-first searches require a great deal of bookkeeping for keeping track of all competing nodes, contrary to the great efficiencies possible in depth-first searches.

Depth-first searches, on the other hand, tend to be forced to stop at inappropriate moments thus giving rise to the horizon effect (number of possible states is immense and only a small portion can be searched). They also tend to investigate huge trees, large parts of which have nothing to do with any solution (since every potential arc of the losing side must be refuted). However, these large trees sometimes turn up something that the evaluation functions would not have found were they guiding the search. Sometimes the efficiencies and discovery potential of the depth-first methods appear to out-weight what best-first methods have to offer. In fact, both methods have some glaring deficiencies.

Optimizing Best-First Search

In practice, several shortcuts have been devised to simplify the computation of GBFS. First, if the evaluation function $f(n)$ used for node selection always provides optimistic estimates of the final costs of the candidate paths evaluated, then we can terminate the search as soon as the first goal node is selected for expansion, without compromising the optimality of the solution used. This guarantee is called *admissibility* and is, in fact, the basis of the branch-and-bound method. This we can observe, for example in a chess game, where goal is property of a configuration and not the property of path from start node. Hence, once a winning configuration is reached, there is no need to try it from other paths.

Second, we are often able to purge from tree $T$, large sets of paths that are recognized at an early stage to be dominated (i.e., superior) by other paths in $T$. This becomes particularly easy if the evaluation function $f$ is *order-preserving*, that is, if, for any two paths $p_1$ and $p_2$, leading from **S** to $n$, and for any common extension $p_3$ of those paths, the following holds (Fig. 9.3) [2]:

$$f(p_1) \geq f(p_2) \Rightarrow f(p_1 p_3) \geq f(p_2 p_3). \tag{9.2}$$

The property of *Order-preserving* is a judgmental version of the principle of *optimality* in *Dynamic Programming*. The principle states that, a path $p_1$ is judged to be more meritorious than another path $p_2$, such that both are paths from one source state **S** to some future state $n$, and there is a common extension $p_3$ of $p_1$ and $p_2$. In such a scenario, the common extension cannot later reverse the judgment made earlier. Under such conditions, there is no need to keep multiple copies of nodes in the tree $T$. Every time, the expansion process generates a node $n$, which already resides in $T$, only lower path to node $n$ be maintained, and the link from more expensive parent of $n$ is discarded. This has been illustrated in Fig. 9.3.

**Fig. 9.3** Order-preserving in
GBFS
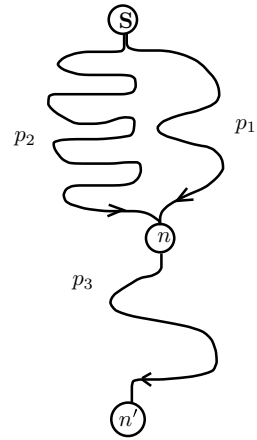


The best-first search allows revisiting the decisions. This is possible when a newly generated state by expansion of one of the state in Open-list is found in the closed-list also. The best-first would retain the shorter path to this node, and purge the other to save space. However, in a variant of best-first, called, *greedy best-first* search once a state is visited the decision is final and the state is not visited again, thus eventually accepting the suboptimal solution. This however, does not require the *Close-list*, thus saving the memory space significantly.

Special Cases of Best-First Search

Individual best-first search algorithms differ primarily in the cost function $f(n)$. If $f(n)$ is the total depth of node $n$ (not the distance from start), best-first search becomes breadth-first search. Note that breadth-first searches all the closer nodes (to start) before farther nodes. If $f(n) = g(n)$, where $g(n)$ is the cost of the current path from the start state to node $n$, then best-first search becomes Dijkstra's single-source shortest-path algorithm.

If $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic estimate of the cost of reaching a goal from node $n$, then best-first search becomes a new algorithm $A^*$ algorithm.

Breadth-first search can terminate as soon as a goal node is generated, while Dijkstra's algorithm and $A^*$ must wait until a goal node is chosen for expansion to guarantee *optimality*. Every node generated by a best-first search is stored in either the Open- or Closed-lists, for the following reasons:

1. To detect when the same state has previously been generated. This is to prevent expanding it more than once.
2. To generate the solution path once a goal is reached. This is done by saving with each node a pointer to its parent node along an optimal path to the node, and then tracing these pointers back from the goal state to the initial state.
3. To choose only the node, which is at shorter distance, when a newly generated node already exists in the closed-list.

The primary drawback of best-first search is its memory requirements. By storing all nodes generated, best-first search typically exhausts the available memory in very short time on most machines.

While breadth-first search manages the Open-list as a first-in first-out queue, generalized best-first searches manage the Open-list as a *priority-queue* in order to facilitate efficiently determining the best node to expand next.

All of these algorithms suffer the same memory limitation as breadth-first search, since they store all nodes generated in their "Open"or "Closed"lists, and will exhaust the available memory in a very short time.

## 9.5 Heuristic Determination of Minimum Cost Paths

The objective of heuristic determination of minimum cost path is to find an algorithm that searches a graph, $G = (V, E)$ to obtain an optimal path from start node $S$ to its perfect goal node $t$. In the search process, each time a new node is expanded, two things are stored with each successor node: 1. The cost of reaching to $n$ through a least cost path created so far, and 2. A pointer to the predecessor node of $n$. Ultimately, the algorithm gets terminated at some goal node $t$, and no further nodes are expanded. At this state, we can reconstruct the minimum cost path from $S$ to $t$, simply by chaining back the nodes from $t$ to $S$ through the pointers to predecessor nodes.

In order to expand as few nodes as possible for searching an *optimal* path, the search algorithm must constantly make an informed decision about what node is to be expanded next. The expansion of nodes that are not going to be in the optimal path, will result to wastage of efforts. On the other hand, if the algorithm ignores the nodes that might be in the optimal path, it will fail to find such a path, in that case the algorithm is not *admissible*. Thus, a good algorithm obviously needs some way to evaluate the available nodes to determine which node to expand next.

Consider that an *evaluation function* could be calculated for some node $n$. Let, $f^*(n)$ is estimated minimum distance from start state to goal state, constrained through node $n$. Assume that this evaluation function be defined in such a way that the node with smallest value of $f^*$ is expanded next. We will show that for a suitable choice of the evaluation function $f^*$, the algorithm $A^*$ is guaranteed to provide an optimal path to a preferred goal node from the start node $S$, which is sufficient condition for the *admissibility* of the algorithm.

### 9.5.1 Search Algorithm A*

By far, the most studied version of best-first-search is the algorithm $A^*$, which was developed for additive cost measures, that is, where the cost of a path is defined as the sum of the costs of its arcs. The $A^*$ is in fact a family of algorithms, we will see it shortly. The algorithm makes use of ordered state-space search and the estimated

heuristic cost to determine the evaluation function $f^*$, a function which provides the goal state. This process is similar to the best-first search, but now it is unique in the sense that it defines $f^*$, which will provide a guarantee of optimal path to the goal state. The $A^*$ algorithm is in the class of *branch-and-bound* algorithms, which are common in use in operations research for finding the solution of a problem, in the form of a shortest path in a graph [1].

The evaluation function $f^*(n)$ estimates the quality of the solution path through node $n$, and it is based on values returned from two components: 1. $g^*(n)$, and 2. $h^*(n)$. The first component is the *minimal cost* of a path from a start state to $n$, and the second component, called *heuristic value*, is a *lower bound* on the minimal cost of a solution path from state $n$ to goal state.

In graphs, $g^*$ can have error only in the direction of overestimating the minimal cost. In future steps of the algorithm, if a shorter path is found, the value of $g^*$ can be readjusted to lower side. The function $h^*$ carries the heuristic information, such that it has the capability that ensures that value of $h^*(n)$ is less than $h(n)$. This later condition is essential to the optimality of $A^*$ algorithm. The property, as per which, $h^*(n)$ is always less than $h(n)$, is called *admissibility condition*.

To match this cost measure, $A^*$ employs an additive evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the currently evaluated path from $S$ to $n$ and $h$ is a heuristic estimate of the cost of the path remaining between $n$ and some goal node. Since $g(n)$ is *order-preserving* and $h(n)$ depends only on the description of the node $n$, therefore $f(n)$ is also order-preserving, and one is justified in discarding all but one parent for each node.

The admissible search algorithm for $A^*$ (A-star) is given as Algorithm 9.3.

---

**Algorithm 9.3** Admissible search $A^*$(Input: **G**, **S**, **Goal**)

---

1: **Open** $= [S]$
2: **Closed** $= []$
3: compute $f^*(n)$ for all $n \in$ **Open**
4: **repeat**
5:     select the open node $n$ whose $f^*(n)$ is smallest
6:     resolve ties arbitrarily, but always in favor of any node $n \in$ **Goal**
7:     **if** $n \in$ **Goal then**
8:        move $n$ to **Closed**
9:        terminate algorithm
10:    **else**
11:        move $n$ to **Closed**
12:        apply successor operator to $n$
13:        calculate $f^*$ for each successor $n_i$ of $n$
14:        move all $n_i \notin$ **Closed** to **Open**
15:        move to **Open** any $n_i \in$ **Closed** and for which $f^*(n_i)$ is smaller now than it was when $n_i$ was in **Closed**
16:    **end if**
17: **until Open** $= nil$
18: return *fail*

---

## 9.5.2 The Evaluation Function

Let, for any graph $G$ and any set of goal nodes $Goal$, let us assume that $f(n)$ is the actual cost of an optimal path that is restricted to go through only the node $n$, i.e., from source $S$ to a preferred goal node $n$. Note that at the begin of the $A^*$ search algorithm, the constrained node $n$ is nothing but $S$. Hence, $g(n) = g(S) = 0$. Therefore, $f(S) = h(S)$ is the cost of unconstrained optimal path from node $S$ to preferred goal node, what so ever it is. Actually, for every node $n$ on optimal path, the condition $f(n) = f(S)$ holds, and for every node $n$ not on an optimal path, $f(n) > f(S)$ holds. Thus, although $f(n)$ may not be known in advance, it seems reasonable to use the evaluation function $f^*(n)$ as an estimate of $f(n)$. This is because determination of the true value of $f(n)$ may be main problem of interest.

In the following, we present some properties of search algorithm $A^*$ where cost $f(n)$ of an optimal path through node $n$ is estimated using an evaluation function $f^*(n)$. The function $f(n)$ can be written as the sum of two parts, $g(n)$ and $f(n)$:

$$f(n) = g(n) + h(n). \tag{9.3}$$

In the above, $g(n)$ is the actual cost of an optimal path from node $S$ to $n$, and $h(n)$ is the actual cost of an optimal path from node $n$ to a preferred goal node.

If we had the estimates of $g$ and $h$, we could easily get the estimate of $f(n)$, as the simple addition of the two. An obvious choice for $g^*(n)$ is, so far smallest cost path found by the algorithm, from $S$ to $n$. This indicates that $g^*(n) \geq g(n)$.

Through a simple example we will explain that the above estimate is easy to compute as the algorithm progresses through its computations [6].

**Example 9.2** Consider the subgraph shown in Fig. 9.4, with start node $S$ and three other nodes $n_1, n_2, n_3$, with cost on edges as the weights.

Having given this, we trace the algorithm $A^*$ as it proceeds. With $S$ as start node, $n_1$ and $n_2$ are the successor nodes. The estimates for $g^*(n_1)$ and $g^*(n_2)$ are then 4 and 9, respectively. Let, $A^*$ expands the next node as $n_1$, and obtains the successors $n_2$ and $n_3$. At this stage $g^*(n_3) = 4 + 3 = 7$, and $g^*(n_2)$ is $4 + 4 = 8$. The value of $g^*(n_1)$ ultimately remains equal to 4, irrespective of the goal node. □

**Fig. 9.4** Admissibility test
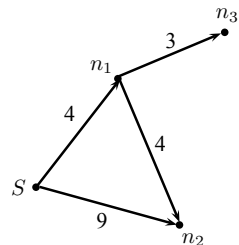
We have the following arguments for an estimate $h^*(n)$ of $h(n)$, which we are not able to compute for this example, as no criteria for heuristics is specified here. We usually rely on information from the problem domain for heuristics. For example, many problems that can be represented in the form of problem of finding minimum cost path through a graph that contains some "physical" information, and this information is used to form the basis for estimation of $h^*$. When considering the connection between cities through roads, the value $h^*(n)$ might be air distance from city $n$ to goal city, because this distance is the shortest possible length of any road connecting city $n$ to the goal city. Thus, it is lower bound on $h(n)$. However, the above conclusion is based on the assumption that air connectivity between any two cities follows a straight line rule.

As another example, in an 8-puzzle problem, at node $n$, the distance $h^*(n)$ might be equal to the number of tiles dislocated with respect to the goal state.

We will discuss later about using information from specific problem domains, to form estimate of $f^*$. However, first we can prove that if $h^*$ is any lower bound of $h$, then the algorithm $A^*$ is admissible.

**Example 9.3** $A^*$-Search.

Figure 9.5 shows a graph, heuristic function table for $h(n)$ for every node in the graph, and tree constructed for $A^*$ search for the graph, for given start state $S$ and goal state $G$. To expand the next node, the one having smallest value of function $f$ is chosen out of the nodes in the frontiers. The function $f$ for a node $n$ is sum of three values: the $g$ value of the parent of $n$, the distance from parent of $n$ to the node $n$, and heuristic value (estimated distance from $n$ to goal, given in the table) indicated by $h$. In the case of a tie, i.e., two states having equal values of $f$, the one to the left of the tree is chosen.

If, in addition, $h(n)$ is a lower bound to the cost of any continuation path from $n$ to goal, then $f(n)$ is an optimistic estimate of all possible solutions containing the currently evaluated path. Then, terminating $A^*$ upon the selection of the first goal node does not compromise its admissibility. Several other properties of $A^*$ can be established if admissibility holds, such as the conditions for node expansion, node reopening, and the fact that the number of nodes expanded decreases with increasing $h$.

Based on the criteria set for $A^*$ (i.e., to always expand the node having smallest value of $f$). The distance to goal ($f$) for each node $n$ is: distance from source $S$ to parent, plus distance from parent to this node $n$, plus distance $h$ from this node $n$ to goal. For current node $n = A$, these distances are 0, 1, 6, respectively. The order in which nodes have been expanded for Fig. 9.5a, and shown in search tree in Fig. 9.5c with start node $S$ and goal node $G$ are: $(S, 0)$, $(B, 6)$, $(A, 7)$, $(B, 5)$, $(C, 6)$, $(C, 7)$ (with parent $A$), $(C, 7)$ (with parent $B$), $(G, 8)$, $(G, 9)$, $(G, 12)$. Finally, we note that the best path is corresponding to goal $(G, 8)$, and it is: $S, A, B, C, G$. Note that, in the $A^*$-tree, we followed the sequence $(S, 0)$, $(B, 6)$, with $(A, 7)$ and not $(C, 7)$, which are equally weighted. We chose the node to the left-side subtree. Had we chosen, $(C, 7)$ in place of $(A, 7)$, we would have reached to $(G, 9)$ as next node, which incidentally was not a good choice.                                                                                    □

(a) Graph.

| State | h |
|-------|---|
| S | 7 |
| A | 6 |
| B | 2 |
| C | 1 |
| G | 0 |

(b) Heuristic Table.
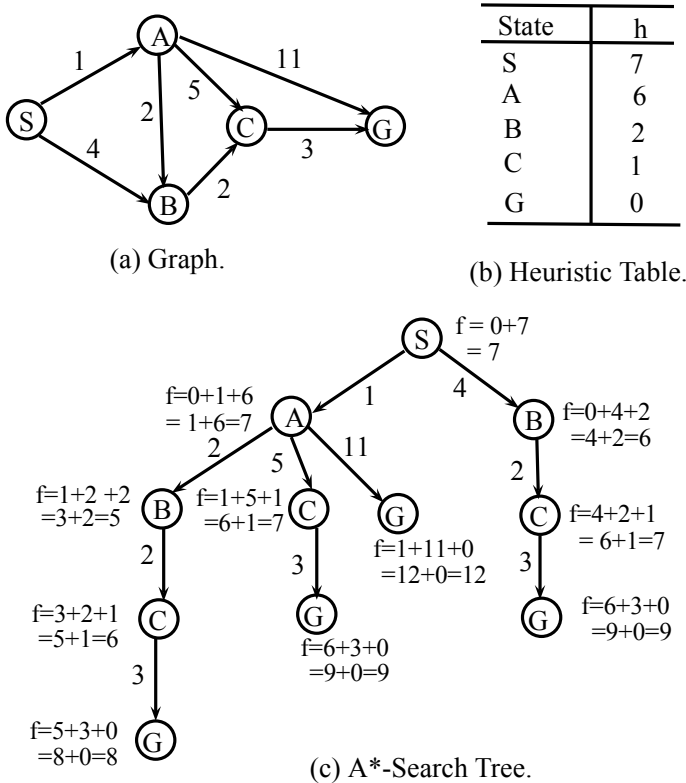


(c) A*-Search Tree.

**Fig. 9.5** Graph and A*-Search tree

## 9.5.3 Analysis of A* Search

The $A^*$ is actually a family of search algorithms, as mentioned earlier, and many other search algorithms are special cases of this algorithm.

The estimated value $h^*(n)$ for $h(n)$ can be obtained from the problem domain. For example, in the case of 8-puzzle problem or the 8-queen problem, this value is inverse of the number of tiles out of place with the goal or the inverse of number of queens giving checks to other queens. Even better value is, actual number of moves from $n$ to goal.

Let us consider the specific cases of $f^*(n)$ as follows:

- When $h = 0$, then $g = d$ (the distance to goal in the search tree). This algorithm is called as **A**, and it is identical to Breadth-First Search (BFS).
- We claimed that the *BFS* algorithm is guaranteed to find the minimum path length to the goal node. If $h$ is lower bound on $h^*$, i.e., $h(n) \leq h^*(n)$, for all $n$, then the algorithm will find an optimal path to a goal node. This algorithm is called as **A***.

### *9.5.4 Optimality of Algorithm A\**

A search algorithm *B* is called *optimal* if there does not exist any other search algorithm performing the searching in less time or space or can do the job by expanding fewer nodes, having a guarantee of solution quality as that of algorithm *A*. Hence, if $h(n)$ is lower bound on $h^*(n)$ then the solution of $\mathbf{A}^*$ is *admissible*. This estimate also concludes that any open node *n* may even be arbitrarily close to a preferred goal node.

In one way, we can define an optimal search algorithm as one that picks the correct next node at each choice. However, this specification of an algorithm is not of much use as this much specification is insufficient to design an algorithm. In fact, whether such an algorithm may ever exist is also an open question in itself [6].

## 9.6 Comparison of Heuristics Approaches

The heuristic search that finds the shortest path to a goal wherever it exists is called *admissible*. We may like to know, in what sense one heuristics is better than other, is called *informedness* of the heuristics.

When search is made, it is expected that same node will not be accessible from a shorter path later on. This property is called *monotonicity*.

The breadth-first search algorithm is admissible algorithm, because it searches a path at level *n*, before searching the paths at level $n + 1$, hence if the goal exists at level *n* it will be certainly found out. However the BFS algorithm is too expensive as a general purpose algorithm.

The $\mathbf{A}^*$ algorithm does not require $g(n) = g^*(n)$. This shows that there may be subgoals in the path, which are longer than $g^*(n)$; this is due to monotonicity [8].

**Definition 9.1** (*Monotonicity*) A heuristic function *h* is monotonous if for all the nodes $n_i, n_j$, where $n_j$ is descendant of $n_i$, such that

$$h(n_i) - h(n_j) \leq cost(n_i, n_j), \tag{9.4}$$

where $cost(n_i, n_j)$ is actual cost, in number of moves from node $n_i$ to $n_j$.

**Definition 9.2** (*Informedness*) For a problem, suppose there are two $A^*$ heuristic functions $h_1$ and $h_2$. Then, if $h_1(n) \leq h_2(n)$, for all states *n* in the search space, the heuristic $h_1$ is called *more informed* that $h_2$.

For example, the criteria of number of tiles out of place, in the 8-puzzle is better informed than the breadth-first or depth-first search methods. Similarly, the heuristics which calculates the number of transitions to reach the goal is better informed than the one considering the heuristics based on the number of tiles out of place. In general, a more informed is an algorithm, there is less expansion of space for searching [6].

Approaches to Better Heuristics

From the above two examples, we note that in some cases of search, the path matters (TSP), while in other the path does not matter, and only the final configuration matters (8-puzzle).

A simple algorithm for heuristic search could be considering only single state at a time rather than many states corresponding to many paths sprouting at the same time. Such algorithms are called *local search* in contrast to the *global search*, which maintains many active paths at the same time. The local search algorithms consume much less memory, usually a constant amount.

The *Branch-and-bound* algorithms are implicitly enumeration algorithms. These are the principal general methods for finding out optimal solutions for discrete optimization problems. The branch-and-bound algorithms are based on the following parameters:

$$(D, E, L, N, P, U), \tag{9.5}$$

where

$D$: Node dominance function,
$E$: Set of node elimination rules,
$L$: Node lower bound solution cost function,
$N$: Next node selection rule,
$P$: Partitioning or branching rule, and
$U$: Upper bound solution cost function.

A branch-and-bound algorithm is a two-step algorithm: first step is a *splitting or branching* step which returns two or more smaller sets $S_1, S_2, \ldots$, whose union covers $S$, where $S$ is set of candidates. Minimum of $f(x)$ over $S$ is $min\{v_1, v_2, \ldots\}$, where each $v_i$ is the minimum of $f(x)$ within $S_i$. The recursive application of this step defines a tree structure (a search tree) whose nodes are the subsets of $S$. The second step, called *bounding*, computes upper and lower bounds of the minimum value of $f(x)$, within a given subset of $S$.

Application of the branch-and-bound technique has grown rapidly. Representative examples of this include: *flow-shop* and *job-shop* sequencing problem, traveling salesman problem, *integer programming* problem, and *general quadratic assignment* problem. Though, the branch-and-bound algorithms are usually more efficient than complete enumeration, however, these algorithms have computational requirements that usually grow exponentially or high degree polynomial of the problem size $n$. In these cases, their usefulness is limited to small size problems.

These are other search techniques based on "natural phenomena". Under this we are going to discuss two techniques: (1) *Simulated annealing*, which is based on changes in the properties of metals and alloys due to heating them to higher temperature and then slowly decreasing their temperature; and (2) Something based on the Darwin's theory of evolution, called *genetic algorithms*.

## 9.7   Simulated Annealing

Annealing is process of treatment of metal or alloy by heating to a predetermined temperature, holding for a certain time, and then cooling to room temperature to increase ductility and reduce brittleness. The process of annealing is carried out intermittently during the working of a piece of a metal to restore ductility lost through repeated hammering or other working. Annealing is also done for relief of internal stresses. The annealing temperature varies with metals and alloys, and with properties desired, but must be done within a range, that prevents the growth of crystals. It is an optimization algorithm, its strength is that it avoids getting caught at local maxima— the solutions that are better than nearby, but not best.

Simulated Annealing (SA) is a probabilistic search for the global optimization of a problem for locating a good approximation to the global optimum of a given function, in a large search space. The name of the process and its inspiration come from *annealing* in metallurgical processes, where a function $E(S)$ needs to be minimized. This function is analogous to the internal energy of the system in that state. The goal of SA is to bring the system from some arbitrary initial state, to a state having minimum possible energy [7].

Process

The SA makes use of heuristics to reach to the goal state, such that at each step, the heuristic considers some neighboring state $s'$ of the current state $s$, and probabilistically decides of moving the system to move to $s'$ state or staying in $s$. When the above sequence of steps are repeated, the probabilities ultimately move the system to more stable states at lower energy. Typically, the iterations continue until the system reaches to a state that is good enough for the application, or until a given number of iterations are exhausted.

In SA we make one change from the normal heuristic search; we attempt to minimize the function's value instead of maximizing. So, instead of heuristic function it is called *object function*. This is like a *valley descending* rather than *hill-climbing*. Note that in 8-puzzle, for hill-climbing, we compute inverse of number of tiles out of place with respect to goal to obtain heuristic value. So, if zero tiles are out of place (i.e., at goal), the heuristic function is infinite. This would correspond to object function as zero (or minimum).

The physical substances usually move from higher energy configuration to lower levels, so that the valley descending occurs naturally. But, there is some probability that a transition to higher energy will occur, given by

$$p = e^{\frac{\Delta E}{kT}}, \tag{9.6}$$

where $\Delta E$ is positive change in energy level (difference, i.e., current cost–new cost, so $\Delta E$ is negative in valley descending), $T$ is temperature in Kelvin absolute temperature scale, and $k$ is *Boltzmann's* constant. As per this property, the probability to a large uphill move will be lower than probability of small move. Also, the probability that a large uphill move will take place, decreases as the temperature $(T)$

**Fig. 9.6**  Probability of uphill move in simulated annealing, as a function of temperature and change in energy
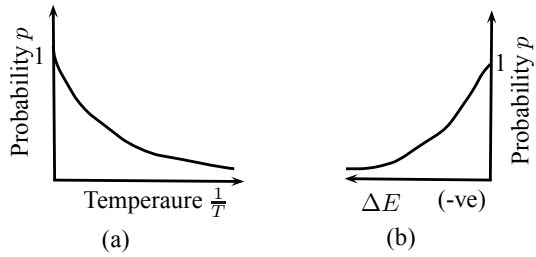


decreases. Figure 9.6 shows the effect of increase of temperature and decrease of $\Delta E$, on probability.

In other words, the uphill moves are more possible when temperature is high, but as the temperature decreases, relatively small uphill moves are made until finally process converge to a local minimum configuration.

The rate at which system cools is called *annealing schedule*. If cooling occurs too fast, the physical system will form stable regions high energy. That is, local but not global minima is reached. If a slower schedule is used, a uniform crystalline structure, which corresponds to a global minimum, will develop.

In search techniques, change in $E$ $(\Delta E)$ is equal to change in object function. The constant $k$ represents the correspondence between the unit of temperature and unit of energy. Since it is constant, we take probability $p$ in Eq. (9.6) as

$$p' = e^{\frac{\Delta E}{T}}. \tag{9.7}$$

SA mimics annealing process in metallurgy by combination of random search and hill-climbing. During metallurgical annealing, alloys are cooled at a controlled rate to allow for the formation of larger crystals. Larger crystals are chemically at a lower energy state than smaller ones; alloys made of crystals in the lowest energy state are comparably stronger and more rigid. At a high temperature, the search is a random walk, and as the temperature lowers the search gradually transits to a local search. Capturing this idea in an algorithm yields a random process over a space of configurations where the probability of actually moving to a new configuration is determined by the difference in energy and the current temperature. Algorithm 9.4 shows the steps for this process.

We note that SA uses

1. Iterative improvement,
2. Local random search,
3. Exploration, and
4. Greedy search.

When the temperature is high, atoms can move anywhere freely, and have equal probability. When temperature does down, this freedom is reduced.

---

**Algorithm 9.4** Simulated Annealing

---
1: $T$ = high
2: generate random solution
3: calculate energy ($E$) of the solution
4: set initial temperature $T$ (sufficiently high)
5: (Gradually decrease the temperature)
6: **while** $T >$ cut-off temperature **do**
7:     test solution $\leftarrow$ solution
8:     **for** $n$ iterations **do**
9:         adjust test solution
10:         calculate energy $E$ of test solution
11:         $\Delta E = E_1 - E_2$
12:         **if** $\Delta E < 0.1$ **then**
13:             update solution and energy $E$
14:         **else if** $e^{\frac{\Delta E}{T}} > random(0\ to\ 1)$ **then**
15:             update solution and $E$
16:         **end if**
17:         decrease $T$
18:     **end for**
19: **end while**
20: end

---

Formal Approach

The Boltzmann probability function tends to return True at higher temperature and False at lower temperature; thus in essence the search gradually shifts from random walk to local hill-climb.

A simulated annealing algorithm is suitable for minimization of an objective function $f$, having the mapping, $f : \mathbf{S} \to \mathbb{R}$, where $\mathbf{S}$ is some finite *search space*, and $\mathbb{R}$ is real number. Typically, the search spaces, designated as $\mathbf{S}_n$, comprise sets of bit strings $\{0, 1\}^n$ of fixed length or the set of all possible the permutations over the set $\{1, 2, ..., n\}$.

When considering the search space $\mathbf{S}$, it is necessary to define some notion of *neighborhood $N$*, which is a relation $N \subseteq \mathbf{S} \times \mathbf{S}$. A function

$$N : \mathbf{S} \to \mathscr{P}(\mathbf{S}) \tag{9.8}$$

refers to the neighborhood of a search point $s \in \mathbf{S}$, expressed as

$$N(s) = \{s' \in \mathbf{S} \mid (s, s') \in N\}. \tag{9.9}$$

Simulated annealing is considered efficient if it can locate a global maximum of $f$ at sufficiently high probability, and at the same time use fewer number of steps.

SA is a widely used heuristic to NP-complete problems that appear in real life from job-shop scheduling to groundwater remediation design.

Most analysis of search algorithms usually focuses on the worst-case situation. There are relatively few discussions of the average performance of heuristic

algorithms, because the analysis is usually more difficult and the nature of the appropriate average to study is not always clear. However, as the size of optimization problems increases, the worst-case analysis of a problem will become increasingly irrelevant, and the average performance of algorithms will dominate the analysis of practical applications. This large number domain of statistical mechanics, and hence of simulated annealing.

## 9.8 Genetic Algorithms

The Genetic Algorithms (GAs) are search procedures based on the process of *natural selection* and *genetics*. These are increasingly used in applications in difficult search problems, optimization, and machine-learning, across a wide spectrum of human endeavor. A GA processes a finite population of fixed-length binary strings. In practice, the strings are: bit codes, $k$-ary codes, real (floating-point) codes, permutation (order) codes, etc. Each of these has their place, but here we examine a simple GA to better understand basic mechanics and principles [3].

A simple GA consists of three operators: *selection, crossover, and mutation*. Selection is the survival of the fittest within the GA. Figure 9.7 shows the sequence of operations on a population $P_n$ and producing next population $P_{n+1}$. To understand the operation of the genetic algorithm, a population of three individuals is taken. Each is assigned a fitness value by the function $F$, called *fitness function*. On the basis of these fitnesses, the selection phase assigns the first individual (00111) zero copies, the second (111000) two, and the third (01010) one copy. After selection, the genetic operators are applied probabilistically; the first individual has its first bit mutated from a 1 to a 0, and crossover combines the next two individuals into two new ones. The resulting population is shown in the box labeled $T_{n+1}$. Algorithm 9.5 shows the steps for search using GA.
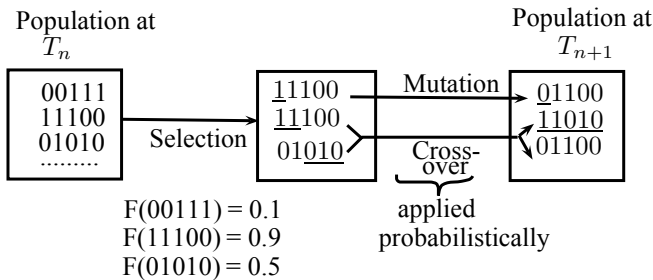


**Fig. 9.7** Sequence of operations in GA

---

**Algorithm 9.5** Genetic Algorithm(Input: Initial Population, fitness function, percent for mutation, selection threshhold)

---

1: **Initialize** the population with random candidate solutions
2: Apply fitness function to **Evaluate** each candidate's fitness value
3: **repeat**
4:    **Select** parents based on fitness value
5:    **Recombine** pairs of parents (crossover)
6:    **Mutate** resulting offspring
7:    Apply fitness function to **Evaluate** new candidates' fitness value
8: **until** termination condition/goal is reached

---

There are many ways to achieve effective selection, including ranking, tournament, and proportionate schemes, but the key notion is to give preference to better individuals. Consider that individuals are strings of fixed length. In a selection game of two-party of such individuals, pairs of strings are drawn randomly from the parental (original) population, and the better/fitting individuals places an identical copy into the mating pool. When the whole population is selected in this manner, every individual will participate in two tournaments; the best individuals in the population will win both trials. The median individual will typically win one trial and those worst, do not win at all.

For the selection to function, there must be some way of determining who is fitter individual. This evaluation can come directly from the formal objective function, or from the subjective judgment by a human observer. The ordering used is usually partial ordering.

The population holds representations of possible solutions. It usually has a fixed size and is a multi-set. Selection operators usually take whole population into account, i.e., reproductive probabilities are relative to current generation and diversity of a population refers to the number of different fitnesses.

If we were to do nothing but selection, GAs would not be very interesting because the trajectory of populations could contain nothing but changing proportions of strings contained in the original population. In fact, if run repeatedly, selection alone is a fairly expensive way of—with high probability-filling a population with the best structure of the initial population [4].

## 9.8.1  Exploring Different Structures

To do something more sensible, the GA needs to explore different structures. The main operator used in GAs is *crossover*, which can be one-point or multi-point crossover. A simple one-point crossover is performed using these three steps:

1. two individuals structures are chosen from the population using selection operator, and considered for mating,

2. a crossover site along the string length is chosen uniformly at random, and,
3. the values following the crossover site are exchanged between the two strings.

Let the two strings be, $A = 00000$ and $B = 11111$. If the random choice of a cross site turns up at 2, the two new strings following the crossover will be $A' = 00111$ and $B' = 11000$. These resulting strings are placed in the new population pool, and the process continues pair-by-pair from original population, until the new population is completely filled with "off-springs" constructed from the bits and pieces of good (selected) parents [5].

### 9.8.2 Process of Innovation in Human

Note that, the *selection* and *crossover* are simple operators, which do the job of: generating random number, copying string, and exchange of partial strings. However, their combined action makes much of the genetic algorithm's search ability. To understand this, we need to think the processing required to be done by human (us) when we innovate. Often we combine the notions that worked well in one context, with those that worked well in another context, to generate possibly better ideas (new notions) of how to attack the problem at hand. In similar way, the GAs juxtapose many different, highly fit substrings (called as notions) through the combined actions of selection and crossover to form new strings (can be called as ideas).

### 9.8.3 Mutation Operator

If selection and crossover provide much of the innovative capability of a GA, what is the role of the mutation operator? In a binary-coded GA, mutation is the occasional (low-probability) alteration of a bit position, and with other codes a variety of diversity generating operators may be used. By itself, mutation induces a simple random walk through string space. When used with selection alone, the combination form a parallel, noise-tolerant hill-climbing algorithm. When used together with selection and crossover, mutation acts as both insurance policy and as a hill-climber.

### 9.8.4 GA Applications

The simplest GAs are discrete, nonlinear, stochastic, highly dimensional algorithms operating on problems of infinite varieties. Due to this, GAs are hard to design and analyze [5].

The nature of problems GAs can solve are:

- GAs can solve hard problems quickly and reliably,
- GAs are easy to interface to existing simulations and models,
- GAs are extensible, and
- GAs are easy to hybridize.

Because GAs use very little problem-specific information, they are remarkably easy to connect to extant application code. Many algorithms require high degree of interconnection between the solver and the objective function. For example, dynamic programming requires a stagewise decomposition of the problem that not only limit its applicability, but also can require massive rearrangement of system models and objective functions. GAs on the other hand have clean interface, requiring no more than the ability to propose a solution and receive its evaluation.

Although there are many problems for which the genetic algorithm can evolve a good solution in reasonable time. There are also problems for which they are not suitable, such as problems in which it is important to find the exact global optimum. The domains for which one is likely to choose an adaptive method such as the genetic algorithm are precisely those about which we typically have little analytical knowledge, they are complex, noisy, or dynamic (changing over time). These characteristics make it virtually impossible to predict with certainty how well a particular algorithm will perform on a particular problem, especially if the algorithm is nondeterministic, as is the case with the genetic algorithm. In spite of this difficulty, there are fairly extensive theories about how and why genetic algorithms work in idealized settings.

**Example 9.4**  4-Queen Puzzle.

Suppose we choose to solve the problem for $N = 4$. This means that the board size is $4^2 = 16$, and the number of queens we can fit inside the board without crossing each other is 4. A configuration of 4 queens can be represented as shown in Fig. 9.8, using 4-digit string made of decimal numbers in the range 1–4. Each digit in a string represents the position of queen in that column. Thus, all queens in the left-to-right diagonal will be represented by a string 1234.

To solve this problem we take initial populations as [1234, 2342, 4312, 3431]. Let us recombine by randomly choosing the crossover after digit position 2. We recombine 1, 2 and 3, 4 members of population, producing [1242, 2334, 4331, 3412]. When this is combined with the original population, we get [1234, 2342, 4312, 3431, 1242, 2334, 4331, 3412]. Next, a random mutation is applied on members 3431 and 2334, changing the third digit gives 3421, 2324. Thus new population is, [1234, 2342, 4312, 3421, 1242, 2324, 4331, 3412].

The fitness of a string is proportional to the inverse of number of queens giving check in each string. for example, in the configuration in Fig. 9.8, total number of checks of $Q_1 \ldots Q_4$ are: $2 + 3 + 2 + 1 = 8$. Similarly in configuration 1234, total number of checks are 12.

This is because each queen is crossing the remaining three. These numbers in the remaining seven configurations are: 8, 4, 4, 4, 6, 8, 8. Thus fitness functions of

**Fig. 9.8** 4-Queens' board configuration
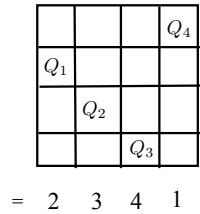


above population of elements are $[\frac{1}{12}, \frac{1}{8}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{6}, \frac{1}{8}, \frac{1}{8}]$. Thus, if we need to keep a population of size 4, of more fitter members their fitness functions are $[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{6}]$. These further combine next time, with population size of 4, having population of [4312, 3421, 1242, 2324]. This sequence can go on until goal is found in one of the configuration. $\qquad\square$

## 9.9 Summary

The domain of combinatorial optimization consists of a set of problems, which requires development of efficient techniques for finding the minimum or maximum of a function having many independent variables. This function is called *cost function* or *objective function*, and represents a quantitative measure of the "goodness" of some complex system. Because these combinatorial class of problems contain many situations of practical importance, *heuristic* methods have been developed, which require computations proportional to only a small polynomial of $n$, where $n$ is size of the problem. These heuristics cannot be generalized, and are unfortunately problem-specific.

Two basic strategies are common for heuristics: "divide-and-conquer" and "iterative improvement". The first approach divides/splits the problem into subproblems of manageable sizes, then solves each subproblem, and finally the sub-solutions are patched back together, to get the desired solution.

In iterative improvement-based approach, the heuristics starts with the system in a known configuration, and then, a standard rearrangement operation is applied to all parts of the system in turn, until a rearranged configuration that improves the cost function is discovered. The rearranged configuration then becomes the new configuration of the system, and the process is repeated until no further improvements are found.

These methods depend on some heuristics determined by the nature of the problem. The heuristics is defined in the form of a function, say $f$, which somehow represents the mapping to the total distance between start node and the goal node. For any given node $n$, the total distance between start and goal node is $f(n)$, such that

$$f(n) = g(n) + h(n),  \qquad\qquad (9.10)$$

where $g(n)$ is distance between start node and the node $n$, and $h(n)$ is the distance between node $n$ and the goal node.

One of the heuristic methods is *hill-climbing*. The name comes from the fact that to reach the top of a hill, one selects the steepest path at every node, out of the number of alternatives available. A Hill-climbing method is called *greedy local search*.

Among all the heuristic-based problem-solving strategies, informed *best-first* search strategy is one of the most popular methods to exploit heuristic information to cut down the search time. This method assesses exploration along the direction of highest merit, using heuristic information.

Several shortcuts help to simplify the computation of best-first search. First, if the evaluation function $f(n)$ used for node selection always provides optimistic estimates of the final costs of the candidate paths evaluated, then we can terminate the search as soon as the first goal node is selected for expansion, without compromising the optimality of the solution used. This guarantee is called *admissibility*.

We are often able to purge from search tree $T$, large sets of paths that are recognized at an early stage to be dominated by other paths in $T$. This becomes possible if the evaluation function $f$ is *order-preserving*, that is, if for any two paths $p_1$ and $p_2$, leading from start node $S$ to $n$, and for any common extension $p_3$ of those paths, the following holds:

$$f(p_1) \geq f(p_2) \Rightarrow f(p_1 p_3) \geq f(p_2 p_3)  \qquad\qquad (9.11)$$

The most studied version of best-first-search is the algorithm $A^*$, which provides *additive cost measures*, that is, where the cost of a path is defined as the sum of the costs of its arcs. This algorithm uses ordered state-space search and estimated heuristic to a goal state $f^*$, like the best-first search.[3] But, $f^*$ is unique in the sense that it can guarantee an optimal path to goal.

Based on values returned from two components: $g^*(n)$ and $h^*(n)$, the evaluation function $f^*(n)$ estimates the quality of a solution path through node $n$. The one component, i.e., $g^*(n)$, is the minimal cost of a path from start node to node $n$, and $h^*(n)$ is a lower bound on the minimal cost of a solution path from node $n$ to a goal node.

There are other search techniques based on "natural phenomena". Under this there are two techniques: (1) *Simulated annealing*, which is based on changes in the properties of metals and alloys due to heating them to higher temperature and then slowly decreasing their temperature; and (2) Something based on the Darwin's theory of evolution, called *genetic algorithms*. Annealing is process of treatment of metal or alloy by heating to a predetermined temperature, holding for a certain time, and then cooling to room temperature to improve ductility and reduce brittleness. The process annealing is carried out intermittently during the working of a piece of a metal to restore ductility lost through repeated hammering or other working.

---

[3] $f^*$ is also known as evaluation function.

**Table 9.1** Heuristic search methods

| S.No. | Search algorithm | Properties |
|---|---|---|
| 1. | Best-First Search | It depends on definition of $f(n)$. If $f(n) = h(n)$, then it is likely not optimal, and potentially incomplete. But, $A^*$ is a type of best-first search, which is complete and optimal. It is due to its choice of $f(n)$ that combines $g(n)$ and $h(n)$ |
| 2. | Hill-Climbing | It is Non-optimal, Incomplete like DFS, follows heuristics |
| 3. | Beam Search | It is like BFS, expand nodes in $f(n)$ order, and Incomplete for small $k$ ($k$ best nodes from successors are considered for further expansion). But, Complete, and like BFS for $k =$ infinity. It is Non-optimal. When $k = 1$, Beam search is analogous to Hill-Climbing method without backtracking |
| 4. | Branch and Bound | It is Optimal, and $g(n)$ is the cost of path from $s$ to node $n$ and $f(n) = g(n) + 0$ |
| 5. | Simulated annealing | Escapes local optima, and is complete and optimal given a long enough cooling schedule |

The Genetic Algorithms (GAs) are search procedures based on natural selection and genetics. A GA processes a finite population of fixed-length binary strings. In practice, all these are bit codes, *kry*-codes real (floating-point) codes, permutation (order) codes, etc.

A simple GA consists of three operators: *selection, crossover, and mutation*. Selection is the survival of the fittest within the GA. Each member of population is assigned a fitness value. On the basis of these fitnesses, the selection phase assigned zero or more copies of each individual. After selection, the genetic operators of crossover and mutation are applied probabilistically to the current population to produce new population.

The special features of GAs are as follows:

- GAs can solve hard problems quickly and reliably,
- GAs are easy to interface to existing simulations and models,
- GAs are extensible, and
- GAs are easy to hybridize.

Table 9.1 gives comparison of various heuristic methods:

## Exercises

1. Answer the following short review questions.

   a. In what condition the best-first search becomes the breadth-first?
   b. What can you infer from the condition: $f(n) = g(n)$?
   c. What can you infer from the condition: $f(n) = h(n)$?

**Fig. 9.9**  State-space search

Start

| 2 | 3 |
|---|---|
| 1 |   |

Goal

| 1 | 2 |
|---|---|
|   | 3 |
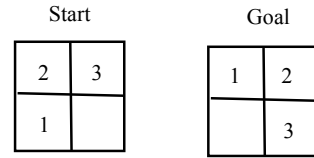
    d. In what situation the $A^*$ search become best-first search?

    e. What is the primary drawback of best-first search?

    f. Which search method(s) use the priority-queue data structure?

2. Consider the 3-puzzle problem, where the board is $2 \times 2$ and there are three tiles, numbered 1, 2, and 3, and *blank* tile. There are four operators, which move the *blank* tile up, down, left, and right. The start and goal states are given in Fig. 9.9. Show, how the path to the goal can be found using

    a. Breath-first search.

    b. Depth-first search.

    c. $A^*$ search having $g(n)$ equal to number of moves from start state, and $h(n)$ is number of misplaced tiles.

Assume that there is no possibility to remember states that have been visited earlier. Also, use the given operators in the given order unless the search method defines otherwise. Label each visited node with a number indicating the order in which they are visited. If a search method does not find a solution, explain why this happened.

3. Explain what algorithms or heuristics are suitable for solving constraint satisfaction problems under the following situations. Justify your answers.

    a. The problem is so tightly constrained that it is highly unlikely that solutions exist.

    b. The domain sizes vary significantly: some variables have very large domains (over 1,000 values) and some have very small domains (with fewer than 10 values).

    c. *Eight-Queens Problem*: Arrange eight queens on a chess board in such a manner that none of them can attack any of the others. (Note: A queen will attack another queen if it is crossing other queen while moving horizontally, vertically, or diagonally).

    d. The set of variables and set of domains are handled by a computer say, $M$. Each constraint is handled by a networked computer, say $N$. Traffic in the networks is slow. To check a particular constraint, computer $M$ sends a message to computer $N$ through the network, which in turn will send a message back to indicate whether the constraint is satisfied or violated.

4. Suggest a heuristic function for the 8-puzzle that sometimes overestimates, and show how it can lead to a suboptimal solution on a particular case.

5. Prove that, if the heuristic function $h$ never overestimates by more than a constant cost $c$, then algorithm $A^*$ making use of $h$ returns a solution whose cost exceeds that of the optimal solution by no more than $c$.

6. Give the name of the algorithms that results from each of the following special cases:

   a. Local beam search with $k = 1$.
   b. Local beam search with $k = \infty$.
   c. Simulated annealing with $T = 0$ at all times.
   d. Genetic algorithm with population size $N = 1$.

7. Explain, how will you use best-first search in each of the following cases? Give the data structure and explain logic.

   a. Speech recognition
   b. PCB design
   c. Routing telephone traffic
   d. Routing Internet traffic
   e. Scene analysis
   f. Mechanical theorem proving

8. What type of data structure is suitable for implementing best-first search, such that each node in the frontier is directly accessible, and all the vertices behind it remain in the order they have been visited.

9. Answer the following in one sentence/one word.

   a. How will you detect during the search of a graph that a particular node has been already visited?
   b. Is the best-first search optimal?
   c. Is the best-first search order-preserving?
   d. Is the best-first search admissible?

10. In the Traveling Salesperson Problem (TSP) one is given a fully connected, weighted, undirected graph and is required to find the Hamiltonian cycle (a cycle visiting all of the nodes in the graph exactly once) that has the least total weight.

    a. Outline how hill-climbing search could be used to solve TSP.
    b. How good results would you expect hill-climbing to attain?
    c. Can other local search algorithms be used to solve TSP?

11. Show that if a heuristic is consistent, then it can never overestimate the cost to reach the goal state. In other words, if a heuristic is monotonic, then it is admissible.

12. Suggest an admissible heuristic that is not consistent.

13. Can GAs have Local maximas? If it is not, how does the GAs tries to avoid it? If yes, justify it.

14. Explain different data structures that can be used to implement the *open* list in BFS, DFS, Best-first search?
15. Find out the worst-case memory requirements for best-first search.
16. If there is no solution, will $A^*$ explore the whole graph? Justify.
17. Define and describe the following terms related to heuristics:
    Admissibility, monotonicity, informedness.
18. Show that:

    a. The $A^*$ will terminate ultimately.
    b. During the execution of the $A^*$ algorithm, there is always a node in the open-list, that lies on the path to the goal.
    c. If there exits a path to goal, the algorithm $A^*$ will terminate by finding the path of to goal.
    d. If there is no solution in $A^*$, the algorithm will explore the whole graph.

19. Discuss the ways using which $h$ function in $f(n) = g(n) + h(n)$ can be improved during the search.
20. Why must the $A^*$ algorithm work properly on a graph search, with graph having cycles?
21. For the graph shown in Fig. 9.5, find out whether the $A^*$ search for this graph is

    a. Optimal?
    b. Oder-preserving?
    c. Complete?
    d. Sound?

22. Apply the BFS algorithm for robot path planing in the presence of obstacles for a square matrix of $8 \times 8$ given in Fig. 9.10. Write an algorithm to generate the frontier paths. Assume that each move of robot in horizontal (H) and vertical (V) covers a unit distance, and the robot can take only the H and V moves. The start and goal nodes are marked as $S$ and $G$. Shaded tiles indicate obstacles, i.e., robot cannot pass through these.
23. Redesign the problem of robot shown in Fig. 9.10 for $A^*$ search. Assume that value of $h$ is number of squares equal to $V - i + H - j$, where V and H are both 8.

**Fig. 9.10**  $8 \times 8$ tiles, with obstacles in shades
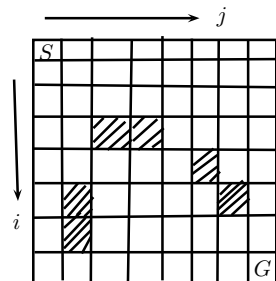
24. Solve the 8-puzzle manually for 20-steps, where heuristic is number of tiles out of place with respect to goal state. Assume that $f^*(n) = g^*(n) + h^*(n) = 0 + h^*(n) = h^*(n)$, so that only the heuristics is deciding factor for next node. Note that algorithm shall be DFS. In case of ties, give preference to those nodes which are to the left of the search tree.

25. Find an appropriate state-space representation for the following problems. Also, suggest suitable heuristics for each.

  a. Cryptarithmetic problems (e.g., TWO + TWO = FOUR)
  b. Towers of Hanoi
  c. Farmer, Fox, Goose, and Grain.

26. Suggest appropriate heuristics for each of the following problems:

  a. Theorem proving using resolution method
  b. Blocks world

27. If $P$ = "heuristic is consistent", and $Q$ = "heuristic is admissible". Then show that $P \Rightarrow Q$. Demonstrate by counter example that $Q \nRightarrow Q$.

28. Consider the magic-puzzle shown in Fig. 9.11. Suggest the formalism for searching the goal state when started from the start state. (Note that in the goal state all the rows, columns, and diagonals have equal sums equal to 15).

29. Make use of GA to solve 4-puzzle (Fig. 9.12). A move consists, sliding of either of tiles 1 or 2, or 3 into the blank tile. Such a movement creates blank tile at a different position, and the process is repeated until goal state is reached. The solution requires not only reaching to the goal state, but also finds the trace path to reach the goal. Construct a suitable fitness function to implement search by GA, the search should consider only those members of the population which correspond to valid moves.

30. For the graph shown in Fig. 9.13, make use of DFS and certain depth cut-off to backtrack the search from that cut-off.

31. Use best-first search for Fig. 9.14 to find out if the search from start node $A$ to goal node $G$ is

**Fig. 9.11** Magic-puzzle

| 1 | 2 | 3 |
|---|---|---|
| 4 | 5 | 6 |
| 7 | 8 | 9 |

Start state

| 6 | 7 | 2 |
|---|---|---|
| 1 | 5 | 9 |
| 8 | 3 | 4 |

Final state

**Fig. 9.12** 4-puzzle

| 1 | 2 |
|---|---|
| 3 |   |

Start state

|   | 1 |
|---|---|
| 2 | 3 |
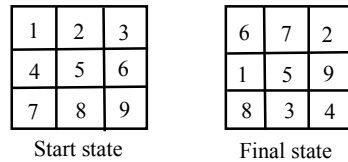
Goal state
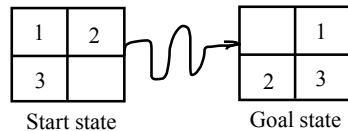
**Fig. 9.13** A graph with start node $S$ goal $G$
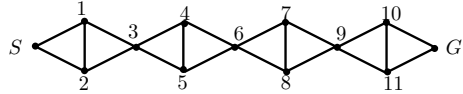
**Fig. 9.14** Graph with start node $A$ and goal node $G$
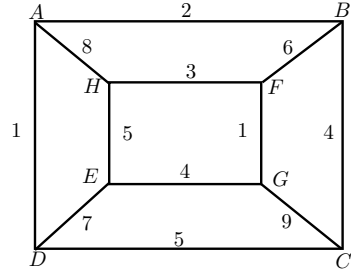
a. Oder-preserving
b. Admissible
c. Optimal

32. How you will apply the simulated annealing in the following scenarios? For each case, give the problem formation so as to compute the $\Delta E$, temperature $T$, the states $s$, $s'$; the state space $\mathbf{S}$, and object function $f : \mathbf{S} \rightarrow \mathbb{R}$, and perform 5–10 iterations steps manually.

    a. 8-puzzle
    b. 8-Queen problem
    c. Tic-tac-toe problem

33. One fine morning you find that your laptop is not booting. There can be enumerable reasons for this. Assume that you are expert in installation and maintenance of laptops. Represent the search process for trouble-shooting the laptop by constructing a search tree.

    a. Suggest, what search method you consider as most appropriate for this? Also, explain the justification of the particular method you have chosen?
    b. What heuristics you would like to suggest for making the search efficient?
    c. What are the characteristics of this search? Comment for admissibility, monotonocity, and completeness of this solution.

34. Assume a population of 10 members, and apply GA to find out the solution by performing five cycles of iterations, each having selection, mutation, and crossover. Verify that we are far closer to the solution than we were in the begin after performing these iterations. Represent the members as bit strings $\{0, 1\}^n$ for some integer $n$. Also, fix up some criteria for fitness function, as well as the probability of mutation, and point of crossover.

   a. 8-puzzle
   b. square-root of a number
   c. Factors of an integer

35. What are the consequences of the following special cases of GA-based search?

   a. Only the selection operation is performed in each iteration, based on the fitness value.
   b. Only the crossover operation is performed in each iteration at a random position.
   c. Only the mutation operation is performed in each iteration at a random bit position.

36. Simulated annealing is guided by a changing "temperature" value that determines the likelihood of visiting nodes that appear to be worse than the current node. How does the search behave for very low and very high temperature values, and why it behaves so?

37. Select the best alternatives in each of the following questions.

   i. The mutation operation is good for the following:
      (a) noise tolerance (b) hill-climbing
      (c) random walk    (d) all above
   ii. The following operation of GA has maximum contribution to search:
      (a) mutation  (b) selection
      (c) crossover (d) fitness function
   iii. What operation of GA is responsible for random walk?
      (a) mutation     (b) crossover
      (c) none above (d) both a and b
   iv. GAs are not good for the following purpose:
      (a) finding exact global optimum (b) local search
      (c) approximate solution            (d) global search
   v. GAs are good in environments which are :
      (a) complex (b) noisy
      (c) dynamic (d) all above
   vi. GA is always:
      (a) **P**    (b) nondeterministic
      (c) **NP** (d) deterministic

# References

1. Bagchi A, Mahanti A (1985) Three approaches to heuristic search in networks. J ACM 32: I:1–27
2. Dechter R, Pearl J (1985) Generalized Best-First Search strategies and the optimality of $A^*$. J ACM 32(3):505–536
3. Forrest S (1993) Genetic algorithms: principles of natural selection applied to computation. Science 261:872–878

4. Goldberg DE (1989) Genetic algorithms in search, optimization and machine learning. Addison-Wesley, Reading
5. Goldberg DE (1994) Genetic and evolutionary algorithms come of age. Commun ACM 37(3):113–119
6. Hart PE et al (1968) A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern 100–107
7. Kirkpatrick S et al (1983) Optimization by simulated annealing. Science 220(4598):671–680
8. Korf RE et al (2005) Frontier search. J ACM 52(5):715–748

# Chapter 10
# Constraint Satisfaction Problems

**Abstract** Constraint Satisfaction Problems (CSPs) is a theory about some special type of problems, where every move of search is subject to fulfillment of certain constraints. The chapter introduces with such problems in the beginning, then presents a formal general model of such problems with analysis, explains the solution approach with the synthesis of constraints using simple, as well extended theory of synthesis. Next, it presents the classes of CSP algorithms—generate and test, backtracking, discusses how efficiency can be increased, propagation of constraints, followed with cryptarithmetics, chapter summary, and then at the end a list of exercises for practicing.

**Keywords** Constraint satisfaction problems (CSP) · Generate and test · Backtracking · CSP representation · Constraints · Synthesizing constraints · Theory of synthesis · Cryptarithmetics

## 10.1 Introduction

This chapter presents the theory of some special type of problems, called, Constraint Satisfaction Problems (CSP), where one need to search the state space, but every move is subject to the fulfillment of certain constraints, which is different from the selection of best fitting state in best-first search or in hill-climbing. Many combinatorial search problems can also be expressed in the form of CSPs, where the aim is to find an assignment of values to a given set of variables, subject to some specified conditions (constraints). For example, the *SAT* (the satisfiability) problem may be viewed as a CSP where variables are assigned only to the *Boolean* values, such that it satisfies the constraints given in the form of clauses.

A general case of a constraint satisfaction problem is known to be *NP-hard*. However, a restricted version of the problem that can be obtained as tractable by imposing restrictions on the form of constraint interconnections [7].

In an algorithm for the solution of CSP, the constraints imposed on the previous states also propagates to the next states. If there is a violation of constraints, we need to backtrack and try other branches in the tree constructed. The constraints only

specify the relationships without specifying a computational procedure to enforce that relationship. Ultimately, the algorithm has to find a solution to the specified problem.

Consider the problem of preparing a time-table for classes in a department, where there are courses, teachers, classrooms, periods, and days of the week. The teachers are assigned subjects, rooms are assigned classes, the time-slots in days and weeks (periods) are assigned the classes, teachers, and subjects, etc. It should be done to satisfy many things, like, all the teachers should get the courses of their choice (as far as possible), there is uniform distribution of teaching load among the teachers, no teacher should be assigned two classes at the same time in a day, neither it should happen with the room, or even two teachers teaching a class at the same time! This is an example of a constraint satisfaction Problem.

It is usually common, to encounter many similar problems to be solved, in job scheduling in project completion, supply chain management, CPU job scheduling, and other optimization problems.

A CSP is a mathematical problem defined in the form of a set of objects, whose collective state during the solution must satisfy a number of constraints or restrictions. A CSP represents entities in a problem in the form of a homogeneous collection of finite constraints over some variables, which is solved by methods called CSP methods. These problems are presently the subject of intense research, in both the fields of Operations Research (OR) and Artificial Intelligence (AI), because the regularity in their formulation provides a common basis to analyze and solve problems of many unrelated fields. The CSPs often exhibit high complexity, requires heuristics and combinatorial search together to solve such problems in a realistic amount to times. Examples of such problems are Satisfiability Modulo Theories (SMT), Answer Set Programming (ASP), and Boolean Satisfiability problem (SAT) [2].

**Learning Outcomes of this Chapter**:

1. Formulate a problem specified in natural language (e.g., English) as a constraint satisfaction problem and implement it using a backtracking algorithm or stochastic local search. [Usage]
2. Order of selection variables and their values for CST. [Familiarity]
3. Complexity issues of CST. [Usage]

## 10.2   CSP Applications

Application of CSPs range from database queries, e.g., find all $x, y, z$ such that $x$, $y$ are components, $z$ is assembly, and failure of $x$ causes the failure of $y$, to scene analysis where it is required to analyze the scene by segmenting it, for example, sky region is labeled blue, vegetation regions are green, and cars are totally surrounded by either grass or sky. Following are some applications of CSPs:

- Computer Vision (Interpreting objects in 3*D* scenes): Scene labeling.
- Solid Modeling: constrained-based design, beautification.
- Advanced Planning and Scheduling: activity scheduling, scheduling production, airline scheduling.
- Assignment problems: stand allocation for Aircrafts, balancing work among different persons, who teaches what class?
- Electrical engineering: fault location, circuit layout computation.
- Network Management and Configuration: planning of cabling of telecommunication networks, network reconfiguration without service interruptions.
- Molecular Biology: chemical hypothesis reasoning, protein docking, DNA sequencing.
- Database Systems: Ensure and/or restore data consistency.
- Cryptography.

We consider a classical graph coloring problem. This problem requires to assign colors to the vertices of a graph in such a way that if any two vertices are joined by an edge in the graph, they will have a different color. The variables of this constraint problem are the nodes, constants (values) are colors in a set, and the constraints are the requirements that neighboring vertices do not have the same color. For our purposes, we will restrict the problem to have a finite set of permissible colors.

**Example 10.1** Map Coloring Problem.

Consider the case of boundary relations between states of Indian territory, where, Madhya Pradesh (MP) is connected to Uttar Pradesh (UP), Rajasthan (RAJ), Gujrat (GUJ), Maharastra (MAH), and Chattisgarh (CHG), all surrounded to MP as shown in Fig. 10.1.

Assuming that we have been given a task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions: *MP*, *RAJ*, *GUJ*, *MAH*, *CHG*, and *UP*. The domain of each variable is the set {*red*, *green*, *blue*}. The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for *RAJ* and *GUJ* is a set of pairs:

**Fig. 10.1** Cities connected as directed graph
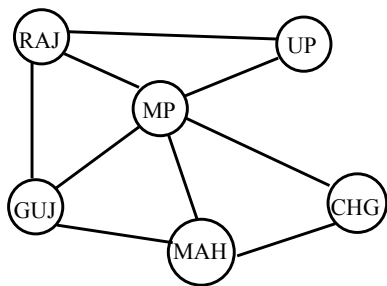
$$\{(red, green), (red, blue), (green, red),$$
$$(green, blue), (blue, red), (blue, green)\}$$

A constraint can also be represented as an inequality $RAJ \neq GUJ$, provided that the constraint satisfaction algorithm has some way to evaluate such expressions. There is a solution for this problem as: $\{RAJ = red, GUJ = green, MAH = red, CHG = green, UP = red, MP = blue\}$. However, there are many other possible solutions. $\qquad\square$

## 10.3  Representation of CSP

Many problems, especially in AI, can be represented using constraint satisfaction methods, in a declarative way by identifying variables of interest for the problem, in a well-defined domain, and assignment of variables is restricted by constraints. Formally, a constraints' network $R$ consists of a finite set of variables $V$, and a set $C$ of constraints. Thus, a constraint satisfaction problem is defined as a triple,

$$\langle V, D, C \rangle \qquad (10.1)$$

where,

$V = \{v_1, v_2, \ldots, v_n\}$, is a finite set of variables;
$D = \{D_1, D_2, \ldots, D_n\}$ is a finite set (domains) of values (which may be finite or infinite); and
$C = \{C_1, C_2, \ldots, C_n\}$ is a finite set of constraints.

Associated with each variable $v$ is a finite, discrete domain $D_v$. A constraint $c$ on the variable set $V_k \subseteq V$ in its extensional form, is a subset of the Cartesian-product of the domains of the afflicted variables. The expression $var(c)$ denotes the tuple of variables on which the constraint is defined. A relation, $rel(c) \subseteq X_v D_v$, and $v \in var(c)$ are the relational information of the constraint $c$. The assignment of a value $d \in D_v$ to a variable $v$ is denoted by $v := d$. A tuple $t$ of a assignments of variables $V_k \subseteq V$ satisfies a constraint $c$, if and only if $t[var(c)] \in rel(c)$.

Each constraint $c_i$ is a pair $(V_i, R_i)$ (usually represented as a matrix), where $V_i \subseteq V$ is a set of variables, called the constraint *scope*, and $R_i$ is a set of (total) functions from $V_i$ to $D$, called the constraint *relation* (an *n-ary* relation). Thus an evaluation of variables is evaluation of a function $f$ from the set of variables to the domain of values, $f : V \rightarrow D$. The evaluation function $f$ satisfies a constraint $\langle (v_1, \ldots, v_n), R \rangle$ if $(f(v_1), \ldots, f(v_n)) \in R$. A solution is an evaluation that satisfies all constraints.

Hence, given an instance of a constraint satisfaction problem, its solution is a function $f$ from the set of variables $(V)$ of that instance to the set of values (domain $D$), i.e., $f : V \rightarrow D$. This mapping is subject to each constraint $(V_i, R_i) \in C$, the restriction of $f$ to $V_i$, denoted $f \mid V_i$, is an element of $R_i$.

**Example 10.2** Simple CSP examples.

For a set of values in domain $D$ as a set of real numbers $\mathbb{R}$, there is a relation on some set of variables $\{u, v, w\}$, which is a set of total functions from $\{u, v, w\}$ to $\mathbb{R}$. Following is a typical relation of this type:

$$\{f : \{u, v, w\} \to \mathbb{R} \mid 5u + 7v + w = 0\}. \tag{10.2}$$

If we fix an ordering on the variables $(w, u, v)$, then the same relation can be represented as a set of 3-tuples of real numbers:

$$\{(a, b, c) \in \mathbb{R}^3 \mid a + 8b + 11c = 0\}. \tag{10.3}$$

$\square$

### 10.3.1 Constraints in CSP

A constraint can often be regarded as a mathematical equation, and the constraint network as a system of simultaneous equations. The solutions of simultaneous equations correspond to the propagation of constraints. However, the constraints are not always restricted to equations, they can also be expressed as *inequalities* and non-numeric relationships. The CSPs are, therefore, excellently suited for qualitative, as well as quantitative modeling of systems and physical relationships [8].

A constraint relation can be finite or infinite. A finite constraint relation can be represented by simply giving an explicit and elaborate list of all the elements in the relation. However, the infinite constraint's relation cannot be listed exhaustively. Alternatively, instead of listing the elements' mapping, the relations in both the cases can be expressed using suitable specification language, or linear equations, or formulas.

There can be many ways to provide the restrictions in mapping values to variables. Following are types of (restrictions) constraints:

1. *Unary Constraints*: Involves single variable, e.g., $MP \neq green$.
2. *Binary Constraints*: Constraints involve pair of variables, e.g., $MP \neq UP$.
3. *Higher order Constraints*: These contain the $\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}$ or more variables, e.g., cryptographic column constraints, and Professors $A, B, C$ cannot be in the same committee together.
4. *Preference or Soft constraints*: For example, the red color is better than green, cost for each variable assignment, etc.

The specifications in the constraint relations indicate the scope for allowed combinations of simultaneous values for the different variables. How many variables in a scope simultaneously satisfy the constraint, is called *arity* of the constraint. For example, *unary* constraint specify the allowed values for a single variable, while the *binary* constraint specify the allowed combinations of values for a pair of variables. In addition, there is an *empty constraint* (i.e., 0, 0), having scope of variables and relations both empty.

We represent constraints' relations as sets of *functions*, rather than sets of *tuples*. In fact, both the representations are equivalent, as by fixing the ordering of variables in the scope of constraints, we can associate the functions with the corresponding tuples. But, we shall prefer to use functional representations because it simplifies the definitions, as the following examples demonstrate.

**Example 10.3** CSP for predicate expression.

Let the variables take Boolean values, and the set of possible values for the variables is $\{T, F\}$. Given this the logical formula '$x_1 \wedge x_2 \wedge \neg x_3$'can be used to specify the constraint with scope $\{x_1, x_2, x_3\}$, such that the following constraint relation holds.

$$\{f : \{x_1, x_2, x_3\} \rightarrow \{T, F\} \mid f(x_1) \wedge f(x_2) \wedge \neg f(x_3) \equiv T\}. \qquad (10.4)$$

In a similar way, when the possible values is a set of real numbers $\mathbb{R}$, then the equation '$x_1 + 2x_2 = 7$'can be used to specify the constraint with scope $\{x_1, x_2\}$ and following constraint relation holds.

$$\{f : \{x_1, x_2\} \rightarrow \mathbb{R} \mid f(x_1 + 2x_2 = 7\}. \qquad (10.5)$$

$\square$

For the representation of constraints, such as the relationship between height and weight of a person, one of the following possibilities can be used:

- *Table*: as a lookup table.
- *Functions*: *Normal weight = height (in cm) − 100.*
- *Heuristics*: For example, the under weight can be specified as:

$$\frac{actual\ weight}{normal\ weight} < 0.8 \qquad (10.6)$$

Propagation algorithm may be distinguished according to what can be propagated through a variable. It can be classified as:

- Fixed Value quantity: For example, $C = 5$.
- Set of values, for example, $C \in \{3, 4, 6, 7\}$.
- Symbolic expression, for example, $C = 2y$.

For propagation of fixed value or set, a simple *forward chaining* is sufficient as a constraint strategy for rules. The application of symbolic expressions must be controlled heuristically.

## 10.3.2 Variables in CSP

A constraint satisfaction problem can be represented as a *constraint graph*, where the vertices correspond to the variables and the edges to the binary constraints. The constraints specify which pairs of values are allowed. If all pairs are allowed, we say that there is no constraint. If we place the vertices in a linear order, we obtain an *ordered constraint graph*. Such an ordering corresponds to the levels of a backtrack search tree at which the variables are instantiated. The order in which a backtrack process chooses variables to instantiate is termed the *vertical order* of search, corresponding to depth in the backtrack tree.

Discrete Variables

For a finite domain with $n$ variables, and domain size $|D|$, there are $O(|D|^n)$ total number of assignments. The example is Boolean Satisfiability problem (SAT), where there are $n$ variables $(v_1, \ldots, v_n)$, and these can be assigned in total $2^n$ ways; for 8-queens, the complexity is $O(8^{64})$. For infinite domains, the data are integers, characters, and strings. Examples of linear constraints are: $v_1 + v_2 + v_3 = 1050$, $0 \leq v_1 + v_2 + 2 * v_4 \leq 5000$, $v_1 - 3 * v_5 \geq 300$, etc.

Nonlinear constraint for minimization can be defined as follows: Minimize $f(x)$ subject to $g_i(x) \leq 0$ for each $i \in \{1, \ldots, m\}$, and $h_j(x) = 0$ for each $j \in \{1, \ldots, p\}$, and $x \in X$. A maximization problem can be defined in a similar way.

Considering a *CSP* of *job scheduling*, where variables are start and end days of job. They need a constraint language like, $startjob_i + 5 \leq endjob_j$. There are infinitely many solutions to this problem. If the constraints are *linear*, then it is solvable, else there is no general algorithm. Linear constraints are solvable in polynomial time [6].

Continuous Variables

Building an airline schedule or class schedule uses counter variables.

**Example 10.4** CSP Problem for an electric Circuit.

In an electrical circuit, $R$ is a constraint, given the voltages $V_1$ and $V_2$ (see Fig. 10.2).

**Fig. 10.2** Constraint in an electric Circuit
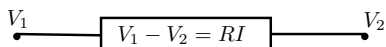
In an extension of the above logic, all the circuit components can be represented as constraints and connected by common variables (voltage and currents) to form a constraint network. If the voltage and currents of power source are known, the values are propagated through the constraint network in the same way as a real current, so that a value can be calculated for each variable.                                   □

In the following we will consider how restricting the allowed constraints affects the complexity of this decision problem.

**Definition 10.1**  Decision problem.

A CSP can also be considered as a decision problem as follows: For any set of constraints $CSP(\Gamma)$ for the solution of the problem represented as a set of formulas $\Gamma$, and with every,

*Instance*: is a finite set of constraints $C \subseteq \Gamma$ which are to be satisfied, then we ask,
*Question*: Does $C$ has a solution?
  If there exists some algorithm that solves every instance in this $CSP(\Gamma)$ in polynomial time, then we say that $CSP(\Gamma)$ is 'tractable', and refer to $C$ as a tractable set of constraints.                                   □

## 10.4   Solving a CSP

It is helpful to visualize a CSP as a constraint graph (network). The nodes of the graph correspond to variables of the problem and the edges correspond to constraints.
  The representation of states in CSP is in accordance with a standard pattern—a set of variables with assigned values, successor function, and goal test are written in generic way. In addition, one can develop effective and generic heuristics that require no additional domain-specific expertise.

It is fairly easy to see that a CSP can be as a standard search problem as follows:

- *Initial state*: It is an empty assignment {}, where all variables are unassigned.
- *Successor function*: A value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables, and does not violate the constraint.
- *Goal test*: Whether the current assignment is the complete assignment?
- *Path cost*: A constant cost (e.g., 1) for every step.

  All the CSPs have similar steps for the solution. Every solution appears at depth $n$ with $n$ variables, this indicates that it is a depth-first search approach. The Path is irrelevant, so it can also use complete-state formulation. For domain size $|D| = d$, the branching factor is,

$$b = (n - i)d \tag{10.7}$$

at depth $i$. Hence the total number of leaves are:

$$\prod_{i=0}^{n-1}(n-i)d^n = n!d^n. \tag{10.8}$$

### 10.4.1  Synthesizing the Constraints

A constraint expression is a conjunction of constraints. The constraints can be represented in the form of a network such that variables are represented as nodes, and each constraint is a link/arc/edge between two nodes. Further, the constraints are restricted as *unary* or *binary* constraints. A unary constraint is a predicate on single variable, and a binary constraint, a predicate on two variables. Consider a small constraint satisfaction problem of coloring a two vertex *complete graph*[1] using a single color, say red. This graph contains vertices $v_1, v_2$, and $(v_1, v_2)$ is the edge joining these vertices. The coloring problem here can be expressed as a CSP: $X_1, X_2$ are variables that represent the colors of nodes $v_1, v_2$, respectively. The binary constraint gives a restriction that $X_1$ and $X_2$ are not of the same color. Also, a unary constraint on both the $X_1$ and $X_2$ requires these nodes to be red. Let the initial domain set for $X_1$ and $X_2$ are colors available as {red, green}; having given this, the problem is represented as a constraint network, shown in figure below [5].

$$red \subset \{red\ green\}_1 \xrightarrow{\text{not-same-color}} \{red\ green\}_2 \supset red$$
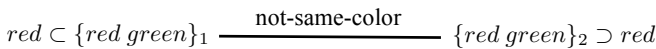
In the figure, the nodes identified as $\{red\ green\}_1$ and $\{red\ green\}_2$ contain possible values for variables $X_1$ and $X_2$. The loop symbol ($\subset$) at each node indicates that a unary constraint is applicable on each node, and the link "not-same-color", between nodes corresponds to binary constraint between them. The first level of inconsistency in constraint networks is *node inconsistency*. Note that, the potential domain values in this example are red and green, while the unary constraint for each node is red. Hence, we can immediately eliminate the green from both the nodes $X_1$ and $X_2$. On doing this elimination, we get the following network.

$$\subset \{red\}_1 \xrightarrow{\text{not-same-color}} \{red\}_2 \supset$$
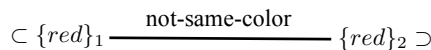
The second level of inconsistency is *arc inconsistency*. The arc between $X_1$ and $X_2$ is inconsistent because, for the value of "red" in $X_1$, it does not satisfy the constraint "red is not-same-color as red." To satisfy this constraint, we remove the color "red" from both the nodes. This cuts down the search space, but unfortunately, this case reflects the fact that the problem is impossible to solve. Hence, in this case, there is no global solution, and the network is "unsatisfiable."

---

[1]A *complete graph* is one in which all possible edges between pairs of vertices are present.

$\{red\ green\}_2$

$\{red\ green\}_1$ _____ $\{red\ green\}_3$
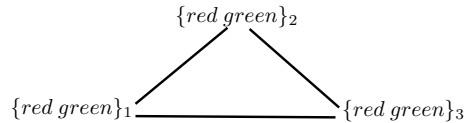
It is quite possible for a network to have no arc inconsistencies, but still, it can be unsatisfiable. Now, let us consider the problem of coloring a complete three-vertex graph with two colors, represented in Fig. 10.3.

Let us assume that a set of possible values of color for each variable is $\{red\ green\}$. The binary constraint between a pair of nodes is the same, "is not the same color as." This network is arc-consistent, e.g., given a value "red" for $X_1$, the $X_2$ may be assigned "green", hence "red is not the same as green" is satisfied. But, there is no way to choose single values $a_1$, $a_2$, $a_3$ for $X_1, X_2, X_3$, respectively, so that all the three binary constraints are simultaneously satisfied. If we choose red for $X_1$, the binary constraint forces us to choose green for $X_2$. Having chosen green for $X_2$, the binary constraint forces us to choose red for $X_3$, and that forces us to choose green for $X_1$, but for $X_1$, red color has been already chosen.

Nevertheless, it may be helpful to remove arc inconsistencies from a network. This involves comparing nodes with their neighbors as we did above. Each node must be so compared; however, comparisons can cause changes (deletions) in the network and so the comparisons must be iterated until a stable network is reached. These iterations can propagate constraints some distance through the network. The comparisons at each node can theoretically be performed in parallel and this parallel pass iterated [3].

Thus, removing arc inconsistencies involves several distinct ideas: local constraints are globally propagated through iteration of parallel local operations. It remains to be seen which aspects of this process are most significant to its application. The parallel possibilities may prove to be particularly important.

A more powerful notion of inconsistency is called as *path inconsistency*, where a network is path inconsistent if there are two nodes $X_1$ and $X_2$ such that '$a$' satisfies $X_1$, '$b$' satisfies $X_2$, $a$, and $b$ together satisfy the binary constraint between them, yet there is some other path through the network from $X_1$ to $X_2$, such that there is no set of values, one for each node along the path, which includes $a$ and $b$, and can simultaneously satisfy all constraints along the path. For example, the network in Fig. 10.3 is path inconsistent: red satisfies $X_1$, green $X_3$, red is not the same color as green: however, there is no value for $X_2$ which will satisfy the constraints between $X_1$ and $X_2$, and between $X_2$ and $X_3$, while $X_1$ is red, $X_3$ is green.

Consider the problem of coloring the complete four-vertex graph with three colors (Fig. 10.4). Each node contains red, green, and blue, and each arc again represents the constraint "is not the same color as." In particular, path consistency does not fully determine the set of values satisfying the global constraint, which in this inconsistent case is the empty set.

**Fig. 10.4**  Network with
path inconsistency
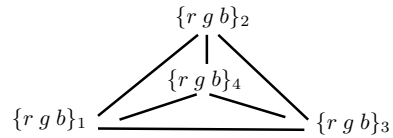


In conclusion, arc and path consistency algorithms may reduce the search space, but do not in general fully synthesize the global constraint. When there are multiple solutions, additional searches will be required to specify the several acceptable combinations of values. Even a unique solution may require further search to determine, and the consistency algorithms may even fail to reveal that no solutions at all exist.

### 10.4.2   An Extended Theory for Synthesizing

As the coloring problem suggests, the general problem of synthesizing the global constraint is NP-complete, and thus unlikely to have an efficient (polynomial time) solution. On the other hand, it is found that in specific applications it may be possible to greatly facilitate the search for solutions.

There are two key observations that have motivated this approach.

1. Node, arc, and path consistency in a constraint network for $n$ variables can be generalized to a concept of $k$-consistency for any $k \leq n$.
2. The given constraints can be represented by nodes, as opposed to links, in a constraint network; we can add nodes representing $k$-ary constraints to a constraint network for all $k \leq n$ (irrespective of whether a corresponding $k$-ary constraint is given or not). These constraints are then propagated in the augmented net to obtain higher levels of consistency.

Continuing in this manner by adding successively higher levels nodes to the network and propagating the constraints in the augmented net, one can get the $k$-ary consistency for all $k$, with maximum $k$ equal to a total number of nodes in the network, i.e., $n$. It is clear that constraints are not required to be restricted to binary constraints. Ensuring that there are no lower level inconsistencies, one moves progressively to achieve higher level consistencies. However, it may lead to a combinatorial explosion when we try to ensure that there are no inconsistencies when we progress to achieve higher level consistencies. The final result is a network where $n$-ary nodes specify explicitly, the $n$-ary constraints which we aim to synthesize, and from here, no further search is required.

**Example 10.5**  A Preliminary example based on Synthesis Algorithm.

We consider a crude example of the synthesis algorithm in operation, by way of motivation for the formal description which follows. Assume that following are
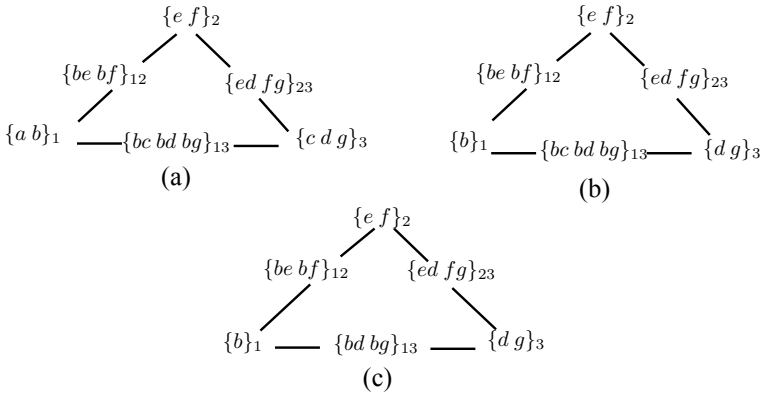
**Fig. 10.5** Applying synthesis algorithm

constraints given on variables $X_1$, $X_2$, $X_3$: a unary constraint $C_1$ specifies that $X_1$ must be either $a$ or $b$ ($C_1 = \{a\ b\}$). Similarly, for $X_2$, the $C_2 = \{e\ f\}$ and for $X_3$, the $C_3 = \{c\ d\ g\}$. The binary constraint between $X_1$ and $X_2$ states that: $X_1 = b$, $X_2 = e$, or $X_1 = b$, $X_2 = f$. Hence, binary constraint $C_{12} = \{be\ bf\}$. In similar way, $C_{13} = \{bc\ bd\ bg\}$ and $C_{23} = \{ed\ f\ g\}$.

We are interested to find out the choices for $X_1$, $X_2$, $X_3$, if any, that can simultaneously satisfy all these constraints. We begin building the constraint network with three nodes representing the unary constraints on the three variables. Next, we add nodes representing the binary constraints, and link them to the unary constraints (e.g. $\{be\ bf\}_{12}$ represents $C_{12}$. The combined figure is shown as Fig. 10.5a.

After we add and link node $C_{12}$ we look at node $C_1$ and find that element $a$ does not occur in any member of $C_{12}$. We delete $a$ from $C_1$. Similarly, we delete $c$ from $C_3$ after adding $C_{23}$. The constraint network now appears as in Fig. 10.5b.

Now from $C_3$ we look at $C_{13}$ and find that there is an element $bc$ in $C_{13}$ which requires $c$ as a value for $X_3$, while $c$ is no longer in $C_3$. We remove $bc$ from $C_{13}$, as shown in Fig. 10.5c. From this we note that the problem has two solutions as follows:

| $X_1$ | $X_2$ | $X_3$ |
|-------|-------|-------|
| $b$ | $e$ | $d$, |
| $b$ | $f$ | $g$. |

But, but the consistency of all the variables have not been simultaneously verified yet, which is covered using the concept of *augmented network*.                      □

Augmented Network

So far we have merely achieved a sort of "arc consistency" (though we indicate the restriction of the pair $bc$ (in $C_{13}$), as well as the elements $a$ and $c$).

**Fig. 10.6**  Augmented
network
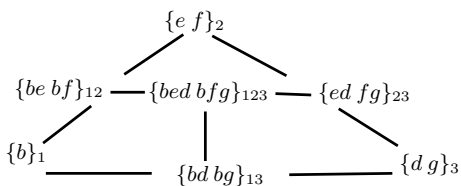


Next, we add a node for the *ternary* constraint (i.e., $k$-ary constant $k \leq n, k = 3$, $n = 3$). Note that, there was no order-three constraint given originally, hence we could assume initially of no constraint of order three. But, from the given binary and unary constraints, we could construct the ternary constraint.

Note that, $C_1$ and $C_{23}$ together allow only the {*bed bf g*} as set of triples. This is used as the ternary node and we link it to the binary nodes shown in Fig. 10.6, which makes an *augmented* network.

We look at the new node from its neighbors and vice versa, as we did earlier, to ensure consistency of the sort we obtained earlier between neighboring nodes. $C_{13}$ is consistent with the new node: *bd* is part of *bed*, *bg* part of *bf g*. Similarly $C_{12}$ and $C_{23}$ are consistent with the ternary node. If necessary, we could propagate deletions around until local consistency is achieved on this augmented network. However, in this case, the network is already stable; no further changes are required. The ternary node represents the synthesis of the given constraints. There are two ways to simultaneously satisfy the given constraints: $X_1 = b, X_2 = e, X_3 = d$; or $X_1 = b$, $X_2 = f, X_3 = g$.

In a similar way, nodes can be created for higher order, like, quaternary, and $k$-ry consistency checks.

**Example 10.6**  Applying the Synthesizing algorithm to solve the coloring problem.

We apply the Synthesizing algorithm discussed above for this purpose. There are a total of six unary constraints $C_1-C_6$, and nine arc constraints. The arc, which is for binary, indicates $C_i$ "color is not the same as" $C_j$. To begin with, all the states are assigned $rgb$ (Fig. 10.7a). After this, to start with *MP* is constrained to $r$, therefore, $r$ is removed from the rest (Fig. 10.7b). In the next and final stage node consistency is assured, starting from *UP* and propagating to the rest of the network, simultaneously, maintaining the arc-consistency also as shown in figure (Fig. 10.7c).

The progressive assignment of colors is also shown in Table 10.1.                □

## 10.5   Solution Approaches to CSPs

A constraint satisfaction problem is a high-level description of a problem, where, a model of the problem is represented as a set of variables and their domain values. The problem statement is in the form of constraints that specify the relations between the
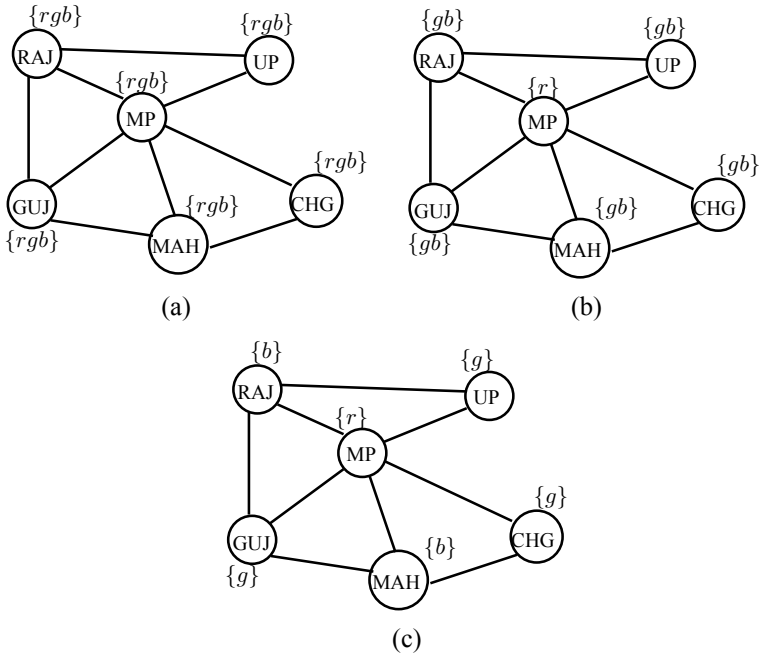
**Fig. 10.7** Graph through synthesizing algorithm

**Table 10.1** Graph coloring as state-space search

| Steps↓ Regions→ | MP | UP | RAJ | GUJ | MAH | CHG |
|---|---|---|---|---|---|---|
| 0. Initial: | {} | {} | {} | {} | {} | {} |
| 1. *rgb* | *rgb* | *rgb* | *rgb* | *rgb* | *rgb* | *rgb* |
| 2. (*MP* = *r*) | *r* | *gb* | *gb* | *gb* | *gb* | *gb* |
| 3. (*UP* = *g*) | *r* | *g* | *b* | *gb* | *gb* | *gb* |
| 4. (*GUJ* = *g*) | *r* | *g* | *b* | *g* | *b* | *gb* |
| 5. (*CHG* = *g*) | *r* | *g* | *b* | *g* | *b* | *g* |

variables. However, these constraints only specify the relations between variables, without any computational procedure/method to enforce such relations. The solution to such CSP is an algorithm which arrives at results, while satisfying the constraints relations. The finite domains' constraint satisfaction problems are typically solved using some type of search. The most common techniques used for solutions to CSPs comprise three things: 1. some variant of backtracking search, constraint propagation, and local search [4].

There are several variants for backtracking. The backtracking ensures that the solution is consistent, and it saves the search time once it finds that a particular iteration of search is not going to lead to a solution. The backtrack is usually based

on the value of a single variable, but it can be done on more than one variable. The learning through constraints infers and saves the new constraints which can later be used to avoid part of the search. We can foresee the effects of choosing a variable or value, through look ahead, and can determine in advance whether the sub-problem is going to be satisfied or not.

The propagation of constraints is useful because it turns a problem into one that is equivalent but is usually simpler to solve. In addition, it proves satisfiability or unsatisfiability of the problem. However, the later is not always guaranteed to happen, but it always happens for some forms of constraint propagation or some kind of problems, or both.

The process of *Constraint propagation* in the course of solution of a CSP problem modifies the CSP, in other words, the constraint propagation enforces a local consistency, which is a set of conditions related to the consistency of a group of variables or constraints, or both.

The *Local search* methods are nothing but incomplete satisfiability algorithms. They may find a solution to a problem, but may fail also even if the problem is satisfiable. The local search works iteratively to improve an assignment over variables. At each step of the iteration, a small number of variables are changed values, with the aim of increasing the number of constraints satisfied by this assignment.

A local search algorithm specific to CSPs is *min-conflicts* algorithm. The local search appears to work well when changes to variables are of random nature. The search, when integrated with local search, is called a hybrid search, and found to give better results.

Yet another approach for CSP solution is *Neighborhood substitution*. As per this, whenever a value *a* for a variable is such that it can be replaced in all constraints by another value *b*, then *a* is eliminated. The neighborhood substitutions are useful to find a single solution, as well all the possible solutions.

## 10.6 CSP Algorithms

We will consider three solution methods for constraint satisfaction problems: Generate-and-Test, Backtracking (possibly Dependency Directed), and Consistency Driven. Solving a CSP could mean to find:

- *One solution*, without preference as to which one,
- *All solutions*,
- An *optimal*, or at least a good solution.

The nature of methods for solving a CSP are:

1. **Combinatorial** methods for finite domain of values, *D*: The Solutions can be found by systematic search in *D*, either traversing the space of partial solutions, or explore the space of complete value assignments.

2. **Analytical** methods for the infinite domain of values $D$: Solutions here can be found by analyzing the constraints as some (generalized) equation system, either solving the constraints simultaneously or considering the constraints sequentially one by one.

A systematic search can be carried out in one of the two ways, either using *Generate and Test* approach or through *Backtracking*.

### 10.6.1   Generate and Test

We generate one by one all possible complete variable assignments and for each, we test if it satisfies all constraints. The corresponding program structure is very simple, just nested loops, one per variable. In the innermost loop, we test each constraint. In most situation, this method is intolerably slow. The steps for generate-and-test is shown as Algorithm 10.1.

---

**Algorithm 10.1** Generate-and-Test

---

1: **repeat**
2:     Assign a value to each variable.
3:     **if** it is a solution **then**
4:         return *Success*
5:     **else**
6:         modify the assignment
7:     **end if**
8: **until** All assignments are done
9: return *fail*

---

This algorithm searches all of the domain of values, $D$. However, improvements are possible, like, use an informed/smart generator such that the conflicts found by the tester are minimized. Such algorithms are called *stochastic* algorithms.

Example for generate and test is assigning colors to nodes in the graph where we assign values to variables and modify the assignment till we get a solution; and generating decimal digits for assignment to variables in the cryptoarithmetic problem 10.9 discussed later.

Merging the generator with the tester results in a new algorithm, called *backtracking* algorithm.

### 10.6.2   Backtracking

In the following, we go into the deeper concepts of backtracking to solve typical problems, and will try to express the complexity of solutions in mathematical forms.

Using a computer algorithm to answer a question like "How many ways are there to do so and so?", or "list all possible ways for ....", usually requires an exhaustive search out of a set of potential solutions. One technique for organizing these searches is the backtrack, which works by continually trying to extend partial solutions. At each stage of the search, if an extension of the current partial solution is not possible or not found, we backtrack to the original partial solution, and try a different extension of the existing partial solution. This method is, however, used in a wide range of combinatorial problems, which includes, parsing, game playing, and optimization [1].

We assume that the solution to a CSP problem consists of a set of vector $(a_1, a_2, \ldots)$ of undetermined length, which satisfies certain constraints on the components that makes it a solution. Each $a_i$ $(i = 1, 2, \ldots)$ is a member of a finite and linearly ordered set $A_i$. Thus, an exhaustive search must consider all the elements of set $A_1 \times A_2 \times \cdots \times A_i$, for $i = 0, 1, 2, \ldots$ as potential solutions. At the begin, we start with the null vector (say, $\Lambda$) as a partial solution, and the constraints tell us which of the members of $A_1$ are candidates for $a_1$. Let, this set be $S_1 \subseteq A_1$, the least element of $S_1$ is $a_1$, and the partial solution is $(a_1)$. In general, various constraints which describe solutions show that what subset $S_k$ of $A_k$ comprises candidates for the extension of the partial solution $(a_1, a_2, \ldots, a_{k-1})$ to $(a_1, a_2, \ldots, a_{k-1}, a_k)$. Suppose the partial solution $(a_1, a_2, \ldots, a_{k-1})$ allows for no possibilities for $a_k$, then $S_k = \phi$, hence we backtrack, and make a new choice for $a_{k-1}$. If no more new choices exist for $a_{k-1}$, then we backtrack still further and make a new choice for $a_{k-2}$, and this goes on, till a value next to previously selected value comes.

It is helpful to picture the above discussed process in terms of a tree traversal in Fig. 10.8. The subset of $A_1 \times A_2 \times \cdots \times A_i$, $i = 0, 1, 2, \ldots$ which, when searched, can be represented as a tree as follows: the node at 0th level /root node of the tree is null vector. The children of the root are the choices for $a_1$. In general, the nodes at the $k$th level are the choices for $a_k$, given the choices made for $a_1, a_2, \ldots, a_{k-1}$ as indicated by the ancestors of these nodes. In the tree, the dotted lines show the backtrack traverse. Since the traversal goes as deep as possible in the tree before backing up to explore other parts of the tree, such a search is called a depth-first search (DFS).



**Fig. 10.8** Backtrack-tree searches as depth-first search

Start

Choices for $a_1$

Choices for $a_2$, given $a_1$

Choices for $a_3$, given $a_1$ and $a_2$
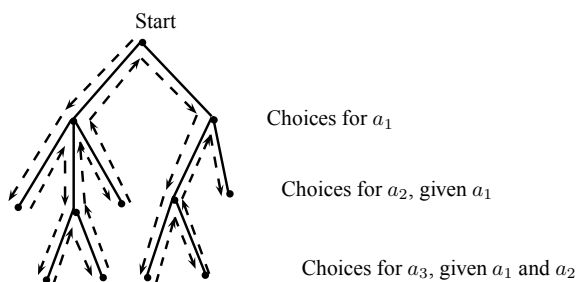
Backtracking Algorithm

Backtracking is performed through recursion, where it maintains a partial assignment of the variables. All variables are initially unassigned. At each step of the algorithm, a variable is chosen and all possible values are assigned to it in turn. For each value assigned, consistency of partial assignment is checked to find out whether it matches with the constraints. For each case of a consistency check, a recursive call is performed. When all values for a variable have been tried, the algorithm backtracks and goes back to previous variable to try a different value for this and start again. In the basic backtracking algorithm, consistency is defined as—satisfaction of all constraints whose variables are all assigned.

As the backtrack is no more than an "educated" exhaustive search procedure; it should be stressed that there exist numerous problems that even the most sophisticated application of backtrack will not solve in a reasonable length of time.

The checking or verifying the assignment of variables is not done in arbitrary order. The variables are tried in some order, so as to place in the front those variables that are highly constrained or with smaller ranges. This ordering has an impact on the efficiency of solution algorithms, as once the more constrained variables are satisfied, there are less chances of backtracking the solution. In the process we start assigning values to variables, check constraint satisfaction at the earliest possible time, and extend an assignment if the constraints involving the currently bound variables are satisfied.

The Algorithm 10.2 shows the steps for backtracking.

---

**Algorithm 10.2** Recursive-backtrack()

---
1: assignment={}
2: **if** assignment is complete **then**
3:    return assignment;
4: **end if**
5: $v$ = select-unassigned-variable();
6: **for** each value $d$ in order-of-domain values **do**
7:    **if** $d$ is consistent with assignment according to constraints **then**
8:       add $\{v := d\}$ to assignment;
9:       result := Recursive-backtrack();
10:      **if** result is NOT failure **then**
11:         return result;
12:      **else**
13:         remove $\{v := d\}$ from assignment;
14:      **end if**
15:   **end if**
16: **end for**
17: return *fail*

---

The variable and value ordering in backtracking are important. Note the line contained by this backtracking algorithm:

$$v = select-unassigned-variable();$$

which selects the next unassigned variable. A static variable never results in a most efficient search. We pickup that variable which fails quickly (i.e., with fever legal moves), to help faster exploration of state space.

However, the following are the problems with backtracking:

- *Thrashing*. This is repeated failure due to the same reason generated by older variable assignments.
- *Redundancy*. This is rediscovering the same inconsistencies.

An example, we use a backtracking algorithm to solve the problem of $n \times n$ non-attacking queens, i.e., in how many ways can $n$ queens be placed on an $n \times n$ chessboard so that no two queens are attacking each other? The following example presents the solution for $n = 4$ queens problem [10].

**Example 10.7** Four/Eight Queens problem.

"Eight queens" is a classic problem of combinatorial analysis, which requires placement of eight "queens" on a $8 \times 8$ chessboard such that no pair of queen "attack" each other, i.e., no two queens are on a common row or column or diagonal. This problem is ideally suited for a solution by backtracking algorithm. Instead of 8 queens, here we use its simpler version—a scaled down version to place four queens on a $4 \times 4$ chessboard so that no two queens attack each other.

Instead of beginning by examining of all $^{16}C_4 = 1820$ ways of selecting the locations for four queens on 16 squares of $4 \times 4$ board, the first observation is that each row is to contain exactly one queen. Thus, to begin with, a queen is placed on the first square of the first row (Fig. 10.9a), and then another queen on the first available square of the second row (Fig. 10.9b). Observing that this prevents placing a queen anywhere in the third row (Fig. 10.9c), it is necessary to backtrack to the second row and move the queen there one square to the right (Fig. 10.9d), so that there is room for another queen in the third row.

We note that now there are no available squares (Fig. 10.9e) in the fourth row, as every position in fourth row is crossed by some already placed queen. Therefore, it is necessary to backtrack all the way to the first row (Fig. 10.9f). After this, there is one available location in the second row (Fig. 10.9g), followed by suitable locations in the third and fourth rows (Fig. 10.9h), which gives us a solution. Note that, backtracking reveals that this solution is unique except for its mirror images, which are, of course, also solutions. The two mirror images can be obtained by rotating the board along the $X$ axis and $Y$ axis.

A backtrack search techniques cut down the search space, as they need not test all combinations of possible assignments to variables—a solution backtracks whenever it comes across an inconsistency of assignment. However, the backtrack often exhibits costly "thrashing" behavior.

Similar to 4-queen, the eight queens puzzle is the problem of placing eight chess queens on an $8 \times 8$ chessboard so that no two queens attack each other (see Fig. 10.10).
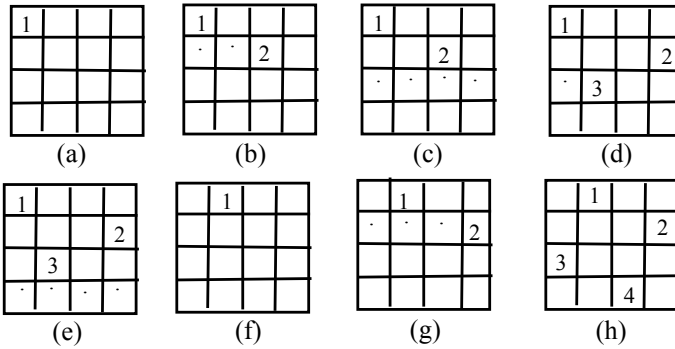
**Fig. 10.9** Backtrack based solution for four queens Problem

**Fig. 10.10** Solutions to
non-attacking queens on the
$8 \times 8$ chessboard
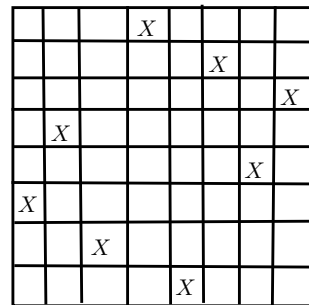


If we extend, a solution to $n$-queens problem requires that no two queens share
the same row, column, or diagonal. The eight queens puzzle is an example of the
more general $n$-queens problem of placing $n$ queens on an $n \times n$ chessboard, where
solutions exist for all natural numbers $n$ with the exception of 2 and 3. The eight
queens puzzle has 92 distinct solutions.                                          □

## 10.6.3  Efficiency Considerations

Though the backtracking algorithm does not try all the alternatives of assignments,
hence it is a relatively efficient algorithm. But, there is a number of heuristics, using
which we can cut the cost, and make it more efficient. The following are the ways to
improve the backtrack and make it more efficient [1].

*Branch merging*

The branches of the search path isomorphic to the already searched path should not
be searched and instead be merged with the already searched path. This will reduce
the total search time.

*Preclusion*

In the generation of solutions through backtracking, the backtracking should occur as soon as it is discovered that the current partial solution is in fact not leading to a solution, and will only add to the time required for the solution.

*Search rearrangement*

The search should be rearranged such that nodes of low degree occur at the beginning of the search, and of high degree occur in the latter part of the search. This will not contribute to the explosive growth of search tree in the beginning, and high branching at the ending nodes does not contribute to much complexity. Since preclusion frequently occurs at a fixed depth, in this strategy fewer nodes may need to be examined. As an example, when faced with the choice of several ways of extending the partial solution, e.g., which square to tile next or in which column to place the next queen, we choose the one that offers the fewest alternatives.

*Branch and bound*

When searching for a solution of *minimum cost*, once a solution is found all the partial solutions with greater cost than the minimum are discarded. When this approach is used, it is possible to get a good solution early in the search. Note that, this technique is possible only when the costs are additive in nature.

## 10.7  Propagating of Constraints

The solution to CSP requires propagation of constraints. Let us find out what is the basic mechanism to propagate these constraints. Consider that we have to locally propagate a constraint $C_J$ to a neighboring constraint $C_K$. For this, we should remove all $a_K \in C_K$ which do not satisfy $C_J$.

A global propagation of a constraint is defined recursively. To globally propagate a neighboring constraint $C_J$, first locally propagate $C_J$ to $C_K$, then if anything was removed from $C_K$ due to local propagation, globally propagate $C_K$ through all its neighbors. The rest of the propagation is similar to *arc consistency* algorithm (see page no. ).

Yet another situation is, a constraint is looked at only when a variable is chosen, by selecting the variable that is yet unassigned. This also will reduce the propagation overheads as, before this time it is not required, and hence we are not concerned.

In the following, we discuss the various techniques for the propagation of constraints.

### 10.7.1   Forward Checking

We can drastically reduce the search space by early looking at some of the constraints, even before the search is started. This process is called *forwarding checking*. When a variable, say $X$ is assigned a value, the forward checking process checks each of the variable $Y$ that is unassigned and connected to $X$ by some constraint. And, a value in $Y$'s domain is deleted that is inconsistent with the value chosen for $X$. To apply this in Fig. 10.7, for map coloring, we keep check of remaining legal moves for unassigned variables, as shown in the Table 10.1. When 'red'is assigned to MP, the red color is dropped from all the rest, otherwise, it will conflict with any red with any of the other variables. Hence, in the table, once the MP is assigned R, the R is deleted from rest of all states (i.e., variables $Y$). This has reduced the *ranching* on variables all together, by propagation of information from MP to rest of all states.

   In the above, we terminate the assignment when any variable has no legal moves. Try yourself the forward checking for the 4-queen problem yourself.

### 10.7.2   Degree of Heuristics

*Variable with the largest number of constraints first*

Select that variable as the next variable for assignment, which is involved in the largest number of constraints on other unassigned variables. This will leave less number of assignment to other variables, hence if it does not succeed, it will return back quickly without wasting must of the space and time.

*Least Constraints value heuristics*

Given a variable, choosing the least constraining value leaves the maximum flexibility for subsequent assignments.

**Example 10.8**   Constraint Propagation.

A constraint graph is shown in Fig. 10.11, for the domain values {1, 2, 3, 4}. Following are the constraints:

**Fig. 10.11**   Constraint graph
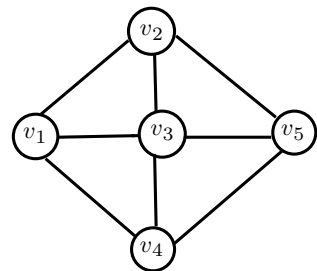
1. There are two unary constraints: Variable $v_1$ cannot take values 3, 4, and variable $v_2$ cannot take value 4.
2. There are eight binary constraints, stating that variables connected by edges cannot have the same value.

It is required to find the solution using the following heuristics: *minimum value heuristics* (MVH), *degree heuristics* (DH), and *forward checking* (FC).

$MVH : v_1 = 1$, the minimum value integer is selected first.
$FC + MVH : v_2 = 2$, next minimum value is selected with forward checking.
$FC + MVH + DH : v_3 = 3$, the $v_3$ is maximum constrained, due to $v_1$ and $v_2$ adjacent nodes.
$FC + MVH : v_4 = 4$, forward checking with minimum value heuristics.
$FC : v_5 = 1$

The first variable $v_1$ is assigned the minimum value (i.e., 1) out of the available set of values at the start. The forward checking and MVH assign next, minimum available value to $v_2$, while checking forward, i.e., $v_3$, $v_5$, that there is no conflict. Next using the FC heuristics, MVH and DH, we assign 3 to $v_3$. Using FC and MVH we assign 4 to $v_4$. Finally, the variable $v_5$ is assigned 1, through forward checking.

In the above, the *degree heuristics* select that variable, which is involved in the largest number of constraints on other unassigned variables. It is useful for tie-breaking. In other words, it is selecting the least *constraints*, where given a variable choose the least constraining value (leaves the maximum flexibility for subsequent assignments).                                                                    □

## 10.8  Cryptarithmetics

The cryptarithmetic codes are typical of encrypting some information using these codes, or hiding an encryption key such that it can be recovered on solution of the given constraints. The cryptarithmatic puzzles are CSPs that lie at the heart studies of human and computer and computer problem-solving. Usually, the problem to be solved is to find unique digit assignments to each of the letters such that the numbers represented by words do the correct addition or subtraction, or even any given arithmetic. However, this problem turns out to be complex. In decimal representation, the constraint of a unique digit mapping to a unique letter reduces the total number of states, from $10^n$ to $10!/(10 - n)!$, where $n$ is a number of unique letters in the problem.

An example of such a problem is:

```
   GERALD
   DONALD
   -------
   ROBERT
```

This has $n = 10$, and one solution is:

```
197485
526485
-------
723970
```

For solving such a cryptic problem requires an agent to perform a search. The speed at which these problems can be solved depends on many factors, like, initial conditions, and the particular sequence of actions chosen by the agent to solve it while moving through the search space. The chosen sequence depends on the knowledge about the problem/puzzle to be solved, as well as the knowledge about which steps to be taken next.

A simple *non-cooperative* search strategy is—generate-and-test, where an assignment is made to each letter and tested to see if the problem is solved or partly solved.

In the following example, we show the detailed steps of generate and test, and use backtrack to solve a similar problem [9].

**Example 10.9**  Cryptographic puzzle.

Given, a sum as below, find out the values of these 8-variables ($S, E, N, D,$ $M, O, R, Y \in \{0, 1, \ldots, 9\}$) satisfying the validity of this sum, such that each variable gets a unique value from set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

```
  S E N D
+ M O R E
=========
M O N E Y
```

For the above, the constraints can be formulated as follows:

$$c_1 : D + E = Y + 10 \times v_1$$
$$c_2 : v_1 + N + R = E + v_2 \times 10$$
$$c_3 : v_2 + E + O = N + v_3 \times 10$$
$$c_4 : v_3 + S + M = O + v_4 \times 10$$
$$c_5 : M = v_4$$

In above, variables $v_1, \ldots, v_4 \in \{0, 1\}$ are carries. Also, Since $M \neq 0$, so, $S \neq 0$. In the above, $c_1 \ldots c_5$ are constraints. The value range for variables is as follows:

$$D, E, N, R, O, Y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\},$$
$$M, S \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

From above, we get straight forward, $v_4 = M = 1$. Thus, 1 can be removed from the value sets of all the other variables. Hence:

$$D, E, N, R, O, Y \in \{0, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

Assume that $v_3 = 1$.

$$c_4 : v_3 + S + M = O + v_4 \times 10$$
$$1 + S + 1 = O + 1 \times 10$$

Hence, $S = 8$ or $S = 9$. Since, $M = 1$ requires $O \neq 1$. So, $O = 0$, and $S = 8$.

$$c_3 : v_2 + E + O = N + v_3 \times 10$$
$$\therefore v_2 + E = N + v_3 \times 10 = 10 + N.$$

The above cannot be satisfied for any constraint value for $E$, since $E < 10$. This concludes that assumption $v_3 = 1$ was wrong. Hence, this and all the derived values must be reset, and we backtrack to select $v_3 = 0$.

$$c_4 : v_3 + S = O + 9$$
$$S = O + 9.$$

Therefore, the only solution is $S = 9$, and $O = 0$. $C_3$ is calculated as:

$$c_3 : v_2 + E + O = N + v_3 \times 10$$
$$v_2 + E + 0 = N + 0 \times 10$$
$$\therefore v_2 + E = N.$$

The assumption $v_2 = 0$ will leads to contradiction $E = N$. Hence, $v_2 = 1$. Therefore, $1 + E = N$. The $c_2$ reduces to:

$$c_2 : v_1 + N + R = E + v_2 \times 10$$
$$v_1 + 1 + E + R = E + 1 \times 10$$
$$v_1 + R = 9.$$

Because, $S$ is already 9, $R$ cannot be 9. Hence, $v_1 = 1$ and $R = 8$.
Let us try to propagate the set of values. The present sets are:

$$D, E, N, Y \in \{2, 3, 4, 5, 6, 7\} \tag{10.9}$$

From the constraint $c_1$, since $v_1 = 1$, we have,

$$c_1 : D + E = Y + 10 \times 1$$

That is,

$$D + E = Y + 10 \tag{10.10}$$

We note that already assigned values to variables are $\{S = 9, O = 0, M = 1, R = 8\}$. From Eqs. 10.9 to 10.10, we note that $D + E \geq 12$, $Y$ can be assigned value $\geq 2$. Various possibilities are:

- $(D, E) = (6, 7)$. Thus, $1 + E = 1 + 7 = 8 = N$. This is not possible, because $R$ is already 8.
- $(D, E) = (7, 6)$. Thus, $N = 1 + E = 7$. This fails, due to conflict with $D = 7$, and $N = 7$.
- $(D, E) = (7, 5)$. Thus, $N = 1 + E = 6$. $Y = D + E - 10 = 2$.

The unique solution of the problem is given below.

```
    S  E  N  D              9  5  6  7
 +  M  O  R  E           +  1  0  8  5
    =========      =>       =========
 M  O  N  E  Y           1  0  6  5  2
```

□

## 10.9   Theoretical Aspects of CSPs

The CSPs are also studies in *computational complexity theory* and in *finite model theory*. The CSPs are well-known for their complexities, which are exponential in nature. To solve a CSP one needs to establish a relation, like mapping nodes in a graph to certain colors, satisfying some constraints; assigning queens to blocks on a chessboard, with certain constraints; mapping letters to digits in a cryptarithmetic, etc. In all these cases we have certain mapping relations, with certain constraints. From these observations an important question that can be asked is, whether for each set of relations, the set of all the CSPs that can be represented using only relations chosen from that set is *P* or *NP-complete*? If such a theorem is true, then CSPs provide one of the largest known subsets of *NP*. The classes of CSP that are known to be tractable, are those where hypergraph of constraints have bounded tree-width and there are restrictions on the set of constraints relations [7].
Some Properties of CSPs are:

- Solution of CSPs is in general NP-complete.
- Before a CSP is solved, identification of restrictions that make the problem tractable is important.
- Each CSP can be converted into a binary CSP.
- *Over-constrained CSP*: The CSP contains more constraints than required which may be inconsistent and / or redundant.
- *Under-constrained CSP*: The CSP that cannot be solved uniquely.

## 10.10 Summary

In Constraint Satisfaction Problems(CSP), one needs to search the state space, but every move is subject to the fulfillment of certain constraint, which is different from the selection of the best fitting state in best-first search or hill-climbing. The aim of their solution is to find an assignment of values to a given set of variables subject to specified constraints.

The CSP problems are common in job scheduling, supply chain management, CPU job scheduling, and other optimization problems. A constraint satisfaction problem is defined as a triple,

$$\langle V, D, C \rangle$$

where $V$ is a finite set of variables, $D$ is a set (domains) of values, and $C$ is a finite set of constraints.

A CSP can be represented as a *constraint graph*, where the vertices correspond to the variables and the edges to the binary constraints. Types of variables in CSP can be either discrete variables or continuous variables. The variables can be further classified as fixed value quantity, set of values, or symbolic expressions. The type of constraints can be unary constraints, binary constraints, higher order Constraints or preference/soft constraints. Application of constraints can be carried out using *synthesis algorithms*.

CSPs are typically solved using: *variants of backtracking*, *constraint propagation*, and *local search*.

The efficiency considerations for CSPs are: preclusion, branch merging, search rearrangement, and branch and bound.

The constraint propagation in a network is done through forward checking.

The selection of the most fitting variables for the first assignment is done through the criteria of degree heuristics, i.e., either a variable having the largest number of available constraints is handled first, or the one having the least constraints value heuristics.

## Exercises

1. How many solutions exists for the map coloring of Indian states (see Fig. 10.1)?
2. A problem called, "Minimal Bandwidth Ordering Problem" is described as follows: Given is a graph $G = (V, E)$, with $V$ as a set of vertices, $E$ as a set of edges $(x, y)$, and $x, y \in V$. If we order the vertices, the *bandwidth* of a vertex $x$ is the maximum distance between $x$ and a vertex $y$ that is adjacent to it. The *bandwidth of ordering* is the maximum bandwidth of all the nodes under that ordering. The *minimal bandwidth ordering problem* is to find an ordering with the minimum bandwidth.

a. Formulate the Minimal Bandwidth Ordering Problem as a CSP. Clearly state the parameters: $V, D, C$. Ignore the minimization requirement.
b. Given part (a) above, describe the topology of the constraint graph.
c. Given the part (a), what is the size of the search space?
d. Are there any constraint satisfaction techniques effective for solving this optimization problem? If yes, explain such techniques, and if no, justify it.

3. *Sudoku* can be viewed as a binary constraint satisfaction problem.

   a. What are the variables of this CSP?
   b. What are their domains?
   c. How would you translate the requirement that no two of the same digit may occur in the same row, column, or block into binary constraints?
   d. Does the requirement that each digit occur at least once in each row, column, or block have to be directly specified? Why or why not?

4. For what size of $n$, the $n$-queen problem has no solution, with $n$ in the range 1-10.

5. Find out, what techniques have been used for propagation of constraints at various stages in the solution of the problem 10.9.

6. What algorithms or heuristics are relevant to solving some constraint satisfaction problems in the following situations. Justify your answers.

   a. The domain sizes of the problem vary significantly: some variables have very large domains (over $1,000$ values) and some have very small domains (less than 10 values).
   b. The problem is so tightly constrained that it is quite unlikely that a solution exists.

7. Solve the following CSP problems:

   a. $TWO + TWO = FOUR$
   b. $ABC + DEF = GHIJ$
   c. $ALFA + BETA + GAMA = DELTA$

8. Give an algorithm for CSP. Justify that this standard algorithm is a depth-first search (DFS). Can the CSP be implemented as:

   a. Breadth-first search,
   b. Best-first search.

   Justify. Also, modify the existing DFS algorithm to work as BFS, and as best-first search.

9. Consider the problem of coloring a complete three-vertex graph with three colors, where the nodes $X_1, X_2, X_3$ are all the set $\{r \ g \ b\}$; $X_{12}, X_{13}$, and $X_{23}$ all equal $\{rg \ rb \ gr \ gb \ br \ bg\}$ and $X_{123} = \{rgb \ rbg \ brg \ bgr \ grb \ gbr\}$, the six possible colorings. Make use of extended synthesizing algorithm to color this graph.

**Fig. 10.12** Tiles to be colored

|     |     |     |
|-----|-----|-----|
| $X_1$ | $X_2$ | $X_3$ |
| $X_4$ | $X_5$ | $X_6$ |
| $X_7$ | $X_8$ | $X_9$ |
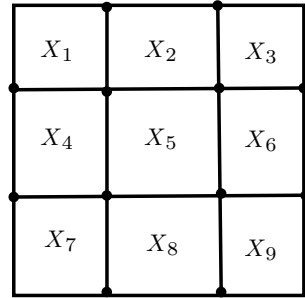
10. For the solution of a CSP problem, explain,

    a. Why it is good heuristics to first choose a variable that is most constrained?
    b. Why it is good heuristics to first choose a value of a variable that is least constrained?

11. For the solution of a CSP problem, explain,

    a. For better heuristics, why it is preferred to first choose a variable that is most constrained?
    b. For better heuristics, why it is preferred to first choose a value of a variable that is least constrained?

12. The Fig. 10.12 shows 9-tiles in the form of variables $(X_1 \ldots X_9)$ for CSP. Write the steps of an algorithm to color these tiles using $R, G, B$ colors such that no adjacent tiles have the same color. Also, so the final assignment of colors to these tiles. (Note: A tile is adjacent to others if it is in the same row or column and near to the other.)

13. In a department of engineering, there are teachers $(T_1 \ldots T_8)$, subjects (courses) $(S_1 \ldots S_6)$, tutorials $(t_1 \ldots t_3)$, lecture theaters $(LT_1 \ldots LT_3)$, tutorial rooms $(tr_1, tr_2)$. The courses and tutorials are to be allocated to teachers and rooms on days $(D_1 \ldots D_5)$, and periods $(P_1 \ldots P_6)$ in a day. The constraints are: no two subjects be given to a teacher on the same day, the teacher should get subject in the same $LT$ every time, as well as the tutorial class in the same $tr$. The tutorial $t_i$ shall be engaged by the same teacher who is engaging the course $S_i$. Define the $\langle V, D, C \rangle$, elements of each set, and show the part of the graph for assignment.

14. Consider the following definition of Comma-Free Codes: They have the property that, if $a_1 a_2 \ldots a_k$, and $b_1 b_2 \ldots b_k$ are in the subset, then none of the "overlaps" $a_2 a_3 \ldots a_k b_1, a_3 a_4 \ldots b_1 b_2, \ldots, a_k b_1 \ldots b_{k-2} b_{k-1}$ are in the subset. Such a subset is called a *comma-free dictionary*. This is because, in any "message" consisting of consecutive words from this collection, such as $a_1 a_2 \ldots a_k, b_1 b_2 \ldots b_k$, one may omit the commas or other separations between words, without the possibility of ambiguity as to word beginning and ending, even if one begins at a random starting point within the message.

In an investigation of codes with some synchronization properties, we consider $n^k$ "words" which consist of $k$ alphabets from a $n$-symbol alphabet. In this, we look for the largest subset of these words which has the comma-free property.

As an example with three and four-letter English words: if BOOK and INK were in the dictionary, then KIN could not be in the dictionary because of the ambiguity in booKINk, as it would result to identification of individual words as boo+KIN+k, which is inconsistent.

Find out other words for $n$-symbols alphabets, and possible upper limits, if any?

## References

1. Bitner JR, Reingold EM (1975) Backtrack Program Tech. Communications of the ACM 18(11):651–655
2. Bordeaux L et al (2006) Propositional satisfiability and constraint programming: a comparative survey. ACM Comput Surv 38(4):1–54. https://doi.org/10.1145/1177352
3. Cooper MC (1997) Fundamental properties of neighborhood substitution constraint satisfaction problems. Artif Intell 90:1–24
4. Dorndorf U et al (2000) Constraint propagation techniques for the disjunctive scheduling problem. Artif Intell 122:189–240
5. Freuder EC (1978) Synthesizing constraint expressions. Commun ACM 21(11):958–966
6. Freuder EC (1985) A sufficient condition for backtrack-bounded search. J ACM 32(4):755–761
7. Jerrum M (2010) Constraint satisfaction problems and computational complexity. Commun ACM 53(9):98–106
8. Michalowski M (2008) A general approach to using problems instance data for model refinement in constraint satisfaction problems. A Dissertation Presented to the Faculty of graduate school, USC, Computer Science Department
9. Scott H (1991) Cooperative solution of constraint satisfaction problems. Science 254:1181–1182
10. Solomon W et al (1965) Backtrack programming. J ACM 12(4):516–524

# Chapter 11
# Adversarial Search and Game Theory

**Abstract** Game theory is the formal study of conflict and cooperation, first time introduced as long back as 1921 by mathematician Emile Borel, then enriched in 1928 by von Neumann and Oskar Morgenstern, and much enriched by Josh Nash, has enormous applications, including in business, and even in the prediction of election results, etc. The game playing is also a search process. The chapter presents the classes of games as combinatorial and games of chance, then further as zero-sum games and non-zero-sum games, the prisoner's dilemma, game playing strategies, the games of perfect information, arbitration scheme in games, minimax search in game playing, and analysis of specific games like tic-tac-toe. The more efficient search processes like alpha and beta are presented, as well as the alpha cutoff and beta cutoff methods to prune the search process are presented and analyzed, followed with chapter summary, and an exhaustive list of exercises along with a number of multiple-choice questions provided at the end.

**Keywords** Game theory · John Nash · Nash equilibria · Zero-sum games · Non-zero-sum games · Games as search · Prisoner's dilemma · Arbitration scheme · Minimax search · Game theory applications · Tic-tac-toe · Alpha-beta search · Alpha cutoff · Beta cutoff

## 11.1 Introduction

The Game theory is the formal study of *conflict* and *cooperation*. Game theory's concepts apply whenever the actions of several agents are interdependent. These agents may be individuals, groups, firms, or their combination. The concepts of game theory provide a language to formulate, structure, analyze, and understand strategic scenarios. *Cooperative game theory* concentrate on what agreements the agents are likely to reach, while *noncooperative game theory* concentrates on what strategies are likely to be adopted by the agents. Both of these traditions have adopted models of bargaining in which each agent's risk posture is conveyed by comparing his preferences for risky and riskless alternatives.

The formal game theory was introduced in 1921 by mathematician Emile Borel, which was further advanced by the mathematician John von Neumann in 1928 in a text by title "Theory of parlor games." The Game theory became a field of study in mathematics, subsequent to the publication of the book "Theory of Games and Economic Behavior" in 1944 by John von Neumann and the economist Oskar Morgenstern. John Nash, in the year 1950, demonstrated that finite games have always an equilibrium state. At this *equilibrium state* all the players choose actions that are the best for them given their opponents' choices. From that time onward, the concept of equilibrium has remained the focal point of analysis of noncooperative game theory. In the year 1960, the scope of game theory got widened and it was being in problems related to war and politics. The whole subject of game theory caught special attention when Nobel prize was awarded jointly to John Nash, John Harsanyi, and Reinhard Selten, in the field of Economics, in 1994.

Another branch of game theory, called, *Combinatorial Game theory* (CGT) studies strategies and mathematics of two-player games of *perfect knowledge* such as *chess* or *go*. But generally concentrates on simpler games like *nim* or solving *endgames* or their special cases. An important difference between this subject (i.e., CGT) and *classical game theory* (a branch of economics, after von Neumann and John Nash) is that in CGT, game players are assumed to move in sequence rather than simultaneously, so there is no scope in randomization or information hiding strategies.

The CGT does not study games with importance in computer knowledge called games of chance, like poker (card games). The combinatorial games include games like chess, checkers, Go, Arimaa, Hex, and Connect. They also include one player combinatorial puzzles, and even no player automata.

In CGT, moves are represented as game-tree, and the trees can be searched while making moves in the game. A game-trees is a special kind of semantic tree where nodes represent board configurations and branches indicate how one board configuration can be transformed into another configuration by a single move. The decisions regarding the next move are made by two *adversaries*, who play the game, hence the name *adversarial search*.

Since the number of nodes in a binary or higher order tree generally increases exponentially with increase in depth of the tree, it is usually unfeasible to do an exhaustive search in a game-tree. The search-procedures, called as, *alpha-beta* search are commonly used to guarantee correct results without exhaustively searching for such trees. However, such procedures still require complete search of the tree in special situations. The fact of these game-trees is that some times they are so large that you cannot fit them even in the hard-disk, hence practically impossible to completely search such trees.

Improved results can be obtained in the game playing computer programs, by searching the tree to only a limited depth. This is possible by static evaluation functions to estimate the utility vales at some given nodes. This is followed by computing the utility values at shallower nodes in the tree assuming that estimated utility values were correct. This technique is called *heuristic game-tree searching*, is used by us in "real-life" decision situations.

It is often a universal agreement that when a game-tree is heuristically searched, increasing the depth of search always improves the quality of decision. This has been demonstrated with many game-playing computer programs, but such results are purely empirical.

This chapter presents the idea of game theory, where each of the opponents search the tree, like DFS and BFS. However, the strategy of each player's move (in two-player game) is to defeat others. There is a score value, which one player (called *maximizer*) tries to maximize, while the other (called *minimizer*) tries to minimize. The search time can be reduced using *alpha-beta* search method.

The games theory has applications in finance and business systems, economics, and other areas where one needs to build a model in two more contradicting situations [7].

**Learning Outcomes of this Chapter**:

1. Game theory and its applications. [Familiarity]
2. Equilibrium state in games. [Usage]
3. Types of games. [Familiarity]
4. Compare and contrast basic search issues with game playing issues. [Familiarity]
5. Apply minimax search with alpha-beta pruning to prune search space in a two-player game. [Usage]

## 11.2  Classification of Games

Games are classified into several categories based on certain significant features, the most obvious of which is the number of players involved. A game can thus be designated as one-person, two-person, or $n$-person ($n > 2$). A player, i.e., a participant in a game need not be a single person. If each member of a group has the same feelings about how the game should progress, and end, the members may be called as a single player. A player may also be a corporation, or a basketball team, or even a country.

In games of perfect information, such as chess, each player knows everything about the game at all times. The Poker, on the other hand, is a game of imperfect information because players do not know about the cards that others are having.

The extent to which the players are opposed or coincidence is another basis of classifying the games. The *zero-sum* or to be precise *constant-sum games* are completely competitive. The Poker, for example, is zero-sum because the combined wealth of the players remains constant; if one player wins the other must lose. In non-zero-sum games, all players can be winners or losers. In a dispute between management versus labor, if some agreement is not reached, both will be losers. For example, if a labor strike cannot be avoided.

In cooperative games, the bidders may communicate and make agreement as binding to both the players, and in noncooperative they may not. The examples of players in cooperative games are car salesman versus customer, management versus employees, whereas, bidders for government contracts are supposed to be noncooperative bidders.

The games can further be classified as *finite* and *infinite* games. The games are finite if the player has to make only a finite number of decisions, and has only a finite number of alternatives. When either the decisions are infinite or the alternatives are infinite, the game is called infinite.

The simplest of all is a one-person game, with no opponent. As an example, for going to the office he/she may choose to go on foot or ride on a public transport or hire a cab, are the alternate moves. Each one of us as an individual always play many one-person games daily, like, doing shopping, doing homework given by a class teacher, managing kitchen, cleaning house, doing meditation; assuming that no other person or robot is involved in the above tasks. Each of these games (the tasks), has many steps or moves, and one needs to choose a move among number of available next move/choices. For example, while shopping, we may have options to: first collect vegetable items or the grocery items, and before the final move of making payment of bills, we may choose the move as, pay by cash, or by card.

## 11.3   Game Playing Strategy

Let there be a facility by which in a board game, every unique board configuration can be converted into a unique single, overall quality value in the form of an integer number. Further, consider that positive number indicates the move in favor of one player, and negative—in the favor of his/her opponent. The degree of favoredness is the absolute value of that number. Let us call ourself as *maximizer* player, and the opponent as *minimizer*. Also, let us imagine that there is a number (call it static *score*, accumulated so far), the maximizer will make a move such that the score increases to maximum, while the opponent (adversary) makes a move so that by addition of quality number, the score number minimizes.

The game playing is favorite for AI search, because:

- It is a structured task, often a symbol of "intelligence",
- clear definition of success and failure,
- does not require large amounts of knowledge (at first glance),
- focus on games of perfect information, and
- multi-player, and chance game.

The difference between the games and search problems is that the games playing are highly unpredictable, as one does not know what move the opponent is going to make. And, only based on the opponent's move the other player has to make a move.[1] The other difference is that the time limits for a move are in realistic times, hence which one is unlikely to find a goal in that time, approximations are necessary for both the players. The formal system behind these games is called *Game Theory* [6].

---

[1]A player can learn from the opponent's past moves as to how and what strategy the opponent followed. But, this requires the learning ability. Hence, the game's moves shall not be based current state, but past also. This has not been considered for the present discussion.

## 11.4 Two-Person Zero-Sum Games

The term "zero-sum" (or equivalently "constant-sum") means that players have diametrically opposed interests. The term comes from parlor games like poker, where there is a fixed amount of money around the table. If one player wins some money, others have to lose an equivalent amount. Two nations trading make a non-zero-sum game since both are simultaneously in gain. An equilibrium point is a stable outcome of the game associated with a pair of strategies [2].

Consider the Fig. 11.1, which is played between you and your opposite. The numerical values in the cells indicate the figures in dollars your opponent will pay to you. To start the game, assume that you select row $B$. Next, your opponent will select a column, say $II$. Therefore, your opponent will pay you \$4. Next, say, you select row $C$, your opponent, suppose selects column $III$. In this case, you pay to your opponent one dollar, as the value at the junction is $-1$. The two-player zero-sum game is a perfect information game.

A strategy in game theory is a complete plan of action that describes what a player will do under all possible circumstances. There are poor strategies and good strategies.

Other examples of two-person zero-sum games of perfect information are *chess*, *checker*, and Japanese *go*. Such games are strictly determined, i.e., rational players making use of all available information can deduce a strategy that is clearly optimal, hence the outcome of such games are deterministic. In chess, for example, one of these three possibilities exists: (1) white has a winning strategy, or (2) black has a winning strategy, or (3) black and white each have a strategy that is winning or draw [9].

Two-Person Non-zero-sum Games

Most games that arise in practice are non-zero-sum games, where players have both common as well as opposed interests. For example, the buyer vs. seller is a non-zero-sum game. The buyer wants a lower price and seller a higher price, but both want that deal to be executed. Similar is the case with the game of hostile countries, who may disagree on many issues but try to avoid the war. Many obvious properties of zeros-sum games are not available in non-zero-sum games. One of the differences is the effect of communication on the game, which does not help the opponent in zero-sum game.

**Fig. 11.1** Two-person zero-sum game

YOUR OPPONENT

|  |  | I | II | III |
|---|---|---|---|---|
|  | A | 5 | -2 | 1 |
| YOU | B | 6 | 4 | 2 |
|  | C | 0 | 8 | -1 |

The game of labor-union vs factory owner is a non-zero-sum game. If the management is properly informed about the demands of a labor union, probably, the strike may get withdrawn, benefiting both the laborer and factory owner. The games where players communicate, make a binding agreement between two parties is called *cooperative games*, and where players are not allowed to communicate are *noncooperative games*. In a non-zero-sum game, unlike in Fig. 11.1, each table entry consists of a pair of numbers. This is because the combined wealth of the players is not constant. Since it is not possible to deduce one player's payoff from the payoff of the other player, both player's payoff must be specified. The prisoner's dilemma is the best example of non-zero two-player noncooperative [2].

## 11.5   The Prisoner's Dilemma

Two criminals *A* and *B* are arrested and interrogated separately. Each suspect may either confess (*defect* the other prisoner) with a hope of a lighter sentence, or may refuse to talk (*cooperate* the other prisoner). The police do not have sufficient information to convict the suspects unless at least one of them confesses. If they cooperate, then both will be convicted to minor offense and sentenced to a month in jail in the absence of supportive evidence. If both are defecting, then both will be sentenced to jail for six months. If *A* confesses and *B* does not, then the *A* will be set free but the *B* will be sentenced to one year in jail, and vice versa when *A* does not confess and *B* confesses.

The police explain these outcomes to both suspects and tell each one that the other suspect knows the deal as well. Each suspect must choose his action without knowing what the other will do. This is shown in a table in Fig. 11.2.

If we observe closely the outcome of different choices that are available to the suspects, it becomes clear that regardless of what one suspect chooses, the other suspect feels better off by choosing to defect, which means he confesses. This concludes that both the suspects choose to defect, and stay for six months in jail. This choice is clearly a less desirable outcome than that of only one month in jail, which is the result if both cooperate to each other by not confessing (i.e., cooperating).

We assume that there is no honor among the criminals, and each one's sole concern is to save himself. This game is called *prisoner's dilemma*. Since a suspect must

**Fig. 11.2** Prisoner's dilemma

|  |  | Suspect *B* | |
|---|---|---|---|
|  |  | Confess | Do not confess |
| Suspect *A* | Confess | (6 mon., 6mon.) | (0 mon., 1 yr.) |
|  | Do not confess | (1 yr., 0 mon.) | (1 mon., 1 mon.) |

**Fig. 11.3** Prisoner's
dilemma applied for business

|  | | Firm $B$ | |
|---|---|---|---|
|  | | 1K dollar | 2k dollar |
| Firm $A$ | 1k dollar | (5k, 5k) | (3k, 6k) |
|  | 2k dollar | (6k, 3k) | (4k, 4k) |

make his own decision without knowing that of others, he must consider each of his
partner's alternatives and anticipate the effect of those on him.

Supposes that $A$ confesses, $B$ must go to jail for one year or six months. Alternatively, if $A$ does not confess, then either $B$ is set free or given one month's jail. In
either case, for $B$ it is better to confess!

However, if both cooperate to each other, both will not confess, and both go for
miner term in jail of 1 month. If both confess, they go to jail for 6 months each.

Further, we conclude that both may be cooperative or uncooperative. In cooperative case, each does better than they were uncooperative. Also, for any fixed strategy
of the opponent, other player does better by playing uncooperatively [2].

**Example 11.1** Prisoner's dilemma applied for business problem.

Consider the Fig. 11.3, which shows that there are two business firms $A$ and $B$, trading
in different regions, and each spending either $1,000 or $2,000 for publicity. The total
of publicity and sale figure is $12,000 (12k) always. If both the firms spend $1k for
publicity, their sale is $5k each, if both spend $2k, the sale of each is $4k. Similarly,
if $A$ spends $1k and $B$ spends $2k for publicity, $A$ has sale of $3k, while $B$'s sale is
$6k. Similarly in the reverse case. We assume that in $2k publicity expenditure for
each together, the extra market efforts is wasted, hence the sale comes down.

If any party is spending $2k each year, it will induce the other firm to spend also
$2k next time. A more optimistic strategy is to signal your intent to the opponent of
$1k expenditure, so he also spends $1k, and this cooperation results to $5k sale for
each. But, two competing firms know that they would not do the business forever,
hence they loose the cooperation, and each spends $2k, and the profit comes down
to $4k each.                                                                          □

Similar conditions to the prisoner's dilemma may appear in water consumption by
the citizens. Imagine that there is water shortage and citizens are urged to cut down
the water consumption. Each citizen responds to these requests by considering his
own interest, so no citizen will conserve water. This is with the prevailing concept:
any saving by an individual has a negligible effect on the city's water supply. If
everyone acts as per his own interest, the results will be catastrophic for all.

Imagine that two rival nations are preparing their military budget. Each nation
wants to maintain a military advantage over the other by building a powerful army,
and each spends accordingly. They ultimately end up having the same relative power
and good deal of power.

## 11.6  Two-Player Game Strategies

There are two key assumptions in playing the games:

1. Every player in the market acts as per his/her self interest. The players pursue well-defined exogenous objectives, i.e., they are rational. The players understand and try to maximize their payoff functions.
2. While selecting a strategy/plan of action, a player estimates the potential response/ reactions of other players. The player also takes into account his/her knowledge or expectation of other players' behavior, that is, the player reasons strategically.

Dominant Strategies

A dominant strategy is one that provides roughly the highest payoff for a particular player, irrespective of what strategies are employed by other players [9].

A formal definition of dominant strategy can be given as follows: Let there be a game played between $n$ players, and let $p_i(s)$ is the pay off to player $i$ as a function of all other players' strategies $s$, where,

$$s = (s_1, s_2, \ldots, s_n). \tag{11.1}$$

For the sake of simplicity, let us define $s_{-i}$ as a combined strategy of all $n$ players, except player $i$ as,

$$s_{-i} = (s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n). \tag{11.2}$$

The strategy $s_i$ is called dominant for player $i$ if for any strategy $s_{-i}$ and any alternative strategy $s_i'$, provided that the following pay off relation, holds for player $i$ with respect to all players $s_{-i}$.

$$p_i(s_i, s_{-i}) \geq p_i(s_i', s_{-i}). \tag{11.3}$$

**Definition 11.1** (*Dominant strategy*) For a player $i$, a strategy $s_i$ is called a dominant, if no matter what the other players choose, playing $s_i$ maximizes it's pay off.

**Definition 11.2** (*Strictly dominated strategy*) For a player $i$, a strategy $s_i$ is called strictly dominated, if there exists another strategy $s_i'$ such that no matter what the other players choose, playing $s_i$ gives player $i$ a higher payoff than playing any alternate strategy $s_i'$. That is,

$$p_i(s_i, s_{-i}) > p_i(s_i', s_{-i}). \tag{11.4}$$

In the game of prisoner's dilemma, "defect" is a dominant strategy and "cooperate" is a strictly dominated strategy for both players. To understand the idea of a dominant strategy, let us try to understand the following example.
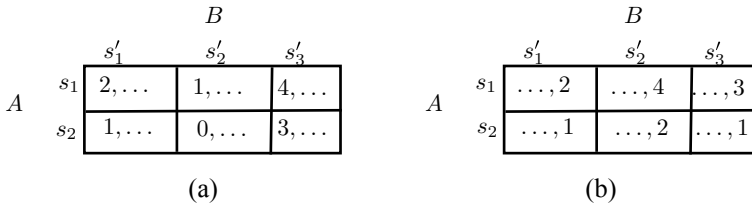
**Fig. 11.4** Dominant strategies

**Example 11.2** Dominant Strategies.

Consider the strategies $s_1$, $s_2$ and $s_1'$, $s_2'$, $s_3'$ for players $A$, $B$, respectively, for a two-player game shown in Fig. 11.4a, b.

We note from (a) that strategy $s_1$ always provide a higher payoff to player $A$, than strategy $s_2$, it is irrespective of what strategy is chosen by player $B$. Hence, $s_1$ is dominant strategy for player $A$. If we focus to player $B$ (Fig. 11.4b), $s_2'$ is a dominant strategy for it, no matter what the player $A$ chooses. Thus we predict that the outcome of the game will have each of the players choosing their dominant strategy, so the outcome is $(s_1, s_2')$.                                                                               □

It is not necessary that a dominant strategy should exist in a game. However, if it exists, it greatly simplifies the choices the players should make, helps in the determination of the outcome of the game. A rational player would never choose to play with a strictly dominating strategy; hence these strategies can be eliminated from any consideration. In case of a prisoner's dilemma, when strictly dominating strategy "cooperate" is eliminated from the possible choices of both players, there remains only (defect, defect) as the possible outcome of this game. However, if a strictly dominant strategy exists, it will remain the only choice for a player. This scheme is called *modified prisoner's dilemma game*.

Given a set of strategies, say, $s_{-i}$ chosen by the other players, a strategy $s_i$ maximizing player $i$'s pay off is called the best response of player $i$. Formally, in an $n$-player game, the *best response function $R_i$* of player $i$ is expressed as,

$$R_i(s_i) = argmax \ p_i(s_i, s_{-i}). \tag{11.5}$$

In the above equation, $p_i(s_i, s_{-i})$ is the payoff function of player $i$ for the strategy profile $(s_i, s_{-i})$.

By simply eliminating the strictly dominating strategies in a *modified prisoner's dilemma*, it is always not possible to predict the final outcome of the game that, however, may simplify the solution. Note that, in such a case we may find only the (*defect, defect*) strategy. This is possible because many a time in a game, there are no dominated strategies! Hence, there is a need for a stronger solution concept—a theory that constructs a notion of *equilibrium* state to which a complex chain of thinking finally converges. In that case, the strategies of all players will be mutually

consistent, i.e., each player will choose his/her best response against the choices of
other players. For such a theory to work, the equilibrium state must exist. We will
discuss this in the Nash arbitration section.

## 11.7    Games of Perfect Information

An important question in game theory is whether all participants in a game are equally
updated to the progressive information about the game, and even in the presence of
any external factors that might affect the pay offs or outcomes of the game. In the
case of "prisoner's dilemma," the common-sense knowledge rules form the game
rules, and these rules are known to everyone. Such games are the games of *complete
information*.

On the other hand, in a game of *incomplete* information, at least one player is
uncertain about other player's pay off function. For example, a sealed bid quotation
of a software by many bidders for government purchase, is a game of *incomplete*
information, because a bidder does not know how much other bidders have quoted. In
other words, in a game of incomplete information, the bidders' pay off functions
are not common knowledge. In game theory these situations are called *static games*.

In contrast to static games, *dynamic games* have multiple stages, such as in the
game of *chess* or *bargaining* while buying. The static games are also called *simulta-
neous move games*, due to the way they are played. However, this simultaneity is not
reflected in the actual sequence of the moves in the game, but it is in respect of the
information available to the players,[2] while they make their choices. For example, in
a case of sealed bid tender, the process of receiving bids may run for hours or days,
and the participants are free to submit their bids any day during that week (in online
it is all the hours of the day!), subject to some deadline time and date. Such bids
are still called simultaneous move games because the bidders are not affected due to
others' actions while they are choosing their own actions.

In the above discussions, the word *strategy* may be taken as a complete action plan
for all possible ways the game can proceed. For example, a player could delegate the
strategy to its representative, or write a computer code so that the representative or
computer would know exactly what to play in all possible situations, every time the
player is supposed to make a move [11].

## 11.8    Games of Imperfect Information

The simplest two-person zero-sum games of imperfect information has saddle points.
Such games have predetermined outcomes provided it is rational play. The predeter-
mined outcome is called the value of the game. Consider the table given in Fig. 11.5.

---

[2]This information is helpful to players to choose alternative moves.

The two campaigning political parties, *A* and *B* must each decide how to handle a disputed issue in a village. They can support it, oppose it, or evade it. Each party must make a decision without knowing what its rivals will do. Each pair of decision $(x, y)$, where $x$, $y$ either of three decisions by party *A* and *B*, determine the percentage of votes that each party receives in this village. Each party wants to maximize its own percentage of the vote. The numbers in the matrix in Fig. 11.5 indicate the percentage share of votes for party *A*, the balance (100 min this value) is that of party *B*. Suppose party *A* oppose the issue and *B* supports it, then *A* gets 80% and *B* gets 20%.

The *A*'s decision seems difficult at first because it depends upon *B*'s strategy. The player *A* does best to support if *B* evades; *A* does best to oppose if *B* supports; and *A* does best to evade if *B* evades it. We note that party *A* must consider the *B*'s strategy before deciding its own strategy. Whatever *A* decides, *B* gets the largest share by opposing it. Interestingly, once this is realized by *A*, he prefers to settle for 30% ! This 30%:70% votes for *A*:*B* is called game's *saddle point* [11].

A better way of finding the saddle point is to determine the *maximin* and *minimax* values. Using this method, first *A* determines the minimum percentage of votes it can obtain for each of its strategies, then selects the maximum of these three. This can be shown by representation in the form of a tree (see Fig. 11.6), where *B*'s decisions are nodes of a sub-tree, and its root is *A*'s decision. The minimum percentage *A* will get if it supports, oppose, and evade are: 20, 25, 30, respectively. The largest of these,

Party *B*

|  | support | oppose | evade |
|---|---|---|---|
| support | 60 | 20 | 90 |
| oppose | 80 | 25 | 75 |
| evade | 35 | 30 | 40 |

Party *A* (row labels: support, oppose, evade)

**Fig. 11.5**  Imperfect information game
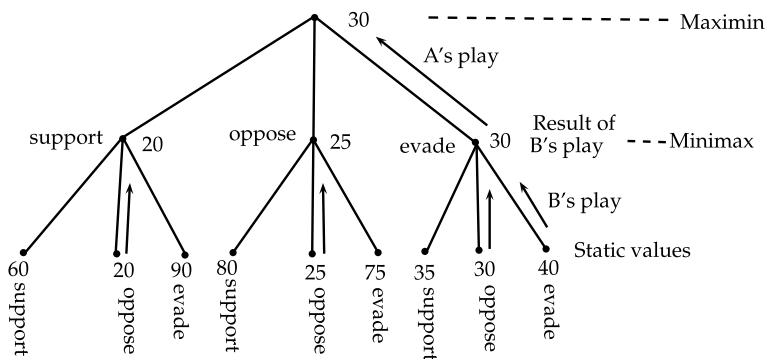


**Fig. 11.6**  Selection of maximin and minimax

i.e. 30, is maximin (maximum of the minimums) value. Similarly, each strategy the *B* chooses, it determines the maximum percentage of votes *A* can win, i.e. minimum *B* can win (similar to what we did for *A*). In this case, if *B* supports, oppose, and evade, the maximum *A* gets is: 80, 30, 90, respectively. Now, the *B* obtains its highest percentage by minimizing *A*'s maximum percentage of the vote. Here, the smallest of *A*'s maximum is 30. This 30 is *B*'s minimax (minimum of maximums) value. Because both the maximin and minimax are 30, hence 30 is a saddle point. This selection process is shown in Fig. 11.6.

## 11.9   Nash Arbitration Scheme

The player's in a bargaining game often want to make the most favorable agreement that they can, and make all the attempts to avoid any risk of losing the agreement. For example, a seller does not want to lose the customer, but at the same time keeping the maximum profit as far as possible. Also, the customer would like to buy it at the best price. A similar situations occur when one of the two fighting nations tries to arrive at peace. In the above situations, the problem is to reach arbitration that somehow reflects the strengths of the players, so that you get the effects of negotiation without risk. The Nash arbitration suggests the following procedure [10]:

> Assume that two parties are in the process of negotiating a contract. The parties may be two nations, seller-customer, management vs. labor, etc. Also, assume that failure to agree, i.e., no trade, no sale, or labor strike to go on, would have *utility* of zero to both parties. Nash then selects a single arbitrated outcome from all agreements that are within the power of the players, such that the outcomes of the product of the players' utilities is maximized.

The four properties for such outcomes are the following:

1. The arbitrated outcome should be independent of the utility function, and should depend only on the preferences of the players.
2. The arbitrated outcome should be *Pareto optimal*, i.e., there should not be any other outcome in which both the players simultaneously do better.
3. The arbitrated outcome should be independent of irrelevant alternatives.
4. In symmetric games, the arbitrated outcome has the same utility for both players. That is, in a situation if player *A* has utility *x* and *B* has utility *y*, then there exists also an outcome *y* for *A* and *x* for *B* player.

**Definition 11.3** (*Pareto domination*) An outcome '*A*' of a game is considered as Pareto dominating some other outcome '*B*' in a game, if at least one player in the game is better off in outcome '*A*' and no other player is worse off.

The disadvantage of Nash is that before the Nash arbitration is applied, the utility function of both the players must be known. In practice, the utility functions, are often misrepresented by the player to turn them to their advantage. Further, the Nash scheme is neither predictable nor enforceable. It is rather a priori agreement obtained by abstracting away many relevant factors, such as bargaining strengths of players $A$ and $B$, cultural norms, etc. It is often found that Nash arbitration often happens unfair, the poor become poorer and the rich as richer.

The idea of a Nash equilibrium (NE) in a game played by two or more players is based on the fundamental principle of a system at rest.

**Definition 11.4** (*Nash equilibrium*) A Nash equilibrium is a set of strategies for all $n$ players in a game, such that every player is playing a best response to every other player's best response. Formally, an NE is a combined strategy $s^*$, as a consequence of all players strategies $s_1, \ldots, s_n$, of $n$-players, expressed as,

$$s^* = (s_1^*;\ s_2^*;\ \ldots;\ s_i^*;\ \ldots;\ s_n^*). \tag{11.6}$$

In other words, the NE is a pay off of strategies $(s_i^*, s_{-i}^*)$ such that each player's strategy $(s_i)$ is an optimal response to the other players' strategy $(s_{-i})$, expressed as:

$$p_i(s_i^*, s_{-i}^*) \geq p_i(s_i'^*, s_{-i}^*). \tag{11.7}$$

**Example 11.3** Nash Equilibrium.

Consider that a rich man and a poor man get one million dollars if they agree on how they share it between them. If they fail to agree, they get nothing. In this case, the Nash equilibrium will give more money to the richer person [9].

When more money is received than a person already possess—people play it safe. Most, unless they are very rich, prefer to sure of thousand dollars against a chance of a million dollars But a large insurance company would even prefer the chance of one million losings and in fact,gladly accepts much less attractive risks every day. This indifference is reflected in the poor man's utility function more strongly than in the rich man's utility function. A utility function that correctly reflects the poor man's situation would be square-root function: $100 would be 10 tiles, $16 as four tiles, and $1 as one tile only. Thus, the poor man would be indifferent on a chance of $10,000 and sure of $2000. □

The general approach for solving imperfect-information games is shown in the Fig. 11.7. To begin with, the original game is abstracted to generate a similar but smaller game. The size of the game is reduced to the extent that the equilibrium-finding algorithm can be easily applied on this. In the next step, the abstract game is solved for equilibrium or near-equilibrium state.

A Nash equilibrium is based on the notion of rational play, which is a profile of strategies of all the players, with one strategy per player, such that no player can increase or decrease the payoff by switching to a different strategy. The strategy for a player means the characteristic of the play carried out when his/her turn comes.
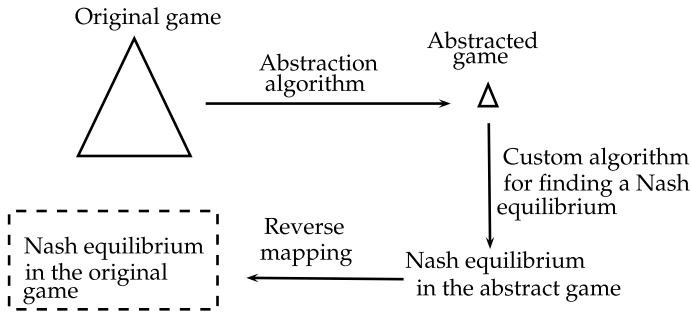
**Fig. 11.7** A general approach to solve imperfect-information games

For each information set (i.e., collection of game states), it is the probability with which the player selects each of his available actions. Due to the private nature of the information of other players, an information set cannot be distinguished by any player as whose turn is it. At the end of the game, the strategies from the abstract game are mapped back to the original game [13].

There are two kinds of abstract games: (1) *Information abstraction*, where it is assumed that a player does not know some information—that he knows, due to the reason that information sets are bundled, and, (2) *Lossless abstraction algorithm*, which results to an abstract game, and each equilibrium from this abstract game is also equilibrium in the original game.

## 11.10    *n*-Person Games

Theoretically, *n*-person game where players do not communicate, is in no way different from two-person games. As per the von Neumann and Morgenstern theory, in such games coalitions can be formed by joining together various subgroup of players, where each of them has a value associated with them—a minimum amount the coalition can attain. Usually, such subgroups may exist in a legislative body or in merging business groups. These *n*-person games are described in characteristic function form—first listing the players, then various coalitions, and then the value of these coalitions. The value of coalition function is additive, and in any given game there are many solutions.

Equilibrium Points in *n*-person Games

We can formally define the concept of an *n*-person game follows: each player in the game has associated with it a finite set of strategies, called *pure strategies*, and there is a well-defined set of payments to *n*-players. The set of pure strategies is a *n*-tuple of strategies, one for each player. The mixed strategies are nothing but the probability distributions over the pure strategies, for which the payoff functions are the expectations of the players.

For *n*-players, a point in product space is representation of any *n*-tuple of strategies, which is obtained by multiplying *n*-strategy spaces of the players. One such *n*-tuple (say, $t_1$) counters another *n*-tuple (say, $t_2$), if the strategy of each player in $t_1$ results to highest possible expectation for its players against the $n - 1$ strategies of the players in $t_2$. An *equilibrium* point occurs when an *n*-tuple is countering itself.

A correspondence of each of the *n*-tuple $t_i$ ($i = 1 \ldots n$), with its set of all countering *n*-tuples, say, $t_{ij}$, provides a one-to-many mapping of the product space into itself. By definition of countering it is clear that a set of countering points $t_j$ of a point $t_i$ is convex. Assuming that the payoff function is continuous, the mapping graph will be a closed one. This closedness is formally defined as follows.

**Definition 11.5** (*Closed-mapping*) If $p_1$, $p_2$, ..., $p_i$, ... and $q_1$, $q_2$, ..., $q_i$, ... are sequences of points in the product space, such that $q_i \rightarrow q$ and $p_i \rightarrow p$, and $q_i$ counters $p_i$, then $q$ counters $p$. Such a mapping is called closed mapping.   $\square$

In the above discussions, because the graph is closed and the image of each point under the mapping is convex, the mapping has a fixed point in its image. Therefore, there is an equilibrium point.

In the case of *two-person zero-sum* game, the "main theorem" and the "existence of an equilibrium point", are two equivalent statements. In such case, any two equilibrium points ultimately lead to the same expectation for the players. However, its occurrence is not always necessary [5].

## 11.11   Representation of Two-Player Games

Chess has certain attractive features that many, more complex tasks do not have. The available options (moves) and goal (checkmate) are sharply defined. The discrete model of chess is called a game-tree, and it is the general mathematical model on which the theory of two-player zero-sum games of perfect information is based. Since all the legal moves are known to both the players at all the times, therefore this is a game of perfect information. This also is a game of zero-sum, because when one player looses, the other gains it.

There is a root node at the top of a chess tree, that represents the initial setup (configuration) on the board, say $c_0$. Corresponding to each opening move, there is an arc leading to another node that represents a new board configuration (say, $c_{0i}$) after this move is made. If there are *n* number of such maximum configurations to which a legal move can be taken from the root node, then the nodes corresponding to these configurations are $c_{01} \ldots c_{0n}$. This process of construction of news configurations or new nodes goes on as we progress in playing the chess game. Thus, constructing a game-tree—a recursive structure of the tree, with the root node and sub-trees under that, with arcs pointing to the potential next states, such that each of the game-trees would be a smaller game-tree. The number of arcs/edges leaving a node is referred to as *branching factor*, and distance of any node from the root, in terms of a number of in-between nodes, is called *depth*. If *b* and *d* are the average branching factor and

depth of the tree, respectively, then the tree contains at least $b^d$ nodes. When the current state of the game is a leaf node, the game terminates.

Each leaf node has a value associated with it, that corresponds to the payoff of that particular outcome. A game can have any payoff associated with each outcome, which helps in deciding to choose the move of higher payoffs. However, the standard parlor games have result like *win*, *loss*, or some times *draw*.

In the case of two-player games, the players take alternate turns to make a next move, which chooses a state from among the children states of the current state. If the two-player game is a *zero-sum game*, one player attempts to choose a move of maximum value, and other players would always choose a move of minimum value. An approach that tells a player what to choose next, is called the strategy of the game. The chosen strategy (called control strategy) controls the flow of the game. In principle, the decision to choose which node should be the next node is a simple one—any state that is one move from the leaf node can be assigned a value which is best of its children—this can be either maximum or minimum value of the children under the node.

This maximum or minimum depends on which player out of the two is playing. The states that are two moves away from the leaf then take on the value of their best children. This goes on until each child node of the current node is assigned a value. The best move is chosen based on this strategy. In the two-player zero-sum game, the player that maximizes is called *maximizer*, and the one minimizing is called *minimizer*. Accordingly, the method of assigning values and choosing moves is called the *minimax algorithm*, and it defines the optimal move to be made from each state in the game.

Complexities

The network of configurations and move (links) come up as a search tree, called *game-tree*. The start node is at level 0, and other nodes levels from the top are at levels $1, 2, \ldots, d$ for a game-tree of depth $d$. If $b$ is the maximum branching factor, the worst-case time and space complexities for breadth-first search (BFS) are both $O(b^d)$. For depth-first search (DFS), time and space complexities are $O(b^d)$, $O(d)$, respectively.

## 11.12   Minimax Search

The Minimax search is based on *minimax theorem*, which is the fundamental theorem of game theory, stated in 1928 by mathematician John von Neumann.

**Theorem 11.1**  *Minimax Theorem.*

The minimax theorem says that every two-person zero-sum game of a finite number of states has a solution in mixed strategies. If $A$ and $B$ are the players in such game, there is a value $V$ and mixed strategies for these players, such that if player $A$ adopts
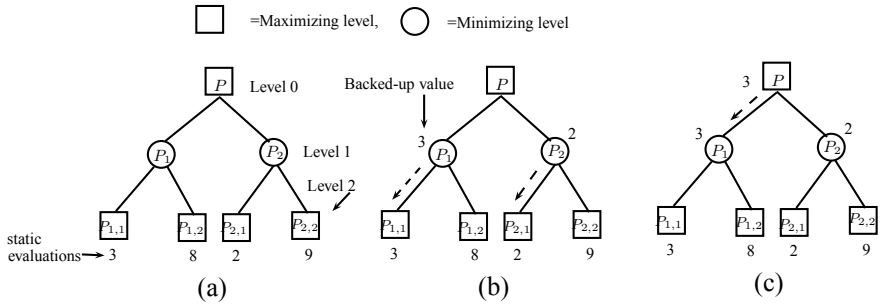
**Fig. 11.8**  MINIMAX search

its mixed strategy, the outcome will be at least as favorable to $A$ as $V$. Whereas, if $B$ adopts this mixed strategy, the outcome will be no more favorable to $A$ than $V$. Thus $A$ and $B$ have the power to enforce the outcome of $V$.                        ☐

**Definition 11.6**  Utility Theory.

In some of the previous examples, we discussed that players want to maximize their profit. But in real-life situations, the game players have some other goals. The examples are: buying lottery tickets, playing gambling in casinos, and buying so and so insurance policies. The game theory states, how to attain these goals. Hence, it is important to define a *utility function* that would reflect an individual's preferences while playing/making moves. Basically, the utility function assigns to each of the player's alternatives (i.e., moves), a number, that convey the relative attractiveness of the alternative. Maximizing the player's utility automatically determines his most preferred action. This is called Utility theory.                                ☐

Consider the Fig. 11.8, where the two-players (*maximizer* and *minimizer*) play alternately. The game has static values at the lowest level, i.e., at the bottom of the tree. The game-tree is shown with three levels 0 (for root), 1, and 2. The maximizer chooses move at level 0, minimizer at level 1, and at the next level, i.e., 2, the static values are available. The maximizer might hope to get to the situation yielding score 9, he knows that minimizer might choose a move deflecting the play towards the score 2. In general, the maximizer (here at level 0) must take the choices available to the minimizer at the next level, into cognizance. And, similarly, the minimizer must also take into cognizance the choices available to maximizer at the next level down [1].

Eventually, the limits of exploration are reached and the static evaluator provides the direct basis for selecting among the alternatives. The minimizer may choose the values 3 and 2, at the level just up from the static evaluations (Fig. 11.8b). Knowing these scores the maximizer at level one up makes the best choice between 2 and 3, at level zero (see Fig. 11.8c).

The procedure by which the scoring information passes up the game-tree is called MINIMAX algorithm because the score at any level $i$ is either minimum or the maximum of what is available at the level $i + 1$. The recursive form of the MINIMAX

algorithm is shown as Algorithm 11.1. The idea of *minimaxing* is to translate the board into a *static number*. However, the process may be expensive due to the generation of a large number of paths.

---
**Algorithm 11.1** MINIMAX Algorithm
---
1: **if** limit of search has reached **then**
2:     compute the static value of the current position relative to the appropriate player;
3:     Report the result;
4: **else**
5:     **if** level is minimizing level **then**
6:         call MINIMAX Algorithm on the children of current position;
7:         Report minimum of the result;
8:     **else**
9:         (level is maximizing level.)
10:         call MINIMAX Algorithm on the children of current position;
11:         Report maximum of the result;
12:     **end if**
13: **end if**
---

The *minimax* algorithm has the following properties [14]:

1. The algorithm (i.e., inference) is *complete* if tree is finite.
2. The worst-case Time complexity is $O(b^d)$, where $b$ is branching factor, and $d$ is total distance.
3. The worst-case Space complexity is $O(d)$ equal to depth-first exploration.

**Example 11.4** Grundy's game.

Consider a typical player game, called *Grundy's* game, where there is a stack of seven coins of equal size, indicated by initial configuration as (7). The starting player MAX makes a move so that the stack is broken into two stacks of unequal size. Figure 11.9 shows the alternate moves as: (6, 1), (5, 2), (4, 3). The other player, MIN, has the next move, which breaks the sub-stacks created further into unequal parts. This process goes on for alternate players. No Further move is allowed from a node if unequal partitioning is no more possible from that node. The player who first cannot play the next move is game the loser.

We note that for the node (2, 2, 1, 1, 1) MIN is not able to make the next move, hence MIN loses. The player that makes the first move is generally a MAX player. Thus after this move, the node positions occupied are called MAX nodes, and from these nodes, the move is due to MIN player. This leads to the transition to MIN-MAX nodes.

At every configuration, there are choices available either for MAX or MIN player for the next move. For example, MAX will try to foresee the possible next moves available to MIN after he has made a move, and accordingly chooses this next move so the MIN cannot have a good move, and he is more close to a winning strategy. The MIN also keeps a similar strategy. Finally, the game ends at a node where no further move is possible by any of the players.                                    □
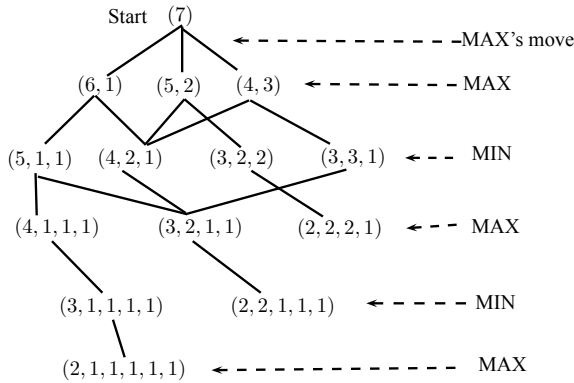
**Fig. 11.9** MINIMAX game-tree

## 11.13 Tic-tac-toe Game Analysis

The tic-tac-toe is a board game, with $3 \times 3 = 9$ positions, and played between two-players through alternate moves. The game has a total 765 different positions, and 26,830 possible games. The initial configuration is the empty board. The one player, called MAX makes the first move with $X$, and the other player, called MIN player, makes a $O$ move. The player which is first in placing all $X$ or $O$ array as row or column or diagonal wins the game [8].

Figure 11.10 shows some initial configuration and of some moves of this game.

The game of tic-tac-toe generates all the possible moves at level $i$ before generating the moves of level $i + 1$, hence, it is breadth-first search (BFS). We note that while
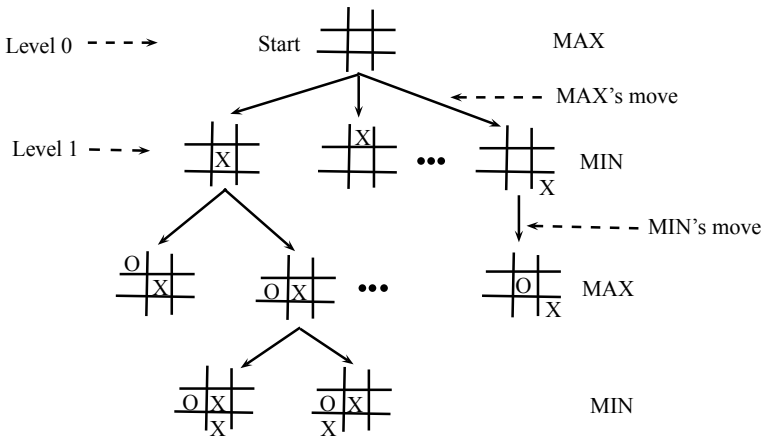


**Fig. 11.10** Tic-tac-toe with some initial moves

**Fig. 11.11** A tic-tac-toe
game
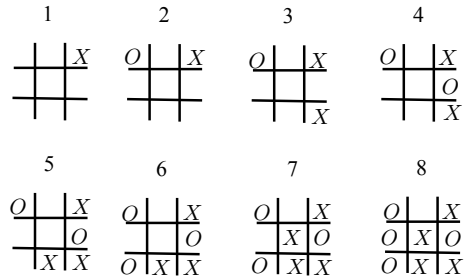


in start configuration (level 0) having all empty positions, the player MAX can make
moves to level 1. The player MAX will choose moves in such a way that it leads
to a configuration ultimately such that all the $X$'s are continuous and are in a line,
or column or diagonal. At the same time, the MAX will choose such a move that it
becomes difficult for MIN later to make a move to a configuration with all $O$'s in a
line, or column or diagonal. Thus, each of MAX and MIN makes the moves so that
it is winning for them and blocking for the other player.

The Fig. 11.11 shows total of 8 moves of this game, for a particular sequence of
moves by alternate players $X$ and $O$, numbered as 1–8 Finally, the player $O$ wins
the game.

To decide which configuration is superior to move into, we associate a weight
function that will be maximized by the MAX player, and the MIN player will try to
minimize it. If a configuration is indicated by $c$, then this weight function is defined
as:

$$f(c) = m - n, \tag{11.8}$$

where, $m$ = number of complete rows, columns, and diagonals that are still open
(fully) for the next move of MAX, and

$n$ = number of complete rows, columns, and diagonals that are still open for the
next move of MIN.

If $c$ is a winning configuration for MAX, then $f(c) = \infty$, and if $c$ is winning
configuration for MIN then $f(c) = -\infty$. If a configuration is like in Fig. 11.12, then
$f(c) = 4 - 6 = -2$. The $m$ are solid arrows (Fig. 11.12a), and $n$ are shown as dotted
arrows (Fig. 11.12b).

**Fig. 11.12** Computing the
weight function
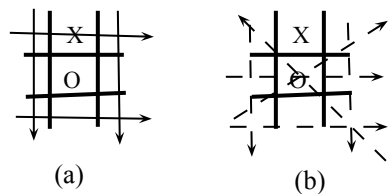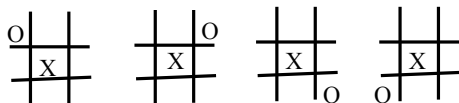
**Fig. 11.13**   Identical configurations in tic-tac-toe

If we make use of symmetries in generating successor positions, then all the game states shown in Fig. 11.13 are identical.

For efficiency reasons, early in the game, the branching factor is kept small by symmetries, and late in the game, it is kept small by a number of open positions available. The first move is always important.

**Example 11.5**   Computation of static values in tic-tac-toe game.

The Fig. 11.14 demonstrates the computation of this weight function at each of the node. To win the game, the MAX at level 0, should make move to such a node at level 1 that produces a maximum value of function $f(c)$ at level 1. It is this maximum value that is backed up to level 0. This value is as a result of the maximum value of the corresponding nodes that are under the MAX node at level say 1, (in general, for MAX at level $i$ it backup the maximum of values those at level $i + 1$). The MIN node at level $i + 1$ should make a move that produces minimum value from those at level $i + 2$. This process goes on until the *maximizer* sees the *static values* at the last level (bottom) of the tree. Similarly, the *minimizer* will also, choose the next move by attempting to see the static values at the last node.
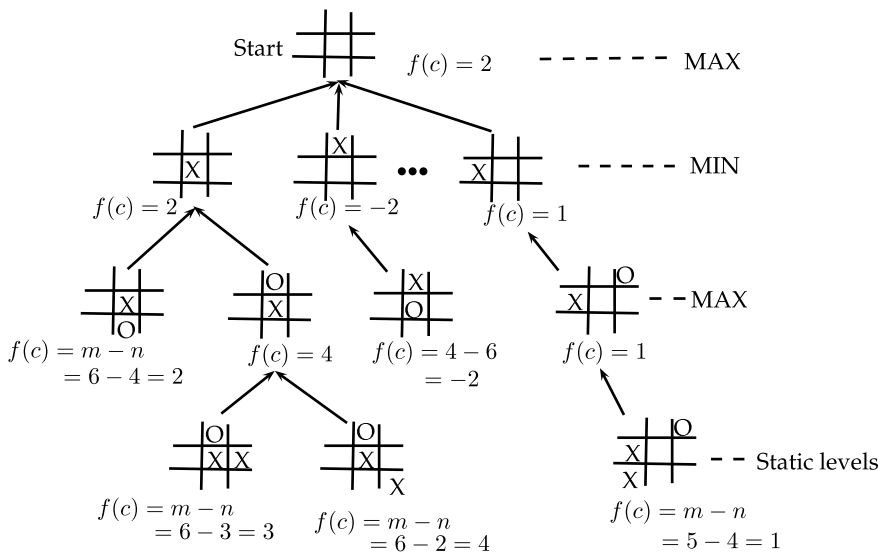


**Fig. 11.14**   Tic-tac-toe with backup of static values

The static values at the bottom nodes in this example are shown as 1, 3, 4, 0, −1. The values at the upper levels, i.e, parents of static nodes, and their parents are called *backup values.* The backup values are based on the "look-ahead" of the game-tree, and this depends on the lower levels nodes, in fact even those that are near the end of the game-tree.

Some of the nodes may represent a win for MIN, hence they may have value −∞. When evaluations are backed-up, the MAX's best move is that which avoids its immediate defeat. In the case of MAX, the win corresponds to a node value of +∞.                                                                                                   □

## 11.14   Alpha-Beta Search

Alpha-Beta is a game-tree search, which is an improvement over the minimax procedure, but equivalent to minimax in some respect—both the procedures will always choose the same depth successor at best, and they will assign the same value to it. The Alpha-beta method turns out to be several orders magnitude faster than minimax—it saves the time by not searching certain branches of three, which are not fit for any contribution for winning strategy of the game. This is because under certain conditions, the values of certain branches do not affect the value(s) which is ultimately backed-up[3] to higher levels of the tree. Therefore, these branches are not required to be evaluated. When the alpha-beta procedure detects these conditions, it stops work on one branch and skips to another. This event is called an alpha or beta cutoff [12].

At first, it appears that *static evaluator* must be used on each leaf node at the bottom of the search tree. But, it is not required. To see how alpha-beta search works, consider the example shown in Fig. 11.15. Alpha-beta starts just like a minimax procedure by evaluating all the successors of $P_1$. The minimum of these static values is then backed-up to $P_1$, since $P_1$ is a MIN position. The backed-up value of $P_1$ is *alpha* and has a value 5 in this example.

Alpha is the lower limit for the backed-up value of the top position, $P$. Since, $P$ is the max position, we backup the value of the largest value successor of $P$. Since we have evaluated only one successor at this time, we do not know that it will be 5 or larger. The value of alpha may change as the other successors are evaluated, but it can only increase, not decrease.

Having evaluated $P_1$, the procedure begin work on $P_2$. An *Alpha cutoff* takes place at $P_{2,1} = 3$, as it is less than alpha. Since $P_2$ is a Min position, $V_{2,1}$ is an upper limit for $V_2$ (value of node $P_2$). Since $V_2$ is less than alpha, $P_2$ is definitely eliminated as a candidate for the largest valued successor of $P$. There is no point in evaluating the other successors of $P_2$ so the procedure begins work on $P_3$ next.

The above cutoffs save the machine a good deal of time. The alpha cutoff at $P_{2,1}$ means that the machine need not bother to evaluate $P_{2,2}$ and $P_{2,3}$. A second alpha

---

[3]Selection of minimum of the payoffs or maximum of the payoffs (depending on which player has a move) of children of any node $v_k$ and moving it to node $v_k$, is called *backing-up* of values.
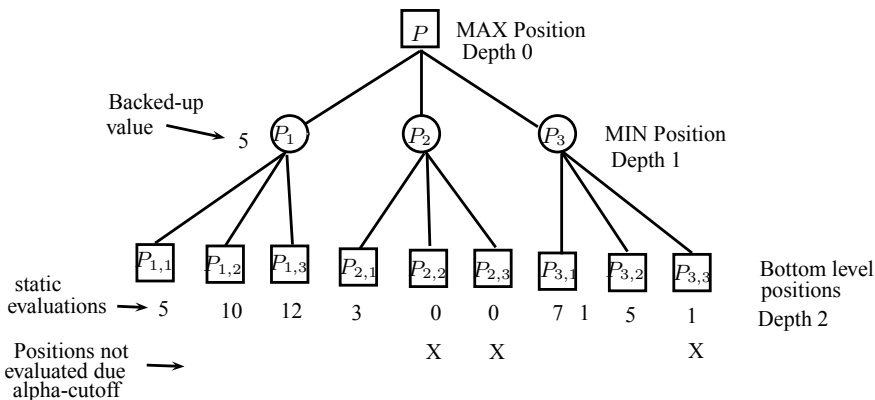
**Fig. 11.15** Alpha-beta search
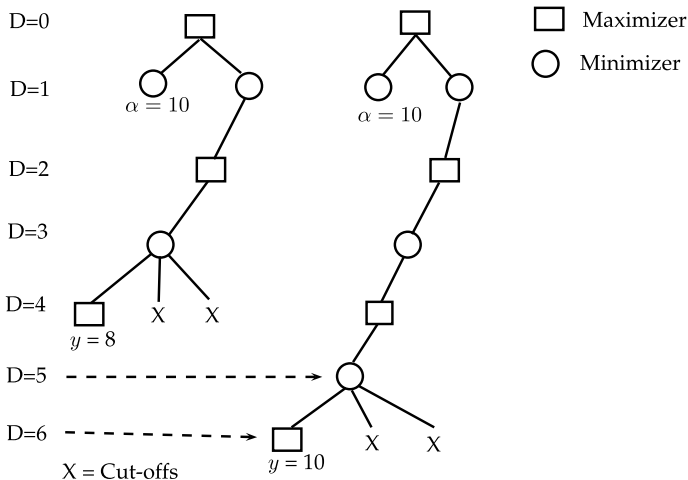


**Fig. 11.16** Deep alpha cutoffs

cutoff occurs at $P_{3,2}$, which eliminates $P_{3,3}$. Thus, in the Fig. 11.15 alpha-beta search program would evaluate only six of the bottom level successors, while a minimax program would evaluate all nine.

Although the example is given only for a tree of three levels, it is clear that the procedure will work just the same below any Max position $P_x$, at any depth in a large tree, provided only that there are at least two levels below $P_x$. If there were more levels below $P_{1,1}$ in the example, then we could use the backup value of $P_{1,1}$ instead of the static value. If there were more levels above $P$, then we could backup of $P$.

Moreover, it is possible to pass a value of alpha down from the top of a large tree. Thus, an alpha established at depth 1 could be used to produce cutoffs at depths 2, 4, and 6. These deep cutoffs are illustrated in Fig. 11.16.

Alpha is defined by the values of the successors of a Max positions (i.e., odd depths, here it is $P_1$, $P_2$, $P_3$ nodes, in Fig. 11.15), while alpha cutoff occur among the successors of a Min position (even depths, here it is $P_{1,1} \ldots P_{3,3}$, in Fig. 11.15). It is possible to define another variable, *Beta*, which is established at even depths and generates cutoffs at odd depths. The action of the beta cutoff is exactly inverse of alpha cutoffs.

The effect of alpha-beta cutoff is to make the tree space grow slower with depth. Thus, the advantage over simple MINIMAX. It is about twice as good as at Maximum depth Dmax = 3 and about thirty times as good at Dmax = 6. This is typical. It is good to have depth-dependent program. Such a measure is called as depth relation (DR),

$$DR = \frac{\log N}{\log N_{MM}} \tag{11.9}$$

where, $N$ is a number of nodes at the bottom of the tree, and $N_{MM}$ is the number of nodes at the bottom of the tree in minimax search. $DR = [0, 1]$, is effective depth in comparison to minimax procedure.

Alpha-beta search is equivalent to minimax in the sense the two procedures will choose the same depth successor as best and will always give the same value for the successor.

### 11.14.1   Complexities Analysis of Alpha-Beta

The benefit of alpha-beta pruning lies in the fact that branches of search tree can be eliminated so that search tree can be limited to more promising sub-trees and deeper search can be performed in lesser time. Thus it belongs to the *branch-and-bound* class of algorithms.

If move ordering for alpha-beta is optimal (best moves are always selected first), then the total number of leaf nodes to be evaluated are $O(b^{\frac{d}{2}}) = O(\sqrt{b^d})$. The explanation of above is that all the first players' moves must be studied to find the best one, but for each, only the best second player's move is needed to refute all but the first (and best) player move-alphabet ensures that no second player moves need to be considered. Thus, the complexity ($O(b^{\frac{d}{2}})$ is possible if nodes are ordered. If they are at random then it is $O(b^{\frac{3d}{4}})$.

Thus, it can be demonstrated that the number of static evaluations needed to discover the best move in an optimally arranged tree is $O(b^{d/2})$. Let the static evaluator function be represented by a variable $c$, which needs to be maximized and minimized at different levels. We note in Fig. 11.15 that evaluation of $c$ is not required at level 1 (shown with values 5 and $\leq 3$ for minimizing level). This is because we do not need to evaluate for $\leq 3$, as the value 5 is carried forward to the root. Since static evaluations are required only at the alternate levels, the depth of search reduces to half. Therefore the time *time complexity* is $O(b^{d/2})$.

### *11.14.2 Improving the Efficiency of Alpha-Beta*

The number of cutoffs generated by alpha-beta procedure depends on the order in which successors are evaluated. In Fig. 11.15, if the machine had evaluated $P_2$, $P_3$ then $P_1$, alpha would have been 0, then 1, then 5, and there would have been no alpha cutoff.

This fact suggests the possibility of improving on the alpha-beta procedure by ordering successors of a position in order to generate a large number of alpha-beta cutoffs.

The ordering can be achieved by ordering the nodes of a search tree at their static values. The largest value successor of a max position is put first and reverse order is used for the successors of min position. This procedure is based on the assumption that their static value of a position is positively correlated with the deeper, backup value at that position. Since ordering obtained may not be correct at other levels, dynamic ordering may be adopted.

Consider the symmetric configurations in Fig. 11.17. We note that the cost function in all these is equal. Thus, if these configurations are merged in the alpha-beta search, or even in the *minimax* search, the number of nodes in the game-tree can be substantially reduced. Similarly, many other symmetric configurations exist, and those, similar, can be merged. All this helps in boosting the efficiency of the alpha-beta search.

The common techniques for assignments of tip values to the game-trees are based on probabilistic assumptions, simulations, and closed form results. Though the alpha-beta search is considered as optimal *minimax* algorithm under sequential traversal of nodes of the game-tree, it can be far efficient if game-trees are traversed in parallel. The problem, if any with sequential alpha-beta search occurs when the optimal path moves towards the right of the tree. The later is because the alpha-beta search actually explores the game-tree from left to right. Since a parallel alpha-beta search can explore multiple paths simultaneously in all the regions, it obtains a better global perspective of the tree and can cut off the search in the regions where sequential alpha-beta cannot do. A parallel search algorithm can perform a parallel tree-traversal using state-space search.
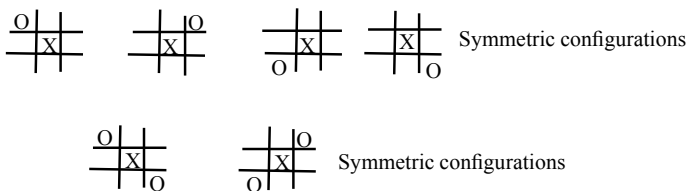


**Fig. 11.17** Symmetric configurations

## 11.15   Sponsored Search

Due to the prevalence of more and more business, commerce, and trading activities on the WWW, the sponsored search is becoming very important. The sponsored search is based on input from many resources, like Web-search engines, users, and content providers. Here, contents providers are companies or individuals owning the Website hosting the contents.

The keywords selected by the contents provider are matched against the users' query by the web search engines, and the corresponding link is displayed in highlighted form. The content providers pay to the search engine company in many cases, whenever a user enters one of these terms for search, and clicks on the *sponsored link* displayed. On occasions, the user is required to go one step further and perform some specified action on a web-page on the content provider's website. The content providers can identify those phrases from the user queries that are most likely to be related to the required Web sites. Often the sponsored search platforms provide the capability to the content providers to tailor the presented material on the sponsored link that conforms to user queries.

In fact, it is not necessary that these simple approaches would always work. Consider a situation, where many different content providers assign the work of responding to user queries to a search, all of them want the search engine to respond to the same term or phrase of the user query. In such a scenario, an electronic auction system will rank the sponsored links, in such a way that the highest bidder will be of the topmost rank, the next lower bidder will get the next link, and so on. That means, the one who pays more will get the links of his/her product and services listed at top, and only they will get the customers, because the customers usually do not open all the links.

The web search engines also adopt other ranking factors apart from the price paid by the bidder. For example, in the past, some sponsored link used to get much higher clicks, is likely to be displayed among the top, or at the top. This is because a link that gets more clicks, is likely to be more relevant. And, when the link is relevant, the user is potentially a good customer for the content provider. A sponsored link with maximum clicks will generally produce a major part of the profit for the search engine. Hence, both the content providers and the search engines have higher monetary incentives for providing the contents to the user which are relevant.

The key-phrase selection can be conceptually viewed as dynamic form of Web site meta-tagging focused on the user. The contents providers can change one or more of following:

   phrases to be searched,
   bid price for various phrases,
   degree of matching of terms,
   time based restrictions, like displaying at 10 AM, evening, at early morning, etc,
   geographical restrictions, and,
   amount they bid for a given period.

From the above discussions, we conclude that the content providers become active participants in the sponsored search process [3, 4].

## 11.16   **Playing Chess with Computer**

One of the well known, landmark example of the game is the chess game, played between grandmaster Garry Kasparov and IBM Deep Blue Computer in 1997. Following are some characteristics of this competition:

- The system used for this purpose comprised of 30 number of IBM RS6000 machines, running a program for search, and a large number of custom VLSI chess processors that performed the job of chess move generation, and ordering of these moves. This hardware searched for the last few levels of the game-tree, and evaluated the leaf nodes.
- Deepblue searched more than 100 million nodes per second on average, with peak speed as high as 330 million nodes per second.
- This software generated 30 million positions per move reaching a depth of 14 quite often.
- Heart of the search algorithm was a standard iterative deepening alpha-beta search.
- In some cases, the search reached a depth of 40 plies.
- The evaluation function for a proper move comprised over 8000 features, many of them described a highly specific pattern of pieces.
- A large, end-game database, which comprised solved positions, were also used.
- The chess program also used an opening book with about 4000 positions, and a database of 700,000 grand-master games for consulting for consensus for next moves.

In the above, the IBM's Deep Blue software beat the World Chess Champion in a series of six matches.

In the more recent chess playing computer games, varieties of heuristics for pruning are commonly used to reduce the effective branching factor from 35 down to less than 3, hence they are more powerful with even far less processing capabilities of processors.

## 11.17   **Summary**

The Game theory is the formal study of *conflict* and *cooperation*. Another branch of game theory, called, *combinatorial Game theory* (CGT) studies strategies and mathematics of two-player games of *perfect knowledge* such as *Chess* or *Go*. In games of *perfect information*, such as Chess, each player knows everything about the game at all times. Poker is a game of imperfect information, as other players do not know your cards. In zero-sum games, if one player loses, the other will gain; Poker, Chess, and Parlor are games in this category. Games of bargaining, trade, agreements, are non-zero-sum games, where all the players gain on successful completion of the game, and loose in the event of failure.

Strategy for two-player game is: every unique board configuration is converted into a unique single integer number. While one player tries to maximize this number, the opponent does the reverse.

*Prisoner's dilemma* is a game of conflict and cooperation; and complete information, such that each player is aware of strategies of the other player.

A dominant strategy provides roughly the highest payoff for a player, irrespective of what strategies are employed by the other players.

For the games of the bargain, *Nash arbitration scheme* is used to arrive at an agreement, called Nash equilibrium (state), which is a set of strategies for all $n$ players such that each player is playing the best response to every other player's best response. The equilibrium state follows the principle of a system at rest.

The scoring in two-player game is done using the MINIMAX algorithm, where the score at any level $i$ is either minimum or the maximum of what is available at the level $i + 1$.

Games like, tic-tac-toe, and chess requires the search, which can be conducted using the minimax search. However, this approach has exponential complexity for searching the game-tree.

An improved search method, called *alpha-beta search* approach, saves the search time by pruning the game-tree through alpha and beta cutoffs.

## Exercises

1. Demonstrate the min-max search for the tic-tac-toe puzzle (for 10 moves).
2. Demonstrate the alpha-beta search for the tic-tac-toe puzzle.
3. Apply alpha-beta search (from left to right) to the game-tree in Fig. 11.18. Show the backed-up value of each node. Mark with an X any branches that are not searched. Identify these as alpha/beta cutoffs. Mark the best move with an arrow from the root node.
   a. Identify these as alpha/beta cutoffs, mark the best moves with an arrow from the root node.
   b. Compute the time complexity of search with worst-case branching factor of 3 and height of the tree as $h$.
4. A game *nim* is played as follows: there are two players who remove one, two, or three coins, alternately from a stack of five coins. Represent this game playing as a search tree. Suggest any suitable strategy for winning this game?
5. Given the tree in Fig. 11.19, explore this tree using the alpha-beta procedure. Indicate all parts of this tree that are cutoff. Also, indicate the winning path(s), and strike out all the values that are not required to be computed.

   a. Identify these as alpha/beta cutoffs, mark the best moves with an arrow from root node.
   b. compute the time complexity of search with worst-case branching factor of 3 and height of tree as $h$.
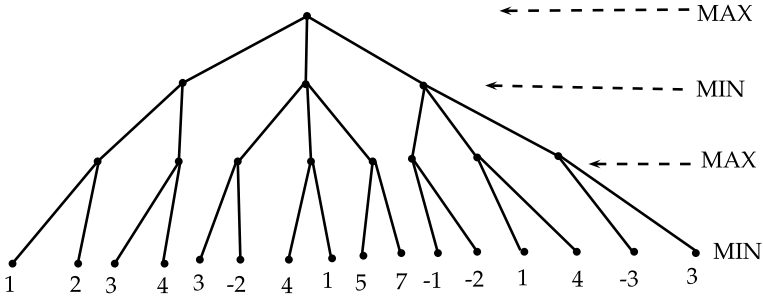
**Fig. 11.18** Game-tree

**Fig. 11.19** Alpha-beta search
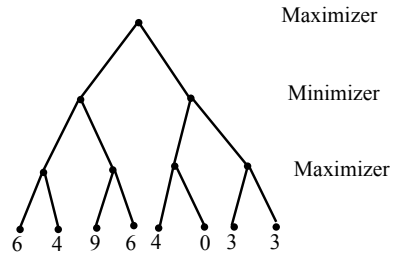


6. Consider the min-max tree shown in the Fig. 11.20, whose leaves are labeled with natural numbers; *n* and *m* are variables.

   a. Assign values to *n* and *m* such that, to compute the value at the root node, no alpha-beta cutoff is possible. Compute the value of the root node.
   b. Assign values to *n* and *m* such that, to compute the value at the root node, an alpha-beta cutoff is possible. Indicate the cut and compute the value of the root node.

7. Following are either two-player or many player games, and standard minimax/alpha-beta search techniques are to be applied for game playing. Explain, how well these techniques apply to these games? Give the formal approach for each, where possible.

   a. Basketball
   b. Badminton
   c. Football
   d. Soccer
   e. Tennis
   f. Kabbadi

8. Consider two firms *A* and *B*, that give service in the same market. The firms incur constant average costs of $2 per unit, and are free to choose a high price of $10 or a lower price of $5 per unit for marketing. When both firms set a high price, the total demand is 5,000 units which are split evenly between the two

**Fig. 11.20** Min-max tree



MAX

3    4    $n$    $m$
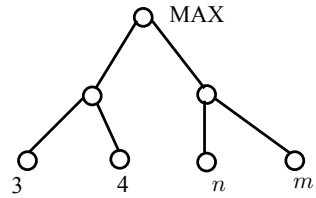
**Fig. 11.21** A two-player
game

$B$

|       | $s'_1$ | $s'_2$ | $s'_3$ |
|-------|--------|--------|--------|
| $s_1$ | $2,3$  | $6,1$  | $2,3$  |
| $s_2$ | $2,3$  | $3,4$  | $5,1$  |
| $s_3$ | $4,1$  | $5,4$  | $4,2$  |

$A$

**Table 11.1** Two-player game move choices

|       | $s'_1$  | $s'_2$  | $s'_3$  |
|-------|---------|---------|---------|
| $s_1$ | (1, 1)  | (3, 4)  | (2, 1)  |
| $s_2$ | (3, 3)  | (0, 4)  | (0, 9)  |
| $s_3$ | (2, 4)  | (2, 5)  | (8, 1)  |

firms. When both set a low price, the total demand is 10,000 units, which is again split evenly. If one firm sets a low price and the second a high price, the low priced firm sells 8,000 units, and the high priced firm only 1,000 units.
Analyze the pricing decisions of the firms $A$ and $B$ as a noncooperative game.

a. Construct the payoff matrix, where the elements of each cell of the matrix are the two firms' profits.
b. Derive the equilibrium set of strategies.
c. Is it the game of prisoners' dilemma? Justify.

9. Find out the Nash equilibrium for the game shown in Fig. 11.21, which is played between two-players $A$ and $B$, having strategies $s_1$, $s_2$, $s_3$ and $s'_1$, $s'_2$, $s'_3$, respectively.
10. Consider the Table 11.1 for two-player game.

a. Find out the maximum moves by the game to win/loose.
b. Which moves are dominated?
c. Identify all the best responses.
d. Is there any Nash equilibrium?

11. Draw a payoff matrix for the following game of scheduling a party. Also, find maximin moves, domination, best responses, and Nash equilibrium (if it exists).

**Table 11.2** Two-player perfect information game

|            | Cooperate | Defect |
|------------|-----------|--------|
| Cooperate  | (4, 2)    | (2, 6) |
| Defect     | (6, 0)    | (0, 4) |

Let the friends $A$ and $B$ do not speak to each other, but have many common friends. Both of them want to invite these friends to the party, either on Saturday or Sunday. However, both of them prefer Sunday over Saturday. If both decide the party for the same day, it will be considered as a disaster with a (negative) payoff of $500 for both. If they plan the party on different days, the one proposing Sunday gets a payoff of $250, and the other of $200.

12. For two-players game shown in Table 11.2, find out all Nash equilibria states, possible domination, maximin moves, and best responses.
13. Find out the *saddle point* for the game shown in Fig. 11.5.
14. What can be the true considerations, for example, to reach to Nash equilibrium faster in a buyer vs seller game?
15. Are we in a position to determine in advance that arbitration between two parties will result in an equilibrium state? Justify for yes/no.
16. Justify the following statement: "It is often found that Nash arbitration is often unfair, in which rich becomes richer and poor becomes poorer."
17. For any minimax game of even height game-tree, show that final results of the game shall be the same, irrespective of who plays first.
18. Apply the minimax algorithm for the problem of prisoner's dilemma (Fig. 11.2), and demonstrate the backup of values from static levels.
19. Why the game of tic-tac-toe is BFS search? Justify.
20. Write an algorithm, to compute the weight of any general configuration of tic-tac-toe given any number of $X$s and $O$s on arbitrary position. Note that if $0 \leq |X| \leq 5$ then $0 \leq |O| \leq 4$ and vice versa.
21. In a game search tree with the root node at level 0, suppose the alpha cutoff occurs at level $i$. What is the minimum required depth of the tree? Justify your answer.
22. For a game-tree of depth $d$, with branching factor $b$ uniform at all the sub-tree, let alpha cutoff occurs at every alternate depth due to the second node from left in each sub-tree. What is an expression for the time-complexity of the search using only alpha cutoff? Assume that the time-complexity of the minimax algorithm is $O(b^d)$.
23. Assume that in a game-tree, the alpha and beta cutoff occurs at every alternate level. Find out the time complexity of this search.
24. Show that if static nodes are ordered in order of their static values, then the time complexity of alpha-beta search is $O(b^{\frac{d}{2}})$.
25. Show that if static node weights are in random order, then the time complexity of alpha-beta search is $O(b^{\frac{3d}{4}})$.

26. In the Fig. 11.15, find out the alpha-beta cutoffs in each case, when order of sub-trees are $P_3 P_2 P_1$, $P_2 P_3 P_1$, and $P_2 P_1 P_3$. What you conclude by this change in order?
27. Are the winning results of minimax and alpha-beta search tree identical for a given search tree? Justify.
28. Show that cost functions of symmetric configurations in tic-tac-toe are identical.
29. Two firms Alpha and Beta serve the same market. They have constant average costs of \$2 per unit. The firms can choose either a high price (\$10) or a low price (\$5) per unit for their product. When both firms set a high price, the total demand is 10,000 units, which is split evenly between the two firms. When both set a low price, the total demand is 18,000 units, which is again split evenly. If one firm sets a low price and the other a high price, the low priced firm sells 15,000 units, while the high priced firm sells only 2,000 units. Analyze the pricing decisions of the two firms as a noncooperative game.

    a. In the scenarios mentioned above, form representation, construct the payoff matrix, where the elements of each cell of the matrix are the two firms' profits.
    b. Derive the equilibrium set of strategies.
    c. Explain why this problem an example of the prisoners' dilemma game.

30. Write a parallel search algorithm for an alpha-beta search to reduce the search space.
31. Select one or more than one, or True/False, or nil answer in the following multiple-choice questions:

    a. Prisoner's dilemma is following type(s) of game:
       (A) Noncooperative game          (B) Non-zero-sum game
       (C) Imperfect information game    (D) all the above
    b. Poker is following type(s) of game:
       (A) Zero-sum             (B) Perfect Information
       (C) Non-zero sum         (C) None of above
    c. Which of the following are static games?
       (A) Prisoner's dilemma (B) Bid of winning a contract
       (C) Chess               (D) Poker
    d. Which of the following are called simultaneous move games?
       (A) Prisoner's dilemma (B) Bid of winning a contract
       (C) Chess               (D) Poker
    e. The tic-tac-toe is following type of search:
       (A) Iterative deepening     (B) DFS
       (C) Best-first search       (D) BFS
    f. Alpha is defined by the values of successors of:
       (A) MAX position        (B) MIN position
       (C) Both MAX and MIN (D) None of above
    g. Alpha cutoff occurs among the successors of:
       (A) MIN position                (B) MAX position
       (C) Depends of the structure of the tree (D) None of above

h. Beta is defined by the values of successors of:
   (A) MAX position        (B) MIN position
   (C) Can be any position (D) None of above
i. Beta cutoff occurs among the successors of:
   (A) MIN position                        (B) MAX position
   (C) Depends of the structure of the tree (D) None of above
j. The alpha-beta search belong to the following class of algorithm:
   (A) Hill-climbing        (B) Branch-and-bound
   (C) Divide-and-conquer (D) Iterative approximation
k. Alpha is defined at odd level of the tree (T/F)?
l. Beta is defined at even level of the tree (T/F)?

# References

1. Abramson B (1989) Control strategies for two-player games. ACM Comput Surv 21(2):137–161
2. Davis MD (1983) Game theory—a nontechnical introduction. Dover, New York
3. Jansen BJ, Spink A (2007) Sponsored search: is money a motivator for providing relevant results? Computer 8:52–57
4. Jansen BJ et al (2009) The components and impact of sponsored search. Computer 5:98–101
5. Nash JF Jr (1950) Equilibrium points in $n$-person games. Proc of the Nat Aca of Sciences 36(1):48–49
6. Nau DS (1983) Decision quality as a function of search depth on game trees. J ACM 30(4):687–708
7. Neumann JV, Morgenstern O (2007) Theory of games and economic behavior (Commemor edn), Princeton University Press
8. Nilsson NJ (1980) Principles of artificial intelligence, 3rd edn. Narosa Publishing, India
9. Prisner E (2014) Game theory through examples, Electronic edn. Mathematical Association of America. ISBN 978-1-61444-115-1
10. Roth AE (1983) Towards a theory of bargaining: an experimental study in economics. Science 220:687–691
11. Schaeffer J, Herik HJ (2002) Games, computers, and artificial intelligence. Artif Intell 134:1–7
12. Slagle JR, Dixon JK (1969) Experiments with some programs that search game trees. J ACM 16(2):189–207
13. Sandholm T (2015) Solving imperfect-information games. Science 347(6218)
14. Stockman GC (1979) A minimax algorithm better than alpha-beta? Artif Intell 12:179–196

# Chapter 12
# Reasoning in Uncertain Environments

**Abstract** Real-life propositions are neither fully true nor false, also there is always some uncertainty associated with them. Therefore, the reasoning using real-life knowledge should also be in accord. This chapter is aimed to fulfill the above objectives. The chapter presents the prerequisites—foundations of probability theory, conditional probability, Bayes theorem, and Bayesian networks which are graphical representation of conditional probability, propagation of beliefs through these networks, and the limitations of Bayes theorem. Application of Bayesian probability has been demonstrated for specific problem solutions. Another theory—the Dempster–Shafer theory of evidence—which provides better results as the evidences increase is presented, and has been applied on worked examples. Reasoning using fuzzy sets is yet another approach for reasoning in uncertain environments—a theory where membership of sets is partial. Inferencing using fuzzy relations and fuzzy rules has been demonstrated, followed by chapter summary, and a set of exercises at the end.

**Keywords** Reasoning in uncertainty · Probability theory · Conditional probability · Bayes theorem · Belief networks · Belief propagation · Dempster–Shafer theory · Fuzzy sets · Fuzzy relations · Fuzzy inference

## 12.1 Introduction

The real-life propositions are neither fully true nor false, and there is uncertainty associated with them. Therefore, the reasoning drawn from these are also probabilistic. This requires probabilistic reasoning for decision-making. This chapter presents two approaches for probabilistic reasoning—the Bayes theorem and Dempster–Shafer theory. The first is implemented as Bayesian belief networks, where nodes (*events*) are connected using directed graph with edges of the graph representing *cause–effect* relationships; the beliefs propagate in a network as per the rules of Bayes conditional probability. The Dempster–Shafer (D-S) theory is based on evidential reasoning such that evidences are conjuncted, and as more and more evidences take part in the reasoning, the ignorance interval decreases. The classical probability theory is a special case of D-S theory of *Evidential Reasoning*.

In the case of *Classical Logic*, a proposition is always taken either as true or false. However, in a real-life scenario one cannot always say a proposition to be either 100% true or 100% false, but known to be true with a certain probability. For example, one can say that the proposition: "Moon rover will function alright," may have a probability of being true as 30%, which is calculated based on the success rate of previous probes sent to the Moon. However, the proposition "Sun rover will function alright" has a probability of being true as 0%, due to extreme temperature at the Sun's surface. Because such uncertainties are common in expert systems when they are deployed in real-life applications, the knowledge representation as well as the reasoning system of the expert system must be extended to work like in a real-life situation.

Fundamentally, there are two basic approaches to reasoning in uncertain domains: *probabilistic reasoning* and *non-monotonic reasoning*. The basis of probabilistic reasoning is to attach some probability value to each proposition to express the uncertainty of the event. These uncertainties may be derived from existing statistical information available as the database of a corpus, or these may be estimated by the experts.

Facilities for handling uncertainty have long been an integral part of knowledge based system. In the early days of rule-based programming, the predominant methods used variants on probability calculus to combine certainty factors associated with applicable rules. Although it was recognized that certainty factors did not conform to the well-established theory of probability, these methods were nevertheless favored because the probabilistic techniques available at the time required either specifying an intractable number of parameters or assumed an unrealistic set of independence of relationships.

The most important area, for understanding the uncertainties is medical diagnostics or troubleshooting of any system; in the first it is required to identify the patterns (diagnoses) on the basis of their properties (symptoms), while in the second it is common to recognize the faults on the basis of system behavior. However, there is no one-to-one mapping of symptoms and diagnoses, and the uncertainties of mapping symptoms with diagnoses prevail due to the following reasons:

- ascertaining of the symptoms,
- proper evaluation of symptoms, and
- insufficient criteria about reckoning the scheme.

Generally, the end user of an expert system identifies and ascertains about the symptoms and their uncertainties, while the uncertainties in the evaluation of symptoms are carried out by the experts. Because different people judge these uncertainties, they are likely to judge it differently, and there is no scope of normalization. Consequently, it is likely to be a large uncertainty. Often the reckoning methods are also simple, which are some "ad hoc representations", on account of lack of theoretical foundations.

Many different methods for representing and reasoning with uncertain knowledge have been developed during the last three decades, including the certainty factor

calculus, Dempster–Shafer theory, possibilistic logic, fuzzy logic, and Bayesian networks (also called belief networks and causal probabilistic networks).

**Learning Outcomes of this Chapter**:

1. Identify examples of knowledge representations for reasoning under uncertainty. [Familiarity]
2. Make a probabilistic inference in a real-world problem using Bayes theorem to determine the probability of a hypothesis given evidence. [Usage]
3. Apply Bayes rule to determine the probability of a hypothesis given evidence. [Usage]
4. Describe the complexities of temporal probabilistic reasoning. [Familiarity]
5. State the complexity of exact inference. Identify methods for approximate inference. [Familiarity]
6. Design and implement an HMM as one example of a temporal probabilistic system. [Usage]
7. Explain how conditional independence assertions allow for greater efficiency of probabilistic systems. [Assessment]
8. Make a probabilistic inference in a real-world problem using Dempster–Shafer's theorem to determine the probability of a hypothesis given evidence. [Usage]

## 12.2 Foundations of Probability Theory

In the following, we list the commonly used terminology of the probability theory.

Considering an uncertain even $E$, the *probability of its occurrence* is a measure of the degree of likelihood of its occurrence.

A *sample space S* is the name given to the set of all possible events.

A *probability measure* is a function $P(E_i)$ that maps every event outcomes, $E_1, E_2, \ldots, E_n$ from event space $S$, to real numbers in the range $[0, 1]$.

These outcomes satisfy the following *axioms* (basic rules) of probability:

1. for any event $E \subseteq S$, there is $0 \leq P(E) \leq 1$,
2. an *outcome* from space $S$ that is certain is expressed as $P(S) = 1$, and
3. when events $E_i, E_j$ are *mutually exclusive*, i.e., for any events $E_i, E_j$, there is $E_i \cap E_j = \phi$, for all $i \neq j$. In that case, $P(E_1 \cup E_2 \cup \ldots \cup E_n) = P(E_1) + P(E_2) + \cdots + P(E_n) = 1$.

Using these three axioms, and the rules of set theory, the basic laws of probability can be derived. However, the axioms alone are not sufficient to compute the probability of an outcome, because it requires an understanding of the corresponding distribution of events. The distribution of events must be established using one of the following approaches:

1. Collection of experimental data to perform statistical estimates about the underlying distributions,

2. Characterization of processes using theoretical argument,
3. Understanding about the basic processes are helpful to assign subjective probability.

Some examples of computation of probability are given below. The upper case variable names represent the sets.

$$P(A) = \frac{count\ of\ (All\ A)}{count\ of\ (all\ events)} \tag{12.1}$$

$$P(A \cap B) = \frac{count\ of\ (All\ A\ and\ B\ together)}{count\ of\ (all\ B's)} \tag{12.2}$$

$$\begin{aligned} P(A \cap B) &= P(A \mid B)P(B) \\ &= P(B \mid A)P(A). \end{aligned} \tag{12.3}$$

Therefore,

$$P(A|B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{12.4}$$

Most of the knowledge we deal with is uncertain, hence the conclusions based on available evidences and past experiences are incomplete. Many a time it is only possible to obtain partial knowledge about the outcomes.

## 12.3   Conditional Probability and Bayes Theorem

The Bayesian networks provide a powerful framework for modeling uncertain interactions between variables in any domain of concern. The interactions between the variables are represented in two ways: 1. *Qualitative way*, using *directed acyclic graph*, and 2. *Quantitative way*, which makes use of *conditional probability* distribution for every variable in the network. This probability distribution-based system allows to express the relationship between the variables, as functional, relational, or logical.

Along with the above, the *dynamic probability theory* provides a mechanism for revising the probabilities of events in a coherent way, as the evidences become available. We represent the conditional probability, i.e., the probability of occurrence of an event $A$, given that the event $B$ has already occurred, as $P(A|B)$. Note that, if the events $A$ and $B$ are totally independent, then $P(A|B)$ and $P(A)$ will have the same result, otherwise, due to its dependency, the occurrence of $A$ will be effected, and accordingly the probability $P(A|B)$. Here, occurrence of event $A$ is called *hypothesis* and occurrence of event $B$ is called as *evidence*. If we are counting the number of sample points, we are interested in the fraction of events $B$ for which $A$ is also true,

i.e., $(B \cap A)$ is what fraction of $B$? The meaning of fraction is like the $A$ and $B$ are fractions of the universe set $\Omega$. In $(B \cap A)$, both $A$ and $B$ are partly included, which is also represented as $(A, B)$. From this it becomes clear that

$$P(A|B) = \frac{P(A, B)}{P(B)}. \tag{12.5}$$

The above equation is often written as

$$P(A, B) = P(A|B)P(B). \tag{12.6}$$

Equation (12.6) is also called "product rule" or simple form of *Bayes* theorem. It is important to note that this form of the rule is not often stated a definition, but a theorem, which can be derived from simpler assumptions. The term $P(A|B)$ in the equation is called *posterior probability*.

The Bayes theorem is used to tell us how to obtain a posterior probability of a hypothesis $A$ once we have observed some evidence $B$, given the *prior probability* $P(A)$ of event $A$, and probability $P(B|A)$—the *likelihood* of observing $B$—were $A$ to be given. The theorem is stated as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{12.7}$$

The formula (12.7), though simple, has abundant practical importance in the area such as diagnosis, may it be patient or a fault diagnosis in software or an electric circuit, or any other similar situation. Note that it is often easier to conclude the probability of observing a symptom given a disease, than the reverse—that is, of a disease, given a symptom. Yet, in practice it is the latter (i.e., find out the disease or fault given the symptom), which is required to be computed. See what the doctors do, it is the disease (hypothesis) they try to find out given the symptom (evidence).

There are certain conditions under which the Bayes theorem is valid. Let us assume that there is only one symptom ($S$) and corresponding only one diagnosis ($D$), and having given this we will try to extend the formula of the Bayes theorem. The conditional probability for this is expressed as $P(S|D)$, which says about relative frequency of occurrence of the symptom $S$ when diagnosis $D$ is true. The term $P(S|D)$ can be expressed by

$$P(S|D) = \frac{|S \cap D|}{|D|}, \tag{12.8}$$

where

$|S|$ is frequency of occurrence of symptom $S$,
$|D|$ is frequency of the occurrence of diagnosis $D$, and
$|S \cap D|$ is frequency of simultaneous occurrence of both $S$ and $D$.

The probability $P(D|S)$, called *posterior probability* or *conditional probability* of diagnosis $D$, assuming a symptom $S$, is given by Bayes theorem as

$$P(D|S) = \frac{P(S|D)P(D)}{P(S)}. \tag{12.9}$$

The term $P(S|D)$ is called *likelihood* of symptom $S$ when diagnosis is $D$, and $P(D)$ is called *prior probability*.

The above can be proved as follows:

$$
\begin{aligned}
P(D|S) &= \frac{|S \cap D|}{|S|} \\
&= \frac{|D|}{|D|} * \frac{|S \cap D|}{|S|} \\
&= |D| * \frac{|S \cap D|}{|D| * |S|} \\
&= |D| * \frac{|S \cap D|}{|D|} * \frac{1}{|S|} \\
&= \frac{P(S|D) * P(D)}{P(S)} ; \text{using Eq. (12.8)}
\end{aligned}
$$

The denominator terms $P(S)$ is called the *normalizing* factor.

**Example 12.1**  Suppose that the following statistics hold for some disease:

$P(tuberculosis) = 0.012,$
$P(cough) = 0.10,$ (for over 3 weeks)
$P(cough \mid tuberculosis) = 0.90,$ i.e., if a patient has tuberculosis, the coughing is found in 90% of the cases.

Given this, compute the probability of having tuberculosis, given that coughing exists in those persons, for over 3 weeks.

$$
\begin{aligned}
P(tuberculosis|cough) &= \frac{P(cough|tuberculosis) \times P(tuberculosis)}{P(cough)} \\
&= \frac{0.9 \times 0.012}{0.10} \\
&= 0.108.
\end{aligned}
$$

Hence, the relative probability that a patient who had a tuberculosis was found to be having also the cough is, 8.33 (= 0.9/0.108) times the probability of finding tuberculosis in the coughing patients.                                                 □

Relative Probability

The Bayes theorem allows the determination of the most probable diagnosis, assuming the presence of symptoms $S_1 \ldots S_m$. The probability is determined using: 1. *a priori* probabilities $P(D_i)$ that belong to a set of diagnoses, and 2. *likelihood* $P(S_j/D_i)$. The latter is the relative frequency of the occurrence of a symptom $S_j$ given that the diagnosis is $D_i$. In the Bayes probability, an absolute probability is not important, but a relative probability $P_r$ is determined. $P_r$ is the probability of a diagnosis compared to the other diagnoses, which is computed as

$$P_r(D_i|S_1 \wedge S_2 \wedge \ldots \wedge S_m) = \frac{P(D_i)\ P(S_1|D_i)\ldots P(S_m|D_i)}{\Sigma_{j=1}^{n} P(D_j)\ P(S_1|D_j)\ldots P(S_m|D_j)}. \quad (12.10)$$

Note that in the above definition, the *normalization* factor of $P(S_1 \wedge \ldots \wedge S_m)$, which is a common denominator, in the numerator and denominator terms, has been dropped.

When we are considering many symptoms, as in the above case, it is necessary that the correlation of every relevant combination of symptoms to disease be determined. However, if this happens, it would lead to a combinatorial explosion of diagnoses, e.g., with 100 all possible symptoms, there are $2^{100}$ different constellations of diagnoses. This figure is more that $10^{30}$!

Therefore, while using the Bayes theorem it is always assumed that the symptoms are independent of each other, i.e., for two symptoms, $S_i$ and $S_j$

$$P(S_i \wedge S_j|D) = P(S_i|D)P(S_j|D). \quad (12.11)$$

The exception to the above is allowed when symptoms are caused directly by the same diagnosis.

From Eqs. (12.10) to (12.11), it follows that

$$\begin{aligned} P(D|S_1 \wedge \ldots \wedge S_m) &= \frac{P(S_1 \wedge \ldots \wedge S_m|D)}{P(S_1 \wedge \ldots \wedge S_m)P(D)} \\ &= \frac{P(S_1|D)\ldots P(S_m|D)}{P(S_1 \wedge \ldots \wedge S_m)P(D)}. \end{aligned}$$

The Bayes theorem in the above form is correct, subject to the fulfillment of the following conditions:

1. The symptoms may depend only on the diagnoses, and must be independent of each other. This becomes a critical point, when more and more symptoms are ascertained.
2. The set of diagnoses is *complete*.
3. Single fault assumption or mutual exclusion of diagnoses—presence of any one diagnosis automatically excludes the presence of other diagnoses. This criteria is justified only in a relatively small set of diagnoses.
4. The requirement of correct and complete statistics of prior probabilities ($P(D)$) of the diagnoses, and conditional symptom-diagnosis probabilities($P(S|D)$).

## 12.4   Bayesian Networks

The Bayesian network is a graphical representation of the Bayes theorem—a set of variables represented as *cause–effect relations*, where variables are nodes and edges represent the relevance or influence relation between variables. Absence of an edge between two variables indicates independence of the variables, i.e., nothing can be inferred about the state of one variable, having given the state of the other variable. A variable may assume the values from a collection such that this collection is a mutually exclusive as well as collectively exhaustive set. A variable is allowed to be discrete—having a finite countable number of states—or it may be continuous—infinitely many possible states.

The lowercase letters in Bayesian networks represent single variables, while uppercase letters represent sets of variables. To assign state $k$ to variable $x$, we write as $x = k$. When the state of every variable is comprised in set $X$, then this set (an observation) is an instance of $X$. Set of all the instances is represented by a universal set $U$, which is the joint space of all the instances. A joint probability distribution over set $U$ is the probability distribution over joint space $U$. Considering $X$, and $Y$ as sets, the expression $P(X|Y)$ denotes the joint probability distribution over the set $X$, one for each conditional corresponding to every instance in the joint space of $Y$.

The Bayesian networks are good for structuring probabilistic information about a situation in a systematic way. This information is provided in localized and coherent way. These networks also provide a collection of algorithms, which help in deriving many information as implications of the conditionals, that can lead to important conclusions, and decisions about the given situation. These may be for example, in the area of genetics—mapping genes onto a chromosome, in message transmission and reception—finding the most likely message that was sent across a noisy channel, in system reliability—computing the overall reliability of a system, in online sales, e.g., to identify the most likely users who would respond to a given advertisement, in image processing, e.g., restoring a noisy image, and so on.

In more technical terms, a Bayesian network is a representation in a compact form of probability distribution using graphical method. The traditional methods were usually too large to handle probability and statistics computations, such as using tables and equations. Typically, a Bayesian network of 100s variables can be easily constructed and used for reasoning successfully about probability distribution [4].

### 12.4.1   Constructing a Bayesian Network

Though the definition of a Bayesian network is based on *conditional independence*, these networks are usually constructed using the notions of *cause* and *effect*. In simple words, for a given set of variables, we construct a Bayesian network by connecting every cause to its immediate effect using arrows in that direction. In almost all the cases, this results in Bayesian networks, whose conditional independence are implicit and accurate.
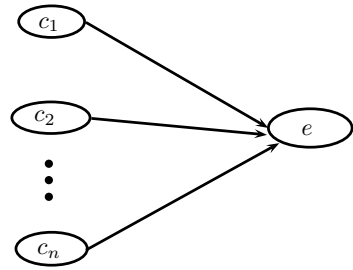
Figure 12.1 shows a Bayesian network with many causes ($c_1, c_2, \ldots, c_n$) and a single effect $e$. Due to this, specification of its local distribution can be quite onerous in the general case. As per the definition for a Bayesian network, it is necessary to compute the probability distribution of the node conditional on every instance of its parents. For example, if a node $x$ has $n$ number of parents, each is binary in nature (either 0 or 1 only), then we need to specify total $2^n$ probability distributions for the node $x$. We can reduce the complexity of this computation in such cases by adding more structures in the model. For example, we can convert our model into a $n$-way interaction model by associating with each cause node an inhibitory mechanism that prevents the cause from producing the effect. In such a model, the effect will be absent if and only if all the inhibiting mechanisms associated with the present cause node are active. And, if any of the inhibiting mechanism for this node is inactive, the cause node will produce the effect.

### 12.4.2 Bayesian Network for Chain of Variables

A problem domain is nothing but a set of variables. Thus, a Bayesian network for the domain $\{x_1, \ldots, x_n\}$ can represent a joint probability distribution over these variables. This distribution consists of a set of local conditional probability distribution, that when combined with a set of assertions of conditional independence, gives a joint global distribution.

The composition is based on the chain rule of probability (see Fig. 12.2), which dictates that

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | x_1, \ldots, x_{i-1}). \tag{12.12}$$

For each given variable $x_i$, let us assume that $S_i \subseteq \{x_1, \ldots, x_{i-1}\}$ is a set of variables which causes the variable $x_i$, and $\{x_1, \ldots, x_{i-1}\}$ to be conditionally independent. That is

**Fig. 12.2** Chain of domain
variables
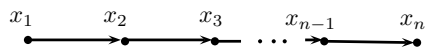
$$P(x_i|x_1, \ldots, x_{i-1}) = P(x_i|S_i). \tag{12.13}$$

The approach to this idea is that the distribution of $x_i$ can be represented as conditional on a parent set of $S_i$ which is substantially smaller than $\{x_1, \ldots, x_{i-1}\}$. Given these two sets, a Bayesian network can be represented as a directed acyclic graph such that each variables $x_1, \ldots, x_n$ corresponds to a node in that graph and the parents of the node corresponding to the variables in $S_i$. Note that since the graph coincides with the conditioning set $S_i$, the assertions of the conditional dependence are directly encoded by the Bayesian network structure. This is expressed in Eq. (12.13).

A conditional probability distribution $P(x_i|S_i)$ is associated with each node $x_i$, such that there is one distribution for each instance of $S_i$. By combining the Eqs. (12.12) and (12.13), we obtain a Bayesian network for $\{x_1, \ldots, x_n\}$ which uniquely determines a joint probability distribution for these variables. That is,

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|S_i). \tag{12.14}$$

Because the joint probability distribution for any network can be determined using Bayesian network for a given domain, this network can be used in principle to compute any probability of interest. Consider that we are given a simple Bayesian network, with structure $w \to x \to y \to z$, and we are interested to find out the probability "of occurrence of event $w$ given that the event $z$ has already occurred," which can be expressed as $P(w|z)$. From the Bayes rule, and making use of the above structure $w \ldots z$, the equation can be expressed as follows.

$$P(w|z) = \frac{P(w, z)}{P(z)} = \frac{\sum_{x,y} P(w, x, y, z)}{\sum_{w,x,y} P(w, x, y, z)}. \tag{12.15}$$

In the above, $P(w, x, y, z)$ is called the joint probability distribution computed for the Bayesian network $w \to x \to y \to z$. On observation of numerator and denominator terms of Eq. (12.15), we can understand that this approach is not feasible, due to the reason that it requires summing over exponential number of terms. However, we can exploit the conditional independence encoded in a Bayesian network to arrive at a more efficient computation. Using this feature, the network structure of Eq. (12.15) can be transformed to

$$P(w|z) = \frac{\sum_{x,y} P(w, x, y, z)}{\sum_{w,x,y} P(w, x, y, z)}$$

$$= \frac{P(w) \sum_{x} P(x|w) \sum_{y} P(y|x) P(z|y)}{\sum_{w} P(w) \sum_{x} P(w) P(x|w) \sum_{y} P(y|x) P(z|y)}. \tag{12.16}$$

Equation (12.16) indicates that using conditional independence (i.e., independence of conditional variable), we can often reduce the dimensions of an equation by rewriting the "sums over multiple variables as a product of sums over a single variable," or at least to a lesser number of variables.

A *probabilistic inference* is a general problem of computing probabilities of interest from a joint probability distribution, which may be possibly implicit. It may be noted that all the exact algorithms for probabilistic inference using Bayesian networks exploit conditional independence in the manner described above.

For drawing probabilistic inferences, it is possible to exploit the conditional independence in a Bayesian network, however, the *exact inference* in an arbitrary Bayesian network is *NP-hard*.[1] But, actually, for many applications, the Bayesian networks are of small size, or they can be simplified sufficiently, such that the complexity issues are not important. For applications where usual inference are impracticable, there are equivalent applications existing which are particular network tailored or suited for specific queries.

### 12.4.3 Independence of Variables

The notions of *independence* and *conditional independence* are fundamentals notions of probability theory. It is the combination of quantitative information of numerical parameters and the qualitative information, that makes the probability theory so expressive. Consider $x$ and $y$ as two variables that are independent of each other. The corresponding probabilistic expression of occurrence of events corresponding to these variables is

$$P(x, y) = P(x)P(y). \tag{12.17}$$

Given three variables $x, y, z$, the probability distribution of these can be decomposed into a joint probability distribution that comprises terms, each of two variables, as shown in Eq. 12.18 and Fig. 12.3.

$$\begin{aligned} P(x, y, z) &= P(x, y|z)P(z) \\ &= P(x|z)P(y|z)P(z). \end{aligned} \tag{12.18}$$

For the situation shown in Fig. 12.3, the variables $x$ and $y$ are called *marginally independent*, but they are *conditional dependent*, for a given variable $z$. It is possible to convince ourselves using the following example. Let the first variable have the assignment $x$ = "rain", and the second is $y$ = "sprinkler on". When both are True, they will cause the lawn to become wet.

So far we have not made any observation about the lawn, and occurrence of $x$ (rain) and $y$ (sprinkler on) are independent. However, once it is observed that the lawn

---

[1]NP-Hard: Non-deterministic Polynomial-hard, i.e., whose polynomial nature of complexity is unknown, and these problems are considered as the hardest problems in Computer Science.

Rain                    Sprinkler on
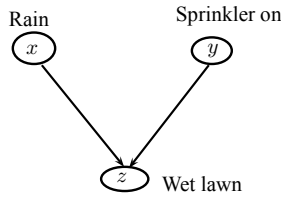


**Fig. 12.3** Variables $x$ and $y$ are "Conditionally Dependent Given" $z$
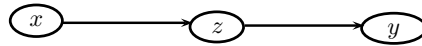


**Fig. 12.4** Variables $x$ and $y$ are conditionally independent given $z$

is wet, and it is confirmed that it raining, it automatically influences the probability of the sprinkler, that it is "on". This probability distribution is therefore

$$P(x, y, z) = P(z|x, y)P(x)P(y). \tag{12.19}$$

The next example shows the cause–effect relationships shown in Fig. 12.4. In this case, the probability distribution is given by

$$P(x, y, z) = P(y|z)P(z|x)P(x). \tag{12.20}$$

### *12.4.4   Propagation in Bayesian Belief Networks*

The aim of constructing a Bayesian network is—for a given observation (evidence), answer a query about the probability distribution over the values of query variables. A Bayesian network having full specification contains all the information needed to answer all the queries of probability distribution over these variables. The propagation of evidence in these networks helps in drawing conclusions. Usually, we make use of the term "propagation" only, instead of propagation of evidence.

This kind of representation is called *Bayesian belief network*. The inference in this network amounts to the propagation of probabilities of a given and related information through the network to one or more conclusion nodes. If we consider representing the knowledge related to a set of variables, say, $x_1, x_2, \ldots, x_n$, using their point probability distribution, $P(x_1, x_2, \ldots, x_n)$, it will require a total of $2^n$ entries to store the entire distribution explicitly. Further, for the determination of probability say $x_i$ will require summing all the remaining $x_i$, which will make it prohibitively expensive in terms of computation.

However, once it is represented using causal relationships, the joint probability can be computed much faster. Figure 12.5 shows the cause and effect relationships

**Fig. 12.5** Bayesian belief network
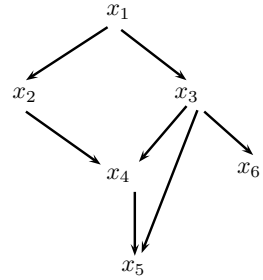


between variables $x_1 \ldots x_5$, and the joint probability distribution can be given as

$$P(x_5|x_4 \wedge x_3)P(x_4|x_2 \wedge x_3)P(x_2|x_1)P(x_3|x_1)P(x_6|x_3)P(x_1). \qquad (12.21)$$

**Example 12.2** Some examples of Bayesian belief networks.

Following are causal dependencies of variables, their corresponding belief networks, and probability distributions:

 (i) Let $W$ = "Worn piston rings", that causes $O$ = "excessive oil consumption", which in turn causes $L$ = "low oil level in fuel tank". Figure 12.6 shows the cause–effect relationships, and the joint probability distribution for this is given by

$$P(W, O, L) = P(W) \, P(O|W) \, P(L|O). \qquad (12.22)$$

 (ii) As another example, let $W$ = "Worn piston rings" cause both $B$ = "blue exhaust", as well as, $L$ = "low oil level" (see Fig. 12.7). The joint probability distribution is given by

$$P(W, B, L) = P(W) \, P(B|W) \, P(L|W). \qquad (12.23)$$

(iii) Consider another example, with variables, $L$ = "Low oil level", which can be caused either by $C$ = "excessive consumption" or by $E$ = "oil leak" (see Fig. 12.8). The joint probability distribution is given by

$$P(C, E, L) = P(C) \, P(E) \, P(L|C, E). \qquad (12.24)$$

$\square$

Based on the above discussions, we have the following steps for the construction of a Bayesian network:
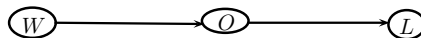


**Fig. 12.6** Bayesian network-1

**Fig. 12.7** Bayesian
network-2
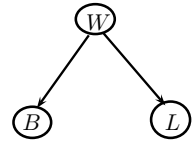


**Fig. 12.8** Bayesian
network-3
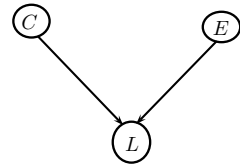


**Fig. 12.9** Bayesian
network-4
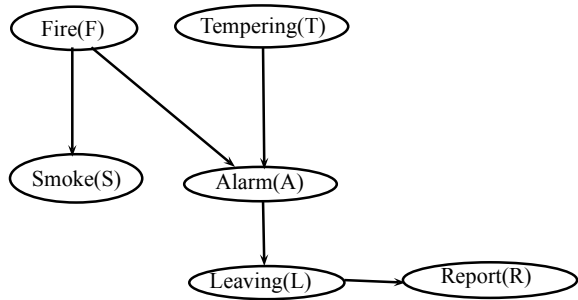


1. Construct belief (event) nodes corresponding to the events,
2. For each node in belief network, make all links corresponding to the cause–effect
   relationships,
3. Compute probability at each nodes based on the probabilities of the premises
   nodes, and then,
4. Compute the joint probability distribution of the network.

**Example 12.3**  A Bayesian network for fire security system.

Figure 12.9 shows a small network of six binary variables given in Tables 12.1 and
12.2. The use of such network may be to compute answers to probabilistic queries, for
example, we may like to know the probability of fire, given that people are reported
to be leaving from the room, or answers for queries like what is the probability of
smoke given that the alarm is off.

A Bayesian network has two components: a structure in the form of a directed
acyclic graph, and a set of conditional probability tables. The acyclic graph's nodes

**Table 12.1**  Two-variable table

| Fire | Smoke | $P(S|F)$ |
|------|-------|----------|
| False | True | 0.01 |
| True | True | 0.90 |

**Table 12.2**  Four-variable table

| Fire($F$) | Tempering($T$) | Alarm($A$) | $P(A|F, T)$ |
|-----------|----------------|------------|-------------|
| True      | True           | True       | 0.50        |
| False     | True           | True       | 0.85        |
| True      | False          | True       | 0.99        |
| False     | False          | True       | 0.0001      |

correspond to variables of interest (see Fig. 12.9), and the graphs edges have a formal interpretation in the form of direct causal influences. A Bayesian network must include the conditional probability tables (CPTs) for each variable quantifying the relationship between a variable and its parents in the network.

Consider that there are variables $A$ = "Alarm", $F$ = "tempering", and $F$ = "Fire", each of which may be True or False. Figure 12.9 shows the acyclic graph, and Table 12.2 shows the conditional probability distribution of $A$, given its parents as "Fire" and "Tempering". As per this CPT, the probability of occurrence of the event $A = True$, given the evidence $F = True$ and $T = False$ (row 3), is $P(A = True|F = True, T = False) = 0.99$. This probability is called the *network parameter*.

The guaranteed *consistency* and *completeness* are the main features of Bayesian networks. The latter is possible because there is one and only one probability distribution, which satisfies the network constraints.

A Bayesian network with $n$ binary variables will induce a unique probability distribution for over $2^n$ instantiations; such instantiations for the network shown in Fig. 12.9 are 64. This distribution provides sufficient information to predict probability for each and every event we can express using variables $S, F, T, A, L, R$, appearing in this network. An event may be a combination of these variables with True/False values. An example of an event may be to found out as "probability of an Alarm and Tempering, given no smoke, and a report has indicated people leaving the building."                                                                                              □

An important advantage of Bayesian networks is the availability of efficient algorithms for computing such probability distributions. In the absence of these networks, it would require explicit generation of required probability distributions. Generating such distributions explicitly is infeasible for a large number of variables. Interestingly, in areas such as genetics, reliability analysis, and information theory, the already existing algorithms are subsumed by the more general algorithms for Bayesian networks.

### 12.4.5  Causality and Independence

We will try to discover the central insight behind Bayesian networks, due to which it becomes possible to represent large probability distributions in a compact way. Consider Fig. 12.9 and the associated Conditional Probability Tables (CPTs), Tables 12.1

and 12.2. Each probability appearing in a CPT specifies a constraint which must be satisfied by the distribution induced by the network. Consider that a distribution must assign the probability of 0.1 to the event of "having smoke without fire", i.e., $P(S = True|F = False)$, where $S$ and $F$ are variables, respectively for "smoke" and "fire". These constraints, however, are insufficient to help in concluding a unique probability distribution. So, what additional information we need? This answer lies in the structure of a Bayesian network; as we know, that additional constraints are specified by the Bayesian network in the form of probabilistic *conditional independence*. As per that, once the parent of every variable is known, the variable in the structure is assumed to be independent of its non-descendent parents. Figure 12.9 shows that the variable $L$ that stands for "leaving the premises" is taken as independent of its non-descendant parents $T$, $F$, and $S$, once its parent $A$ becomes known. Put differently, as soon as the variable $A$ becomes known, the probability distribution of variable $L$ will not change on the availability of new information about the variables $T$, $F$, and $S$.

As a different case in Fig. 12.9, we assume that variable $A$ has no dependence on variable $S$ (a non-descendant parent of $A$). This assumption becomes true as soon as the parents $F$ and $T$ of variable $A$ becomes known. These independence constraints, due to non-descendant parents are called *Markovian* assumptions of a Bayesian network [2].

From the above discussion, do we mean that whenever a Bayesian network is constructed, it is necessary to verify the conditional non-dependencies? This, in fact depends on the construction method adopted. There are three main approaches to the construction of Bayesian networks:

1. subjective construction,
2. a construction based on synthesis from other specifications, and
3. construction by learning from data.

The first method in the above approaches is somewhat less systematic, as one rarely thinks about the conditional independence while constructing a Bayesian network. Instead, one thinks about *causality*, i.e., adding an edge $E_i \rightarrow E_j$, from event $E_i$ to event $E_J$, whenever $E_i$ is recognized as a direct cause of $E_j$. This results in a causal structure where Markovian assumption is read as: "each variable becomes independent of its non-dependents, once direct causes are known."

A distribution of probabilities induced due to a Bayesian network also satisfies additional independencies beyond the Markovian. All these independent variables can be identified using a graphical test, called *d-separation*.[2] As per this test, any two variables $E_i$ and $E_j$ shall be considered to be independent if every path between $E_i$ and $E_j$ is blocked by a third variable $E_k$. For instance, consider the path $\alpha$ as shown in Fig. 12.9.

$$\alpha : S \leftarrow F \rightarrow A \leftarrow T. \tag{12.25}$$

---

[2]*d-separation:* A method to determine which variables are independent in a Bayes net [3].

Now, consider that alarm $A$ is triggered. The path in Eq. (12.25) can be used to show a dependency between variables $A$ and $T$ as follows: we know that observing $S$ (smoke as evidence) increases the likelihood that fire $F$ has taken place. This is due to the direct cause–effect relation. Also, the increased likelihood that $F$ has taken place explains away the tempering $T$ as cause of the alarm, i.e., there are lesser chances that the alarm is due to tempering. Hence, a path in Eq. (12.25) can be used as a dependence between $S$ and $T$. Therefore, the variables $S$ and $T$ are not independent due to the presence of *unblocked* path between them.

However, instead of variable $A$, if $F$ is a given variable, this path cannot be used to show a dependency between $S$ and $T$, and in that case the path will be *blocked* by $F$.

### 12.4.6 Hidden Markov Models

The Hidden Markov Models (HMMs) are useful for modeling dynamic systems with some states not observable, and when it is required to make inferences about changing states, given the sequence of outputs they generate. The potential applications of HMMs are those that require temporal pattern recognition, like speech processing, handwriting recognition, recognizing gestures, and in bioinformatics [2].

Figure 12.10a shows an HMM—a model of a system with three states $a, b, c$, and with three outputs $x, y, z$, and possible transitions between the states. There is probability associated with each transition, e.g., there is a transition from state $b$ to $c$ with 20% probability. Each state can produce certain output, with some probability, e.g., state $b$ can produce output $z$ with probability 10%. An HMM of Fig. 12.10a has been represented by a Bayesian network as shown in Fig. 12.10b. There are two variables, $S_t$ (for state at time $t$), and $O_t$ (for system output at time $t$), with $t = 1, 2, \ldots, n$. The variable $S_t$ has three values $a, b, c$, and $O_t$ also has three values $x, y, z$. Using *d-separation* on this network, one can derive the characteristic properties of HMMs, i.e., once the system state at time $t$ is known, its states and outputs are independent at time $> t$, and also independent at time $< t$. Figure 12.10b is a simple *dynamic Bayesian network*.



(a) HMM          (b) Dynamic Basian network.
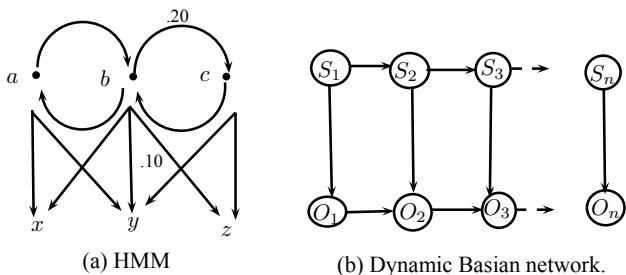
**Fig. 12.10** Hidden Markov model

### 12.4.7   Construction Process of Bayesian Networks

One approach for the construction of Bayesian networks is mostly subjective—reflecting on the available knowledge in the form of say, perceptions about causal influences. This knowledge is captured into a Bayesian network, e.g., the network shown in Fig. 12.9 we discussed earlier.

Automatic Synthesizing of networks

The second method for Bayesian networks construction synthesizes these networks automatically using some other type of formal knowledge. Many jobs, related to reliability and diagnosis, require system analysis. A Bayesian network can be automatically synthesized from the formal system design of such systems. We consider a job of reliability, where a reliability block diagram is used for reliability analysis, with system components being connected to effect their reliability and dependencies, as shown in Fig. 12.11a. The various components shown are power supply to power the Fan-1 and Fan-2, which collectively cool the processor-1 and 2, and these processors are interfaced to the a hard disk drive. The processor-1 requires the availability of either Fan-1 or 2, and each fan requires the availability of power supply. We are interested to compute the overall reliability of the system, i.e., probability of its availability, given the reliability of each of the components in the system. Figure 12.11b shows the systematic conversion of each block of the reliability block diagram into a Bayesian network fragment, where Subsystem-1, Subsystem-2, and Block-B are assumed to be available [2].

Figure 12.12 shows the corresponding Bayesian network constructed using reliability blocks. Using the reliability of individual components we can construct Con-



(a) Reliability block diagram.



(b) Bayesian network fragments
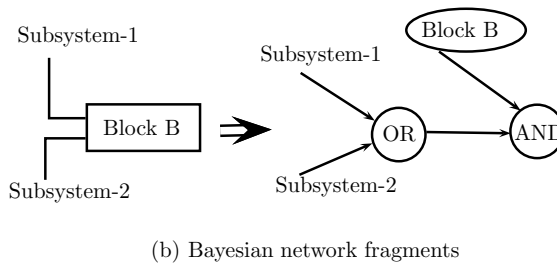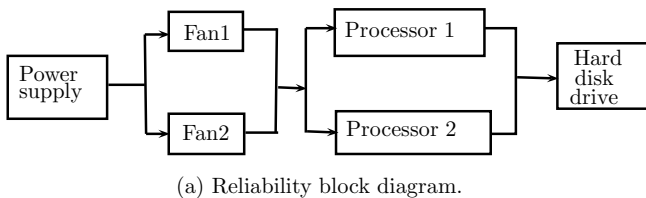
**Fig. 12.11**  Syntheizing a Bayesian network

**Fig. 12.12**  A Bayesian
network for reliability
analysis
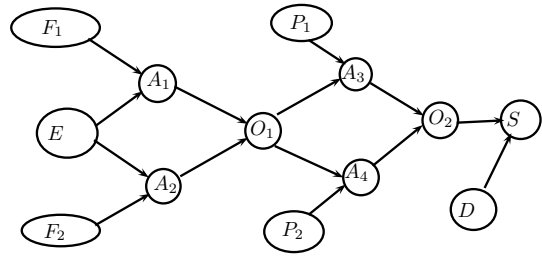


ditional Probability Tables (CPTs) of this system. Consider the variables as follows: $E$ is for power supply, $F_1$ and $F_2$ for fans, $P_1$, $P_2$ for processors, and $D$ for hard disk. Let these variables represent the availability of the corresponding component. Let us assume that variable $S$ represents the availability of the whole system. The variables $A_i$ and $O_j$ represent the logical $AND$ and $OR$, respectively.

Network Constructing by learning from data

A third approach for the construction of Bayesian networks works on the principle of learning from data, like patients medical records, airline ticket records, or buying patterns of customers, etc. Such data sets can help the network to learn parameters given the structure of the network, or it can learn both the network and its parameters when the data set is complete. However, learning only the parameter is an easier task.

Since the learning itself is an inductive task, the learning process is guided using the induction principle. Two main principles of inductive-based learning are 1. maximum likelihood function, and 2. learning using the Bayesian approach. The first approach is suitable to those Bayesian networks that maximize the probability of observing the given data set, while the second approach uses some prior information that encodes preferences on a Bayesian network, along with the likelihood principle.

Let us assume that we are interested in learning the network parameters. Learning using the Bayesian approach allows to put a *prior distribution* on each network parameter's possible values. The data set and the prior distribution together induce new distribution on the values of that parameter, called *posterior distribution*. This posterior distribution is then used to pick a value for that parameter, e.g. distribution mean. As an alternative, we can also use different parameter values while computing answers to queries. This can be done by averaging over to all possible parameter values according to their posterior probabilities.

Given a Bayesian network as in Fig. 12.12, we can find out the overall reliability of this system. Also, for example, given that the system is unavailable, we can find the most likely configuration of the two fans, or the processors. Further, given a Bayesian network, we can answer the questions such as What single component can be replaced to increase the system reliability by, say, 10%? These are the example questions which can be answered in these domains using the principles of Bayesian probability distributions.

## 12.5   Dempster–Shafer Theory of Evidence

The Dempster–Shafer Theory (DST) of evidence can be used for modeling several single pieces of evidences within a single *hypothesis relations*,[3] or a single piece of evidence in relations that are multi-hypotheses type. The DST is useful for the assessment of uncertainty of a system where actually only one hypothesis is true. Another approach to DST, called the *reliability-oriented* approach, contains the system with all hypotheses—pieces of *evidence* and *data sources*. The hypothesis is a collection of all possible states (e.g., faults) of the system, which is under consideration [5].

In DST, it is a precondition that all hypotheses are singletons (i.e., elements) of some *frame of discernment*—a finite universal set $\Omega$, with $2^{|\Omega|}$ subsets. However, we will write it simply as $2^{\Omega}$ subsets. A subset of $\Omega$ is a single hypothesis or a conjunction of hypotheses. The subsets $2^{\Omega}$ are unique and not all of them disjoint, however, it is mandatory that all the hypotheses are disjoint and mutually exclusive, in addition to being unique.

The *pieces of evidences* above are *symptoms* or *events*. An example of a symptom or evidence is the failure that has occurred or may occur in the system. An evidence is always related to a single hypothesis or to a set of hypotheses (multi-hypothesis). Though theoretically it can be debated, it is not allowed in DST that many different evidences may lead to conclude the same hypothesis or the same collection of hypotheses. A relation between a piece of evidence and a hypothesis is qualitative, which correspond to a chain of cause–effect relations—an evidence implies an existence of hypothesis or hypotheses, like in the Bayes rule.

The DST represents a subjective viewpoint in the form of a computation for an unknown objective fact. The *data sources* used in DST are persons, organizations, or other entities that provide the information. Using a data source, the mapping in Eq. (12.26) assigns an *evidential weight* to a diagnosis set $A \subseteq \Omega$. Since $A \in 2^{\Omega}$, the set $A$ contains a single hypothesis or a set of hypotheses. The probability of any diagnosis $A$ is a function whose value is between 0 and 1, and is expressed by

$$m : 2^{\Omega} \rightarrow [0, 1]. \tag{12.26}$$

Note that the difference between DST with probability theory is that, the DST mapping clearly distinguishes between the evidence measures and probabilities.

**Definition 12.1** (*Focal Element*) Each diagnosis set $A \subseteq \Omega$, for which $m(A) > 0$, is called a *focal element*.

**Definition 12.2** (*Basic Probability Assignment*) The function $m$ is called a Basic Probability Assignment (BPA) that fulfills the condition:

$$\sum_{A \subseteq \Omega} m(A) = 1. \tag{12.27}$$

---

[3] $H \rightarrow E$ is a hypothesis relation, where $H$ is hypothesis and $E$ is evidence.

Meaning of the above statement is that, for the evidences presented by each data source which are equal in weight, it is necessary that all statements of a single data source are normalized. In other words, no data source is more important than the others. We assume that if a data source is null, then its BPA should also be null, i.e., $m(\phi) = 0$.

A belief measure is given by the function, which also ranges between 0 and 1,

$$bel : 2^{\Omega} \rightarrow [0, 1], \tag{12.28}$$

such that

$$bel(A) = \sum_{B \subseteq A;\ B \neq \phi} m(B). \tag{12.29}$$

Counterpart of belief (*bel*) is called *plausibility* measure *pl*, which again is a mapping $pl : 2^{\Omega} \rightarrow [0, 1]$, and is defined as

$$pl(A) = \sum_{B \cap A \neq \phi} m(B). \tag{12.30}$$

It is important to note that $pl(A)$ above is not be understood as a complement of $bel(A)$. For a focal element $m(A) > 0$, and $A \subseteq \Omega$, it always holds that $bel(A) \leq pl(A)$. The difference between plausibility and belief is evidential interval range, called *uncertainty interval*. Thus,

$$uncertanty\ interval = pl(A) - bel(A). \tag{12.31}$$

As we add more and more evidences in the system, the uncertainty interval reduces, as will be seen in examples in the following. This is a natural and logical property of DST.

### 12.5.1  Dempster–Shafer Rule of Combination

The DST allows combination of evidences to predict the hypothesis with a stronger belief. It is called *Dempster's rule of combination*, which aggregates two independent bodies of evidence, into one body of evidence, provided that they are defined within the same frame of discernment. The Dempster's rule of combination combines two evidences and computes new basic probability assignment as

$$m(A) = m_1 \otimes m_2\ (A) = \frac{\sum\limits_{A = B \cap C} m_1(B) * m_2(C)}{1 - \sum\limits_{B \cap C = \phi} m_1(B) * m_2(C)}. \tag{12.32}$$

In the above, $A \neq \phi$ and numerator part in the equation represent the accumulated evidences for the sets $B$ and $C$, that supports the hypothesis $A$ (since $A = B \cap C$), and the denominator is called the *normalization factor* [5].

## 12.5.2  Dempster–Shafer Versus Bayes Theory

The Dempster–Shafer Theory (DST) (also called *theory of belief functions*) is the generalization of the Bayes theory of conditional probability. To carry out the computation of probability, the Bayes theory requires actual probabilities for each question of interest. But in DST, the degree of belief of one question is based on the probabilities of the related questions. The "degree of belief" may or may not have real mathematical properties of probability, but, how much they actually differ from probabilities will depend on how closely the two questions are related.

The DST is based on two ideas: 1. obtaining degrees of belief for one question from subjective probabilities of related question, and 2. use Dempster's rule for combining such degrees of belief when they are based on independent items of evidence.

The DST differs from the application of Bayes theorem in the following respects:

1. Unlike the Bayes theorem, the DST allows the representation of diagnoses in the form of classes and hierarchies, as shown in Fig. 12.13.
2. For a probability of $X\%$ for a diagnosis, the difference $(100 - X)\%$ is not taken as against the diagnosis, but interpreted as an *uncertainty interval*. That is, the probability of diagnosis lies between $X\%$ and 100%, and as more and more evidence is added the diagnosis may tend toward 100%.
3. Probability against a diagnosis is considered as the probability of the complement of the diagnosis. For example, if an event set is $\{A, B, C, D\}$, the probability against the event "$A$" is probability of $\{B, C, D\}$, which will have further distribution.

In spite of many advantages of DST, evaluation of probabilities in DST is more complex than in the Bayesian probabilities. This is because, probabilities for a set of diagnoses are related to one another in DST, hence probability of a set of diagnoses needs to be calculated from the distribution of probabilities over all *sets* of diagnoses.

**Fig. 12.13**  Hierarchy of evidences
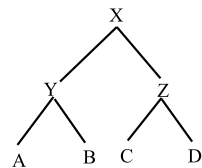
**Example 12.4** Computing Probability distribution of a biased coin.

Consider a biased coin, where the probability of arriving at the head is $P(H) = 0.5$. Since it is biased we do not know as to what the rest 50% is attributed to. In the DS theory, this is called the *ignorance level*. Since we do not know where the rest of the probability goes, we assign it to the universe $U = \{H, T\}$. Which means that it can attribute to $\{T\}$ and $\{H, T\}$.

Now let us assume that there is an *evidence*, that probability for arriving at the tail is 46%. Due to this new evidence, the balance probability assignment is $1 - 0.45 - 0.46 = 0.09$, and this can be assigned to the set $\{H, T\}$, that cannot be attributed to $\{H\}$ or $\{T\}$, unless there is some further evidence available. Hence, 0.09 probability assigned stands for the coin standing vertical when it is tossed. □

**Example 12.5** Using DST for finding probabilities for certain diseases.

Let us assume that in a certain isolated island the universe set of disease is $\Omega = \{malaria, typhoid, common cold\}$. A team of Doctors visit this island and perform a laboratory test $(T_1)$ on a patient, which shows that it is the case of 40% for Malaria. Hence, as per the DS theory, the balance 60% goes to the ignorance level for the entire universe {malaria, typhoid, common cold}.

Fearing lack of confidence, the Doctors ask the patient for a second laboratory test, for some other parameters; the evidence now shows a belief case of 30% for Malaria. Hence, the rest 30% goes to the entire universe {malaria, typhoid, common cold}. Thus ignorance is 30%.

The Doctor then asks for a third laboratory test, to further add to the evidences, and the finding shows that the belief for Typhoid is 20%. This makes the total belief of 90%, and the ignorance level reduces to 10%, which again is assigned to the universe set {malaria, typhoid, common cold}. □

The above example shows that as more and more evidence is added, we get more and more strong belief about the distribution of probability on various diagnoses, and the ignorance level reduces. This, in fact, is what it should be, and the natural case of reasoning in a probability case.

Let us assume that the kind or errors which can occur in software testing is the set: {data validation errors, computation errors, transmission errors, output error}. In this case also we use DST, similar to the cases discussed above, and can compute the joint probability distribution of errors.

**Example 12.6** Find out the distribution of probabilities for some diseases shown by a hierarchically structured diagnoses as shown in Fig. 12.13. Consider that the following are the probabilities of certain events: against $A = 40\%$, and evidence for $Y = 70\%$.

Let event $E_1$ correspond to against $A = 40\%$, and event $E_2$ : correspond to $Y = 70\%$. The evidence against $A$ means evidence for the complement of $A$, which is the set of evidences $\{B, C, D\}$. From Fig. 12.13, the evidence for $Y$ is evidence for set $\{A, B\}$.

**Table 12.3**  Combination of evidences

| $E_1 \downarrow E_2 \rightarrow$ | $\{A, B\} = 0.7$ | $\{A, B, C, D\} = 0.3$ |
|---|---|---|
| $\{B, C, D\} = 0.6$ | $\{B\} = 0.42$ | $\{B, C, D\} = 0.18$ |
| $\{A, B, C, D\} = 0.4$ | $\{A, B\} = 0.28$ | $\{A, B, C, D\} = 0.12$ |

In DST, when the evidences are combined, the probabilities get *multiplied* and the resultant set is the *intersection* of original sets. Table 12.3 shows the new distribution.

The probability of an event or a set of events is expressed not as an absolute value but an uncertainty interval $[a, b]$, where $a$ and $b$ are lower and upper probability limits, respectively. The value $a$ is computed as the sum of the probabilities of the set and its subsets, while $b$ is a value $100 - c$, where $c$ is the sum of the probabilities of complements of the set and complements of the subsets corresponding to $a$.

The probability range $[a, b]$ for diagnoses $\{A\}$ and $\{B\}$ are computed as follows:

For diagnosis $\{A\}$:

$$[a, b] = [\{A\}, 1 - (\{B, C, D\} + \{B, C\} + \{B, D\} + \{C, D\} + \{B\} + \{C\} + \{D\} + \{\})]$$
$$= [0, 1 - 0.6] = [0, 0.4].$$

Hence, the probability of diagnosis $\{A\}$ lies between 0 and 40%.

We know that in DST, probabilities are multiplied and intersection is performed. Since probability for $\{A, B\}$ is 0.7 and for $\{B, C, D\}$ is 0.6, the probability for $\{B\}$ ($\{A, B\} \cap \{B, C, D\} = \{B\}$) is 0.42. Thus, the probability range for diagnosis $B$ is

$$[a, b] = [\{B\}, 1 - (\{A, C, D\} + \{A, C\} + \{A, D\} + \{C, D\} + \{A\} + \{C\} + \{D\} + \{\})]$$
$$= [0.42, 1 - 0.0] = [0.42, 1].$$

Hence, the probability for $\{B\}$ lies between 42 and 100%.

If a further evidence $E_3$ of 20% for $D$ is added, the distribution of probabilities is changed as shown in Table 12.4.

The empty bracket $\{\}$ shows that there is no diagnosis attributed to this fraction of probability. Since it is assumed that the set of diagnoses are complete, we can

**Table 12.4**  Combination of evidences after a new evidence $E_3$ is added

| $E_1 \& E_2 \downarrow E_3 \rightarrow$ | $\{D\} = 0.2$ | $\{A, B, C, D\} = 0.8$ |
|---|---|---|
| $\{B\} = 0.42$ | $\{\} = 0.084$ | $\{B\} = 0.336$ |
| $\{A, B\} = 0.28$ | $\{\} = 0.056$ | $\{A, B\} = 0.224$ |
| $\{B, C, D\} = 0.18$ | $\{D\} = 0.036$ | $\{B, C, D\} = 0.144$ |
| $\{A, B, C, D\} = 0.12$ | $\{D\} = 0.024$ | $\{A, B, C, D\} = 0.096$ |

eliminate the empty set and the remaining sets of the probabilities can be taken as 100%. This can be done by dividing all probabilities of non-empty sets by a factor equal to $(1 - s)$, where $s$ is the sum of the probabilities of empty sets. For this example, $s = 0.084 + 0.056 = 0.14$ and $1 - s = 1 - 0.14 = 0.86$. Now, the sum of probabilities without an empty set is 1.0. Next, we obtain the following results using Dempster's rule of combination.

$$\{B\} = \frac{0.336}{0.86} = 39.0\%$$

$$\{D\} = \frac{0.06}{0.86} = 7.0\%$$

$$\{A, B\} = \frac{0.224}{0.86} = 26.0\%$$

$$\{B, C, D\} = \frac{0.144}{0.86} = 16.8\%$$

$$\{A, B, C, D\} = \frac{0.096}{0.86} = 11.2\%.$$

The probabilities for $\{A\}$ and $\{B\}$ are computed as follows:

$\{A\} = [\{A\}, 1 - (\{B, C, D\} + \ldots + \{B\} + \{D\})] = [0, 0.372]$, i.e., 0–37.2%.
$\{B\} = [\{B\}, 1 - (\{A, C, D\} + \ldots + \{D\})] = [0.39, 0.93]$, i.e., 39–93.0%.    □

From the above exercise, we note that the uncertainty interval (probability of $\{A, B, C, D\}$) has decreased after addition of new evidence $E_3$. In addition, the probabilities of $A$ and $B$ have turned out to be more precise, i.e., narrow.

However, we note that the computation required in DST is combinatorial, since with $n$ diagnoses a total of $2^n$ number of sets need to be computed [7].

## 12.6  Fuzzy Sets, Fuzzy Logic, and Fuzzy Inferences

In *classical* set theory, an element $x$ of the universe $U$ either absolutely belongs to a set $A \subseteq U$ or does not belong to it at all. This membership relation between the element $x$ and set $A$ is called *crisp*, i.e., either Yes/No (or On/Off). A fuzzy set is a generalization of a classical set by allowing the degree of membership, which is a real number [0, 1]. In extreme cases the degree is 0, i.e., the element does not belong to the set or 1, or the element fully belongs to the set.

To understand the idea of a fuzzy set, let us assume that people in an organization are the universe and consider a set of "young" people in this universe. The youngness is definitely not a step function from 1 to 0, as one attains a certain age, say 30 years. In fact, it would be natural if we associate a degree of youngness for each element of

age, for example, ({$Anand$/1, $Babita$/0.8, $Choudhary$/0.3}). That means, perhaps *Anand* is 23 years old, *Babita* is 27 years old, and *Choudhary* is in his 50s. The membership function of a set maps each element of the universe to some degree of association with the set.

The fuzzy set concept is somewhat similar to the concept of the set in the classical set theory. When we say, "Sun is a star", it means the Sun belongs to the set of stars, and when we say, "Moon is a satellite", it means the Moon belongs to the set of satellites of the Earth. At the same time, both these statements are true, hence, they both map to *true* or 1. The statement "the Sun is a black hole" is *false* as it does not belong to the set of blacks holes yet. Hence, the statement maps to *false* or 0. Thus, mapping of a statement in classical logic is either to 1 and 0, having crisp values. Since, with each statement in the classical set theory having logical value 0 or 1, the set of these statements is called a *crisp set*, and also, these sets have a logic associated, ($T/F$) due to the membership of their elements. Thus, sets and logic are two sides of the same coin, they go together. Various properties of fuzzy sets, like *relations*, *logic*, and *inferences*, have counterparts in the classical sets [1].

To propose a formal representation, a fuzzy set $A$ is represented as $A = \{\frac{u}{a(u)} \mid u \in U\}$, where $u$ is an element, and $a(u)$ the membership function, called *characteristic function*, and represents the degree of belongingness; $U$ is universe set. The notation $u \in U$ means "every $u \in U$".

The Fuzzy operations have counterparts operations in classical set theory as *union*, *intersection*, *complement*, *binary relation*, and *composition* of relations.

The common definitions of fuzzy set operations are as follows.

$$A \cup B = \left\{ \frac{u}{max(a(u), b(u))} \mid u \in U \right\} \tag{12.33}$$

$$A \cap B = \left\{ \frac{u}{min(a(u), b(u))} \mid u \in U \right\} \tag{12.34}$$

and complement of $A$ is expressed as

$$A' = \left\{ \frac{u}{1 - a(u)} \mid u \in U \right\}. \tag{12.35}$$

It is possible to derive fuzzy versions of familiar properties of ordinary sets, such as *commutative* laws, *associative laws*, and *De Morgan's laws*, based on the above definitions [6].

The definition of fuzzy logic has semantic issues also; as per that fuzzy logic it has two different senses: 1. It is a logical system, which is aimed to formalize the approximate reasoning, 2. It is rooted in a multivalued logic, but its objective is quite different from traditional multivalued logical system. It accounts for a concept of linguistic variables (large, small, big, old, cloudy, etc.), canonical form, fuzzy if-then rule, fuzzy quantifiers, etc. In a broad sense, fuzzy logic is governed by the fuzzy set theory.
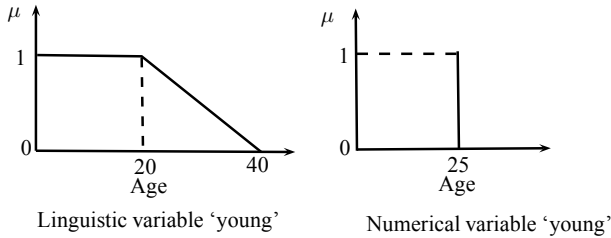
**Fig. 12.14** Linguistic and numerical variable "young"

The fuzzy arithmetic, fuzzy mathematical programming, fuzzy topology, fuzzy graph theory, and fuzzy data analysis are other branches of fuzzy set theory. It is indeed quite likely that most theories will be *fuzzified* in this way. The impetus for the transition from a crisp set theory to fuzzy theory results from the fact that both—the applicability to real-world problems and generality of the theory—are substantially enhanced when the concepts of a set are replaced by a fuzzy set.

The concept of a *linguistic variable* plays a central role in the applications of fuzzy logic. Consider the linguistic variable, say *Age* whose values are *young, youth,* and *old*, with "young" defined by a membership function $\mu$ as shown in Fig. 12.14. It is interesting to note that a numerical value such as 25 years is definitely simpler than a function like *young*. The value *young* represents a choice out of three values—*young, youth, old*—but 25 represents a choice, which is out of 100 values. Consequently, the linguistic variable *young* may be viewed as a method of *data compression*.

We achieve the same effect in quantization (conversion of an integer into a binary number), where the values are in intervals. In contrast to quantization, the values are overlapping in fuzzy sets. The advantage of granulation over quantizations are 1. granulation is more general, 2. it mimics the way humans interpret linguistic values (i.e., not as intervals), and 3. a transition from one linguistic value to the next higher and lower is gradual rather than abrupt. This results in a *continuity* and *robustness* of the system.

## 12.6.1  Fuzzy Composition Relation

Let us assume that $A$ and $B$ are fuzzy sets of universes $U$ and $V$, respectively. The Cartesian product $U \times V$ is defined just like for ordinary sets. We define the Cartesian product $A \times B$ as follows,

$$\left\{ \frac{(u, v)}{min(a(u), b(v))} \mid u \in U, v \in V \right\}. \tag{12.36}$$

Every fuzzy binary relation (or a mapping relation or fuzzy relation), from fuzzy set $U$ to fuzzy set $V$, is a fuzzy subset of relation $R$ defined below. Here $m(u, v)$ is called a *membership function*, having the range [0, 1].

$$R : U \times V = \left\{ \frac{(u, v)}{m(u, v)} \mid u \in U, v \in V \right\}. \tag{12.37}$$

A fuzzy relation $R$ from set $A$ to $B$ can be defined as

$$R_{AB} = \left\{ \frac{(u, v)}{m(u, v)} \mid m(u, v) \le a(u), m(u, v) \le b(u), u \in U, v \in V \right\}. \tag{12.38}$$

Consider that a similar relation $R_{BC}$ exists from fuzzy set $B$ to $C$. As in classical sets, a *composition* relation of relations $R_{AB}$ and $R_{BC}$, i.e., $R_{AB} \circ R_{BC}$ is obtained as

$$R_{AB} \circ R_{BC} = \bigcup \left\{ max \left[ \frac{(a, c)}{min(m_R(a, b), m_S(b, c))} \right] \right\}. \tag{12.39}$$

The *fuzzy sets* are extensions and more general forms of ordinary sets; accordingly, the *fuzzy logic* is an extension of the ordinary logic. Just as there are correspondences between ordinary sets and ordinary logic, there exists correspondence between fuzzy set theory and fuzzy logic. For example, the set operations of *union* (*OR*), *intersection* (*AND*), and *complement* (*NOT*) exists in classical set theory (classical logic) as well as in fuzzy system. The degree of belongingness of an element in a fuzzy set corresponds to the truth value of the proposition in fuzzy logic.

A *fuzzy implication* is viewed as describing a relation between two fuzzy sets. Using fuzzy logic we can represent a fuzzy implication such as $A \rightarrow B$, i.e., if "$A$ then $B$", where $A$ and $B$ are fuzzy sets. For example, if $A$ is fuzzy set *young*, and $B$ is fuzzy set *small*, then $A \rightarrow B$ may mean, if *young* then *small*. A simple definition is $A \rightarrow B = A \times B$, where $A \times B$ is the Cartesian product of fuzzy sets $A$ and $B$.

### 12.6.1.1  Inferencing in Fuzzy Logic

Let $R$ be a fuzzy relation from set $U$ to $V$ (i.e., $R : U \rightarrow V$), $X$ be a fuzzy subset of $U$, and $Y$ be a fuzzy subset of $V$. Given these, we can define a *composition rule* of *fuzzy inference* as

$$Y = X \circ R \tag{12.40}$$

where $Y$ is said to be induced by $X$ and $R$. A *fuzzy inference* is based on fuzzy implication and the compositional rule of inference. A fuzzy inference is defined as follows:

Given:
*Implication*: "If $A$ then $B$"
*Premise*: "$X$ is true",

Derive:
*Conclusion Y*.

To derive a fuzzy inference, we perform the following steps:

1. Given, "if *A* then *B*", compute the fuzzy implication as a fuzzy relation, $R = A \times B$,
2. Induce *Y* by $Y = X \circ R$.


## 12.6.2  *Fuzzy Rules and Fuzzy Graphs*

We have noted that linguistic variable in fuzzy logic behaves as a source of data compression, as a linguistic variable may stand for a large number of values. The fuzzy logic, in addition, provides *fuzzy if-then rule* or simply the *fuzzy rule*, and a *fuzzy-graph*. The fuzzy rule and fuzzy graph bear the same relation to numerical value dependencies that the linguistic variable has with numerical values [8].

Figure 12.15 shows a fuzzy graph $f^*$ of a function dependence $f : X \rightarrow Y$, where $X \in U$, $Y \in V$, are linguistic variables. Let the set of values in linguistic variables *X* and *Y* be $A_i$, $B_i$, respectively. This graph is an approximate compressed representation of *f* in the form of $f^*$ as

$$f^* = A_1 \times B_i + A_2 \times B_2 + \ldots + A_n \times B_n, \tag{12.41}$$

which is equal to

$$f^* = \sum_{i=1}^{n} A_i \times B_i. \tag{12.42}$$

$A_i$, $B_i$ (for $i = 1 \ldots n$) are called *contiguous fuzzy subsets* of sets *U* and *V*, respectively. Also, $A_i \times B_i$ is the Cartesian product of $A_i$, $B_i$, and "+" in Eq. (12.41) represents the operation of *disjunction*—a union operation. When expressed more explicitly in the form of *membership functions*, we have

**Fig. 12.15** A function and corresponding fuzzy graph
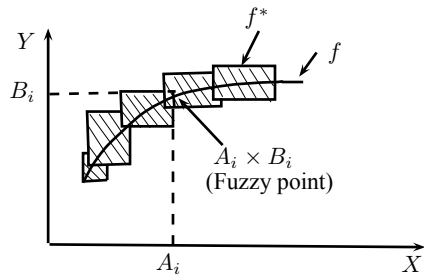
**Table 12.5** Fuzzy relation $f^*$

| $f^*$ | A | B |
|---|---|---|
| | $A_1$ | $B_1$ |
| | $A_2$ | $B_2$ |
| | . | . |
| | $A_n$ | $B_n$ |

$$\mu_{f^*}(u, v) = V_i(\mu_{A_i} \wedge \mu_{B_i}). \tag{12.43}$$

where $\mu$ is the membership function. The operation of $\wedge$ is *min*, $\vee$ is *max*, $u \in U$, and $v \in V$.

A fuzzy graph can also be represented as a *fuzzy relation $f^*$*, as shown in Table 12.5.

Also, a fuzzy graph can be represented as a collection of *if-then* rules as follows:

$$f^* : \; if \; X \; is \; A_1 \; then \; Y \; is \; B_1$$
$$if \; X \; is \; A_2 \; then \; Y \; is \; B_2$$
$$\cdots$$
$$if \; X \; is \; A_n \; then \; Y \; is \; B_n. \tag{12.44}$$

In other words, "$(X, Y)$ is $A_i \times B_i$". Given a fuzzy *if-then* rule set as follows,

$$f^* : \; if \; X \; is \; small \; then \; Y \; is \; large$$
$$if \; X \; is \; medium \; then \; Y \; is \; medium$$
$$\cdots$$
$$if \; X \; is \; large \; then \; Y \; is \; small. \tag{12.45}$$

This rule can be represented equivalently as a fuzzy graph (see Fig. 12.16) using the fuzzy expression for $f^*$,

$$f^* = small \times large + medium \times medium + \cdots + large \times small. \tag{12.46}$$

An important concept in a fuzzy graph is any type of function (or relation) that can be approximated by a fuzzy graph. For example, a normal distribution for probability

**Fig. 12.16** Fuzzy graph for a fuzzy relation (Eq. 12.46)
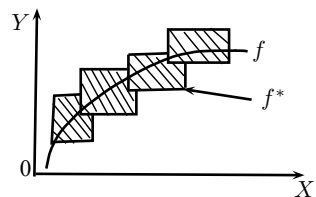
**Fig. 12.17** Fuzzy graph for a probability distribution
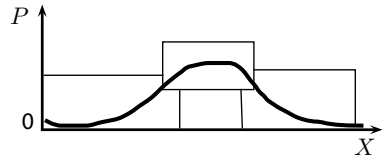


is represented as a fuzzy graph, as shown in Fig. 12.17. This representation is useful in decision-making and fault diagnosis. Here, the probability $P$ is represented by

$$P = small \times medium + medium \times large + large \times small. \qquad (12.47)$$

The constraint for probability is that it sums to 1, i.e., $\Sigma_i P_i = 1$.

### 12.6.3 Fuzzy Graph Operations

The interesting point in fuzzy calculus is that it aims to develop computational procedures for basic operations on fuzzy graphs. These operations are generalizations of basic operations on crisp sets and functions, and can be defined as follows: The required computations can be simplified if the operation of "$*$" is taken as monotonically increasing. That is, if $a$, $b$, $a'$, and $b'$ are real numbers, then the following holds:

$$a' \geq a, b' \geq b \Rightarrow a' * b' \geq a * b$$
$$a' \leq a, b' \leq b \Rightarrow a' * b' \leq a * b. \qquad (12.48)$$

From these operations, it can be easily deduced that "$*$" is a distributive operator over "$\vee$ (*max*)" and "$\wedge$ (*min*)". Accordingly,

$$a * (b \vee c) = a * b \vee a * c$$
$$a * (b \wedge c) = a * b \wedge a * c. \qquad (12.49)$$

From the above, we conclude that if

$$f^* = \sum_i A_i \times B_i \qquad (12.50)$$

is a fuzzy graph and $C$ is a fuzzy set, then

$$C * \left( \sum_i A_i \times B_i \right) = \sum_i C * (A_i \times B_i). \qquad (12.51)$$
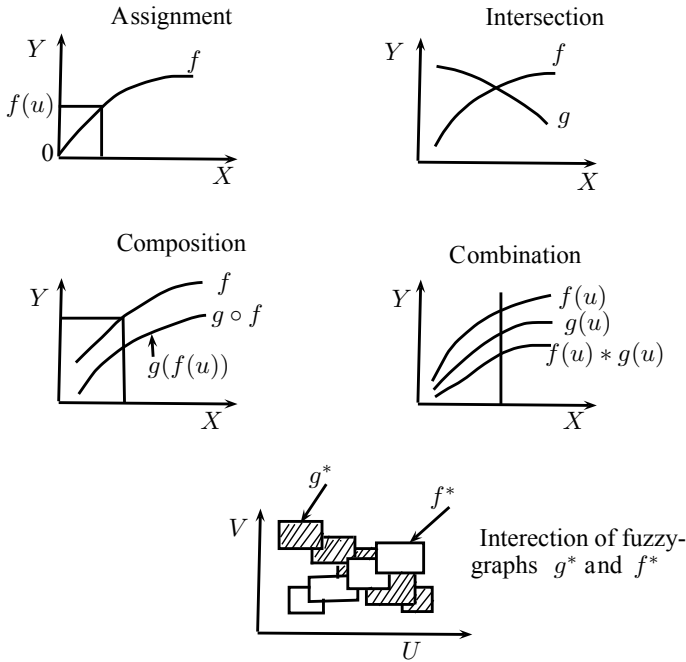
**Fig. 12.18**  Basic operations on functions and relations of fuzzy sets

Consider finding the *intersection* of fuzzy graphs (Fig. 12.18) $f^*$ and $g^*$, where

$$f^* = \sum_i A_i \times B_i \tag{12.52}$$

and

$$g^* = \sum_j C_j \times D_j. \tag{12.53}$$

From $f^*$ and $g^*$, we obtain the fuzzy intersection as

$$f^* \cap g^* = \sum_{i,j} (A_i \times B_i) \cap (C_j \times D_j) \tag{12.54}$$

and, the view of distributivity of "∩" ultimately reduces to

$$f^* \cap g^* = \sum_{i,j} (A_i \cap C_j) \times (B_i \cap D_j). \tag{12.55}$$

These operations on *fuzzy functions* and *fuzzy relations* are shown in Fig. 12.18.

### *12.6.4 Fuzzy Hybrid Systems*

Different forms of hybrid systems are designed as combinations of Fuzzy Logic, Neural Networks, and Genetic Algorithms. The fundamental concepts of such hybrid systems is to complement each other's weaknesses, that creates a new approach to solving problems. For example, in fuzzy logic systems, there is no learning capability, there is no memory, and there is no capability of pattern recognition. Fuzzy systems combined with neural networks will have all these three capabilities.

Fuzzy systems are extensively used for control, like fuzzy controllers, specially fuzzy PID (proportional, integral and derivative) controllers.

In spite of their wider applicability, the fuzzy systems have the following limitations: The stability is a major issue in their control. There is no formal theory guaranteeing their stability. Fuzzy systems have no memory and lack the learning capabilities. Determining good membership functions and tuning fuzzy systems are not always easy. In addition, verification and validation of fuzzy systems require extensive testing. In light of these limitations, the fuzzy systems are being used more and more, and they become better suited when combined with the neural networks and genetic algorithms.

## 12.7 Summary

Probability and Bayes theorem
In classical logic, a proposition is always taken as either true or false, while in real life, truth value of a proposition may be known only with a certain probability. This requires probabilistic reasoning. Two basic approaches for reasoning are *probabilistic reasoning* and *non-monotonic reasoning*.

Many different methods exist for representing and reasoning with uncertain knowledge, these include Bayesian networks, Dempster–Shafer theory, possibilistic logic, and fuzzy Logic.

Bayesian networks offer a powerful framework for the modeling of uncertainties. These networks are represented in two different ways: (1) Qualitative approach, by means of *directed acyclic graphs*, and (2) Quantitative approach, by specifying a *conditional probability distribution* for every variable present in the network. The conditional probability is represented by

$$P(A|B) = \frac{P(A, B)}{P(B)} \tag{12.56}$$

where comma denotes the conjunction of events.

The Bayes theorem is used to find out conditional probability. The theorem is stated as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{12.57}$$

In a Bayesian network, a node corresponds to a variable, which assumes values from a collection of mutually exclusive and collective exhaustive states, and the edges represent relevance or influences between variables. As per the definition of Bayesian networks, it is required to access the probability distribution of the node conditional on every instance of its parents. Thus, if a node has $n$-binary-valued parents, there are $2^n$ probability distributions for that node. A *probabilistic inference* is nothing but computing probabilities of interest from a (possibly implicit) joint probability distribution. All exact algorithms for probabilistic inference in Bayesian networks exploit the criteria of *conditional independence*.

Solving the problem of finding the exact inference in an arbitrary Bayesian network is NP-hard.

The inferencing in a network amounts to *propagation of probabilities* of a given and related information through the network to one ore more conclusion nodes.

The subjective approach for the construction of Bayesian networks reflects on the available knowledge (typically, perceptions about causal influences). Other method for Bayesian networks is based on automatically synthesizing these networks for the purpose of *reliability* and *diagnosis*. This approach is used to synthesize a Bayesian network automatically with the help of formal knowledge of system design.

The third method for constructing Bayesian networks is based on learning from data, such as medical records, customers' purchases data, or data records of customers applied for finances. These data sets are used to learn the network parameters, given their structures, or can learn both the structure and network parameters.

Dempster–Shafer Theory

The Dempster–Shafer Theory (DST) of evidence is used to model several single pieces of evidence within a single hypothesis relations, or a single piece of evidence within multi-hypotheses relations.

The combination rule of DST aggregates two independent bodies of evidence into one body of evidence, provided that they are defined within the same frame of discernment.

The DST is based on two ideas: 1. to obtain the degree of belief for one question using the subjective probabilities of a related question, and 2. making use of Dempster's rule to combine degrees of two belief which are based on independent items of evidence.

In DST, the difference $(100 - X)\%$ with respect to a probability of $X\%$ for a diagnosis is not valued against the diagnosis, but interpreted as an uncertainty interval, i.e., the probability of a diagnosis is not $X\%$, but lies between $X$ and $100\%$.

Evaluation of probabilities in DST is more complicated than in Bayes theorem, because the probabilities for a set of diagnoses must be related to one another, and the probability of a particular set of diagnoses must be calculated from the distribution of probabilities over all sets of diagnoses.

Fuzzy logic, sets, and reasoning

A fuzzy set is a generalization of ordinary set by allowing the degree of membership a real number [0, 1], in contrast to the membership of 0 and 1 in ordinary set theory. The

fuzzy operations that have counterparts in classical set theory are *union*, *intersection*, *complement*, *binary relation*, and *composition* of relations.

Other branches of fuzzy set theory are fuzzy mathematical programming, fuzzy arithmetic, fuzzy topology, fuzzy graph theory, and fuzzy data analysis.

In fuzzy logic systems, there is no learning capability, no memory, and no capability of pattern recognition. Fuzzy systems combined with neural networks will have all these three capabilities.

Different forms of hybrid systems are constructed using a combination of fuzzy logic and other areas. These are Fuzzy-neuro systems, Fuzzy-GA (fuzzy and genetic algorithms), and Fuzzy-neuro-GA. They help to complement each other's weaknesses, and create new systems that are more robust to solving problems.

## Exercises

1. We assume a domain of 5-card poker hands out of a deck of 52 cards. Answer the following under the assumption that it is a fair deal.

   a. How many 5-card hands can be there (i.e., number of atomic events in joint probability distribution).
   b. What is the probability of an atomic event?
   c. What is the probability that a hand will comprise four cards of the same rank?

2. Either prove it is true or give a counterexample in each of the following statements.

   a. If $P(a \mid b, c) = P(a)$, then show that $P(b \mid c) = P(b)$.
   b. If $P(a \mid b, c) = P(b \mid a, c)$, then show that $P(a \mid c) = P(b \mid c)$.
   c. If $P(a \mid b) = P(a)$, then show that $P(a \mid b, c) = P(a \mid c)$.

3. Consider that for a coin, the probability that when tossed the head appears up is $x$ and for tails it is $1 - x$. Answer the following:

   a. Given that the value of $x$ is unknown, are the outcomes of successive flips of this coin independent of each other? Justify your answer in the case of head and tail.
   b. Given that we know the value of $x$, are the outcomes of successive flips of the coin independent of each other? Justify.

4. Show that following statements of conditional independence,

$$P(X \mid Z)P(Y|Z) = P(X, Y \mid Z)$$

are also equivalent to each of the following statements,

$$P(Y \mid Z) = P(Y \mid X, Z),$$

and

$$P(X \mid Z) = P(X \mid Y, Z).$$

5. Out of two new nuclear power stations, one of them will give an alarm when the temperature gauge sensing the core temperature exceeds a given threshold. Let the variables be Boolean types: $A$ = alarm sounds, $FA$ =alarm be faulty, and $FG$ = gauge be faulty. The multivalued nodes are $G$ = reading of gauge, and $T$ = actual core temperature.

   a. Given that the gauge is more likely to fail when the core temperature goes too high, draw a Bayesian network for this domain.
   b. Assume that there are just two possible temperatures: *actual* and *measured*, *normal* and *high*. Let the probability that the gauge gives the correct reading be $x$ when it is working, and $y$ when it is faulty. Find out the conditional probability associated with $G$.
   c. Assume that the alarm and gauge are correctly working, and the alarm sounds. Find out an expression for the probability in terms of the various conditional probabilities in the network, that the temperature of the core is too high.
   d. Let the alarm work correctly unless it is faulty. In case of being faulty, it never sounds. Give the conditional probability table associated with $A$.

6. Compute the graphical representation of the fuzzy membership in the graph 12.19 for following set operations:

   a. *Small* $\cap$ *Tall*
   b. (*Small* $\cup$ *Medium*)–*Tall*

**Fig. 12.19** Membership functions: *Small, Medium, Tall*
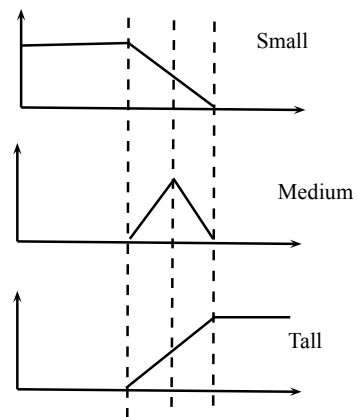
7. Given the fuzzy sets $A = \{\frac{a}{0.5}, \frac{b}{0.9}, \frac{e}{1}\}$, $B = \{\frac{b}{0.7}, \frac{c}{0.9}, \frac{d}{0.1}\}$, Compute $A \cup B$, $A \cap B$, $A'$, $B'$.

8. Let $X = \{a, b, c, d, e\}$, $A = \{\frac{a}{0.5}, \frac{c}{0.3}, \frac{e}{1}\}$. Compute:

    a. $\bar{A}$
    b. $\bar{A} \cap A$
    c. $\bar{A} \cup A$

# References

1. Chowdhary KR (2015) Fundamentals of discrete mathematical structures, 3rd edn. EEE, PHI India
2. Darwiche A (2010) Bayesian networks. Commun ACM 53(12):80–90
3. http://web.mit.edu/jmn/www/6.034/d-separation.pdf. Cited 19 Dec 2017
4. Heckerman D, Wellman MP (1995) Bayesian networks. Commun ACM 38(3):27–30
5. Kay RU (2007) Fundamentals of the dempster-shafer theory and its applications to system safety and reliability modelling. J Pol Saf Reliab Asso 2:283–295
6. Munakata T, Jani Y (1994) Fuzzy systems: an overview. Commun ACM 37(3):69–77
7. Puppe F (1993) Systematic introduction to expert systems. Springer
8. Zadeh LA (1994) Fuzzy logic, neural networks, soft computing. Commun ACM 37(3):77–84

# Chapter 13
# Machine Learning

**Abstract**  To adapt to the environment it necessary that intelligent machines must have the capability to learn. This chapter presents the basic concepts and techniques of learning found in humans, as well as their implementation aspects for machines. The chapter presents the challenges of building learning capabilities in machines, types of machine learning, and the relative efforts needed to build these learning capabilities. The philosophy of the discipline of machine learning is presented. The basic model of learning is discussed, followed by the classes of learning—supervised and unsupervised—then various techniques of inductive learning—argument based learning, online concept learning, propositional and relational learning, and learning through decision trees—are presented in sufficient details. Other techniques like discovery-based learning, reinforced learning, learning and reasoning through analogy, explanation-based learning are presented, with some worked examples. Finally, the potential applications of machine learning, the basic research problems in machine learning, followed by chapter summary, and a set of exercises are appended.

**Keywords**  Machine learning · Machine learning classes · Machine learning model · Supervised learning · Unsupervised learning · Inductive learning · Decision-tree-based learning · Argument-based learning · Propositional learning · Discovery-based learning · Learning through analogy · Reinforced learning · Explanation-based learning · Machine learning applications · Research problems

## 13.1  Introduction

Learning is one of the most important activities of human beings and living beings in general, which help us in adapting to the environment. Humans learn from nature as well as from special learning environments through different techniques that vary in complexity. The learning requires transformations of ideas and information structures in the human mind.

This chapter presents the basic concepts and examples of learning techniques found in humans, and their implementations aspects for learning in machines. A common view holds that learning involves making changes in the learning system

that will improve it in some way. These changes are *adaptive*, because it is found that the system does some task or tasks more effectively the next time when they are drawn from the same population of tasks.

Most of the knowledge of the world we have around us, and about the explicit set of tasks, is not formalized, and even not available in text form. The latter is a necessary requirement for it to be understood by any computer program. This is the reason it is not easy to write a program for a computer to do many tasks that we humans do so easily, such as understanding spoken sentences, images, languages, or driving a car. Any attempt to achieve this, i.e., organizing sets of facts in elaborate data structures to enable computers to understand it, has achieved very little success.

The concept of *learning* is based on the principles of training the computing machines, and enabling them to teach themselves. Tenet of these machines/programs is related to what we consider as good decision. For humans and animals, evolutionary principles dictate that any decision made by them should lead to such behaviors that optimize the chances of *survival* and *reproduction*. In the societies of human beings, a good decision is one that might include social interactions that bring *status*, or a sense of *well-being*. For a machine such as a self-driving car, such criteria cannot exist! So the quality of decision-making for a self-driving car is measured based on the criteria as to how close the autonomous vehicle imitates the behavior of trained human drivers.

It is important to note that the knowledge required to make a good decision in a particular situation is not necessarily clear to the extent to code it into a computer program. Consider, for example, that a mouse has knowledge about its surroundings, and has an innate sense of where to sniff and how to find food or mates, and at the same time save itself from predators. It is not easy for any programmer to specify step-by-step procedures or a set of instructions to produce such behaviors for some artificial mouse. However, this all remains encoded in the brain of a mouse in the form of knowledge.

Before we think about creating computers (programs) that can train themselves, it is important to answer the fundamental question as to how we humans acquire knowledge. Partly, the knowledge in humans may be innate, but most of it is acquired through experience and observation. What we know intuitively has been often learned from examples and practice—the process which cannot be turned into a clear sequence of steps as a computer program. Since 1950 people have looked for ways and means and tried to refine general principles that allow animals or humans, to acquire knowledge through experience. These are the subjects of discussion in this chapter. Machine learning aims to create theories and procedures—*learning algorithms*—that allow machines to learn. One such is learning from examples presented to them, called learning by examples [2].

**Learning Outcomes of This Chapter**:

1. List the differences among the three main styles of learning: supervised, reinforcement, and unsupervised. [Familiarity]

2. Identify examples of classification tasks, including the available input features and output to be predicted. [Familiarity]
3. Explain the difference between inductive and deductive learning. [Familiarity]
4. Implement simple algorithms for supervised learning, reinforcement learning, and unsupervised learning. [Usage]
5. Determine which of the three learning styles is appropriate to a particular problem domain. [Usage]
6. Evaluate the performance of a simple learning system on a real-world data set. [Assessment]
7. Characterize the state of the art in learning theory, including its achievements and its shortcomings. [Familiarity]

## 13.2 Types of Machine Learning

The field of machine learning is concerned with the development of computational theories for learning processes and building learning machines. Because the ability to learn is a fundamental to any intelligent behavior, the concerns and goals of machine learning are central to the discipline of Artificial Intelligence. In the learning process, the learner transforms the information provided by a teacher (or environment) into some new form, and these are stored in that form for use in the future. The nature of this knowledge and transformation are the deciding factors for the type of learning strategy used. Following are the fundamental *strategies of learning*.

Rote Learning
Learning by Instruction
Learning by Deduction
Learning by Analogy
Learning by Induction (or Similarity)
Reinforcement Learning
Discovery-based Learning

These strategies are in increasing order of complexity of transformation required in the initial information, which is stored as knowledge base. The complexity orders of these methods are such that, an increasing difficulty level on the part of the student (learner) results in a correspondingly decreasing complexity on the part of the teacher (environment), and vice versa. Following is the brief introduction about each of these methods.

Rote Learning

The *rote* learning is the simplest among all the learning, and requires the least amount of inferencing from the knowledge. To accomplish the inferencing later, the original knowledge is copied in the same form in which it will be used later. Hence, the information from the teacher is more or less directly accepted and memorized by the

learner. This kind of learning is used when we learn the tables in the early school classes. A major concern in this learning is how to index the stored knowledge for future retrieval.

Learning by Instruction

The learning which is next higher in efforts after rote learning is learning by *instruction*. This approach requires the knowledge to be transformed into an operational form before it can be integrated into the knowledge base. This learning takes place among the students in a class when the class teacher presents a number of facts directly in a properly organized form to the students.

In learning by instruction (or *learning by being told*), the basic transformations performed by a learner are *selection* and *reformation* (mainly at a syntactic level) of information provided by the teacher.

Note that, depending on how the teachers presents the knowledge in the class, different types of learning take place, and learning by instructions is not the only learning induced by a teacher into his/her students.

Learning by Deduction

*Deductive learning* is carried out through a sequence of deductive inference steps using known facts. Using this, new facts and knowledge are logically derived. For example, if we have the knowledge of a rule, say "a father's father is a grand father," and we have the case that $a$ is the father of $b$ and $b$ is the father of $c$, then we can say that $a$ is the grand father of $c$. Here we have deduced the result, hence the name deductive learning.

Learning by Analogy

*Analogical learning* provides learning of new concepts through the use of similar known concepts or solutions. This is used, for example, when we solve some problems in our college examination, based on the solution of somewhat similar solutions we have already done at home. Also, for example, learning to drive a bike when we already know how to drive a bicycle is an example of analogical learning.

Learning by Induction

We make use of an inductive learning when after seeing a number of instances or examples of a concept we formulate a general concept.

In any human learning activity, mostly the strategies out of those discussed above are involved. It is important to understand the difference between these strategies on the part of designing a learning system. Though most current artificial systems focus on any single learning strategy, one may expect that machine learning research will give increasing attention to a multi-strategy approach, the one which is close to the human learning system. We as a learner use almost always a combination of strategies in our daily life. For example, remembering a telephone number, we use rote learning; seeing someone wearing a new dress every day, we learn that so-and-so is rich through inductive learning. Learning that those students who complete their

homework in time get more Cumulative Percentile Index (CPI) is also an example of inductive learning.

Another example, when we learn Java language having learned *C++* is analogical learning. When we are asked to solve a new puzzle, it is *discovery-based learning*. Whenever, a teacher appreciates the student's efforts by saying "very good", for asking an interesting question, or answering a question, it is *reinforcement-based learning* for the student.

When you have concluded based on some clues that a teacher is not in a mood of teaching, most likely you have used deductive learning (or reasoning).

## 13.3 Discipline of Machine Learning

Any field of scientific study is best defined by the central question it asks. That way, the field of Machine Learning seeks the answer to the following questions:

> 1. How to build a computer system that can automatically improve its performance with experience, and 2. What are the fundamental laws that govern the learning processes?

The faculty of *Learning*, like the faculty of *Intelligence*, involves such a vast range of processes that it is difficult to define it precisely. A dictionary definition of learning includes phrases such as "to gain knowledge, or understanding of, or skill in, by study, instruction, or experience." Another dictionary definition says, learning is "modification of a behavioral tendency by experience." Zoologists and psychologists also study the learning processes in animals and humans.

This chapter focuses on learning in machines. There are several similarities between learning in animals and in machines. Many techniques in machine learning are derived from the theories of animal and human learning and are presented in the form of computational/cognitive models by psychologists. It is quite likely that the concepts and techniques being explored by researchers in machine learning may illuminate certain aspects of biological learning.

Regarding the machines, it is roughly said that a machine learns whenever it changes in its structure or program or data, based its input or in response to external information. This change takes place in such a manner that its expected future performance improves. A change like addition of a record in a database, though makes a change in data structure, does not fall in the discipline of learning. As another example, the performance of a speech-recognition machine improves after hearing several samples of a person's speech—we feel convinced that the machine has learned. This process is obviously a learning process.

Machine learning usually refers to the changes in systems that perform tasks such as recognition, diagnosis, planning, robot control, prediction, and their combinations. These tasks are usually associated with artificial intelligence (AI). The "changes" mentioned above is in one of the two types: 1. enhancements in already performed task by a system, or 2. creating special ability in the system.

One obvious question related to machine learning is "Why should machines be built to learn, and why not to build them to perform the desired task in the first place?" There are several reasons why machine learning is important. The one we have already mentioned is that understanding the learning process will help us in building theories which will help us understand how the animals and humans learn. It is like when we learned optics, we were able to better understand human vision system. However, there are the following important reasons why we should study machine learning, and why there should be machines that learn:

(a) There are many tasks that cannot be defined well in advance, except by examples. Like many times we are able to specify input/output relations as pairs of data but a precise relationship between inputs and desired outputs cannot be specified. For example, the weight-to-height ratio of a male as input and health as output. We would like that based on these input–output sample examples, the machine be able to adjust its internal structure, so that for new inputs it produces correct outputs, and thus suitably constrain its input/output function to approximate the relationship implicit in the given examples.

(b) It is possible that among large sets of data, like—credit cards, medical records, weather data, astronomical data, etc., there are hidden important relationships and correlations. The machine learning methods can often be used to extract these relationships and patterns through the process of *data mining*, a field of machine learning.

(c) The machines produced by the human designers often do not work as desired in certain environments. In fact, when the machine was designed, certain characteristics of the working environment might not be completely known at design time, hence could not be incorporated in the design. This often happens in space explorations—like missions to Moon, Mars, Saturn, and even to Sun. Machine learning methods can improve/modify the performance of the system on the site in the existing machine design.

(d) The amount of knowledge available and required for certain tasks may be too large for explicit encoding by humans. But machines can be designed that learn this knowledge gradually with time, and might be able to capture it continuously on the fly as it needs.

(e) There is no need to redesign the machines if there are machines that can adapt to a changing environment. Such machines, once designed, can learn with situations, and functions for a long time—for example in long planetary missions.

With the continuous redesign of AI, the changes are taking place in the world knowledge as well as vocabulary.

The questions in the domain of AI cover a broad range of learning tasks, like: How to design an autonomous mobile robot that learns to navigate using its own

experience? How to mine the data of historical medical records to learn that a given patient $X$ will respond best to what treatments? How to build search engines that automatically customize to the user's interests and other similar questions.

The following is a more precise definition of machine learning.

**Definition 13.1** (*Machine Learning*) A machine learns with respect to a particular task $T$, performance metric $P$, and type of experience $E$, if it reliably improves its performance $P$ at task $T$, following the experience $E$.

Depending on how we specify $T$, $P$, and $E$, the learning task might also be called by names such as *data mining*, *autonomous discovery*, and *database updating*, *programming by example*.

The field of machine learning focuses on how to make computers that program themselves through their experience, with provision of some initial structures. This is in contrast to the field of computer science that so far focused primarily on manually programming computers.

Other fields that are closely related to Machine Learning are Psychology and Neuroscience, which are concerned with the study of human and animal learning, and are collectively called *Cognitive Science*. The questions—How do animals learn, and how can computers be made to learn—most probably have highly intertwined answers. The insights into machine learning far outweigh due to contributions from the fields of statistics and computer science than due to studies in human learning behavior. The main reason behind this is the poor state of understanding about human learning processes. However, the relation between machine learning and human learning behavior is continuously becoming stronger, mainly due to the new machine learning algorithms, such as *temporal difference learning*. These algorithms are being suggested as the explanations for *neural signals* observed in learning in the animals.

The primarily goal of machine learning is to design computers that can adapt and learn from their experience without intervention from programmers. Following is an example: the present software engineering, data-intensive software construction and testing, is aimed to construct programs that are correct and robust. It is data-intensive software construction, i.e., given a set of input data to be processed, code functions that produce output (data), such that the code should correctly function even when input data changes, and even on the face of incorrect data, should reject them rather than producing wrong results. A machine can be designed to learn from the pattern of data and continuously improve its performance until it reaches the desired performance. The error (difference) between the expected performance and the actual performance can be fed back to act as a tuning to the software, that is, to make it learn. This latter is the automation of human-intensive task of software engineering that we call testing.

Machine learning methods for software engineering are appropriate whenever *hand engineering* of software is difficult and yet data is available to be analyzed by learning algorithms. Following are the situations where machine learning algorithms are expected to outperform humans:

 (i) human expertise is lacking compared to the machine intelligence in genome
     programs—the genome programs are basically evolution process, which can be
     used in applications, like drug design;
 (ii) the characteristics of the task changes frequently, like in airline seat sales strate-
     gies, where the price is decided by supply-versus-demand ratio;
(iii) humans have limited capability of introspection, like required in computer vision,
     speech recognition, and motor control;
(iv) a learning program can be customized for individual users, e.g., information
     filtering, user interfaces, language, etc.

However, there is yet one limitation of machine learning—the field of machine
learning is inherently stochastic—hence its application may not be appropriate for
tasks where completely error-free performance can be guaranteed, e.g., in life-support
control in intensive care.

## 13.4  Learning Model

The process of learning requires that new knowledge structures be created due to
some form of input stimuli. Then, this new knowledge is assimilated in the existing
knowledge and is tested for its utility and efficiency. That is, it will be used in per-
forming some task. Figure 13.1 shows a general learning model. The environment is
part of the overall system for learning. The environment may be taken as a teacher or
the one provided by nature which produces random stimuli, or a room with obstacles
where a robot needs to navigate from one wall to another, or may be a classroom
with a teacher and students who are taught using a blackboard and chalk. The com-
munication language between a teacher and a learner may be the same in which a
knowledge base was created or may be different.

The input to the learner component may be physical stimuli or some kind
of description or is symbolic. The information by the learner component cre-
ates/modifies the knowledge structures in the knowledge base, which is used by
the performance component to carry out some tasks, like solving some problems,
game playing, etc. The response produced by the performance component is evalu-



**Fig. 13.1**  A learning model
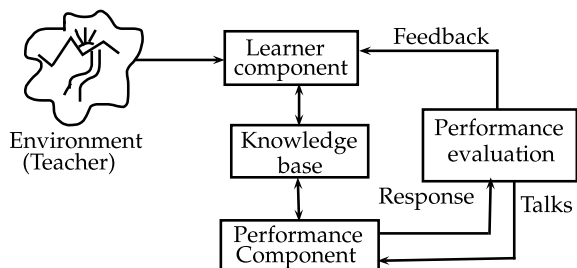
ated by the performance evaluator component, and a feedback generated by this is sent to the learner to decide if the performance is acceptable, based on which the learner updates its knowledge base. This cycle is repeated until the performance of the system is reached to the desired level.

In the sections, we present specific learning techniques in more detail. We present first the fundamental categories of learning; a learning may be basically classified as belonging to supervised or unsupervised learning.

## 13.5 Classes of Learning

When there is essentially a teacher present for learning to take place, it is *supervised learning*, and a learning achievable without a teacher is called unsupervised learning. Teacher, here, means not necessarily a physical teacher, but even nature is a teacher. For example, we put fingers in hot water and we get it burnt. So here, feedback is from hot water, which teaches us that if you again put the fingers in hot water any time in future, it would be burnt in similar way.

### 13.5.1 *Supervised Learning*

Supervised learning methods require an oracle to be made, such that it forecasts the classes (categories) to which the given examples would belong and then classifies the examples in those categories. These methods assume

(a) Existence of some teacher (environment),
(b) A fitness function to measure the fitness of an example for a class, and
(c) Some external method of classifying the training instances.

A supervised learning method uses machine learning techniques to induce a classifier from sense-annotated data sets. As an example of supervised learning for text classification, a classifier picks one word at a time, perform its word sense disambiguation (WSD), and then performs a classification task in order to assign the appropriate sense to each instance of the word.

A classifier typically learns with the help of a training set containing examples in which any given target word has been manually tagged with the sense form the sense inventory of some reference dictionary.

In general, a learning method is called *supervised* learning if the learning process requires some intervention from the user. Some approaches in supervised learning require the user to provide training examples. This is done as a pre-process to appropriately tag the examples occurring in the training set. Alternatively, it can be done as an online process where examples are dynamically tagged as needed during the learning process.

*Propositional learning* is a category of supervised learning in which examples of concepts are represented in terms of either *zero order logic* or *attribute-value logic*.

This representation has equivalent expressiveness in a strict mathematical sense. The rules are often learned from positive examples. The examples of concepts are represented as sets of attributes in the form of slots in knowledge frames, whose values are heads of syntactic phrases occurring within the training documents. The rules learned through these approaches are used in performing the tasks, like *Information Extraction (IE)* from parsed free text, *text classification*, *question-answering*, etc. The learning process identifies the syntactic phrases that contain the heads to fill these slots. Often, there is a partial match between these phrases and the exact slot filler, which is sufficient when the aim is to extract only an approximate of the relevant information. If an approximate match fails, there is necessity of post-processing.

Other class of supervised learning is *relational learning*, which is based on representing the examples of concepts in terms of *First-Order Predicate Logic* (FOPL)—as attributes and relations between textual elements. More details of propositional and relational learning are covered in sections that follow.

### 13.5.2   Unsupervised Learning

This learning eliminates the need of a teacher, and the learner is solely responsible to form his own concepts and evaluate these for learning. In fact, the unsupervised methods have to discover the concept classes to which the given examples belong, i.e., mapping the examples to concept classes.

For example, scientists are usually not blessed to have a teacher to help them pursue the research, instead they propose hypothesis to explain the observations made by them, and evaluate their hypotheses based on criteria like generality, simplicity, and elegance, and test these hypotheses through experiments designed by themselves.

Another example of unsupervised learning is *unsupervised WSD*, which is based on unlabeled corpora, and do not exploit any manually sense-tagged corpus to provide a sense choice for a word in the context.

The *discovery-based learning* is also a category of unsupervised learning.

## 13.6   Inductive Learning

The ability of human beings to master meeting the complex demands is mainly based on the human ability to exploit previous experiences, like if we have previous experience of walking through the Thar desert unaided for five miles from 10 years ago, we will not hesitate to repeat the next adventure now even for still more distance. Based on our previous experiences, we are able to predict the characteristics or relations of man-made or natural objects, we can reason about possible outcomes of actions, and we can apply the previous successful routines and strategies to new tasks and problems, as in the case of journey through a desert example. It is understood that the basic process to expand the knowledge is—first construct a hypothesis such that

we can transfer the knowledge from previous experience/patterns to new situations. This approach of learning is called *inductive inference* [6].

Inductive inference is a mechanism of generalization over the observed regularities in the examples. Every learning algorithm must make some a priori assumptions to allow for the generalization, called *inductive bias*, which also provides a rational basis to allow for the transfer of learned hypothesis to the new situation. For this transfer, the *language bias* or *restrictions* characterizes the language in which the induced hypothesis is represented. A bias for search or preference characterizes the method of selecting the proper hypothesis. Every machine learning algorithm, be it symbolic, or statistical, or neural, makes such a priori assumptions. This basic learning approach is also valid for humans, whose learning is taken as a base mechanism, that takes place at all levels, from concept acquisition to learning high-level schemata.

The acquired concepts and strategic rules need to be communicated, so that they are available as declarative structures of knowledge. Hence, inductive learning programming should be *symbolic programming*.

In the abstract, we can view the output of inductive learning as a set of *Production rules* of the form,

**In** *situation* **do** *action*

where *action* may be something overt, or an internal action, or even an inference. We use the *concept* sometimes for the right-hand side (i.e., in place of action), and the word *concept definition* or *pattern* for the left-hand side (i.e., in place of situation), to formalize the inductive learning from one or more instances in which $action_i$ was an appropriate action response to $situation_i$. Based on these instances, we infer that the general version $action_{gen}$ is the appropriate type of action in response to the general situation type $situation_{gen}$.

$$
\begin{array}{ll}
situation_1 & action_1 \\
situation_2 & action_2 \\
\dots & \dots \\
situation_{gen} & action_{gen}.
\end{array}
$$

The above example shows that inductive learning is the generalization from a set of examples; and with little pondering we note that this is one of the most fundamental learning types among humans. Learning of concepts is a typical inductive learning approach—given the examples of some concepts, such as "cat"—it will allow us to correctly recognize future instances of this concept; similarly, learning the concept that something is a "good stock investment", it will allow us to correctly perform instances of this type in the future, i.e., if an investment has been done intelligently, and has been found rewarding, there are good chances that, in future also we may adopt a similar strategy. Such instances are very common in life, hence, the most common application of inductive learning is in *concept learning*.

**Definition 13.2** (*Concept Learning*) Concept learning is, given the number of positive and negative examples of some concepts with some background knowledge, to find a general description or hypothesis of the concepts describing all the positive examples only, and ignoring all the negative examples.                          □

**Definition 13.3** (*Deductive System*) A logical system whose conclusions *logically follow* from a set of input facts, and the system is *sound*, then the system is called a deductive system.                          □

To contrast the deduction with an induction in logic-based systems, consider that we have the *training set* with the following formulas:

$$Ball(Obj_1), Round(Obj_1), Ball(Obj_2), Round(Obj_2),$$
$$Ball(Obj_3), Round(Obj_3). \tag{13.1}$$

Using these formulas if a learning system draws the conclusion $(\forall x)[Ball(x) \to Round(x)]$ then that system is an *inductive* learner. Note that this conclusion does not logically follow from the facts. Hence, conclusion is useful when there are no rules of the form $[Ball(x), Ball(x) \to Round(x)]$.

Unlike deduction, which is based on general axioms, induction is based on specific facts (examples). Induction has two goals: 1. formulate acceptable general assertions, that explain the given facts, and 2. to predict unseen facts. Thus, an inductive inference is aimed to provide a complete and correct description about a given phenomenon using specific/partial observations about it. Such inductively obtained description is true for at least already seen examples, but there is no guarantee of its correctness about the new examples.

Inductive inference methods can be used in two types of applications: 1. as an interactive tool for acquiring knowledge using examples, or 2. the method is used as part of some learning system. In the first case, a user has strong control through the examples provided by him/her, which decides the type of knowledge going to be acquired. Whereas in the second application, the inductive procedures (methods) are activated when some system component wants to learn using positive/negative examples. These examples constitute the feedback from which the system can achieve the completion of the current task.

In any relevant domain, the *background knowledge* is a deciding factor for assumptions and constraints imposed on examples and descriptions generated. The background knowledge is represented using either *declarative format* or *procedural format*. Concept learning requires searching through a large space of hypotheses (descriptions), each as a sequence of instructions for executing a specific task, with a goal to finding a hypothesis that best explains the examples. The instruction in the hypothesis are implicitly defined using the hypothesis representation language.

Inductive Programming

*Inductive* learning (also called concept learning) can be further classified according to the following perspectives, depending on whether it is based on

- Supervised learning,
- Unsupervised learning,
- Concept hierarchies, or
- Single/multi-label learning.

*Inductive programming* is concerned with *learning computer programs*, designed to work with incomplete specifications—samples of desired input/output behavior, along with some constraints. These programs are called *induced programs*, and are represented in the declarative form using functional programming or logic programming languages. To be used in machine learning, inductive programming creates program hypotheses in the form of generalized recursive programs. In contrast to the *classification-based learning*, inductive program hypotheses are supposed to cover all the given examples correctly. This is due to the fact that for a program it is expected that the desired input/output relation holds for legal inputs.

The inductive programming is used with two general approaches: 1. *generate-and-test*, and 2. *analytical* approach. The first enumerates syntactically correct programs and tests each against the given examples, with search guided through some search strategy. It is obvious that a cognitive system does not learn rules by the generate-and-test approach.

### 13.6.1 Argument-Based Learning

A challenge with machine learning is to deal with large spaces of possible hypotheses, and the search associated with that space. This problem is better addressed using expert domain knowledge to constrain the search. Further, the problem is somewhat simplified, as machine learning allows to use to some extent the general domain knowledge, which holds true for the entire domain. For constraining the search, it allows the use of "local" expert's knowledge relating to specific situations, which is valid for chosen learning examples. For this, a domain expert provides some learning examples, these together with other examples are input to the learning algorithms. These examples are explained in the form of arguments: *for* and *against*, and called as *argumented* examples; hence the learning using such examples is called *Argument-Based Learning* (ABL) [5].

**Definition 13.4** (*Learning from examples*) Having given some examples (arguments), find a theory that is consistent with them. □

To explain the idea of argument-based learning, we consider a simple learning problem called "Learning about approval of credit limit." Each example consists of information, as shown in Table 13.1, which comprises, apart from other things, the decision of the manager (Yes/ No), indicting whether the credit limit is approved or not.

**Table 13.1** Examples for credit limit approval

| Customer's name | Repayments' regularity | Rich | Height | Credit approved |
|---|---|---|---|---|
| Mrs. Red | No | Yes | Tall | Yes |
| Mr. Green | No | No | Short | No |
| Miss Blue | Yes | No | Medium | No |

An algorithm after learning from these examples will *induce* the following rule:

$$IF\ Height = Tall\ THEN\ Credit\ approved = Yes. \tag{13.2}$$

Now, we want to see how this rule will modify when it faces the arguments.

**Definition 13.5** (*Arguments-based learning*) Given the examples and supporting arguments, find a theory that explains the examples using arguments.  □

To understand what it means to "explain the examples using given arguments," see the examples in Table 13.1. Let us assume that an expert gave an *argument*: "Mrs. Red was allowed credit because she is rich." Next, we refer to the rule (13.2), which states that all tall people were approved credit. Note that this rule correctly classifies Mrs. Red (as tall), but does not include in the classification the argument (of rich) for Mrs. Red. Neither the rule mentions Mrs. Red's property, namely she is rich. Hence, we say that this rule (13.2) does not "AB-cover" Mrs. Red. Therefore, an ABL algorithm induces other rule,

$$IF\ Rich = yes\ THEN\ Credit\ approved = Yes \tag{13.3}$$

which explains Mrs. Red's example, using the given argument: credit was approved because *Rich = Yes*; now, this rule AB-covers Mrs. Red. Based on this discussion we note that the use of arguments in learning contributes to the following advantages:

1. They impose constraints over the space of possible hypotheses, thus reducing both time and space complexity of each search, enabling faster and more efficient induction of theories;
2. An induced theory is more meaningful for an expert system since it is consistent with given arguments as well as examples.

In fact, there may be many possible hypotheses from the perspective of a machine learning method that may explain the given examples sufficiently well. But these hypotheses should be simple enough to be understandable to expert (systems). Using arguments should lead to hypotheses that explain given examples in similar terms to that used by an expert, and should correspond to full justifications.

The argument-based examples is a resource for domain expert, which is added as partial domain knowledge in a learning system in advance. This prior knowledge

in the form of arguments as individual examples is simple to explain, concrete, and specific, in contrast to the general theories, which do not possess these properties. Since an argument is for a specific example, it is not required to be true for the entire domain, and suffices that it is true in the context of the given argumented example. As a test, consider a possible expert's argument to reject credit approval to Mr. Green: "Mr. Green did not get credit approval because he does not repay regularly" (see Table 13.1). This is a reasonable argument, however it does not hold for the entire domain, as Mrs. Red was approved credit, although she did not pay regularly. This generalization of the context of an argumented example is carried out by an argument-based learning algorithm only to the extent that it preserves the consistency with other examples.

### 13.6.2  Mutual Online Concept Learning

A mutual *online concept learning* system is an agent-based system (see more details in Chap. 16) with communication channel for exchange of information among the agents (see Fig. 13.2). It is assumed that the communication channel is an ideal one without any noise. A one-agent framework includes the basic elements like *concept*, *instance producing mechanism*, *instances*, *instance interpreting mechanism*, and *concept adaption mechanism* (or *online concept learning*), with communication channel shared among agents.
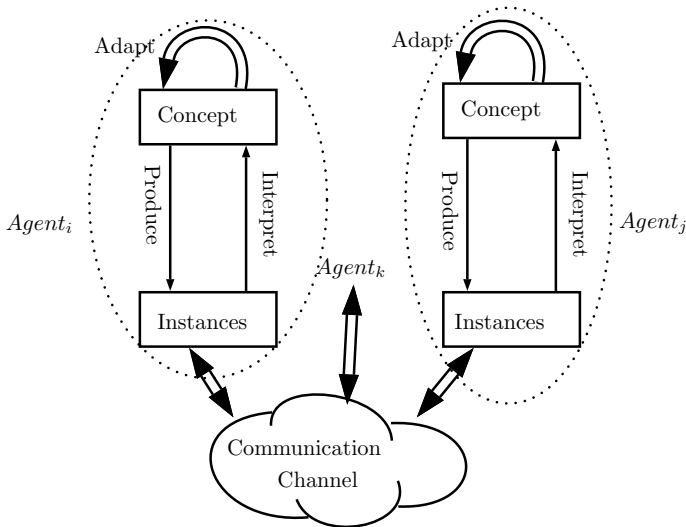


**Fig. 13.2**  Mutual online concept learning framework

In simple terms, a *concept* is a mapping from inputs to outputs. When taken as a function, it is a deterministic function, i.e., if the input is repeated, the output will follow the same sequence. When we think of it as a mapping with some probability distribution associated, then the output is also probabilistic in nature. As a deterministic function $f$, it can be formalized as $f : X \rightarrow Y$. Considering it a *Boolean concept*, the domain is $X = \{0, 1\}^n$, and its range is $Y = \{0, 1\}$. For implementing a concept, we can choose from a variety of representations, such as *propositional logic*, *first-order logic*, or *neural networks*. When a concept is viewed as a function, an *instance* $(x)$ is just a specific case in the function's domain $X$, i.e., $x \in X$ [9].

When it is required for an agent to express a concept to the outside world, it uses an *inverse function* to generate an instance using a concept, shown as *produce* operation in Fig. 13.2. There be many instances that can be used to express the same concept if the mapping from input to output is not one-to-one.

An agent may receive an instance from another agent through a communication channel, and may *adapt* its concept like probability, network connection weights, etc. This adaptation may become helpful from the point of view that the receiving agent may perform its task better next time due to updated concept or knowledge, in terms of getting more benefits/payoff. Let us assume that, for an agent $i$, an associated concept is $\mathscr{C}_i^t$ at time $t$, and for agent $j$ it is $\mathscr{C}_j^t$. Let the instance $instance(\mathscr{C}_j^t)$ be generated by $j$ at time $t$, and this instance be received by agent $i$. At the next time $t + 1$, the agent $i$ updates its concept to $\mathscr{C}_i^{t+1}$ by adapting from its previous concept (at time $t$) together with the instance of concept of agent $j$ at time $t$, i.e.,

$$\mathscr{C}_i^{t+1} = adapt(\mathscr{C}_i^t, instance(\mathscr{C}_j^t)), \quad j \neq i. \tag{13.4}$$

The representation in the learning processes, as well as the knowledge representation, is an important consideration for system performance. For real values-based instances, the space in concept learning can be of $n$-dimensions: $\mathbb{R}^n$, and $\{0, 1\}^n$ for Boolean values. The concepts learned from instances are based on linear threshold values, such that there is a hyperplane in $\mathbb{R}^n$ that separates the points on which the function is 1 from the points on which it is 0.

Figure 13.2 indicates the logic of concept based on a broad sense, but does not specify the time order of agent interactions between the agents. This interaction can be performed in any of the two modes: *serial* or *concurrent*. In the first case, at each time step only one agent is generating instances or updating concepts, and other agents should be idle. In the second (concurrent) case, two or more agents may simultaneously generate the instances and update the concepts. There is no difference between these agents while working in the concurrent mode, however when in the serial mode, the two agents are not strictly equivalent since there must be one agent that starts the mutual process first, and so makes an initial impact on the direction of formation of a concept.

### 13.6.3  Single-Agent Online Concept Learning

For the sake of simplicity, we consider a single-agent for online concept learning from a teacher/environment, that is static. In contrast to a single-agent system, the environment is continuously changing (not fixed) in a *multi-agent* online concept learning system, because the agents act on the environment shared by themselves.

The learning task to be performed is as follows: obtain the target *concept* or *function* $f^* : \{0, 1\}^n \to \{1, -1\}$, that maps each *instance*[1] to the correct *label* or *class* in response to the learning instances and feedback from the teacher [9].

A sequence of trials is needed for the online learning task to take place. In a trial, the following order is followed by the events:

1. an instance is received by a learner from a teacher;
2. the instance is labeled as 1 or −1 by the teacher;
3. the teacher tells the learner whether the label is correct or not;
4. this feedback is used by the learner to update its concept.

In the above steps, each new trial begins when the previous trial ends.
A learning algorithm called the *Perceptron algorithm* takes real-valued inputs in the form of a vector, and outputs as 1 if the result of computation is greater than a threshold ($\theta$), otherwise −1 is the output [3].

To be precise, if an input instance is $\vec{x} = (x_1, \ldots, x_n)$, the output computed by the perceptron is $f(\vec{x})$:

$$f(\vec{x}) = \begin{cases} +1 & \text{if } \sum_{i=1}^{n} w_i x_i > \theta \\ -1 & \text{otherwise.} \end{cases} \tag{13.5}$$

In the above, $\vec{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n$ is called the current weight vector of the perceptron. For the sake of convenience we take threshold $\theta$ as 0, and instead add a constant $x_0 = 1$ with a weight $w_0$. For the sake of compactness the perceptron function in Eq. (13.5) is written as

$$f(\vec{x}) = sgn(\vec{w} . \vec{x}).$$

Let $\vec{w} . \vec{x} = y$. Now, we can rewrite Eq. (13.5) as

$$sgn(y) = \begin{cases} +1 & \text{if } y > 0 \\ -1 & \text{otherwise.} \end{cases}$$

It is to be noted that each weight vector $\vec{w}$ defines a *perceptron function* or concept, hence updating a weight vector is equivalent to updating a concept.

Learning of a concept requires choosing values for the weight vector $\vec{w} = (w_1, \ldots, w_n)$. At the start of the algorithm, the initial weights are chosen as

---

[1]Instance and example are the same here, which are input to the function.

$\overrightarrow{w} = (0, \ldots, 0)$. On receiving an input instance $\overrightarrow{x} = (x_1, \ldots, x_n)$, the learning algorithm predicts the label of $\overrightarrow{x}$ to be $f(\overrightarrow{x})$. For the sake of compactness we represent $f(\overrightarrow{x})$ by $y$.

If it is found that the predicted label is correct, then there are no changes in the weight vector. But if the predicted label is wrong, the weight vector is updated using the perceptron learning rule given below. Here, $\lambda$ is the learning rate.

$$\overrightarrow{w} = \overrightarrow{w} + \lambda y . \overrightarrow{x} . \tag{13.6}$$

Multiple Agent Concept Learning

It is an extension of the single-agent concept learning. In the multi-agent system, every agent behaves both as a teacher and a learner, with a dynamic environment. This is due to the reason that the other agents being part of the learner environment also change their concepts as they learn from each other. Since there is no teacher or environment existing initially, there is no fixed concept to be learned. Accordingly, at the begin, the concept to be learned is dynamically formed due to the interaction among the agents.

### 13.6.4  Propositional and Relational Learning

The learning methods making use of "propositional calculus" are called *attribute-value/propositional* learners. They use objects with a fixed set of attributes, selected from a predefined set of values. The learning based on first-order relational descriptions—the *relational learning*—induces descriptions of relations and uses the objects described in terms of their components and relations among the components. As an example, if an object $X$ has attributes $a$, $b$, $c$ then this can be represented as a tree with $X$ as the root and $a$, $b$, $c$ as leaves, showing $X$ having relation with each leaf. This relation is the *propositional* relation. The background knowledge of relations is formed using the language of examples and concept descriptions, which is typically a first-order logic. Particularly, the learners that induce hypotheses in the form of logic programs, i.e., *Horn Clauses*, are called inductive logic programming systems.

The inductive learning has application in automatic construction of knowledge base in expert systems, which are an alternative to classic knowledge acquisition systems. The inductive techniques are also used for refinement of existing knowledge base, since they facilitate the detection of inconsistencies, redundancies, lack of knowledge, and also are helpful in the simplification of the rules provided by the domain expert. Due to their capability to detect patterns present in the input examples, the inductive methods are also used in the domains of biology, psychology, medicine, and genetics.

### 13.6.5   *Learning Through Decision Trees*

This is a combination of *induction-based* learning as well as *supervised* learning. In decision trees, every record is associated with a unique leaf node of the decision tree. The criteria for choosing a unique leaf node is to start from the root node, and repeatedly choose a child node based on the splitting criteria, which evaluates a condition on the input record at the current node.

The decision tree is a predictive model for learning, which represents the classification rules using a tree structure that recursively partitions the training data set into two subsets. Each internal node of the tree (all, except the leaf nodes) performs a *test* on the feature value, which results in two outcomes, one of those is selected, and ultimately reaches the terminal node where the prediction is performed.

An algorithm for decision-tree construction comprises two states: 1. construction of decision tree, and 2. pruning the tree. In the first step, the decision tree grows top-down in the greedy way. The data is examined by selecting the split condition at each node, starting with the root node. The data is then partitioned and procedure applied recursively. In state 2, the tree already constructed is pruned to control its size. Sophisticated pruning methods select the tree in such a way that it minimizes the prediction errors.

The decision trees tests one feature at each internal node, with one branch for each feature value, and the class predictions is carried out at the leaves.

Algorithm 13.1 is an algorithm for *decision-tree learning* (*relational learning*). It recursively calls the function $DT(T, E_{curr})$ representing knowledge that has been learned using decision trees. The variable $T$ represents the decision tree constructed so far, and $E_{curr}$ is the set of input examples. If all input examples in $E_{curr}$ are in the same class $C$ (i.e., representing the same concept), the decision tree will contain only one leaf corresponding to $C$. For example, the proposition $E_1 = $ "A smiling person is a friend," can be represented as a single leaf tree as shown in Fig. 13.3, with $C = friendly$. If the examples in the input belong to $n$ number $C_1, \ldots, C_n$, an attribute is chosen by the algorithm and based on this attribute, $E$ is divided into sets $E_1, \ldots, E_n$, which are disjoint. Each partition $E_i$ consists of the examples with the same value in the attribute selected. Each set $E_i$ is applied to the algorithm until the sets which have been obtained are left with elements belonging to the unique class. Finally, terminal nodes of the tree are the resultant disjoint classes $C_1, \ldots, C_n$ into which the examples are classified.

**Fig. 13.3**  A decision tree
with single proposition

$E_1 = is\text{-}smiling$

| yes

friendly
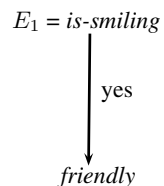
**Algorithm 13.1** Algorithm for Decision tree

1: Initialize: $E_{curr} = E$, $T = nil$
2: *function DT* $(T, E_{curr})$
3: **if** all examples in $E_{curr} \in Class\ C_s$ **then**
4:    generate a leaf labeled $C_s$
5: **else**
6:    ; {generate new node}
7:    select the most informative attribute $A$ with values $\{v_1, \ldots, v_n\}$
8:    Split $E_{curr}$ into subsets $E_1, \ldots, E_n$ according to values $v_1, \ldots, v_n$
9:    **for** $i = 1$ to $n$ **do**
10:       $DT(T_i, E_i)$
11:    **end for**
12: **end if**
13: Output: Decision-tree $T$ with the root $A$ and subtrees $T_1, \ldots, T_n$
14: End

**Fig. 13.4** A decision tree for root domain
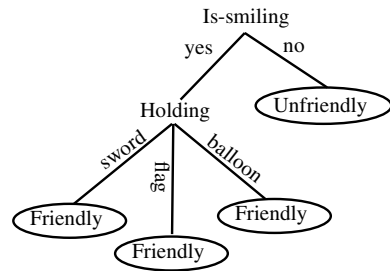


Any unseen example is classified as follows: start from the root node, attributes are tested at internal nodes, repeat the process and proceed to a terminal node. It is assumed that in an example that belong to a different class, at least one of the attributes has a different value.

In a decision tree, each internal node is labeled by an attribute and links. The attribute is selected from a set of possible values. In the example of Fig. 13.4, one of the attributes is "is smiling" and the set of its values is {*yes*, *no*}. Another attribute is "holding" with the set of values as {*sword*, *flag*, *balloon*}. The decision-tree construction has two important objectives: one is selection of the most informative attribute having the closest sense to the context in the example, and the second is to minimize the number of tests. Note that the number of tests is height (or depth) of the tree—paths length from the root to terminal node, and it directly defines the difficulty level of the problem [4].

**Example 13.1** Decision tree for financing a client.

Consider financing an individual by a financing company. The finance may be based on the good credit history or other factors.

Figure 13.5 shows the details, which causes learning. The learning helps in classifying the individual loan applications 1 . . . 16 into various categories shown in this figure.                                                                                                               □
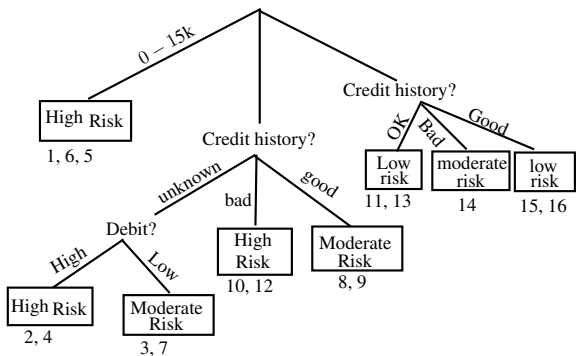
**Fig. 13.5** Learning through decision tree



**Fig. 13.6** Disambiguation-tree for *bank*
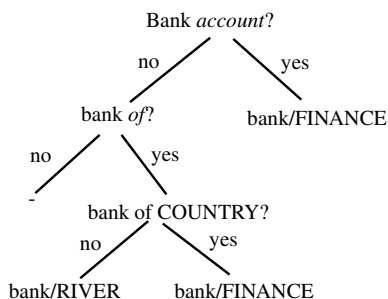


**Example 13.2** Decision tree for Word Sense Disambiguation (WSD).

Consider an example of a decision tree for Word Sense Disambiguation (WSD), i.e., to find out the correct sense of a given word in a phrase. We are interested in the word "bank" in a sentence: "we sat on the bank of Ganges." The process of disambiguation is demonstrated through a decision tree shown in Fig. 13.6.

The figure shows that the tree is traversed such that for every internal node it asks a question, having the answer. The answer to each branching is provided from the hidden knowledge in the sentence, as well as through commonsense knowledge. We note that it follows the path of *no-yes-no* from the root node until the terminal node, and the choice of sense *bank/RIVER* is made. In the tree, the node "bank of COUNTRY" stands for, for example, "bank of India", "Bank of America", etc. The terminal nodes for which no choice can be made based on specific feature values are indicated with empty values by "−".                                                                         □

Construction of the decision tree requires extensive access to the training database. Sometimes the training database is so huge that it does not fit the memory, in that case some efficient data access method is needed to achieve the scalability.

During the construction of a decision tree it requires construction of statistics in memory for each node, and is carried out in a single scan. The split at any node is based on some database partition that satisfies the split criteria.

The statistics for splits is updated in the memory at each node in a path starting from root node up to terminal node. The partitioning at each node should satisfy the splitting criteria corresponding to that node, and sufficient statistical data in the form of corpus is necessary for the construction of any learning decision tree.

## 13.7  Discovery-Based Learning

The discovery-based learning requires the maximum learning efforts among all types of learning. One of the earliest discovery-based programs is *Automated Mathematician* (AM), which derives a number of concepts in mathematics. It is based on the concepts of set theory. The AM discovers the natural numbers by modifying a concept called "bag", which is a generalization of the concept of a set in *set theory*. The concept "bag"permits multiple occurrences of set elements, e.g., $\{a, a, a, b, c, c, d, d\}$. Hence, the natural number 4 is represented as $\{1, 1, 1, 1\}$. An addition of the natural numbers is shown as the union of sets: $\{1, 1, 1, 1\} \cup \{1, 1\} = 6$. The operation of multiplication is the series of additions: $3 * 4 = \{1, 1, 1\} \cup \{1, 1, 1\} \cup \{1, 1, 1\} \cup \{1, 1, 1\} = 12$. The operation of subtraction is the difference of the sets: $4 - 3 = \{1, 1, 1, 1\} - \{1, 1, 1\} = \{1\}$. The division is obtained by repeated the set differences. A prime number is an integer having only two dividers, the unity and the number itself.

One approach for discovery-based learning is learning by concept hierarchies, as explained in the following.

Learning by Discovering the Concept Hierarchies

One of the most general approaches to solving a complex problem is to decompose the problem into smaller, less complex, and manageable subproblems. This principle is the foundation for *structured induction* in machine learning, where a concept hierarchy is defined and rules are learnt for each of the (sub)concepts. In this process, the concept hierarchy can be manually constructed or it can be automated, instead of learning a single complex classification rule from examples.

Consider that given five Boolean attributes $x_1, \ldots, x_5$, it is required to learn Boolean function $y$, expressed as

$$y = (x_4 \wedge x_5) \oplus (x_3 \wedge (x_1 \oplus x_2)). \qquad (13.7)$$

For five attributes there is space of 32 points. Consider that out of these, there are total 24 randomly selected points (i.e., 75% of the total) are to be selected as *examples* for learning. These are called as attribute-value vectors, and are useful for hiding the actual structure of the original expression. In Eq. 13.7 it is possible to justify that the best possible structure of sub-concepts will be as given in Fig. 13.7, with definition of functions as $f_1 = \oplus, f_2 = \wedge, f_3 = \wedge$, and $f_4 = \oplus$.

An approach for learning by concept hierarchies is based on function decomposition. Consider decomposing a function $y = f(X)$ into $y = g(A, h(B))$. The orig-

**Fig. 13.7** Hierarchy of concepts as a tree
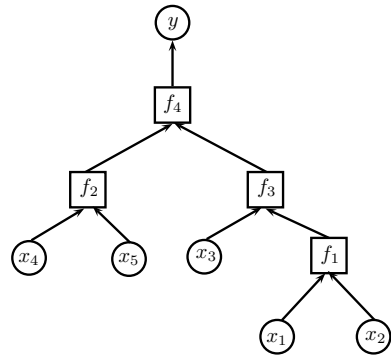


**Fig. 13.8** **a** Original decision tree, **b** Decision tree after decomposing



(a)                                          (b)
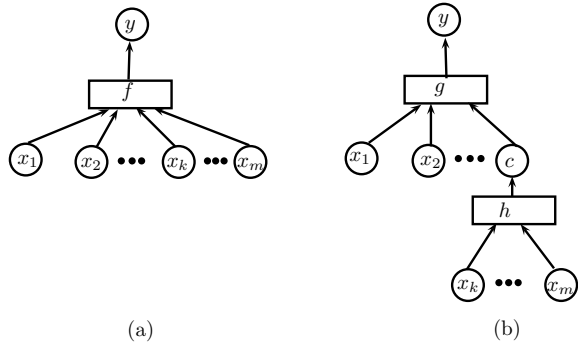
inal function is shown in Fig. 13.8a and b shows the decomposed function. In the original function, $X = \{x_1, x_2, ..., x_k, ..., x_m\}$, and after decomposition, $A = \{x_1, x_2\}$, $B = \{x_k, ..., x_m\}$ are sets of input attributes, and $y$ is variable, which represent the class. Also, input attributes $X = A \cup B$. The $f$, $g$, $h$ are functions, which are partially specified as examples, i.e., by sets of attribute-value vectors with assigned classes. The functions $g$ and $h$ are determined in the decomposition process, such that their joint complexity is lower than the complexity of $F$. The attribute $c$ is called intermediate concept and defined as $c = h(B)$, which is discovered in the decomposition process.

The function decomposition can be applied recursively on the functions $g$ and $h$, which should result in a hierarchy of concepts, as shown in Fig. 13.8b. For each concept in the hierarchy, like $h(B)$, there is a corresponding function ($h$ here), that decides the dependency of the concept, e.g., concept $c$, on its immediate descendants in the hierarchy [11].

## 13.8   Reinforcement Learning

Our learning is through feedback of our actions in the real world. Human beings always learn by interactions with the environment/teacher, which are *cause* and *effect* relations. The environment or the world around us is the teacher, but its lessons are often difficult to detect or grasp by the mind or analyze, hence hard to learn. The best example is learning by a dog or a young child, where good actions are rewarded and bad actions are discouraged. It is not defined as a more general example, but the actions with the world.

The programs that improve their performance at some task by rewards and punishments from the environment are examples of *Reinforcement Learning* (RL). For many automatic learning procedures for real-world tasks, like job-shop scheduling and elevator scheduling, RL has been found to be very successful. The total *discounted* reward the learner receives is optimized in the reinforcement learning, i.e., a reward that is received after one time step is considered equivalent to a fraction of the same reward received immediately before [1, 8].

There are four components of RL:

1. a policy,
2. a reward function,
3. a value mapping, and
4. a model of environment.

Learning agent's choices and methods of action are defined under a given policy. The policy is represented using a set of production rules. Sometimes the policy could also be probabilistic in nature. The reward function is a relationship between the state and goal, which maps each state into a reward measure, and indicates the need of that action to achieve the goal.

A reinforcement-based learning differs from supervised learning in the sense that a reinforcement system may not have a teacher to respond to each action, instead the learner itself creates a policy for interpreting the feedback. For example, playing in a particular style leads to winning a chess game for a player is reinforcement learning. The RL is a commonly used framework for problems requiring a number of agents, but their presence does not contribute to much of complexity as RL needs only limited feedback for learning. With the objective to maximize the expected reward, RL algorithms attempt to learn policies that help in taking optimal sequential decisions. However, this learning may be intractably slow as the methods scale up to more real-world problems with large state spaces.

A commonly used approach to solving RL problems makes use of a function, called *value function approximation*, using which an agent tries to learn a value for each state. This value is a long-term expected reward from entering a particular state, and such values are used for deriving a policy.

The state space for real-world problems contains infinitely a large number of possible states features. Hence, the designer of any such task must pick up only the most relevant features. Consider the task: Travel to Mumbai for work, and the

weather report for New Delhi is not likely to be relevant. From these we should construct a feature set. In most real-world situations, the feature set for a problem is broken down into a number of subsets, such that each subset can learn a specific concept of the domain. In this case, some concepts/feature-set subsets may be more important than the others.

**Example 13.3** Formulate a task of navigation to be carried out by an agent, with a goal to investigate the best plan to go from point $A$ to point $B$, and may choose a path and transport method of walking, driving, taxi, bus, train, or air.

The feature set of the agent's state space may include

– Positions of $A$ and $B$,
– Raining (yes/no),
– Type of shoes of agent,
– Agent is with umbrella (Yes/no),
– Current time, and
– Day of week.

Using positions as features, the agent can learn the concept of position and basic path planning. The features raining, shoes, and umbrella are useful in learning as to how the weather governs the policy, and the features of time, and day in a week may be useful in learning to handle traffic, for example. A conventional approach to solving this problem through RL is to learn in a 6D space (positions, raining, shoes, umbrella, time, weekday) when all the features are taken into account.            □

It is important to note that, the order in which features have been selected plays a significant role, in terms of difficulty level of the problem, and robustness of its solution.

### 13.8.1 Some Functions in Reinforcement Learning

In general, in a RL system, an agent recognizes itself in some state $p \in S$, then takes some action $a \in A$, and then recognizes itself in a new state $q$. The new state $q$ in which the agent enters from its previous state $p$ is decided by the agent's *transition function*:

$$T(S \times A) \rightarrow S. \tag{13.8}$$

In addition to the state change, the agent receives a reward $r$ for arriving to state $q$, based on the *reward function*:

$$R(S \times A) \rightarrow \mathbb{R}. \tag{13.9}$$

A function called *value function* $V^\pi(p)$ is based on the average sum-of-rewards received when an agent starts in state $p$, and enters state $q$ following a policy $\pi$. In this condition, we express the relation between the value function and *optimal policy* $V^*(p)$ as

$$\forall \pi, p : V^*(p) \geq V^\pi(p). \tag{13.10}$$

The RL is found successful in many domains, which includes many real-world domains, where it is used to optimize discounted total rewards. However, the most natural criteria in many other domains are based on optimizing the average reward per time step.

## 13.8.2 Supervised Versus Reinforcement Learning

The learning tasks are generally divided into two classes:

- *One-shot decisions* are tasks like classification and prediction, and
- *Sequential decision tasks* are like control, optimization, and planning.

The one-shot tasks are decision-based tasks that are usually formulated as *supervised learning* tasks. The learning algorithm in these tasks is provided with a set of input–output pairs, which act as relations between input and output. The input is the description of information used for making the decision, while the output is the description of the correct decision. For example, in handwriting recognition, the input is an image of the handwritten text of a digit/letter and the output is the identity of the digit/letter. Once such an algorithm is trained, it can recognize continuous handwritten text comprising sequence of letters and digit combinations, with good accuracy depending on how well the learning algorithm has been trained.

The sequential decision tasks are often formulated as *reinforcement learning* tasks. Learning algorithm in such tasks is part of an agent that interacts with the external environment. At each step, the agent observes the current state of the environment, then selects and executes some action, and on execution of this action the environment changes its state. Due to the state–action combination, the environment provides some feedback in the form of immediate reward to the agent, and the process goes on.

The reinforcement learning process maximizes the long-term rewards received by the agent. For example, when this learning is used in a setting in manufacturing, the current state is equal to the list of orders to be fulfilled along with the current status of manufacturing facility. The actions may change the settings of various production machines to maximize the overall profit. The latter can be achieved, for example by reducing energy consumption by machines as well as the idle times of machines. The immediate rewards include, saving in cost of each action, and revenue received when the order is completed.

## 13.9 Learning and Reasoning by Analogy

For analogical reasoning, an example in the form of a known solution is sufficient for learning a new solution. We use analogy when we learn about electric currents (e.g., Kirchhoff's current law, $I_3 = I_1 + I_2$ (see Fig. 13.9) by thinking about water pipes flow rates $Q_3 = Q_1 + Q_2$. While learning the subjects, like medicine, law, and economics, we also make use of analogy. Other examples are learning to play a card game of bridge from the knowledge of hearts, or programming a new algorithm using previously learned programming examples and concepts.

Some specific competence is necessary in using analogies to do certain kinds of learning and reasoning. The learning takes place when a constraint description is generated in one domain using analogy, having given a description with constraints in another domain. For example, we can better learn the Kirchhoff's current law using the knowledge about water flows in pipes, as illustrated in Fig. 13.9. Learning takes place when an analogy is used to answer a question about one situation, having given another situation that has preceded the current situation [7].

Analogy plays an important role in our reasoning process. We frequently explain or justify one phenomenon with another, where a previous experience often serves as framework or pattern for a new *analogous* experiences. A familiar experience is often in dealing with new experiences. Since so many of our acts are near repetitions of our previous acts, analogical-based learning is predominant in our living style. An implemented system for analogical learning has ingredients [10], discussed below.

*Representation of Extensible Relations*

The situations for analogical learning are represented based on relations between pairs of parts. In addition, more descriptions can be attached to the relations when needed.

*Importance-Dominated Matching*

Best possible match is found based on what is important in the situations, and then the similarity is measured between two situations. The importance is computed based on various kinds of constraints, and cause is a common importance-determining constraint.

**Fig. 13.9** Analogical problem-solving



From known flow rates:
$Q_3 = Q_1 + Q_2$

Infer Kirchoffs current Law:
$I_3 = ??$
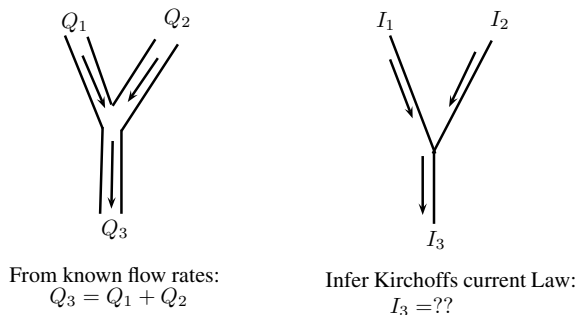
*Analogy-Driven Constraint Learning*

A constraint such as the one based on Kirchhoff's law is learned as a result of mapping parts of a situation belonging to a well-understood domain into the parts of another situation in a poorly understood domain.

*Analogy-Driven Reasoning*

Many times we are interested to find out if a particular relation holds. The *causes* found in a remembered situation can supply suggestive precedents.

Analogical reasoning shall best work subject to the following restrictions:

- *Symbolic sufficiency*. A situation to a previous instance can be represented using certain classes, properties, actions, etc. However, these collections should be simple, so that matching can be performed efficiently.
- *Description-based similarity*. A situation can be said to be similar to a previous one, if it can be mapped.
- *Constraint-based similarity*. Two situations are said to be similar if they are governed by the same constraint.

A program written for implementing reasoning by analogy should function based on identifiable principles, and must do the reasoning and learning by analogy, as in the following situations:

1. A class teacher explains to a student that a voltage drop across a resistance is calculated similar to the way we find the pressure drop across a section of a pipe having known the water flow rate and friction in it. With this analogy, a student can find the voltage drop across a resistance without the knowledge of Ohm's law.
2. A class teacher tells about two different resistances $r_1, r_2$, their voltage drops $v_1, v_1$, and currents $i_1, i_2$, respectively, and the student will find out the Ohm's law.
3. A class teacher suggests to generalize the principle using the flow of water in a pipe and the flow of current in a resistance, and the student concludes that there is a linear relation between the forces (potential/pressure) and flows (current/water flow rate).

From the above examples of analogical reasoning, we understand that a practice with some specific situations in one domain enables the invention of a law. Alternatively, once several examples of one law are known, the comparison results in the generalization of the law, like in the second example above. Analogies are likenesses or similarities between things that are otherwise different. The participants or things in analogies are unlimited: they may be concepts, objects, problems, solutions, plans, situations, episodes, and so on.

Analogies play a dominant role in human reasoning and learning processes— previously remembered experiences are transformed and extended to fit into new situations; old experiences may provide the explanations or scenarios that tend to

match with the new situations in some aspects and, therefore, may offer the promise of suggesting a solution to the new situation.

The difference between *inductive* learning and *analogical* learning is that, induction or similarity-based learning is based on the observation of a number of training examples, but analogical and explanation-based learning requires only one example in the form of a known solution or a past experience, as a sufficient criteria for learning to take place.

Analogical Reasoning

Consider the statement: "I have a Pomeranian dog, who is friendly with the children. Whenever I see another Pomeranian dog, I assume that he too will be friendly with the children." This type of reasoning we usually perform in our daily life. But it also appears that it does not guarantee the truth. The reason being that is the existence of one or few shared characteristics does not mean that all the remaining characteristic are identical.

For analogical reasoning to work, it is necessary that some relevant and known similarities between two objects or situations be give a sufficient base to a reason to believe that there are further similarities between them, and that should help to answer an open question. However, it is not always that way. At the most, analogical thinking can give the base to think of some probabilities, and these may be the base for probabilistic judgment. The kind of thinking that takes place in analogical reasoning has a simple structure:

1. Object $A$ possesses a set of characteristics, say $X$,
2. Object $B$ shares the characteristics $X$,
3. The object $A$ has characteristic $Y$ also,
4. Because the object $A$ and $B$ share characteristic $X$, based on this we conclude what is not known yet, that is, $B$ shares $Y$ as well, which is not necessarily correct.

In Fig. 13.10a, the analogical reasoning will work, while in (b) it may not.

**Definition 13.6** (*Reasoning by Analogy*) It is the process of inferring that a *conclusion property $Y$* holds of a particular object or situation $B$ (called target), given that $B$ shares a property or set of properties $X$ with another object/situation $A$ (called source), which has the set of properties $X$.

**Fig. 13.10** Similarities computing for analogical reasoning



(a)                                  (b)
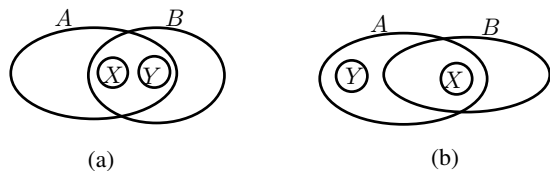
$$A(X) \wedge B(X)$$
$$A(Y)$$
$$- - - - -$$
$$\therefore B(Y). \tag{13.11}$$

The set of common properties $X$ is the similarity between $A$ and $B$, and the conclusion property $Y$ is projected from $A$ onto $B$. The process may be summarized by the following statements.

*Justification*. The analogical reasoning method defined in Eq. (13.11) can be justified by a two-step process: 1. from the first premise $A(X) \wedge B(X)$, conclude a *generalization* $\forall X[A(X) \rightarrow B(X)]$, and 2. instantiate the property $X$ and generalize it to $Y$, then apply the inference rule of modus ponens to obtain the conclusion $B(Y)$. Due to this process, only the first step is non-deductive. Hence, it appears as if the problem of justifying the analogy has been reduced to the problem of justifying a single-instance inductive generalization.                                                        □

A criteria for the evaluation of strength of an (enumerative) induction states that, as the number of instances confirming the generalizations increases, an inference increases in plausibility. If this is the only criteria used for analogical inference, then all the conclusions projected by any analogy without counterexamples should also be plausible. But, this fails. Consider the example, if the inspection of an Indian peacock reveals that its tail is more colorful than its legs, a projection of this conclusion onto an unseen peacock is plausible. But, projecting that a hanging feather on the peacock's tail will be observed on a second peacock is not plausible. Anyone who has closely looked at the tail of even a single peacock will have no counterexamples to these conclusions. Also, both conclusion properties can be projected, so the difference in agreeableness is accounted for some other criterion. Hence, the problem of analogy is different from (enumerative) induction because the analogy requires a stronger criterion for plausibility.

The analogical learning is a *nontrivial* form of inference, i.e., the conclusions based on the analogies are not necessarily logical entailments of the previous (source) knowledge. Analogical is a form of plausible reasoning for the current situation.

The steps in Eq. (13.11) require many other things, like the criteria for similarity measure, some form of indexing, efficient recall mechanism, a flexible mapping between base and object domain and, various levels of abstraction.

Following are the *steps* to perform the analogical reasoning:

1. *Analogue Recognition*. A new situation or problem is encountered and recognized as being similar to the previously known situation.
2. *Criteria for competence*. It is concerned with the knowledge required for any particular set of algorithms to exhibit the specified behavior.
3. *Representation of situation by extensible relations*. Extending the mapped experience, i.e., the newly mapped analogous are modified and extended to fit the target situation.

4. *Determining Analogy and Matching*. Here, two patterns are selected for their similarities and mapped from the base to the target domain.
5. *Similarity measures*. Various similarity measures are defined, and matchings are evaluated as per those.

   The *Access and Recall* mechanisms are useful features, as per which the similarity feature of a new problem serves as an index to the previously solved problem. This will help us in recalling the previous problem when required.
6. *Abstraction*. The matchings are abstracted away to generalize it.

The newly formulated solution is validated for its applicability through some trial process, like theorem provers or simulation. If the validation is supported, a generalized solution is formed, which accounts for old and new, both, which is equal to a new learning.

When a case is inferred analogous to another on the basis of a unifying principle such that it is accepted without having been tested against other possibilities, such inference due to the analogical reasoning will be faulty. In addition, when some similarities between two cases are considered decisive on the basis of insufficient investigation of relevant differences, such analogical reasoning will also go wrong.

## 13.10   A Framework of Symbol-Based Learning

Given the data, one way to characterize the learning problem is based on the goal/target of the learner algorithm, which may be a concept, or description of a class of objects. A learning algorithm may also acquire plans, heuristics for problem solution, or other form of procedural knowledge in any structure discussed so far, including the predicate logic.

The set of operations performed by the algorithm may include generalizations rule, heuristics rule, or a plan that satisfies the goal. The typical operations are generalizations or specializations of symbolic expressions, adjusting the weight of a neural network, etc. In concept learning, a learner may generalize a definition as follows:

$color(bird, black) \rightarrow is(bird, crow)$.
$\exists x \, color(X, black) \rightarrow is(X, crow)$.

There is a potential concept space, which is searched by the learner algorithm and to find the concept. The complexity of this concept space is the main complexity of the problem solution. The learning program/algorithm must be committed to order and the direction of the search, as well as it should make use of the available training data and heuristics, to carry out the search efficiently. The learner may make use of the heuristics, and, for example, learn the concept *ball*, by using the first example as a candidate concept:

$size(object_1, small) \land color(object_1, red) \land shape(object_1, round)$

and generalize using the second example:

$size(object_2, small) \land color(object_2, red) \land shape(object_2, round)$.

Here we have used *heuristics* and not *induction* for learning.

## 13.11   Explanation-Based Learning

One type of *deductive learning* is Explanation-based Learning (EBL). Following types of information are required to implement the explanation-based learning:

1. A formal statement of the *goal concept* to be learned;
2. *Domain theory*: It relates to the concept and the training example;
3. *A training example*: At least one positive training example of the concept is needed. It is an instance of the target, what the training program sees in the world;
4. *A Domain History*: It is a set of facts and rules used for explaining, how a training example is an instance of the good concept; and
5. *Operational criteria*: These are some means of describing the form, which a concept definition may assume.

The EBL makes use of an explicitly represented domain theory to construct an explanation of a training example, which is usually a proof that logically follows the knowledge base. An EBL begins with following prerequisites:

– A target Concept
– Learning by Analogy
– Learning by Instruction
– Learning by Induction
– Learning by Deduction
– Reinforcement Learning
– Discovery-based Learning, which determines the effective definition of this concept, called *local concept*, which requires a high-level description.

Consider using EBL to learn that a given specific object "this cup" is a cup. The target concept is a *rule* in predicate form, which is to be used to infer whether a given object is in fact a cup. We express this as

$$premise(X) \rightarrow cup(X). \tag{13.12}$$

In this implication, the premise is a *conjunctive expression* with variable $X$, and predicate expression $part(X, Y)$ indicates that $X's$ part is $Y$. Let the *domain knowledge* include

$canbelifted(X) \land holdsliquid(X) \rightarrow cup(X)$

$$light(X) \land part(X, handle) \rightarrow canbelifted(X)$$
$$part(X, Y) \land concave(X) \land pointsup(X) \rightarrow up(X)$$
$$small(X) \rightarrow light(X)$$

where predicate $holdsliquid(X)$ means that object $X$ holds liquid, $canbelifted(X)$ means $X$ can be lifted, and $pointsup(X)$ means object $X$ points upwards.

A training example should be an instance of the concept for the goal concept, as represented by the following facts of predicate logic.

$cup(thiscup)$.
$small(thiscup)$.
$put(thiscup, handle)$.
$own(bob, thiscup)$.
$part(thiscup, bottom)$.
$part(thiscup, bowl)$.
$color(thiscup, red)$.
$pointsup(bowl)$.
$concave(bowl)$.

The operational criteria should be such that the target concept can be defined in terms of observable structured properties of object, like it points up and its parts:bottom, handle, bowl, etc. The domain rules are provided in such a way that these enable the learner (algorithm) to infer whether a description is operational or not. Making use of this approach, a theorem prover can construct an explanation as to why the given example is in fact an instance of the training concept—a proof that confirms that the target concept logically follows from the example. This is shown in Fig. 13.11, as a generalized proof tree.

The EBL has the following advantages:

1. The training examples often contain many irrelevant information (or noise), such as "make of the cup". However, the domain theory allows the learner to select only the relevant aspects of training example.



**Fig. 13.11** Explanation-based learning to identify the object cup
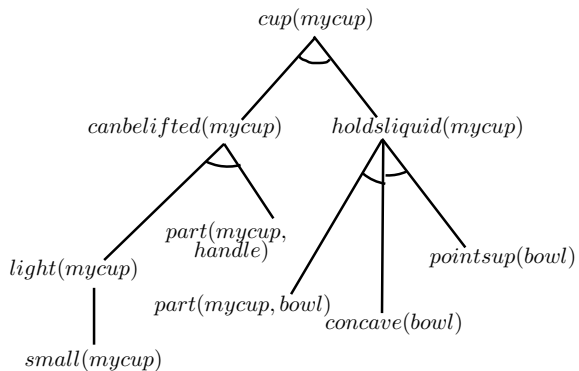
2. Usually, an example may allow for many possible generalizations, and most of them may not be useful. However, since the EBL inference logically follows the example, the generalization of EBL are logically consistent with the domain theory.

## 13.12   Machine Learning Applications

The domain of machine learning applications is continuously expanding. In the following we present some of the major application areas of machine learning:

Machine Perception

These approaches are used for learning to perform segmentation, feature extraction, and classification. All these provide higher performance and greater ease of implementation than traditional systems built from individual hand-built components.

Large-Scale Information Management and Data Analysis

In the present time, the data are being gathered in businesses and scientific applications far faster than ever before. This volume of data cannot be disseminated by any manual method. The machine learning methods are helpful in finding patterns and relationships in these data sets.

Information Filtering and Information Retrieval

Finding out useful information from an abundant size of collections is impossible using any manual methods. Models of the information needs of users are constructed based on machine-learning methods for various available information sources. Using these models it is possible to process the huge volume of information, both offline and online, to filter the undesired information and deliver to users only the information they need. However, for this the users should first express their need in natural language or symbolic form.

Control and Optimization

The *adaptive control methods* are more of futuristic systems, which are aimed to provide large-scale optimization, e.g., over the entire enterprises for planning, scheduling, manufacturing, inventory management, and logistics. The machine learning methods in controls and optimizations may also make it possible for new kinds of applications, such as smart buildings and smart vehicles.

Speech Recognition

All the presently available speech recognition systems in the market, make use of machine learning in some form, to train the system to recognize speech of unknown users as well of specific users. Before shipping, such systems (programs) are trained in speaker-independent mode, and after it is delivered to the individual users, it is

trained in speaker-dependent mode to accurately recognize his/her voice. Chapter 20 presents the speech-recognition techniques in more detail.

Computer Vision

The computer vision (or machine vision) applications range from face-recognition systems to sophisticated systems that automatically classify microscope images of cells. Many of the computer systems are more accurate than hand-crafted programs. Some of the important applications of computer vision are handwriting recognition, fingerprint recognition, and face recognition, which are available at a commercial scale.

Model Building

Many areas in science and engineering require construction of complex models, to support monitoring, diagnosis, repair, prediction, and extrapolation. Such models are a combination of programming and adaptation. Programming is used for the construction of the model, while adaptation is aimed to modify the model and calibrate it for data. The designers of such models need better tools for construction, calibration, and managing the models. These models have applications in medical science, strategic systems, space research, etc.

Machine Learning Techniques to Make Computers Easier to Use

Computers are still not easier to use due to ignorance of the user. While using a computer, each user has certain goals (about tasks, resources, etc.) and different preferences (that include, styles, habits, and abilities). Computer systems have very small features for these yet. Some examples are spell checker, command repetition, exception handling in high-level languages, in operating system, and in databases. However, these things are progressively appearing in plenty in smart phones, which include suggesting the next word while sending email or message, suggesting navigation path for drivers through map applications, storing the schedules and appointments and reminding through alarms, progressive learning of languages, etc. There is trend now of smart-watches which are claiming to provide solution for many things related to health and health monitoring.

## 13.13 Basic Research Problems in Machines Learning

There are wide-ranging machine-learning algorithms, from *general*—"off-the-shelf" methods for a variety of problems to *custom methods* suited for specific tasks. For example, general methods are learning through decision trees, rule sets, probabilistic networks, and feedforward neural networks, which have been applied to many different problems. On the other side, speech recognition algorithms based on hidden Markov models (HMMs), and methods for inferring using binding-site geometry for drug design are application-specific.

Many fundamental research problems in machine learning are concerned about the entire range from general to specific spectrum.

1. It is not known yet, as to what all the possible general methods are, as so far only the four techniques exist in the general category: algorithms for decision trees, rules, Bayes nets, and neural nets. It is not clear as to what other convenient representational formalism should be explored.
2. It is not known as to what extent these four general methods are optimal, or if there is scope for improvement in these algorithms.
3. What are truly the theoretical and engineering principles, that can be used as guidelines for the development of application-specific machine learning algorithms? The development of such algorithms is currently expensive, time-consuming, and mostly ad hoc in nature.
4. What are the efficient and convenient methods that can be used to acquire, represent, and incorporate the application-specific knowledge into general learning methods?

In addition, there are several other challenging questions of fundamental nature in the sequential decision-making tasks.

1. What are the situations in which it is preferable to represent and learn a *policy* directly, instead of learning a *value function* first? The difference between learning a policy versus value function is a policy specifies what action should be chosen in each state, but a value function provides accumulated long-term reward for each state. The best action can be selected based on the criteria: the one that will lead to the state with the highest value as per the value function. Some methods, like genetic algorithms, genetic programming, and parameterized policies, search directly in policy space. However, the other methods, like temporal difference learning and real-time dynamic programming, construct value functions.
2. What should be the properties of an ideal value function approximator?
3. What are the best ways to resolve the trade-off between *exploitation* (i.e., execution of the current policy) and *exploration* (i.e., search for better policies)?
4. What are the ways to solve large and partially observable Markov decision problems?

## 13.14   Summary

In every learning situation, a learner transforms the information received from an environment/teacher into some new form, and stores in that format for future use. The type of learning strategy used decides the type and magnitude of this transformation. Following are the basic learning strategies:

1. Rote Learning.
2. Learning by Instruction.

3. Learning by Deduction.
4. Learning by Analogy.
5. Learning by Induction.
6. Reinforcement Learning.
7. Discovery-based Learning.

These strategies are in increasing order of complexity of learning as well as the magnitude of transformation required.

A general learning model comprises: learner unit, knowledge base, performance evaluation, and the teacher, which may or may not include a physical teacher.

The learning techniques are fundamentally classified as *supervised learning* and *unsupervised learning*. The major difference between these is that the supervised learning requires the presence of a teacher, while the unsupervised technique does not require the teacher component.

*Inductive Learning* is a generalization carried out using a set of examples. It is one of the most commonly used learning techniques used by humans. Learning through *concepts* is a typical inductive learning process: given the examples of some concepts, such as "cat", we attempt to conclude a definition such that it will be helpful to correctly recognize future instances of the concept "cat". Note that, inductive learning makes use of specific facts/examples rather than general axioms as in the deductive processes.

While searching for possible hypotheses in the large space, the space can be constrained using the domain knowledge of the experts. *Learning from examples* is given the examples find a theory that is consistent with the examples. The explanation examples are called *argumented* examples, and these are in the form of arguments: *for* and *against*.

Based on the idea, whether it is a single concept or multiple concepts that are to be learned, we call it *single-concept* or *multiple-concept learning*. A *concept* is a function (a mapping), such that the learning task is supposed to discover this function, called *target function f* $^*$ : $\{0, 1\}^n \rightarrow \{0, 1\}$. The function maps each *instance* to the correct *label* (also called *class*). In the learning process, the system responds to learning instances and also to feedback received from the teacher.

The *attribute-value* learners or *propositional learners* are the methods that use a formalism of propositional calculus, where objects are defined with a fixed set of attributes. The learning methods based on first-order relational descriptions are called *relational learners*; they induce descriptions of relations, and make use of objects described in terms of their components and relations among components.

Automated Mathematician (AM) is one of the earliest *discovery-based learning* programs, deriving a number of concepts in mathematics. It is based on the concepts of set theory. AM discovers the natural numbers by modifying concept of "bag", which is a generalization of set. An approach to solving a problems, which is among the most general, is to decompose the problem into smaller, less complex, and better manageable subproblems. This principle is the foundation for learning by *concept hierarchies*. The rules for each concepts and sub-concepts are learned as a *discovery learning*.

*Reinforcement Learning* (RL) is an example of programs, which improve their performance for some task based on the criteria of rewards and punishments from the environment. Most approaches to reinforcement learning optimize the total *discounted* reward received by the learner.

For *analogical reasoning*, an example in the form of a known solution is sufficient for learning a new solution. It is the process of inferring that a *conclusion property Y* holds a particular object or situation *B* (called target): from the fact that *B* shares a property or set of properties *X* with another object/situation *A* (called source), which has the set of properties *X*. The set of common properties *X* is the similarity between *A* and *B*, and the conclusion property *Y* is projected from *A* onto *B*.

Goal of most learning algorithms is a concept, or a general description of a class of objects, represented in predicate logic. There is a potential concept space, which the learner searches to find the concept. A learner may generalize a definition as follows:

$color(bird, black) \rightarrow is(bird, crow)$
$\exists x \, color(X, black) \rightarrow is(X, crow).$

The above is *symbol-based* learning.

One type of *deductive learning* is Explanation-based Learning (EBL). The following information is required to implement it: a *goal concept*, *domain theory*, *a training example*, *a domain history*, and *operational criteria*.

Potential Applications of machine learning are machine perception, information management and large-scale data analysis, information filtering and retrieval, control and optimization, software engineering, speech recognition, computer vision, model building, etc.

## Exercises

1. At the end of the day, try to recollect any five important events you have witnessed this day. Try to associate the type of learning you have used in these events.
2. Analyze and propose the type of structures you consider are appropriate in the following learning processes:

   a. Rote learning
   b. Inductive learning
   c. Supervised learning
   d. Unsupervised learning
   e. Learning by example
   f. Analogical learning

   Answer the above questions, in reference to some programming language, e.g., *C*. And, describe the estimated algorithmic steps. For example, for rote learning

you may take the example of the number tables, and indexing you need, so that one can speak the table of say "5", and also one can answer $5 \times 6 = 30$.

3. What is the difference between the *learning by induction* versus *learning by examples*.

4. Explain the major difference between analogical learning and inductive learning in respect of the approach used for learning.

5. Suggest a learning method for each of the following, explaining why the suggested method is appropriate, and provide logical steps to learn using it.

    a. To learn how to drive a car after having observed a trained driver, while you are riding along with the same.

    b. To learn how to drive a car after having known the driving of a bullock-cart.

    c. To learn how to drive a car after having learned the driving of a tractor.

    d. To learn how to fly an aircraft after having very closely observed the birds flying, like eagle, crane, and hawk.

    e. Learning to keep your wallet protected after having lost it due to theft.

6. The helicopter takes off straight, instead of having a run before taking off. It has similarity with peacock in the birds. Explain a mathematical model, which supports the learning by analogy of a copter with a peacock.

# References

1. Ahmadi M et al (2007) IFSA: incremental feature-set augmentation for reinforcement learning tasks. In: The sixth international joint conference on autonomous agents and multi-agent systems, pp 1128–1135
2. Bengio Y (2016) Machines who learn. Sci Am 6:38–43
3. Domingos P (2012) A few useful things to know about machine learning. Commun ACM 55(10):78–87
4. Mantaras RL, Armengol E (1998) Machine learning from examples: inductive and Lazy methods. Data Knowl Eng 25(1998):99–123
5. Mozina M et al (2007) Argument based machine learning. Artif Intell 171:922–937
6. Schmid U, Kitzelmann E (2011) Inductive rule learning on the knowledge level. Cogn Syst Res 12:237–248
7. Sunstein CR (1993) On analogical reasoning. Harv Law Rev 106:741–791
8. Tadepalli P, OK D (1998) Model-based average reward reinforcement learning. Artif Intell 100:177–224
9. Wang J, Gasser L (2002) Mutual online concept learning for multiple agents. In: AAMAS'02, July 15–19, Bologna, Italy
10. Winston PH (1980) Learning and reasoning by analogy. Commun ACM 23(12):689–703
11. Zupan B et al (1999) Learning by discovering concept hierarchies. Artif Intell 109:211–242

# Chapter 14
# Statistical Learning Theory

**Abstract**  A machine learning system, in general, learns from the environment, but statistical machine learning programs (systems) learn from the data. This chapter presents techniques for statistical machine learning using Support Vector Machines (SVM) to recognize the patterns and classify them, predicting structured objects using SVM, k-nearest neighbor method for classification, and Naive Bayes classifiers. The artificial neural networks are presented with brief introduction to error-correction rules, Boltzmann learning, Hebbian rule, competitive learning rule, and deep learning. The instance-based learning is treated in details with its algorithm and learning task. The chapter concludes with a summary, and a set of practice exercises.

**Keywords**  Statistical machine learning · Support Vector Machines (SVM) · K-nearest method · Naive Bayes classifier · Artificial Neural Networks (ANN) · Boltzmann learning · Hebbian rule · Deep learning · Instance-Based Learning (IBL)

## 14.1  Introduction

Statistical machine learning systems help the programs automatically learn from the data. This is an attractive option as an alternative to manually coding every rule in a program. Machine learning is used in computer science and beyond, e.g., in spam filters in email, web searching, placement advertisement, stock trading, credit scoring, drug design, fraud detection, and in many more applications. The statistical machine learning can be said to be a kind of a data mining.

The field of machine learning developed in a new direction during 1979–80, with innovations like *decision trees* and *rule learning*. These methods were applied in expert systems. In the late 1980s, there were renewals of research interests in machine learning architectures that used *perceptron* as the basic building block. This was particularly because the limitations which were highlighted by *Minksy* and *Papert* in the early days of AI were overcome by *multilayer networks* that used simple computing elements. The latter used perceptrons like nodes called *neural networks*. These networks were trained using biologically inspired backpropagation

algorithms, which performed gradient descent on error-based cost functions. This new approach raised the hopes not only of creating machines that were capable to learn, but also of understanding the basic mechanisms used by biological learning systems.

The machine learning algorithms can determine how to perform important tasks through generalization from examples. A machine learning-based solution, at the end, turns out to be a feasible and cost-effective solution when compared with the manual programming approach. In statistical-based machine learning approaches, as more data becomes available, it becomes possible to tackle more ambitious problems.

In the previous chapter we concentrated on broad classification of machine learning techniques, and basic principles of each of them. In the present chapter we will discuss new and emerging techniques, which are mainly statistical based. For the purpose of illustration, we focus on the most mature and widely used classification for these techniques.

**Learning Outcomes of This Chapter**:

1. Apply the simple statistical learning algorithm such as Naive Bayesian Classifier to a classification task and measure the classifier's accuracy. [Usage]
2. Apply the simple statistical learning algorithm such as SVM to a classification task and measure the classifier's accuracy. [Usage]
3. Evaluate the performance of a simple learning system on a real-world data set. [Assessment]
4. Compare and contrast each of the following techniques, providing examples of, in what condition each strategy is superior: decision trees, neural networks, and belief networks, deep learning. [Assessment]

## 14.2   Classification

One of the sub-fields of predictive modeling is supervised pattern classification. It is a task of training a *model* based on labeled training data, such that the model can later be used to assign predefined class labels to new objects. Figure 14.1 shows this process.

**Definition 14.1** (*Classifier*) We refer to a *classifier* as a system that typically has inputs vector of discrete and/or continuous feature values and outputs a single discrete value, the *class*.                                                                                      □

In that sense, for email filtering we use a classifier that classifies the email messages into "spam" and "no spam" classes. The input to this classifier may be a Boolean vector,

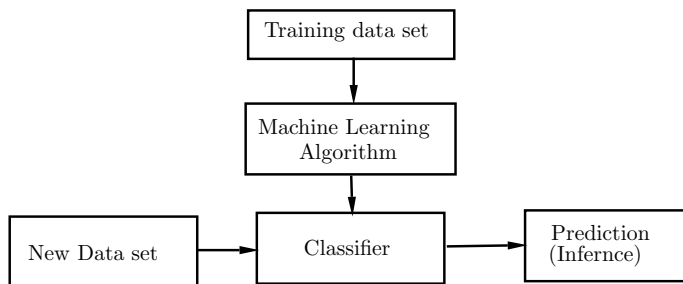$$\mathbf{x} = (x_1, \ldots, x_j, \ldots, x_d), \tag{14.1}$$

**Fig. 14.1** A classifier's block diagram

where $x_1, \ldots, x_d$ corresponds to words from some standard dictionary, such that the element $x_j = 1$ (True) if the $j$th word of that dictionary is present in the email under consideration. If the $j$th word from the dictionary is absent in email, then $x_j = 0$. The aim is naturally to classify the emails based on these words. A learner (program) inputs a training set of examples, comprising $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,d})$, as an observed input, $y_i$ as the supposed to be corresponding output, and also the classifier, which is supposed to be the class of this email. The test, that the learner program (i.e., classifier) has successfully learned is, whether this classifier produces the correct output $y_t$ for future examples $\mathbf{x}_t$. That is, whether the spam filter has correctly classified the first-time seen email messages as spam or not spam, or some other classes as dictated by the classifier [6].

Though there are thousands of learning algorithms today, the learning can be expressed as comprising three parts: representation, evaluation, and optimization.

*Representation*

The classifier must be represented in some formal language. Choosing a representation amounts to choosing a set of classifiers with the capability to learn. The representing set is called the *hypothesis space* of the learner, i.e., it covers those features of the inputs to which we give importance. The classifier must be in the hypothesis space, otherwise it cannot learn.

*Evaluation*

The evaluation function, called *object function* or *scoring function* of a learning algorithm, should be able to distinguish between good classifiers from bad classifiers.

*Optimization*

This function is useful for searching a method for a classifier that helps in scoring the highest. Proper choice of optimization technique is key to the efficiency of the learner.

The learners can be divided into two major types: 1. whose representation comprises fixed size, e.g., *linear classifiers*, 2. those whose representation can grow with data, e.g., *decision trees*.

## 14.3   Support Vector Machines

Due to Support Vector Machines (SVMs), a new approach is introduced to machine learning; the method is based more on statistical foundations and less on biological plausibility. SVMs are trained by minimizing a convex cost function, which is a measure of the margin of correct classification. This margin, in the case of the perceptrons, is used for measuring the mistake complexity. In SVMs, the optimization of this margin is justified by a rigorous statistical analysis. The SVMs use binary classification, i.e., input is always split into two classes. This approach is proved useful in a wide range of applications, which vary from text classifications, e.g., "Is this article related to my search query?" to bioinformatics, e.g., "Do these micro-array profiles indicate cancerous cells?"

The SVMs were originally used for problems in binary classification, where it was required to distinguish an object belonging to one of the two categories. For a long time, people attempted to solve such problems by simply reducing them to binary classification problems. However, these reductions failed to exploit about the structure of the predicted objects. But SVMs are capable of exploiting the structure also, e.g., the conventional SVMs can be applied for parts-of-speech tagging applications, which requires resolving the sense of each word in a sentence using the context of the word, i.e., surrounding text.

The support vector machines are falling in the category of *supervised learning models* based on associative learning algorithms that analyze the data and recognize patterns, hence they have applications in classification and regression analysis. After having trained by some training examples, each belonging to one of the two categories, a SVM training algorithm builds a model that assigns new examples into one of the two categories. Thus, an SVM is a *non-probabilistic linear* classifier. They are powerful approaches to predictive modeling with success in a number of applications, which include handwritten digit and alphabet recognition, face detection, text categorization, etc.

An SVM is a model where examples are nothing but representation of points in space, that are mapped into two classes, such that the examples of the separate categories are divided by a clear gap that should be as wide as possible. Once the learning has taken place, any amount of new examples are then mapped into that same space and predicted to belong to a category depending on which side of the gap they fall on.

The SVMs fit in the context of classification where the attribute of the objects whose value is to be predicted, called *dependent attribute*, has two possible values: 0 and 1. The classification is performed on a *surface* (in three or more dimensions) in the space of predictor attributes, that separate the points with dependent attribute $= 0$, from those with dependent attribute $= 1$ (or it may be $-1$ and $+1$ respectively). An optimal separating surface is computed by maximizing the margin of separation as shown in Fig. 14.2, where, data point with 0 attribute are labeled by circle ($\circ$s) and those with attribute 1 are labeled as squares ($\square$s). The figure shows a separable problem in a 2D space. The margin of separation is the distance between the boundary
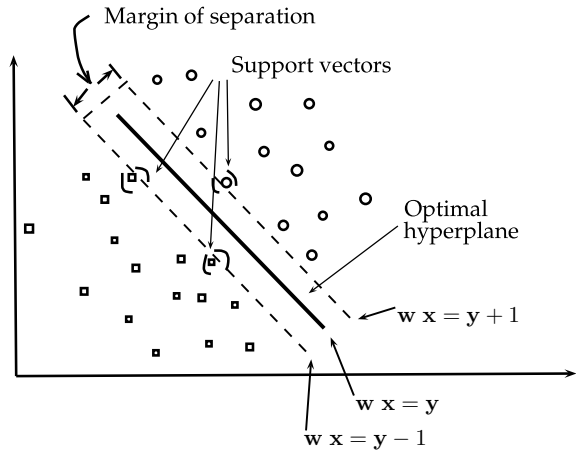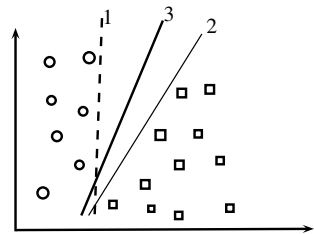
**Fig. 14.2**  Classification
through SVM



**Fig. 14.3**  A case of different
margin in SVM



of points with dependent attribute $= 0$ and the boundary of those with dependent
attribute $= 1$. The variables **x**, **y** and **w**$'$ are vectors.

The margin (which is a hyperplane) is the measure of "safety" in separating the
two sets of points, the larger is better. Figure 14.3 shows three separator lines between
the set of data values of two categories, with separator line 3 producing the maximum
margin. Computing the optimal separating surface, in a standard SVM formulation
requires solving an optimization problem, which is quadratic in nature [2, 3].

### 14.3.1   *Learning Pattern Recognition from Examples*

The SVMs are based on *Analogical Learning*, which we discussed in the previous
chapter. The Support Vector Machine (or Support Vector Network (SVN)) maps the
input vectors into some high-dimensional feature space that we call **Z**, through some
nonlinear mapping, which is chosen a priori. A linear decision surface is constructed
in this space, with special properties that ensure high generalization capability of the
network. This approach gives rise to two problems: the first is conceptual in nature,
while the other is technical. The conceptual problem is regarding how to find out a

separating hyperplane that will generalize well in the presence of very high dimension of the feature space. The technical problem is about how to computationally treat such a high dimensionality space problem?

The conceptual part of this problem can be solved for the case of *optimal hyperplanes* for separable classes. An optimal hyperplane is a linear decision function with maximal margin between the vectors of two classes (see Fig. 14.2). To construct such a hyperplane, only a small amount of data is required, called *support vectors*, which determine the margin. The optimal hyperplanes should separate the training data without errors. The optimal hyperplane algorithm is described as follows.

The set of labeled training patterns,

$$(y_1, \mathbf{x_1}), \ldots, (y_\ell, \mathbf{x_\ell}), \quad y_i \in \{-1, 1\}, \tag{14.2}$$

are linearly separable if there exists a vector $\mathbf{w}$ and a scalar $b$ (for bias) such that the following inequalities are valid for all elements of the training set given in Eq. (14.2).

$$\begin{aligned}
\mathbf{w}.\mathbf{x}_i + b &\geq 1 \quad \textit{if} \quad y_i = 1, \\
\mathbf{w}.\mathbf{x}_i + b &\leq 1 \quad \textit{if} \quad y_i = -1.
\end{aligned}$$
$$\tag{14.3}$$

The inequalities of Eq. (14.3) can be rewritten as

$$y_i(\mathbf{w}.\mathbf{x_i} + b) \geq 1, \quad i = 1, \ldots, \ell. \tag{14.4}$$

The optimal hyperplane, which is the unique one separating the training data with a maximal margin, can be expressed by

$$\mathbf{w_0}.\mathbf{x} + b_0 = 0. \tag{14.5}$$

As an application of character recognition, Fig. 14.4 shows a classification of an unknown pattern (which appears like the decimal digit 7) using a support vector machine. A pattern in the input space is compared with support vectors $\mathbf{x}_k$, resulting in values $u_1, u_i, u_j, u_s$. The resulting values are nonlinearly transformed using *Lagrangian* multipliers. A linear function $(F)$ of these transformed values determines the output of the classifier [5].

For the hyperplane expression (14.4), we use a standard optimization technique through which we construct the Lagrangian function,

$$L(\mathbf{w}, b, \Lambda) = \frac{1}{2}\mathbf{w}.\mathbf{w} - \sum_{i=1}^{\ell} \alpha_i[y_i(\mathbf{x}_i.\mathbf{w} + b) - 1], \tag{14.6}$$

where $\Lambda^T = (\alpha_1, \ldots, \alpha_\ell)$ is the vector of nonnegative Lagrange multipliers corresponding to the constraints.

**Fig. 14.4** Classification of
an unknown pattern
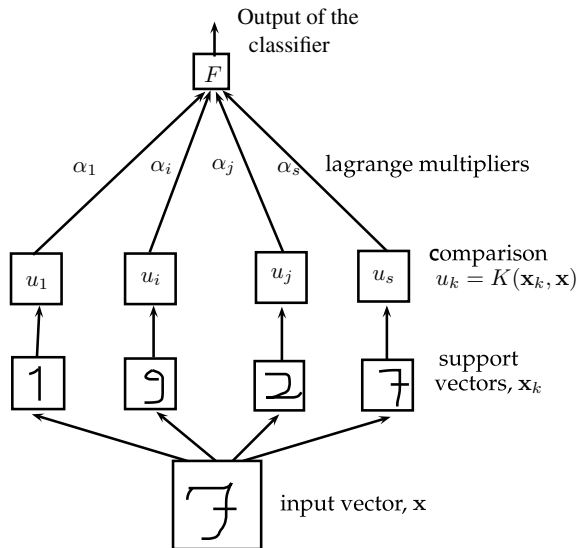


## 14.3.2  *Maximum Margin Training Algorithm*

As discussed above, in SVMs it is attempted to find out the support vectors with maximum margin. An algorithm for this purpose finds a decision function for $n$-dimension pattern vectors $\mathbf{x}$, which belong to either of the two classes, $A$ or $B$. Input to the training algorithm is a set of $k$ examples $\mathbf{x}_1, ..., \mathbf{x}_k$ with labels $y_1, ..., y_k$, respectively.

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_k, y_k) \tag{14.7}$$

$$\text{where} \quad \begin{cases} y_i = +1 & \text{if } \mathbf{x}_i \in \text{class } A \\ y_i = -1 & \text{if } \mathbf{x}_i \in \text{class } B. \end{cases} \tag{14.8}$$

Making use of these training examples during the learning phase, the SVM algorithm finds the decision functions $D(\mathbf{x})$, where $\mathbf{x}$ is an unknown pattern. Once the training phase is over, the class of unknown pattern is predicted using the following rule:

$$\mathbf{x} \in A, \ \text{if } D(\mathbf{x}) > 0$$
$$\mathbf{x} \in B, \ \text{otherwise.}$$

It is mandatory that the decision functions $D(\mathbf{x})$ are linear in their parameters, however they are not restricted to linear dependencies of $\mathbf{x}$. These functions can be expressed either in the direct space, or in dual space. The notation used for direct space is identical to the one used in perceptions, i.e.,

$$D(\mathbf{x}) = \sum_{i=1}^{k} w\varphi(\mathbf{x}_i) + b. \tag{14.9}$$

In the above equation, $\varphi$ is a predefined functions of $\mathbf{x}$, and $w$ and $b$ are the adjustable parameters of the decision function. In the dual space, the decision functions are of the form,

$$D(\mathbf{x}) = \sum_{i=1}^{k} \alpha_i \, K(\mathbf{x}_i, \mathbf{x}) + b, \tag{14.10}$$

where coefficients $\alpha_i$ are the parameters to be adjusted, and $\mathbf{x}_i$ are the training patterns. The function $K$ is a predefined kernel, for example, radial bias function or a potential function.

## 14.4   Predicting Structured Objects Using SVM

A potential drawback of SVMs and other statistical learning methods is that they treat the category structure as "flat"and do not consider the relationships between categories, which are commonly used for expressing the concepts as hierarchies or as taxonomies. The taxonomies or taxonomical structures offer clear advantages in supporting tasks like browsing, searching, or visualization. It is to realize that all the real-word concepts have complex hierarchical structures. For example, in browsing, the list of answers to a query can be taken as the children of the root (keyword). Further, each answer link would correspond to a web page, which may have many links, thus making a tree structure.

To better appreciate the above, consider the problem of natural language parsing as shown in Fig. 14.5, where the parser receives the input in the form of a sentence of natural language, and the output is a parser-tree as a decomposition of the sentence into its constituents. While support vector machines are used in NLP applications, such as Word Sense Disambiguation (WSD), parsing is not suited as a problem to be solved through SVMs for classification. This is because, in parsing, the output is not a classification of yes/no but a labeled tree, representing the structure of a sentence. So the question is, how can we take an SVM and learn predicting a parse-tree?

The question above arises not only for predicting the structured trees but also for a variety of other structures, like DNA sequence, or sequence ordering for image segments, etc.

Let us assume that there is multi-class SVM for document classification, where each document belongs to exactly one category. Let $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ be a set of $n$ labeled training documents. Here, $\mathbf{x}_i \in \Re^d$ denotes a standard vector representation for the *i-th* training document, and there are total $d$ number of such documents. Each label is $y_i$, where $y_i \in \mathscr{Y} \equiv \{1, \ldots, k\}$, and $k$ is the total number of categories.
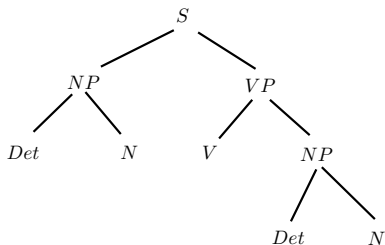
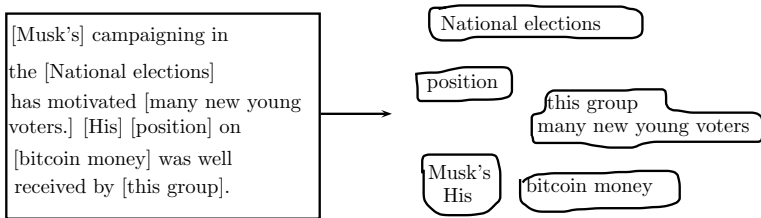**Fig. 14.5**  Parse-tree for "The dog chased the thief"



**Fig. 14.6**  Resolving the equivalent noun-phrase co-references

Let us assume that there is weight vector $\mathbf{w}_y$ for every class $1 \leq y \leq k$. We will refer to the stacked vector of all weights by $\mathbf{w} = (\mathbf{w}_1, \ldots, \mathbf{w}_k)$. The *structured output prediction* is the term used for this kind of prediction, where our objective is to learn a function $h : \mathcal{X} \to \mathcal{Y}$, that maps the inputs $\mathbf{x} \in \mathcal{X}$ to complex and structured output $y \in \mathcal{Y}$. In the structured output prediction, tasks range from predicting an equivalence relation, say, in noun-phrase co-reference resolution, to predicting a correct and well-formed machine translation.

**Example 14.1**  Resolving the equivalent noun-phrase co-references.

Consider a situation that a man called "Musk" contests in the elections and hence campaigns to seeks votes as a presidential candidate, and suggests election manifesto of promoting the bitcoin money if he wins the elections. Figure 14.6 shows an example of resolving noun-phrase *co-reference*, where there is an equivalence relation between a noun phrase (e.g., "Musk") and its co-reference "His", and similarly between "this group"and "many young voters".

The problems of this type exists elsewhere also, for example, in image segmentation, for determining an equivalence relation, say $y$, over a matrix of pixels $\mathbf{x}$, and the problem of web search to predict a document ranking $y$ for a given query $\mathbf{x}$. The *structural SVMs* (SSVMs) are suitable for use, as well as to address these all, and a large range of prediction tasks with structured output.                                      $\square$

## 14.5   Working of Structural SVMs

How to map the input to some structured output? Basically, a task of prediction of a structure is similar to a task of *multi-class learning*. Each possible parse-tree $y \in \mathcal{Y}$ may correspond to one class, and classifying a new example $\mathbf{x}$ is nothing but predicting its correct *class* out of many possible classes, as shown in Fig. 14.7.

However, the problem is that there is a very large number of classes $\mathcal{Y}$. In case of parsing, the number of possible parse-trees are exponential as a function of the length of the sentence. This situation is similar for most other prediction problems, which output some structure. Hence, there is need for finding a *compact* representation of output spaces, which are large in size. In addition, a *single prediction* for an example is computationally challenging, this requires the enumeration of all the outputs. The third challenge is, how to distinguish between two wrong parse-trees, where one may be closer to the correct one. This, we call as *prediction error* [9, 11].

We can derive a structural SVM from a multi-class SVM, such that for each class $y$, the multi-class SVMs use a weight vector $\mathbf{w}_y$, and each input $\mathbf{x}$ has a score for each class $y$ via the function,

$$f(\mathbf{x}, y) \equiv \mathbf{w}_y . \Phi(\mathbf{x}), \tag{14.11}$$

where $\Phi$ is a function that extracts the vector $\Phi(\mathbf{x})$ of binary or numeric features, from $\mathbf{x}$. Hence, every feature has additive-weighted influence in the modeled compatibility between inputs $\mathbf{x}$ and classes $y$. For the classification of $\mathbf{x}$, a prediction rule $h(\mathbf{x})$ chooses simply the highest-scoring class as the predicted output. This is expressed by

$$h(\mathbf{x}) \equiv argmax_{y \in \mathcal{Y}} f(\mathbf{x}, y). \tag{14.12}$$



**Fig. 14.7** Resolving to correct structure through *structural SVM*
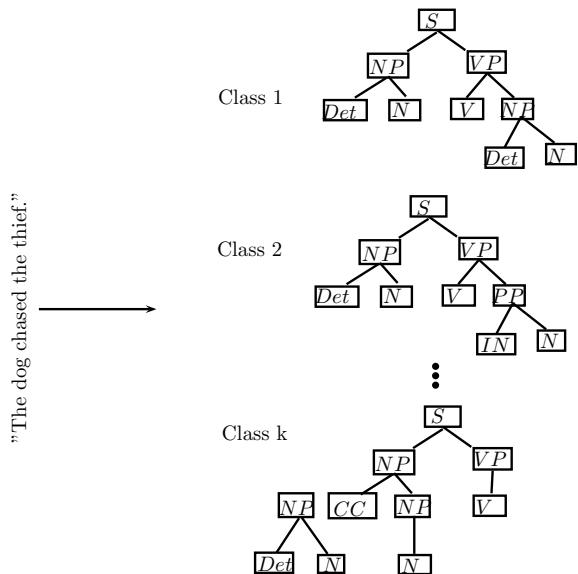
This will result in a correct prediction of output $y$ for input $\mathbf{x}$, provided that the weights $\mathbf{w} = (\mathbf{w}_1, \ldots, \mathbf{w}_k)$ are chosen such that the inequalities $f(\mathbf{x}, \bar{y}) < f(\mathbf{x}, y)$ hold true for all incorrect outputs $\bar{y} \neq y$. For a given training sample $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, this leads directly to a margin (a hard-margin) formulation of the learning problem by requiring a fixed margin ($= 1$) separation of all training examples, while using the norm of $\mathbf{w}$ as a regularizer:

$$min_w \ \frac{1}{2}\|\mathbf{w}\|^2, \ \text{ such that } f(\mathbf{x}_i, y_i) - f(\mathbf{x}_i, \bar{y}) \geq 1, \ (\forall i, \bar{y} \neq y_i). \qquad (14.13)$$

For a $k$-class problem, the optimization problem has a total $n(k-1)$ inequalities, that are linear in $\mathbf{w}$. This is because, one can expand

$$f(\mathbf{x}_i, y_i) - f(\mathbf{x}_i, \bar{y}) = (\mathbf{w}_{yi} - \mathbf{w}_{\bar{y}}).\Phi(\mathbf{x}_i), \qquad (14.14)$$

which is a convex quadratic program.

The drawback of Eq. (14.13) for structured output is there is a generalization across the inputs $\mathbf{x}$, but output is without generalization due to a separate weight vector $\mathbf{w}_y$ for each class $y$. It is therefore not advisable to reduce the output prediction to multi-class classification, as the number of possible outputs are likely to become very large. This problem is solved using a new function $\Psi(\mathbf{x}, y)$ in place of $\Phi(\mathbf{x}, y)$ to extract features from input–output pairs. This new function is called as *joint feature map*. This will yield compatibility functions due to contributions from the combined properties of both inputs and outputs. Since the compatibility functions is defined via $f(\mathbf{x}, y) \equiv \mathbf{w}.\Psi(\mathbf{x}, y)$, the number of parameters simply will be equal to the number of features extracted via $\Psi$, and that may not depend on the number of classes $|\mathscr{Y}|$.

## 14.6  k-Nearest Neighbor Method

The nearest neighbor method is a *statistical* learning method, also called sometimes as *memory-based method*. The method is used for clustering of objects based on some similarity in their attributes. The $k$-nearest neighbor ($k$-NN) or simply nearest neighbor method is an *analogical* type of learning. Given a training set $\theta$ of $N$ number of labeled patterns, each with $n$ attributes, a nearest neighbor algorithm decides that some new pattern $\mathbf{x}$ belongs to the same category to which its closest neighbors in $\theta$ belong. In other words, a $k$-nearest neighbor algorithm assigns a new pattern, $\mathbf{x}$, to that category $\mathbf{x_i}$ to which the majority of its $k$-closest neighbors belong.

**Example 14.2**  Nearest Neighbor.

Consider there are five cars manufactured by company $A$, and for every car ten important attributes have been identified, like engine size, color, wheel size, fuel used, etc. Then, training set size is $N = 5$, and attribute size of each training set is $n = 10$. Let a new model is introduced by company $B$, and the nearest neighbors'

size is taken as $k = 4$. In this situation, a car from the company $A$ having four attributes common to the new car is, say model-Zeta. Therefore, Zeta being the nearest neighbor to the new model from company $B$, the new model is put in the class of model-Zeta.                                                                                        □

If $k$ is relatively large, there are less chances of the decision going wrong due to noisy training pattern close to **x**. However, the large values of $k$ also reduce the sharpness or acuteness of this method. The $k$-NN method can be thought of as estimating the values of the probabilities of belongingness to class, given an input pattern **x**.

The $k$-NN approach has been shown to be a useful non-parameteric technique for *regression* and *classification*. In both cases, the input comprises the $k$-closest neighbors of the vector. However, finding the $k$-NNs for a test sample, among $N$ design samples, is a time-consuming process, particularly for large $N$. The reordering of these samples requires $N(N - 1)/2$ pairwise distance computations, which is a time-consuming process for large values of $N$.

When used for object classification, the $k$-NN algorithm has many practical applications, e.g., in the areas of artificial intelligence, pattern recognition, statistics, cognitive psychology, vision analysis, and medicine, to name a few. The decision rule in $k$-NN provides a simple non-parameteric procedure for the assignment of a class label to the input pattern based on the class labels represented by the $k$-closest (for example, in terms of Euclidean distance) neighbors of the vector.

### 14.6.1  k-NN Search Algorithm

Let us assume that we are given $N$ design samples, $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, each of which is $n$-dimensional sample. With this, it is required to compute the $k$-nearest neighbors ($k < n$) in a test sample **y**, as measured by an appropriate distance function $d(\mathbf{y}, \mathbf{x}_i)$ [7].

There is no need of preprocessing of the labeled sample set in the nearest neighbor classifier. The nearest neighbor classification rule assigns an input sample vector **y** of unknown classification, to a class which is THE nearest neighbor to it (see Algorithm 14.1). The idea of the nearest neighbor can be extended to $k$-nearest neighbors with the vectors **y** being assigned to the class that is represented by a majority among the $k$-nearest neighbors. In the algorithm, $d_k$ stands for distance from **y** to $\mathbf{x}_k$, for $k$-nearest neighbor.

When more than one neighbor is taken into account, it is likely that there may be a tie among the qualifying classes, which corresponds to maximum number of neighbors in the group of $k$-nearest neighbors. There is a simple way to handle this problem by restricting the possible values of $k$. For example, if there is two-class problem, and $k$ is restricted to odd values only, no tie can occur.

Occurrence of a tie can be handled using the following approach. An unclassified (i.e., sample) vector is assigned to the class, of those labels that are tied, and for that class the sum of distances from the sample to each neighbor in the class is a minimum. There are chances that there may still remain a tie.

---

**Algorithm 14.1** K-Nearest Neighbor Algorithm

---

1: Let $W = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ //set of $n$ labeled samples.
2: Input $\mathbf{y}$ //whose class is to be determined
3: Set $k$ to any value in $1 \ldots n$
4: Let $S = \phi$ //initial set of k-nearest neighbors (for given $k$)
5: **for** $i = 1$ to $n$ **do**
6:    compute distance $d_i(\mathbf{y}, \mathbf{x}_i)$ from $\mathbf{y}$ to $\mathbf{x}_i$
7:    **if** $(d_i \leq d_k)$ **then**
8:       $S = S \cup \{\mathbf{x}_i\}$
9:    **else**
10:       **if** $\mathbf{x}_i$ is more close to $\mathbf{y}$ than some previous neighbor **then**
11:          Delete farthest in the $S$
12:          $S = S \cup \{\mathbf{x}_i\}$
13:       **end if**
14:    **end if**
15: **end for**
16: Determine class in $S$ that is in majority
17: **if** (there is tie) **then**
18:    Compute the sum of distances of neighbors in each class with the one having tie
19:    **if** (there is no tie) **then**
20:       Classify $\mathbf{y}$ into minimum found class
21:    **end if**
22: **else**
23:    Classify $\mathbf{y}$ into the class of majority
24: **end if**
25: End

---

For numerical attributes, the distance metric used in the nearest neighbor is *Euclidean distance*. Considering two patterns, $\mathbf{x}_1 = x_{1_1}, x_{1_2}, \ldots, x_{1_m}$, and $\mathbf{x}_2 = x_{2_1}, x_{2_2}, \ldots, x_{2_m}$, distance between two using Euclidean is given by

$$d(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{\sum_{j=1}^{m}(x_{1_j} - x_{2_j})^2}. \tag{14.15}$$

To keep the spread of attribute values along each dimension approximately same, the above value of the distance is usually modified by scaling of the features. In that case the distance is represented by the following equation, where $a_j^2$ is the scale factor for the dimension $j$.

$$d(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{\sum_{j=1}^{m} a_j^2 (x_{1_j} - x_{2_j})^2}. \tag{14.16}$$

**Example 14.3**  Finding the nearest neighbor with $k$ patterns.

Figure 14.8 shows a problem of $k$-nearest neighbor classification, where $k = 6$. There are two patterns of each categories 1, 2, and 3, with numbers 1, 2, 3 marked after each pattern symbol, respectively. A collection of dots makes a pattern. For example,

**Fig. 14.8** $k$-nearest neighbor problem
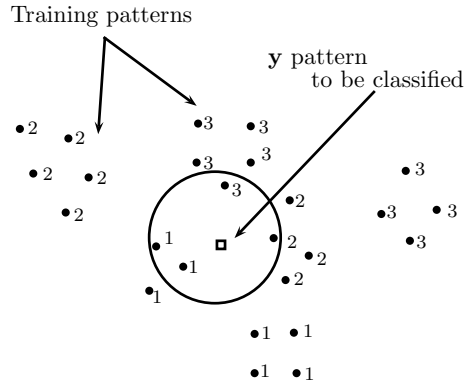


in the two patterns of category 1, there are three and four dots, each followed by 1s. The pattern **y**, represented by a box, is to be classified. In the circle enclosing, since the majority are in the category of 1, the new pattern **y** should be classified in category 1. □

## 14.7　Naive Bayes Classifiers

The Bayesian networks are powerful tools for decision-making and reasoning under uncertainty. These networks are specified by two components:

– *Graphical component*. It expresses uncertainty of causal relations, and consists of Directed Acyclic Graph (DAG) to represent causal relations and conditional probabilities of each node, given the parent of each. The vertices represent events and edges which are the relations between them.
– *Numerical component*. It quantifies different links in the DAG through conditional probability distribution of each node in the contexts of its parents.

Basically, the Bayes theorem permits optimal prediction of a *class* of new instances, given a vector $\mathbf{x} = \{x_1, ..., x_n\}$, of attribute's values. Since there is always insufficient training data to obtain accurate prediction of full joint probability distribution, the straight forward application of Bayes theorem is impracticable for machine learning. Therefore, there is need for making assumptions of independence to make the inference feasible.

The Bayesian classifier, called Naive Bayes, i.e., "straw man", approach takes this to the extreme when it is assumed that attributes are statistically independent, given the value of the class attribute. However, the above assumption never holds true, but still the Naive Bayes performs exceptionally well in most classification problems. Apart from this, the Naive Bayes approach is computationally efficient, as the training is linear in both the number of instances and attributes, and simple to implement [4].

**Fig. 14.9** Naive Bayes
network structure
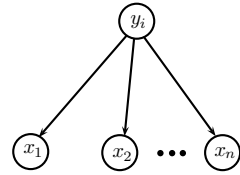


A Naive Bayes is composed of a DAG with one root node (called parent), which is an unobserved node representing *class* of each object to which the testing set belongs, and several leaf nodes, corresponding to observed nodes, with strong assumption of independence among these leaf nodes in the contexts of their common parent. The leaf nodes represent different *attribute* (*features*) specifying this object. Note that the Naive Bayes are composed of *two levels* only as shown in Fig. 14.9, whereas a Bayesian network may consist of many levels.

The decision of a Bayesian classifier is represented as a matrix of $P(x_j|y_k)$, which specifies the probability of occurrence of each feature value $(x_1, \ldots, x_j, \ldots, x_n)$ given each class $y_k$. To classify a new example having features among $x_1 \ldots x_n$, we make use of Bayes theorem as

$$argmax_{y_i} \, P(y_i|\bigwedge x_j) = argmax_{y_i} \, \frac{P(\bigwedge x_j|y_i)P(y_i)}{\sum_k P(\bigwedge x_j|y_k)P(y_k)} \qquad (14.17)$$

that computes $P(y_i|\bigwedge x_j)$, which is the probability of the example in class $y_i$ given the features $x_j$. The subexpression $\bigwedge x_j$ denotes a conjunct of attribute values all occurring in an example. The summation is performed over $N$ $(1 \leq k \leq N)$ classes, and the above probability is calculated for each class $y_i$, and then the class of the highest probability is selected. The probability $P(y_k)$ is estimated from the distribution of the training examples among classes. If independence of all attributes is assumed under a given context (i.e., class), then $P(\bigwedge x_j|y_k)$ can be calculated using

$$P(\bigwedge x_j|y_k) = \prod_j P(x_j|y_k). \qquad (14.18)$$

The values $P(x_j|y_k)$ are calculated from the probability matrix. Here, $x_j$ is the evidence on attribute nodes, which can be dispatched into pitches of $n$ evidence of features, $x_1, \ldots, x_j, \ldots, x_n$. Because Naive Bayes is working on the assumption that these attributes are independent of each other (given the parent node), their combined probability is obtained by substituting Eq. (14.18) into (14.17), which results in,

$$argmax_{y_i} \, P(y_i|\bigwedge x_j) = argmax_{y_i} \, \frac{\prod_j P(x_j|y_i)P(y_i)}{\sum_k [\prod_j P(x_j|y_k)P(y_k)]}$$

$$= argmax_{y_i} \, \prod_j P(x_j|y_i)P(y_i). \qquad (14.19)$$

Note that, there is no need to explicitly compute the denominator $\sum_k [\prod_j P(x_j|y_k)P(y_k)]$, since it is common among all the computation being a normalization constant. In some practice, $log$ is computed instead of a product, because probabilities involved can be very small.

The Bayesian learning method, as a classifier, builds the matrix $P(x_j|y_k)$ from training examples, by examining the frequency values in each class. It is possible to compute this matrix incrementally by incorporating one instance at a time. Alternatively, it can be constructed (i.e., without incremental approach), using all the data at a time.

Due to its simple structure, the Naive Bayes has many advantages. It is efficient, as the inference (i.e., classification) is achieved in a *linear time*. However, the Bayes network with general structure has complexity of *NP-complete*. Naive Bayes construction is incremental, in the sense that it can be easily updated, e.g., addition of new cases. The major problem with Naive Bayes is the assumption of strong independence relation, i.e., assumption of independence of features in the context of session class is not always true, and leads to negative influence on the inferred results.

Naive Bayes is most commonly used in text classification, where words are features, and presence/absence of a word can be used to determine the topic of a document.

Given $N$ number of training examples, each with $n$ number of attributes, complexity of generating probability matrix for Bayes classifier is $O(n.N)$. Hence, the classifier is substantially faster as the runtime is independent of the decision "rule"generated. Apart from this, the basic operation is performed only once.

## 14.8  Artificial Neural Networks

The science of machine learning is mostly experimental as there is a no universal learning algorithm existing yet. That is clear from the fact that, given a number of tasks, none can make a computer to learn every task well. A knowledge-acquisition algorithm is always required to be tested on learning tasks and data, that are specific to a given situation, and it is irrespective of whether it is recognizing a sunrise or it is doing a language translation. There is no method to prove that the given algorithm will be consistently better for all the situations. However, the human behavior apparently contradicts. We are fairly good at general learning abilities due to which we are able to master a number of tasks, like playing chess and playing cards. These arguments suggest and might serve as inspirations for building machines with some form of general intelligence. Therefore, the use of Artificial Neural Networks (ANN), which is a brain model, appears to be a logical justification for building intelligence systems [8].

The basic unit of the brain for performing the computation is a cell, called *neuron*; each one of them sends a signal to other neurons through very small gaps between the cells, called *synaptic clefts*. The property of any neuron of sending a signal through

this gap, and the amplitude of the signal together, is called as *synaptic strength*. As a neuron learns enough, its synaptic strength increases, and in that situation, if it is stimulated by an electrical impulse, there are better chances that it would send messages to its neighboring neurons.

The current neural-based learning algorithms need close involvement of humans, for producing better results. Majority of these algorithms are based on *supervised learning*, where each training example is carried out using human-crafted labels, about what is being learned. Consider an example of a picture of sunrise associated with the caption: "Sunrise". In that instance, the goal of the learning algorithm is to take the photograph as an input, and produce output, the name of the object in the image, i.e., "sunrise". We know that the mathematical process of transforming an input to the output is a *function*. The synaptic strength, which is a numerical value, produces this function, which corresponds to the solution to the learning through ANN.

It is interesting to note that it can be achieved through rote learning also, but it would not be useful. In fact, when we want to teach to the algorithm "what the sunrise is", then to have the algorithm recognize any sunrise, even the one for which we have not trained! This is the ultimate goal of machine learning algorithm.

The Artificial Neural Networks (ANN) possess the following important properties:

Learning ability,
Massive parallelism,
Adaptability,
Fault tolerance,
Distributed representation and computation,
Generalization ability, and
Low energy consumption.

which make them candidate for many applications. Although the details of the proposals vary, the most common models for learning and computation take the *neuron* as the basic processing unit. Each processing unit is characterized by the following:

– an activity level to represent polarization state of a neuron,
– an output value to represent firing rate of the neuron,
– a set of input connections,
– *synapses* on the cell and its dendrite,
– a bias value to represent an internal resting level of a neuron, and
– a set of output connections to represent a neuron's *axonal* projections.

Each of these aspects of the unit are represented mathematically by real numbers.

Hence, each connection of a neuron has an associated weight, called *synaptic strength*, which influences the effect of the incoming input on the activity of the unit. This weight is either positive, called *excitatory* or, negative, called *inhibitory*.

The basic model of artificial neuron with binary threshold is shown in Fig. 14.10. The mathematical neuron computes the weight as the sum of its *n* number of input
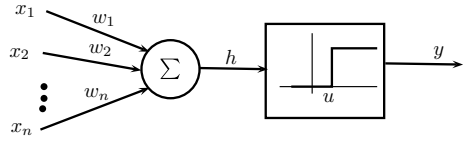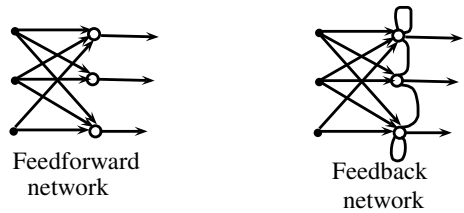
**Fig. 14.10** Basic model of
an artificial neuron



**Fig. 14.11** Single-layer
feedforward and feedback
networks



Feedforward
network

Feedback
network

signals $x_1, \ldots, x_n$, and the generated output is 1 if the sum is above some threshold
value $u$, otherwise the output is zero. This can be represented by

$$y = \theta \Big( \sum_{i=1}^{n} w_i x_i - u \Big) \tag{14.20}$$

where $\theta(.)$ is called a unit step function at 0, and $w_i$ is the *synapse* weight associated
with the *ith* input. For the sake of simplicity, we consider the threshold $u$ as another
weight, $w_0 = -u$ attached to the neuron with a constant input $x_0 = 1$. A properly
chosen weight allows a synchronous arrangement of such neurons to perform uni-
versal computations. There is a crude analogy of this neuron model to biological
neurons as follows: the wires and interconnections model the *axons* and *dendrites*,
respectively, in the biological neuron; connection weights in this model correspond
to *synapses* in biological neuron; and threshold function approximates the activity in
*soma*. However, this model is an overly simplified one of a true biological neuron.

The ANNs can be considered as weighted directed graphs, where artificial neurons
act as nodes, and directed edges with weights are connections between neurons, and
between outputs and inputs of neuron.

Based on the connection pattern an ANN can be classified as

1. *Feedforward networks*: In these, the direct graphs have no loops, and
2. *Recurrent feedback networks*: There are loops because of feedback connections.

Figure 14.11 shows the single-layer feedforward and feedback neural networks.
The most common family of *feedforward networks* is the *multilayer perceptron*. The
neurons in these networks are organized into layers with unidirectional connections
between them. Usually, the feedforward networks are of a *static* type, as they produce
only one set of output values instead of a sequence of values for a given input. These
networks are without memory, as their output is independent of the previous states
of the network.

The feedback networks are called *recurrent networks*, and are dynamic systems. Neurons' output patterns are computed when a new input pattern is presented to these networks. Due to feedback paths, the input to each neuron is then modified, which causes the network to enter a new state.

The problem of *learning* in neural networks is simply the problem of finding a set of connection strengths which allows the network to carry out the desired computation [10].

An ANN usually learns of the connection weights from the available training patterns, the performance of which gets improved over time through iterative updating of the weights. The ability of ANNs to automatically learn from examples makes them attractive and exciting. From a given collection of representative examples, the ANNs learn underlying rules as a form of input–output relationship, instead of following the rules specified by human experts. This simple factor is a major advantage of neural networks over traditional experts systems.

To understand this learning process, first there is need of a model of learning environment in which these neural networks operate. That means, we must know about what information is available to the network. This model is called the *learning paradigm*. The second requirement is to know about how these weights are updated. This means to know, as to which rule controls the updating process of the weight. In fact, a learning algorithm refers to some procedure which uses learning rules to adjust the weights of inputs.

All the three paradigms of learning exist in the neural networks, i.e., *supervised learning*, *unsupervised learning*, and *hybrid learning*. Supervised learning requires a teacher, where the network is provided with correct answers, i.e., output, for every input pattern. The weights are so determined that they allow the network to produce answers, which are as close as possible to known correct answers.

The unsupervised learning-based neural networks explores the underlying structure in the data, or correlations between patterns in the data, and organizes patterns into categories based on the correlations.

A hybrid learning ANN combines both the supervised and unsupervised approaches.

The learning rules in the neural networks are of four basic types. These are *error correction*, *Boltzmann*, *Hebbian*, and *Competitive learning*. Their basic principles are presented in the following.

## 14.8.1  Error-Correction Rules

A learning based on error-correction rules makes use of simple concepts. During the training, the input $x_1, \ldots, x_n$ is applied to the network, and the flows through the network generates a set of values in the units of output $y$. As the next step, the actual output $y$ is compared with the desired target $d$. If the output and the target match, no change is made to weights. If there is no match, change is made to weights of some of the connections. The problem is to find out as which connections in the

network were at fault that caused the error and resulting in a mismatch. Obviously, it is *supervised learning* paradigm. The principle used for error-correction learning rules is based on the error signal $(d - y)$, to modify the connection weights so as to gradually reduce the error magnitude.

The *perception-based learning* works on this principle of error correction. A *perceptron* comprises a single neuron with adjustable weights, $w_i$, $i = 1, \ldots, n$, and a threshold value $u$, as shown in Fig. 14.10 (page no. 432). Given an input vector $\mathbf{x} = (x_1, \ldots, x_n)^t$, where $t$ is the iteration number, net input to the neuron is expressed as

$$v = \sum_{i=1}^{n} w_i x_i - u. \qquad (14.21)$$

If $v > 0$ the output $y$ of the perceptron is $+1$, otherwise it is 0. In a *classification* problem (of two classes, say $A$ and $B$), the perceptron assigns an input pattern to class $A$ if $y = 1$, and to class $B$ if $y = 0$.

---

**Algorithm 14.2** Perceptron learning Algorithm

---

1: Initialize weights $w_1, ..., w_n$, and threshold $u$ to some small random numbers,
2: Apply a small pattern input vector $(x_1, \ldots, x_n)^t$ and evaluate the output of the neuron, as per equation (14.21),
3: Update each weight $w_i$ $(i = 1, ..., n)$ according to: $w_i(t + 1) = w_i(t) + \eta(d - y)x_i$.

---

Algorithm 14.2 is *backpropagation* learning algorithm, based on *error-correction principle*, where $d$ is the desired output, $t$ is the iteration number, and $\eta$ $(0.0 < \eta < 1.0)$ is the gain, i.e., size of the step.

## 14.8.2   Boltzmann Learning

The learning in ANN based on Boltzmann machines has many properties: they are symmetric and recurrent (i.e., feedback) networks. It consists of binary units ($+1$ for "on", and $-1$ for "off"). The symmetric property means, weight on the connection from unit number $i$ to unit $j$ is identical to weight from unit $j$ to unit $i$, formally, $w_{ij} = w_{ji}$. The neurons are also of two types: 1. *Visible neurons* are a subset of the entire set of neurons, and they interact with the environs, and 2. *Hidden neurons* are the remaining neurons, which do not interact. Each neuron is a stochastic unit, which generates and outputs (or it is a state) according to the Boltzmann distribution of statistical machines.

A Boltzmann machine also operates in one of two modes: 1. *Clamped* mode, where visible neurons are clamped onto a specific state determined by the environment; and 2. *Free-running* mode, in which both the visible and hidden neurons are allowed to operate freely.

The Boltzmann learning algorithm works to adjust the connection weights in such a way that the desired probability distribution is satisfied by the states of the visible units. According to the rule of Boltzmann learning, the change in the connection weight $w_{ij}$ is given by

$$\Delta w_{ij} = \eta(\overline{\rho}_{ij} - \rho_{ij}), \tag{14.22}$$

where $\eta$ is learning rate, $\overline{\rho}_{ij}$ and $\rho_{ij}$ are correlations between the states of units $i$ and $j$ when the network operates in the clamped mode and free-running mode, respectively. The values of $\overline{\rho}_{ij}$ and $\rho_{ij}$ are estimates using Monte Carlo experiments [8].

### 14.8.3   Hebbian Rule

The Hebbian learning rule specifies the magnitude of the weight by which the connection between two units is increased/decreased in proportion to the product of their activation. It builds on the Hebbs's learning rule, which states that the connections between two neurons might be strengthened if the corresponding neurons fire simultaneously. This rule works well as long as the input patterns are uncorrelated, however, this condition places serious limitations on the Hebbian learning rule. A Hebbian rule for ANN is described as

$$w_{ij}(t + 1) = w_{ij}(t) + \eta y_j(t)\, x_i(t), \tag{14.23}$$

where $x_i$, $y_j$ are the output values of neurons $i$ and $j$, respectively. These are connected by the weight $w_{ij}$, and $\eta$ represents the learning rate. An important property of this approach is that learning is carried out locally, that is, the change in synapse weight depends only on the activities of two neurons connected by it. This simplifies the implementation of the circuit.

A more powerful learning rule is the delta rule, that utilizes the discrepancy between the desired and actual output of each output unit to change the weights feeding into it.

### 14.8.4   Competitive Learning Rules

The competitive-learning units compete among themselves for activation, which is in contrast to the Hebbian learning, where multiple output units can be fired together. Hence, only one output unit is active at any given time. The biological neurons follow this type of learning.

The competitive learning often categorizes or clusters the input data, where similar patterns are grouped by the network and represented by a single unit. The grouping is carried out automatically, which is actually based on the correlations.

The simplest possible competitive learning network has a single layer of output units, as shown in Fig. 14.11, where each output unit $j$ connects to all the input units $x_i$s, through the weights $w_{ij}$, $i = 1, \ldots, n$. Each output unit also connects to all other units via inhibitory weights, but has a self feedback with an excitatory weight. Due to the competition, only the unit $i$ with the largest net input becomes the winner. This can be expressed by

$$\forall i \; \mathbf{w}_i^* \; \mathbf{x} \geq \mathbf{w}_i \; \mathbf{x}. \tag{14.24}$$

In this learning, only the weights of the winner unit gets updated.

### 14.8.5   Deep Learning

Beginning from 2005, deep learning—a neural net-based approach, which is driven for its inspiration from brain science—began to come into its own, and has now become a singular force propelling AI research forward.

The deep learning is concerned with simulation of ANNs that "learn" gradually, in the areas of image processing, speech recognition, and understanding, and even to make the decisions of their own. The basic technique relies on ANNs, which do not precisely mimic as to how the actual neurons are working. Instead of this they are based on the general principles of mathematics that allow them to learn from examples to recognize people or objects in a photograph, and translate the spoken language from one to another. The technologies based on deep learning have transformed the AI research, and have produced far accurate results in speech recognition, computer vision, natural language processing, and robotics.

To be successful in generalizing after observing a number of examples, deep learning network needs more than just the examples. For example, it depends on hypotheses about the data and assumptions about what can be a possible solution for a particular problem. A typical hypothesis that can be built into a system might conclude that if data input for a particular function in two situations are almost similar, the output should not change drastically. For example, on altering a few pixels in an image of a dog will not transform it into a picture of cat.

One type of a neural network that consists of hypotheses about images is called *convolutional* neural network. These networks when used in deep learning have many layers of neurons, which are organized in such a way that the output is less sensitive to the deviation from the original object, due to changes in the input image. For example, we will note the changes in a face when viewed from different angles, however we will still recognize it correctly. A well-trained deep learning network will also do a similar job.

The design of convolutional networks take their inspirations from multilayered structure of *visual cortex*—part of the brain that receives input from eyes. Too many layers of virtual neurons in a convolutional neural network are what that make the network "deep", and hence it is better able to learn about the world about it.

## 14.9   Instance-Based Learning

Instance-based learning (IBL) approaches uses supervised learning techniques. There are several variants of IBL, e.g., *exemplar-based learning*, *case-based reasoning*, and *memory-based learning*. Though all these methods emphasize somewhat different aspects, all these approaches are founded on the basic concept of an *instance* or a *case*, as a basis of knowledge representation and reasoning. The meaning of *case* here is, observation or example, or incident, which is a single experience, e.g., a pattern, along with its solution, is a problem of pattern recognition.

In general, a problem along with its solution is a case-based reasoning. To highlight the main properties of IBL, it is important to understand its difference with *model-based* learning. As a typical case, IBL methods learn by simply storing some of the observed examples. The processing of these inputs are differed until a prediction or some other query is actually requested. Later, predictions are derived by combining information from stored examples, in some way. Once the query is answered, the prediction and intermediate results are discarded.

In contrast, the *model-based* or *inductive-based* learning derive predictions in an indirect way as follows: as a first step, observed data is used in order to induce a model, say, a *decision tree* or a *regression function*. As a second step, the predictions are obtained using this model, which can also serve as the function of explaining.

Generally, the model-based algorithms, also called *eager algorithms*, carry higher complexity during the training phase than the instance-based algorithms. The latter are also called *lazy* algorithms, where learning is basically storing of selected algorithms. The lazy methods also need more storage requirements, in a linear order of the size of the input, and higher computational cost compared to deriving of a prediction.

The IBL algorithms make use of specific instances, instead of pre-compiled abstractions during the prediction. They also describe the probabilistic concepts, because they use *similarity* functions to yield graded matches between instances.

The IBL algorithms are derived from the nearest neighbor (NN) pattern classifiers, which also do not save and use only selected instances to generate the classification predictions. Thus, they are also called as edited NN algorithms. They also maintain the prefect consistency with the initial training set.

### 14.9.1   Learning Task

The instance-based learning makes use of the supervised approach, or learning from examples. The input is a sequence of instances, where each instance is a set of $n$ attribute–value pair, thus creating an $n$-dimensional instance space. Only one of these attributes corresponds to a category attribute, and the other attributes are predictor attributes. A category is a set of all instances in the instance space which have the same value for their category attribute. For the sake of simplicity, it is assumed that categories are disjoint.

The main output of IBL algorithms is *concept* or *concept description*. The algorithm is a function that maps instances to categories, i.e., for a given instance from a instance space, it produces the classification—a predicted value for instance's category attribute.

An instance-based concept description comprises a set of stored instances and, some information about their past performances, e.g., the number of correct and incorrect classification predictions. The set of instances can change once each training instance has been processed. The concept descriptions are decided based on how the selected similarities and classification functions of the IBL algorithm make use of the current set of saved instances. In all the IBL algorithms, at least two of the following three components constitute these functions [1].

*Function of Similarity*

This function computes the similarity between two instances: one is a training instance, and the other is in the concept description. The similarities are represented in numeric values.

*Classification Function*

The input to this function are 1. similarity function's result, and 2. classification performance records of instances in the concept description, which produces output of the classification.

*Concept Description Updater*

An updater keeps records of classification performance and resolves as to which instance to include in the concept description. Inputs to the concept description updater are training instance, similarity results, classification results, and current concept description. The output of the updater is the modified concept description.

The first two functions in the above decide how the saved instances in the concept description can be used to predict the category attributes. Thus, concept description in IBL not only comprises a set of instances, but also these two functions.

There is an important difference between IBL algorithms and most other supervised learning methods: the IBL algorithms do not construct explicit abstractions, like decision trees or rules. Most of the learning algorithms produce generalizations using the instances, and use simple matching procedures to classify the instances when presented in future. This eliminates the need of storing rigid generalizations of concept descriptions for IBL algorithms.

### 14.9.2  IBL Algorithm

Algorithm 14.3 is the simple instance-based learning algorithm. The similarity function, $sim(x, y)$, used in the algorithm is expressed by

$$sim(x, y) = - \sqrt{\sum_{i=1}^{n} f(x_i, y_i)}. \tag{14.25}$$

To compute the similarity, the instances have $n$ attributes. For numerical attributes, the relation used is

$$f(x_i, y_i) = (x_i, y_i)^2. \tag{14.26}$$

For attributes with Boolean and symbolic values, $f(x_i, y_i) = (x_i \neq y_i)$, i.e., $f$ is false when $x_i \neq y_i$. The missing attribute values are taken as having maximum difference from the value present. If both are missing, then $f(x_i, y_i) = 1$. IBL Algorithm 14.3 is very similar to the nearest neighbor algorithm (see page no. 425) we have studied earlier. The only difference in the IBL algorithm is that it normalizes its attributes' ranges, processes instances incrementally, and has a simple policy that allows missing values of attributes [1].

It is important to note that this algorithm's concept description changes over time. In the $k$-NN algorithm, the classification function simply assigns classifications according to the nearest neighbor policy. In the IBL algorithm, we can find out instances in the instance space that will be classified by each of the stored instances. The term *CD* in the algorithm stands for *concept description*, and *TS* is the *training set*.

---

**Algorithm 14.3** IBL algorithm

---

1: Initialize: $CD = \phi$
2: **for** every $x \in TS$ **do**
3:     **for** every $y \in CD$ **do**
4:         $SIM[y] = sim(x, y)$
5:     **end for**
6:     $y_{max} = \exists y \in CD$ with maximal $SIM[y]$
7:     **if** $class(x) = class(y_{max})$ **then**
8:         classification = True
9:     **else**
10:         Classification = False
11:     **end if**
12:     $CD = CD \cup \{x\}$
13: **end for**
14: End

---

## 14.10 Summary

Machine learning systems help the programs automatically learn from the data; it can be said to be a kind of a data mining. Machine learning is used in computer science and beyond, e.g., in search engines, spam filters, advertisements, credit scoring, fraud detection, stock trading, drug design, and in many other applications.

A learning algorithm can be expressed as comprising three parts:

1. *Representation*,
2. *Evaluation*, and
3. *Optimization*.

The popular approaches of statistical machine learning are *support vector machine*, *k-nearest neighbor* algorithm, *Naive Bayes*, and *instance-based learning*.

The support vector machines (SVMs) are *supervised learning models* which use associative learning algorithms, which can analyze the data and recognize patterns. Thus they have applications in classification and regression analysis.

An SVM is a model where examples are nothing but a representation of points in space, which are mapped to two classes, so that the examples of the separate categories are divided by a clear gap that is as wide as possible. The latter is to help in making clear distinctions between the categories. New examples are then mapped into that same space and predicted about their category based on which side of the gap they fall on. The support vector machine (or support vector network (SVN)) maps the input vectors into some high-dimensional feature space through some nonlinear mapping chosen a priori. A limitation of SVMs and other statistical learning methods is that the category structure is treated as "flat"and that they do not consider any relationships between categories, which are commonly expressed in concept *hierarchies* or *taxonomies*. The *structured SVMs* are capable of learning the taxonomical architectures.

The problem with taxonomical structures is that the number of classes are very large. For example, in parsing, the number of possible parse-trees is the exponential factor of the length of the sentence, and this scenario is similar for a majority of other problems that are designed to predict the output, which is structured in nature. Thus, there is need for exploring a more compact representation for these large output spaces.

The $k$- nearest neighbor method is a *statistical* method, where, given a training set $\theta$ of $n$ labeled patterns, a nearest neighbor algorithm decides that some new pattern, $\mathbf{x}$, belongs to the same category, as do the closest neighbors in $\theta$. The $k$-NN algorithms are used for the classification of objects, in many practical applications, e.g., in the areas of artificial intelligence, pattern recognition, statistics, cognitive psychology, vision analysis, and in medicines. Under many circumstances, the $k$-NN algorithm is used to perform the classification.

Bayes networks are powerful tools for decision-making and reasoning under uncertainty. These networks are specified using two components: 1. a *graphical component*, composed of directed acyclic graph (DAG) to represent causal relations, and 2. *numerical component,* consisting in a quantification of different links in the DAG by conditional probability distribution. The Bayesian classifier (Naive Bayes), i.e., "straw man" assumes that the attributes in the examples are statistically independent of each other, given the value of the class attribute, which makes it computationally efficient.

The most common models for *learning* and *computation* take the *neuron* as the basic processing unit. Each such processing unit has following characteristics:

1. *Activity level*. It is the state of polarization of a neuron.
2. *Output value*. It depends on the firing rate of the neuron.
3. *Input connections*. It is the collection of *synapses* on the cell and their dendrite.
4. *Bias value*. It is an internal resting level of the neuron.
5. *Output connections*. These are neuron's *axonal* projections.

The Artificial Neural Networks (ANNs) are weighted directed graphs, where nodes are treated as artificial neurons and directed edges (with weights) are connections between neurons, and also they act as connections between input and outputs. The problem of learning in neural networks is the problem of finding a set of connection strengths which can allow the network in future to carry out the desired computation. There exist all three paradigms for learning in a neural network: *supervised*, *unsupervised*, and *hybrid*.

Instance-based learning (IBL) approaches is supervised machine learning technique. Several variants of instance-based approaches have been devised, e.g., *memory-based learning*, *exemplar-based learning*, and *case-based reasoning*. The information provided by the stored examples is used in some way, to perform the predictions in the IBL. Once the query has been answered, the prediction itself and other intermediate results are discarded.

As opposed to IBL, the *model-based* or *inductive learning* methods derive predictions in an indirect way: in step one, the observed data is used in order to induce a model, say a *decision tree* or a *regression function*. Then in the second step, the predictions are obtained on the basis of this model, which can also serve other purposes like explaining or justifying the inferences.

## Exercises

1. "The task of text categorization is to assign a given document to one of the categories out of a fixed set of categories. This is done on the basis text contents. The Naive Bayes model is often used for this purpose, where a query variable is the document category and the "effect" variables are presence/ absence of each word in the language. It is assumed that words occur independently in the documents, and their frequencies determine the document category." For this statement,

   a. Explain how such a models can be constructed, given a set of "training data" in the form of documents that have been already assigned to categories.
   b. Explain how to categorize a new document.
   c. Is the independence assumption reasonable? Justify your answer.

2. What linear or nonlinear function is used by an SVM for performing classification? How is an input vector $\mathbf{x}_i$ (instance) assigned to the positive or negative classes.

**Fig. 14.12** Training data for SVMs



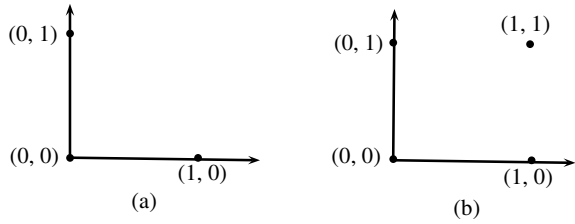(a)                                                                              (b)

**Table 14.1** Data set for machine learning

| Department | Class | Age | Salary K$s |
|---|---|---|---|
| Sales | Programmar | 36 . . . 40 | 51 . . . 55 |
| Sales | Assistant | 31 . . . 35 | 31 . . . 35 |
| Sales | Assistant | 36 . . . 40 | 36 . . . 60 |
| Production | Assistant | 26 . . . 30 | 51 . . . 55 |
| Production | Programmar | 36 . . . 40 | 71 . . . 75 |
| Production | Assistant | 31 . . . 35 | 51 . . . 55 |
| Marketing | Programmar | 41 . . . 45 | 51 . . . 55 |
| Marketing | Assistant | 36 . . . 40 | 46 . . . 50 |

3. Consider the SVM for the training data given in $\Re^2$, in Fig. 14.12a, b; find out the separating hyperplanes in both the cases.
4. Why is the Naive Bayes classification called Naive? What are the main goals behind this classification?
5. Consider the data given in Table 14.1,
   and use these to train a Naive Bayes classifier with designation attribute as the class label and all the remaining attributes regarded as input. Once you have your Naive Bayesian classifier, test the following unseen instances to find out the class:

   a. Marketing, $36 \ldots 41$, $51K \ldots 55K$
   b. Sale, $36 \ldots 41$, $71K \ldots 75K$

# References

1. Aha DW et al (1991) Instance-based learning algorithms. Mach Learn 6:37–66
2. Boser EB et al (1992) A training algorithm for optimal margin classifiers. In: Proceedings of the fifth annual workshop on computational theory, COLT' 92. ACM, New York, pp 144–152
3. Bradley P (2002) Scaling mining algorithms. Commun ACM 45(8):38–43
4. Clark P, Niblett T (1989) The CN2 induction algorithm. Mach Learn 3:261–283
5. Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20:273–297
6. Domingos P (2012) A few useful things to know about machine learning. Commun ACM 55(10):78–87

7. Fukunaga K, Narendra PM (1975) A branch and bound algorithm for computing K-nearest neighbors. IEEE Trans Comput 750–753
8. Jain AK et al (1996) Artificial neural networks: a tutorial. Computer 3:31–46
9. Joachims T (2009) Predicting structured objects with support vector machines. Commun ACM 52(11):97–104
10. Rummelhart DE et al (1994) The basic ideas in neural networks. Commun ACM 37(3):86–92
11. Shawe-Taylor J (2009) Machine learning for complex predictions. Commun ACM 52(11):96–96

# Chapter 15
# Automated Planning

**Abstract** Automated planning deals with the tasks of finding out ordered sets of actions that allow a system to transform an initial state to a state satisfying goal specification. The set of actions is a plan, and it belongs to PSPACE-complete. Automatic planing or scheduling generates a set of actions automatically. This chapter presents the nature of automated planning, the classical planning problem, agent types that execute the problem, and worked examples. It also covers, the concepts and implementation aspects of forward planning, partial-order planning, planning languages, a case study of general planning language—STRIPS, and search strategies. Planning with propositional logic, planning graphs, and hierarchical network planning are demonstrated. The multiagent planning techniques are presented for goal and task refinement, decentralized planning, and on how to do coordination after it is planned. This is followed by the chapter summary, and a set of exercises.

**Keywords** Automated planning · Scheduling · Forward planning · Partial-order planning · Planning languages · STRIPS · Multiagent planning

## 15.1 Introduction

Automated planning is concerned with the problem of finding a set of actions to be carried out in an ordered way to complete a job, such that they allow a system to transform an initial state into a state that satisfies the goal specification. In a deterministic system, these sets of actions are called *plans*, while in nondeterministic systems they are called *policies*. Finding a plan, or even deciding its existence, has shown to be *PSPACE-complete*[1] in the deterministic planning, unless severe restrictions are applied to the search [1].

Developing automated methods for reasoning about plans and schedules and for generating the plans has remained the part AI, which helped both the autonomous

---

[1]In the theory of computational complexity, a decision problem is in complexity class *PSPACE-complete* if it can be solved using a memory whose size is polynomial on the size of input (i.e., *polynomial space*), and if every other problem that can be solved in polynomial space can be transformed into it in polynomial time.

445

and human agents. There is always a need for planning when an agent wants to control the evolution of its environment. An algorithm for a planning problem has the following inputs:

1. possible courses of actions,
2. a predictive model that underlies the dynamics, and
3. a performance measure that evaluates the courses of actions.

The output or solution to a planning problem is one or more courses of actions that satisfies the specified requirements for the minimum performance. Hence, a planning problem involves deciding "what" actions to be performed and "when" they should be performed. The "when" part of the problem has been traditionally called the *scheduling problem*. The separation between "what" and "when" is motivated by computational considerations. In fact the algorithm for automating scheduling has been around for a long time, for example, in the areas of operations research as well for automating the complete planning problems.

The classical scheduling methods are concerned with the planning problems where resource constraints are static, and prespecified in numerical form. This is in contrast to the AI methods for scheduling that allow declarative specifications for both symbolic and numeric resource constraints and handle dynamic changes in the constraints, i.e., while in execution. The Constraint Satisfaction Problems (CSFs) (see Chap. 10) form the canonical backbone of most AI scheduling methods, and there are several sophisticated heuristic search strategies that are common for such problems [11].

Although the classical panning model has been prevailing for a long time, a majority of research in planning is toward planning in environments that are *dynamic*, *stochastic*, and in a *partially observable* world. These requirements are not compatible with the classic planning assumptions, which are for deterministic, static, and fully observable world. To cope with the partially observable environments, gathering of information is designed as part of the planning activity, and the existing classical planning techniques are extended to accommodate interleaving of scheduling and planing tasks. The environments that are stochastic are modeled using Markov decision processes, while the planning in such environments requires construction of policies for the corresponding Markov decision processes [6].

All variants of domain-independent planning problem are known to be computationally hard (*P-SPACE complete* or worst); efficiency can be improved only through exploitation of problem distribution space and through the knowledge of domain structure. For improving the efficiency of plan generation, formal planning models are preferred. Following are the basic approaches for achieving efficiency enhancement:

1. split both, domain and the problem into tractable sub-components using decomposition techniques,
2. find out the control information using inductive and speedup learning techniques,
3. define abstractions for expected problem distribution, and
4. define language(s) to express domain-specific search control information.

Another approach to planning is extending and complementing the planning models that are based exclusively on techniques that define subgoals. A promising approach is using constraint-logic-programming-type models that allow both: defining subgoals and constraint-satisfaction criteria. These work together in controlling the planner, which results in handling action selection and scheduling of planning problems through appropriate computational models. In addition to the above, plan generation can also be carried out using network flow approaches, which are based on a form of disjunctive projection of future states of the agent [9].

So far, both the *classical* and *stochastic* planning techniques are in common used in the design of autonomous agents. The AI planning techniques are also common in software domains such as database query planning and Internet browsing, however, they have less impacted the problems that support industrial applications. Although the industrial area has an enormous scope for planning problems—be it project planning, process planning, or maintenance planning—the automation so far has progressed to supporting scheduling, particularly concentrating on harder *action-selection* problem, which are more concerned with the humans domain.

The type of plannings, which are *incremental* or *interactive* planning in nature, are new areas in plannings. However, many planning applications involve reasoning with a variety of constraints that are not of temporal type, e.g., the problems of path planning, assembly planning, and manufacturing problems need *spatial* and *geometric-reasoning* capabilities. Due to the nature of these constraints, it is neither possible to ignore or abstract away, nor it is advisable to encode them in a homogeneous representation. These conditions make it necessary to have an interaction between the planner and the reasoners. Hence, understanding of the modalities of interaction between a planner and the external reasoners, which may be both humans or machines, is thus important [7].

**Learning Outcomes of This Chapter**:

1. Define the concept of a planning system and how it differs from classical search techniques. [Familiarity]
2. Describe the differences between planning as search, operator-based planning, and propositional planning, providing examples of domains where each is most applicable. [Familiarity]

## 15.2 Automated Planning

Automated planning techniques are now commonly being applied in a number of tasks, that include, robotics, process planning, web-based information gathering, autonomous agents, and spacecraft mission control. A solution to a problem in automated planning can be described in terms of a sequence of steps that transforms some initial description of the problem state, for example, the initial configuration of a puzzle, into a description satisfying a specified goal criterion. The steps (transformations) are called *operators* and a problem is defined in terms of a set of operators

and a language for describing problem states. The problem states correspond to instantaneous descriptions of the world and operators correspond to *actions* that an agent can perform to change the state of the world [3].

The planning is about how an agent achieves its goals, even the simplest ones an agent must reason about the future. Since the goal is not achievable in a single step, the number of steps to be carried out needs to be broken up into subtasks, and steps for each needs to be the goal; what an agent will do in the next step also depends on its past.

To complete each subtask, there are actions, which need to be carried out; each action has a subsequent state as well as a preceding state. To be simple at start of this subject, the following assumptions are made:

1. the actions are deterministic, i.e., the agent can determine the consequent of the actions,
2. the world is fully observable, i.e., the agent can observe the correct state of the world, and
3. the closed world assumption, i.e., the facts not described in the world are false.

## 15.3 The Basic Planning Problem

A basic planning problem usually comprises an initial world description, a description of the goal world, and a set of actions (sometimes also called *operators*) that map a world description to another. A solution is a sequence of actions leading from the initial world description to the goal world description, referred to as a *plan*.

A *deterministic action* is a *partial function* from states to states.[2] For example, a robot can move block $x$ onto $y$, if $x$ has top-clear, $y$ has top-clear, and obviously, $x \neq y$. Also, all the alternatives are not available. A *precondition* of an action decides about when the action can be carried out, and whether the resulting state due to an action is the effect of the actions.

**Definition 15.1** (*Planning Problem*) The relevant part of the world is in a certain state, but managers or directors would like it to be in another state. The (abstract) problem of how one should get from the current state of the world through a sequence of actions to the desired goal state is a planning problem. □

AI planning techniques are techniques to search for a plan: *forward planning* is a planning technique building a plan starting from the initial state; *backward planning* starts from the goal states; and *least-commitment planning* constructs plans by adding actions in a non-sequential order.

Ideally, to solve such planning problems, we would like to have a general planning-problem solver. However, such an algorithm, solving all planning problems, can be proven to be non-existing (that is, the general planning problem is undecidable). We

---

[2]Partial function: Every state "(state, action)" pair does not necessarily result in a state.

therefore, try to concentrate on a simplification of the general planning problem called *the classical planning problem*. Although not all realistic problems can be modeled as a classical planning problem, they can help to solve more complex problems.

### 15.3.1   *The Classical Planning Problem*

Classical planning problem is the simplest case of planning, where the environment is static and deterministic, and the planner has complete knowledge about the current state of the world. Two important issues are faced by the classical planners: 1. To model the actions and changes, and 2. To organize search for plans (i.e., sequences of actions) that are capable of achieving the goals. The *logic programming* and *nonmonotonic reasoning* are frequently used in planning and search, however, many implemented planners have used a variant of action model, called the STanford Research Institute Problem Solver (STRIPS). This model represents the state of the world in terms of state variables and their values, and actions as state-transforming functions, which are deterministic in nature. Most of the early planners modeled the state-space exploration of the world-states as a search, and the transitions between states represented the actions.

The state exploration method worked suitable only for problems having small search space, due to the complexity in space and time. Instead, the utility of manipulating partial plans during search became popular, and this led to the design of algorithms that search in the space of partial plans.

The classical planning problem is defined as follows. Given

1. a description of the known part of the *initial state* of the world (in a formal language, usually propositional logic) denoted by **I**,
2. a description of the *goal* (i.e., a set of goal states), denoted by **G**, and
3. a description of the possible *atomic actions* (**R** (i.e., rule)) that can be performed, modeled as state transformation functions,

determine a plan, i.e., a sequence of actions that transforms each of the states fitting the initial configuration of the world into one of the goal states. Thus, classical planning problem is a tuple $\langle I, G, R \rangle$. Consider the following example of planning the "Transport a passenger by cab."

**Example 15.1**  Classical planning problem of "Transporting by cab."

Suppose that initially (i.e., in all states of the world that match the description **I**), there is a cab at a location $A$, represented by a binary state variable $cab(A)$, and a passenger at a location $B$, represented by $passgr(B)$. In each of the states described by **G** the passenger should be at a location $C$, denoted by $passgr(C)$. Furthermore, suppose that there are three actions (move, load, unload) that can transform (some part of) the state of the world.

**Table 15.1** Classical planning problem

| State | Transition | Comment |
|---|---|---|
| 0. | *I* | ; Initial state |
| 1. | *move*(*A*, *B*) | ; cab moves from location A to B |
| 2. | *load*(*passgr*) | ; passenger gets into cab |
| 3. | *move*(*B*, *C*) | ; cab moves from location B to C |
| 4. | *unload*(*passgr*) | ; passenger unloads from cab |
| 5. | *G* | ; goal: passenger at location C |

Following are the steps for actions:

1. The cab can move from one location to another: *move*(*x*, *y*) with *x*, *y* ∈ {*A*, *B*, *C*}. This action requires that a priori *cab*(*x*) holds, and ensures that in the resulting state ¬*cab*(*x*) and *cab*(*y*) hold, that is cab is not at place *x* as well not at the place *y*.
2. The passenger can get into the cab: *load*(*passgr*). This action requires a priori *cab*(*x*) and *passgr*(*y*) and *x* = *y*, and in the resulting state both ¬*passgr*(*y*) and *passgr*(*cab*) (i.e., passenger in cab) should hold.
3. The passenger can get out of the cab: *unload*(). This action requires that cab is at location *x* (*cab*(*x*)) and passenger in cab (*passgr*(*cab*)), and results in ¬*passgr*(*cab*) and *passgr*(*x*).

With **I** as the initial set of states, and **G** as a set of goal states, the sequence of state transitions can be indicated as shown in Table 15.1. □

## 15.3.2 Agent Types

Agents are classified according to the techniques they employ in their decision making:

1. *Reactive agents*: They base their next decision solely on their current sensory input.
2. *Planning agents*: They base their course of action considering the anticipated future situations, possibly as a result of their own actions.

Whether an agent should plan or it should be reactive, depends on the particular situation it finds itself in. Consider the case where an agent has to plan a route from one place to another. A *reactive* agent might use a compass to plot its course, whereas a *planning agent* would consult a map. Clearly, the planning agent will come up with the shortest route in most cases, as it will not be confronted with uncrossable rivers and one-way hills. On the other hand, there are also situations where a reactive agent can be at least as effective, for instance, if there are no maps to consult such as in

**Fig. 15.1**  World states and
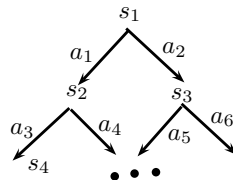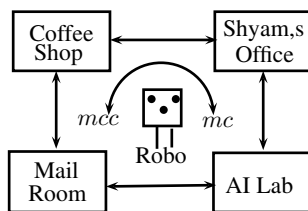agent (robot) actions



**Table 15.2**  Table for
mappings: State × Action →
State

| State | Action | Resulting state |
|-------|--------|-----------------|
| $s_1$ | $a_1$ | $s_2$ |
| $s_1$ | $a_2$ | $s_3$ |
| $s_2$ | $a_3$ | $s_4$ |
| $s_2$ | $a_4$ | $s_5$ |
| … | … | … |

**Fig. 15.2**  Delivery robot
with service locations



a domain of planetary exploration, like Mars or Moon. Nevertheless, the ability to
plan ahead is invaluable in many domains.

Considering the *states set* as $s_1$, $s_2$, ..., and set of *actions* as $a_1$, $a_2$, ..., they can
be represented using a tree shown in Fig. 15.1 or explicitly by Table 15.2.

We consider the following example to better understand the actions and states
world, where agent is a robot, called *Robo*.

**Example 15.2**  A delivery Robot's (Robo's) planning.

A delivery robot shown in Fig. 15.2 is responsible for some jobs. It can pick up mail
from the mail room and deliver it to Shyam's office, and also can pick up coffee from
the coffee room and can deliver it to Shyam's office. The robot, called *Robo*, can also
reach these places by moving clockwise (*mc*) as well as moving counterclockwise
(*mcc*). Assume that the mail it handles is a postal mail (not the email). The domain
of the world is represented by the terminology described as follows [13].

Various *locations* are represented by the following symbols. Each of them can be
true/false.

*cs*:     robo at coffee shop
*off*:    robo at Shyam's office
*mr*:    robo at mail room
*lab*:    robo at AI lab

Following are the various *states*' variables of the world, which can also be true/false:

*mw*:   Mail waiting in the mail room
*rhc*:   Robo is holding coffee
*swc*:   Shyam wants coffee
*rhm*:   Robo is holding mail

Absence of a state is represented by its negation. For example, $\overline{rhc}$ indicates that Robo is not holding coffee. Various *actions* performed by the Robo are

*mc:*    Robo moves clockwise
*mcc:*   Robo moves counterclockwise
*puc:*   Pick up coffee
*dc:*    Deliver coffee
*pum:*   Pick up Mail
*dm:*    Deliver Mail

Presence of an action symbol indicates that the action is true.

A state (of the world) may comprise many parameters or preconditions for the action to take place at that state. For example, the state,

$$\langle lab, \overline{rhc}, swc, \overline{mw}, rhm \rangle \qquad (15.1)$$

indicates that the Robo is in an AI lab, Robo has no coffee in hand, Shyam wants coffee, mail is not waiting, and Robo holds the mail. Another state,

$$\langle lab, rhc, swc, \overline{mw}, \overline{rhm} \rangle, \qquad (15.2)$$

indicates that the Robo is in the lab, Robo is holding coffee, Shyam is waiting for coffee, mail is waiting in the mail room, and Robo is not holding the mail. Table 15.3 shows the transitions for certain states. For example, in the third row, we note that after performing the action *dm* (deliver mail), in new state created, the state $\overline{rhm}$ indicates that variable "Robo is holding mail" is false.

**Table 15.3**   Some mapping: State $\times$ Action $\to$ State, for Fig. 15.2

| State | Action | Resulting state |
|---|---|---|
| $\langle lab, \overline{rhc}, swc, \overline{mw}, rhm \rangle$ | *mc* | $\langle mr, \overline{rhc}, swc, \overline{mw}, rhm \rangle$ |
| $\langle lab, \overline{rhc}, swc, \overline{mw}, rhm \rangle$ | *mcc* | $\langle off, \overline{rhc}, swc, \overline{mw}, rhm \rangle$ |
| $\langle off, \overline{rhc}, swc, \overline{mw}, rhm \rangle$ | *dm* | $\langle off, \overline{rhc}, swc, \overline{mw}, rhm \rangle$ |
| ... | ... | ... |

## 15.4  Forward Planning

One of the simplest plannings is to treat the planning as a path planning problem in the state-space graph. The nodes here are states, transitions are actions, and results of actions are also states. A forward planner searches the state-space graph from the start state for the goal state. Figure 15.3 shows the state-space graph for forward planning with the start state as $\langle cs, \overline{rhc}, swc, mw, \overline{rhm} \rangle$, with three transitions from start state, corresponding to the actions: pick up coffee (*puc*), Robo moves clockwise (*mc*), and Robo moves counterclockwise (*mcc*). If we choose the action *puc*, the next state is $\langle cs, rhc, swc, mw, \overline{rhm} \rangle$. The new state indicates that the Robo still remains facing the coffee shop; since it picked up coffee, the robot holding coffee is no more false, mail waiting remains true (unchanged). The new states as a consequence of the various actions are self-explanatory, and are similar to this description [2, 13].

Note that, being closed reasoning, in the world of actions we explicitly specify each of the variable as True or False.

The branching factor in Fig. 15.3 is 3, and the search can be done in DFS or BFS. Theoretically, since, the Robo can be at any of the four locations, and the other four parameters in a state can be true/false, there are $4 \times 2 \times 2 \times 2 \times 2 = 64$ total possible states in the world. Obviously, all of these states are not possible to reach in a graph search.

The representation above is simple and clear, but it is not suitable due to following reasons:

- there are too many states to acquire, reason, and represent,
- small change in the requirements will need a major change in the model, for example, if we need to have information about robot battery level to be added as one of the parameters, the entire structure gets modified.
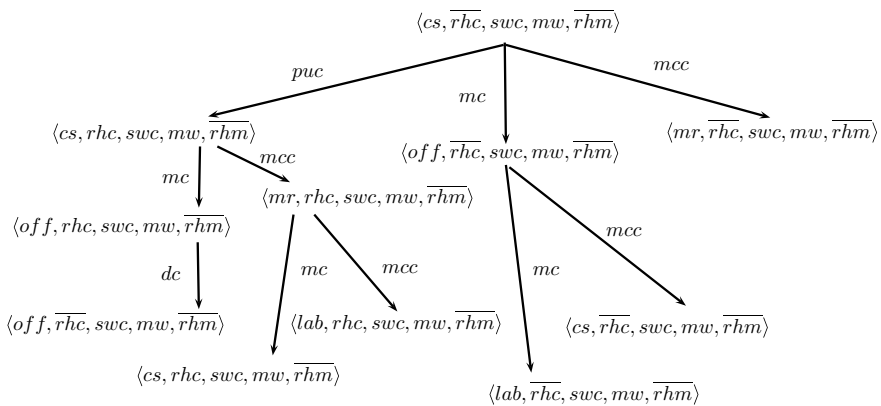


**Fig. 15.3**  State space for forward planning

The improvement in complexity is possible, and can be based on the following criteria: we note that the actions have a structure and that can be used to make actions compact. We also note that a precondition of an action should be true before the action takes place. For example, the action of the Robo to pick up the coffee (puc) requires the precondition of "Robo's location is coffee shop and Robo does not hold the coffee", which is expressed as $cs \wedge \overline{rhc}$. This means that *puc* is not available at other preconditions (or constraints) [11].

## 15.5  Partial-Order Planning

A *partial-order planning* is a search carried out by refining the partial plans through addition of actions and orderings. Alternatively, a search may proceed through performing abstract actions in the plan, by replacing fragments capable of carrying out those actions—the planning is called *hierarchical planning*. It has been now well understood that even the state-space search methods are nothing but other ways of refining partial plans. This refining is done by growing the prefix or suffix of the plan and different ways of refining a partial plan can be suitably interleaved.

The forward planner enforces a *total ordering* on actions at all the stages in the planning process. The idea of partial ordering between actions only commits to an ordering between actions when forced. A partial ordering is a "$\leq$" relation, that is, *reflexive*, *transitive*, and *antisymmetric*. It is the set of actions together with partial ordering, representing a "before relation" on actions, such that any total ordering of the actions, consistent with partial ordering, will solve the goal from the initial state. That is, $Act_0 < act_1$ means action $Act_0$ appears before the $Act_1$ in the partial order.

A partial-order plan comprises the following sets:

1. *Set of steps.* Each step maps to some operator, except the start and end step. There are no preconditions for start step and start state is its post-condition. Similarly, the final step has the goals as its preconditions and has no post-conditions.
2. *Set of orderings of steps.* Each ordering is a pair of steps, in the form of previous and next step. The start step is ordered first of all, and the finish step is ordered after all the steps.
3. *Set of causal links.* Each causal link is a pair of steps and a proposition, which is a post-condition of the first step and a precondition of the second. The first step is always ordered before the second step.

In fact, a partial planner works as follows: begin with actions *start* and *finish*, and with partial order *start* $<$ *finish*. The planner maintains an agenda set of $\langle P, A \rangle$ pairs, where, $A$ is action in plan, and $P$ is the *precondition* of $A$. First, $\langle G, finish \rangle$ is chosen, such that $G$ is the precondition for $Goal$. Then at each stage, a pair $\langle P, act_1 \rangle$ is chosen from the agenda, where $P$ is the precondition for action $act_1$. Subsequent to this, $act_0$ is chosen to achieve $P$, which is either already in the plan or it is *start* to achieve $P$, or it could be a new, that is, added to the plan. The $act_0$ must occur before

$act_1$, that adds a new causal link. Any action to delete $P$ must happen after $act_1$ or before $act_0$. If $act_0$ is a new action, its preconditions are added to the agenda, and the process continues till the agenda is empty.

## 15.6 Planning Languages

The STRIPS formulation for planning problems uses a simple and general format for specifying operators with clear semantics. It employs propositional logic as a language for describing states of the world, with a set of conditions and Boolean variables to describe states. A complete assignment maps the set of conditions to possible values. For example, truth values and a partial assignment maps a subset of conditions to values, and a complete assignment is a state. An operator comprises two *partial assignments*—first is *preconditions* or *results*, which decide the states for applying the operator. The second is *post-conditions*, that decide the next state resulting when an operator is applied in a particular start state. In addition, the implicit frame axioms indicate that the value of any condition, which is not mentioned in an operator's post-conditions, is unchanged due to the application of an operator.

A planning problem instance consists of a set of operators, an Initial state, and a Goal. The Goal may be a Boolean formula also. Generally, a solution is in the form of a partially ordered multi-set of operators, which satisfies the following condition: any total ordering consistent with the given partial order transforms the initial state assignment into a new assignment that satisfies the Goal formula. The STRIPS formulation provides the semantic foundations for many extensions, including those handling external events, multiple agents, probabilistic transformations, and variants that allow the agent to observe aspects of the current state and choose actions conditioned on observations.

The STRIPS is action-centric representation, which is based on the idea that most things are not affected by single action. It specifies *action*, *precondition*, and *effect*. For example, for the goal: "Robo to pick up coffee", we write

$$precondition : cs \wedge \overline{rhc}$$
$$effect : rhc.$$

The other features are unaffected in the above. The action of delivering coffee ($dc$) is

$$precondition : off \wedge rhc$$
$$effect : \overline{rhc} \wedge \overline{swc}.$$

Apart from the above, you need to specify the initial states and goal.

The *action* or the *rule* has two formats:

– *Causal Rule*, when a feature gets a new value, and
– *Frame Rule*, when a feature keeps its value.

We can demonstrate the change of Robo's location (see Fig. 15.2), from current $RLoc^0$ to a new location $RLoc^1$ on account of action $mcc$, $mc$, and one which is neither $mcc$ nor $mc$, as follows:

$$(RLoc^1 = cs) \leftarrow (RLoc^0 = off) \wedge (Act = mcc)$$
$$(RLoc^1 = cs) \leftarrow (RLoc^0 = mr) \wedge (Act = mc)$$
$$(RLoc^1 = cs) \leftarrow (RLoc^0 = cs) \wedge (Act \neq mcc) \wedge (Act \neq mc). \tag{15.3}$$

In the above formulas, the first two are causal rules, and the last one is the frame rule. Similarly, the state "Robo holds coffee" in the resulting state would depend on whether it was holding coffee in the previous state and its action:

$$rhc^1 \leftarrow rhc^0 \wedge Act \neq dc$$
$$rhc^1 \leftarrow Act^0 = puc \tag{15.4}$$

where the first is frame rule, and the second is the causal rule.

### 15.6.1   A General Planning Language

STRIPS is a problem-solving program implemented in LISP and has been used in the application of robotic research; it is a member of the class of problem solvers that search a space of *world models* to find a model through which the goal is achieved. We assume that, for some world model, there exists a set of operators; each of such operators transforms the world model into some other world model. Having this, the task of the problem solver is to determine a composition of operators that transform a given initial world model into one that satisfies the goal condition [3].

The primary objectives in robotics-based class of problems are rearranging physical objects, and navigation of robot—problems that require general world models, and are more complex than those used in the solution of puzzles and games. Usually, a list or simple matrix structures is adequate to represent a state of such problems. The world model for a robot problem solver must comprise a large number of facts and relations about the position of the robot, and about the positions and attributes of objects, open spaces, and boundaries. In STRIPS, the world model is represented by a set of well-formed formulas in the first-order predicate logic (FOPL) [4].

A solution is built using operators, which are the basic elements of general robotics planning language. For a robot problem, each operator corresponds to an action routine whose execution causes a robot to take that action. For example, we might have a routine to push back if the robot touches the wall, or a routine to lift and object, or to grip an object, and so on, there are a large number of routines that correspond to actions of a robot.

## 15.6.2 The Operation of STRIPS

The problem space for STRIPS language comprises the initial world model, a set of available operators with their effects on the world model, and a goal statement. The world model is represented by a set of well-formed formulas (wffs) of FOPL, e.g., to describe a world model in which the robot is at location $a$ and boxes $B$, $C$ are at locations $b$, $c$ respectively, we would use the following wffs to express this knowledge [4]:

$$ATR(a)$$
$$AT(B, b)$$
$$AT(C, c).$$

We might also use the wffs to express the general rule that an object $u$ at place $x$ is not in a place $y$, i.e., an object cannot exists at two places.

$$(\forall u \, \forall x \, \forall y)\{[AT(u, x) \wedge (x \neq y)] \Rightarrow \neg AT(u, y)\}. \tag{15.5}$$

We can represent a complex world model using the first-order predicate logic, and can use standard theorem-proving programs to answer questions about the model. The operators are grouped into families of operators, called *schema*. For example, an operator *goto* for moving the robot from one point $m$ on the floor to other point $n$ is a schema. For this, distinct operators (one for each pair of points) are grouped into a family of *goto* operators, and $goto(m, n)$ represents a move from the initial position $m$ to the final position $n$. The members of *goto* schema are $goto(m, a_1)$, $goto(a_1, a_2)$, ..., $goto(a_k, n)$. In STRIPS, specific constants will already have been chosen for the operator parameters when an operator is applied to a world model.

First of all, it is necessary to determine whether or not there is an instance of an operator schema applicable to the current world model. It is required that an instance of the corresponding wffs (well-formed formulas) schema exists, and it logically follows from the model. Each operator schema is defined by a description having two parts: conditions under which the operator can be applied, and the effects of application of that operator. The precondition for an operator schema is represented as a wff. The effects of application of an operator are defined by a list of wffs that must be added to the model, and a list of wffs that are no longer true, hence must be deleted. As an example, consider the question of applying instances of the operator subschema $goto(m, b)$ to a world model containing the wff $ATR(a)$. If the precondition wff schema of $goto(m, n)$ is $ATR(m)$, then we note that the instance $ATR(a)$ can be proved from the world model. Thus, an applicable instance of $goto(m, b)$ is $goto(a, b)$.

It is important to understand the difference between the parameters in wff schema, and existentially and universally quantified variables. These variables are used in FOPL formula in theorem-proving programs (e.g., Resolution theorem) that would handle wff schema. For example, in resolution, $\forall x \forall y \, goto(x, y)$ will have substitution

$\{a/x, b/y\}$, to make it a clause, whereas in STRIPS there is a chain of pairs of $(x, y)$ to implement $goto(x, y)$, hence it will require some modifications.

A goal statement is also in the form of a wff, e.g., a task of moving the boxes $B$ and $C$ to locations $d, e$, respectively, might be expressed a goal as

$$AT(B, d) \wedge AT(C, e).$$

In summary, the problem in state space for STRIPS comprises three entities:

1. *Initial world model.* It is a set of wffs to describe the present state of the world.
2. *Set of operators.* The operators and their description in the form of effects and preconditions as wffs.
3. *Goal.* It is a condition in the form of a wff.

The problem is taken as solved when a world model that satisfies the goal wff is deduced using the initial world model with the application of operators.

### 15.6.3   Search Strategy

A very simple problem-solving strategy is to apply all the applicable operators to the initial world model and create a set of successor models. Then, continue to apply all applicable operators to these successors and to their descendants generated through DFS or BFS search, until a model is deduced having the goal formula as a theorem. However, it is quite likely that in most real-word problems modeled using this approach, the number of operators applicable to any given world model will be too large. Hence, such a simple system would generate trees that are large in number as well as size. Hence, such world models would be impractical.

An alternative strategy is to extract differences between current world model and the goal model, with the objective as to how to move from one to another with minimum changes. Then find out the operators that may reduce the differences between these models. Once the relevant operators are found out, we try to solve the subproblem that produces a world model to which it is applicable. If such a model is found, then we apply the relevant operators and consider the original goal in the resulting model.

## 15.7   Planning with Propositional Logic

Consider the graph/tree shown in Fig. 15.4, with $s_0$ as start state, and $a_0 \ldots a_{n-1}$ as actions representing a path leading to the goal state $s_n$. If $s_0, a_0 \ldots a_{n-1}, s_n$ are considered as propositional expressions, then Eq. [10],

**Fig. 15.4** Propositional
planning
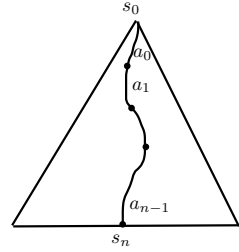


$$p = s_0 \wedge (a_0 \wedge a_1 \wedge \cdots \wedge a_{n-1}) \wedge s_n \qquad (15.6)$$

is a propositional expression representing the path to goal state from start state. If there is only one goal, then $p$ is the only path to goal, and all other propositions $p'$ are false. Equation 15.6 can always be transformed into a *CNF* (conjunctive normal form) or *SAT* expression. Thus, planning through propositional logic is to find a satisfiability expression, comprising *start state*, *path*, and *goal state*, i.e., logical sentence equal to

$$initial\ state \wedge all\ proposition\ action\ descriptions \wedge goal. \qquad (15.7)$$

In other words, a model that satisfies a sentence will assign true to all actions that are part of a correct plan. If the planning problem is unsolvable then there is no sentence that is satisfiable. The following example demonstrates planning with propositional logic [12].

**Example 15.3** Flight Planning.

*Initial Plan*: Let us assume that initially, the plane $p_1$ is at *DEL* (Delhi) and $p_2$ at *CAL* (Calcutta). The actions should correspond to flying these planes so that the goal: "$p_1$ at *CAL*, and $p_2$ at *DEL*" is satisfied.

The initial state (0) is represented by

$$At(p_1, DEL)^0 \wedge At(p_2, CAL)^0.$$

Since the propositional logic has no *closed world*, it is also necessary to show that initially the planes $p_1, p_2$ are not *CAL, DEL*, respectively, i.e.,

$$\neg At(p_1, CAL)^0 \wedge \neg At(p_2, DEL)^0.$$

The initial conditions correspond to start state $s_0$.

The goal also needs to be specified with a particular time step. Since it is not known how many time steps it would consume to iterate the action, a worst case time limit needs to be specified so that either the goal is reached within that time limit, else the solution is terminated as failure.

We test the following assertion for a goal at time $T = 0$ (at start),

$$At(p_1, CAL)^0 \wedge At(p_2, DEL)^0.$$

If that fails, apply certain actions and again try it at time $T = 1$, and so on, up to time $T = T_{max}$. This is shown in Algorithm 15.1, where for every iteration, for the next value of $T$ (time), the problem is translated to *SAT* (satisfiability) problem using the procedure *translate-to-SAT*. This results in *CNF* expression and a mapping of the solution to the problem. The *SAT-Solver* procedure returns the assignment for the above *CNF* expression. If the assignment is satisfying the solution (i.e., not null), the solution is extracted for the present mapping, and assignment is returned as result. If this does not happen, the procedure is iterated as per the $T_{max}$ iteration time [8].

---

**Algorithm 15.1** Planning with Propositional Logic

---
1: *INPUT* : *A planning Problem*;
2:                   $T_{max}$: an upper limit for plan length
3: **for** $T = 0$ to $T_{max}$ **do**
4:     *cnf , mapping* ← *translate-to-SAT* (*problem*, *T*)
5:     *assignment* ← *SAT-Solver*(*cnf*)
6:     **if** Assignment is not null **then**
7:         Return *Extract-solution*(*asignment*, *mapping*)
8:     **end if**
9: **end for**
10: Return Failure
11: End

---

### 15.7.1   Encoding Action Descriptions

We have a propositional symbol for each occurrence. For the plane $p_1$ to be at *CAL*, there is a proposition:

$$At(p_1, CAL) \Leftrightarrow (At(p_1, CAL)^0 \wedge \neg fly(p_1, CAL, DEL))$$
$$\vee (At(p_1, DEL)^0 \wedge fly(p_1, DEL, CAL)^0) \quad (15.8)$$

There ought to be a plan that tries to achieve the goal at $T = 1$. Now suppose CNF is

$$Initial\ state \wedge successor\ state\ axioms \wedge goal^1,$$

that is, goal is true at $T = 1$, we check and verify that

$$fly(p_1, DEL, CAL)^0 \wedge fly(p_2, CAL, DEL)^0$$

is the model, and other assignments are false.

## *15.7.2 Analysis*

For the proposition *fly* having general form as $fly(p, o_1, o_2)$, with time $T$-steps, the number of propositions represented by $|p|$, and objects ($o_1, o_2$, etc.) as $O$, the complexity expression is given by,

$$fly(p, o_1, o_2) \Rightarrow T \times |p| \times |o_1| \times |o_2|$$
$$\Rightarrow T \times |p| \times |O|^2$$
$$\Rightarrow T \times |Act| \times |O|^p$$

where $T$ is the number of time steps, $p$ is *arity* of function (here it is 2), $O$ is the number of objects $o_1, o_2$, etc. We note that the complexity is exponential. The term $|Act|$ is for the action, like *fly*, i.e., how many predicates are there in total [8].

## 15.8 Planning Graphs

The planning graphs give better heuristics and consist of a sequence of levels, for *time steps* in plan. Each level has literals (constant values) which have become true because of the previous action, and each level has preconditions for the next action. The planning graphs represent the *actions* as well as *inactions*. The following example demonstrates the application of planning graphs [5, 12].

**Example 15.4** Problem of the solution to "have a Pizza and eat Pizza".

$$init(have(pizza))$$
$$Goal(have(pizza) \wedge eaten(pizza))$$
$$Action(eat(pizza)$$
$$Precond : have(pizza)$$
$$Effect : \neg have(pizza) \wedge eaten(pizza))$$
$$Action(cook(pizza)$$
$$Precond : \neg have(pizza)$$
$$Effect : have(pizza)).$$

The planning graph for these actions is shown in Fig. 15.5. The box action in the figure indicates the *mutual exclusions* of actions.

In the planning graph, all the actions $A_i$ at level $i$ contain all actions that are applicable at state $S_i$, along with constraints saying which part of the actions cannot be executed. Every state at level $S_i$ contains all literals that could result from any choice of actions at $A_{i-1}$, along with constraint saying which part of actions cannot be executed.
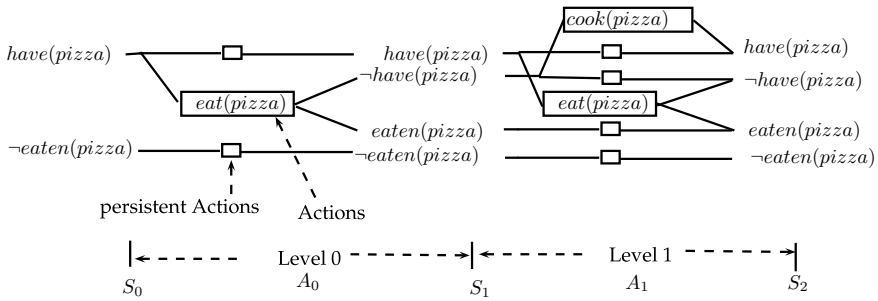
**Fig. 15.5** A planning graph for "To have pizza and eat pizza"

We note that graph plan provides lesser complexity because it does not require choosing about all actions. It just records the impossibility of certain choices using *mutex*, i.e., through mutual exclusion links. For example, when ¬*have*(*pizza*) is chosen, the action *have*(*pizza*) is excluded. This results in the complexity as a function of low polynomial actions and literals [11].

## 15.9   Hierarchical Task Network Planning

In the hierarchical task planning, each level of hierarchy is decomposed into smaller levels. It is common for areas like military mission, administration, program development, where a task is reduced to small number of activities at the next level, so that the computational effort of arranging those activities is low. This results in reduction in complexity to linear time from the original exponential [13].

Consider an example of "building a house", where the task of house building can be decomposed to acquiring land, preparation of design map, obtaining the NOC (no objection certificate) from municipal corporation, arranging for a house loan, hiring a builder, paying the builder, and so on, as shown in Fig. 15.6.
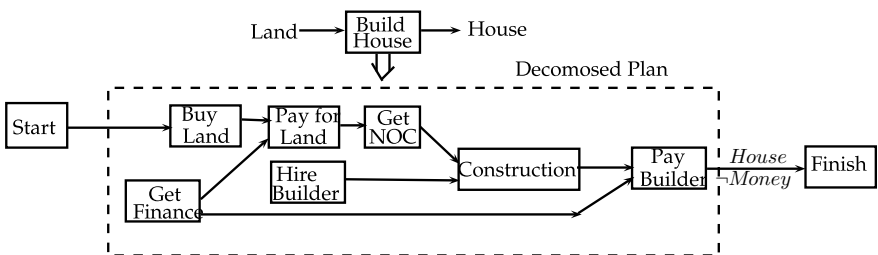


**Fig. 15.6** Decomposition of task of house building

We understand that some of these activities can be done in parallel, but not all. Thus, there is a partial-order relation between those actions. The decomposition can be expressed as *decompose*(*a, d*), where action *a* is decomposed into a partial-order plan *d*. Various activities for building a house as per the plan in Fig. 15.6 can be formally described as follows.

$$Action(Buildhouse, \; Precond : BuyLand,$$
$$Effect : House) \tag{15.9}$$

$$Action(Buyland, \; Precond : Money,$$
$$Effect : Land \wedge \neg Money) \tag{15.10}$$

$$Action(Getfiance, \; Precond : Goodcredit,$$
$$Effect : Money \wedge Mortgage) \tag{15.11}$$

$$Action(Hirebuilder, \; Precondition : Nil,$$
$$Effect : contract) \tag{15.12}$$

$$Action(Construction, \; Precond : NOC \wedge Builderhired,$$
$$Effect : Housebuilt \wedge \neg NOC) \tag{15.13}$$

$$Action(Paybuilder, \; Precond : Money \wedge Housebuilt,$$
$$Effect : \neg Money \wedge house \wedge \neg Contract) \tag{15.14}$$

$$Decompose(steps : \{A_1 : Get \; NOC, \; A_2 : Hirebuilder,$$
$$A_3 : construction, A_4 : Paybuilder\} \tag{15.15}$$

$$Orderings : \{start \prec A_1 \prec A_3 \prec A_4 \prec Finish;$$
$$Start \prec A_2 \prec A_3\}. \tag{15.16}$$

Here $A_i \prec A_j$ indicates that activity $A_i$ precedes the activity $A_j$.

The linking of states and activities through activities/resources for the house building plan can be expressed as follows:

$$Links : \{start \; \underline{Land} \; A_1, \; Start \; \underrightarrow{Money} \; A_4, A_1 \; \underrightarrow{NOC} \; A_3,$$
$$A_2 \; \underrightarrow{Contract} \; A_3, \; A_3 \; \underrightarrow{Housebuilt} \; A_4, A_4 \; \underrightarrow{House} \; Finish,$$
$$A_4 \; \underrightarrow{\neg Money} \; Finish.\} \tag{15.17}$$

## 15.10   Multiagent Planning Systems

There are a number of good reasons for having multiple agents creating plans:

1. The agents may represent real-life entities, which may require the privacy and autonomy also to be maintained.
2. Changing of an existing distributed system to a multiagent system is less costly than to a centralized system .
3. Creating and maintaining plans locally allows for more efficient reaction, especially when communication is limited, and
4. Dividing a planning problem into smaller pieces and solving those in parallel turns out to be many times more efficient. This is true particularly when the individual planning problems are not tightly coupled.

   In spite of many benefits of multiagent systems, following are the challenges in developing multiagents planning:

1. How to put additional constraints upon the agents before planning, such that their resulting plans can still be coordinated?
2. How to efficiently construct plans in a distributed environment?
3. How to make collaborative decisions when there are multiple options, and agents have their own preferences for these options?
4. In what condition should a planning agent ask for more specific information to the user?
5. How to find out the magnitude of privacy lost in the process of coordinating plans?

**Definition 15.2** Multiagent planning problem.

In general, a multiagent planning problem is a problem of planning by and for a group of agents. Except for centralized multiagent planning problems, each agent in such a problem has in fact a private, individual planning problem. A typical individual planning problem of an agent includes a set of operations with some costs attached, and pre- and post-conditions that it can perform, and a set of goals (with reward values), and the current (initial) state of this agent. The following statement captures the concept of multiagent planning:

$$Multiagent\ planning = planning + coordination. \qquad (15.18)$$

□

   The solution to a multiagent planning problem is a plan: a partially ordered sequence of actions that, when executed successfully, results in a set of achieved goals for some of the agents. Most techniques can deal with problems where the actions and goals of the agents are 1. only weakly dependent upon each other, 2. agents are cooperative, and 3. communication is reliable. However, in general a multiagent planning problem may encounter a lot variety of situations along these three axes. Some characteristics are described below.

1. From independent to strongly related.

   a. *Independent:* There are no shared resources, and no dependencies. For example, a robot is lifting a box.
   b. *Strongly related:* They have joint actions, and the resources are shared. For example, car assembly is a case of strongly related.

2. From *cooperative* to *self-interest* agents: The participating agents interested in optimizing their own utility is the case of self-interesting agents, while the agents involved in supply chain management is an example of cooperative.

3. From *no communication* to *reliable communication*: In hostile environments agents may not or cannot communicate during execution. This may require agents to be equipped in advance, with or without some initial communication before the execution starts. The robots rescuing people in disaster scenarios, or working on a inter-planetary exploration mission, are examples of the first type, while agents working in a supply chain management is an example of the second category.

## 15.11   Multiagent Planning Techniques

Multiagent planning techniques cover quite a range of solutions to different phases of the problem. In general, the following *phases* can be distinguished in task sharing:

1. Allocate goals to agents.
2. Refine goals into subtasks.
3. Schedule subtasks by adding resource allocation including the agents, and timing constraints.
4. Communicate planning choices (i.e., prior steps) to recognize and resolve conflicts.
5. Execute the plans.

Planning is a combination of phases 2 and 3 in the above, which are often interleaved. Any of these steps could be performed by one agent or some subset. Not all the above phases of general multiagent planning process need to be considered in every multiagent planning problem. For example, there is no need for phase 1 if there are no common or global goals for the multiagents. Some multiagents system may combine different phases, for example, while constructing their plans agents may coordinate with each other due to the combination of phases 2, 3, and 4. Alternatively, robots may postpone coordination until the execution phase (i.e., combining phase 4, 5). This may result, for example, when they unexpectedly encounter each other while following their planned routes.

It is possible in general, to interleave any combination of five phases listed above, depending on the nature of the planning problem, resulting in a wide variety of possible problem sub-classes. In the following we present the phases in more detail.

### 15.11.1   Goal and Task Allocation

Centralized methods often take care of the assignment of goals and tasks to agents during planning. There are, however, many other methods to assign tasks in a more distributed way, giving the agents a higher degree of autonomy and privacy. For example, complex task allocation protocols may be used, or auctions and market simulations can also be used.

One way to assign the tasks to agents is through the method of auction, which is a way to assign a task to that agent who attaches (or bids) the highest value or the lowest cost of performing it, called *private value*. A protocol called *Vickrey* auction is a frequent example in multiagent systems, where each agent makes a closed bid, and the task is assigned to the highest bidder, but not on the price of that bidder, but for the price of the second highest bidder! The Vickery protocol has the good property due to which the bidding agents will simply bid their true private values, with no need for additional reasoning about it is worth for the others.

Economics and market simulations can also be the basis for allocation of resources among agents. For example, it is shown how costs can be turned into a coordination device. These methods are useful for task assignment (phase 2), and also for the coordination of agents after plan construction (phase 5).

In concerning the value-oriented environments, these game-theoretical approaches become more important where agents reason about cost of their decision-making (or communication).

### 15.11.2   Goal and Task Refinement

Task assignment can be done through a single agent, using Hierarchical Task Networks or nonlinear planning. More than one planner with more sophisticated models of temporal extent can be introduced, along with centralizing as well as combining the phases 2 through 4.

### 15.11.3   Decentralized Planning

Instead of one agent planning for rest of the agents, the second and third phases can be implemented through local planning by each of the agents. In principle, any planning technique can be used in this condition, and different agents may even use different techniques. Some of the approaches join the individual plannings (phases 2 and 3), along with the coordination of the plans (phase 4).

A distributed version of a planner can be used to integrate phases 1 through 4, to plan for a single agent in parallel.

In one setting, each agent has a partial knowledge of the plans of other agents using some specialized plan representation techniques. In such kind of environment, coordination is achieved as follows: If an agent *A* informs the other agent *B* about part of its own plan, then agent *B* merges this information into its own partial global plan. The agent *B* can then try to improve the global plan by, for example, eliminating redundancy in the plan, and this improved plan is shown to the other agents, who might reject, or accept, or modify it. This process may run concurrently with the execution of the (first part of the) local plan.

### 15.11.4   Coordination After Planning

One of the tasks is the planning of coordination after plans are constructed on individual basis (phase 4), called *plan merging* . Its objective is construction of a joint plan for a set of agents, given the individual plans of each of the participating agents. In coordination planning, every pair of agents helps each other by changing the state of the world such that the conditions of the other agent become satisfied. In this process, changing the state of the world may be helpful to these two, but may also interfere with the correct conditions of the remaining $n - 2$ agents, assuming a system of $n$ agents.

To specify the constraints on plans, one approach is to use propositional temporal logic, which will ensure that only feasible states of the environment are reached. A theorem prover algorithm generates a sequences of communication actions, on receiving these constraints. In fact, these communication actions implement semaphores that guarantee that no event will fail. For resolving conflicts, restrictions are required to be introduced on individual plans in phase 3, which will also ensure efficient merging.

A different approach to plan merging uses the distributed approach to improve social welfare, based on the sum of the benefits of all agents. This approach uses a process of group constraint aggregation, where agents construct an improved global plan by voting for joint actions, in an incremental way. The agents even propose algorithms to deal with insincere agents, and to interleave planning, coordination, and execution [11].

## 15.12   Summary

When an agent is interested in controlling the evolution of its environment, there is a need for planning. Thinking as an algorithm, a planning problem has an input in the form of possible courses of actions, a predictive model for the required dynamics, and a measure for performance to evaluate the courses of actions. The output or solution of this algorithm is one or more courses of actions that satisfy the specified requirements for performance.

The agents are classified as *reactive* and *planning agents*.

The classical planning problem is to plan reaching the goal state(s) from the initial state, for a given set of actions. A majority of research in planning is toward planning in the environments that are *dynamic*, *stochastic*, and *partially observable*. To carry out this, the existing classical planning techniques are extended to allow interleaving of planning and scheduling.

Automated planning techniques are being applied in many domains, that include robotics, process planning, web-based information gathering, autonomous agents, and spacecraft mission controls. In automated planning, a solution to a problem can be described in terms of a sequence of steps that transforms some initial description of the problem state, for example, the initial configuration of a puzzle, into a description satisfying a specified goal criterion. For a simple automated planning problem, called *classical planning problem*, the following assumptions are made:

1. the actions are deterministic,
2. the world is fully observable, and
3. the closed world assumption.

Agents are classified according to the techniques they employ in decision-making:

1. Reactive agents, and
2. Planning agents.

Forward planning, one of the simplest planning, is path planning where nodes are states, transitions are actions, and results of actions are also states. A forward planner searches the state-space graph from start state for goal state. One problem with forward planner is its time complexity, which is exponential.

A *partial-order plan* consists of (1) a set of steps, each mapping to an operator, (2) a set of orderings, and (3) a set of causal links.

STRIPS is an action-centric language, based on the idea that most things are not affected by a single action. It specifies *action*, *precondition*, and *effect*. The problem space for STRIPS is defined by an initial world model, a set of operators, and a goal condition state.

Planning through *propositional logic* is to find a satisfiability expression, which can be given as

$$\text{initial state} \wedge \text{all proposition action descriptions} \wedge \text{goal}.$$

Other approach for automated planning is *planning graphs*, which consists of a sequence of levels, for *time steps* in plan, representing *actions* as well as *inactions*.

In the areas such as military mission, administration, program development, where a task is reduced to a small number of activities at the next level, such that the computational effort of arranging those activities is low, the *hierarchical task planning* using networks is preferred.

Splitting a planning problem into smaller problems and solving these in parallel turns out to be more efficient, thus motivating the use of multiple agents for creating plans. Multiagent planning covers the following *phases*:

1. Allocate goals.
2. Refine goals into subtasks.
3. Add resource including the agents.
4. Communicate planning choices.
5. Execute the plans.

## Exercises

1. Consider the standard Towers of Hanoi problem with 3 pegs and 4 number of disks
   ($d_1, d_2, d_3, d_4$, with $d_1$ at the top). The disks are to be transferred from start-peg
   to end-peg, using intermediate peg, one at a time such that at no time larger disk
   comes over the smaller. The disk $d_1$ is the smallest and $d_4$ is the largest. Make
   use of only 3-predicates: unary predicate: *clear*, and binary predicates: *on* and
   *smaller*, and only one action: *puton*$(x, y)$ needs to be used.
   Write the domain of the problem, and make use of *forward planning* to plan the
   solution to move all the 4 disks from start-peg to end-peg.
2. Given the 3-SAT problem:

$$(\neg p_1 \vee p_2 \vee p_3) \wedge (p_1 \vee \neg p_2 \vee p_3) \wedge (p_1 \vee p_2 \vee \neg p_3),$$

   solve it using *forward planning*. (Hint: you need to assume some *operators* (i.e.,
   actions) to assign the values to variables $p_1 \ldots p_3$.)
3. Given the Table $T$, and blocks $A, B, C, D$, having different positions on the table,
   apply STRIPS to plan the solution of the following problem:
   Initial state **I** as

   $$clear(A), clear(B), clear(C), clear(D),$$
   $$on(A, T), on(B, T), on(C, T), on(D, T),$$

   i.e., the blocks $A \ldots D$ are on the table, and their tops are clear.
   Final State **F** :
   $on(A, B), on(B, C), on(C, D), on(D, T), clear(A)$.
   Use the action *puton*$(X, Y)$, $x \neq y$, where $X$ is a block $A \ldots D$ and $Y$ is either
   table $T$ or block $A \ldots D$. Give the forward planning to reach state **G** starting with
   state **I**.
4. Use STRIPS for planning of the following problem: You are at home, and you
   have money, and you are required to buy milk. Assume the necessary start and
   goal states, actions, preconditions, and results for this planning job.
5. Give the STRIPS representations to actions: pick up mail and deliver mail (ref.
   Fig. 15.2).
6. Suppose the robot (in Fig. 15.2) cannot carry both coffee and mail at the same
   time. Make use of some constraints to provide the planning for this situation.

Assume that the robot can carry a box in which it can place objects, so that it can carry the box and the box can hold the mail and coffee.

7. Modify the problem in Fig. 15.2, so that the robot has the work of cleaning the four rooms (mail room, office, coffee shop, lab). Assume that it will clean the room only when the room is unclean, and will not consume more than one rotation *mcc* or *mc* to reach any of these rooms.

8. Using the method of hierarchical task network planning, provide the automated planning for the following problems:

   a. Shopping grocery items from market.
   b. Deliver a lecture of AI.
   c. Robot path planning to cover the diagonal in a room.

9. Assume that you have three operators:

   $f_1$ : Precondition: $a$; effect: $\neg a \wedge b$
   $f_2$ : Precondition: $a \wedge c$; effect: $\neg a \wedge b \wedge \neg c$
   $f_3$ : Precondition: $b \wedge c$; effect: $\neg c \wedge d$

   Show the first three layers (proposition, action, and proposition) of the graph plan when the initial state is $a \wedge c$ ($a$ and $c$ both are true). Include the mutual exclusions and justify each of them.

# References

1. Bborrajo D et al (2015) Progress in case-based planning. ACM Comput Surv 47(35):1–35
2. Bonet B, Geffner H (2001) Planning as heuristic search. Artifi Intell 129:5–33
3. Dean T (1996) Automated planning. ACM Comput Surv 28(1):85–88
4. Fikes RE, Nilsson NJ (1971) STRIPS: a new approach to the application of theorem proving to problem solving. Artifi Intell 2:189–208
5. Jonsson P et al (2000) Towards efficient universal planning: a randomized approach. Artifi Intell 117:1–29
6. Kaelbling LP et al (1998) Planning and acting in partially observable stochastic domains. Artifi Intell 101:99–134
7. Kambhampati S (1995) AI planning: a prospectus on theory and applications. ACM Comput Surv 27(3):334–336
8. Kautz H, Selman B (1992) Planning as satisfiability. In: Proceedings of the 10th European conference on artificial intelligence (ECAI 92), Vienna, Austria
9. Leckie C, Zukerman I (1998) Inductive learning of search control rules for planning. Artifi Intell 101:63–98
10. Melis E, Siekmann J (1999) Knowledge-based proof planning. Artifi Intell 115:65–105
11. Nareyek A (2005) Constraints and AI planning. Intell Syst 03(04):2005
12. Oh SC et al (2005) A comparative illustration of AI planning-based web services composition. ACM SIGecom Exchanges 5(5):1–10
13. Russell SJ, Norvig P (2005) Artificial intelligence—a modern approach, 2nd edn, Pearson

# Chapter 16
# Intelligent Agents

**Abstract** The intelligent agents are being viewed as new theoretical models of computation that more closely reflects current computing reality, aimed as new generation models for complex and distributed systems. An agent system can work as a single agent, or as a multiagent system. The intelligent agents have many applications—they are used in software engineering, in buying and selling—like online sales, bids, trading; the agents are also modeled for decision-making—with preferences and criteria for making decisions. This chapter also presents the classification of agents, agent system architecture, how the agents should coordinate among themselves, and the formation of a coalition between agents. The multiagents communicate with each other using agents' communication languages which are oriented towards performing actions. Other categories of agents are mobile agents—programs which can be moved to any far off place, and can communicate with the environment. The chapter ends with chapter summary, and the set of exercises.

**Keywords** Intelligent agent · Mobile agent · Multiagents · Agents' coordination · Cooperative agents · Agents' coalition · Software agents

## 16.1 Introduction

An *artificial agent* or *intelligent agent* is a recent term in computer science, and specifically in artificial intelligence. There is a number of definitions of agents. The agents are viewed as a new theoretical model of computation, that reflects current computing reality in a better (tangible) way than the existing model of Turing Machine. They are being projected as a next-generation model to engineer the complex and distributed systems.

Among many characterizations of agents, the following definition is most common: An agent is an encapsulated computer system, which is situated in some environment and it is flexible and capable of autonomous action in that environment in order to meet its desired goals. There are associated number of questions about this definition that require further explanation, which becomes somewhat clear through the extended definition of agents, in the following:

1. Agents are entities for problem-solving for clearly identifiable problems with well-defined boundaries and interfaces;
2. They receive inputs related to the state of their environment through sensors, when embedded in an environment, and act on the environment through effectors;
3. An agent is designed to fulfill a specific requirement, and it has a particular goal to be achieved;
4. They have control over both their own behavior and over the internal state, a property called *autonomous*;
5. The agents have flexible problem-solving behavior. They are both *reactive* (able to respond in time to the changes occurring in their environment) and *proactive* (able to act in anticipation of future goals).

Agents are also being used as a framework to bring together various AI's sub-areas to design and build intelligent systems. In spite of the intense interest of research community and progress made in the area of agents, a number of fundamental questions about the nature and the use of the agent-oriented approach remain unanswered, which are as follows:

- What are the fundamental concepts and notions of agent-based computing?
- What makes the agent-based approach a natural and powerful computational model?
- What are the implications of agent-based computing, in the wider perspectives, for AI and computer science in general?

**Learning Outcomes of this Chapter**

1. List the defining characteristics of an intelligent agent. [Familiarity]
2. Characterize and contrast the standard agent architectures. [Assessment]
3. Describe the applications of agent theory to domains such as software agents, personal assistants, and believable agents. [Familiarity]
4. Describe the primary paradigms used by learning agents. [Familiarity]
5. Demonstrate using appropriate examples how multiagent systems support agent interaction. [Usage]
6. Syntactic structure of agent languages. [Familiarity]

## 16.2   Classification of Agents

Although there is no universally accepted definition of an agent, however, as per the most commonly used definitions, an agent is a proactive software component that interacts with its environment, as well as it interacts with other agents on behalf of its user, and reacts to the changes in its environment. A component is called agent if it exhibits several of the properties given below.

Autonomous

It is the property of an agent, as per which it proactively initiates the activities as per its goal. An agent has its own thread of control, can act on behalf of its user, and without necessarily depending on the messages from other agents.

Mobile

An agent can move itself from one execution context to another. For this activity, it can move its code, and carry on executing from the current point onward, or it can start afresh. Other modes of execution can be, serialization of its code and state, such that it may continue its execution in a new context, and at the same time, it may retain the same old state and can continue to work.

Adaptable

An agent is adaptable to a new environment; its behavior can change after its deployment through its own learning, downloading new capabilities, and through user customization.

Knowledgeable

A software agent has reasoning capability, due to which it can reason about the acquired information, about the knowledge of other agents, its user, and about its goals.

Collaborative

Some agents are called collaborative agents, which can communicate with other agents and work in a cooperative manner. This collaboration can be formed either in a static manner or a dynamic manner. The collection of such agents is called multiagent system.

Persistent

The infrastructure used by the agents allows them to retain their knowledge and states over an extended period of time. The agents have property of robustness, i.e., they work correctly on the face of some failures at run time.

Many characteristics of Intelligent agents are result of capabilities like *adaptability*, *cooperation*, and *proactivity*. see Fig. 16.1 shows agent taxonomy. The circles correspond to general agent capabilities, and intersection corresponds to the agents having either two or three capabilities. For example, an agent having the capability of proactive and cooperative is called a collaborative agent.

Depending on the functions performed, agents can be classified in one of several major categories.

Adaptive agents

They can learn from their previous experience, and can change how they should behave in a given situation, and can also behave differently in given situations.

**Fig. 16.1** Agent taxonomy
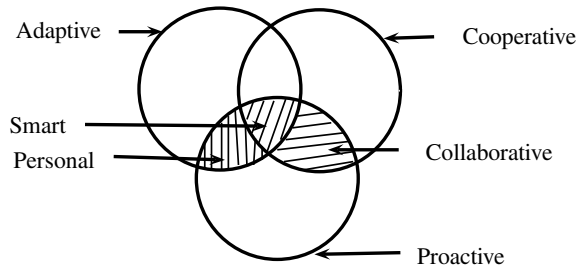


### Cooperative agents

They can communicate with other agents, and their action can be according to the results of the communication performed.

### Proactive agents

These agents can initiate proactive actions, i.e., without any prompting from the user or other agents.

### Personal agents

They are proactive and can interact directly with a user. While interacting with the user, they present some personality or character, can do monitoring and adapting to the user's activities, can learn the user's style and preferences. They can automate or simplify certain rote tasks. Many software tool-kits, e.g., Microsoft Agent, offer software services set that support the presentation of software agents as interactive personalities and includes natural language and animation capabilities.

### Collaborative agents

These agents are proactive and cooperate with other agents. They communicate and interact in groups, many times on behalf of a number of users or organizations, or services. Multiple agents exchange messages to negotiate or share information. Some of their applications are: online auctions, planning, negotiation, logistics, supply-chain management, and telecommunication services.

### Smart agents

The smart agents exhibit a combination of all capabilities, i.e., they are adaptive, cooperate with other agents, and are proactive.

### Mobile agents

These agents are sent to remote sites to collect the information, and forward it to the central or any other location. Before sending the results to a specified location, these agents can aggregate and analyze data or perform some local control. They are typically implemented in any of the following languages: Java, Java-based component technologies, VBScript, Perl, TCL, or Python. The data-intensive processing is

usually performed at the source, as this avoids the shipment of bandwidth consuming raw data. Examples of such applications are network management agents, Internet spiders, and NASA's mobile agents for human planetary exploration, etc.

## 16.3 Multiagent Systems

The multiagent systems are used by organizations or people with different /conflicting goals, and having proprietary information. In such systems, multiagent system is required to handle their interactions. As an example, in a manufacturing scenario where company *A* produces launching-pad of missiles but subcontracts to company *B* to produce the missiles. To build the whole system comprising missiles launcher and missile, the internals of both companies must be modeled. However, none of these companies are ready to share the details with the other company. Possibly, the two companies may reach to some agreement, or when not ready to share any details due to protocols imposed by the government, multiagent system (MAS) can be created, with one agent for each company, that represents the goals and interest of each company [10].

As another example, consider a teaching time-table system for a college. This domain requires, different agents to represent the interest of different people in the college. Faculty wants their classes should be evenly distributed throughout the week, with possibly all classes in as few rooms as possible, management wants that all the resources be used fully, students want that no more than two/three theory classes be held each day. Similarly, the technicians will have their own requirements. In such a scenario, a multiagent system, where different constraints are handled by agents separately, can create time table static/dynamic to best meet all the constraints.

The multiagent system creates parallelism by assigning different tasks to different agents, hence making overall a fast response system. In addition, many agent's systems will have redundant agents, this helps in building robustness in the system. This is possible because, the control and responsibilities are shared among the agents, hence the system can tolerate failures of some of the agents, and still working correctly and efficiently. The areas of applications that requires graceful degradation at the time of failure, instead of sudden failure, are suitable domains where multiagent systems' use is welcomed. However, if single entity or processor or agent controls everything, the then entire system may crash if there is a single failure.

The multiagent systems have the benefit of scalability. Because they are inherently modular, it is easier to add new systems to a multiagent system than to add new capabilities to monolithic systems. Due to the flexibility available, it is easier to program a multiagent system.

**Fig. 16.2** A general
single-agent framework
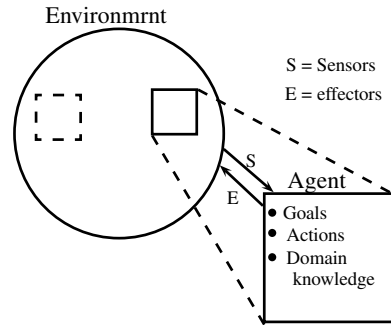


## 16.3.1  Single-Agent Framework

Though it might appear that a single-agent system might be simpler in concern
to dealing with a fixed complex task, however, the opposite is often true. In fact,
when control is distributed among number of agents, an individual agent can be
simpler. A general agent is a single-agent system, together with the environment,
and the interaction between agent and environment. An agent is itself part of the
environment, but generally, the agents are considered to have extra-environmental
components, which are independent entities, having their own goals, knowledge, and
actions. In a single-agent system, no other entities are recognized by the system (see
Fig. 16.2).

## 16.3.2  Multiagent Framework

Following are the taxonomies of multiagent systems:

1. Agent granularity, which can be course or fine.
2. Heterogeneity of agent's knowledge, can be redundant or specialized.
3. Methods used for distributing control can be: benevolent or competitive, team or
   hierarchical, static or shifting roles.
4. The agents can communicate among other agents, in blackboard or through mes-
   sages, it can be low-level or high-level communication.

In a multiagent system, there are several agents which are capable of modeling
each other's goals and actions. In a general multiagent system, there may be direct
interactions among the agents. The inter-agent communication is viewed separate
from communication with the environment. A major difference with single-agent
system is that, in multiagent systems, the environment dynamics can be determined
by the other agents also, which can affect the environment in an unpredictable way.
Thus, all multiagent systems can be treated as having dynamic environments.

**Fig. 16.3** A fully general multiagent framework
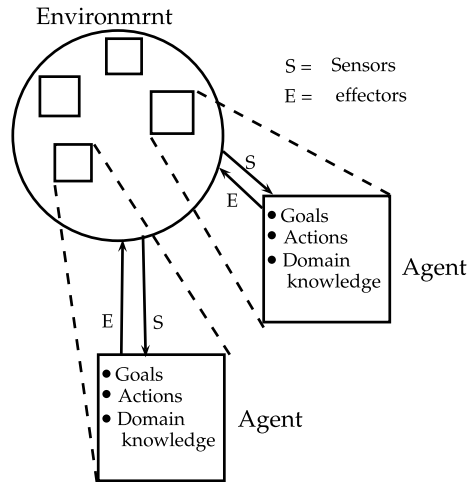
The Fig. 16.3 shows multiagent environment, where each agent is part of the environment, as well as can be modeled as a separate entity. There may be any number of agents with different degrees of heterogeneity and with or without the ability to communicate directly [10].

### 16.3.3 Multiagent Interactions

In agent-oriented view of the world, it has been found that most problems require the participation of multiple agents to represent the distributed nature of the problems, with multiple locations of control, and multiple competing interests. In addition, the agents need to interact with each other to manage dependencies resulting from their existence in a common environment, and to achieve their individual objectives. Such interactions may vary greatly—from a simple information exchanges to, in a more complex form—to request for particular action for coordination/cooperation, or negotiation, or arranging interdependent activities.

Agent interactions are differing on two characteristics with respect to computational models, like networking and shared computing: 1. Agent-oriented interactions are conceptualized as taking place at knowledge level—realized in terms of what goals should be followed, by whom, and at what time. 2. The agents are flexible problem solvers, operating in an environment that is partially observable, and agents have partial control over it. Therefore, the interactions need to be also handled in a similar and flexible manner.

The agents make use computational models to make run-time decisions about the type and scope of their interactions, and also to initiate and respond to interactions that were not anticipated at the time of design of the system [5].
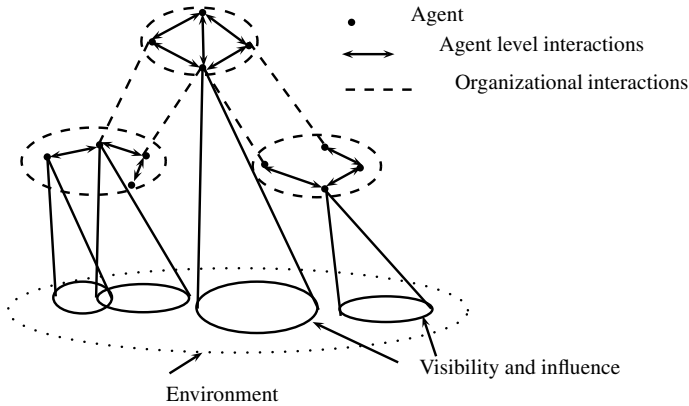
**Fig. 16.4** Canonical view of an agent-based system

Agents usually act to achieve objectives as individuals, or as a group for some larger problem-solving initiative. Hence, when agents interact with the environment, there is a hidden organizational context in them, that defines the nature of the relationship between agents—as peers working together in a team or one may be the manager of the other agents, hence may influence their subordinates' behaviors.

Since agents are required to make decisions about the various types of interactions at run time, there is a need for an explicit representation of organizational relationships of agents. Often, these relationships are subject to frequent changes, for example, the agents working on the computation of social interaction must take care of existing relationships in social networks, and should also support the evolution of these relations. The evolution is due to the creation of new relations, and due to the exit of members from these social networks. Looking at these examples, we understand that the life span of these relationships can vary from just long enough to deliver a particular service once to a permanent bond.

To cope with the dynamic scenario of variety and dynamics of relationships in agents, their protocols needs to be devised to support organizational groups to be formed and dismantled, there is need of specified mechanisms to ensure that these groups act together in a coherent way, and also there is need of structures to characterize macro behavior in a collective way.

The Fig. 16.4 shows the essential concepts of agent-based interactions.

The agents capable of having the features presented above are the *Intelligent agents* (also called software agents). These agents are autonomous components, have their own goals and beliefs. They are designed with the capability to reason about their behavior: both present and future, and offer abundant scope for fast, and incremental development of Web-based enterprise applications. The developers can use these systems for a variety of complex and dynamic domains, which range from e-commerce to research on planetary exploration systems.

## 16.4   Basic Architecture of Agent System

Many complex and intelligent agents navigate on the Internet, collect the relevant data and process them, perform various tasks including data analysis and data communication, and make the decisions on behalf of their users. The present generation of intelligent software agents can manage, organize, and communicate huge amounts of data on behalf of their users. For example, the agents in e-commerce applications can dynamically discover and compose e-services and mediate interactions. They can also be used to serve as delegates to handle routine tasks, like, monitoring activities, set up contracts, win the bid, execute business processes, and can find the best services [3].

Agents reside and execute in a conceptual and physical location called an *agency* (see Fig. 16.5), which provides facilities for locating and messaging to mobile agents and also to those that are detached agents. The agency also facilitates for collecting knowledge about agents. The core of the agency is an agent platform, with component model infrastructure, which provides the local services to agents. The core also includes the proxies to access remote services like security, agent management, communication, persistence, and naming. For mobile agents, the agent platform also provides agent transport. Some additional services provided by most agent systems are in the form of specialized agents that reside in some remote agency. There are some standard service agents, like a *broker*, *auctioneer*, or *community maker*, which augment the basic agents infrastructure. The agent platform along with service agents monitor and control message exchange by detecting any violation of rules while they engage in communication. The agent's platform is the system's core, however, in addition to this, a *component model* infrastructure empowers the agents with local services and proxy access to remote services.

An agent system comprises components with simple interfaces. The major part of the system's capability results from its loose coupling, which helps the agents to interact dynamically through the exchange of messages asynchronously. For communication with each other, the agents must follow some common and well-defined

**Fig. 16.5**   Agent system architecture
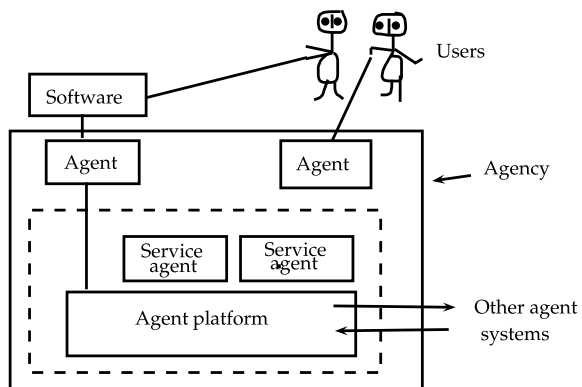
protocols. Communication of agents is through a language, called ACL (agents communication language). It is a specialized declarative language, which defines the structure and pattern of interaction between agents. This language is associated with the component model, and partitions the messages into many parts, which are relatively independent of each other. The commonly used message partitions are: message type, addressing, context, content of the message, domain description, and expected conversation patterns. Due to the message protocols and its partitioning of the message, it is easier to dynamically extend the agents to new problem domains, while the system checks conformance to expectations and allows the component model infrastructure to manage messages and agents.

The agents interact among themselves using a set of vocabularies, called ontology, which is designed for the application domain of the agents. The word set in the vocabulary describe the things, their attributes, action performed, various relationships, meanings, and how the agent's system use this vocabulary to structure the interactions, and access the devices.

## 16.5   Agents' Coordination

For performing complex tasks there is requirements to integrate a group of agents to coordinate the activity. This is possible by a multiagent system, working in either of the two modes, static or dynamic. The agents can coordinate amongst themselves, and also with people. This coordination requires messaging between the agents, the sequence of the messages may have many possible levels of choreography, depending on how loosely or tightly the allowed interactions are controlled by the system. It is common practice that, instead of directly programming as code to handle the messages coordination, some graphical models or high-level declarative rules are used. These models/rules make it easier to visualize how the agents interact. The agent system can use explicit rules or models to monitor or enforce compliance, which makes the programmer's task simple [8].

It is not always the case that coordination would mean cooperation. For example, an effective competitor will coordinate the decisions to maximize his/her advantages against the opponent. This may be seen in the planning of product promotion by a company to undercut a rival.

Various coordination strategies have emerged for computational agents. However, it is not possible to devise a coordination strategy that works equally well in all situations. If any such strategy exists, it can be easily applied for an unlimited number of constructs employed today, such as governments, corporations, markets, teams, committees, professional societies, mailing groups, etc. Whatever strategy is adopted, certain situations can stress it to be a breaking point. Any adopted coordination strategy must now be concerned about how to scale to increasingly complex situations. To map the space of potential coordination strategies, we must find out important dimensions along which they must scale and then evaluate their response to complexities along with those dimensions.

### 16.5.1  Sharing Among Cooperative Agents

Benefits at a global level of an agent system are bound to improve if all agents cooperate. However, cooperation among the agents is difficult to realize, particularly, in situations when agents are self-interested. For example, if a number of agents are trying to get the same resource, say download a specific file, the download speed is bound to decrease. Instead of this approach, if they work on grouping a social decision that is mutually beneficial, it will be good for all of them. Therefore, designing mechanisms that promote cooperation among self-interested agents is important. In fact, several game theory approaches have been found to be useful for the study of cooperation in agents, e.g., the *Prisoner's Dilemma* (PD) as a theoretical framework (see Chap. 11), which is well known for this purpose. The PD can be useful for understanding the role of local interactions to maintain cooperation among the agents. It is based on the conflict of interest, i.e., between what is the best for the individual (i.e., defection) and what is best for the group (i.e., cooperation). This creates a situation of *social dilemma*. Therefore, specific mechanisms are required to evolve cooperation to help the population to overcome this dilemma.

There are three basic approaches to prevent social dilemmas, and to promote and stabilize cooperation as follows:

Coalition-based mechanism

The coalition-based approach is useful for establishing collaboration among agents, with an individual having properties and objectives. These mechanisms use a *tax model*, due to which agents can achieve cooperation when coalitions are formed around some emerging leaders. To maintain coalitions, the leaders charge some tax from their agents in favor of some benefit (e.g., guaranteed cooperation, protection from cheaters, etc). The concept of coalition has been used in the game theory for a long time, and has been proved useful in real-world economic scenarios. The dynamic coalition formation model considers the grid topology of agents for cooperation between them and makes use of *spatial* prisoner's dilemma.

The coalitions facilitate cooperation between self-interested agents. The first approach is: a leader of coalition is paid by the agents that are in the coalition. The coalition leader also imposes its decision on the agents in the coalition to maximize cooperation. The decision making of each coalition is done in a centralized manner by a single entity, called *leading agent*. The agents' cooperation with their coalition-mates also assumes some restriction in the collaboration.

The coalition-based approach is a clear example of the known trade-off between the benefits versus the costs of collaboration (e.g., taxes). Therefore, this mechanism is called *dynamic coalition formation model*, and also *tax model*.

Partner switching mechanisms

In most real-world situations, the network topology changes frequently. There is an empirical evidence in the games on dynamic topologies that a partner switching leads to cooperative behavior. A variant model of prisoner's dilemma allows agents

to either adjust their strategies or switch their defective partners, with the aim that partner switching may help stabilize cooperation.

Self-governing institutions

The resource allocation in case of self-governing Institutions is modeled in a network, based on a formal characterization of socio-economic principles. An agent should autonomously decide how to behave with respect to coalition-mates and agents outside its coalition. Although some mechanisms promote cooperation on different network topologies, these networks are static.

### 16.5.2   Static Coalition Formation

There are two approaches for static coalition formation: 1. Optimization-based approaches, that focus on finding an optimal coalition, and 2. Game theoretic approaches. The later has applications in many real-world domains, like electronic commerce, auctions, and general resource allocation scenarios. The game theoretic approaches may also involve automated agents [8].

In coalitions with optimization objectives, the challenge in coalition formation is generating coalition structure, which turns out to be an NP-complete problem as a general case, hence the existing algorithms cannot generate solutions in a reasonable time, even with the moderate size of the game (number of agents). Hence, finding an optimal coalition can become intractable because the number of coalition structures grows exponentially with a number of agents.

Among other goals, one goal of coalition formation is to improve cooperation among the agents. The game theory approaches have been widely used to address the issue of cooperation among agents. A class of coalition formation game, called *hedonic games*, is a rich and versatile class for coalition formation, suited for both static and theoretical aspects of coalition formation, and has the property of encapsulation in the stable matching scenarios. The major focus of hedonic games is on critical stability for coalition structures, e.g., Nash stability, individual stability, contractual individual stability, and core stability. These games also characterize conditions under which the set of stable partitions is guaranteed to be non-empty.

### 16.5.3   Dynamic Coalition Formation

In formation of coalition in a dynamic environment, the agents constantly change the coalition they belong to. In such a scenario, since optimality is possible with a very small number of agents, computing of optimal coalition is either infeasible, or may take a time longer than the lifetime of a coalition for any realistic number of agents. Thus, the time constraint to find an optimal coalition prevents its use in a dynamic multiagent system, where some agents have to decide if it is beneficial to them to

join other agents for a small amount of time. This time limitation is due to the fact that for $n$ number of agents, the total number of possible coalition structures to be enumerated are of the order of $O(n^n)$, which is too large unless $n$ is a small number. For large $n$, the computation cannot be carried out in realistic times. Hence, it is necessary to make use of domain knowledge, with mathematical games and some constraints to solve the problem of coalition formation in an efficient way, for the set of agents of any specified characteristics [8].

To form a dynamic coalition, it is required to have decentralized procedures to allow self-interested agents to negotiate the formation of coalitions, as well as to divide the coalition payoffs. In the real-world scenarios, the agents may turn out to be selfish and may focus on improving their own performance, but if they are cooperative, the performance of the whole system will improve. Hence, the theory of non-cooperative games (i.e., agents are selfish) is suitable to model the formation of coalitions and their dynamics. The prisoners' dilemma is a case of this category, since the prisoners are considered as selfish (defecting is the dominant strategy) in this game. A variant of this, called *Iterated Prisoner's Dilemma* (IPD) game is widely used to model various social and economic phenomena, and the cooperation among agents. In the IPD, where a total number of rounds is random or unknown, sustained cooperation strategies are likely to emerge.

## 16.5.4 Iterated Prisoner's Dilemma Coalition Model

We present a model, where a graph (or network topology) representing population is iterated, the *nodes* in the network represent agents, and *edges* represent relations between agents. Such agents interact with their peers in the social neighborhood (the agents to which they are linked), and play the game of *Possessors-Traders* (an agent is either a possessor or a trader). Such agents not only cooperate or defect, but have resources using which they can trade. The agents can form coalitions to increase the cooperation level of the multiagent system. In such coalitions, group decisions can result in mutually beneficial cooperation, that holds over some time. The group decisions lead to, and is an indication of social behavior. In addition, the agents' neighborhood is not static, and can change partners through rewiring. Hence, in addition to the trading of resources, each agent decides the following during the game [8].

– To remain independent or to be part of a coalition, depending on which alternative provides more payoffs.
– Whom to rewire with? As agents change their neighborhood, they rewire to improve the benefits.

In addition, all the agents in a coalition behave like a unit, and all together decide how they should behave with those in the coalition (called insiders), and those outside the coalition (called outsiders). Hence, the decision about what is a coalition and what

is its behavior, is an important criterion in the dynamics of the coalition system. In their behavior the agents work in cycles: trading strategies, rewiring (changing of coalition) strategies, and coalition strategies. The Algorithm 16.1 shows this cycle, where $x$ represents *Payoffs*.

---

**Algorithm 16.1** One cycle of Agents-dynamics

---
1: $x = trade\text{-}with\text{-}all\text{-}neighbors()$;
2: $rewire(x)$;
3: *revise coalition*$(x)$;

---

Agents use certain trading strategies to trade among the agents, based on some model, which can be called by a general name *property ownership and trade* (POT) model. The trading model is based on extension of the Iterated prisoner's Dilemma, in which agents can cooperate or defect the actions. The model of POT comprises two types of players: 1. The *Possessors P*, who own the resources, and 2. *Traders T*, who sell and buy resources.

The strategy of any agent $p_i \in P$ models the practice of ownership, but does not trade. The behavior of $p_i$ depends on whether it owns a resource or not. If $p_i$ owns a resource it acts as a defector, but if it does not, then it cooperates. This strategy is shown in Algorithm 16.2, where $owns(p_i, resource)$ indicates that possessor $p_i$ owns some resource, $defects(p_i)$ means $p_i$ defects, and $cooperate(p_i)$ means $p_i$ cooperates.

---

**Algorithm 16.2** Possessor $p_i$'s Strategy

---
1: **if** $owns(p_i, resource)$ **then**
2:    $defects(p_i)$
3: **else**
4:    $cooperates(p_i)$
5: **end if**

---

An agent $t_j \in T$ is trader, who is willing to sell or buy a resource when dealing with a fellow trader $t_k \in T$. For example, if some $t_j$ has a resource for selling, it will try to get the maximum benefit by selling it. Whenever any pair of traders $(t_j, t_k)$ meet, the trader (say $t_j$) owning the resource values the resource at a random value $y_j$ (but does not make it open), such that $v < y_j < V$, and $v, V \in \mathbb{R}$ (real numbers). In response, the buyer agent $t_k$ offers a value $y_k$ for the resource, such that $v < y_k < V$. If $y_k > y_j$ then the buyer purchases the resource at some random value $y_l$, so that $y_j < y_l \leq y_k$. This is called trader $T$'s strategy, with its logic given in Algorithm 16.3. If the trader plays against who is not a trader, then it is the possessor.

---

**Algorithm 16.3** Trader $T$'s Strategy

---

1: **if** $isTrader(t_j)$ $AND$ $isTrader(t_k)$ **then**
2:   **if** $owns(t_j, resource)$ $AND$ $v < y_j < V$ **then**
3:     Sell for $y_j$
4:   **else**
5:     **if** $owns(t_k, resource)$ $AND$ $y_j < y_k$ **then**
6:       Buy for $y_l$ $AND$ $y_j < y_l \leq y_k$
7:     **else**
8:       Behave as Possessor
9:     **end if**
10:   **end if**
11: **else**
12:   Behave as Possessor
13: **end if**

---

## 16.5.5   Coalition Algorithm

A Algorithm 16.4 shows the basic strategy followed by agents either to join a coalition or leave a coalition (change to a new one). If an agent $a_i$ has the poorest payoff among all its neighboring agents after completion of the previous round of computations (line 1), then $a_i$ makes a new coalition with some agent $a_j$ (line 4), who is free and has the best payoff. If $a_j$ is already in some other coalition, then $a_i$ joins $a_j$'s coalition (line 6). This rule also enables any agent to change from one coalition to another, in case that agent receives poor payoffs in the former coalition [8].

In a dynamic network, agents form coalitions to behave as a unity. An agent can belong to only one coalition at a time. All agents belonging to a coalition are not required to be linked with each other, but behaves as a set to act together to maximize their performance. However, an agent must have at least one link to some agent belonging to its coalition. If that is not the case, it is an isolated agent, hence it must be declared as an independent agent (lines 9–10, Algorithm 16.4). This connection helps an agent to know its coalition information, strategy, share, and divide gains. Again, if an agent changes link, it does not imply that it changes its coalition—it simply rewires to change neighbors.

The agents that are in a coalition, must agree to some specific behavior to play with *insiders* (agents in the coalition) and also with *outsiders* (agents outside the coalition). We assume a flat coalition, i.e., there is no leader or central authority to impose any policy. To decide the coalition behavior in this situation, each agent votes for a strategy of either $P$ or $T$ (for possessors and traders) to play with insiders as well as with outsiders (line 16, Algorithm 16.4).

To decide the vote, each agent uses a Learning Automata (*LA*), trained from its trading history and payoffs (lines 12–15). The LA algorithm keeps two *probability models*: $InProb$, and $OutProb$. The model $[InProbT, InProbP]$ is used to assess the strategy to play against insiders. Here, $InProbT$ is the probability of being inside the coalition as a trader, and $InProbP$ is the probability of being inside the coalition as possessor. In a similar way, $[OutProbT, OutProbP]$ is to assess the strategy to play against outsiders.

---

**Algorithm 16.4** Revise Coalition(Payoffs)

---

1: **if** $poorestPayoff\_from\_neighbors(a_i)$ **then**
2:     $a_j = neighborWithBestPayoff()$
3:     **if** $independent(a_j)$ **then**
4:         $CreatNewCoalition(a_j, a_i)$
5:     **else**
6:         $JoinCoalition(a_j, a_i)$
7:     **end if**
8: **end if**
9: **if** $IsolatedAgent(a_i)$ **then**
10:     $makeIndependent(a_i)$
11: **else**
12:     $[InProbT, InProbP] = UpdateInsidersLA(Payoffs)$
13:     $[OutProbT, OutProbP] = UpdateOutsidersLA(Payoffs)$
14:     $InAction = InActionChoice(ProbInT, ProbInP)$
15:     $OutAction = OutActionChoice(OutProbT, OutProbP)$
16:     $VoteBest(InAction, OutAcion)$
17: **end if**

---

## 16.6  Agent-Based Approach to Software Engineering

In respect of software engineering, we view agents as next-generation components and *agent-oriented software engineering* as an extension of conventional CBSE (case-based software engineering). The developers can integrate different types of agents, like, personal, mobile, and collaborative agents, to build agent-based enterprise systems, covering a wide problem domain area. To patrol the networks to find available resources, special software is used, called Daemons.

Developers often use distributed objects, active objects, and components that can be scripted to implement agents. The agents are often driven by goals and plans instead of procedural code, they encapsulate business or domain knowledge. These agents often differ more from each other by the knowledge they have and the roles they play, than by the differences in their implementing classes and methods. The agents are capable of using different mixes of adaptability, mobility, intelligence, ACL, and even multiagent support. Either AI programming languages or conventional programming languages can be used to implement the agents.

Next, we introduce the techniques for tackling complexity in software [5].

Decomposition

For tackling large problems, the basic technique is to divide the problem into smaller chunks, such that it is better manageable. Each of these chunks is dealt with relative isolation. Since this limits the designer's scope, it helps to tackle the complexity of the issues, because it requires to consider only a small portion of the problem at any given time.

Abstraction

The abstraction is a process of defining a simplified model of any system, such that only the necessary and important details or properties and emphasized, while all unnecessary details are suppressed.

Organization

The process of organization is concerned with the identification and managing the relationships between various problem-solving components. Specifying and implementing organizational relationships are helpful to tackle the complexity due to two reasons: 1. It facilitates the grouping of a number of basic components, which are collectively treated as a unit at a higher level for the purpose of analysis. 2. Due to grouping the components as a unit, as well as to specify the relationships between them, a number of components may work together (cooperate) to provide a particular functionality.

## 16.7   Agents that Buy and Sell

The Software agents were used much earlier for the applications, like filtering information, match people having similar or identical interests, and automating repetitive behavior.

In the recent past, agents have found the applications in e-commerce to conduct business-to-business, business-to-consumer, and consumer-to-consumer transactions. Consider an example of buying and selling, where a company willing to place an order for procurement of stationery, assigns the tasks to agents to monitor the quantity and usage patterns of paper within the company. It also launches the buying agents when paper inventory is low. The *Buying agents* would typically perform the following tasks, more or less, in order [6].

1. collect the information automatically about vendors and the required products which best fulfills the needs of the company,
2. evaluate the various offers from the vendors,
3. make a decision about merchants and products that require further investigation,
4. negotiate the terms of transactions with these merchants, and finally
5. place orders and make automated payments.

There are several descriptive theories and models that seek to capture buying behaviors, e.g., *Nicosia model*, *Howard-Sheth model*, and *Engel-Blackwell model*. All these share six fundamental stages of the buying process:

1. *Identification*. The buyer can be motivated through product information, hence he/she becomes aware of some unmet needs.

2. *Product brokering*. The buying process comprises as its part, the Information Retrieval to determine what to buy. IR consists of evaluation of product alternatives based on the criteria provided by the buyer, whose result is a set of products, called "consideration set."

3. *Merchant brokering*. This activity combines the "consideration set" with merchant-specific information to help customer to decide as from where to buy the goods. This stage also comprises the evaluation of merchant alternatives based on buyer-provided criteria. The later is typically, the price, warranty, delivery time, availability, and reputation, which are not necessarily be in order, but varies case to case.

4. *Negotiation*. This step considers how to settle on the terms of transition. Negotiations vary in duration and complexity, for price and other attributes.

5. *Purchase and delivery*. This step signals either termination of the negotiation stage or occurs some time afterward.

6. *Product service and evaluation*. This involves the post-purchase product service, customer service, and evaluation of the satisfaction of overall buying experience.

In the present online buying and selling, many of the processes and criteria discussed above are in a common place.

The continuous running personalized autonomous agents are well suited to mediate for consumer behaviors, like, information filtering, IR, personalized evaluations, time-based interactions, and complex coordination. Many agents perform constraint-based, and collaborative filtering. Many websites of online-shopping use rule-based techniques to personalize the products offering for individual customers. Some websites use agents to experiment data-mining techniques to discover patterns in customers' purchase behavior, exploit those behaviors for sales, and use these patterns also to help customers to find other products that meet their true requirements.

The product alternatives are compared at the product brokering stage, whereas the merchant alternatives are compared at the merchant brokering stage.

## 16.8  Modeling Agents as Decision Maker

For modeling agents as decision makers, it is necessary to have modeling methods that use formal notions of mental state to represent and reason about agents. The mental states may consist of mental attributes such as beliefs, knowledge, and references. In multiagent systems, the success of one's actions and plans are governed by the actions of other agents. Thus, agents can help in constructing plans that are likely to succeed. The mental level models can bring two informal properties: 1. They provide an abstract way of representing agents, which is implementation-independent, and, 2. These models are built using an intuitive approach, and use attributes, such as goals, beliefs, and intentions. The abstract nature of models have the following practical implications [1]:

1. A single formalism can capture different agents, written in different languages, and running on different hardware platforms,
2. There are no implementation details in abstract models, and
3. Fever lower-level details in abstract models result in faster computation.

### 16.8.1 Issues in Mental Level Modeling

In mental level modeling following are the central questions:

1. *Structure*. The structure holds the designer's initial database of beliefs, goals, intentions, which are manipulated by the agent.
2. *Grounding*. It is the base for the model construction process, which is essential because we cannot directly observe the mental state of another agent.
3. *Existence*. Under what conditions a model will exist? Answer to this question will be helpful in evaluating any proposal for mental level models. Therefore, it is necessary to know, what assumptions are made, or biases we are making when we model agents in this manner.
4. *Choosing a model*. How do we choose between different models that are consistent with our data?

### 16.8.2 Model Structure

A model's mental level structure consists of three key components: *beliefs*, *references*, and *decision criteria*, which in order corresponds to accounting for the agent's perception about the world, its goals, and method of choosing actions under uncertainty, respectively.

The agents' belief help in establishing about which states of the world it considers, are plausible. As an example, the possible worlds of interest may be about weather conditions: *rainy* and *non-rainy*, and let the agent believes rainy to be plausible. In fact, the agent's preference indicates how much it likes each preference. The agent may have two possible *actions*: take an umbrella along to protect from rain, and do not take an umbrella along. The outcomes of these actions are shown in Table 16.1. The agent's preferences tell us how much significance it gives to these values. We will prefer to use real numbers to describe these values, such that larger numbers indicate better outcomes, as shown in Table 16.2 [1].

An agent would choose its action (take or not take umbrella) by applying its decision criteria to the outcome of different actions in the world. A commonly used decision criterion is *maximin*, where the action for the "best worst-case" outcome is chosen. The best outcome out of (10, −4) and (−1, 8) is 10 and the worst is −4. The best in the worst-case is −1, so the agent chooses the action "Do not take an umbrella". This is because the outcome worst −1 is better than worst −4.

**Table 16.1** Decision table for an agent

| Action (↓), Worlds (→) | Rainy | Not-rainy |
|---|---|---|
| 1. Take umbrella | Dry, Heavy | Dry, Heavy, Illogical |
| 2. Do not take umbrella | Wet, Light | Dry, Light |

**Table 16.2** Table with weighted outcomes

| Action (↓), Worlds (→) | Rainy | Not-rainy |
|---|---|---|
| 1. Take umbrella | 10 | −4 |
| 2. Do not take umbrella | −1 | 8 |

Having gone through the above example, we are now in a position for grounding. We can view the problem of describing a mental state of the agent as a CSP (Constraint Satisfaction Problem). The state of the model is such that it should have generated the observed behavior, and it is consistent with the background knowledge.

In the above example, the background knowledge is agents preferences, given in the Table 16.2, and decision criteria is *maximin*. We observe that if the agent goes out without an umbrella, it believes that "no rain will come", for it is had other beliefs it would have taken a different action.

Once an agent's model is constructed, it can be used to predict its future behavior. To give it a formal shape, we consider that an agent $\mathscr{A}$, is described as a state machine, with set of possible local states $L_{\mathscr{A}}$, a set of possible actions $A_{\mathscr{A}}$, and a program, which we call its *protocol* $P_{\mathscr{A}}$. Thus an agent is a tuple,

$$\mathscr{A} = \langle L_{\mathscr{A}}, A_{\mathscr{A}}, P_{\mathscr{A}} \rangle \tag{16.1}$$

where $P_{\mathscr{A}} : L_{\mathscr{A}} \to A_{\mathscr{A}}$. All the agents function with some environment, so we assume $L_{\mathscr{E}}$ as set of all states in the environment. The environment describes every things external to the agent, which may possibly include other agents also. The combined state of the whole system, i.e., both the agent and the environment are referred to as *global state*, and represent by a pair $(l_{\mathscr{A}} \times l_{\mathscr{E}}) \in L_{\mathscr{A}} \times L_{\mathscr{E}}$. We further assume that environment does not perform actions, and agent's actions are deterministic functions of its state and the environment's state. Thus, the set of possible worlds will be only a subset $S$ of the set of global states $L_{\mathscr{A}} \times L_{\mathscr{E}}$. And, finally, a *transition function*,

$$\tau : (L_{\mathscr{A}} \times L_{\mathscr{E}}) \times A_{\mathscr{A}} \to (L_{\mathscr{A}} \times L_{\mathscr{E}}) \tag{16.2}$$

maps a global state and an action to a new global state.

**Example 16.1** An agent $\mathscr{A}$ to model a an air-conditioner's thermostat control.

The modeling of agent $\mathscr{A}$ is shown in Table 16.3, which shows the results of transition function $\tau$. It should have local states $L_{\mathscr{A}} = \{-, +\}$, where '−' corresponds to

**Table 16.3** Transition table for thermostat agent

| Worlds: $L_\mathscr{A} \times L_\mathscr{E} \to$ Action: $A_\mathscr{A} \downarrow$ | $(-, cold)$ | $(+, cold)$ | $(-, ok)$ | $(+, ok)$ | $(-, hot)$ | $(+, hot)$ |
|---|---|---|---|---|---|---|
| Turn-on | $(-, ok)$ | $(+, ok)$ | $(-, hot)$ | $(+, hot)$ | $(-, hot)$ | $(+, hot)$ |
| Turn-off | $(-, cold)$ | $(+, cold)$ | $(-, ok)$ | $(+, ok)$ | $(-, ok)$ | $(+, ok)$ |

the state when the thermostat indicates that the temperature is less than the room temperature, and '+' for temperature greater than or equal to desired room temperature. The thermostat's protocol is given below.

| State | $-$ | $+$ |
|---|---|---|
| Action | Turn-on | Turn-off |

The thermostat's actions are modeled as $A_\mathscr{A} = \{turnon, turnoff\}$, and the environment's states are, $L_\mathscr{E} = \{cold, hot, ok\}$. For the sake of simplicity, we assume that the possible world is $L_\mathscr{A} \times L_\mathscr{E}$, which are displayed in the heading row in the Table 16.3.

Given the set of possible worlds $W$, we can associate with each local state $l$ of the agent (the thermostat), a subset $S$, $W(l)$, comprising of all worlds in which the local state of the agent is $l$. ☐

In the above example, the effects of an action on the environment do not affect the state of the thermostat. In addition, the static one-shot model assumes some simplifications. The first assumption is that the room temperature is affected only by the thermostat and not by external influences. Second assumption is that the thermostat state's actions do not affect its state.

It may be noted that, while the thermostat knows its local state, it knows nothing about the room temperature. Consequently, we made all pairs of $L_\mathscr{A} \times L_\mathscr{E}$ possible, including the $(-, hot)$ as the possible world, indicating that thermostat's local state is indicating low temperature, while the environment state is *hot*. In one aspect, it simplifies the system by assuming all possible worlds, while in other terms, it is a blessing, as this is taken as a situation, where we assume that there may be a measurement error in the thermostat [1].

**Definition 16.1** (*Belief*) A *belief assignment* function $B$ may be defined as $B : L_\mathscr{A} \to (2^S - \phi)$, so that for all $l \in L_\mathscr{A}$ we have $B(l) \subseteq W(l)$. The value $B(l)$ is referred as worlds *plausible* at $l$.

## *16.8.3   Preferences*

The *beliefs* make sense being part of a more detailed description of the agent's mental state, which has more associated aspects. One such aspect is the agent's preference order in the possible worlds, which can be taken as the agent's desire. There are various assumptions about the structure of the agent's preferences, which consider a *total order* on the set of possible worlds $W$. However, we may need a richer algebraic structure, in some cases, e.g., one in which addition is defined. We use a *value function* to represent the agent's preferences.

**Definition 16.2** (*Value function*) A value function is a function $u : S \rightarrow \mathbb{R}$.     □

This numeric approach of representation of agent's preferences is most convenient, where a state $s_1$ is at least as preferred as state $s_2$, *iff* $u(s_1) \geq u(s_2)$.

Considering the example of the thermostat (as agent), the goal of the agent is to make room temperature *ok*. Thus, the thermostat/agent prefers any global state in which the environment's state is *ok*, over any global state which may be *cold* or *hot*, and is indifferent between *cold* and *hot*. And, it is also indifferent between the states, where the environment's states are identical, i.e., $(+, ok)$ and $(-, ok)$. This preference order over possible worlds can be represented by a value function. The value function assigns zero to global state in that environment when the state is either *hot* or *cold*, and assigns 1 where the environment's state is *ok*. This outcome is represented in Table 16.4, where $*$ stands for either $-$ or $+$.

If the exact state of the world was known to the thermostat, it would have no trouble in selecting proper action based on the value of its outcome. Considering the case of value as *cold*, the *Turn-on* action would lead to the best outcome. However, when there is uncertainty, the thermostat must compare *vectors* of plausible outcomes instead of a single outcome. For example, for the belief assignment $B(l) = \{cold, ok\}$, the plausible outcome of the action *Turn-on* is $(1, 0)$, and of the action *turn-off* it is $(0, 1)$.

Given the transition function $\tau$, the belief assignment $B$, and an arbitrary, fixed enumeration of elements of $B(l)$, the *plausible outcomes* of a protocol $P$ in $l$ is a tuple whose $k$th element is the value of the state generated by applying $P$ starting at the state of $B(l)$.

**Table 16.4**   Global outcome preference for an agent

| Worlds ($\rightarrow$) Action ($\downarrow$) | $(*, cold)$ | $(*, hot)$ | $(*, ok)$ |
|---|---|---|---|
| Turn-on | 1 | 0 | 0 |
| Turn-off | 0 | 1 | 1 |

**Table 16.5** Table with weighted outcomes

| Action (↓), Worlds (→) | Rainy | Not-Rainy |
|---|---|---|
| 1. Take umbrella | 10 | −4 |
| 2. Do not take umbrella | −1 | 8 |

### 16.8.4 Decision Criteria

The values can be compared easily, however, it is not clear as how to compare the plausible outcomes. Therefore, we choose some protocols. A strategy for making choice under uncertainty is required that depends on the agent's attitude towards risk. The strategy can be represented by decision criteria, which is a function with a set of plausible outcomes and returning a set of most preferred out of these. For this we make use of *maximin* criteria discussed earlier. Hence, we reproduce the same Table as 16.5.

Note that when both the worlds are plausible, the two plausible outcomes are $(10, -4)$ and $(-1, 8)$. On the condition, the *maximin* criteria is used, the first action, corresponding to "take umbrella" is the most preferred one. But, when the *principle of indifference* is used, the plausible outcome "do not take umbrella" is preferred. Accordingly, decision criteria can be defined as follows:

**Definition 16.3** Decision criteria.

A decision criteria is a function:

$$\rho : \bigcup_{n \in \mathbb{N}} 2^{\mathbb{R}^n} \rightarrow \bigcup_{n \in \mathbb{N}} 2^{\mathbb{R}^n} - \phi \qquad (16.3)$$

that is, from sets of equal length tuples of reals, to sets of equal length tuples of reals, so that $\mathscr{U} \in \bigcup_{n \in \mathbb{N}} 2^{\mathbb{R}^n} - \phi$, we have that $\rho(\mathscr{U}) \subseteq \mathscr{U}$ (i.e., it returns a non-empty subset of the argument set).

Note that the decision criteria can be used to compare tuples. For example, if $\rho\{u, v\} = \{v\}$, then we say that $v$ is more preferred than $u$.

## 16.9 Agent Communication Languages

The agents working together, irrespective of whether they are cooperating or competing, is called a multiagent system. These systems provide higher level of abstraction than the traditional distributed computing. The abstractions are closer to the users' expectations, and allow the designers a higher flexibility in determining the behavior. For example, instead of hard-wiring a specific behavior into the agents, multiagent

system designers design the agents with the capability to negotiate amongst themselves and find out the best course of action for a given situation. The ACLs (Agent Communication Languages) must be flexible enough to accommodate abstractions such as negotiations. But, the same flexibility makes it harder to succeed in understanding their semantics [9].

Due to this reason, we must examine many elements to arrive at he meaning of a communication, which includes, type of meaning, perspective, basis (semantics or pragmatics), context, and coverage, i.e., number of communication actions included.

The formal study of languages comprises three parts: 1. Syntax, which is concerned with organizing the symbols to create the structure of language sentences, 2. Semantics, which deals with what sense is denoted by the sentences and their parts, and 3. Pragmatics, which is concerned with how the sentences are interpreted and used. The combined meaning of a sentence is obtained due to semantics and pragmatics. The pragmatics includes those considerations that are external to the language, like, state of the agents, and the environment in which the text exists. Therefore, the pragmatics can restrict, as to how the agents can relate to one another and how they process the messages which are sent or received. In a situation when agents are not fully cooperative or they cannot find out the implications, they cannot meet the pragmatic requirements.

Semantics versus pragmatics

A perspective can be combined with a type of meaning, either *personal* or *conventional*. In case of personal, the meaning of communication is based on intention and interpretation of receiver/sender. The action, "purge this file" shall be taken by the receiver as *directive*, whereas "This is an old file", shall be taken as an assertion. In Fig. 16.6, the `inform` construct is to give the information to the receiving agent, the `request` construct requests for rain, and so on. In *conventional* meaning, the meaning of communication actions is based on usage conventions. A language is nothing but a system of conventions. Violating the idea of conventions, the traditional approaches go against the wisdom of having different labels for communication actions. The language *KQML* (Knowledge Query Management Language) have all acts as variants of `tell`, whereas for communication language *Arcol*, it is `infom`.
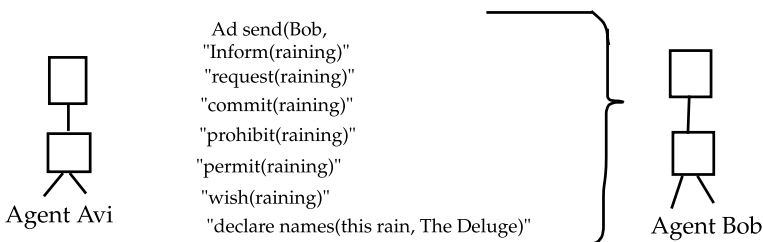


**Fig. 16.6**  An example of agent language

Context

In general, we do not understand a communication without context. Here, in agents, it is agent's physical or simulated environment, which becomes the context. For agents, the social context is not as subtle as for humans, but they must understand what an agent expects from others.

Coverage of communicative acts

When heterogeneous autonomous agents exchange information, the meaning of the exchange is decided by communicative actions. All these actions fall into one of the following categories:

- *Assertive*. This action is to inform. For example, "The door is shut."
- *Directive*. This is for request, for example, "Shut the door". It can also be used for query, e.g., "Can I shut the door?"
- *commissive*. To promise something, e.g, "I will shut the door."
- *Prohibitive*. It can ban something. For example, "Please do not shut the door"
- *Declarative*. It causes events in themselves. For example, "This information is redundant."
- *Expressive*. To express emotions and evaluations. "I wish that hurricane will stop."

Communication actions can be represented in stylized forms like, "I hereby request …" or "I hereby declare …". The grammatical form emphasize that through the language, you not only make statements but perform actions. The action by speaking becomes the essence of communication. Figure 16.6 shows that all primitives of this agent language are *assertive* or *directive*. In the agent language *Arcol*, one can simulate commissiveness using other acts. All the acts can be reduced to the category of assertive, but these categories have only restricted meanings. For example, a request in *Arcol* language is the same as conveying to the receiver that the sensor intends for it to perform the action.

Considering the code given in Fig. 16.6 for agent Avi, each communication act has a challenge for language, which promotes mental agency. The traditional approaches ignore whether Bob has really the capability to cause rain when it is requested or allowed to do so, or whether it can stop the rain when it is prohibited from causing the rain. Similar is the case for, whether Avi can make it rain when he promises; or whether Avi has the authority to permit or prohibit any of Bob's actions or to name whether conditions.

Finally, the ACL approaches conclude that if Avi's designer wants it to comply with, then it does. This is quite unsatisfactory, because it means that agents do not have any reasoning about their limitations.

### 16.9.1   Semantics of Agent Programs

A platform that supports the creation and deployment of multiple software agents must have the capability to interoperate with a wide variety of custom-made, as
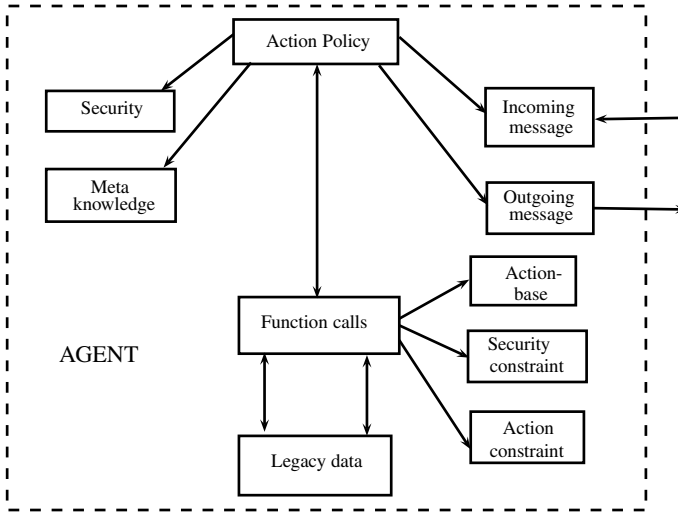
**Fig. 16.7**   Architecture of IMPACT agent system

well as legacy software sources. What it requires for a software package $\mathscr{S}$ to be considered as an agent program, is that it must come accompanied with tools to augment, modify, and message $\mathscr{S}$ to another agent.

Figure 16.7 shows, the architecture of a system, called *IMPACT*, used for the creation and deployment of multiple interactive intelligent agents. It was a joint research project created by the collaborative work of some Universities. In IMPACT, an agent comprises two parts as described below [2].

Software code

It is a program written in any programming language that supports a well-defined API (application programming interface), which may be part of the code or developed separately to augment the code. The program ($S$) may be represented as a pair,

$$S = (T_S, F_S) \tag{16.4}$$

where,

- $T_S$ is set of *all data types* manipulated by this program, and the set is closed under all the subtypes, i.e., if $\tau$ is subtype of $T_S$, then $\tau \in T_S$, and,
- $F_S$ is the set of *all pre-defined functions* of set $S$ that are provided by the package's API.

In other words, $S$ is a collection or hierarchy of objects classes in any standard object data management language.

For example, in Oracle, the database may be viewed as $S = (T_S, F_S)$, where $T_S$ comprises all data types (all attribute domains, tuple of different combinations

of domains, and relations on tuples). Whereas, $\mathsf{F}_S$ is a set of all functions, i.e., all relational operations: select, project, Cartesian products, join, union, etc.

At any given point of time $t$, the *state* of an agent will refer to a set $\mathsf{O}_S(t)$ of objects from the type $\mathsf{T}_S$, managed by its internal software code. An agent may change its state by taking *action*, which may be triggered internally or by processing a message received from other agents. But, an agent cannot directly change the state of another agent, but can do so by issuing a request message to that agent.

Semantic wrapper

A *semantic wrapper* contains a large collection of semantic information. Following are the typical contents of this information (see Fig. 16.7):

1. *Service description*. It is represented in some language, with flexibility to modify it.
2. *Message manager*. It manages the data-structure associated with the message box, and specifies and implements the policies.
3. *Action module*. It takes input of a new message consisting of an event. This message is used to trigger zero or more actions. Thus, the action module requires: (1) action base: the actions the agent may take in principle, conditions the agent state must satisfy for the actions to execute, as well as the effects of those actions, (2) Action requirements: conditions under which the agent is allowed or barred from taking the actions, (3) Action policy: What actions to choose out of many?
4. *Meta-knowledge module*. It provides to the agent information about itself, as well as about other existing agents in the *world*. This knowledge may include statistical information on the reliability of other agents, the speed at which other agents can provide the services, financial charges levied for such services. It also provides the self- knowledge, like about its own performance, analysis of various operations performed by itself.

## *16.9.2   Description Language for Interactive Agents*

An Agent's internal mechanism is based on languages, that describe the agent's behavior and its communication protocols. Examples are (1) *Soar*, a general cognitive architecture for developing systems that exhibit intelligent behavior, and (2) Knowledge Query and Manipulation Language (KQML), a language and protocol for developing large-scale sharable and reusable knowledge-bases [4].

There is another language *Q*, which is used for describing interactions between agents. Rather than depending on the internal mechanism, *Q* provides an interface between computing professionals and scenario writers. Due to change in focus, from internal mechanism to interaction, language's syntax and semantics are quite different. For example, agent accepting requests *on* or *off*, which have the standard meanings. However, if agents received a *move* command, it may have different semantics, like move fast, slow, as detailed by the semantics.

Because the language *Q* is suited for interactions, it is used for scenario writing. Example of primitives for interactions are *cue* and *action*. A cue is an event that triggers interaction, while *actions* are requests to an agent which causes the change in the environment. Unlike the programming languages, the language *Q* does not define the semantics of cues and actions. Since different agents execute the cues and actions in different ways, their semantics depend on corresponding agents. The Example 16.2 demonstrates the *cues* (preceded with question mark) and *actions* (preceded by exclamation mark).

**Example 16.2**  Cues and Actions.

```
(?hear "Hello" : from Tom}
(!walk :from class_room
       :to library)
(!speak "Hello" : to Tom)
(?see  library
       :direction north)
```

□

In the above example, the following cues and actions are there:

1. Agent waits for Tom to say Hello (?`hear`),
2. Tom walks from the class room to library (!`walk`),
3. Agent says hello to Tom (!`speak`), and
4. Agent asks, do you see the library in north (?`see`).

The above are *synchronous* actions, and each one to be followed on completion of the previous.

The *asynchronous* actions allow overlapped execution, like, in the Example 16.2, *walk* can be asynchronous action, we can walk and speak, and so are agents. To represent an action to be executed in asynchronous mode, we precede the action with a double exclamation (!!), e.g., !!*walk*. For this, the agent may say hello to Tom, just after it has begun the walk.

**Example 16.3**  Guarded commands.

```
(guard
    ((?hear "Hello" :from Tom)
     (!speak "Hello" :to Tom) ... )
    ((?see library
            :direction north)
     (!walk :from class_room
            :to library) ... )
(otherwise
     (!send "I am still waiting"
            :to Dickens) ... )))
```

□

Just like the common programming languages, the interaction language *Q* has commands for conditional branching and recursive calls. Apart from this, its has commands, called *guarded commands* for situations that require to observe multiple cues at the same time. A guarded command combines the cues, actions, and forms. After either of the cues becomes true, the guarded command evaluates the corresponding form. If no cue is satisfied, it evaluates the `otherwise` clause, as shown in Example 16.3.

In the above code, if any cue is encountered, e.g., "agent hears hello from Tom," then corresponding forms will be performed, i.e., agent says (replies) "hello to Tom." If no cue is observed by the guard command, it performs the `otherwise` clause, and the agent sends the message "I am still waiting" to Dickens.

A collection of state transitions in the language *Q* constitutes a *scenario*. A scenario defines each state as a guarded command, and it can include the conditions. A program writer can draft scenarios in the form of simple state transitions, which can describe fairly complex tasks. The scenarios can be invoked recursively.

## 16.10   Mobile Agents

All agents are not of the type, mobile agents. An agent sitting at a far off place can communicate with its environment through older time mechanisms, RPC (remote procedure call) and messaging. Such agents are called *stationary agents*, and executes only on the system on which they begin execution (not moving, but stationary). If such agents need information, which is not available on their systems, or they need interaction with an agent (program) residing on other systems, they usually make use of a communication mechanism, such as RPC or messaging.

However, a *mobile agent* is not bound to a particular system on which it begins execution, but it is free to travel to other hosts in the network. Once created in one execution environment, the mobile agent can transport its state (including data and other information), and its code, to other execution environments in the network, when everything is delivered there, it starts execution. The mobile agent is designed with special ability, due to which it can transport itself from one system to another system in the same network. This ability of the agent allows it to move to the system containing an object with whom this agent wants to interact, then to take the advantage of being in the same host or network, as an *object*. There is a number of benefits of using mobile agents rather than doing the same job by remote procedure calls and messaging. Some of the advantages are as follows.

Reduction in network traffic

Distributed systems need communication, involving interactions with multiple destinations to perform a given task. This results to a large traffic in the entire network. The mobile agents permit us to package a conversation and dispatch it to a destination host where interactions take place locally. This gets rid of the flow of raw data in the network. When a large quantity of data is stored at remote hosts, that data is

processed locally by the transported mobile agent rather than transferring over the network. That is, computation is moved to the data center.

Overcome network latency

Critical real-time systems, such as nuclear reactors, and robots in manufacturing processes are required to respond to act to changes in their environments. Controlling such systems through a factory network introduces significant latency, due to many reasons, like network being busy. For critical real-time systems, such latency is not tolerable. Mobile agents offer a solution by moving the required programs and state at the place where it is needed.

Encapsulates protocols

When data is exchanged in a distributed environment, each host owns the code, which implements the protocols. However, when protocols change to add new features, it becomes difficult to upgrade the protocol. Since mobile agents can move to remote hosts to establish "channels", this problem does not occur.

Execute asynchronously and autonomously

The mobile devices often need to rely on expensive and fragile network connections. Tasks that require a continuously active connection between a mobile device and fixed network are not economically, as well as technically feasible. To solve this, tasks are dispatched into the network in the form of mobile agents, which can operate at their ease, can move anywhere, where CPU resources and memory are abundant, and can operate asynchronously and autonomously. The devices can reconnect at a later time to collect back the agent.

Adapt dynamically

The mobile agents have capability to sense their execution environment, and can react autonomously to changes. The mobile multiagent system can distribute the agents geographically among the hosts in the network to perform any required task.

Robust and fault-tolerant

Mobile agents' ability to react dynamically to unfavorable situations is useful to build robust and fault-tolerant distributed systems. For example, if a host is being shutdown, all agents executing on that machine are warned and given time to migrate and continue their operation on another host in the network!

## 16.11   Social Level View of Multiagents

Since intelligence is mainly a social phenomenon, and it is due to the necessity of social life, there is a need to construct socially intelligent systems to understand it, and we need to build social entities to have intelligent systems. The society has adopted a set of social laws, and each agent will be required to obey these laws, and will

assume that all other agents also follow the same. These laws, in one hand, constrain the plans available to agents, and on the other hand, will guarantee certain behaviors on the part of other agents. A social law may include communication protocol which leads to rational deals with multiagents. The protocol may also include rationality constraints for cooperation. The social law also, includes the rules, like those that exist for humans for driving—left-drive or right-drive as it may be prevailing [7].

The idea of traffic rules for mobile robots highlights the important aspects of the artificial social system approach. Rather than having a centralized controller or a robot to continuously negotiate in order to avoid collisions, the better approach is that robot should follow the traffic rules, "keep always to the left of the road".

To consider the applicability of social laws, assume that there is a multi-robot network, and we think of laws for agent mobilization in such systems. In such a network, it is assumed that there is coordination among multiagent (i.e., robots). It is formally defined using the following definition.

**Definition 16.4** (*Multi-robot network*) A multi-robot network consists of a graph $G = (V, E)$, and set $R$ of robots, and a strictly positive length function $\lambda = E \to \mathbb{R}$, such that $\lambda$ associates with each edge $(u, v)$ of $G$, a distance which robot needs to travel to go from $u$ to $v$. We assume that there exists a clock such that a robot is at some node or at some point between the nodes, at each point in the time scale.   □

The action of a robot (agent) is direction and velocity. The velocity is a number of distance units it passes in a unit time. The direction and velocity are decided by the robot when it is a node in the graph. Also, a robot can observe another robot. The robots need to meet the goals which arrive at them in a dynamic fashion. The goals shall be met without collision to other robots. A collision may take place if they are at the same node at the same point of time, or at the same step distance on edge at the same time point. Based on these facts, it is possible to define some social laws.

**Definition 16.5** (*Social law for robot's movement*) Given a graph $G = (V, E)$, the social law for robot's movement determines a subset $A \subseteq E$ of edges in which robot is allowed to move, and restricts the direction of movement along each edge of $A$, and also restricts the velocity at which robots are allowed to move along each edge $e \in A$.   □

In the above definition, the social law is traffic law, which should guarantee that each robot will be able to achieve its goals (say reaching to a destination node), without any collision with other robots. This irrespective of what the other robots do.

Given a multi-robots system, a useful social law is one that guarantees non-collision system, even if all the robots initially enter the graph $G$ at arbitrary nodes, with offset of at least one unit of time from each other, and they obey the social laws. Such a system also guarantees that all the robots will reach their targets ultimately.

Modeling social actions

Design of social laws can be reduced to the problem of finding a route in a graph. For this purpose consider that there is a simple graph $G = (V, E)$, having no cycles,

no parallel edges, no cut-vertex, and has at least three vertices. A graph with no cut-vertex is called *block*.

Let $\mathbb{N}$ be the set positive integers, and $f : V \to \mathbb{N}$ be a labeling of vertices in $G$. For $U \subseteq V$, let *min* $f(U)$ and *max* $f(U)$, respectively, be the smallest and largest labels of two vertices in $U$. An *f-minimal* vertex in $U$ is any $u \in U$ for which $f(u) = min\ f(U)$, and similarly for *f-maximal*.

The labeling $f$ as above indicates a directed graph $G_f = (V, A)$ on the same vertex set as $G$, whose edge set $A \subseteq E$ is obtained from $E$ by removing each edge $(u, v) \in E$ with $f(u) = f(v)$. Alternatively, orienting each edge $(u, v) \in E$ with $f(u) < f(v)$, in the direction $vu$ if $v$ is *f-maximal* and $u$ is *f-minimal* in the block containing $(u, v)$, and in the direction $(u, v)$ otherwise.

Given this scenario, we have the following definition for routing a graph.

**Definition 16.6** (*Routing graph*) A routing of a graph $G = (V, E)$ is labeling $f : V \to \mathbb{N}$ of its vertices, for which the induced graph $G_f$ is strongly connected. A routing under which there is unique *f-minimal* vertex $r \in V$, shall be called *root*. Assigning the *root* is called *rooting* process.                                                     □

The routing of a graph can serve as a basis for useful social laws in multi-robot systems. For this, the robots are required to enter the graph network from an *f-minimal* vertex with an offset of one or more time unit from another. Additionally, the robots are required to move only along the arcs of the graph $G_f$ induced by the routing $f$. For this, a velocity function is defined as $v : A \to \mathbb{R}$ as follows: for $e = (u, v) \in A$, put $v(e) = \lambda(e)/D$ if $u$ is *f-maximal*, and $v$ is *f-minimal*, and put $v(e) = \lambda(e)/(f(v) - f(u))$ otherwise. It is mandatory that robots should move at the calculated velocities.

## 16.12  Summary

We call a component an agent if it exhibits a combination of following characteristics: autonomous, adaptable, knowledgeable, mobile, collaborative, persistent. Accordingly, agents are classified as multiagents, autonomous, adaptable, collaborative, proactive, personal, and mobile agents.

Agents have well-defined boundaries and interfaces, they are autonomous and are capable of flexible, autonomous action in that environment in order to meet its design objectives. An important benefit of multiagents is scalability. Since they are inherently modular, it should be easier to add new agents to a multiagent system than to add new capabilities to a monolithic system. They have flexible problem-solving behavior, and they can be reactive or proactive. While parallelism is achieved using multiagents, robustness and scalability are additional benefits.

In all cases of interactions, there are two major differences of agents when they are compared with networked computing and shared computing: 1. agent-oriented interactions take at knowledge level, and 2. operating in an environment that is partially observable, the apparatus should make run-time decisions.

Software agents navigate on the Internet to collect relevant data, perform tasks, and make decisions autonomously. They can transfer an enormous amount of data on behalf of their users. Some agent systems also include standard service agents, such as broker, auctionerr, or community maker.

Cooperation in multiagents is difficult when agents are self-interested, say, everyone tries to download the same file at the same time, the speed will come down. Thus, they need to communicate and cooperate. The cooperative agents should avoid the situation of a prisoner's dilemma. In cooperative agents coalitions need to be formed. There are two approaches for this: (1) optimization-based, which finds an optimal coalition, and (2) game theoretic approach, which has applications in many real-world domains.

When agents are constantly changing coalitions, there is a need of formation of dynamic coalitions. The total possible coalitions turn out to be of the order of $O(n^n)$ for $n$ number of agents. Hence, the number of agents should be small in number. To study the coalitions phenomena, the agents are represented by nodes of a graph and edges by the links indicating coalitions. The agents which are ready to provide the resources are taken as sellers and those receiving are taken as buyers. This becomes a structure to design a coalition algorithm.

Agents approach can also be applied to *software engineering*, where agents are treated as next-generation components and this software engineering as *case-based software engineering*. The complexity issues in software engineering can be tackled through decomposition, abstraction, and organization. Agents can be assigned the task of buying and selling. Software agents are used for filtering information that matches people-to-people with similar interests, and automate the repetitive behavior.

There are theories to model the buying agents, which share the six fundamental stages of buying processes: need identification, product brokering, merchant brokering negotiations, purchase and delivery, product services and evaluation. For modeling agents as decision makers, formal notions of the mental attributes are used, such as belief, knowledge, and references, accordingly, the modeling is called mental level modeling.

When agents function together in cooperative or competitive mode, the multiagent system must provide the abstractions. Instead of providing specific behavior it is designed flexible and can be coded using agent communication languages. The languages have syntax, semantics, and pragmatics.

Many of the agents are mobile, and can just sit at a far place and communicate with its environment, for example, through remote procedure calls or messaging. A mobile agent has a feature that it can partly execute on one system, and can move to another along with data, and can continue to execute the remaining part. Mobile agents reduce network traffic, overcome network latency, encapsulate protocols, execute asynchronously and autonomously, adapt dynamically, and have features of robustness and fault tolerance.

The society has adopted a set of social laws, and each agent will be required to obey these laws and will assume that all other agents also follow the same. These laws, in one hand, constrain the plans available to agents, and on the other hand, will guarantee certain behaviors on the part of other agents.

## Exercises

1. Label the following as an agent or not an agent. Explain your reasoning with justification for each.

   a. There is a program on a website to collect answers for a questionnaire.
   b. Google's web crawler, i.e., Googlebot.
   c. A distributed IR (Information Retrieval) program to helps you locate Web documents, you are interested in.
   d. A program operating for a supermarket to automatically locate and bid for the lowest food prices.
   e. A mail-filtering program that removes SPAM messages in your e-mail received in your account.
   f. An Internet-wide multi-user game playing program.
   g. A "chatterbot" program aimed to send messages to chat-rooms and try to fool the people to make them believe that messages are coming from real human beings.

2. In a multiagent system agent interact with the environment. How you can model a situation where one agent modifies the environment and the other perceive it, as a dynamic system?

3. How the architecture of a computer system is different from agent system? Give the salient differences, and justify their significance.

4. A rat searches for food, and at the same time it has to save itself from its predators, and expecting any such it either runs away or hides. For example, a single-agent system model of a rat succeeds in protecting itself from predators as well as in searching the food.

5. Explain the coordination and coalition functions between agents. How they differ from each other.

6. Write the coalition algorithm in your own language.

7. Give an example of evidence of the prevailing use of agents in online buying from the online stores.

8. Give a brief note of agent communication languages and compare them with other high-level languages.

## References

1. Brafman RI, Tennenholtz M (1997) Modeling agents as qualitative decision makers. Artif Intell 94:217–268
2. Eiter T et al (1999) Heterogeneous active agents, I: semantics. Artif Intell 108:179–255
3. Griss ML, Pour G (2001) Accelerating development with agent components. Computer 5:37–43
4. Ishida T (2002) Q: a scenario description language for interactive agents. Computer 11:42–47
5. Jennings NR (2000) On agent-based software engineering. Artif Intell 117:277–296
6. Maes P et al (1999) Agents that buy and sell. Commun ACM 42(3):81–91

7. Onn S, Tennenholtz M (1997) Determination of social laws for multi-agent mobilization. Artif Intell 95:155–167
8. Peleteiro A (2014) Fostering cooperation through dynamic coalition formation and partner switching. ACM Trans Auton Adapt Syst 9:1:1–1:31
9. Singh MP (1998) Agent communication languages: rethinking the principles. Computer 12:40–47
10. Stone P, Veloso M (1997) Multiagent systems: a survey from a machine learning perspective. CMU-CS-193 1-37

# Chapter 17
# Data Mining

**Abstract** Data mining, or knowledge discovery in databases, provides the tools to sift through the vast data stores to find the trends, patterns, and correlations that can guide strategic decision-making. The chapter highlights the major applications of data mining, their perspectives, goals of data mining, evolution of data mining algorithms—for transaction data, data streams, graph and text-based data—and classes of data mining algorithms—prediction methods, clustering, and association rules. This is followed with cluster analysis, components of clustering task, pattern representation and feature extraction, similarity measures, and partitional algorithms. Data classification methods like decision trees and association rule mining are presented with worked examples. Sequential pattern mining algorithms are presented with typical pattern mining and worked examples. The chapter concludes with scientific applications of data mining, chapter summary, and list of practice exercises.

## 17.1 Introduction

As Information Technology (IT) has progressed, there has been an abundant increase in volumes of collected data in the recent past from all sorts of varieties. It is, therefore, beyond the capabilities of humans to extract meaningful information from this vast amount of data, and it has become necessary to develop algorithms which can extract meaningful information from these vast stores of data. Searching for useful chunks of information in the huge amounts of data is known as the field of *data mining*. Data mining can be applied to all varieties of formats of data, including relational, transaction, spatial databases, as well as large stores of unstructured data such as the World Wide Web.

The amount of data stored in digital form worldwide has on the average doubled every 9 months, over many years, which is twice the rate for increase of computing power, predicted by Moore's law. This doubling of stored information, called *storage law*, is one of the reasons of motivation for data mining. Irrespective of whether this increasing volume of data will support exploration in commercial or scientific activity, the data is potentially a valuable information [1].

It took many established organizations to accumulate large volumes of data about their employees, suppliers, customers, products, and services. To perform the data mining, also known as *knowledge discovery in databases*, organizations make use of tools to sift through this vast amount of data to find the trends, patterns, and correlations among the data sets, which can guide for strategic decision-making.

The traditional algorithms for data analysis assumed the input data sets of limited number of records, hence the available memory was sufficient. Current databases, however, are much larger to fit in the main memory of most computers. In addition, to be efficient, the techniques for data mining applied to very large databases must be highly scalable. An algorithm is considered as scalable if, given a fixed size of main memory, its runtime increases linearly with the number of records in the input database.

Data mining is concerned with the identification of patterns of important structures in data; these structures may represent patterns, statistical or predictive models of data, or some relationship among the parts of the data. In the context of data mining, the terms like patterns, models, and relationships have definite meanings as follows: a *pattern* is a compact summary of a subset of data (such as people who own a racer car are likely to participate in car races); a data *model* can be model of an entire data set, and it can be predictive. This model can be used to predict the future customers' behaviors (such as whether the customer may buy a so and so product), based on the historical data of interaction with some employees of the company. It can also be a joint probability distribution with reference to some variables.

An algorithm can enumerate a large number of data patterns using a finite database, but it is important to find out what data structure is suitable for a particular data set. Identifying interesting structures and useful data patterns among the large number of possibilities is the job of a good data mining algorithm, and it must do it fast even for large data sets. Consider sales transactions in a database of an online store, where some items' transactions are more frequent than others, e.g., smartphones. The variable values frequently occurring together in such databases could be used to answer, say, which items were bought together from the same store. Such an algorithm could also discover a pattern in the database in terms of demography, with very good confidence.

Data Mining is a Science

The concept of unsupervised learning—from basic facts or axioms—has remained a curiosity since long. In the present scenario, knowledge discovery engines are commonly used to extract general inferences from facts or using training data. Using more structured approach, the statistical methods attempt to quantify the vast amounts of

data by known and intuitively understood models. The approach of problem solution through assembling knowledge from existing data sources is a radical shift from traditional approaches of problem solution.

The nature of typical data sets, in respect of their size, diversity, high dimensionality, their distributed nature, and noisy contents, makes it challenging to decide the formal specifications to any problem. This lack of control gives scope for solutions that are over-fitting, have limited coverage with missing/incorrect data coupled with high dimensionality. Once such problem is specified, the solution techniques deal with the presentation and scalability of the problem, and complexity of the solution. This complete process through which data mining makes its transitions is a *science*, called *data science*.

Due to the emergence of WWW in the form of a large distributed data repository, and the provision of large online databases that can be tapped for significant commercial gain, interest of researchers and commercial organizations toward data mining techniques has gone up exponentially. Many core mining techniques have been explored by the researchers in the major domains, like classification, clustering, rule associations, and time-series analysis. The work on data mining has focused on scaling data mining algorithms to very large data sets. In the following sections of this chapter, we shall discuss the algorithms concerning three classical data mining problems: *market-basket analysis*, *clustering*, and *classification*.

**Learning Outcomes of this Chapter**:

1. Compare and contrast different uses of data mining as evidenced in both research and application. [Assessment]
2. Explain the value of finding associations in market-basket data. [Familiarity]
3. Characterize the kinds of patterns that can be discovered by association rule mining. [Assessment]
4. Describe how to extend a relational system to find patterns using association rules. [Familiarity]
5. Evaluate different methodologies for effective application of data mining. [Assessment]
6. Identify and characterize sources of noise and redundancy in presented data. [Assessment]

## 17.2 Perspectives of Data Mining

The data mining algorithms employ a variety of models to characterize or evaluate patterns. These models are from the areas of statistics, machine learning, databases, and experimental algorithms. In addition, there are mathematical approaches that used such approximation, and technical approaches such as dynamical systems. Following are the important perspectives used in the mining of data, which cover the most domain areas of research in data mining [2, 3].

Induction

It is the most common perspective, which is based on the principle of proceeding from specific to general, and answers the questions, like "Given ten specific examples of good tourist destinations, find out the characteristics of a favorite tourist attraction?" Thus, induction is typically implemented as a search through a space of possible hypotheses. Such searches usually employ some special characteristics or aspect to arrive at a good generalization, like "sand-dunes are favorite."

Compression

Many a time, one set of data may correspond to a number of general concepts. In such cases, mining techniques typically look for the most easily described pattern. This principle is called *Occam's Razor*, which effectively equates mining to compression, such that it becomes possible to describe the original data in a more compact form using learned patterns, rather than exhaustively enumerating original data itself requiring much larger size of space. The solid base to this issue are (1) feasibility of models such as Minimum Description Length (MDL) principle and (2) computational learning theory. Most data mining systems make use of one of these views of compression to establish the effectiveness of mining patterns. For example, if set of a data is of size ten, and the number of patterns mined turns out to be 20 features long, then this mining is not good at compression.

**Definition 17.1** (*Pattern*) A pattern is a single data item of $d$ dimensions, used by a clustering algorithm. It is also called observation, or datum, or feature vector. It is represented by $\mathbf{x} = (x_1, \ldots, x_d)$. $\square$

**Definition 17.2** (*Feature/attribute*) Each of the scalar components $x_i$ of a pattern $\mathbf{x}$ are called features (or attributes) of the pattern. $\square$

Querying

The perspective of query comes from the databases, because most business data reside in industrial and commercial databases and warehouses. Commercial database experts take data mining as a form of query. For mining the data (which often is in the form of text), it required to often enhance the expressiveness of the query, like "find all the customers with similar transactions." The other perspective of querying is to find out suitable model for database, in place of relational, for data mining.

Approximation

This view of mining has an objective to find an accurate model of data, and introduce some deliberate approximation in it to find out some hidden structures in the data. One technique that has been found useful for this is *latent semantic indexing*, which is useful in *document retrieval*. This technique makes use of transformations based on linear algebra and approximations of matrices to locate hidden structures of word usages, which means doing the searches beyond the simple keyword search.

Search

The search is related to *induction*, but its focus is on efficiency, and uses forward-pruning patterns, e.g., frequent itemsets, to restrict the overall pattern space.

Beyond what has been discussed above, there are many other approaches to classify the task of data mining, which fall in various categories:

- based on the *data* they operate on, e.g., discrete data, labeled data, continuous data, and time-series data;
- based on *application* domains, e.g., finance models, economic models, web-log mining, and semi-structured models;
- based on their *induced representations*, e.g., association rules, decision trees, and correlations.

## 17.3  Goals of Data Mining

The field of data mining aims to explore very large data sets efficiently, using methods that are convenient, easy, and practical. However, this should be without extensive training as well without a large work force. All the data mining applications have some common goals, of identifying the patterns in the data, interpretation of these patterns, and then perform the prediction or description either qualitatively or quantitatively in general for all the data including those which may be generated in the near future. Following are the major goals as well as the nature of applications of data mining.

Scaling analysis to large databases

The meaning of scalability is capability to handle large volumes of data which cannot fit in memory of any computer. The objective is to abstract away the patterns from the large databases, which provide information to mining algorithms to search for the patterns.

Scaling to higher dimensional data and models

The normal statistical data analysis is a two-step process—at the first step we formulate some model, then use the data to fit to this model. However, for human beings, formulation of a hypothesis that results in a model is not possible when the data set has a very large number of variables, of the order of thousands, involving various demographics, text document analysis, retail transactions, and web browsing. Through automated discovery from such a large volume of data, a model can be derived, and can be used in lower dimensional spaces, as the problem can be understood much easier at that level.

Automating search

The search requires enumeration of large data sets, creating hypotheses—the jobs beyond human capacity. However, these are done by algorithms meant for this purpose.

Finding pattern and models understandable

The most classical methods score on the models based on accuracy, i.e., how well the model is useful to predicts the data, and on utility, i.e., on the magnitude of benefits due to derived pattern. In addition to these, there are new measures in data mining, like how well the model can be understood, on the novelty of pattern discovered by it, and on how to further simplify the model.

## 17.4   Evolution of Data Mining Algorithms

Data mining is a data-driven field; here mining is concerned with real-world data sets. In traditional data analysis, popular data sets are $d$-dimensional vectors of $\mathbf{x}$ measurements on $N$ number of objects, or $N$ such objects having $d$ number of measurements (or attributes). These data are called multivariate data, and are represented as an $N \times d$ matrix of data [4].

Some of the classical data mining and analysis problems associated with multivariate data are the following:

- *Clustering*: It is the process of learning a function, which can map $\mathbf{x}$ into a set of categories, such that the categories are not known in advance.
- *Classification*: It is the process of learning a function (i.e., learning a mapping) from $\mathbf{x}$ to $y$, where $y$ is a categorical, or already defined scalar target variable of interest.
- *Density estimation*: It is estimating the probability density function (PDF), for vector $\mathbf{x}$, i.e., $p(\mathbf{x})$.
- *Regression*: The regression is the same as classification, except that table $y$ takes real values.

The dimension $d$ of vectors $\mathbf{x}$ is significant in multivariate modeling. For example in applications of text classification, and in clustering of gene expression data, $d$ can be as large as $10^4$. As per the density estimation theory, the amount of data needed to reliably estimate a density function grows exponentially in $d$. However, many predictive problems, like regression and classification, do not require a full $d$-dimensional estimate of the PDF $p(\mathbf{x})$, and instead rely on a simpler problem of determining a conditional probability function $p(y|x)$, where $y$ is variable whose value is required to be predicted.

The first tools used to model multivariate data are old modeling methods from the domain of statistics and machine learning, e.g., logistic regression, linear regression, discriminant analysis, and Näive Bayes. The new predictive models used in this field are additive regression, decision trees, neural networks, as tools for more complex data models. They are more flexible, however they sacrifice the interpreting ability.

## 17.4.1 Transactions Data

A common form of data to be mined in most business contexts is records of individual "transactions" conducted by some individuals. Some of the common examples of these transactions are as follows: while doing purchase of groceries in a store each record is called a "basket", and in case of an individual's surfing a website each record is a description of a page requested in the session. Applying a multivariate concept we can view each of these record sets as sparse $N \times d$ matrix, where each of the $N$ rows in this matrix corresponds to an individual basket or session, and each of the $d$ columns corresponds to a particular item, and an entry $(i, j) \in N \times d$ is *true* if item $j$ was purchased by a customer, or it was requested as part of the session $i$ by the surfer, and it is *false* otherwise.

In actual cases, the parameters $N$ and $d$ can be very large. For example, a large retail chain or online store like Amazon or Flipkart might record on the order of a million baskets (i.e., $N$) per week and may have $10^5$ different items in its store available for purchasing or downloading. That means $10^{11}$ entries of $(i, j)$ cells. These numbers are a challenge for the system to be computationally tractable and suitable for statistical modeling. Considering these numbers for a store, we need to compute a pairwise correlation matrix taking time of the order of $O(Nd^2)$ and memory of $O(d^2)$, which in numeric vales are $10^{16}$ and $10^{10}$, respectively.

However, we note that the $N \times d$ matrix is sparse. Considering each of the $N$ customers buy only $d = 10$ items—the size of grocery basket—thus, only $10/10^5$ entries or 0.01% of this matrix are nonzero. Hence, our objective is to use a small subset of data, of the basket, called *itemset I* as information nugget (IN). An example of itemset is a combination of items as "bread, butter, and porridge" in a basket of grocery store.

There are many algorithms that can bring out all the *frequent itemsets* from a sparse set of transaction data. A specific itemset $I$ ($I \in d$) is called *frequent* if for $I$, the relation $f_i > T$ holds, where $f_i$ is *frequency* (rows count in which all the items in $I$ were purchased), where $T$ is some preselected threshold of rows ($T \in N$). Consider that $f_i = 10$, and some preselected threshold is $T \times N = 0.00005 \times 10^5 = 5$. Obviously, the itemset $I$ is a frequent itemset.

Other approach to data mining algorithms takes a statistical view of basket data, as a density estimation problem instead of a search problem. This has a requirement of an approach to find statistically significant itemset, i.e., an itemset $I$ whose *empirical frequency* deviates significantly from some baseline frequency. For example, a Bayesian conditional probability-based approach can discover complex multi-items' association, which has been ignored by others approaches.

We discussed that the transaction data is in the form of a sparse matrix $N \times d$. This form is in fact not a true picture of data—the real transactional data has significant additional finer structures. This structure, in the case of retail items is that they are arranged in some order, e.g., in the order of product hierarchies, and in the case of the web pages, the order is, they are usually related to each other through hyperlinks. Thus, columns of the sparse matrix of products as well of the web pages can have

themselves further more attributes, like "price" for itemsets and "contents" for web pages, as well as inter-item relationships, like laptop and its cover, web page which is the home page, etc. In addition, the rows of the transactional data (i.e., basket) can have further attributes like purchased as general, weekends, monthly, or seasonally.

## 17.4.2   Data Streams

The transaction data, instead of all available at one time, are in fact arriving continuously as a stream. As they flow by, they are available for mining only once. Similarly, the web logs continue to grow as browsing continues, over time, resulting in a stream of data. In such situations, the data miner's interest is also to study the evolution of the activity. The data streams give challenge, as to how to compute the aggregate of transaction data. One possible approach is to use an incremental learning model, such as classification tree. The further sections in this text discuss in more detail about data streams.

## 17.4.3   Representation of Text-Based Data

The text-based data can be represented in the form of a graph. The $N$ objects can be represented as nodes in a graph, and edges can represent relationships among the objects. Such "data graphs" can be used for representation of web pages, where web pages are nodes and hyperlinks are edges of the graph. These graphs can be represented by adjacency matrix, where nodes are labels for row and columns and edges are entries in the cells of adjacency matrices. These matrices are however, large and sparse. Like in graphs, some nodes have extremely high degree—outgoing or incoming edges—while others may have degree of one only. If nodes of these are sorted accordingly to their degrees, the result is a rule of the form,

$$degree \ \alpha \ \frac{1}{rank^a} \qquad (17.1)$$

where $a$ is called as "degree" of the exponent.

Use of matrices for the representation of graph has benefits that many classical methods in linear algebra can make use of graphs for analyzing the properties of the corresponding problems. For example, to discover from the connectivity information, Google *PageRank* algorithm makes use of a recursive system of equations, that define the importance of any page in terms of the importance of the pages pointing to it, as well as how many such pages are pointing to it. The page rank of each page then can be computed by solving these linear equations.

## 17.5 Classes of Data Mining Algorithms

The process of data mining is increasingly being recognized as a key solution to analyzing, digesting, and understanding the huge volume of digital data generated through businesses, government activities, and through scientific research. Achieving this solution requires the *scaling* of mining algorithms to very large data sets. Majority of the traditional database algorithms access the databases multiple times or sometimes access these randomly. These are not possible when databases are very large. At the same time we need to speed up the computation. The approaches used for designing algorithms for handling such large databases are *prediction methods*, *clustering*, and *association rules* [5].

### 17.5.1 Prediction Methods

The predictive modeling algorithms use data sets of training records as their input. The goal of predictive modeling is to build a model that predicts some specified attribute(s) value from the values of the other attributes. The predictive methods are based on one or more of the following: Bayesian probability-based methods, decision trees, neural networks, and support vector machines. We elaborate the two methods here. The Bayesian approach and neural nets have been discussed in the previous chapters.

Linear Classifiers

A model after the training procedure can be a rule set or the whole training set like the nearest neighbor. In the following we will see that a straight line can be a model as well. In an example, it has been found that the height and weight of persons are available and medical experts have found that some of them are over-weight or under-weight (see table 17.1). Accordingly, the data have been divided into two classes, and are labeled as shown in Fig. 17.1. Consultation with experts may be expensive, so we would like to construct a model from the available data. Then for any new person, given the weight and height, this model could easily predict whether he/she is over- or under-weight.

A model can be a rule like *If weight* $\geq$ 60, *then over-weight.*

**Table 17.1** A training set

| ID | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Weight (kg) | 50 | 60 | 70 | 70 | 80 | 90 |
| Height (m) | 1.6 | 1.7 | 1.9 | 1.5 | 1.7 | 1.6 |
| Over-weighted | No | No | No | Yes | Yes | Yes |

**Fig. 17.1** A training set
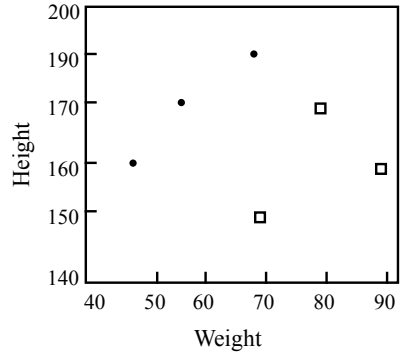


**Fig. 17.2** A linear classifier
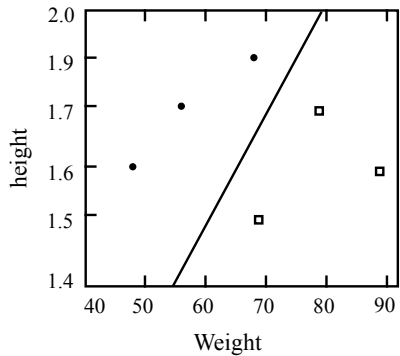


We note that this rule does not make much sense, because some tall people may be thin even though their weights are more than 60. A better model may be the following rule:

*If weight/(height)$^2 \geq 23$, then over-weight.*

The objective of classification is to identify a good model so that future predictions are accurate. Here "weight" and "height" are called *features* or *attributes*. In statistics, they are called variables. Each person is considered as a data instance (or a data observation). Mathematically, we have $\mathbf{x} = [weight, height]$ as a data instance and $y = 1$ or $-1$ as the label of each instance. Here, we have six training instances $x_1, \ldots, x_6$ with corresponding class labels as $y = [-1, -1, -1, 1, 1, 1]^T$ (here $-1$ and 1 mean under-weight and over-weight, respectively).

As discussed above, we now concentrate on the linear classifier, and show that a straight line can be a model also. Figure 17.2 shows that a line is separating the training data of over-weight and under-weight persons, and this line can be expressed by

$$0.2 \times weight - 10 \times height + 3 = 0. \tag{17.2}$$

In general, such a line can be expressed by

$$\mathbf{w}^T\mathbf{x} + b = 0, \tag{17.3}$$

where $\mathbf{x} = [weight, height]^T$, $\mathbf{w} = [0.2, -10]^T$, and $b = 3$. Then, for any new data $\mathbf{x}$, we check whether it is on the left or the right side of the line. That is,

$$if \ \mathbf{w}^T\mathbf{x} + b > 0 \ \text{predict } \mathbf{x} \text{ as "over-weight"},$$
$$< 0 \ \text{predict } \mathbf{x} \text{ as "under-weight"}.$$

Support Vector Machines

The support vector machines (SVMs) (see page 418, Chap. 14, for details) are based on simple principle of classification; they are powerful and popular approaches for predictive modeling. They are successful in a number of applications, such as face detection, handwriting recognition, text classification, and charmed quark detection.

SVMs are suitable for solving classification problems where attributes having two possible values 0 and 1 are to be predicted. The classification using SVMs is performed in a 2D space, where *predictor attributes* (i.e., 0, labeled with circles) separate from those with *dependent attribute* (i.e., 1, labeled with boxes) as shown in Fig. 17.3.

We can compute the optimal separating surface in SVMs by maximizing the *margin* of separation between predictor and dependent attributes. This margin is the distance between boundaries of the points of these two types of attributes (Fig. 17.3), and is important as it is a measure of safety (robustness) in separating the two sets of points, hence larger the margin, the better it is. As per the standard SVM formulation, computation of optimal separating surface requires solving a *quadratic optimization problem* [5].

Decision-Tree construction

Decision trees are specially attractive in data mining applications because any human analysis can easily comprehend the models of decision trees. Further, the construction of decision trees does not require any input parameters or prior knowledge about the data. We start at the root node of the tree, and repeatedly choose a child node based

**Fig. 17.3** Supported Vector Machine-based classification
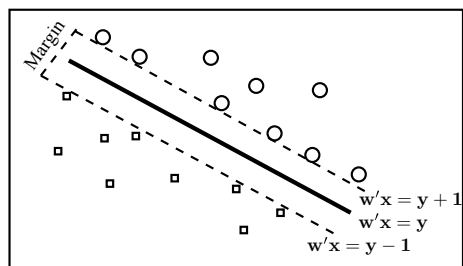
on the *splitting criteria* that evaluates a condition on the input record at the current node. At the final node (leaf node), the process associates the record with the unique leaf node of the tree.

### 17.5.2 Clustering

The process of clustering partitions a set of data, according to some similarity measure, into several groups such that "similar" records are in the same group, so that each group represents a similar subpopulation in the data. As an example, each cluster could be a group of customers, which has similar purchase histories or interactions or some other factors or combinations (Sect. 17.6 discusses clustering in more details).

One technique to achieve scalability in clustering is to incrementally summarize the regions of the data such that denser regions are summarized first. Because a cluster corresponds to a denser region, the records within this region can be summarized collectively through a summarized representation called "cluster feature" (CF). An example of CF is a triple comprising of cluster centroid, cluster radius, and the number of points in the cluster.

The CF-based approach is considered efficient due to two reasons: (1) they consume lesser memory space as all the objects in a cluster are not required to be maintained, and (2) if properly designed, they constitute sufficient information for computing all intra-cluster and inter-cluster measurements required for making clustering decisions. In addition, the distances between clusters, CFs, and radii of clusters, and hence other properties of merged clusters, all can be computed quickly from the CFs of individual clusters.

Some points in clusters can be discarded, while the others can be compressed as defined below.

**Definition 17.3** (*Discardable point*) A point is considered discardable if its membership can be ascertained with high confidence. □

**Definition 17.4** (*Compressible point*) A point that is not discardable, but belongs to a tight subcluster consisting of a set of points that always move between clusters simultaneously, is called a compressible point. □

The CFs are also useful for scaling iterative clustering algorithms, like *k*-means algorithm, and while doing so it identifies three types of points (records): sets of discardable points, sets of compressible points, and a set of main-memory points. In a cluster, only the CF of all the discardable points are retained, and the actual points are discarded. Out of the remaining points, the compressible points are compressed, and those still remaining, since they are neither discardable nor compressible, are designated as main-memory records. The iterative cluster forming algorithm then moves only the main-memory points and the CFs of compressible points, between clusters until a criterion function is optimized, which concludes the formation of a cluster.

### 17.5.3 Association Rules

Under the association rules, we use a concept called *market basket*—a well-defined business activity—which is a collection of items purchased by a customer in an individual transaction. Such a transaction is possible due to a customer's purchase from, say, a grocery store, or an online purchase from a store such as Flipkart and Amazon.

By performing business activities over time, the retailers usually accumulate huge collection of transactions of performing a business activity. A common type of analysis performed on the collections of such transactions' database is to find sets of items, or *itemsets*, that appear together in many transactions. A pattern usually required to be extracted through this analysis consists of an itemset (market basket) and the corresponding number of transactions that contain it, with the objective that the business can use knowledge of these patterns to improve the placement of items in a store together, or can arrange the mail-order catalog pages, or web pages on a website, or use this criteria to motivate potential customers with attractive offers who can buy these itemsets.

The task of *association rule mining* is finding correlation between items in a data set. The initial research in rule mining was largely motivated by the analysis of market-basket data, the result of which allowed companies to better understand purchasing behaviors and, as a consequence, better market audiences [5].

## 17.6 Data Clustering and Cluster Analysis

Data mining and clustering are both exploratory activities, hence clustering methods are well suited for data mining. Clustering is often used as an important initial step in several data mining processes. The data mining approaches that use clustering methods are *predictive modeling, database segmentation*, and *visualization* of large databases [6].

Data mining is often carried out on relational databases, in cases of transactions that have well-defined structure and its columns can be used as features. However, it is also carried out on very large unstructured databases like WWW, where the contents are natural language text, mostly in HTML or XML format.

Clustering is in fact not a new field; it was being used in machine learning, statistics, and biology. However, scalability was not a design goal in these fields, and it was always assumed that complete a data set would fit in the main memory, and focus remained on improving clustering quality and not scalability. Due to this historical reason, majority of the clustering algorithms available today do not scale to large data sets. Clustering methods are used in data mining for segmenting the databases into homogeneous groups, that serve the purposes of data compression, because now we are working with the clusters and not with individual items. It helps to identify characteristics of subpopulations that are targeted for specific purposes

(e.g., marketing certain items aimed at specific section of the population). Clusters in large databases can be visualized, to help the analysts in identifying groups and subgroups that have similar characteristics.

By classification and grouping of objects based on common properties in some meaningful way, we are able to analyze and describe the world. We human beings are skilled enough at dividing the objects into groups (i.e., clusters) and can assign particular objects to these groups. For example, even young children can label the objects, as flowers, animals, trees, books, etc. Hence, clustering is an important criteria for understanding the objects as different classes. Every new object is identified based on whether it belongs to a so and so class or not. For example, if an object $x \in A$, where $A$ is a class, say apple, then we say "$x$ is an apple." Naturally, whether $x \in A$ or not depends on certain attributes of $x$.

The clustering process groups various data objects based only on information in the data items, which describes the objects and their relationships to objects in the same group. The goal of clustering is that the objects within a group be similar/related to one another, and different/unrelated to objects in other groups. The clustering will be better or more distinct, if there is greater similarity/ homogeneity within a group and greater difference between the groups.

The following example demonstrates a sample database as clusters.

**Example 17.1** Clusters of customers' database based on three purchase behaviors: quantity, unit price, and their combination.

From the given data set, for each customer we compute the total number of items purchased and average price of all the items purchased. There are total 9 transactions as shown in Table 17.2, which are distinguished across three clusters. The clustering is based on common properties of items in each group/cluster, as follows: customers in cluster 1 purchased a few high-priced items, customers in cluster 2 purchased many high-priced items, and customers in cluster 3 purchased few low-priced items.                                                                                                              □

In clustering, we organize a collection of patterns into groups based on their similarity. These patterns are usually represented as vectors of measurements, or

**Table 17.2** Data groupings of similar objects

| Cluster no. | <Qty, unit price> |
|---|---|
| Cluster 1 | <2, 1800> |
|  | <3, 2050> |
|  | <5, 2270> |
| Cluster 2 | <15, 1800> |
|  | <18, 2200> |
|  | <12, 2380> |
| Cluster 3 | <3, 250> |
|  | <4, 180> |
|  | <4, 200> |

**Fig. 17.4** Data clustering: **a** input patterns, **b** clusters formed



(a)                                                              (b)
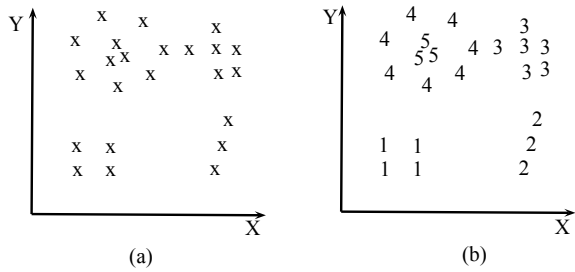
points in a multidimensional space. Intuitively, the patterns in the same cluster are more similar to each other than those in different clusters. Figure 17.4a, b depicts an example of clustering based on patterns, where input patterns are shown in (a), and the desired clusters formed are shown in part (b), where clusters belonging to the same cluster are shown by identical labels. Note that the measurement of the patterns in this case is $(x, y)$ coordinate values.

One technique for clustering is called *supervised learning*, while other is called *unsupervised learning*. In supervised technique, a collection of labels, i.e., pre-classified patterns, are already provided, and the task is to label newly encountered unlabeled patterns. The already provided labels, called training patterns, are used to learn the descriptions of classes that in turn are used to label new unlabeled patterns. In the case of unsupervised learning, the task is to group a given collection of unlabeled patterns into some meaningful clusters. In fact, some kind of labels are there, associated with the clusters this time also, but this category of labels are data driven—obtained solely from the data, and not predefined.

## 17.6.1   Applications of Clustering

Clustering has applications in several fields, which require exploratory pattern-analysis, grouping, decision-making, and machine learning. Some applications, as examples, are given below.

Information Retrieval

The WWW comprises millions of pages of text, and a query to a web search engine, like "colleges" would return hundreds of pages having relevance to the query. However, these results can be grouped (as clusters) based on the categories like graduate programs, fees structures, intake, specializations, etc. Then each individual can be explored by the user.

Biology

Biologists are applying clustering techniques to analyze large volume of genetic information, for example, to find out the genes groups which have similar functions.

Business

In business and commerce, a large amount of information is collected on current and potential customers, then clustering is performed on this information to segment the customers into smaller number of groups so that additional analysis can be performed on each group, for example, to predict marketing potential.

Similarly, there are applications for image segmentation, pattern classification, and data mining, discussed in the following. In many such problems in such areas, there is little prior information (or statistical models) available about the data, and it is requirement that a decision-maker should make as few assumptions about the data as possible. It is under these conditions the clustering approach is useful for exploring the interrelationships among the data points to determine their structure.

## 17.6.2   *General Utilities of Clustering*

Clustering provides an abstraction from individual data objects to clusters comprising these data objects. Thus, each cluster is representative of a data item and can be called as a prototype for the data item. Having this, given a data item, it is possible to determine the closest representative cluster of that data item. Based on this deductive process, the following general *utilities* can be constructed using clustering techniques.

Summarization

Many data analysis methods, such as regression, have time complexity of $O(n^2)$, which makes it computationally difficult for large size of $n$, the number of objects. However, instead of applying the required algorithm to the entire data set, if it is applied on the representative clusters, the complexity will be far less, as it will be decided by the number of clusters, and not the data items.

Nearest neighbors

To find the nearest data item neighbor of a $i$th element, it needs to be compared with $n - i$ elements, and for all $n$ items it is $O(n^2)$ complex. We know that if two data items are in two different clusters, then they cannot be nearer than the corresponding clusters. Thus, if $n$ data items are in a set $C$ clusters, then complexity to find the nearest neighbor is only $O(|C|)$, which may be far less than $O(n^2)$.

Compression

Consider using prototype for representing data items, such that each data item is represented by an index to its cluster. When similar data items are represented by a single cluster holding only one data item, it is effective compression. When this process is applied to sound, image, and video data, there is some loss of information, which is acceptable, however, there is substantial reduction in size.

**Example 17.2**  Consider a group of 12 sales records each indicating sales price, and have been sorted in ascending order as 5, 8, 11, 13, 15, 35, 45, 55, 72, 92, 201, and 215. It is required to partition these into three clusters.

**Table 17.3**  Simple clustering of data

| Cluster 1 | Cluster 2 | Cluster 3 |
|-----------|-----------|-----------|
| 5, 8, 11, 13, 15 | 35, 45, 55, 72, 92 | 201, 215 |

**Fig. 17.5**  Data sets on which centroids appear failed



(a)                                      (b)
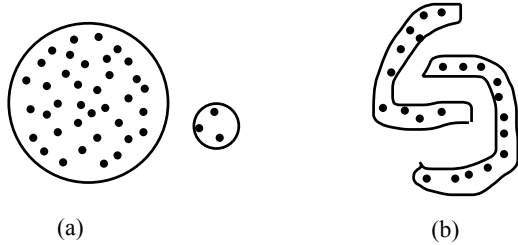
The partitions finally formed are shown in Table 17.3.

This has been obtained using a simple clustering technique that partitions the data along two largest gaps in the data sets.                                                      □

### 17.6.3   Traditional Clustering Methods

The partition-based technique, such as $k$-means partitioning, is based on optimizing a given criterion that attempts to break a data set into $k$-clusters. This approach assumes the cluster shapes as *hyper-ellipsoidal*, and the sizes of all clusters are assumed to be same. However, it cannot find clusters that vary in size, as shown in Fig. 17.5.

However, the Density-Based Spatial Clustering of Applications with Noise (DBscan) clustering technique can be used to construct clusters of arbitrary shapes and sizes. This method defines a cluster to be a *maximum-set* of density connected points—every core point in a cluster has got at least a minimum number of points within a given radius. We can reach to all these points in a cluster from any point in that cluster by traversing through a path of densely connected points, but the points across different cluster cannot. Finally, this technique can be applied only when density can be found out in advance, and it is uniform throughout the data set [6].

### 17.6.4   Clustering Process

A general pattern clustering process has the following steps:

1. pattern representation, which may also include feature selection and extraction,
2. defining proximity measures patterns specific to data domains,

3. grouping of patterns (clustering),
4. optionally, abstraction of data, and
5. optionally, assessment of output.

Figure 17.6 shows the first three steps from above, of a typical case of clustering. The feedback path indicates that the grouping process output could affect feature extraction and similarity computations in the next iteration.

The *pattern representation* depends on a number of criteria, these are available patterns, classes and their number, feature types, and their scale for clustering algorithm. All this information is not in the control of a programmer, hence *feature selection* process helps in identifying the most effective subset of the original features to use in clustering. Through *feature extraction*, one or more transformations of input features is carried out to produce new salient features. Either the selection or selection along with extraction can be used to obtain an appropriate set of features to use in clustering [7].

**Definition 17.5** (*Pattern*) A pattern set is denoted by $\mathscr{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. The $i$th pattern in $\mathscr{X}$ is denoted by $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,d})$. In many cases a pattern set to be clustered is viewed as an $n \times d$ pattern matrix.                     □

*Pattern proximity* or closeness of one pattern to other is usually measured by a distance function defined on a pair of patterns. A simple distance measuring function is *Euclidean distance*, which is often used to reflect dissimilarity between two patterns—the more is the Euclidean distance between two patterns, the more is the dissimilarity between them. And, if the Euclidean distance is zero, the patterns are identical. Other similarity measures can be used to characterize the *conceptual similarity* between patterns.

The *grouping* step can be carried out in many ways. The output of clustering can be *crisp* or *fuzzy*. When clustering required at output is crisp (*hard*), the data is partitioned into groups, whereas when it is fuzzy partition, each pattern has a variable degree of membership of [0, 1], in each of the output clusters. In the other approach, the algorithms based on hierarchical clustering produce a nested series of partitions by merging or splitting of clusters based on similarity measures. In yet another way, partitioning is performed such that, algorithms identify a partition that optimizes a clustering criterion.
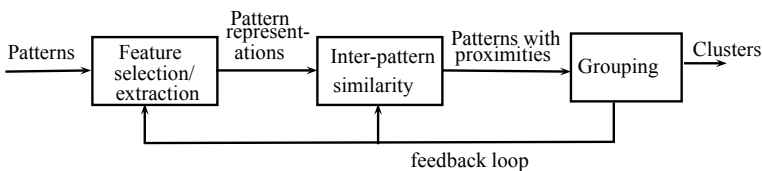


**Fig. 17.6**  Stages in clustering

**Definition 17.6** (*Hard clustering*) It is a technique that assigns a class label $l_i$ to each pattern $\mathbf{x}_i$, which identifies its class. The set of all labels for pattern set $\mathcal{X}$ is $\mathcal{L} = \{l_1, \ldots, l_n\}$, with $l_i = \{1, \ldots, k\}$, where $k$ is the number of clusters.    □

**Definition 17.7** (*Fuzzy clustering*) It is a clustering procedure that assigns to each input pattern $\mathbf{x}_i$ a fractional degree of membership $f_{i,j}$ in each output cluster $j$, for all the $k$ clusters.    □

*Data abstraction* is aimed at extracting simple and compact representation of a data set. Here simplicity is from the point of view of automatic analysis, so that a machine can do the processing efficiently. Alternatively, the simplicity is due to being human oriented, such that the representation is easy to understand and intuitively convincing for humans. Usually, a data abstraction is a compact description of each cluster in the form of cluster prototypes or as representative patterns such as the centroid.

All clustering algorithms produce clusters when presented with data, irrespective of whether the data really contain clusters or not. It is not necessary that every set of data contains some clusters. For example, the continuous sequence 1, 2, . . . , 100, in no way represents a cluster. Only what we can have is all these numbers as clusters, each of size one. So, it is important, how to evaluate the output of a clustering algorithm? Apart from this, to evaluate the cluster quality, we want to know what characterizes a "good" clustering result and a "poor" one? If the data does contain clusters, some clustering algorithms may obtain "better" clusters than others.

Some of the criterias used to compare clustering algorithms are based on (1) the manner in which clusters are formed, (2) data structure of the cluster, and 3. how sensitive the clustering technique is to changes, which do not affect the data structure of the cluster.

## 17.6.5   Pattern Representation and Feature Extraction

A better quality of pattern representation will result in a clustering that is simple and easily understood; on the contrary, when it is a poor representation, it may result in a complex clustering, even some times its true structure may be difficult to recognize. For example, if a standard technique like Cartesian coordinates is used to represent the patterns, many clustering algorithms are likely to fragment the data into two or more clusters. But, if polar coordinate are used for representation of the clusters, the radius coordinate causes tight clustering and a one-cluster solution can be easily obtained.

A pattern can represent a physical object or an abstract notion. A physical object can be a chair, table, book, house, etc., while an abstract notion can be, e.g., a style of writing, attitude, belief, etc. Both of these can be represented in the form of multidimensional vectors, one dimension for a one feature. The features of a pattern can be either quantitative or qualitative. For example, if *weight* and *color* are the two features used, then (*black*, 5) is the representation of a black object with 5 units of weight, for degree of blackness.

The features can be classified as quantitative and qualitative.

1. Quantitative features:

    a. discrete values;
    b. continuous values;
    c. interval values.

2. Qualitative features:

    a. nominal or unordered (e.g., color);
    b. ordinal (for temperature, e.g., cool or hot) or (for sound intensity, e.g., quiet or loud).

Other representations making use of structured features are represented as tree structures. In structured representations, a parent node represents a generalization of its child nodes. For example, a parent node "4-wheeler" could be a *generalization* of child nodes labeled as "cars", "jeep", and"tractor". Further, the node "cars" could be a generalization of car make, "Hundai", "Tata", "Maruti", etc. The generalized form of pattern representation, also called *symbolic objects*, are defined by logical conjunctions of events. These events link values and features, where features can take one or more values and all the objects are not required to be defined on the same set of features.

It is often beneficial to isolate only the most discriminative and descriptive features of the input set, and use these features exclusively in subsequent analysis.The isolation of features can be through selection or extraction. A feature *selection* technique identifies a subset of the existing features for subsequent use, while feature *extraction* technique computes new features from the original set to be used later. In both these cases, the objective is to obtain better classification or computation efficiency or at least one of these.

## 17.7  Clustering Algorithms

The selection of features in a pattern is an essential requirement in statistical pattern recognition. However, in the clustering context, where it lacks class labels, the feature selection is an ad hoc, but a necessity. As it lacks class labels, there can only be a trial-and-error process for the selection of features. The resultant patterns are clustered, and the output is evaluated using a *validity index*. There are some popular feature extraction processes, like principal components analysis, which do not depend on labeled data and can be used directly. The patterns having smaller number of features are beneficial, as the output can be visually inspected by a human.

For clustering the objects/patterns, the first requirement is to find out similarities between them, and more similar patterns are clubbed together to form clusters. In the following, we discuss how to measure the similarities between patterns and some standard algorithms for clustering.

## 17.7.1 Similarity Measures

Similarity is fundamental to the definition of a cluster. Hence, a measure of the similarity between any two patterns drawn from the same feature space is important for clustering procedures. Since there are many feature types and scales, the choice of distance or proximity measure must be done carefully. It is usually common to compute the dissimilarity (distance) between two patterns rather than the similarity [7].

**Definition 17.8** (*Distance measure*) Given two objects $O_1$, $O_2$ from the possible universe of objects, the distance or dissimilarity between $O_1$ and $O_2$ is a real number denoted by $d(O_1, O_2)$. □

Consider that $A$, $B$, and $C$ are three objects or patterns, the following properties hold for the distance measure for these objects:

$d(A, B) = d(B, A)$ : by rule of *Symmetry*,
$d(A, B) = 0$, if and only if $A = B$ : by *Constancy of self-similarity*,
$d(A, B) \geq 0$ : by *Positivity*,
$d(A, B) \leq d(A, C) + d(C, B)$ : by rule of *Triangular inequality*.

The dissimilarity between two patterns is defined on the feature space using the distance measure. Our focus shall be on distance metrics with continuous features. The popular metric for continuous features is the *Euclidean distance*, expressed by

$$d_2(\mathbf{x}_i, \mathbf{x}_j) = \left(\sum_{k=1}^{d}(x_{i,k} - x_{j,k})^2\right)^{1/2}$$
$$= \| \mathbf{x}_i - \mathbf{x}_j \|_2 . \tag{17.4}$$

Equation (17.4) is a special case of the *Minkowski's metric*, where $p$ was taken as 2, expressed by

$$d_p(\mathbf{x}_i, \mathbf{x}_j) = \left(\sum_{k=1}^{d}(x_{i,k} - x_{j,k})^p\right)^{1/p}$$
$$= \| \mathbf{x}_i - \mathbf{x}_j \|_p . \tag{17.5}$$

The approach based on Euclidean distance has an intuitive appeal, and the method is commonly used to evaluate proximity of objects in 2 and 3D spaces. The method works well when the data set comprises "isolated" or "compact" clusters. However, it has a drawback, as there is a tendency of large-scaled feature to dominate the others features. The solutions to this, is to incorporate normalization to the continuous features (with a common range or variance) or some other weighting schemes.

**Example 17.3** Consider a set of 2D data points as shown in Table 17.4, and given a new data point, $x = (2.5, 2.9)$ as a query, rank these database points based on similarity with the query, using Euclidean distance.

**Table 17.4**  Original 2D data

|       | $A_1$ | $A_2$ |
|-------|-------|-------|
| $x_1$ | 1.9   | 1.7   |
| $x_2$ | 2.1   | 2.1   |
| $x_3$ | 2.6   | 3.0   |
| $x_4$ | 2.2   | 2.5   |
| $x_5$ | 1.8   | 2.0   |

**Table 17.5**  Euclidean distances

| Given data point | Euclidean distance with $x$ |
|------------------|-----------------------------|
| $x_1$            | 1.341                       |
| $x_2$            | 0.894                       |
| $x_3$            | 0.141                       |
| $x_4$            | 0.500                       |
| $x_5$            | 1.140                       |

Using Eq. (17.4), we compute the Euclidean distance for the 2D data points $x_1$, …, $x_5$ with respect to the query $x = (2.5, 2.9)$. The result are shown in Table 17.5.

## 17.7.2   Nearest Neighbor Clustering

Since proximity between items plays an intuitive role in clustering, a method based on the *nearest neighbor* distances can be an obvious choice as a basis of clustering procedures.

An iterative algorithm assigns each unlabeled pattern to the cluster of its nearest labeled neighbor pattern, with the condition that the distance to that nearest pattern is below the given threshold.

This process continues until all the input patterns are labeled.

To grow the clusters from the nearest neighbor, a concept called *mutual neighbor distance*, call it $MN_d$, can be used, which is expressed as

$$MN_d(\mathbf{x}_i, \mathbf{x}_j) = C_n(\mathbf{x}_j, \mathbf{x}_i) + C_n(\mathbf{x}_i, \mathbf{x}_j). \qquad (17.6)$$

In the above, $\mathbf{x}_i$, $\mathbf{x}_j$ are patterns, $C_n(\mathbf{x}_i, \mathbf{x}_j)$ represents the count of neighbor numbers of $\mathbf{x}_j$ with respect to $\mathbf{x}_i$. Figure 17.7 illustrates this concept. There are total six patterns, $P, Q, R, S, T, and U$. In part (a), nearest neighbor of pattern $P$ is $Q$, and $Q$'s nearest neighbor is $P$. Also, $C_n(P, Q) = C_n(Q, P) = 1$, hence $MN_d(P, Q) = 2$. If $C_n(Q, R) = 1$ but $C_n(R, Q) = 2$, then $MN_d(Q, R) = C_n(Q, R) + C_n(R, Q) = 3$.

Figure 17.7b is obtained from Fig. 17.7a by adding three new patterns $S, T, and U$. Now, $MN_d(Q, R) = 3$, but $MN_d(P, Q) = 5$. Note that $MN_d$ between $P$ and $Q$ has increased from 2 to 5 due to additional three patterns $S, T, and U$, though the position of $P$ and $Q$ have not changed.
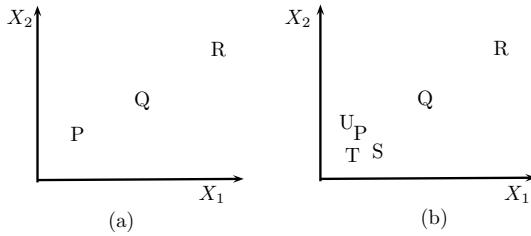
**Fig. 17.7** Nearest neighbor clustering: **a** $P$ and $Q$ are more similar than $P$ and $R$, **b** $Q$ and $R$ are more similar than $Q$ and $P$
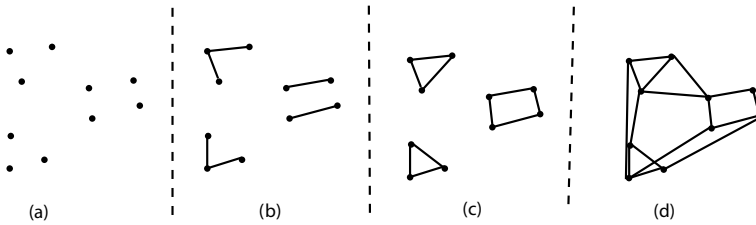


**Fig. 17.8** Construction steps of $k$-nearest neighbor graph using original data: **a** original data, **b** 1- **c** 2- **d** 3-nearest neighbor graphs

A general case of the nearest neighbor algorithm is $k$-nearest neighbor algorithm. Figure 17.8 illustrates for $k = 1$, 2, and 3-nearest neighbor graphs for some simple data set.

There are many advantages of representing data items using a $k$-nearest neighbor groups (clusters). The far apart data items are completely disconnected in this approach, and since the data items are connected with the nearer items such that the weights on the edges in the graph are indicators of nearness, these weights are also indicators of population density in the data items' space. Since the items in sparser and denser regions are modeled uniformly, the sparsity of the representation results in algorithms that are computationally more efficient.

## 17.7.3 Partitional Algorithms

A partition-based clustering algorithm first obtains a single partition of the data, without any structure. As the next step, clusters are produced by optimization of a *criterion function* defined locally (over a subset of the patterns) or defined globally (on the entire set of the patterns). Searching for a set of possible labeling for an optimum value of a criterion is computationally expensive, and combinatorial in nature. To simplify this, the algorithm is typically run multiple times with different starting states, and the best configuration obtained from all of the runs is used as the output clustering.

The above discussed algorithm has a problem of providing a choice of the desired number of output clusters. However, it is computationally more efficient than hierarchical clustering, when used for a large data set.

## Squared Error Algorithms

The squared error function approach is the most intuitive concept for partitional clustering, as it is ideally suited for compact and isolated clusters. For an input set of $\mathscr{X}$ patterns, the squared error for clustering $\mathscr{C}$, consisting $K$ clusters $(C_1, \ldots, C_K)$, can be expressed as

$$e^2(\mathscr{X}, \mathscr{C}) = \sum_{j=1}^{K} \sum_{i=1}^{m_j} \| \mathbf{x}_i^{(j)} - \mathbf{c}_j \|^2 . \tag{17.7}$$

In Eq. (17.7), $\mathbf{c}_j$ is centroid of the $j$th cluster in total $K$ clusters formed, $m_j$ is the number of patterns in the $j$th cluster, and $\mathbf{x}_i^{(j)}$ is the $i$th pattern in the $j$th cluster.

---

**Algorithm 17.1** Squared Error Clustering Algorithm

---

1: Select an initial partition $\mathbf{X}$ of patterns, with a fixed $k$ number of clusters, and cluster centres
2: **repeat**
3:    **for** each pattern $\mathbf{x}_i \in \mathbf{X}$ **do**
4:       Find centroid $\mathbf{c}_j$ (of cluster $C_j$) having minimum distance with pattern $\mathbf{x}_i$
5:       $C_j = C_j \cup \{\mathbf{x}_i\}$
6:       Compute the new centroids (cluster centers) of all the clusters
7:    **end for**
8:    Merge and split clusters based on some heuristic criterion
9: **until** convergence is achieved
10: **end**

---

The steps of the squared error clustering algorithm are listed in Algorithm 17.1. The repetition in the *repeat …until* loop continues until the convergence is achieved, i.e., the cluster membership is stable.

## $k$-means partitional clustering

The $k$-means is a partitional clustering technique that tries to find a $k$ number of clusters, the count is specified by the user. These are represented by their centroids. It is the simplest and the most commonly used algorithm that uses the *squared error* criterion. The $k$-means algorithm starts with a random initial partition and keeps reassigning the patterns to clusters based on the similarity between the pattern and the cluster centers (centroid distances) until a convergence condition is reached. In the process of clustering, there is no reassignment of any pattern from one cluster to another, which gives it a property of linear time complexity. In other words, the squared error decreases to some minimum threshold after some number of iterations.

Following are the major advantages of $k$-means algorithm. (1) It is easy to implement, (2) its time complexity is $O(n)$, where $n$ is the number of patterns. However, its

**Fig. 17.9** The *k*-means
clustering is sensitive to
initial partition
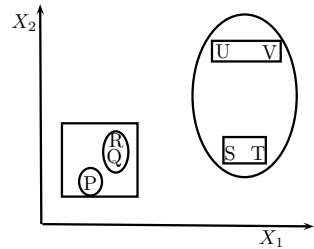
disadvantage is that it is sensitive to selection of the initial partition—if not properly
selected it may converge to a *local minima* of the criterion function value.

   The following example demonstrates creation of clusters based on the *k*-means
algorithm.

**Example 17.4**  Using the *k*-means approach to perform partitioning.

Figure 17.9 shows 2D patterns $P$, $Q$, $R$, $S$, $T$, $U$, $and V$. The process is started with
initial patterns $P$, $Q$, and $R$. Around these, three ($k = 3$) clusters are to be con-
structed. We end up with the partition $\{\{P\}, \{Q, R\}, \{S, T, U, V\}\}$, where three clus-
ters are shown by ellipses. The squared error criterion value turns out to be much
larger for this partition. This will happen, for example, for the centroid versus the
patterns in the largest ellipse. Hence, we construct a better partition $\{\{P, Q, R\},
\{S, T\}, \{U, V\}\}$, where clusters are shown by rectangles. This grouping results in the
global minimum value of the squared error criterion function, for clustering com-
prising of $k = 3$ clusters. The correct three-cluster solution is obtained by choosing,
for example, $P$, $S$, and $U$ as the initial cluster means, which will form the partition
as $\{\{P, Q, R\}, \{S, T\}, \{U, V\}\}$ [7].                                                                    □

## 17.8  Comparison of Clustering Techniques

A collection of all the clusters corresponding to a given input data set is gener-
ally called as clustering. In the following discussions, we shall distinguish between
various types of clustering approaches such as partitional (i.e., un-nested) versus
hierarchical (i.e., nested), monothetic versus polythetic, and hard versus fuzzy [7].

Partitional versus Hierarchical

At the top level, distinction is made between hierarchical and partitional approaches
for clustering, where hierarchical-based methods produce a nested series of partitions.
However, the partitional (un-nested) methods produce only one partition. In the
hierarchical clustering, the features are used sequentially, whereas in the creation of
partitional clustering they are used simultaneously.

   The hierarchical clustering algorithms produce a sequence of clusters, which
are nested in nature. They have a single all-inclusive cluster at the top, and single
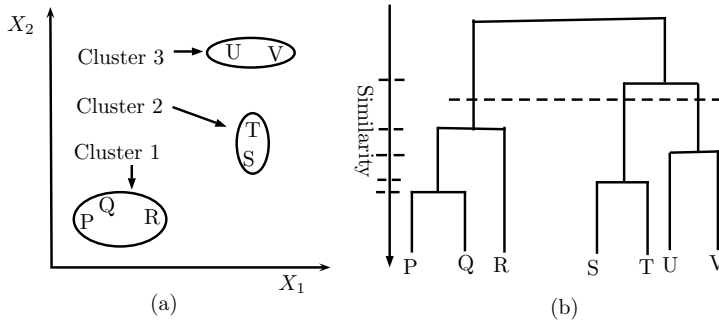
**Fig. 17.10**   **a** Data items in three clusters, **b** dendogram for clusters in (**a**)

point clusters at the bottom of the hierarchy. At the start, the algorithms, called *Agglomerative* hierarchical algorithms, treat each data point as a separate cluster. After this initial step, each further step merges two clusters that are most similar. For demonstration of this, see Fig. 17.10b, from bottom towards top. For total $n$ number of data points at the beginning, each such step compares each node with each other node, hence the worst case analysis is $O(n^2)$. After each merger, the total number of clusters reduce by one. The user can repeat these steps until the desired number of clusters are obtained or the distance between two closet clusters goes above a certain threshold. Since in the worst case $n$ number of data points reduce to one cluster in total $n-1$ number of steps, the worst case time complexity of this algorithm is $O(n^3)$.

In some of the hierarchical methods, each cluster is represented by its *centroid*. A centroid of a cluster is a data point, which is closest to the centre of that cluster. In centroid-based methods, the distance between two clusters can be measured by how similar the centroids of these clusters are. However, the centroid-based scheme of distance calculation fails when some data points in a cluster are closer to the centroid of other clusters than the centroid of their own cluster.

The working of a hierarchical clustering algorithm is illustrated in Fig. 17.10a. It makes use of a 2D data set, in the form of seven patterns labeled as $P$, $Q$, $R$, $S$, $T$, $U$, and $V$. After merging these data sets, three clusters (1, 2, and 3) are obtained. A hierarchical algorithm results in a nested grouping of patterns, which produces a dendrogram, as illustrated in Fig. 17.10b. The similarity levels at which groupings change are also marked in Fig. 17.10b; any two clusters can also be merged to form a larger cluster based on the minimum distance criteria. In this example, clusters 2 and 3 can be merged to make a single bigger cluster.

A partitional clustering is created simply by dividing the set of input data objects into non-overlapping subsets at the output such that each data object is in exactly one subset, i.e., a cluster. A simple partitional (monothetic) clustering considers the features sequentially to divide the given collection of patterns (see Fig. 17.11).

**Fig. 17.11** Clustering based
on monothetic partitioning
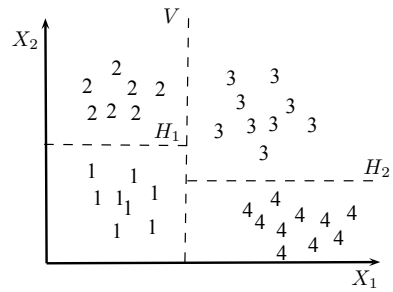


Monothetic versus Polythetic

This criteria is related to the sequential versus simultaneous use of features for performing the clustering. Most clustering algorithms are of the type *polythetic*, i.e., first, all the features enter into the algorithm before the computation of distances between patterns begin. Computation of distances, based on which clustering decisions are made, are based on all these features, and not any specific feature. However, a *monothetic* algorithm considers the features *sequentially* to divide the given collection of patterns into clusters.

Figure 17.11 illustrates the the construction of monothetic type of clusters. It shows that the collection is divided into two groups based on feature $x_{1,i} \in X_1$, which are separated by a broken vertical line $V$. These two clusters are independently further divided using the feature $x_{2,j} \in X_2$, indicated by broken lines $H_1$ and $H_2$. The major problem with this method is that, for a patterns' dimensionality of $d$, it generates $2^d$ clusters, which is exponential. For example, for even moderately large values of $d$, (say $d > 50$), which is typical in Information Retrieval (IR) applications, the number of clusters generated by this algorithm is $10^{15}$—a division of data set into uninterestingly small and fragmented clusters.

A monothetic class is defined in terms of features, such that these features are both sufficient as well as necessary to identify the members of that class. For example, people are clustered in age groups 0–25, 26–50, and greater than 50 years old. And, of course there may be many other features existing for each individual in each cluster, but one essential feature is the range of age, which is strictly followed in each group.

A broad set of criteria that are neither the compulsory requirement, nor are sufficient, are used to define a polythetic class. A certain minimal number of defining characteristics must be possessed by each member of the category, but it is not necessary that some feature must compulsorily be found in each member of the category. The distance between the members defines the membership in a class.

Incremental versus non-incremental

Incremental algorithms are useful when the pattern set to be used is very large, and the constraints imposed on execution time, or memory space, or both of these affect the nature of the algorithm. The traditional clustering algorithms were not designed to work with large data sets, hence the feature of scalability was absent. However, this feature is very much in need in most present day applications of large data sets.

The field of data mining has led to the development of clustering techniques that minimize the number of scans through a pattern space, reduce the number of patterns examined during execution, or reduce the data structures' size in the algorithm. The main advantage of incremental algorithms is that for them it is not required to store the entire pattern matrix in the memory, to be used again and again. But, instead, only part of that is stored at a time, which helps in reducing the total requirement of space. These algorithms are generally not iterative, hence the execution time is also small.

Algorithm 17.2 illustrates a typical incremental clustering algorithm. Let $D_0$ be the first data item, and $C_0$ be the first cluster.

---

**Algorithm 17.2** Incremental clustering Algorithm

---

1: $C_0 = \{D_0\}$
2: $d = 1$ ; data item counter
3: $k = 0$ ; cluster counter
4: **while** there is data-item available in input **do**
5:     Pickup the next data item $D_d$
6:     Compute distance of $D_d$ from all existing clusters' centroids
7:     Let $m$ is minimum distance, from centroid of cluster $C_m$
8:     **if** $m$ greater than threshold **then**
9:         $k = k + 1$ ; new cluster
10:         Create new cluster $C_k$
11:         $C_k = C_k \cup \{D_d\}$
12:     **else**
13:         $C_m = C_m \cup \{D_d\}$
14:     **end if**
15:     $d = d + 1$ ; next data-item
16: **end while**
17: **end**

---

Hard versus Fuzzy Clustering

A hard clustering algorithm assigns each input pattern to a single cluster in the output produced. However, a fuzzy clustering method may assign a degree of membership (belongingness) for each input pattern to several clusters in the output produced. A fuzzy clustering can be converted to a hard clustering by assigning each pattern to only one cluster, that which possesses the largest measure of membership of that pattern, and the partial membership of other clusters is ignored.

## 17.9   Classification

The basic idea of the data classification problem can be simply described as follows: given a training data with known labels or classes (e.g., as shown in Table 17.6), we would like to learn a model, so that it can be used to predict data with unknown labels. Let us consider that we have identified some customers through clustering

**Table 17.6**  Sample training database

| Record ID | Employment | Age | Salary | Group |
|-----------|------------|-----|--------|-------|
| 1 | Self | 30 | 30K | $C$ |
| 2 | Industry | 35 | 40K | $C$ |
| 3 | Self | 35 | 60K | $A$ |
| 4 | Self | 30 | 70K | $A$ |
| 5 | Industry | 35 | 40K | $C$ |
| 6 | Academia | 50 | 70K | $D$ |
| 7 | Self | 45 | 60K | $D$ |
| 8 | Academia | 30 | 70K | $B$ |
| 9 | Industry | 35 | 60K | $B$ |

of the aggregated purchase information about the currently existing customers for a certain company (see Table 17.2, page no. 520). Further, we have also acquired the mailing list of potential customers out of these, with their demographic information. As the next step, we would like to assign each person in the mailing list to one of the three groups: $A$, $B$, and $C$, as shown in Table 17.6. The latter is for the purpose of mailing them a catalog of items tailored to the individual's buying patterns. This task, data mining, makes use of historical information about current customers to help in the prediction of cluster membership of new customers.

Let us assume that the training database with historical information has records with attributes: ⟨salary, age, employment, and group⟩. The goal is to build a model that takes as input the *predictor attributes* and outputs a value for the *dependent attribute*. When the dependent attribute is a numerical value, the problem is called *regression*, otherwise it is a *classification* problem. In our present discussion the dependent attributes (also called *class labels*) are $A$, $B$, and $C$, hence it is a classification problem. The Table 17.6 is a sample training database with four predictor attributes: salary, age, and employment, and group as dependent attribute.

There are many classification models for data mining applications. These include the following: genetic algorithms, Bayesian networks, neural networks, log-linear methods, statistical methods, and decision tables. The classification trees (i.e., decision trees) are popular due to the following reasons:

- Their representation is intuitive, which makes classification model easy to understand,
- An analyst does not need to supply any input parameter for construction of decision trees,
- The accuracy of prediction of decision trees is better than other classification methods,
- It is possible to construct decision trees from very large training databases using algorithms that are scalable and fast.

Decision Trees

The decision trees are tree structures whose leaves are classifications and their branches are conjunctions of features that lead to classifications. Their significance is because many data mining methods generate decision trees. The problem solution methods learn these decision trees. One approach to learning a decision tree is to split the example set into subsets, based on the value test of some attribute. This process is repeated recursively on the subsets, with each split value becoming a sub-tree root. The splitting stops when the subset becomes so small that further splitting is not possible, i.e., the subset example contains only one classification. A split is considered best if it produces the minimum number of classification in the subset. That means, subsequent learning generates smaller subtrees, which will require less further splitting, in effect reducing the number of steps for solution of Problem [8].

A decision tree algorithm used for the classification comprises two steps: tree building and pruning. In the first step, most of the decision tree is grown top-down in a greedy way. Starting with the root node, the database is examined by a method, called "split selection", that selects the split condition at each node. Then, the database is partitioned and the procedure is applied recursively. In the pruning stage, the tree constructed in the previous phase is pruned to control its size. The pruning methods select the tree in a way that minimizes prediction errors. In some cases, decision-tree construction algorithms separate the process of tree building and pruning, but other algorithms interleave these processes to avoid unnecessary expansion of some nodes.

Algorithm 17.3 shows a sample tree building phase, node $n$ where tree is to be split. At the output, the algorithm provides a decision tree for the data partition $D$, and new node value $n$ which is the root of the decision tree.

---

**Algorithm 17.3** Sample code for Tree building

---

1: % Input: Data-partition $D$, Node $n$, split selection criteria $P$
2: % Output: Decision tree for $D$, with it root as $n$
3: *Build-Tree*($n$, $D$, $P$)
4: Apply $P$ to $D$, and find splitting criteria for node $n$
5: **if** $n$ splits **then**
6:    Create its child nodes $n_1$, $n_2$
7:    *partition D* into $D_1$, $D_2$ using split criteria
8:    *Build-Tree*($n_1$, $D_1$, $P$)
9:    *Build-tree*($n_2$, $D_2$, $P$)
10: **end if**

---

The choice of splitting criteria determines the equality of the decision tree. If the training database does not fit into memory, we need a scalable data access method. Many scalable algorithms are designed with a built-in feature, which ensures that only a small set of statistics, like aggregate measures and counts, be sufficient to implement the split selection method. Since aggregated data is far smaller in size than the actual data, it is possible to construct the statistics in memory for each node in a single scan over the corresponding database partition. These nodes satisfy the

Fig. 17.12 Sample decision
tree for catalog mailing
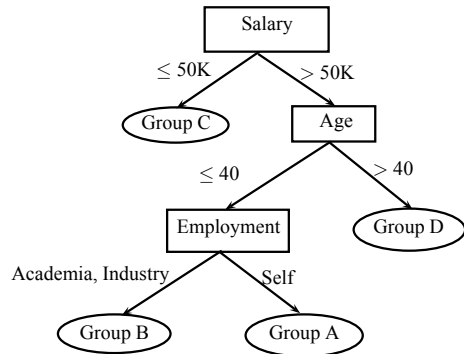


splitting criteria, such that in the process of recursively splitting we ultimately reach the class label node at leaf. In a decision tree, each internal node is labeled with a prediction attribute, called *splitting attribute*, and each leaf node is labeled with a *class label*.

**Example 17.5** Figure 17.12 shows the decision tree for a training data set as shown in Table 17.6. The splitting attributes are salary, age, and employment, and the class labels are Groups $A$, $B$, $C$, $and D$.

Every edge that originates from an internal node is labeled with a *splitting predicate*, which involves only the node's splitting attribute. The splitting predicate in the decision tree (Fig. 17.12) are $\leq 50K$, $>50K$, $\leq 40$, $>40$, *Self*, *Academia, and Industry*. The splitting predicate has a property that any record will take a unique path from the root to exactly one leaf node, that is the class label of that record. At a node, the combined information about splitting attributes and splitting predicates is called the *splitting criterion*. □

## 17.10 Association Rule Mining

Association rules are a set of significant correlations, frequent patterns, associations, or causal structures from data sets found in various types of databases. Such databases are transactional databases, relational databases, and other forms of data repositories. Mining of association rules is capturing those correlations, patterns, and rules and representing them in the form of some *if ...then* rules. For example, given a set of transactions, each transaction comprising a set of items, an *association rule* can be an implication, $X \Rightarrow Y$, where $X$ and $Y$ are sets of items, indicating that presence of itemset $X$ implies the itemset $Y$. Consider that an insurance company finds a strong correlation between two sets of policies $X$ and $Y$ of the form $X \Rightarrow Y$, which may be an indicator that customers holding policy set $X$ were also likely to hold policy set $Y$, where $X$ and $Y$ may have one or more elements. Now the company

could more effectively target marketing the policy $Y$ through those clients who hold policy $X$ but not $Y$, to motivate them to buy policy $Y$. In effect, the association rule represents the knowledge about purchasing behavior of customers. The association rule mining has been effectively applied to many different domains that include *market-basket analysis* in commercial environments, astrophysics, crime prevention, fluid dynamics, and counter terrorism, i.e., all the areas in which a relationship between objects can be concluded as useful knowledge [5, 9].

Association rules analysis is a technique to uncover how items are associated with each other. There are three commonly used ways to measure the association, which indicate the strength of the association.

**Definition 17.9** (*Support*) For a given set of data items, an association rule has *support s* for some set of items $X$ if $s$ percent of transactions include all the items of set $X$.                                                                                  ☐

In the sales transactions, for example, if we discover that sale of some items beyond a certain proportion tend to have a significant impact on the total profits, we might consider using that proportion as *support threshold*.

**Definition 17.10** (*Confidence*) "Confidence" says how likely the itemset $Y$ is purchased when itemset $X$ is purchased, i.e., to determine the value of $X \rightarrow Y$. For a given set of data items, an association rule has *confidence c* if $c$ percent of transactions that contain itemset $X$ also contain itemset $Y$.                                           ☐

"Confidence" is measured by the proportion of transactions with item $X$, in which $Y$ also appears. For the total number of transactions $T$, we compute $c$ as

$$
\begin{aligned}
c &= \frac{(X, Y)}{X} \\
&= \frac{(X, Y)/T}{X/T} \\
&= \frac{support(X, Y)}{support(X)}.
\end{aligned}
$$

In association rule mining, usually the goal is to discover all association rules having the value of support and confidence greater than some user-specified minimum threshold value [8].

**Definition 17.11** (*Lift*) "Lift" ($l$) says how much it is likely that the itemset $Y$ will be purchased whenever itemset $X$ is purchased, while controlling "lift" for how popular the item $Y$ is.

The lift is measured as

$$
l = \frac{support(X, Y)}{support(X) \times support(Y)}. \tag{17.8}
$$

A lift value greater than 1 means item $Y$ is likely to be bought if $X$ is bought, while its value less than 1 means $Y$ is unlikely to be bought if $X$ is bought.

**Table 17.7**  Transactional database

| Tran. ID | Cust. ID | Item name | Price (in $) | Date |
|----------|----------|-----------|--------------|------|
| 101 | 201 | *Laptop* | 1500 | 8/20/2018 |
| 101 | 201 | *Tablet* | 300 | 8/20/2018 |
| 101 | 201 | *Smartphone* | 100 | 8/20/2018 |
| 102 | 201 | *Music system* | 500 | 8/25/2018 |
| 102 | 201 | *Smartphone* | 100 | 8/25/2018 |
| 103 | 202 | *Laptop* | 1500 | 8/30/2018 |
| 103 | 202 | *Music system* | 500 | 8/30/2018 |
| 103 | 202 | *Smartphone* | 100 | 8/30/2018 |

**Example 17.6**  Given the sales transactions in Table 17.7, find out the "support" for the following:

1. Laptop,
2. Smartphone, and
3. Laptop and Smartphone

and "confidence" of a music system with respect to

1. Laptop,
2. Smartphone, and
3. Laptop and Smartphone.

There are three transactions in this table: numbers 101, 102, and 103, stored in a relational database system. We note that out of the three transactions, two have Laptop, three have Smartphone, and two have Laptop and Smartphone combined. Accordingly, their "support" is 67, 100, and 67 percent, respectively.

In the second part, we are interested to compute confidence of "Laptop" → "Music system", i.e.,

$$c = \frac{support(laptop,\ music\ system)}{support(laptop)}$$
$$= \frac{1/3}{2/3}$$
$$= 50\%.$$

The "lift" for the above is computed as follows.

$$c = \frac{support(laptop,\ music\ system)}{support(laptop) \times support(music\ system)}$$
$$= \frac{1/3}{2/3 \times 2/3}$$
$$= 0.75.$$

The *lift* of 0.75 ($< 1$) indicates that item $Y$ is unlikely to be bought by a customer who is buying item $X$.                                                                                    □

Most of the algorithms for association rules mining comprise two distinct steps:

1. *Find all sets of items with minimum support.* The data are of the order of millions of transactions, and a mining algorithm needs to count a large number of candidate itemsets to identify the frequent ones; hence this phase is computationally expensive, and often time consuming.
2. *Generation of inferences.* The inferences or rules can be generated directly from the frequent itemsets, and there is no need to scan the data sets again.

In the above two steps, most of the time is usually consumed by the first step, hence the scalability becomes more important. The techniques for scalability can be divided into two groups: 1) those techniques which reduce the total number of candidates to be counted; and 2) those which make candidates' counting more efficient. The first category of techniques make use of the property—subsets of a frequent itemset must also be frequent, and uses this concept as a pruning technique to reduce the number of itemsets to be counted.

Variations of actual problem

The other approaches used in the data mining algorithms are focused on variations of the actual problem. For example, for data sets and *support* values where the frequent itemsets are very long—with $n$ items there are $2^n$ possible frequent subsets—this makes finding all frequent itemsets an intractable problem. However, by looking ahead, the set of maximal frequent itemsets can still be found efficiently. We can make use of the following criteria for this: if an itemset is identified as infrequent, its subsets are also infrequent, and hence none of its subsets need to be counted. A key approach to this solution is to maximize the probability that itemsets counted due to looking ahead are actually frequent, which requires estimation of some heuristics. For this, a good heuristic can be to bias candidates' generation such that most frequent items appear in most of the candidate groups. This heuristic will make it more likely for high-frequency items to be part of long and frequent itemsets.

Why at all the finding of frequent itemsets is a nontrivial problem? The first reason is the number of customer transactions in the database, which can be so large that it usually does not fit into the memory of a system. The second is the potential number of frequent itemsets is exponential on the number of different items, although the actual number of different itemsets can be much smaller. If a table has four different itemsets, there can be $2^4 - 1 = 15$ potential frequent itemsets. But, if minimum support is taken as 60%, only five itemsets are actually frequent. Thus, there is need of algorithms that are scalable with respect to the number of transactions, such that they need to examine as few infrequent itemsets as possible.

Nested Hash Tables

This is a new and efficient technique, which makes use of hash tables in a nested manner to check as to which itemsets are contained in a transaction. This approach is more effective when it is required to count shorter itemsets. A technique for longer itemsets is *database projection* technique, given as Algorithm 17.4.

---

**Algorithm 17.4** Database Projection algorithm for counting itemsets

---
1: Partition the candidate itemset into groups such that candidates in each group share a set of common items.
2: Discards transactions $T_d$, that do not include all the common items, from $T$. ($T =$ all transactions)

3: Discards the common items from remaining set ($T - T_d$) (since they are known to be present.)
4: Discard the items not present in any of the candidates.
5: Count each of the candidate group.

---

This reduction in the number and size of the remaining transactions can result in substantial improvements in the speed of counting.

## 17.11   Sequential Pattern Mining Algorithms

Sequential pattern mining is aimed at discovering patterns of frequent subsequences in a sequence database. This database usually comprises a large number of sequences of ordered events, with or without a notion of time; each such subsequence is considered as a database record. Such sequences commonly occur in any metric space, either as a *total ordering* or *partial ordering*. There can be many examples of such sequences: events in time, in codons, or nucleotide in amino acid, in computer networks, in website traversals, or characters in a text string are all examples, where existence of a sequence may be significant, and the detection of frequent subsequences might be useful. *Sequential pattern mining* has emerged as a technology to discover such subsequences [10].

The sequential pattern mining problem may be defined as follows: given a database of sequences, such that each sequence is a list of transactions ordered by transaction time, find out all sequential patterns with a user-specified *minimum support* value. Here, a transaction consists of a set of items. The support of a pattern is the number of data sequences that contain the pattern as a fraction of the total number of transactions [11]. This parameter indicates a *minimum* number of sequences in which a pattern must appear to be considered frequent. For example, if a user sets the minimum support threshold to 2 sequences, the task of sequential pattern mining consists of finding all subsequences appearing in at least 2 sequences of the input database.

### 17.11.1  Problem Statement

A sequential pattern mining problem can be stated using the following examples.

- Can we find out some strong correlations between users doing online purchasing from a web page based on their behavior patterns versus the sequences of pages visited and/or speed at which he/she browses a website pages?
- A *market basket* is a record with fields: customer ID, and list of all items, with most of the item fields as False, and True only where the customer has bought those items. Can we derive through analysis, more useful information about the buying pattern of the a customer, if information related to time and sequence is also included in the market basket?
- Can we recognize a user as a genuine user or hacker based on the suspicious behavior observed due to analyzing a sequence of commands entered by that user?
- Can we declare the elements of actions as "best practices", based on the outcomes and analysis of sequence of actions performed on any activity that results in the given outcomes?
- Given the sequences of alerts and status by any system before it has failed, can we determine those sequences set or subsequence set through some analysis which are confirmatory predictions about the failure? That is, whether it is possible to map those sequences to failure?

In simple terms, having many significant or important events occurring over the scale of time, space, or some other metric, it is required to learn from the data by considering the ordered sequences appearing in the data.

### 17.11.2  Notations for Sequential Pattern Mining

We want to have a formally defined notation to describe the problem of mining sequential patterns. We base our discussions on *market basket*, consisting of customer ID and list of items. Let the set of items in the form of *literals* be $\mathscr{I} = \{i_1, i_2, \ldots, i_m\}$. A non-empty unordered collection of items $\alpha = (i_1, i_2, \ldots, i_k)$ is an *event*. However, for ease of processing, and without any loss of generality, we assume them to be in lexicographic order. A *sequence* $(\alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_l)$ is an ordered list of events, and a sequence with $k$-elements, where $k = \sum_j |\alpha_j|$ is a $k$-sequence, i.e., the total sum of events in that sequence. For example, $(A \rightarrow BC)$ is a 3-sequence, as there are three events $A$, $B$, $C$ in this sequence. A sequence $\langle \alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_m \rangle$ is called as a *subsequence* of another sequence $\langle \beta_1 \rightarrow \beta_2 \ldots \rightarrow \beta_n \rangle$, if there exist integers $i_1 < i_2 < \cdots < i_m$ such that $\alpha_1 \subseteq \beta_{i_1}, \alpha_2 \subseteq \beta_{i_2}, \ldots, \alpha_m \subseteq \beta_{i_m}$, and $m \leq n$. For example, the sequence $\langle A \rightarrow BC \rangle$ is a subsequence of $\langle AE \rightarrow F \rightarrow BCD \rangle$, since $A \subseteq AE$ and $BC \subseteq BCD$. Note that the order of events is also preserved in this case.

Consider that input sequences are stored in a database $\mathcal{D}$, and input sequence is an event with fields: *sequence-id*, *event time*, and the *list of items*. It is assumed that event time is unique in every event, hence this field can be used as the event ID. We define the *support* (the frequency) of a sequence, as the number of input sequences in the database that contains the sequence $\alpha_s$ and express it as $\sigma(\alpha, \mathcal{D})$.

The support can also calculated by a different method. For user-specified minimum support value $min\_supp$ (a frequency), a sequence is frequent if the sequence occurs more than $min\_supp$ times. We denote the frequent $k$-sequences as $\mathscr{F}_k$. In addition, a frequent sequence is called as *maximal* if it is not a subsequence of any other frequent sequence. Further, a frequent sequence is called *maximal* if it is not a subsequence of any other frequent sequence.

The above discussion can be termed as the problem definition for sequence mining algorithms, whose data are transaction database or they are transaction data sets.

## 17.11.3  *Typical Sequential Pattern Mining*

We consider an example of customer's retail transactions or purchase sequences in a store, which shows for each customer the store items he/she purchased in every week for a month's duration. Out of these sequences of customer purchases, each sequence can be represented using a schema $[TransactionID/CustomerID,$ $\langle$ordered sequence of events$\rangle]$, where each sequence may be a set of store items like *bread*, *butter*, *milk*, *vegetable*, etc. Considering only two customers, a sequence in the database may be $[T1, \langle(bread, butter), (bread, vegetable, butter), (bread),$ $(coffee, milk)\rangle]; [T2, \langle(milk), (coffee, bread), (vegetbable, milk)\rangle]$. We note that customer having transaction ID $T1$ made a purchase in each of the 4 weeks in the month, while the customer with transaction ID $T2$ made purchases during 3 weeks. There can be one or more items purchased in each of an event, i.e., during each market visit. Hence, different sequences can have different lengths ($T1$ has four and $T2$ has three), and each event in a sequence can have one or more items in its set.

Sequence databases are mined using a sequential pattern mining algorithm, which looks for repeated patterns, called *frequent sequences*. These patterns can be later used to find associations between different items or events in their data, with end use like marketing campaigns, reorganization of business, prediction, planning, etc. Among the most common applications of sequential pattern mining, are web-based applications, making use of WWW for e-commerce, business, trading, etc. Typical applications of web usage mining are the areas, like user modeling such as web content personalization, prefetching and catching, reorganization of website, e-commerce, and business intelligence.

*Web usage mining* is an application of sequential pattern mining, which confines to finding user's navigational patterns on WWW. These patterns are extracted from web logs. Against the market-basket concept, where an event comprises the number of items, the ordered events in web mining are each comprising only one item. This is because a web user can access only one page at a time, and not many items in

an event. If, instead of an event time, a *time window* is considered for web mining, then there can be more than one item (multiple pages), and in that case it becomes a general case like market basket, having many items in an event, where an event is in the form of a time window.

**Example 17.7** Sequential pattern mining to raise promotional sale in a online store.

As discussed above, when a user is accessing web pages, such that only one page is accessed at a time, it gives rise to a sequence database, which is equal to only one item in each sequence's ordered event list.

Let $E = \{a, b, c, d, e, f\}$ be a set of events for six products' web pages accessed by different users in different order in some online sales application. The $E$ also indicates a set of six products.

We assume that web accesses database is web log, and for four users the web access sequence is in the form of four records: $[T1, \langle adbce \rangle]$; $[T2, \langle abecdf \rangle]$; $[T3, \langle babfaec \rangle]$; $[T4, \langle abfac \rangle]$. From the web-log pattern mining of this web sequence database, we find a frequent sequence *abc*, indicating that all the users who visit product *a*'s web page http://www.prompt-sale.com/a.htm also immediately visit product *b*'s web page http://www.prompt-sale.com/b.htm and then visit product *c*'s page http://www.prompt-sale.com/c.htm (These are only ficticious names of portals).

Having the patterns sequence of visiting the web pages in a certain order, e.g., the customers who visit the web page for product *a* will also visit web page *b* and *c*, it is possible to increase the sales of product *b* and *c*. By placing promotional discounts on product *a*'s web page, which is visited a number of times in sequence, it is possible to increase the sale of other products *b*, and *c*.                                □

The web log can be maintained on the server where the web page is stored. However, it can also be stored on the client side, or on a proxy server. Each choice has its own advantages and disadvantages concerning finding the users' relevant patterns and navigational sessions.

### 17.11.4   Apriori-Based Algorithm

Apriori algorithm is used for mining of frequent itemsets, and for *association rule* learning over transactional databases. The algorithm works based on the identification of frequent individual items in the database and extending them to larger and larger itemsets, as long as those itemsets appear frequently in the database. Apriori has the property that "any subset of a frequent itemset must also be frequent." This algorithm makes several rounds of computation to compute frequent itemsets, such that in the *i*th round it computes all frequent *i*-itemsets. A round in the Apriori algorithm has the following three steps:

1. candidate generation,
2. candidate counting, and

3. discarding unimportant candidates.

In the $i$th round of the candidate generation step, it generates a set of candidates with $i$ itemsets. Then, in the candidate counting step, it scans the transaction database and counts the support of candidate itemsets. In the third step, candidates with support lower than a user-specified minimum threshold are discarded, and frequent $i$ itemsets are retained [10].

In the first round, the *Apriori* algorithm generates a set of candidate itemsets each containing all 1 itemsets, and counts their support value. Therefore, after the first round, all frequent 1 itemsets are known. Similarly, the candidate itemsets generated during the candidate generation step of round two are roughly all pairs of items. Apriori algorithm reduces the set of candidate itemsets through pruning—a priori, based on the knowledge about infrequent itemsets obtained from the previous rounds. Those are the itemsets which cannot be frequent. The concept of pruning used here is based on the general rule that "if an itemset is frequent", all its subsets must also be frequent. Hence, before we go to the candidates' counting step, it is better to discard every candidate itemset whose subset is infrequent [11].

The *Apriori*-based algorithm is presented as Algorithm 17.5. The task of the algorithm is to find the set $F$ of frequent sequences (also known as frequent sequential patterns), for a given minimum support threshold $min\_supp$.

---

**Algorithm 17.5** Apriori-based algorithm

---

1: Start scanning of database in Apriori-based algorithm at $k = 1$
2: Let $C_1$ = initial sequence generated in 1st generation (set of *candidate* 1-*sequences*)
3: **repeat**
4:   $k = k + 1$ (next step)
5:   Scan the entire database (In $k$th-iteration it finds frequent itemsets of size $k$)
6:   Join $F_{k-1}$ frequent itemsets with itself to generate $C_k$ (set of candidate sequences in the $k$th-iteration).
7:   Prune sequences in $C_k$ which have no subsequences in $F_{k-1}$ (i.e., they are not large)
8:   Create frequent itemsets $F_k$ by adding all sequences from $C_k$ with support $\geq min\_supp$
9: **until** no more candidate sequences left
10: **end**

---

The scanning of the database in an Apriori-based algorithm starts at $k = 1$. Let us call the initial sequence generated in the first generation as $C_1$, the set of candidate 1-sequences (in $k$th generation as $C_k$). Subsequently, the database is scanned by the algorithm several times. Let, in $k$th-iteration it find frequent itemsets $F_k$, of size $k$ ($k \geq 2$). Next, the algorithm performs Apriori-generate and join ($\bowtie_{ap-gen}$ $F_1$), to join $k - 1$ frequent itemsets in $F_{k-1}$ with itself to generate $C_k$, the set of candidate sequences in the $k$th-iteration. It then prunes sequences in $C_k$ which have no subsequences in $F_{k-1}$ (i.e., which are not large), and creates $F_k$ by adding all sequences from $C_k$ with support at least equal to $min\_supp$. The above iteration process goes on for $k = 1, 2, \ldots$ until there are no more candidate sequences left.

**Example 17.8** Illustrating iteration steps in the Apriori algorithm.

Let the total number of items after scanning in the fist generation be 5, i.e., $C_1 = \{a : 6, b : 6, c : 3, d : 3, e : 5\}$, here every item has been represented with its support in the form $s : n$, where $s$ is the sequence, and $n$ is the support count. Let us assume that a minimum absolute support is $min\_supp = 4$ (i.e., 67%). After pruning is over, the list of frequent 1-sequences is $F_1 = \{a : 6, b : 6, e : 5\}$. Using the join operation ($\bowtie$), the Apriori has generated candidate sequences at $k = 2$ as, $C_2 = F_1 \bowtie_{ap-gen} F_1 = \{aa : 3, ab : 5, ae : 4, ba : 3, bb : 4, be : 4, ea : 1, eb : 2, ee : 0\}$. Here, $\bowtie_{ap-gen}$ stands for Apriori "generate and set-join operation." Note that sequences like $ac, ad$ are not considered as their subsequences: $a, c$ and $a, d$ are each pair not frequent, and similar rules applied to the others cases. The support count in the sequences of $C_2$ is as per their frequency in the database.

Next, it prunes all sequences in $C_2$ that do not have frequent subsequences (i.e., their subsequences do not appear in $F_1$), and so on. The final resulting frequent set is $f_s = U_k F_k = \{a, b, e, ab, ae, bb, be, aba\}$, where $U_k$ is the union of $k$ number of frequent sequences of $F_k$. The pruning procedure performed on itemsets in $C_k$ removes sequences that have infrequent subsequences.  □

**Example 17.9** Perform data mining using Apriori-based algorithm to find out the frequent itemsets for the sales transaction data, for a minimum support of 60%.

Consider the transactional database shown in Table 17.7 (page number 539); for convenience of reference it is reproduced as Table 17.8.

For $min\_supp$ value of 60%, an itemset is frequent if it is present in at least two transactions. As per the Apriori algorithm, we perform many rounds through the given database. In the first round all single items, which are candidate itemsets, are counted during the candidate *counting step*. The frequent itemsets resulting in 1-sequence itemset are $\{Laptop(67\%), \text{Smartphone}(100\%), \text{Music system}(67\%)\}$.

In the second round, the pairs are counted as frequent itemsets, and only those pairs are candidates in which the individual item is frequent in round one, and they are present in at least 60% of the transactions. As per these restrictions, the 2-sequence

**Table 17.8** Transactional database

| Tran. ID | Cust. ID | Item name | Price (in $) | Date |
|---|---|---|---|---|
| 101 | 201 | *Laptop* | 1500 | 8/20/2018 |
| 101 | 201 | *Tablet* | 300 | 8/20/2018 |
| 101 | 201 | *Smartphone* | 100 | 8/20/2018 |
| 102 | 201 | *Music system* | 500 | 8/25/2018 |
| 102 | 201 | *Smartphone* | 100 | 8/25/2018 |
| 103 | 202 | *Laptop* | 1500 | 8/30/2018 |
| 103 | 202 | *Music system* | 500 | 8/30/2018 |
| 103 | 202 | *Smartphone* | 100 | 8/30/2018 |

itemsets are *{Laptop, Smartphone}*(67%), *{Music system, Smartphone}*(67%). Note that the pairs in which *{Tablet}* is an item are dropped, as it is not a frequent item in the first round. However, it disqualifies as a pair also due to minimum support.

In round three, no itemset qualifies for 3-sequence. Hence, with respect to a minimum support of 60%, the frequent itemsets in this sample database and their support values are

$\{Laptop\}$ 67%,
$\{Music system\}$ 67%,
$\{smartphone\}$ 100%,
$\{Laptop, smartphone\}$ 67%, and
$\{Music system, smartphone\}$ 67%.

The Apriori algorithm counts the support of all frequent itemsets, as well of those infrequent candidate itemsets that could not be eliminated during the pruning step. These latter items sets are called *negative border*. Note that an item is in negative border if it is infrequent, but its subsets are frequent. In the example above, the only negative border itemset is {*Laptop, Music system*}. It is necessary that all subsets of an itemset in the negative border are frequent, in the absence of that the item should have been eliminated by the subset pruning step.                                                □

We know that the total number of subsequences possible for $n$ items is $2^n$, which is exponential. Accordingly, as the number of sequences in the database becomes larger, the Apriori techniques suffer exponential growth of candidate sequences during execution, and consequently, suffer increased delays in mining.

The Apriori is a family of algorithms; they are better suited for discovering intra-transaction associations and then to generate rules about the discovered associations. However, the task of mining the sequences is actually discovering inter-transaction associations, i.e., sequential patterns across the same or similar data. Hence, the algorithm uses transactional databases as its data source.

One form of the Apriori algorithm has *horizontal formatting*, i.e., the original data is sorted first by Customer_Id and then by Transaction-time, so that each customer's transactions appear together, but in the order of time stamp of the transaction. This transformation results to the database, where the time stamps determine the order of events. The mining is then performed on these database using the breadth-first search (BFS) approach. Following are the steps of Apriori-based algorithm using horizontal formatting.

1. *Sort phase.* This phase transforms the data set from the original database into customer sequence database by sorting on the primary key as Customer_Id, secondary key as Transaction_Time.
2. L_itemset phase. This phase finds all *large itemsets L* (which meet the minimum support).
3. *Transformation phase.* As there is a repeated requirement to find out which of the long sequences are present in the customer sequence, each customer's sequence is transformed by replacing the corresponding transaction with the set of L_itemsets contained in that transaction. The transactions without any L_itemsets

are dropped, as well as the customer sequences not containing any L_itemsets, but these customer sequences are still considered as part of database for the purpose of customers' count.

4. *Sequence phase.* The sequence phase mines the set of L_itemsets to find the frequent subsequences. The algorithm make multiple passes over the data, with each pass beginning with a *seed-set*, for producing potentially large sequences (i.e., candidates), and these candidates' support is also calculated during this pass. The sequences not meeting the minimum support criteria are pruned, and the remaining become the seed-set for the next pass. The process starts with the large 1-sequences, and terminates when either no candidates are generated or none meet the minimum support.

5. *Maximal Phase.* The phase finds all maximal sequences among the set of large sequences. The process is similar to finding all subsets of a given itemset, the algorithm is similar as well.

**Example 17.10** (*Association rule mining*) Consider the database having four transactions as shown in Table 17.9. It is required to find the association rules if the minimum support is 60% and minimum confidence is 80%.

We perform the following steps for the solution.

1. We note that item $a$ appears in four transaction, hence, it has support of 100. Accordingly, the items with frequency and support are $(a, 4, 100\%)$, $(b, 4, 100\%)$, $(c, 2, 50\%)$, $(d, 3, 75\%)$, $(e, 2, 50\%)$, and $(f, 1, 25\%)$.

2. Next, construct the itemsets having two items using the previous phase with support $\geq 60\%$: $(a, b, 4, 100\%)$, $(a, d, 3, 75\%)$, and $(b, d, 3, 75\%)$.

3. Next, construct the itemsets having three items using the previous phase with support $\geq 60\%$ or more : $(a, b, d, 3, 75\%)$.

Next, we formulate rules and compute their confidence. For this, we take the items from the previous phases with confidence at least 60% (see Table 17.10).

The rules with confidence less than 80% are pruned, and we are left with rules shown in Table 17.11.

**Table 17.9** Transactions database

| Transaction_Id | Itemset |
|---|---|
| $T_1$ | $\{a, b, d, f\}$ |
| $T_2$ | $\{a, b, c, d, e\}$ |
| $T_3$ | $\{a, b, c, e\}$ |
| $T_4$ | $\{a, b, d\}$ |

**Table 17.10** Rules with confidence $\geq 60\%$

| Rule | Confidence $c$ (%) |
|---|---|
| $a \rightarrow b = P(b\|a) = \frac{\|b \cap a\|}{\|a\|} = \frac{4}{4}$ | 100 |
| $b \rightarrow a = P(a\|b) = \frac{\|a \cap b\|}{\|b\|} = \frac{4}{4}$ | 100 |
| $a \rightarrow d = P(d\|a) = \frac{\|d \cap a\|}{\|a\|} = \frac{3}{4}$ | 75 |
| $d \rightarrow a = P(a\|d) = \frac{\|a \cap d\|}{\|d\|} = \frac{3}{3}$ | 100 |
| $b \rightarrow d = P(d\|b) = \frac{\|d \cap b\|}{\|b\|} = \frac{3}{4}$ | 75 |
| $d \rightarrow b = P(b\|d) = \frac{\|b \cap d\|}{\|d\|} = \frac{3}{3}$ | 100 |
| $ab \rightarrow d = P(d\|ab) = \frac{\|d \cap ab\|}{\|ab\|} = \frac{3}{4}$ | 75 |
| $d \rightarrow ab = P(ab\|d) = \frac{\|ab \cap d\|}{\|d\|} = \frac{3}{3}$ | 100 |
| $ad \rightarrow b = P(b\|ad) = \frac{\|ad \cap b\|}{\|ad\|} = \frac{3}{3}$ | 100 |
| $b \rightarrow ad = P(ad\|b) = \frac{\|b \cap ad\|}{\|b\|} = \frac{3}{4}$ | 75 |
| $bd \rightarrow a = P(a\|bd) = \frac{\|bd \cap a\|}{\|bd\|} = \frac{3}{2}$ | 100 |
| $a \rightarrow bd = P(bd\|a) = \frac{\|a \cap bd\|}{\|a\|} = \frac{3}{4}$ | 75 |

**Table 17.11** Rules with confidence $\geq 80\%$

| Rule | Confidence $c$ (%) |
|---|---|
| $a \rightarrow b = P(b\|a) = \frac{\|b \cap a\|}{\|a\|} = \frac{4}{4}$ | 100 |
| $b \rightarrow a = P(a\|b) = \frac{\|a \cap b\|}{\|b\|} = \frac{4}{4}$ | 100 |
| $d \rightarrow a = P(a\|d) = \frac{\|a \cap d\|}{\|d\|} = \frac{3}{3}$ | 100 |
| $d \rightarrow b = P(b\|d) = \frac{\|b \cap d\|}{\|d\|} = \frac{3}{3}$ | 100 |
| $d \rightarrow ab = P(ab\|d) = \frac{\|ab \cap d\|}{\|d\|} = \frac{3}{3}$ | 100 |
| $ad \rightarrow b = P(b\|ad) = \frac{\|ad \cap b\|}{\|ad\|} = \frac{3}{3}$ | 100 |
| $bd \rightarrow a = P(a\|bd) = \frac{\|bd \cap a\|}{\|bd\|} = \frac{3}{2}$ | 100 |

## 17.12   Scientific Applications in Data Mining

The scientific applications have resulted in the accumulation of high-dimensional data, data in the form of data streams, and temporal and spatial data. Following are potential scientific applications of data mining [12].

Biomedical Engineering

Examples of data in biological sciences include genome DNA sequence sequencing of organisms, as well as large macro molecules such as proteins, and RNA. Due to the large size availability of this data, there is need to create systems for organizing, storing, and dissemination. In addition, there is abundance of potential for automated learning from these data sets. A number of robust machine learning and data mining algorithms have emerged to take advantage of previous knowledge, and to create new knowledge. Consequently, biology is changing from the field of research which was dominated by—"formulation of hypothesis, conducting experiments, and evaluate results", to more of a computational science, with attitude of "collecting and

storing data, mining new hypotheses, and confirm these with data or supplement by experiment." These results/data can be combined with the clinical data to achieve better results and resolution for treatments.

Telecommunications

Due to the wide use of telecommunication in the world, vast quantities of high-quality data is available already, mainly including call details collected at network switches mainly for billing, which can be used for data mining tasks in detection of toll fraud and marketing to consumers. Based on the data mining algorithms on telecommunications data, the following benefits can be drawn:

- *New architectures for networks.* The new generation network infrastructures may adapt to changes in traffic demands dynamically, to be achieved through data mining techniques to understand and predict the network load.
- *Mobility and micro-billing.* Consumer-related activities may have separate billing patterns.
- *Mobile services.* Data mining may enable customers for adaptive solutions, so that they can get it through a few keystrokes.
- *Security.* Data may be collected and maintained through the records of billing, travels, migration, to obtain national security-related information, to ensure the national security.

Geospatial Data

The geographical data has scope and uses, which include digital data of all sorts, which can be created, stored, processed, and disseminated by government and private organizations, many of them through high-resolution remote sensing devices, data collected through geographical positioning systems, and other devices, which are position aware, like cellular phones.

The new pattern mining and clustering algorithms, which are highly scalable, are useful to discover new and unexpected patterns, trends, and relationships embedded in these data.

Climate Data and Earth's Ecosystems

The climate data acquired through the terrestrial observation, Earth-observation satellites, and ecosystem models provides lot of data, which can be mined for patterns to discover future ecological problems and their management, including ecology and even the health of the planet Earth. Such data consists of global snapshots of the Earth, at typical intervals, with variables like atmosphere, land, and ocean (sea surface temperature, precipitation), or accumulation of carbon.

There are two main components of Earth science data mining: (1) Modeling ecological data, and (2) Design of efficient algorithms for discovering potential patterns in these data. Through these patterns as well the existing patterns, it may be possible to predict the effects, such as El Nino and tornado, etc.

## 17.13 Summary

The digital storage capacity has doubled over every 9 months—this is the phenomena, which has necessitated the mining of data. Data mining provides tools, to sift through vast amount of data accumulated by organizations over the years, to find the trends, patterns, and correlations, which can be helpful in planning.

Data mining process is confronted with dimensionality, size, diversity, noise, and distributed nature of data sets, which makes the formal specification of the problem, a challenge. The solution techniques needs to deal with complexity, scalability, and presentation. The entire process through which data mining makes its transitions is called *data science*. The algorithms on data mining address three classical data mining problems, i.e., clustering, market bucket analysis, and density estimation.

One important difference between the traditional databases, and data mining is the size of data in later, such that data does not fit in the storage of computer, therefore, the mining algorithm should be scalable. Data mining is concerned with the identification of interesting data structure in the data, which may be patterns, statistical or predictive model, or relation information. A model can be used to predict future customer behaviors, e.g., the customer "A" may buy the so and products and goods.

The major goals of data mining are

- scaling analysis of large databases,
- scaling to higher dimensional and data models,
- automated search, and
- finding patterns and models understandable.

Five perspectives from the existing areas of machine learning (statistics, algorithms, and databases) are used in data mining. These comprise induction, compression, querying, approximation, and search.

The data mining techniques are also classified based on the following criteria:

- their *induced representations*, e.g., decision trees, association rules, and correlations;
- the *data* on which they operate, e.g., continuous, discrete, labeled, time series; or
- *application domains*, e.g., finance, economic models, web-log mining, and semi-structured models.

The common form of data for data mining is records of individuals, like transaction record or record in the form of requested web page—the set of such records can be viewed as *N rows × d column* matrix, and entry $(i, j)$ is 1, for example, in a transaction, if item $j$ was purchased in the session $i$. The itemsets $(d)$ in a row is called *basket*. In one approach to data mining, the objective is to find statistically significant itemset, that is, whose frequency is higher than some baseline frequency. The transaction data is often in the form of a continuous stream, like continuous sales transactions, or web logs. This gives a challenge on how to compute aggregate of the data. The *incremental learning* algorithms are used in such cases.

Many of the traditional algorithms access the databases multiple times, or randomly for analysis. This is not possible when databases are very large. The algorithm used for such large databases are *prediction methods, clustering*, and *association rules*. The clustering partitions a set of records according to some similarity function, into groups such that "similar" records are in the same group, an identification of similar subpopulations in the data. Clustered records are represented by a summarized feature called *cluster feature* (CF), which are efficient as they occupy less space. This grouping is sufficient for inter-cluster and intra-cluster measurements. A pattern **x** (which may be a feature vector, a datum, or observation) is a single data item used by the clustering algorithms. It is typically a vector of $d$ dimensions, represented as $\mathbf{x} = (x_1, \ldots, x_d)$. In the feature vector **x**, the individual scalar components $x_i$ are called features (or attributes).

Structured features can be represented using a tree data structure. The *feature selection* techniques identify a subset of the existing features to be used later, while *feature extraction* techniques compute new features from the original set. It is common to measure the dissimilarity between two patterns by measure of distance between these objects, say, $O_1$ and $O_2$, determined by $d(O_1, O_2)$.

The *association rules*, another method of classification, makes the use of market basket—collection of items purchased by a customer—finds correlation between itemsets, helping to understand the purchase behavior of customers. The clustering process groups the data objects using the information found in the data items as well as the relation between data items. Clustering is also called *unsupervised classification*, where it is required to group the given unlabeled patterns into meaningful clusters. The *supervised classification* classifies the unlabeled patterns into pre-classified patterns.

General applications of clustering are for pattern analysis, decision-making, and machine learning. Typical applications are Information Retrieval (IR), analysis of semantic information, analysis of business (sales) transactions, e.g., to find out potential customers, etc. The utilities like summarization, nearest neighbor, and compression can also be constructed based on techniques of data clustering. Typical pattern clustering activity involves—pattern representation, feature selection and extraction, pattern proximity measures, and data groupings (clustering).

Proximity or similarity measure is done by a distance function in Euclidean distance, where grouping can be based on many criterions, like soft (fuzzy) versus hard partitioning, hierarchical versus partitional algorithms, etc. Patterns for clustering algorithm may be represented using Cartesian coordinates or polar coordinates, and a pattern can measure either a physical object, like chairs, or an abstract notion, like style of writing. Different types of clustering are categories, which are distinguished as having contrast characteristics, e.g., monothetic *versus* polythetic, hierarchical (nested) *versus* partitional (un-nested), exclusive versus fuzzy, and complete versus partial. The hierarchical clustering algorithm yields nested grouping of patterns in the form of a *dendogram*, such that two clusters can be nested to form a larger cluster based on minimum distance criterion.

When the patterns set to be used is very large such that constraints on memory size and execution time affect the algorithm, we use the incremental algorithm. The latter has advantage that it does not require the entire matrix to be stored in the memory, hence space requirement is less. Since these algorithms are not iterative, the time required is also less.

Since proximity between items plays an intuitive role of clustering, the nearest neighbor approach is a useful basis of clustering procedures, which iteratively assigns each unlabeled pattern to the cluster of the nearest labeled pattern provided that the distance with that is below some prespecified threshold value. The partitional technique of clustering produce clusters by optimizing a criterion function defined on a subset of patterns or on all patterns. Its algorithm is run multiple number of times, and every time with different starting points, and the best outcome is considered from all the runs.

Many data mining approaches generate classification trees—a tree whose leaves are classifications and their branches are conjunctions of features that lead to those classifications. The decision-tree algorithm has two phases—tree building and tree pruning. In the tree building phase, the algorithm examines the database using the *split selection method* to compute the locally best criteria. This works recursively to build trees.

We can mine the rules in the data, called *association rules*. The rules mining algorithms capture the set of important correlations present in the database. For a given set of transactions where each transaction is a set of items, an associative (or association) rule can be an implication $X \Rightarrow Y$, e.g., in case of sales transactions, this may indicate that those who purchase itemset $X$ may also purchase itemset $Y$. Therefore, for selling itemset $Y$, one may target the people whose bought $X$, and not again who bought itemset $Y$. The association rule mining consists of two steps: (1) find all itemsets with minimum support, and then, (2) generate the inferences from these minimum support itemsets.

*Sequential pattern mining* technique can be described as given a database of sequences where each sequence is a list of transactions ordered by transaction time, discover all sequential patterns having a minimum support. A *sequence* in a sequential pattern mining is an ordered list of events, where an event is denoted as $\langle i_1, i_2, \ldots, i_k \rangle$, and $i_j$ is an item.

A sequential pattern mining algorithm mines the database of sequences, looking for repeating patterns (known as frequent sequences). These sequences can be analyzed to find associations between different items or events in their data for purposes, like marketing campaigns, business reorganization, prediction, and planning.

*Web usage mining* is an application of sequential pattern mining concerned with finding user navigational patterns on the World Wide Web, by extracting knowledge from web logs.

A commonly used family of algorithms, called Apriori algorithm, computes frequent itemsets through several rounds, such that in a round $i$ it computes all $i$-itemsets' frequent transactions. A round has two steps: 1) candidate generate, 2) candidate counting, i.e., those having support greater than $min\_supp$.

The scientific applications have resulted in accumulation of high-dimensional data, stream data, and spatial and temporal data. Hence, they are most appropriate fields for data mining algorithms. Some of the applications of data mining in scientific fields are Biomedical engineering, Telecommunication, Geospatial data, Climate data, and Earth ecosystems.

## Exercises

1. How is data mining different from querying databases, like Oracle or MySql?
2. What were the trends in Information Technology (IT), which gave birth to the field of data mining? Why did the field of data mining emerge so late compared to databases?
3. Are the goals presented in this chapter identically applicable to all the data mining domains, for example, mining of data generated due to collision in particle accelerators versus online sales transactions versus Twitter data? Justify your answer.
4. Given that the *Apriori algorithm* makes use of prior knowledge of subset support properties,

   a. Show that all non-empty subsets of a set of frequent items must also be frequent.
   b. Show that the support of any non-empty subset $R$ of itemset $S$ must be at least as large as the support of $S$.

5. The algorithms for frequent patterns mining consider only distinct items in a transaction (market basket or shopping basket). However, the multiple occurrences of an item are common in a shopping basket, e.g., we often buy the things like 10 eggs, 3 breads, 2 kg dalia, 4 kg oil, 2 kg milk, etc., and this can be important in transaction data analysis. Suggest an approach on how you will modify the Apriori algorithm, or propose alternate method to efficiently consider multiple occurrences of items?
6. Assume nonnegative prices of items in a store, and find out the nature of constraint they represent in each of the following cases. Also suggest, how you will mine the association rules in these.

   a. At least one Sudoku game.
   b. Itemsets, whose sum of prices is less than $250.
   c. There is one free item, and other items, whose sum of prices is equal or more than $300.
   d. The average price of all the items is between $200 and $500.

7. Given a decision tree, you have the following options:

   a. Convert the decision tree into rules, and then prune the resulting rules,
   b. Prune the decision tree and then convert the pruned tree into rules.

   Critically analyze both the approaches, and discuss their merits and demerits.

8. Find out the worst case time complexity of decision-tree algorithm. Assume that data set is $D$, each item has $n$ number of attributes, and the number of training tuples are $|D|$. (Hint. Answer is $|D|.n.\log|D|$.)

9. It is required to cluster a given set of data into three clusters, where $(x, y)$ represent the location of the object, and the distance function is Euclidean distance. The points are $P_1(3, 12)$, $P_2(3, 8)$, $P_3(9, 5)$, $Q_1(4, 7)$, $Q_2(6, 4)$, $Q_3(7, 5)$, $R_1(2, 3)$, and $R_2(5, 8)$. Use the $k$-means algorithm to show the three cluster centers after the first round of execution, and after the final round of execution.

# References

1. Fayyad U, Uthurysamy R (2002) Evolving data mining into solutions for insights. Commun ACM 45(8):28–31
2. Han J et al (1999) Constraint-based multidimensional data mining. Computer 4:46–50
3. Ramakrishnan N, Ananth YG (1999) Data mining: from serendipity to science. Computer 8:34–37
4. Smyth P et al (2002) Data-driven evolution of data mining algorithms. Commun ACM 45(8):33–37
5. Bradley P et al (2002) Scaling mining algorithms to large databases. Commun ACM 45(8):38–43
6. Karpis George et al (1999) Hierarchical clustering using dynamic modelling. Computer 4:68–75
7. Jain AK et al (1999) Data clustering: a review. ACM Comput Surv 31(3):264–323
8. Ganti V et al (1999) Mining very large databases. Computer 8:38–45
9. Ceglar A, Roddick JF (2006) Association mining. ACM Comput Surv 38(2):1–46
10. Nizar R et al (2010) A taxonomy of sequential pattern mining algorithms. ACM Comput Surv 43:1:3.1–3.41
11. Carl H et al (2013) Sequential pattern mining—approaches and algorithms. ACM Comput Surv 45(2):19:1–19:39
12. Han J et al (2002) Emerging scientific applications in data mining. Commun ACM 45(8):54–58

# Chapter 18
# Information Retrieval

**Abstract** Information retrieval (IR) is the identification of documents or other units of information in a collection that are relevant to a particular information need—a set of questions to which someone would like to find an answer. This chapter presents the basic strategies of IR in length along with their analysis, particularly emphasizing the vector space and probabilistic models of IR, with worked examples in each category; gives the detailed coverage to construction and maintenance of index, and its parallel processing. The fuzzy logic-based retrieval, concept-based retrieval techniques, their algorithms, and worked examples are presented; and Automatic Query Expansion has been dealt with at length. Application of Bayesian networks, and inferences using these have been demonstrated for IR. The newly emerged semantic web for futuristic IR and its applications have been introduced; and the design aspects of distributed IR suited for currently distributed information resources are treated in depth. The chapter ends with the summary and a set of practice exercises.

## 18.1  Introduction

The problem of Information Retrieval in the present context is highly relevant when the volume of information generated is much more than the individuals can easily digest. Consequently, it is extremely difficult to search, locate, and disseminate the precisely desired information from the storage media, irrespective of whether it is local or globally distributed. Ever-increasing information needs to be continually added to the storage systems. This process has been fueled further by the arrival of the Internet and the World Wide Web, as well as digital libraries, research publications repositories, electronic editions of newspapers, journals, and magazines.

Given a set of documents or information collection and information need, Information Retrieval (IR) identifies the documents or other units of information in the

collection, which are relevant to that particular information need. The information need is in the form of a set of questions one is interested to find an answer to. The aim of IR is to locate the document or file which holds the desired information. Sometimes, it is also required to locate the actual position of the required information in the document selected when the document is of a large size.

Here are some examples of IR tasks: finding an article in the Times of India that discusses the freedom struggle of India; searching the recent postings in Twitter that are related to a particular model of smart phones; finding the entries referring to butterflies in an online encyclopedia, etc. [2].

The early IR methods were based on manually assigned keywords to the documents and required complicated Boolean queries. These methods were primarily used by retrieval experts. However, in the 1970s, automatic indexing and natural language queries gained popularity, and the IR facility became more and more available to non-experts. The documents were commonly indexed by automatically considering all the terms in them as independent words. The documents were represented using the set of these keywords, known as Bag-of-Words (BOW). The query formatting also became simplified in the form of short natural language formulation. This, however, added noise in the documents' representation, but the basic methodology for indexing remained the same. Due to this, the non-expert users faced increasingly more troubles, due to the "vocabulary problem". This was because the keywords chosen were often different than those used by the authors of the relevant documents. Due to this, systems' *recall*[1] rates came down. In other cases, the contextual differences between ambiguous keywords were not properly resolved through the BOW's approach, which reduced the *precision* of the results. These two problems are generally referred to as *synonymy* and *polysemy*, respectively [16].

To solve the problem of polysemy,[2] automatic Word Sense Disambiguation algorithms helped to disambiguate the documents' contents as well as the query. The disambiguation algorithms make use of resources like WordNet Thesaurus, corpora text, or co-occurrence data to find the possible senses of a word and map word occurrences to the correct sense. The disambiguated senses are used in indexing and in query processing, which would help to retrieve only the documents that match the correct sense. Unfortunately, sometimes the inaccuracy and errors of automatic disambiguation are more dangerous than not using the disambiguation at all [5].

The models of Information retrieval assume each document described by a set of representative keywords, called *index* terms. An index term is a word from a document, which is supposed to represent the semantics of the document. In general the index terms are *nouns*, as nouns carry maximum meaning of a sentence. Each of the index terms in a document is not equally important; some of the index terms describe the given document better than others. An index term which appears in every document in a given set of documents is of no use. However, a term which appears in only, say, five documents out of thousands of documents should distinctly identify

---

[1] *Precision* and *recall* are parameters which represent the performance of any IR system.

[2] *Polysemy* is an ambiguity in an individual word or phrase, such that the word can be used (in different contexts) to express two or more different meanings.

those documents. Accordingly, an index term should be assigned some numerical *weight* to capture this effect. If $k_i$ is an index term, $d_j$ is a document then $w_{i,j} \geq 0$ is the weight associated with the (keyword, document) pair $(k_i, d_j)$.

Consider that there are a number of documents, and a query is put to retrieve the documents relevant to this query. Ideally, the retrieval algorithm must return all the documents that are relevant to that query, i.e., retrieved documents (say *Ret*) are the same set as the relevant documents (say, *Rel*) in the entire set of documents. However, in a realistic situation, systems may retrieve many non-relevant documents also along with the relevant ones. To measure the effectiveness of retrieval, two ratios, *precision* and *recall*, are used. The precision is the ratio of the number of relevant documents retrieved to the total number of documents retrieved, and it provides the quality of the answer set. However, this does not consider the total number of relevant documents available in the original set. A system may have 90% precision if nine documents are relevant out of total ten that it has retrieved, and there are a total of 100 relevant documents in the store. Hence, the total relevant documents available also matters. The *recall* is ratio of number of relevant documents retrieved to the total number of relevant documents in the entire collection (see Fig. 18.1).

Intuitively, to keep precision high, one need to be overly careful (assuming that human is itself an algorithm). In that case too a few documents will get retrieved. But, in that process many useful documents would be left behind, making recall very low. On the other hand, if one puts efforts to achieve high recall, i.e., trying to retrieve the maximum number of relevant documents out of the total relevant count (a case of too much aggressiveness), many non-relevant documents may also be retrieved, resulting in poor precision. A typical such scenario is shown in Fig. 18.2.



$$Precision = \frac{Relevant\ retrieved}{Total\ Retrieved} = \frac{|Rel \cap Ret|}{Ret}$$
$$Recall = \frac{Relevant\ retrieved}{Total\ Relevant} = \frac{|Rel \cap Ret|}{Rel}$$
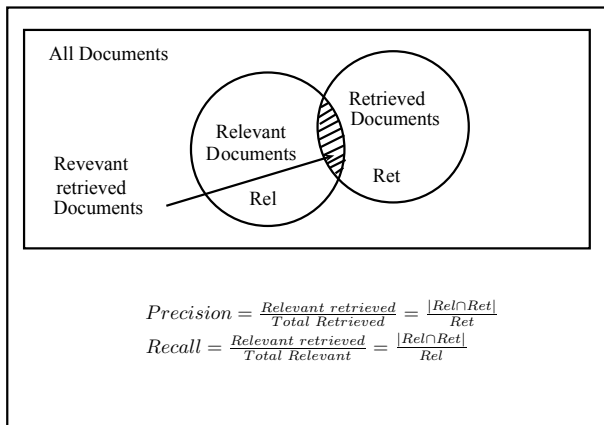
**Fig. 18.1** Precision and Recall

**Fig. 18.2**  Typical relation
between Precision and
Recall
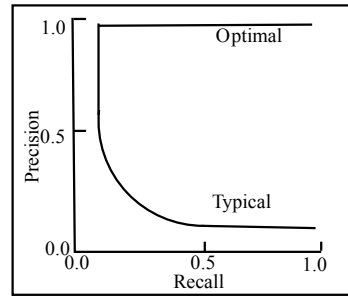


**Learning Outcomes of This Chapter**:

1. Explain basic information storage and retrieval concepts. [Familiarity]
2. Describe what issues are specific to efficient information retrieval. [Familiarity]
3. Give applications of alternative search strategies and explain why the particular
   search strategy is appropriate for the application. [Assessment]
4. Design and implement a small to medium size information storage and retrieval
   system, or digital library. [Usage]
5. Describe some of the technical solutions to the problems related to archiving and
   preserving information in a digital library. [Familiarity]
6. Generate an index file for a collection of resources. [Usage]
7. Explain the role of an inverted index in locating a document in a collection.
   [Familiarity]
8. Explain how stemming and stop words affect indexing. [Familiarity]
9. Describe key challenges in web crawling, e.g., detecting duplicate documents,
   determining the crawling frontier. [Familiarity]

## 18.2  Retrieval Strategies

Consider a set of documents $D_1, D_2, \ldots, D_n$ and query $Q$, a retrieval strategy is
an algorithm for information retrieval that assigns a similarity measure $sim(Q, D_i)$
for $i = 1, 2, \ldots, n$, between query $Q$ and document set $D$. Following are the most
common retrieval strategies [14].

- *Boolean Indexing*. A Boolean query results in ranking based on some score
  assigned to the terms. This can be achieved by associating a weight with each
  query term, which can be used to compute the similarity coefficient.
- *Vector Space Model*. In this model, the document $D_i$ and query $Q$ are each repre-
  sented as vectors in the *term* space. The model computes the similarity $sim(Q, D_i)$
  between two vectors.
- *Probabilistic Model*. A probability of relevance of a document to a query is based on
  the *likelihood* that a term (i.e., query term) will appear in a relevant document. This

probability is computed for each term in the collection of documents. For terms that match between a query and document, the similarity measure is computed as a combination of probabilities of each of the matching terms. This similarity is a measure of relevance of the query to the document.

- *Inference networks*. A Bayesian network is used to infer the relevance of a document to a query, the later is a measure of similarity between the query and the document.
- *Fuzzy set-based retrieval*. In this IR approach, a document is mapped to a *fuzzy set*.[3] For example, *rain*/0.9 versus *rain*/0.2 shows that in the first case, the meaning of the keyword *rain* indicates heavy rain, while in the second set it shows a very mild rain [7].

## 18.3 Boolean Model of IR System

The Boolean model for IR is a simple information retrieval model, based on set theory and Boolean algebra. Since the concept of set is quite intuitive, the Boolean model provides the framework that is easy to grasp by a common user of an IR system, as well as simple to implement. Also, the Boolean queries in the form of Boolean expressions, are precise in their semantics [14].

The Boolean model considers that the *index terms*, which are the representation of documents, are either present or absent in a document. Consequently, the index terms' weights are taken as binary (1 for presence and 0 for absence), i.e., for a document $D_i$ the index term $t_j$ is either present or absent in $D_i$, i.e., weight $w_{ij} \in \{0, 1\}$. A query $Q$ is composed of index terms connected through the binary operators $\wedge, \vee, \neg$ (and, or, not). A query can be expressed as *conjunctive normal form* (CNF) or *disjunctive normal form* (DNF). For example the query "$Q = t_a \wedge (t_b \vee \neg t_c)$" = "$(t_a \wedge t_b) \vee (t_a \wedge \neg t_c)$" can be written in DNF as $\overrightarrow{Q} = (1, 0, 0) \vee (1, 1, 0) \vee (1, 1, 1)$, where each term is a binary weighted vector corresponding to the tuple $(t_a, t_b, t_c)$. Figure 18.3 represent this query using a Venn diagram.

IR is a process of matching the *patterns* in the user *inquiry* with the patterns in the prospective *text documents*. If the inquiry words are taken as a set of words $X$, and the text document words are taken as a set of words $Y$, then IR is nothing but finding the binary relation $X \times Y$. However, words may appear in different morphological forms. It is, therefore, necessary that—before the matching is performed, the words in the inquiry as well as the words in the text document are reduced to their basic form—called *stem words*, by a process called *stemming*.

Consider that $X$ is a Boolean set inquiry (or query) keywords, $Y$ is a Boolean set representing the keywords in the document. With this, a binary relation from $X$ to $Y$ can be represented as

---

[3] A set that contained not only the elements but a number associated with each element that indicates the strength of the membership of the term.
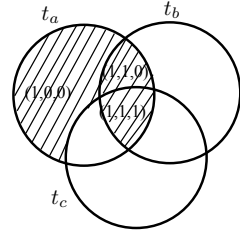
$$R : X \times Y \in \{0, 1\}. \tag{18.1}$$

If for every $x \in X$, there is a corresponding $y \in Y$, then we say $R(x, y) = 1$, i.e., $x$ is related to $y$, otherwise $R(x, y) = 0$.

Since in a Boolean model a query term $x$ is either related to a document term $y$ (i.e., $R(x, y) = 1$), or not related to $y$ ($R(x, y) = 0$), there are sharp boundaries of relations, hence a Boolean-based model is also called *Crisp set* -based model. The following example demonstrates crisp set-based IR.

**Example 18.1** Crisp set-based Information Retrieval.

Consider a set of text documents $Y$ consisting of documents $y_1, y_2, ..., y_n$, as potential documents to be searched for the keywords $x_1, x_2, ..., x_m$ in the inquiry set $X$. Assume that $n = 2$, and $y_1, y_2$ are as given below: (quoted from, "Einstein—The Life and Times," by Ronald W. Clark).

$y_1$: "Thus the new concept of the subatomic world was even by 1920 beginning to produce a gulf. Bohr, Born, and a number of Einstein's other contemporaries, as well as the many of younger men who were in great part responsible for the new idea readily jumped the gap. Einstein stayed where he was. Therefore, the scene in many ways paralleled that into which he has launched his theory of relativity two decades earlier. But then he had been in the iconoclastic vanguard; now he took up station with the small conservative rearguard."

$y_2$: "Plank, the man of honor who had yet not signed the manifesto of 93, had in fact for the first time done as much to keep Einstein in Berlin as he had done to bring him there in 1914. His letter, which, in Einstein's words, had induced him ...."

Let the inquiry be "Einstein's Scientific Theory of Relativity"; and therefore the corresponding set of keywords in the inquiry is $X = \{x_1 = $ "Einstein", $x_2 = $ "Scientific", $x_3 = $ "Theory of Relativity" $\}$, and document set $Y$ consists of documents $y_1$, $y_2$, therefore, $Y = \{y_1, y_2\}$.

In crisp set-based IR, it is required to find $R$—a subset of $X \times Y$, i.e., $R$ ("Einstein", $y_1$), $R$ ("Scientific", $y_1$), $R$ ("Theory of Relativity", $y_1$), $R$ ("Einstein", $y_2$), $R$ ("Scientific", $y_2$), $R$ ("Theory of Relativity", $y_2$). The Boolean relation $R$ for the example under consideration is given in Table 18.1.

It may be seen that while $y_1$ has two matchings, $y_2$ has only one matching. Hence, Table 18.1 shows that the relative relevance of $y_1$ with respect to $y_2$, given by the sum of matching counts in $y_1$ divided by the sum of matching counts in $y_2$ is 2. For larger

**Table 18.1** Crisp set relation between query and document

| Query ($X$) | Document ($Y$) | |
|---|---|---|
| | $y_1$ | $y_2$ |
| $x_1$ | 1 | 1 |
| $x_2$ | 0 | 0 |
| $x_3$ | 1 | 0 |

number of documents: $y_1$, $y_2$, ..., $y_n$, the relevance can be computed in the similar manner for a specified inquiry. Once the relevance is computed, the IR system lists the documents in the order of their relevance.

In the classical binary logic, there is either 100% match for an index term in the inquiry and corresponding term in the document, so $R(x_i, y_i) = 1$; or there is no match at all with $R(x_i, y_i) = 0$. However, the real-life situations are often different. It may happen that the two terms from index and document which are being matched are two forms of the same word, e.g., *real* and *reality*, *phrase* and *phrases*, *exact* and *exactness*, etc. In all these cases the crisp logic returns zero value of relevance. □

## 18.4 Vector Space Model

The vector space model determines the measure of similarity ($sim(Q, D_i)$) between a query and a document using a vector that represents query $Q$, and document $D_i$. The model is based on the intuitive notion that meaning of a document is conveyed by the words present in the document. Figure 18.4 represents the concepts of vectors for a query $Q$ and documents $D_1$, $D_2$, ..., $D_n$. Here $t$ is the total number of terms in the collection [12].

**Fig. 18.4** Vector space model
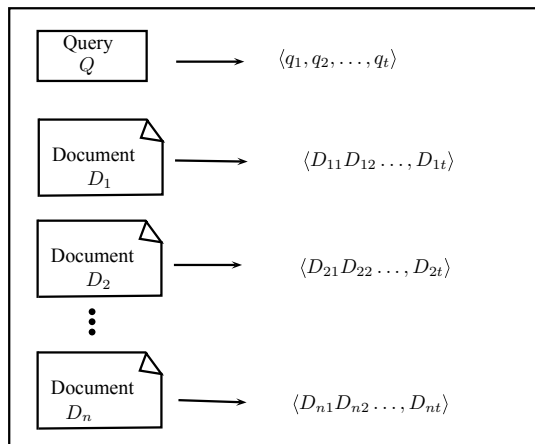
The vector space model is based on a method which compares how close the query vector is to the document vector. The traditional method of determining the closeness between two vectors is the angle between them, where the angle is computed using dot products of two vectors. In our context here, the similarity coefficient is used instead of the angle. If a term is present in the vector, 1 is placed else 0 is placed in the corresponding position in the vector. To account for multiple occurrences of a term in the document, frequency of that term is used instead of merely its presence/absence. Thus, for a query $\langle a, b \rangle$, two documents' vectors, $\langle 1, 0 \rangle$, and $\langle 5, 1 \rangle$ indicate that in the first terms $a$ and $b$ have frequency of 1 and 0, respectively, and for the second these frequencies are 5 and 1, respectively.

The similarity between query and documents can be computed as the distance between the query and each of the document vectors. If a document has the same vector as the query, their distance is minimum and have the highest similarity.

Instead of specifying the list of terms in a query, a user is often interested to indicate that certain terms are more important than others. One approach to this is that user indicates a higher weight to specific terms by manually specifying the weight. The other approach is automatically assigning the weight equal to the number of times (frequency of the term) a term appears in a document. But, if a common term appears in every document, it cannot be used to distinguish a rare document from the rest. For example, if a term appears only in very few documents, say two documents comprising such a term out of a thousand documents, then that term is an important criterion to distinguish those two documents as relevant to that term. In this case, the similarity is proportional to the ratio of the total number of documents ($d$) to the documents count (say $df_j$, the document frequency) which have the occurrence of this rare term. The ratio is called *inverse document frequency*, $idf$.

The important terms encountered in the above discussion are defined and listed below as

$t$ : *Terms*. It is the number of distinct terms (keywords) in the entire collection of documents.

$tf_{ij}$ : *Term frequency*. It is the number of occurrences of the term $t_i$ in the document $D_j$.

$df_j$ : *Document frequency*. It is the number of documents that contain the term.

$idf_j$ : *Inverse document frequency*. It is equal to $\log \frac{d}{df_j}$, where $d$ is the total number of documents.

Each document is represented as a vector, with a total $t$ number of components, and each entry in the vector corresponds to a distinct term from the entire collection set. In addition, each component in a vector is filled with *weights* computed for the corresponding term. This weight is automatically assigned based on how frequently the term has occurred in the *entire documents collection*, and another weight is based on how often the term has appeared in the *particular document*. The weight of a term in a document increases, the more often it appears in that document, and decreases, the more often it appears in other documents.

The weight of a term in a document is defined as a combination of *term frequency* $tf$, and the inverse document frequency $idf$. Each term has a position in the vector,

if it is present in the document, that position is marked with its weight, else marked as weight 0. To compute the weight ($dw_{ij}$) of a term $t_i$ in a document $D_j$, the following equation of $tf.idf$ is used:

$$dw_{ij} = tf_{ij} \times idf_j. \tag{18.2}$$

When an information retrieval system (actually it retrieves the document carrying the needed information) is used to query a document collection using a query of $n$ number of terms, the system computes one *document vector*, $\langle dw_{1j}, dw_{2j}, \ldots, dw_{nj} \rangle$ of size $n$ for each of the documents $j$, and these vectors' components are filled with weights as discussed above. Similarly, a *query vector*, $\langle qw_1, qw_2, \ldots, qw_n \rangle$ is computed for the terms found in the query. Note that, size of the query vector is also $n$. The similarity measure between query $Q$ and a document $D_j$ is defined as the dot product of two vectors.

$$sim(Q, D_j) = \sum_{i=1}^{n} qw_i \times dw_{ij}. \tag{18.3}$$

## 18.5  Indexing

The vector space model and other retrieval strategies make use of an inverted index file structure to avoid the length of search in the keywords of every document for which relevancy is to be established. Instead of searching into a document, an inverted index is generated in advance for all the keywords in the document set. An entry for each of the $n$ terms ($t_1, \ldots, t_n$) is stored in an index structure, like the one shown in Fig. 18.5. For each term $t_i$, a pointer references to a linked list, which contains an entry for each document containing this term as well as the term frequency in that document is present. For example in row $i$, there are entries for ($d_j, tf_{ij}$) forming a connected list for term $t_i$. The $d_j$ is a document in which term frequency is $tf_{ij}$ for the term $t_i$. The figure shows that the term $t_i$ has a frequency of 7 in document $d_3$. This indexing structure has the advantage that the retrieval system can search the term quickly, as well as the documents in which the term appears [18].

### 18.5.1  Index Construction

A key challenge in the construction of an index is the size of the data involved in the index itself (see Fig. 18.5), which is a dynamic data structure typically used for cross-reference generation, but cannot be kept together in the memory of a typical system due to its size. The task to be performed here is a *matrix transposition*, given that the documents' terms' matrix is very sparse. Such matrices are not so easy to manipulate directly as an array. Therefore, the index construction makes use

**Fig. 18.5**  Inverted index file
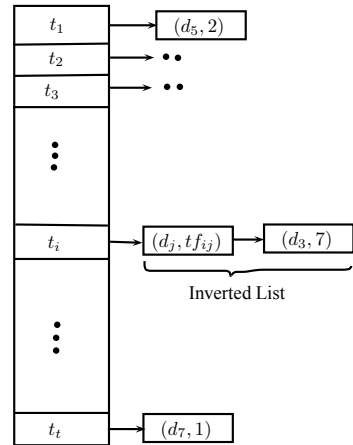


of index compression techniques, together with distribute-comparison-based sorting techniques.

Algorithm 18.1 is a simple in-memory inversion algorithm. The key idea in this algorithm is that the first pass through the documents to be indexed collects terms frequency ($tf_{ij}$) information, which is sufficient for the construction of an inverted index. The index is stored in the memory in the form of a template. The second pass places the pointers, shown by arrows, at their correct positions in the template as shown in Fig. 18.5.

---

**Algorithm 18.1** To build an inverted index using the *in-memory technique*

---

1: **Pass I**: Make an initial pass over the collection of documents.
2: For each term $t_i$, count its term frequency $tf_{ij}$ in each document $d_j$, and determine the upper bound $u_{t_i}$ in bytes, on the length of the inverted list for $t_i$.
3: Allocate an in-memory array of $\sum_{t_i} u_{t_i}$ bytes, and, for each term $t_i$, create a pointer $c_{t_i}$ to the start of a corresponding block of $u_{t_i}$ bytes.
4: **Pass II**. Process the collection of documents a second time.
5: For each document $d_j$, and for each term $t_i \in d_j$, append a code representing $\langle d_j, tf_{ij} \rangle$ at $c_{t_i}$, and update $c_{t_i}$.
6: **Pass over in memory Index**: Make sequential pass over these index, for each term $t_i$, copy the $tf_{ij}$ representations of the $\langle d_j, tf_{ij} \rangle$ pointers from the allocated $u_{t_i}$ bytes to the inverted file, and compress if required.

---

It is possible to extend the in-memory technique to *data collection* technique. In this technique, the index size may exceed the memory size. This is possible by laying off the index skeleton on the disk, while partial sequences of indexes are created in the memory, and each one is transferred to the disk in a skip-sequential manner into a template in a large file. Using this extended method, and using compression, it is possible to create indexes of the size of terabytes using a memory of a moderate size of 4–8 GB—a size common in the present time.

Parallel Processing of Index

For parallel processing of indexes, they can be constructed in parallel and can be merged after regular intervals (see Algorithm 18.2). The merge-based inversion technique reads the documents and indexes them in the memory until a fixed capacity is achieved. Every inverted list is represented using a structure, which can grow as more information about the index terms become available. For this structure, dynamic resizable arrays are most appropriate. As soon as the memory is full to its predefined capacity, the indexes are flushed out to disk in a single run, such that the inverted lists are stored in the disk file in a lexicographic order. This lexicographic order later becomes useful for the sequential merging of the indexes. Since the runs of these subindexes are never used to answer a query, there is no need to store their vocabulary in an explicit structure, hence each run can be written at the head of its inverted list. Once a run is written into the hard disk, it is fully deleted from the memory so that the construction of the next run begins with initially empty vocabulary.

---

**Algorithm 18.2** To build an inverted index using *Merge-based technique*

1: **while** all the documents are not processed **do**
2:    Initialize an in-memory index, using a dynamic structure for the vocabularies and a static coding scheme for inverted lists; store lists either in dynamically resized array or in linked blocks.
3:    Read documents and insert $\langle d, tf \rangle$ pointers into the in-memory index, continuing until all allocated memory is consumed.
4:    Flush this temporary index to disk, including its vocabulary.
5: **end while**
6: Merge all the set of partial indexes to form a single index, compressing the inverted lists if required.

---

When all the documents have been processed, the runs available in the disk are merged to get the final index. The merging process builds the final vocabulary on the fly. Since the runs from the disk are read (into a buffer) for merging, a sufficiently large size of the buffer will reduce the disk accesses, hence making the process further faster. However, if the free disk space is limited, the final index can be written back into the RAM at the space occupied by the runs, progressively as they are processed, which is helpful in representing the final inverted lists more efficiently. The latter becomes possible because the final index is typically smaller than the runs, hence the vocabulary information is not duplicated.

The index construction using the merge-based approach is common and practical for data collections of all sizes. It is scalable, and operates efficiently in a memory size as small as 100 MB. The overheads of a disk space can be limited to a small fraction of the final index, as it requires only one parsing pass over the data, and the method can be extended from keyword indexing to phrase indexing.

## *18.5.2   Index Maintenance*

Inserting a new document into the text collections amounts to inserting a few bytes to the end of every inverted list depending on how much the terms in that document are, i.e., size of the document. Since a document may consist of 100s–1000s of terms, such insertion of terms requires fetching and extending 100s of inverted lists, and it may require 10–20 s in the worst cases, to rebuild the inverted list for the new collection of documents. However, the merge-based inversion approach can index thousands of documents per second, making this method almost 10,000-times faster [12].

For fast insertion of terms for indexing into the inverted list, the disk resident part of the list be not accessed frequently, else it will reduce the overall speed. The practical solution is to amortize the update cost over a sequence of insertions. There are many properties of text databases, which allow strategies for cost amortization. The new documents are not immediately available for searching, if they are searchable they can be made available through a temporary in-memory index—that is, the last subindex in the merging.

There are three broad categories available for updating the index as the new documents get added into the collection, they are *rebuild* from scratch, *merge* an existing index with an index of new documents, and *incremental update*.

Rebuild

There are applications where the index is not updated at all, but it is rebuilt from scratch at some regular intervals. The presence of new documents is established through crawling, and an update is not immediately needed for these documents. This approach is considered economical even for gigabytes of data, where rebuilding takes just a few minutes, rather than updating the existing index, which anyway will require fetching the document as well as the index to be updated.

Intermittent Merge

Number of inverted lists can be maintained for the documents collection, in the memory of a system. Having the lists in memory, it is easy (less complex) to insert new documents into these lists when the new documents are discovered. This is carried out using Algorithm 18.2 discussed on page 567, through the process of merging the indexes. Alternatively, the existing documents' index remains in a standard inverted file, and new documents are indexed as an inverted file data structure in the memory. With this, the two indexes can share a common vocabulary. In this process, both the indexes are made available to queries, and the result of any query is the union of the query results from both of these inverted indexes. When the size of the in-memory index crosses a threshold size, it is merged into the index file on the disk.

Other approaches for indexing use *incremental update*, or *choice of alternative strategies* for indexing as a combination, suffix arrays, wavelet trees, Bayesian inferences, predicate-based indexing, and probabilistic indexing [14].

## 18.6 Probabilistic Retrieval Model

The probabilistic model of retrieval computes the *similarity measure* $(sim(q, d_i))$ between the query $q \in Q$ and a document $d_i \in D$ as the probability that $d_i$ is relevant to $q$, where $D$ is the set of documents in the collection, and $Q$ is the set of queries. The probabilistic retrieval method estimates the term weight based on how often the term appears in the *relevant* but does not appear in the *non-relevant* documents. The term weight is calculated using the *probability ranking principle*, which is based on the assumption that optimal performance is observed when documents are ranked on their relevance to the query. In the approach used, probabilities are first assigned to components of the query and then each of these is used as evidence in computing the probability that a given document is relevant to the query [8].

Each term in the query is also assigned a weight corresponding to the probability that the document term matched with the query will retrieve a relevant document. The weights of the query terms are aggregated to obtain the final measure of relevance. A probability-based information retrieval system ranks the documents in decreasing order of probability of relevance to the user's information needs. Following are essential preconditions for this probabilistic retrieval model:

- Retrieval accuracy is dependent on how accurately the query and document have been represented, and does not directly depend on the documents and the queries,
- The representation of documents and queries may not be accurate due to a variety of uncertainties prevailing in the method of representation itself.

Figure 18.6 shows the conceptual probabilistic model for IR, where event space is represented by $Q \times D$, such that $Q = \{q_1, q_2, q_3, \ldots\}$ is a set of queries representing the information need, and $D = \{d_1, d_2, d_3, \ldots, \}$ is a finite set of documents. Each query $q_i$ and document $d_j$ is in the form of *descriptors*, where $q_i$ and $d_j$ are set of terms (i.e., keywords). The descriptor is a binary-valued vector, and each element in that corresponds to a term. Every query is taken as a unique event, i.e., two identical queries at different times are treated as different events. We assume that $G$ is a set of possible *relevance judgments* for the Cartesian product of documents' set $D$ and queries set $Q$. Let the relevance relationship be $r$, between the query set and document set, in the form of a mapping $r : Q \times D \to G$. In case of Boolean IR, a document is either relevant to a query or not, hence for any query $q_i$ and document $d_j$, there is $r(q_i, d_j) \to \{0, 1\}$.

**Fig. 18.6** Conceptual probabilistic model of IR
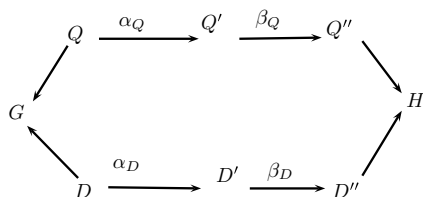
In fact, an IR system does not directly handle the documents and the queries, but handles them indirectly, in the forms of their representations. For example, a document is represented in the form of *index terms*, and a query is represented as a Boolean expression comprising terms and Boolean operators (see page for more detail). Let us assume that $Q'$ and $D'$ are representations of queries and documents, respectively. These representations have a mapping from the original query and documents through some functions, which are expressed as $\alpha_Q : Q \to Q'$ and $\alpha_D : D \to D'$. Hence, if there are two different documents, but represented with the identical set of index terms, then they will be mapped onto the identical representation.

Further mapping is introduced from the representation ($Q'$, and $D'$) to *object descriptions*, to make the models more general. This is done by supplementing a *weight* to the index term forms of the queries' and documents' representations. The weight is a real number. Let us assume that these object descriptions, for query set and document set, respectively, are $Q''$ and $D''$, and the mappings as $\beta_Q : Q' \to Q''$ and $\beta_D : D' \to D''$, respectively. Due to the introduction of weight the new mapping shows a more accurate relevance relation between the query and its descriptor set, and similar is the case for the document set to their descriptors. Therefore, the more correct value of the relevance function $r$ is $r : Q'' \times D'' \to H$.

For a submitted query $q_i \in Q$ to the IR system, the documents $d_j \in D$ are ranked according to the decreasing order of $r(q_i'', d_j'')$, such that the document with the highest rank (of relevance) is at the top. The job of an IR system that ranks the documents in the order of their relevancy for a query $q_i''$ is to calculate relevance and rank every document $d_j \in D''$. Often, for the sake of simplicity, the description and representation are treated the same, and both are represented as the form of set of terms [4].

## 18.7   Fuzzy Logic-Based IR

The fuzzy retrieval technique is based on the fuzzy set theory and fuzzy logic—an extension of the classical set theory. This fuzzy retrieval technique is based on the concept that the word matching between the inquiry word set and the text word set should not be limited to the perfect matching with the stem words. But, since the words in inquiry also match with their synonyms in the text documents, the matching should be graded, depending on the degree or level of matching in the range from 0 to 1. The extremes of this range, a special case in fuzzy logic, corresponds to the Boolean matching. Fuzzy logic is more realistic than the Boolean logic, simply due to the fact that it considers the exact as well as vague matching, the latter being more frequently encountered in the real world [7].

In the relations over fuzzy sets, the elements of two sets have a degree of association as a form of relation rather than simply—related (binary 1) or not related (binary 0). The degree of association ranges from 0 to 1, where 0 indicates the total

absence of relation and a 1 indicates the total presence of the relation; therefore, a fuzzy relation between query keywords set and of document's keywords is

$$R : X \times Y \in [0, 1] \tag{18.4}$$

and the range of $R(x, y)$ varies from 0 to 1 depending on how close the $y$, $(y \in Y)$ is associated with $x$, $(x \in X)$.

Information Retrieval Using Fuzzy Sets

The membership value $R(x_i, y_j)$ specifies, for each $x_i \in X$, $y_j \in Y$, the grade of relevance of index term $x_i$ with the document $y_j$. The grade of relevance depends on many factors [3]:

1. position of term $y_j$ in the text document, if the document is a research article, and $y_j$ appears in the list of keywords, the abstract, or in the conclusion part, the relevance is higher;
2. frequency of occurrence of $y_j$ in the document;
3. $x_i$ and $y_j$ are terms formed from the same basic stem word; and
4. $y_j$ is synonym of $x_i$—the proximity of the meaning of $x_i$ and $y_j$ decide value of $R(x_i, y_j)$'s closeness with 1.

The criteria (1) and (2) above are user-defined and they can be programmed in the implementation according to the user needs. The stem word criteria (3) requires a data structure similar to the one shown below in Fig. 18.7, which helps to locate the stem word for a given word, and then the stem word is substituted in the original text before the retrieval technique is applied to it [4].

Other, more often used approach for stemming is through some stemming algorithm, e.g., Porter's stemmer, which takes the benefit of certain patterns in the words to obtain the stem word. For example, for the words with "ing" at the end, the stem word can be obtained by removing the "ing" part, like "book" from "booking". Also, there are other features, like removing "ed" at the end of the past tense of a verb; we obtain its stem word, say "book" and "look" from "booked" and "looked", respectively. Along with this, there are some more complex patterns, like "goose" from "geese", etc.

**Fig. 18.7** Data Structure for finding stem words
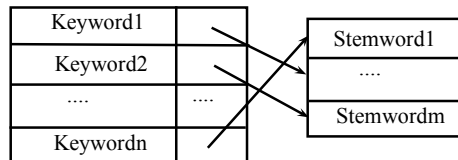
Another important relation for IR based on criteria (iv) above is the fuzzy thesaurus, which plays a pivotal role for FIR. The fuzzy thesaurus shows the relationship between the pairs of words based on their centrality or degree of relevance. The structure of a *fuzzy thesaurus* $(T)$ is

$$\langle WC1 \rangle \langle WC2 \rangle \langle RD \rangle$$

where $WC$ stands for *word category* and $RD$ is the degree of relationship between the words $WC1$ and $WC2$. A typical examples for this can be as follows.

> attraction, love,        0.8
> studious,   hardworking, 0.9
> war,        crime,       0.7.

A relation $\langle x_i \rangle$, $\langle x_j \rangle$, $\langle 1.0 \rangle$ shows that $x_i$ is a perfect synonym of $x_j$. The fuzzy thesaurus can be manually constructed, or can be generated from the lexicons. *Transitivity relationship* can be applied by computing the missing relationship degrees from the existing ones. The thesaurus, say $T$, is a *reflexive fuzzy relation*, defined over $X^2$. For each pair of index terms $(x_i, x_j) \in X^2$, the $T(x_i, x_j)$ expresses the degree of association of $x_j$ with $x_i$, such that the degree to which the meaning of the index term $x_j$ is compatible with the meaning of the index term $x_i$. The objective of this relation is to deal with the problem of *synonyms* among the index terms, for example a document's term is a synonym of query term or vice versa. The relation helps to identify the relevant documents which otherwise would not be selected in the absence of a perfect match between the keywords in the user inquiry and those in the text document.

Different approaches can be used for the construction of fuzzy thesaurus. For example, experts in the domain of text can be asked to identify, in a given set of index terms, the pairs of words whose meaning they consider are associated, and provide the degree of association for each pair. In Fuzzy Information Retrieval (FIR) an inquiry can be expressed in the form of a fuzzy set (say $Q$) based on the index term $X$. Then, by composing $Q$ with the fuzzy thesaurus $T$, we obtain a new fuzzy set on $X$, say $A$—which represents the *augmented inquiry*, i.e.,

$$A = Q \circ T, \tag{18.5}$$

where "$\circ$" is called *max-min* composition operator, such that

$$A(x_j) = max\text{-}min[Q(x_i), T(x_i, x_j)]. \tag{18.6}$$

Here, $x_i \in X$, for all $x_j \in X$. The retrieved documents, expressed by a fuzzy set $F$ defined over $Y$, are then obtained by composing the augmented inquiry, expressed by the fuzzy set $A$, with the relevance relation $R$, i.e.,

$$F = A * R \tag{18.7}$$

where "$*$" is a matching operator, which evaluates the degree of fuzzy matching by multiplying the fuzzy membership of the augmented inquiry with the fuzzy membership of the corresponding words in the text. Finally, the relevance measure of the text with the inquiry under consideration is obtained by summing all the values of fuzzy matching for the text.

**Example 18.2** Fuzzy logic-based Information Retrieval.

Let the terms be $x_i$, $i = 1, 6$, representing keywords—"Einstein", "scientific", "Theory of Relativity", "Bohr", "subatomic", and "New idea", respectively. Let the given inquiry be $Q =$ *"Einstein's Scientific Theory of Relativity"*, and the vector representation of corresponding fuzzy inquiry be

$$\begin{matrix} x_1 \ x_2 \ x_3 \\ Q = [1 \ \ 0.6 \ \ 0.8] \end{matrix} \tag{18.8}$$

where 1, 0.6, and 0.8 are called the *centralities* of $x_1$ (Einstein's), $x_2$ (Scientific), $x_3$ (Theory of Relativity), respectively. The centrality indicates the presence of certain qualities, whose computations are modeled as a computation of fuzzy membership degree. The relevant part of the fuzzy thesaurus $T$, restricted to the support of $Q$, is given by the matrix:

$$T = \begin{bmatrix} & x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \\ x_1 & 1 \ \ .6 \ \ .9 \ \ 0 \ \ .1 \ \ 0 \\ x_2 & .6 \ \ 1 \ \ .8 \ \ .6 \ \ .5 \ \ .8 \\ x_3 & .9 \ \ .8 \ \ 1 \ \ 0 \ \ 0 \ \ .7 \end{bmatrix}. \tag{18.9}$$

Note that terms' pairs $(x_1, x_1)$, $(x_2, x_2)$, and $(x_3, x_3)$ has each a fuzzy matching of 1. On using (18.5), and the data given by Eqs. (18.8) and (18.9), we get

$$A = \begin{bmatrix} 1 \ .8 \ .9 \ .6 \ .5 \ .7 \end{bmatrix}. \tag{18.10}$$

The values in augmented query $A$ are obtained as follows: we show the computation for third element ($A_3$), i.e., 0.9 as

$$\begin{aligned} A_3 &= max(min(1 \times 0.9), min(0.6 \times 0.8), min(0.8 \times 1)) \\ &= max(0.9, 0.6, 0.8) \\ &= 0.9. \end{aligned}$$

Assume that relevance relation $R$ (i.,e., $R(x_i, y_j)$) is given by the matrix,

$$R = \begin{bmatrix} & y_1 & y_2 \\ x_1 & .7 & .3 \\ x_2 & .3 & .1 \\ x_3 & .6 & .1 \\ x_4 & .6 & 0 \\ x_5 & .6 & .3 \\ x_6 & .6 & .1 \end{bmatrix}, \tag{18.11}$$

where $y_1$, $y_2$ are the documents related to index terms $x_i$, $i = 1, \dots, 6$. Using Eq. (18.7) and data given by (18.10) and (18.11), we get fuzzy relevance relation $F$ as

$$F = A * R = \begin{bmatrix} 2.56 & 0.69 \end{bmatrix}. \tag{18.12}$$

Equation (18.12) shows that the relative relevance of $y_1$ with respect to $y_2$ is 2.56/0.69, i.e., 3.7. Thus, the result appears to be more realistic in comparison to the result obtained using crisp (binary) logic. This fact is also supported by the contents of texts $y_1$ and $y_2$. Once the FIR system lists the documents with their relevance values, the user can now decide whether to inspect all the retrieved documents supported by the fuzzy set $F$ or to inspect only some of the documents depending on the degree of association of the document with the index terms.                                    □

The use of fuzzy set theory for IR shows that the fuzzy relevance relation and fuzzy thesaurus are more expressive than their crisp set counterparts. Also, since the degree of association is returned along with the retrieved documents, it helps the user to decide the order in which the documents can be viewed, particularly when the documents are in large number.

The FIR promises a higher potential for cross-language text processing and IR. Every language and its semantics have a close association with the culture in which it has its roots, and therefore, exact matching terms for any language are not possible in other languages. In fact a degree of relevance or, only fuzzy relation exists between the matching words of the two or more languages. We have planned to work on the cross-language areas, which include English, and Sanskrit-based Indian languages [3].

## 18.8   Concept-Based IR

The keyword-based approach we discussed above is also called the BOW (Bag-of-Words) approach. The concept-based information retrieval makes use of *semantic concepts* for the representation of the documents and queries, instead of (or in addition to) keywords; and the approach performs the retrieval in the concept space of query and documents. Hence, this retrieval model is less dependent on the specific terms (keywords) used, and yields a match even when the same notion is described by a different set of terms in the query or in the target document, or in both. This helps in eliminating the problem of *synonymy*. Since more relevant documents are likely

to be retrieved from the store, the *recall* rate increases. In the similar way, if the concepts chosen for the words, particularly the ambiguous words in the query and document are accurate, the non-relevant documents that were retrieved with the BOW approach could be eliminated from the results. Since the non-relevant documents are lesser in the retrieved documents now, it increases the *precision* rate. Note that in the keyword-based approach of IR, non-relevant documents' share increases in the retrieved documents when the keywords have multiple meanings, e.g., "bank" (of river and of money). The problem of many senses of a word is called *polysemy*.

The concept-based methods can be characterized by the following three parameters:

1. *Concept representation.* The concept-based IR makes use of real-life concepts that closely resemble human perception. The concepts are based on the language of the text of documents and queries.
2. *Mapping method used.* It is the method used for mapping the natural language text to concepts. In fact, the ideal mechanism is the manual approach, which builds a hand-crafted ontology of concepts along with a list of words to be assigned to each concept. However, this approach is inefficient and time consuming. The automatic mapping between concepts and words can also be done through machine learning, but with loss of accuracy.
3. *Use in IR.* The concepts are used during the indexing and retrieval phases of the documents. A simpler but less-accurate method applies the concept analysis in one stage only. For example, it is better to apply the concept analysis in the query expansion rather than document retrieval.

A large and diverse knowledge repository, like Wikipedia, can create a powerful concept ontology, well-suited for concept-based IR. A wide range and exhaustive coverage of topics in Wikipedia's diversity, coupled with automatic ontology-building capability, can be used for the purpose of highly fine-grained ontology. Additionally, the inverted index language provides a mapping from massive natural language text *terms* of the entire Wikipedia collections, to the concepts as per the context and sense of the text terms. This entire process produces a powerful classifier that automatically maps any given text fragment to its concept ontology [9].

### 18.8.1  Concept-Based Indexing

A concept-based IR algorithm maps documents and queries individually to the Wikipedia concept space. The indexing and retrieval are performed in this space only. In the Wikipedia-based Semantic Analysis (SA), the semantics of a word are described by a vector that stores the word's association strengths to Wikipedia-based concepts. A concept is in the form of a *concept vector* $\overrightarrow{F_d}$, which is generated from a single Wikipedia article $d$, as a vector of words appearing in that article. The article is weighted by $tf.idf$ score. Once these concept vectors are generated, an inverted index is created for the purpose of mapping back each word to the concepts it is
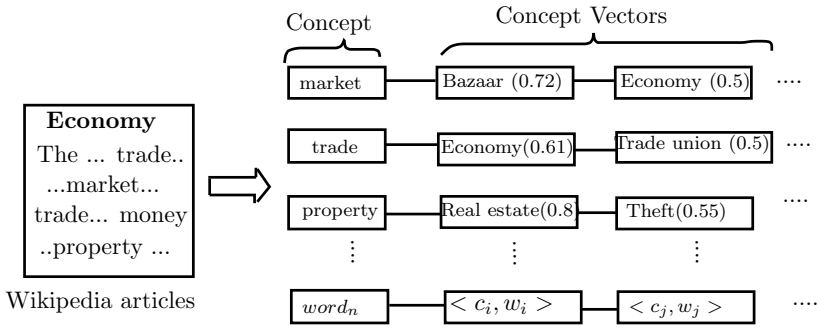
**Fig. 18.8** Semantic analysis from Wikipedia articles

associated with. Thus, each word appearing in the Wikipedia corpus can be seen as triggering one or more concepts $c_j$ it points to the inverted index. An attached weight $w_j$ with the concept represents the degree of association between that word and the concept. This process is illustrated in Fig. 18.8, which shows how the semantic analysis is carried out of a Wikipedia article.[4] The articles and the words in the articles are processed to build a weighted inverted index, which represents each word of article as a vector in the space of all Wikipedia concepts, i.e., articles in the Wikipedia itself.

In the *concept-based indexing*, each document (i.e., article) in the corpus is mapped to a vector of weighted concepts, represented by $\langle c_1, w_1 \rangle, \langle c_2, w_2 \rangle, ..., \langle c_n, w_n \rangle$. Like Bag-of-Words (BOWs) vectors, the concept-based vectors are also sparse, hence concept weights are zero for a large majority of the Wikipedia concepts. Since every word in the document to be indexed may be related to a large number of concepts, and a document containing a collection of words is likely to be related to an even larger number of words, indexing an entire list of related concepts for every document is not feasible. Therefore, only the concepts having the highest weights (for example, "Bazaar", "Economy", etc., in Fig. 18.8) are used for indexing. In a sorted representation of the weighted vector, this subset of concepts is simply its prefix.

It more difficult to map the long documents in full, into the concept space. For example, a small part of a long document might be relevant to a given query, but the semantics of this small part are not likely to be fully represented in the concepts vector of the complete document. Note that, a somewhat similar problem also exists in the Bag-of-Words approach, where the term frequency ($tf$) value is normalized to account for the documents of varying lengths.

Due to the averaging effect of the representation of longer text fragments, and due to the practical limitation to use only a small subset of the representation of concepts, the challenge in the concept-based retrieval technique is even greater. Because the concepts generated for a subset of the larger document, where the subset is relevant

---

[4]A Wikipedia article is an article about some topic, for example, we find articles in the collections of Wikipedia, like websites, Internet, WWW, etc.

and the remaining document consists of non-relevant topics, the representation of the latter need to be pruned out of the index vector. This is necessary as otherwise the concepts weights in the overall document concepts vector might be too low to show any significance of the retrieval results.

Semantic Analysis-Based Indexing Algorithm

An algorithm suitable for indexing larger size documents, based on semantic analysis and using the inverted index, is given as Algorithm 18.3. This algorithm indexes a corpus $D$ of documents using semantic analysis concepts, by trimming of semantic analysis vector to $s$ as the first concepts. Each document $d \in D$ is represented by a concept vector $\overrightarrow{F_d}$. For each concept $\langle c_i, w_i \rangle \in \overrightarrow{F_d}$, the corresponding $\langle d, w_i \rangle$ is added into the inverted inverted index. Here, $c_i$ is the concept and $w_i$ is the concept weight.

To overcome the problem of mapping each large document $d \in D$ in full into concept space, $d$ is divided into smaller fixed length (size $l$) overlapping passages set $P_d$. Then each passage $p \in P_d$ is represented separately by its own generated set of concepts $\overrightarrow{P_d}$.

Each passage $p$ is indexed and can be retrieved as a stand-alone unit of information. For this, all concepts $\langle c_i, w_i \rangle \in \overrightarrow{F_p}$ corresponding to a passage $p$, the passage is ranked separately as an independent unit along with its relevance in its parent document ($d$), shown in the algorithm by "$add\langle p, w_i \rangle$ to $InvIndex[c_i]$". These concepts are indexed in a standard IR inverted index, with each concept having a unique identifier in the form of a token. The score $w_i$, associated with each concept $c_i$ in the vector, is used as the token weight, which is equivalent to term frequency $tf$, in standard text indexing.

---

**Algorithm 18.3** Semantic analysis-based indexing in an inverted index

---

1: **Procedure** SA-Indexing($D, s, l$) {Indexes corpus $D$ using SA concepts; trims SA vector to $s$ as first concept; then segments document to passages, each of length $l$.}
2: **for all** $d \in D$ **do**
3:    $\overrightarrow{F_d} \leftarrow SA(d, s)$
4:    **for all** $\langle c_i, w_i \rangle \in \overrightarrow{F_d}$ **do**
5:      $add\langle d, w_i \rangle$ to $InvIndex[c_i]$
6:    **end for**
7:    $P_d \leftarrow$ Divide-Into-Passages($d, l$)
8:    **for all** $p \in P_d$ **do**
9:      $\overrightarrow{F_p} \leftarrow SA(p, s)$
10:      **for all** $\langle c_i, w_i \rangle \in \overrightarrow{F_p}$ **do**
11:        $add\langle p, w_i \rangle$ to $InvIndex[c_i]$
12:      **end for**
13:    **end for**
14: **end for**

---

## *18.8.2   Retrieval Algorithms*

Once a query is received by an IR system, it is converted into a concept vector by the retrieval algorithm. The representation method used is identical to the one we discussed above, for documents and passages during the indexing. The indexes of full documents and passages, as evidences, are to be combined for ranking. This combination is performed by retrieving both set of results, and then by summing each document's full score with the score of the best performing passage in it. In the next phase, documents are sorted on a combined score, i.e., sum, and the top-scoring documents are output. All the above steps are described in the retrieval algorithm 18.4. The algorithm works as follows:

1. Retrieve the query results, which are based on SA concept, for query $\overrightarrow{q}$, as well as the cutoff concept vector at $s$;
2. Retrieve results for query $\overrightarrow{q}$ using the combined results;
3. Score the document's match to the query using the standard inverted index function *InvIndex-score()*.

---

**Algorithm 18.4** SA-based Retrieval

---

1: **Procedure** SA-Retrieval($\overrightarrow{q}$, $s$) {Retrieve the SA concept-based results for query $\overrightarrow{q}$, and cutoff concept vector at $s$}
2: $\overrightarrow{F_q} \leftarrow SA(\overrightarrow{q}, s)$
3: **return** DocsPass-Retrieve($\overrightarrow{F_q}$)
4: **Procedure** *DocsPass-Retrieve*($\overrightarrow{q}$) {Retrieve results for query $\overrightarrow{q}$ from the combined results; Score the document's match to the query using the standard inverted index function *InvIndex-score()*.}
5: **for all** $d \in D$ **do**
6:     $W_d \leftarrow InvIndex\text{-}Score(\overrightarrow{q}, d)$
7:     **for all** $p \in PASSAGES(d)$ **do**
8:         $W_p \leftarrow InvIndex\text{-}Score(\overrightarrow{q}, p)$
9:     **end for**
10:     $W_d' \leftarrow W_d + max\ W_p$
11: **end for**
12: **return**  ranked list according to $W_d'$

---

The retrieval algorithm 18.4 has a single parameter $s$ that controls the query concept vector's cutoff. The value of $s$ can be chosen the same as that during the indexing, however it is not necessary. If the entire corpus is indexed with large cutoff values, the resultant costs for computation as well as storage will be high, hence it is not advisable to index the entire corpus with large cutoffs. Since the query is much smaller in size, there are no such costs as that for text corpus, hence a finer representation will be beneficial. That can be achieved using a higher value of $s$.

## 18.9 Automatic Query Expansion in IR

Most IR systems, and particularly the web search engines, have a standard user interface comprising of an input box to accept from a user, a query in the form of keywords. The submitted keywords are matched against the collection of index terms to find the documents that contain those keywords. The results are then sorted by various methods. When there are many topic-specific keywords in the user's query, which accurately describe user's information need, the system is likely to return good matches for the query. However, when this query is short—comprising 2–3 words, as the case usually is—there is likely ambiguity in the language of the query, then this simple retrieval model is sensitive to errors [1].

The most serious issue in retrieval effectiveness is the *term mismatch* problem, i.e., indexers and the users do often not use the same keywords. For example, a document uses "tv" while user submits the query with keyword "television". This is known as the *vocabulary problem*. This problem gets compounded due to *polysemy*, i.e., the same word with different/multiple meanings, such as Java (name of language, and also name of a place), and also due to *synonymy*, i.e., different words with the identical or similar meanings, such as "tv" and "television". Synonym words, along with word *inflections* (like in plural forms, "book" versus "books") may result in a failure to retrieve relevant documents. This may decrease the *recall*. The problem of polysemy may cause retrieval of erroneous or non-relevant documents, thus causing a decrease in *precision*.

Several approaches have been proposed, to deal with the vocabulary problem; some of them are the following:

Interactive query refinement,
Relevance feedback,
Word Sense Disambiguation,
Query expansion, and
Search results clustering.

The technique of query expansion is one of the most natural and successful techniques, using which the original query is expanded with other words that best capture the user's actual intent, or it simply produces a more useful query—a query that is more likely to retrieve relevant documents.

As the use of search engines increased over time, the size/length of the user's query has also increased. In the year 2009, an average query's length was 2.3 words. However, there has been an increase in the number of long queries per user or per session of interaction, of five or more words; the most common queries are still those of one, two, and three words. When a query is short, the vocabulary problem is more serious, because the shortage of query terms limits the scope of handling synonymy. At the same time, the reduced resizing of data makes the effects of polysemy more severe. This required the need and scope of Automatic Query Expansion (AQE). Over the years a number of AQE techniques have increased that employ sophisticated methods for finding new features related to query terms. Today,

there are firm theoretical foundations, and a better understanding of the utility and limitations of AQEs. For example, what the critical parameters are that affect the performance of the IR system, what types of AQEs are useful and what not, etc.

Along with the AQE, the basic techniques are being increasingly used in conjunction with mechanisms to increase their effectiveness, like *method combinations*, selection of information source dynamically for expansion, and discriminating policies for the application of methods. These advances have been supported by many experimental findings. The AQE methods have gained their popularity due to the evaluation results obtained at the *Text REtrieval Conference* series (TREC).

Document Ranking using AQE

The IR systems including search engines depend on computing the importance of terms occurring in the query and documents to determine the relevance of document(s) to the queries. The commonly used indicator of relevance is the similarity measure between the query and the document. Considering that a query is represented by $q$ and a document by $d$, the similarity measure between them, $sim(q, d)$, is expressed by

$$sim(q, d) = \sum_{t \in (q \cap d)} w_{t,q} \times w_{t,d} \tag{18.13}$$

where $w_{t,q}$ and $w_{t,d}$ are the weights of term $t$ in a query $q$ and a document $d$, respectively, according to some weighting criteria adopted. The weight of a term $t$ in a document is typically proportional to the term frequency $(tf)$ in that document and to the inverse document frequency $(idf)$. The purpose of $idf$ is to diminish the effect of very frequent terms like "the" for which we are not interested to find the relevance. But we want to increase the effect of terms, which occur rarely, in a few documents only, for example, "blue" and "bird", which might be occurring only in a few documents. If $N$ is the total number of documents and $n$ is the number of documents in which the rare term occurs, the $idf$ for this term is $\log(\frac{N}{n})$. Note that if a term like "the" occurs in every document, then $idf$ is zero, hence it will not contribute to the similarity measure.

The similarity computation between a document and query representation, expressed in Eq. 18.13, can be easily modified by abstracting away from the original underlying weighting model, so as to work for the query expansion. In this revised model, the basic input to AQE module is 1. original query $q$ and, 2. source of data from which to compute the weight and the expansion terms. The output of the AQE module is the query formed (say, $q'$) due to the expansion of query terms, and their associated weights $w'$. These new weighted query terms $\langle q', w' \rangle$ are used for computing the similarity between the query $q'$ and the original document $d$. The new similarity computation formula is expressed by

$$sim(q', d) = \sum_{t \in (q' \cap d)} w'_{t,q'} \times w_{t,d}. \tag{18.14}$$

The commonly used data source for generating new query terms is the collection itself into which we are searching, and sometimes it is the thesaurus to get the synonyms of the keywords. The simplest approach to weight the query expansion terms is to use the same weighting function that is being used by the ranking system. When more complex features in the terms, like phrases, are used for query expansion, the underlying system must be able to handle such features. An example is "Jodhpur departmental store", which though a phrase, can be treated as a single term, for the purpose of indexing.

In the following we address some new areas, in addition to document ranking, where AQE is used heavily; these are: question answering, multimedia IR, information filtering, and cross-language IR. This introduction is followed by pointers to more recent applications [1].

Question Answering

The goal of question answering (QA) is to provide concise responses (instead of full-length documents) to certain types of natural language questions, such as "Who built the Taj Mahal?" Like the Document Ranking (DR), the QA is faced with a fundamental challenge of mismatch between the vocabularies of the question and answer.

To improve on the early stage of QA, which is document retrieval, a common strategy used is to expand the original question terms with terms that are expected to appear in documents containing answers to this question. One important goal of QA translation is to learn the associations between question words and the answer words, which may be synonyms to the words in question. For example, for the question, "Where Taj Mahal is located?," the answer part embedded in a document may be "Taj Mahal is in Agra," or "Taj Mahal is situated in the city of Agra." Here, the words "situated" and "exists" are the synonyms of the word "located". Different resources for AQE for QA include using lexical ontologies like WordNet—a lexical database of synonyms (called *synsets*), and semantic parsing of questions based on roles, and other criterias [7].

Multimedia Information Retrieval

With the widespread use of digital media and digital libraries, the requirements of searching the multimedia documents like image, speech, and video have become important. Up till recently, the multimedia IR systems performed only text-based search over the media *metadata*, like annotations and captions, which are surrounding the *html/xml* descriptions. However, when these metadata are absent, this method is not suitable. Hence, the IR relies on some form of multimedia content analysis, which is implemented in combination with the AQE techniques. For example, a transcription is produced by an automatic speech recognition system for *spoken document retrieval* systems, and this is augmented with related terms, in advance to raising the query. Since automatic speech transcriptions often contain mistakes, this form of a document expansion is very useful for spoken document retrieval.

For image retrieval systems, a typical approach is making use of query examples having visual features, like colors, textures, and shapes. The query is iteratively refined through a relevance feedback.

For video retrieval systems, both the documents and queries are mostly *multi-modal*, i.e., they have both textual as well as visual aspects. An expanded text query is usually compared against the textual description of the visual concepts, and the matched concepts are used for visual refinement.

Information Filtering

Information filtering (IF) is different from IR. It removes redundant or unwanted information from an information stream prior to presentation to a human user. Its main goal is the management of the information overload and increment of the semantic signal-to-noise ratio. The documents arrive continuously and the user's information needs evolve over time, as per the experience of the user. Some examples of information filtering are electronic news, blogs, e-commerce, and e-mails. There are two main approaches to IF: 1. *collaborative IF*, which is based on the preferences of like-minded users, and the other is 2. *content-based IF*. However, these techniques are said to bear a strong conceptual similarity to IR, because the user profile can be modeled as a query and the data stream can be treated as a set of collection of documents. The user profiles (i.e., queries) are learned using relevance feedback techniques, or other forms of query expansion, such as those based on similar users.

Cross-Language Information Retrieval

The Cross-Language Information Retrieval (CLIR) is concerned with retrieving the documents written in a language different than the language of the users' query. Earlier methods of such retrieval consisted of translating a query into the documents' language, and then using the standard techniques of IR. The query translation in these approaches was performed using machine-readable bilingual dictionaries, through machine translation, or using parallel corpora. However, no such translation is absolutely correct, and regardless of the translating resource used, there are usually limitations due to insufficient coverage, for example, there are terms which cannot be translated or do not require the translation, and due to the translation ambiguity from the source language to the target language.

To reduce the errors introduced due to translation, one standard technique is to use query expansion, so that even when the translation does not contain errors, use of semantically similar terms yields better results than those due to the literal translation of terms only. The query expansion can be applied either before the translation of query, or after, or even can be applied at both the places. It has been found that query expansion, before the translation of a query, yields better results than doing it after the translation. The expansion done at both the places provided even better results.
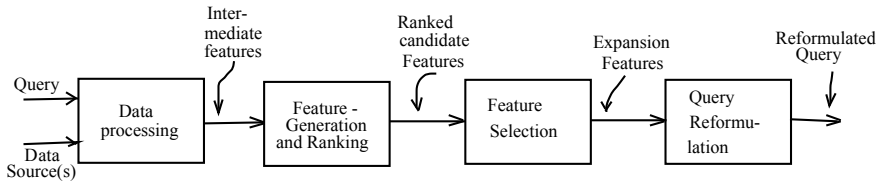
**Fig. 18.9** Automatic Query Expansion process

## 18.9.1   Working of AQE

The Automatic Query Expansion (AQE) is performed in a number of steps, the major steps are

1. Preprocessing the source text,
2. Generating expansion terms (features) and ranking them,
3. Selection of expansion features, and
4. Reformulation of Query.

Figure 18.9 shows these steps of AQE.

The objective of preprocessing the data source data is to transform the raw data source used for expanding the user query into such a format so that it can be processed more efficiently by the subsequent steps. The preprocessing task performs extraction of intermediate features, followed by the construction of data structures for easy access and manipulation of such features. The preprocessing is usually independent of a particular user query to be expanded, but it is specific to the type of data source and expansion method used [1].

To compute the initial retrieval run, it is necessary to index the documents collection, and then run the query against this collection's index. The process of indexing consists of, in order, the following steps:

1. text extraction from documents, which are in a certain format, like HTML, PDF, MS Word, where there is format information as well as text inside them,
2. tokenization of the extracted text,
3. stop-word removal from the tokenized text (removal of common words such as articles and prepositions),
4. stemming (reduction of inflected or derivational words to their root form), e.g., reduce "tokenize, building, training" into "token, build, train", etc.
5. word weighting (score is assigned to each token such that the weight reflects the importance of the tokens in each document).

We take an example of a short HTML text fragment to illustrate the weights associated with tokens.

```
<p><b> An automatic query expansion </b>
increases the query's semantics.</p>
```

In the above HTML document, first, text is extracted, then stop words "the" and "an" are removed, then it is stemmed using Porter's stemmer, and weight is assigned to each word based on their frequency. Finally, we obtain the text representation as follows:

*automat* 0.16, *queri* 0.33, *expan* 0.16, *increase* 0.16, *semantic* 0.16.

This is an example of a very small document, however, it gives an understanding that each document can be represented as a set of weighted terms, such that the total weight of the document is 1. The index is created in the form of a complementary *inverted index* file, which maps terms to documents at the time of query. To reach the location of index terms in the document faster, the system may also store the terms' locations to provide proximity-based search.

The original query is preprocessed to remove the *stop words* and/or extract important terms to be expanded. In the second stage of AQE, the system generates and ranks the candidate expansion terms; most query expansion methods choose only a small proportion of the expansion features (i.e., terms) to be added into the query. Input to this stage of AQE is the original query and the transformed data source, and the output is a set of expansion features, with/without scores.

Once the ranking of the candidate features is carried out, the top elements are selected for query expansion. Selection of these top elements is performed on an individual basis, without regard to mutual dependencies between the expansion features. Usually, only a limited number of features are selected for expansion such that 1. resulting query is not bulky, thus helpful for processing faster, and 2. retrieval effectiveness of a small set of good terms is not necessarily less successful than the effectiveness we get by adding all candidates' expansion features. The addition of expansion features will also be helpful in reducing the noise.

Sometimes, the feature scores are interpreted as probabilities. In that case, only the terms having a probability greater than a certain threshold are selected for consideration.

The last stage of AQE is query reformulation, which describes the expanded query that will be submitted to the IR system. The description means the assignment of proper weight to each feature that is part of the expanded query—the process called query *re-weighting*.

The total time required for an AQE is the sum of two factors, 1. cost of generating expansion features, and 2. increase in the cost for the evaluation of the expanded query (due to its size), against the documents collection. In practice, the second factor is a more critical one. Consider the architecture (data structure) of most ranking systems, which are based on the inverted (linked) lists of $N$ elements, one for each term in the collection. Here, each inverted list specifies the documents in which the particular term occurs, along with a pre-computed score for each term. At the time of query processing, the system retrieves the inverted list of every query term, and updates the score accumulators of the documents present in each list. The execution time of a ranked query is almost linearly proportional to the number of terms in the query;

this is because the query terms are processed one at a time. The AQE runs with sizes of practical interest, for example 10–20 word queries were found to be much slower (by a factor of ten), than the original queries of 3–4 words.

## 18.9.2   Related Techniques for Query Processing

The mismatch of words between the query and documents for relevant documents is an issue in IR for a long time. In the following section, we discuss AQE with respect to alternative strategies in reference to the vocabulary problem [1].

Interactive Query Refinement

In interactive query refinement, the system provides several suggestions for reformation of the query, and it is the user who selects the best choice out of that. With respect to the computations required to be performed, in Interactive Query Refinement (IQE) versus AQE, the first two stages are common with both, i.e., data acquisition and candidate feature extraction. The IQE does not follow the steps of feature selection and query reformulation of AQE. One of the best-known examples of Interactive Query Refinement is the suggestion of complete query, which offers real-time hints to user to complete a search query. This happens when a user progresses in typing the query in the inbox, like we note in many search engines, including Google. The IQE has better potential for superior results than AQE, but generally requires higher expertise on part of the user. Looking from the usability point of view, an IQE provides the user with better control over query processing than the AQE.

Relevance Feedback

The relevance feedback takes two inputs: 1. results initially returned due to the given query, and 2. feedback provided by the user about whether those results are relevant or not. Based on these two inputs, the system submits a new query to the search engine provided that previous results returned were not considered useful for fulfilling the need of the user.

The features in the assessed documents are used to adjust the weights of terms in the original query and/or for adding words to the query. The relevance feedback has the effect of reinforcing the system's original decision. This is done by the user by modifying the expanded query to look closer at the retrieved relevant documents. However, the AQE tries to form a better match with the user's existing intentions, and does not give the user a second chance to rethink, support/reject the results produced by the first query. The specific data sources using which the expansion features are generated in the relevance feedback may be more reliable than the sources generally used by AQE. In the relevance feedback, the user must assess the relevance of the documents, requiring a user to be better trained.

The relevance feedback has directly inspired one of the most popular AQE techniques, called *pseudo-relevance feedback*, which has also provided foundations for modeling query reformulation in a variety of AQE approaches.

Word Sense Disambiguation in IR

The Word Sense Disambiguation (WSD) is the ability of the system to correctly identify the senses of words in context to the remaining (surrounding) text in a document. This identification is carried out in a computational manner. WSD is a natural and well-known approach to the vocabulary problem in IR—usually, even if we do not know the meaning of a word, for example, in a newspaper, we are still able to understand the sentence, as well as able to discover the meaning of that unknown word due to its context words in a sentence [5].

Early work of WSD concentrated on representing words using their dictionary definitions, or using the WordNet Synsets. But, many experiments suggested that this straightforward technique is not effective in IR, at least as long as the selection of the correct sense from the resource is flawed. For example, if precision does result in a good value, say greater than 75%. However, more sophisticated methods in AQE still used the WordNet resource [11].

Instead of depending on short predefined lists of senses, using a corpus is found to be more convenient as an evidence for performing the Word Sense Disambiguation. Due to its nature of the process, it may be called as word sense induction.

In one approach based on the corpus, the context of every occurrence of a word in the corpus is identified and similar contexts are clustered together to help in determining the word senses. This method can provide a maximum disambiguation rate of 90%, hence can be used successfully with an IR system. With the reliance of this method on corpus-based analysis, this approach is similar in spirit to the global AQE techniques.

In another corpus-based WSD technique, a metaphor of small words is applied to word co-occurrence graphs, due to which it is capable of discovering low-frequency senses (senses which are not very common), that are as low as 1%.

Further, in the context of a query to web search engines, where the query is too small, it may be too difficult to disambiguate the word senses in it in the absence of sufficient context available in the query. Therefore, longer queries due to their higher contexts are likely to be helpful in performing the disambiguation, and consequently to produce better results. As a whole, we note that the application of WSD to IR presents the challenges of computational nature, there are limitations of effectiveness as well.

Search Results Clustering

The objective of Search Results Clustering (SRC) is that for the new queries, exactly similar to some previous queries by users, the IR system should store the previous results in some compact forms, so that it can provide the results directly, without performing any search process. For this, the SRC organizes the search results topic-wise, which allows direct access to the documents relevant to the user queries, making overall IR far faster. In contrast to the standard clustering techniques, the SRC algorithms try to optimize the clustering structures, as well as the quality of cluster labels. This is because, a cluster with a poor description (labels) is likely to be entirely omitted by the user, even though it may be pointing to a group of strongly related and relevant documents.

## 18.10   Using Bayesian Networks for IR

A Bayesian network is an annotated directed graph, which can be used to encode a probabilistic relationship among the distinctions of interest in an uncertain reasoning problem. The representation rigorously describes these relationships, and can provide a human-oriented qualitative structure which facilitates a communication between a user and the system based on a probabilistic model. As the computing power is available chiefly even in small systems, the modeling tools based on Bayesian networks are abundantly used in real-world applications, e.g., in forecasting, fault diagnosis, sensor fusion, anti-virus software, automated vision, and manufacturing control [10], [13].

### 18.10.1   Representation of Document and Query

For understanding the basics of Bayesian networks as well as Naive Bayes, the reader may refer to the previous chapters (section 12.4, page no. 344, and section no. 14.7, page no. 428). Using the conditional probability, the probability that a document $d_k$ is relevant to the query $q_j$ can be expressed as $P(R \mid q_j, d_k)$. An accurate definition of probability of relevance depends on the definition of relevance itself. The term *relevance* is to some extent a subjective entity, and depends on many variables, which are functions of documents, user's information need, and the user itself. A perfect retrieval is not achievable in true sense. However, it is possible to define an optimal retrieval for the probabilistic model for IR. This optimal retrieval can be proved theoretically with respect to representations (or descriptions) of documents and information needs [4].

Let us assume that collections of queries and documents are described by a set of index terms. Let $T = \{t_1, t_2, ..., t_n\}$ be the set of terms in the documents' collection, and a query $q_j$ and document $d_k$ are taken as subsets of terms in $T$. For the sake of retrieval, each document is described by the presence/absence of these index terms. Therefore, any document $d_k$ can be represented using a *binary vector*:

$$\overrightarrow{x} = (x_1, x_2, \ldots, x_n), \tag{18.15}$$

where any term $x_i = 1$ if $t_i \in d_k$, and for $t_i \notin d_k$, the $x_i = 0$. A query $q_j$ is also represented in similar way. The basic task of a relevance model-based IR system is to compute the probability that a given document is relevant. This can be achieved by estimating the probability $P(R \mid q_j, d_k)$, for every document $d_k$ in the collection. Since relevancy in every document is computed for a single query, the term $q_j$ being common, and can be dropped, and the relevancy can be expressed using the Bayes theorem as

$$P(R \mid \overrightarrow{x}) = \frac{P(\overrightarrow{x} \mid R)P(R)}{P(\overrightarrow{x})}. \tag{18.16}$$

In the above,

$P(R \mid \overrightarrow{x})$ is called *posterior probability*—the probability of relevance, given that the document is $\overrightarrow{x}$,

$P(\overrightarrow{x} \mid R)$ is called *likelyhood function* or *probability of evidence*, which is the probability of randomly selecting the document of description $\overrightarrow{x}$ from the set $R$ of relevant documents,

$P(R)$ is *prior probability* of relevance, i.e., the probability that a randomly selected document from the entire collection is relevant, and

$P(\overrightarrow{x})$ is probability that a randomly selected document has a description $\overrightarrow{x}$. It is determined as a joint probability distribution of the *n* terms in the collection [6].

Equation (18.16) can be expressed in simple language as

$$Posterior\ probability\ \propto\ likelihood \times\ prior\ probability.$$

### 18.10.2   Bayes Probabilistic Inference Model

The Bayes probabilistic inference makes use of a network, which is an extension to the probability-based IR. The network is a Directed Acyclic Graph (DAG), in which nodes represent *propositional variables* or constants, and the edges represent the dependency relations between the propositions [10].

If *p* and *q* are two propositions, and there is a relation of implication from *p* to *q*, i.e., the first proposition "causes" the second, then *p* is cause and *q* is effect, and is represented by $p \rightarrow q$. In the DAG, *p* and *q* are nodes, and there is an edge from a node marked as *p* to node marked as *q*. A link matrix is stored at node *q*, which specifies the probability $P(p|q)$ for all possible values of variables *p* and *q*. The expression $P(p|q)$ is the expression for the probability of occurrence of event *p* given that *q* has already occurred. In the model we take *q* node as evidence, and it stands for a query. In a scenario, a node has more than one parent (say $p_1$, $p_2$, ...), the link matrix will indicate the dependence of query node *q* on all the parents. The query node *q* now characterizes the dependence relationship between itself and all the nodes $p_1$, $p_2$, ..., which are potential causes. This is illustrated in Fig. 18.10. Using the Bayes theorem, the conditional probability expression can be expanded as

$$P(p|q) = \frac{P(q|p).P(p)}{P(q)}. \tag{18.17}$$

When the set of prior probabilities $P(p)$ are given for the root of a DAG that represents the document, the network can be used to compute the probability of belief associated with all the remaining nodes. Figure 18.10 shows a document $d_i$ at

**Fig. 18.10** Basic Inference
Network Model for IR
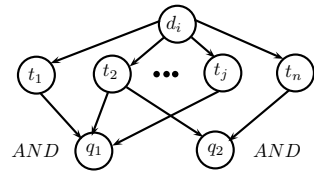


root, corresponding keywords $t_1, \ldots, t_n$, and submitted queries $q_1, q_2$. Note that, $t_1$, ..., $t_n$ are the representation of document $d_i$ [17].

Through the inference network, the *random variables* are associated with the documents, index terms, and user queries. Multiple evidences of query terms in the document's representation for a given query are conjuncted to estimate the probability that the document satisfies the user's information need. For example, in Fig. 18.10, query $q_1$ is conjunct (ANDed) of terms $t_1$, $t_2$, $t_j$, and $q_2 = t_2 \wedge t_j \wedge t_n$. Thus, variables associated with document $d_i$ represent the event that the document is observed. The index terms/document variables are represented as nodes of the DAG, and the edges, which are directed from document nodes to index terms nodes indicate that the observation of document results in an improved belief on its term nodes.

Further, a random variable associated with the user's query node models the event that information need expressed in the form of user's query has been met. The dependency in the form of direction arrows indicate that belief in the query node is function of beliefs in the nodes that correspond to the query terms. In Fig. 18.10, document $d_i$ comprises $t_1, t_2, \ldots, t_j, t_n$ as its index terms. Similarly, the query $q_1$ comprises the query terms $t_1, t_2, \ldots, t_j$, hence, $q_1 = t_1 \wedge t_2 \wedge t_j$, and $q_2 = t_2 \wedge t_n$.

From this Bayes inference model for IR, we note that a set of edges pointing to a node represents the probabilistic dependence between that node and its parents. Through its structure, the Bayes network represents the conditional dependence relation among the variables in the network. These dependence relations provide a framework for retrieving the information [4, 6].

### 18.10.3 Bayes Inference Algorithm

For IR using Bayes inference, a user specifies one or more topics of interest while keeping in mind some document features, the latter are to be used as evidence for topics of interest mentioned above. The task of IR using Bayesian inference network is documented as Algorithm 18.5, which requires building an inference network for the representation of query terms and document features (i.e., terms), and computation of posterior probabilities ($P(p|q)$ in Eq. 18.17) based on the prior probability of the document [6, 10].

---

**Algorithm 18.5** Bayesian Inference-based IR

---

1: *Construct the network representation of query*
2: *{Steps to score all the documents}*
3: **for all** documents **do**
4:    *Extract the features $\{t_1, t_2, ...\}$ from document*
5:    *Label features in network*
6:    *Compute posterior probabilities, $P(p|q)$*
7: **end for**
8: *Rank the documents set in order of posterior probabilities*

---

**Fig. 18.11** Two-Level
Bayesian network model for
IR
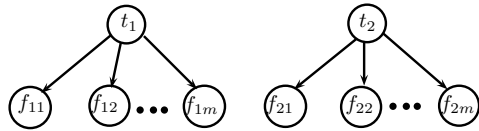


In this algorithm, steps 4, 8 are IR routines, whereas steps 5, 6 are routines to draw inferences.

Figure 18.11 shows the term-weighting architecture of Bayesian network, which indicates that there are topics of interest, shown as $t_1, t_2$, and there are features to be examined, $f_{11}, ..., f_{1m}, f_{21}, ..., f_{2m}$. The features' set and topics set shown here are different, but they can be the same also. The occurrence of the topics $t_1, t_2$ on graph nodes represent the event that the document is related to topics $t_1, t_2$. Whereas, the nodes corresponding to the features represent the events, for example, the features $f_{11}, ..., f_{1m}$ represent the event that these features are present in the document $d_1$.

The network structure in Fig. 18.11 is based on the following assumptions concerning the Bayesian probability:

1. Given the topics $\{t_i, t_2, ...\}$ from the document (the document is relevant to these), the presence or absence of any feature does not imply about the presence or absence of some other feature. In other words, it is assumed that there are no dependency relations between the features.

2. Given that the document relevant to one topic does not affect one's belief about the relevance or non-relevance of that document to any other topic.

In the above, the first assumption specifies the conditional independence of features, when the topic is already given. It is called as *binary independence*. The absence of arcs between feature nodes in Fig. 18.11 is an explicit indication of independence between features. Given these conditions and the Bayes network in Fig. 18.11, we can draw some important conclusion. That is, the assumption "1" will not be valid, if the query includes features that are identical or closely related, like synonyms, because in that case the features are not independent.

Now, for the network in Fig. 18.11, we define two sets of probabilities, given below:

(a) $P(t_1), P(t_2), ..$, called *prior probabilities*, that a document is relevant to topics $t_1, t_2, ...$, etc. If we are discussing one document only, then it is $P(t_1, t_2, ..)$.

(b) The *conditional probability* $P(f_{ij}|t_i)$ of each feature $f_{ij}$ is defined as the probability that feature $f_{ij}$ is present in the document, given that this document is relevant to topic $t_i$.

Given the above, the task of IR is to compute the *posterior probability* $P(t_i \mid f_{i1}, \ldots, f_{im})$, which is the probability that the document is relevant to $t_i$, given that we have observed the presence or absence of all of the features $f_{ij}$, called evidences.

The Bayes rule can be directly applied for this computation. The network topology shown in Fig. 18.11 is called *Bayes inference* and, can be expressed by

$$P(t_i \mid f_{i1}, \ldots, f_{im}) = \frac{P(t_i).P(f_{i1}, \ldots, f_{im} \mid t_i)}{P(f_{i1}.\ldots, f_{im})}. \tag{18.18}$$

Generally, we are not interested in absolute numerical values of the posterior probabilities, but want to just rank the documents by the posteriors. Thus, we can eliminate the denominator term $P(f_{i1} \ldots, f_{im})$ in this equation as long as this denominator remains the same, with varying $t_i$,s. Further, we can simplify this Bayes rule to a *linear decision rule* given in Eq. 18.19, where $I(f_{ij})$ is an indicator variable that equals to 1.0 only if $f_{ij}$ is present in the document and 0.0 otherwise, and $w$ is a coefficient corresponding to a specific (feature, topic) pair:

$$g(t_i \mid f_{i1}, \ldots, f_{im}) = \sum_j I(f_{ij}) \times w(f_{ij}, t_i). \tag{18.19}$$

A careful choice of $w$ results in a ranking of documents in descending order of $g()$, which turns out to be in the same order as that of ranking them in decreasing order of the posterior probabilities. However, since $g()$ does not include the priors probabilities ($P(t_i)$) of the topics $t_i$, which is indicator of the relative *rarity* of the topics, one cannot compare a document's strength of relevance to one topic with respect to its strength of relevance to a different topic.

The coefficients $w$ can be interpreted as weights corresponding to each feature $f_{ij}$ and term $t_i$. Similarly, the function $g()$ can be interpreted as the sum of weights of the features $f_{ij}$ that are present in the document, which is relevant to topic $t_i$. Hence, this method is known as "term weighing".

In the Bayes network topology shown in Fig. 18.11, the query corresponding to each topic, e.g., $t_1$, is represented by its own subnetwork, which is shown disconnected from the subnetworks of other topics' queries. Hence, these isolated models fail to represent the possible relationships between the topics, for example between $t_1, t_2$. Therefore, it is difficult to acquire consistent, feature-conditional probabilities, as well as find out the probabilities by combining the topics.

**Fig. 18.12** Information
retrieval model with two
related topics
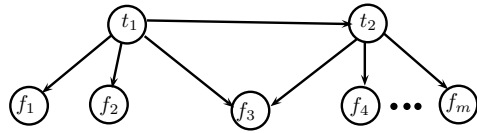


**Fig. 18.13** Multi-topic
query as a single
compound-topic query
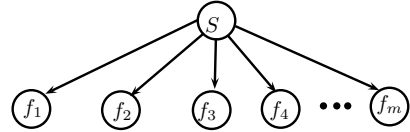


### 18.10.4   Representing Dependent Topics

In the previous section, we discussed that, a document relevant to one topic does
not affect one's belief about the relevance or non-relevance of that document to any
other topic.

   An IR network topology that removes this assumption, and explicitly represents
the relationships among different topics is shown in Fig. 18.12. We note that the
Bayesian inference problem becomes more completed with multiple topics.

   The addition of a relationship between topics in Fig. 18.12 requires two changes in
specifying the network probabilities. In the first one, we must specify what are those
topics ($t_1$, $t_2$, ...), and then compute the strength of the relevance between them. In
the second step, we must compute the probability of each feature ($f_{ij}$), conditioned
on each combination of its parent topic. Having done in this manner, it gives rise to
a number of probabilities, which are combinatorial in size.

   However, there is an approach to simplify the multiple-topic network, such that it
looks like a single-topic network, and its range is only all the possible present/absence
combinations of topics $t_1$, $t_2$, ..., etc. Thus, in a way we have translated the topology
from one form to another. The modified topology of Fig. 18.12 is shown in Fig. 18.13,
where node $S$ represents the compound topic. An advantage of this representation is
that the same set of formulas (18.18) and (18.19) for the computation of probabilities
can be used now also. The disadvantage of this approach is that the compound query
shall contain $2^n$ states for $n$ parent states, which will complicate the computations.

## 18.11   Semantic IR on the Web

The semantic web is a vision of the future WWW. It is an extension of the current
web, where information is given with well-defined meanings, that will better enable
the computers so that computers and people can work in cooperation. However, a
universal implementation of the semantic web—a full substitution for the existing
web—is still far away from reality. Therefore, it could be useful to have a system
that analyzes the documents from a contextual point of view for more accurate

retrieval. The semantic IR (SIR) on web is based on computing semantic relations to evaluate the relevance of documents with a query in a given context, and makes use of structures like lexical chains, semantic networks, and ontologies. The semantic-based approach is context-driven, where keywords (topics/terms) in documents are processed in the context of the information in which they are retrieved. This will help solve the semantic ambiguity so that the retrieval is accurate, and as per the true need of the user [15].

In the recent times, the information and knowledge representation using ontologies have acquired great importance, as it is found to be suitable for strategic requirements. These strategies are intrinsically independent on information codification, which helps to isolate the information, as well as to recover, organize, and integrate it with respect to its content.

Following are the definitions of ontologies.

**Definition 18.1** *Ontology.* An explicit and formal specification of shared conceptualization is called an ontology. It is an abstract model of specified reality, such that the components are clearly identified. The terms in definition are further clarified as follows:

– *Explicit* means type of concepts used and the constraints on them are well defined,
– *Formal* refers to the ontology property of being machine-readable, and
– *Shared* means, a property of ontology of capturing consensual knowledge, accepted by a group of persons.                                                                   □

**Definition 18.2** *Ontology* (Definition-2). An ontology defines the basic *terms* and their *relations* consisting the vocabulary of a topic, as well as the *rules* for combining terms and relations to define extensions to the vocabulary.                             □

The above definition also indicates a path to be followed in order to construct an ontology:

1. first identify the basic terms and their mutual relations;
2. agreement on the rules that arrange them;
3. defining of terms, and relations among concepts.

From the above perspective it is clear that an ontology does not include just the terms that are explicitly defined in it, but keeps provision to derive new terms using defined rules and properties. Also, the ontology can be viewed as a "set of terms and relations between them, which denote the concepts used in a domain."

The concept of semantic relatedness refers to the relations between words and concepts that are in the practice, or those based on the perceptions. There can be several metrics to measure the semantic relatedness of words. Some of the approaches to these metrics are as follows.

• *Thesaurus-based metrics.* These metrics make use of thesaurus where words are related to concepts, and each word is referred to a category by an index structure.

- *Dictionary-based metrics.* Dictionaries are linguistic information sources of our knowledge about the world; they form a knowledge base in which headwords are defined using other headwords and/or their derivatives.
- *Semantic network-based metrics.* These metrics use semantic networks—graphs in which the nodes are the concepts, and the arcs between nodes represent relations between concepts. Number of edges' links between terms (nodes/concepts) in the semantic network, without loops, is a measure of conceptional distance between terms (nodes).

A sequence of related words in a text is called *lexical chain* (a linguistic structure), which may span short distances (adjacent words or sentences) or long distances, covering the entire text. Computing a lexical chains helps in the identification of the main topics of a document. The semantic relatedness measures use lexical chains to perform their computations, the lexical chains are used for IR and related areas, and to explore structure of texts as well. The lexical chains have been also used to index video-conference transcriptions by topic, construction of typical IR system and text segmentation systems, and for automatic generation of hypertext links.

The strength of a relation between words that connect different fragments of the text is measured by their *cohesion*, and the cohesion between lexical units of text is called *lexical cohesion*—the most common type of cohesion. The lexical cohesion can be expressed by repetitions of relations like *synonym*, *hyponym*, or by other linguistic relations between words, such as whole-part and object-property.

Semantic IR Systems

Use of ontologies in IR consists of an approach that identifies important concepts in documents using criteria of semantic relatedness and co-occurrence; this is followed by disambiguation of them using an external general purpose ontology (e.g., Word-Net). On matching the ontology with a document provides a set of scored concept senses (nodes) with weighted links, called semantic representation of the document.

The steps for the process of Semantic IR (SIR) are as follows:

1. For improving the web searches, only important information is selected from the user query, which is helpful in extracting information from documents;
2. The user's query or a phrase expressed in natural language is sent to a lexical processing module;
3. The boundaries of words and phrases are detected through tokenization. The system labels the words using some tagger like Brill's tagger;
4. A phrase parser splits every phrase into several members, as nouns and verbs;
5. The stop words are removed, and the system uses some keywords to represent the main concept of the phrase;
6. The remaining process steps of SIR, like Word Sense Disambiguation (WSD), query expansion, and post-processing, have been already discussed in this chapter.

To solve the problem of polysemy,[5] the concept of "word sense" from WordNet is used, which helps the user interacting with the system to associate with every concept a list of terms semantically are related to.

## 18.12 Distributed IR

When the size of data is very large, or when it is required to support high query volumes, single machine is not enough to support the load of IR, even when the number of the enhancements discussed above have been used. For example, even when the Internet was not so common, in mid-2004, Google search engine processed more than 200 million queries a day against more than 20 GB of crawled data, and it used over 20,000 computers. The requirement in the present time is hundred times bigger. For handling large size of data loads, a combination of *distribution* and *replication* is necessary. The distribution means, document collections and their indexes are split across multiple machines, so that answers to every query is synthesized from many collections of components. The word replication means *mirroring*, which involves making enough identical copies of the system so that the required query load can be handled with acceptable minimum response time [18].

Document-Distributed Architectures

A very simple distributed document's architecture is to partition the collection and allocate one sub-collection to each of the separate processors (see Fig. 18.14). To make use of distributed document architecture, a local index is built and maintained for each sub-collection, and when a query arrives, it is passed to every sub-collection to search and evaluate in parallel against every local index. The sets of sub-collections' answers are then merged in some way to provide an overall answer. The main advantages of such document's partitioning system is that, collection growth is handled by designing one of the hosts in the form of dynamic collection, such that only this host needs to rebuild its index. The parts of the process that are computationally expensive are distributed equally across all the hosts in the computer cluster. These parts are searching the index, updating individual indexes, computation of weights, and $idf$ for documents, etc.

Figure 18.14 shows a simple inverted index of a document-distributed retrieval system. It shows two index partitions: 1. a term-based index partition, and 2. document-based index partition. Elements in a inverted list have format ⟨document number, term-frequency⟩, for example ⟨2, 2⟩ in second row, second column indicates that the term "quick" has frequency 2 in the document number 2, while ⟨3, 1⟩ in the same row, third column indicates that the term "quick" has frequency 1 in the document number 3. Each of the two dashed regions in Fig. 18.14 show one component of a document-distributed retrieval system; with this, one processor indexes all terms

---

[5]Polysemy: A single term with several meanings.

Document number
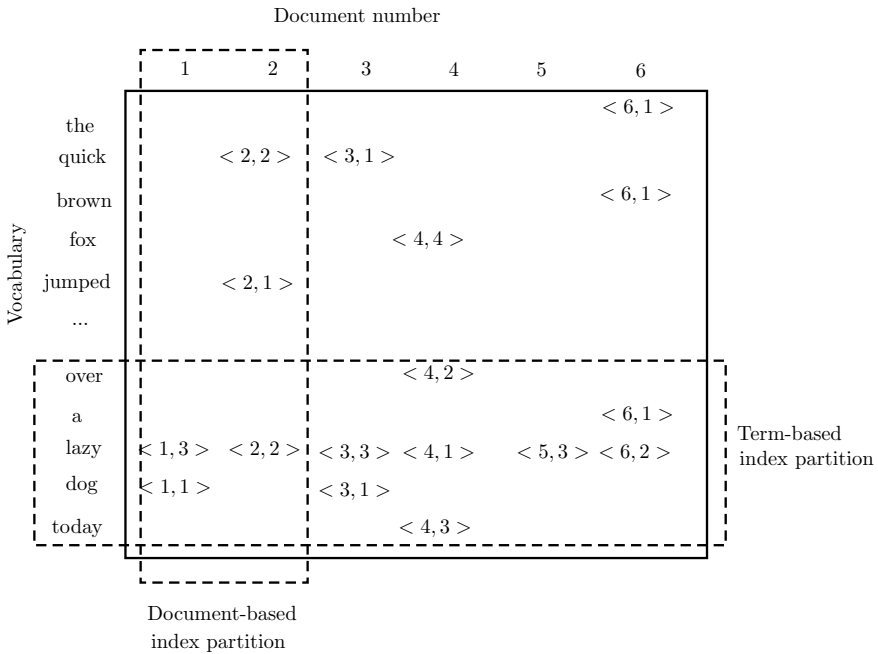


**Fig. 18.14** Partition and distribute index across a cluster of machines

that appear in the first two documents of the collection, creating a document-based partition.

Term-Distributed Architectures

An alternative strategy for partitioning the index is term-based partitioning, where index is split into components by partitioning the vocabulary, with one possible partition shown by the dotted horizontal split in Fig. 18.14. Each processor has full information about a subset of the terms, i.e., all the necessary things to handle a query. Hence, only the relevant subset of the processors needs to respond to a query. The term partitioning has the advantage that it requires fewer disk seek and transfer operations during query evaluation than document partitioning because each term's inverted list is still stored contiguously on a single machine rather than being split in fragments across multiple machines.

On the other hand, each of these disk transfer operations involve more data. Mainly because, in a term-partitioned arrangement we are discussing, the majority of the processing load is on the coordinating machine; the experiments have shown that it can easily become a bottleneck and starve the other processors of work.

Term-Based Versus Document-Based Partitioning

Compared to the term-based partitioning, the document-based partitioning typically results in a better balance of workload and achieves higher throughput for queries.

Also, the document-based partitioning (document distribution) allows for more natural index construction and for document insertion. On the other side, for term-partitioned index, index construction involves first of all, distributing the documents and building the document-partitioned index. Then, once the vocabulary split has been agreed upon by the processors, the index fragments are exchanged between all pairs of processors.

The document-based partition also has the practical advantage of providing the search service even when one of the hosts is offline for some reason, because any answers not resident on that machine are still available to the system. For example, there are 10 machines, $m_1...m_{10}$ and 100 documents $d_1...d_{100}$, with document distribution as follows:

machine $m_1 : d_1...d_{10}$,

...

machine $m_{10} : d_{91}...d_{100}$.

Now, if machine $m_5$ is offline, the queries for all the documents, except $d_{51}...d_{60}$, is answered. Whereas, in the term-based distribution, since index terms of every document are uniformly distributed on all the processors, the query cannot be answered even if one processor is offline. However, in the term distribution case, if any machine is offline or idle, it is immediately noticeable because it will affect queries belonging to almost every document.

Google indexing uses document-based partitioning, with massive replication and redundancy at all the levels, for example at machine level, cluster level, and individual level. In addition, the document partitioning remains effective even if the collaborating systems are independent and unable to exchange their index data. The distributed system makes use of a *meta-searcher*, using which the final result answer list is synthesized from the possibly overlapping answer sets provided by a range of different services.

## 18.13 Summary

Information retrieval (IR) process is the identification of documents or other units of information in a collection that are relevant to particular information needs expressed through queries for which people are interested to find answers. The IR models consider that each document is described by a set of representative keywords, called *index* terms—a word from a document that represents the semantics of the document. In general the index terms are *nouns*. If $t_i$ is an index term, $d_j$ is a document then $w_{i,j} \geq 0$ is the weight associated with the pair $(t_i, d_j)$, or single entity $w_{ij}$.

Two ratios, *precision* and *recall* are used to measure the effectiveness of an IR system. Precision is the ratio of the number of relevant documents retrieved to the total number of documents retrieved, i.e., what fraction of the retrieved documents are relevant, and recall is the ratio of the number of relevant documents retrieved to

the total number of relevant documents in the entire collection, i.e., what fraction of the relevant documents have been retrieved.

The common retrieval strategies are Boolean Model, Vector Space Model, Probabilistic Model, Inference networks, and Fuzzy set-based retrieval.

The *Boolean* model is a simple retrieval model based on Set Theory and Boolean Algebra, Vector Space Model computes the measure of similarity ($sim(q, d_j)$) between query $q \in Q$ and a document $d_j \in D$, where $q$ and $d_j$ are vectors for query and document, and the $Q, D$ are sets for queries and documents, respectively.

The probabilistic retrieval model computes the similarity measure ($sim(q, d_j)$) between the query $q$ and a document $d_j$ as the probability that document $d_j$ is relevant to $q$. This model estimates a query *term's weight* on how often the term appears or does not appear in *relevant* and *non-relevant* documents, respectively. One class of probabilistic approach for IR is Bayesian networks. These networks are annotated directed graphs encoding probabilistic relationship among distinctions of interest, in an uncertain reasoning problem.

The fuzzy retrieval technique is based on fuzzy set theory and fuzzy logic—an extension of the classical set theory. The word matching between the query set and the text word is not limited to the perfect matching, but, the matching is graded, depending on the degree or level of matching in the range from 0 to 1. In *fuzzy set* -based retrieval, a membership value $R(x_i, y_i)$ specifies for each $x_i \in X, y_i \in Y$, the grade of relevance of index term $x_i$ with the document $y_i$.

In a *concept-based* information retrieval, queries and documents are represented using *semantic concepts*, instead of (or in addition to) keywords, and they perform retrieval in that concept space. This results in a retrieval model that is less dependent on the specific terms used, and yields matches even when the same sense is described by different terms in the query and target documents.

The retrieval models make use of an *inverted index* file structure. Instead of searching into a document, an inverted index is generated in advance for all the keywords in the document set. For each term, a pointer references to a linked list, which contains an entry for each document containing this term as well the term frequency in that document. The single key problem with this index is that volume of data involved cannot be held in the main memory. To solve this problem, the indexes are constructed in parallel and can be merged after a regular intervals.

Yet, there is another method for IR that uses an approach based on the measure of semantic relatedness, applied to evaluate the relevance of a document with respect to a query in a given context. The approach makes use of structures like *lexical chains*, *ontologies*, and *semantic networks*. The semantic approach implements a context-based system, such that keywords are processed in the context of the information from which they are retrieved. This approach helps in solving semantic ambiguity, and results in giving a more accurate retrieval, that is based on the real-world interests of the user.

In most information retrievals, user queries are short and the natural language is inherently ambiguous, consequently, the retrieval is prone to errors and omissions. The most critical language issues are polysemy and synonymy. To resolve this, the

query is expanded by appending the synonyms of query terms into the query, called Automatic Query Expansion (AQE).

When large volumes of data sets are involved or when the query volumes are high, one machine may be inadequate to support the users' query load, even when the various enhancements and optimizations are carried out. For handling heavy load of users' queries, a combination of *distribution* and *replication* is required. Distribution means, the document collection and their indexes are split across multiple machines (servers) and that answers to the query as a whole must be synthesized from the various collection components. Replication (or mirroring) involves making enough identical copies of the system so that the required query load can be handled at speed and accuracy.

## Exercises

1. Show how the vector space model can be modeled using an inference network.
2. Consider a documents collection made of 100 documents. Given a query $q$, the set of documents relevant to the users is $D^* = \{d_4, d_{15}, d_{34}, d_{56}, d_{98}\}$. An IR system retrieves the following documents $D = \{d_4, d_{15}, d_{35}, d_{56}, d_{66}, d_{88}, d_{95}\}$

   a. Compute the number of True-Negatives, True-Positives, False-Negatives, False-Positives.
   b. Compute Precision, Recall, and F-measure.

3. Consider an IR scenario in the following: It has been found in some hospital, results of blood tests taken on a specific day are unreliable for diabetic patients due to equipment malfunction. The hospital uses an IR system to identify these patients. Suppose the collection of patients' records contains 10, 000 documents, 500 of that are relevant to the query. The system returns 350 documents, 225 of that are relevant to the query. Answer the following for this scenario:

   a. Calculate the precision and recall for this system.
   b. Based on your results from above, explain how well would you say about the working of hospital's IR system.
   c. Knowing about the precision-recall trade-off, what is likely to happen if an IR system is tuned to aim for 100% precision?
   d. Knowing about the precision-recall trade-off, what is likely to happen if an IR system is tuned to aim for 100% recall?
   e. For the trade-off given scenario, which measure do you think is more important, precision or recall? Why?

4. You are looking for information on "Economic growth in India" in a large document collection, during the period of last 3 years. You decide to search using the terms: *India, banks, growth, economy, business, agriculture*, using an IR system, which recommends three possible documents given below with term frequencies.

| Term | Economy | India | Growth | Banks | Business | Agriculture |
|---|---|---|---|---|---|---|
| Document-1 | 15 | 10 | 3 | 4 | 2 | 9 |
| Document-2 | 0 | 0 | 9 | 8 | 7 | 8 |
| Document-3 | 4 | 2 | 4 | 4 | 6 | 10 |

There is no additional information about the documents. Make use of each of the following models to find out the relevancy of the documents to the query.

   a. Boolean model
   b. Vector space model
   c. *tf.idf* model

5. Take any three small documents of size, approximately 100 words.

   a. Build a matrix of an inverted index for these documents, in the format shown in Fig. 18.5.
   b. Weight terms by their presence/absence (binary), and also and by $tf \times idf$ (with estimated IDFs).
   c. Compute the memory requirements for this inverted index. Make necessary assumptions for character size, pointer size, etc.
   d. Construct a suitable query, and calculate document–query similarity, for the following scenarios:
      i Cosine (with normalization)
      ii Inner product (i.e., cosine without normalization)
      iii Does the normalization has any effect? Justify.

6. Consider that we submit the queries to search engines for searching the needed information on WWW.

   a. Does the search process use a stop-word list?
   b. Can you search "The", "The a", "An a", etc.? Justify.
   c. Is it a practice to search the above terms?
   d. Does the search process use stemming?
   e. Are there different results for two queries "Human body", "Humanly body". Justify your answer.
   f. Does it normalize words to lower case?

7. "Having the knowledge of the sense of a query term may help a document retrieval system, especially for short queries." Why it is not true for longer queries?
8. Comment on the validity of following statements for Boolean model:

   a. "Stemming does not lower the precision of a Boolean retrieval system."
   b. "Stemming does not lower recall of a Boolean retrieval system."

9. Answer the following in brief:

    a. Why is the *idf* of a term always finite?
    b. What is the *idf* of a term that occurs in every document?
    c. What is the *idf* of a term that appears in one document only?
    d. What is the *idf* of a term that appears in no document?

10. Answer the following in brief:

    a. Name three criteria for evaluating a search engine.
    b. What is an easy way to maximize the recall of a search engine?
    c. What is an easy way to maximize the precision of a search engine?

11. What is the difference between *clustering* and *classification*? How can they be used in a complete IR system?

12. Discuss the merits and demerits of following, suggest as to which one will provide better response time?

    a. Document-distributed architecture.
    b. Term-distributed architecture.

# References

1. Carpineto C, Romano G (2012). A Survey of Automatic Query Expansion in Information Retrieval. ACM Comput. Surv. 44(1): 50.https://doi.org/10.1145/2071389.2071390
2. Chowdhary K R, Bansal VS (2001) Current trends in information retrieval. In: The 4th International Conference of Asian Digital Libraries, Dec. 10–12, 2001 Bangalore, pp. 306–319
3. Chowdhary K R, Bansal VS (2003) Fuzzy Logic-based information retrieval. In: Conference proceedings on algorithms and artificial systems, Allied Publishers Pvt. Ltd. pp 297–307. ISBN 81-7764-403-3
4. Chowdhary KR (2004) Natural language processing for word sense disambiguation and information extraction. PhD Thesis, J.N.V. University, Jodhpur, May 2004
5. Chowdhary KR (2005) Word sense disambiguation. J Comput Sci 1(1):30–37
6. Chowdhary KR (2008) Information retrieval from digital libraries using probabilistic-possibilistic inferences. In: IR@INFLIBNET INFLIBNET's Convention Proceedings CAL-IBER 2008 Allahabad, http://ir.inflibnet.ac.in/handle/1944/1225
7. Chowdhary KR, Bansal VS (2006) Information extraction from natural language texts. J Institut Eng (India), 87:14–19
8. Chowdhary KR, Bansal VS (2011) Information retrieval using probability and belief theory. International conference emerging trends in networks and computer communications (ETNCC). https://doi.org/10.1109/ETNCC.2011.5958513
9. Egozi O et al (2011).Concept-based information retrieval using explicit semantic analysis. ACM Trans Informat Syst, 29(2):8.1–8.34. https://doi.org/10.1145/1961209.1961211
10. Fung R, DelFavero B (1995) Applying Bayesian networks to information retrieval. Commun ACM 38(3):42–49
11. Miller GA (1995) WordNet: a lexical database for English. Commun ACM 38(11):39–41. https://doi.org/10.1145/219717.219748
12. Grossman DA, Ophir F (2008) Information retrieval-algorithms and heuristics, 2nd edn. Springer
13. Heckerman D et al (1995) Real-world applications of Bayesian networks. Commun ACM 38(3):24–26
14. Recardo BY, Berthier RN (1999) Modern information retrieval. Addison Wesley-ACM Press

15. Rinaldi AM (2009) An ontology-driven approach for semantic information retrieval on the web. Trans Internet Technol 9(3):10:1–10:24. https://doi.org/10.1145/1552291.1552293
16. Smith LC (1976) Artificial intelligence in information retrieval systems. Informat Process Manage 12:189–222. Pergamon Press
17. Wright (1921) Correlation and causation. Agric Res 20:557–585
18. Zobel J, Moffat A (2006) Inverted files for text search engines. ACM Comput Surv 38(2):1–56

# Chapter 19
# Natural Language Processing

**Abstract** The abundant volume of natural language text in the connected world, though having a large content of knowledge, but it is becoming increasingly difficult to disseminate it by a human to discover the knowledge/wisdom in it, specifically within any given time limits. The automated NLP is aimed to do this job effectively and with accuracy, like a human does it (for a limited of amount text). This chapter presents the challenges of NLP, progress so far made in this field, NLP applications, components of NLP, and grammar of English language—the way machine requires it. In addition, covers the specific areas like probabilistic parsing, ambiguities and their resolution, information extraction, discourse analysis, NL question-answering, commonsense interfaces, commonsense thinking and reasoning, causal-diversity, and various tools for NLP. Finally, the chapter summary, and a set of relevant exercises are presented.

**Keywords** Natural language processing · Challenges of NLP · Natural language parsing · Probabilistic parsing · NL ambiguities · Ambiguity resolution · Commonsense interfaces · Commonsense thinking · Commonsense reasoning · Discourse analysis · Question-answering · Causal-diversity · NLP tools

## 19.1 Introduction

A language is a set of finite length sentences, constructed using a finite alphabet set, or in terms of language syntax, they are constructed using a finite vocabulary of symbols. Since the alphabet set is finite, as well as the length of the sentences, the set of sentences (in a language) is also finite. For example, if alphabet set is of size two, and length of sentences is ten, there can be only 1024 maximum number of sentences possible. However, in many of our theoretical studies we consider the language as infinite, hence size of some sentences may also be infinitely large. For our study of languages, we usually consider the languages without any bound, i.e., infinite, but for specific languages, which are of practical use, we limit our study to finite set. This is because, the machines we would like to use for processing the

languages, are finite machines, they have finite memory, and have finite processing power.

When we are interested in an infinite language set, say $L$, we can use the finite devices called *generating grammars* to investigate the structure of $L$. In that scenario, the theory of language will contain specification of a class of functions from which grammars for a particular language may be drawn. Natural language processing (NLP) is a collection of computational techniques for automatic analysis and representation of human languages, motivated by theory. However, the automatic analysis of text, at par with humans, requires a far deeper understanding of natural language by machines, which is still far from reality. There are many examples of NLP, like, online information retrieval, aggregation, and question-answering, have been mainly based on algorithms relying on the textual representation of web pages, as well NLP to some extent. Such algorithms are very good at retrieving texts (IR), splitting it into parts, checking the spellings, and and word-level analysis, but not successful for analysis at sentence and paragraph level. Hence, when it comes to the question of interpreting sentences and extracting meaningful information, however, the capabilities of these algorithms are still very limited.

NLP, in general, requires high-level symbolic capabilities, which includes the following:

Access and acquisition of lexical, semantic, and episodic characteristics,
Creation and propagation of dynamic bindings,
Manipulation of constituent recursive structures,
Coordination of many processing and learning modules,
Identification of basic language constructs (e.g., objects and actions) and,
Representation of abstract concepts.

All the above capabilities are needed to shift from mere NLP to what is usually called as natural language understanding. The present approaches to NLP are based on the syntactic representation (also called syntactic structures) of text, i.e., relying on the word co-occurrence frequencies. Such algorithms have the limitation that they can process only based on the information they can see in the text being processed, but cannot consider the background information what we humans do. For example, when we say that "Sachin Tendulkar is a good batsman," we understand this sentence due to abundant information we have in our brain about the game of cricket, and bout the successes of "Sachin Tendulkar" in many cricket matches. However, the present algorithms do not have all this background with them, hence their understanding is limited about any given text.

As human text processors, being humans we do not have such limitations, as every word coming across in the text activates a cascade of semantically related concepts, relevant episodes, and sensory experiences, all these enable the completion of complex NLP tasks. These tasks are, for example, word sense disambiguation, textual entailment, and semantic role labeling, all in a quick and effortless way.

Many new computational models are making attempts to bridge the cognitive gap by emulating the processes recognized as being part of the human brain, and used for language processing by humans. These approaches depend on semantic features

that cannot be explicitly expressed through the text. The computational models are useful for theoretical purposes, e.g., scientific studies, such as exploring the nature of linguistic communication and its properties, as well for practical and industrial applications, such as enabling effective human–machine communications.

Challenges of NLP

Developing a program that understands natural language is a difficult problem. Majority of natural languages are large, they contain infinitely many sentences. Also there is much ambiguity in natural language. Many words have several meanings, such as *can*, *bear*, *fly*, *orange*, and the same sentences many times have different meanings in different contexts. Due to this, creation of programs that understands a natural language is a challenging task.

The syntax of a language helps us to decide how the words are being combined to make larger meanings. For example, in the sentence "the dealer sold the merchant a dog," it is important to be clear about what is sold to whom. Some of the common examples are the following:

I saw the Golden gate bridge flying into San Francisco.
(Is the bridge flying?)
I ate dinner with a friend.
I ate dinner with a fork.
Can companies litter the environment
(Is this a statement or question?)

Finally, assuming that we have overcome the problem at the previous levels, we must create an internal representation, and then, some how use the information in an appropriate way. This is the level of *semantics* and *pragmatics*. Here too the ambiguity is prevalent. Consider the following sentences.

*Jack went to store. He found the milk in aisle three. He paid for it and left*.

Here the problem is deciding whether "it" in the sentence refers to aisle or three, the milk, or even the store.

The most important part in the above is what is internal representation, so that these ambiguities in understanding the sentence do not occur and machine understands the way a human being understands the sentences.

**Learning Outcomes of this Chapter**:

1. Define and contrast deterministic and stochastic grammars, providing examples to show the adequacy of each. [Assessment]
2. Simulate, apply, or implement classic and stochastic algorithms for parsing natural language. [Usage]
3. Identify the challenges of representing meaning. [Familiarity]
4. List the advantages of using standard corpora. Identify examples of current corpora for a variety of NLP tasks. [Familiarity]

5. Identify techniques for information retrieval, language translation, and text classification. [Familiarity]
6. Tools for NLP. [Usage]

## 19.2   Progress in NLP

Right from its start, NLP focused on these tasks: natural language translation, information retrieval, information extraction, text summarization, question answering, topic modeling, and the recent one on opinion mining.

Syntax Versus Semantics

The NLP advances that took place after the 1960s paid attention on syntax, and partly on the semantics, but it was more of syntax-driven processing [1]. However, the semantic problems of Natural Language and its processing were obvious requirements from the beginning itself, but strategy adopted was to tackle the syntax first, for the more direct applications. Many works on NLP recognized the need for external knowledge for interpretation and reasoning on input language (Minsky, 1968), who explicitly emphasized semantics, e.g., the general-purpose semantics with case structures for representation and semantically driven processing [17].

First-Order Predicate

One of the most popular Natural Language representation technique used is FOPL (First-Order Predicate Logic), which is a deduction-based logic consisting of *axioms* and *inference* rules. The FOPL can be used to construct relationally rich predicates formulas, using quantification. The inferences can be drawn using formulas, called knowledge base, with the application of *resolution principle*. The FOPL supports the expressions of syntactic, semantic and, to a certain extent pragmatic. The syntax expresses a way of grouping of symbols, so that they are considered properly formed. The semantics specifies the meaning of these well-formed expressions. The pragmatics is more difficult to implement, which specifies to make use of context-related information to provide better correlations between different semantics. The later is important in tasks like word sense disambiguation.

Default Logic

The Default Logic was proposed [15] to formalize the default assumptions, e.g., "all birds fly." However, problems arose when default logic formalized the facts that were true in the majority of cases but false in the exceptional cases, i.e., "exceptions to these general rules," e.g.' "penguins do not fly" [15].

Production Rules

The *production rules* based model is another popular model for representation of Natural Languages (Chomsky, 1956). This model keeps ongoing assertions in a volatile working memory. The production rules comprise as their premises the set of conditions and consequent as a set of actions (i.e., IF <conditions> THEN <actions>).

The basic operations in a production-based system comprise, in order, three steps: 1. Recognize, 2. Resolve the conflict, and 3. Act. These three operations repeat in order as cycle until there are no more rules left in the working memory on which these rules can be applied. The first step (recognize) identified the set of rules (called conflict set) whose premises conditions are satisfied by the current working memory. The second step (resolve conflict) looks into the conflict set and selects a set of suitable rules to execute. The suitability is with the objective of efficiency, such that the task can be completed in shortest time. The "act" step is the third step, which simply executes the actions and updates the working memory. The production rules are created in modular forms for ease of writing as well for optimum use of memory and processing requirements.

OWL

A new model for NLP is Ontology Web Language (OWL), which is an XML-based vocabulary that extends the Resource Description Framework (RDF) language and provides a more comprehensive set for representation of ontology. Some examples of ontology representations are: definition of classes and relations between them, properties of the classes, and constraints on relations and on properties of the classes. The subject–predicate–object model is supported by RDF, which makes assertions about the resource. The RDF-based engines are commonly used for checking semantic inconsistency, which helps to improve on the ontology classification.

Networks

The networks as tools have been commonly used for NLP, where inferencing is used to be the primary goal. For example, the Bayesian belief networks (Pearl, 1985) is the tool for expressing joint probability distribution over many interrelated hypotheses. The variable in such networks are represented using directed acyclic graph (DAG), the edges are causal connections between any two variables, such that the truth of the causing variable directly influences the caused variable. A Bayesian network represents the subjective degree of confidence. The representation explores the role of *prior knowledge* and combines the pieces of evidence of the likelihood of the events.

For computation of joint probability distribution of the belief network, we need to follow the following steps: for each variable $q$ represented through belief network, there is a need to know the probability $P(q|p_1, ..., p_n)$, where $p_1, ..., p_n$ are the parents of $q$. In a large network it requires such computations at every node, making it difficult to determine the joint probability distribution of the entire network. It will require maintenance of probability table at each node in the belief network, which is again a challenging task for large-scale information processing problems.

Finally, the Bayesian networks also have limited expressiveness in NLP, as it is no better than that of proposition logic. Due to these factors, semantic networks are more often used in NLP [13].

Semantics Networks

A semantic network is a graphical notation for representing using interconnected nodes and arcs (Sowa, 1987), which is quite different from belief networks. The

network's focus is on relationships between a concept and a newly defined sub-type, represented using *Isa* keyword. Such a network structure provides generalization, which supports the rule of *inheritance*. The latter has the property of the copying the properties defined for super-type to all the sub-types, there is no need to define the properties again for the sub-types. The information represented through the semantic networks is assumed to be true.

Apart from the inheritance-based networks, another kind of semantic networks is the *assertional network*. This network is meant to assert propositions, and the information it contains is assumed to be contingently true. However, the truth of contingent is not implemented through default logic, but it is based on application of human's commonsense. The proposition also has sufficient reason, such that the reason entails the proposition, e.g., "the rock is hot" with the sufficient reasons being "the sun is shining on the rock" and "whatever the sun shines on is warm" [19].

The concept of semantic network came in the early 1960s, which was further developed by Marvin Minsky, till the 1980s. The base of semantic network is human mind, where it is hypothesized that the human intelligence results from a vast majority of things, and not from any single perfect principle. As per Minsky, the human mind (or even animals') is made of many little things, which he termed as "agents", but each one of them is mindless, when considered as an independent element. But, these agents lead to true intelligence when they work collectively, and responsible for performing functions, like, remembering, generalizing, comparing, analogizing, exemplifying, predicting, simplifying, etc.

The theory of human cognition by Minsky gave birth to many research projects' attempts, that built commonsense knowledge bases for NLP tasks. The major representative projects related to cognition are as follows.

*Cyc*—a Large Knowledge Based System [6],
*WordNet*—a Large Knowledge Based System [8],
*Thought-Treasure*—Natural Language Processing with Thought-Treasure [11], and
*Open Mind Common Sense project*—a second-generation commonsense database [18].

## 19.3  Applications of NLP

The importance of NLP is due to the fact that there is a huge amount of data in WWW, at least 20 billion pages, and that can be used as a big resource, provided that important information can be found from these through NLP. Some of the well-known uses of this data are possible through the following applications [4]:

Indexing and searching large texts,
Information retrieval (IR),
Classification of text into categories,
Information extraction (IE),
Automatic language translation,

Automatic summarization of texts,
Question-answering (QA),
Knowledge acquisition,
Text generations/dialogues.

Some of the above domains of NLP will be discussed in this chapter, and we will get the exposure sufficient enough to solve many problems of NLP however, the field of NLP is so vast that even a book is not sufficient to present in detail all the available techniques.

## 19.4   Components of Natural Language Processing

The NLP is the subject of computational linguistics—the study of computer systems for understanding and generating natural language. The linguistics has two branches—computational linguistics and theoretical linguistics. The computational linguistics has been concerned with developing algorithms for handling a useful range of natural language as input. While the theoretical linguistics has focused primarily on one aspect of language performance, grammatical competence—how people accept some sentences as correctly following grammatical rules and others as ungrammatical. They are concerned with language universals—principals of grammar which apply to all natural languages [16].

Computational linguistics is concerned with the study of natural language analysis and language generation. Further, the language analysis is divided into two domains, namely sentence analysis, and discourse and dialogue structure. Much more is known about the processing of individual sentences than about the determination of discourse structure. Any analysis of discourse structure requires a prerequisite as an analysis of the meaning of individual sentences. However, it is a fact that for many applications, thorough analysis of discourse is not mandatory, and the sentences can be understood without that [14].

The sentence analysis is further divided into *syntax analysis* and *semantic analysis*. The overall objective of sentence analysis is to determine what a sentence "means". In practice, this involves translating the natural language input into a language with simple semantics, for example, *formal logic*, or into a *database command language*. In most systems, the first stage is syntax analysis. Figure 19.1 shows the relations among different components of NLP.

### *19.4.1   Syntax Analysis*

The syntax analysis of a language performs two main functions in analyzing natural language text, which are as follows:

- *Determining Structure.* Given the sentences, the syntax analysis should identify the *subject* and *object* of each verb, and determine what each modifying word is,
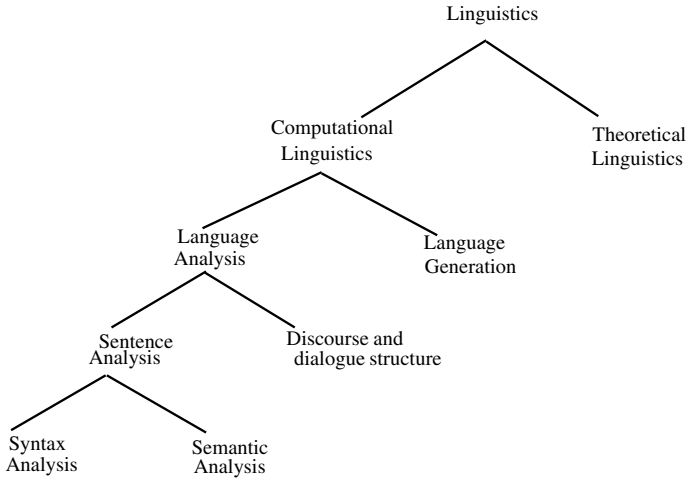
**Fig. 19.1** Components of NLP

and what phrase it modifies. This is achieved by assigning a tree structure to the input, during the process called as *parsing*.

- *Regularizing the syntax structure.* Subsequence processing, i.e., semantic analysis gets simplified if a large number of possible input sentences are mapped into a small number of structures. For example, some material in sentences (enclosed in brackets in the example below) can be omitted, but still maintaining the meaning of these sentences unaltered.

> "He talks faster than John [talks]."
> "John ate cake and Mary [ate] Cookies."

In addition, if the structures are appropriately chosen, operator–operand (e.g., verb–subject, verb–object) relations should be clearly evident in the output of the syntactic stage. This kind of regularization is achieved, when passive voice sentence is converted into active voice, for example.

> "Those grapes were crushed by me,"
> converted to
> "I crushed those grapes."

Some NLP type have chosen to omit syntactic regularization all together, and have semantic component operate directly on the full variety of sentence structures. In such systems, the syntactic regularization is subsumed within the semantic analysis process, which will, however, require more complex semantic rules.

The syntax analysis is performed through grammars (phrase structure grammar) using the process of parsing, and through transformational grammars, discussed in the following sections.

### 19.4.2   Semantic Analysis

The objective of semantic analysis is to determine what a sentence means. The semantics seeks the conditions under which the sentence is true. Almost equivalently, we can say, what are the inference rules for the sentences of the language. Characterizing the semantics of interrogative and imperative sentences are more problematic.

Why the meaning of a sentence is important as long as a sentence is correct or syntactically correct? Here is one reason. Consider that we build a natural language system, our aim is not only to ensure that sentence is correct, but usually we want the system to do some thing in response to input, like, retrieve data, move a robot arm, etc. In general, this will mean translating the natural language input into the formal language of a database retrieval system, a robot command system, etc. In comparison with natural language, these formal languages will have the following properties:

- unambiguous,
- simple rules of interpretation and inference, and
- logical structure determined by the form of the sentence.

Thus, there is a significant similarity between the tasks that translate into such a *practical language*, and those translate into a *logical formalism*.

Formal Languages for Meaning Representation

One method for meaning representation is *propositional Logic*. Using this logic, given that "If it is raining then ground is wet, and It is raining," we can infer that "ground is wet." We observe that as long as we preserve the general structure of the argument, including the "if ... then ..." connective in the first part, we may change the sentences making up the argument and still have a valid argument. For example, "If it is night then Ram is sleeping, and It is night," we may conclude that "Ram is sleeping." And, the overall argument is a valid argument.

The above rule is represented in predicate logic as

Given that:    $P \rightarrow Q$, and
                $P$,
we may conclude: $Q$.

The above rule is called *Modus Ponens*. Similar to propositional logic, the rule of Modus Ponens also exists in *Predicate Logic* also. For details of these logics, refer to chapters two and three in this book.

The other approaches for semantics are *Semantic networks*, and *Conceptual dependency*, which are presented in details in the chapter seven of this book.

### 19.4.3   Discourse Analysis

So far we have discussed the structure and meaning of individual sentences, but what is solution is the meaning of the entire text. Because, the information conveyed by a

text is clearly more than sum of its parts—more than the meaning of its individual sentences. For example, if a text tells a story, describes a procedure, or presents an argument, we must understand the connections between the component sentences in order to have fully understood the story. *Discourse analysis* is the study of these connections between the sentences. Since these connections are implicit in the text, identifying them may be a difficult task.

As a simple example of the challenge of discourse analysis, just consider the following brief description of a naval encounter:

> Just before the dawn, the *Vikrant* sighted the an unidentifiable blue-ship and fired two torpedoes. It sank swiftly, leaving one survivor.

The problem we face is, what stands for "it"? There are four candidates in the first sentence: dawn, Vikrant, blue-ship, and torpedo. The semantic analysis helps to exclude the "dawn" as a meaning for "it", and number agreement excludes "torpedoes". But, still leaves two candidates Vikrant, and blue-Ship, which are both ships. Our syntax and semantic analysis tools will not enable us to resolve between these two, as which stands for "it". To get the noun for "it", we must understand the causal relationship between the firing of torpedoes and sinking of ship. Since Vikrant fired torpedoes, it must have fired on the blue-ship (because Vikrant is not supposed to fire on itself !). Hence, "it" stands for blue-ship.

Since much of the information conveyed by this text is implicit, we cannot claim that the text is adequately understood unless the implicit information is recovered. The role of discourse analysis is to recover this information.

As we have discussed that discourse is a multi-sentence text, discourse comes in many forms, describe a scene, give instructions, or present an argument. The tools for discourse analysis are *Frames*, *Scripts*, and *Conceptual dependencies*, which were covered in detail in chapter seven in this book.

## 19.5   Grammars

The grammar of a language can be viewed as a theory of the structure of that language. Like any mathematical theory, the theory of grammar is based on a a set of finite number of observed sentences, but it projects this to an infinite set of grammatical sentences. This becomes possible by creating general laws (grammatical rules), framed in terms of such hypothetical constructs as the particular phonemes, words, and phrases, of the spoken language under analysis. A properly formulated grammar should be able to determine unambiguously the set of all grammatical sentences in that language [2].

Let us assume that for most languages there are certain clear examples of grammatical and ungrammatical sentences. Consider the following sentences of in English:

1. John ate a sandwich,
2. Sandwich ate a John.

Suppose a large corpus-x does not contain either of these sentences. In this scenario can we say, grammar that is determined for corpus-x will direct the corpus to include the first sentence and exclude second. Even though such simple cases maybe not adequate to confirm that all correct sentences of that language will be tested as correct and all wrongs will be rejected, but still it becomes a strong test.

The first step toward the linguistic analysis of a language is to provide a *finite system* of representation for its sentences. By the grammar of a language $L$ we mean a device that produces all of the strings that are sentences of $L$ and no other strings.

### 19.5.1 Phrase Structure

The description of phrase structure comprises the grouping of words into phrases, which are then grouped into smaller constituent phrases and so on, until the ultimate constituents (generally morphemes) are reached. The phrases are typically classified as *noun phrases* (NP), *verb phrases* (VP), etc. For example, the sentence "the man took the book," might be analyzed as shown in Fig. 19.2.

Evidently, description of sentences in such manner permit considerable simplification compared tor the word-by-word model. This is because the composition of a complex class of expressions such as NP, can be stated just once in the grammar, and this class can be used as a building block at various stages in the construction of sentences. We now question, what form of grammar corresponds to this conception of linguistic structure? We are going to get the answer in the following.

### 19.5.2 Phrase Structure Grammars

A phrase structure grammar is defined using a finite vocabulary (alphabet) $\Sigma$, a finite set $V$ of variables, a finite set $P$ (rewrite rules) of productions of the form $X \rightarrow Y$, where $X$ is a string in $V$, and $Y$ in $V \cup \Sigma$. Hence, a grammar is $G = (V, \Sigma, S, P)$, where $\Sigma$ is set of terminal symbols, which appear at the end of generation, $S$ is start symbol (for sentence). The corresponding language of $G$ is $L(G)$. Given this grammar, we say that a string $\beta \in (V \cup \Sigma)^*$ follows from $\alpha \in (V \cup \Sigma)^*$, then we write it as $\alpha \rightarrow \beta$. Also, if we are able to generate a sentence $w$ from start symbol

**Fig. 19.2** A sentence's structure

| the man | took | the book |
|---|---|---|
| | Verb | NP |
| NP | | |
| | VP | |
| Sentence | | |

$S$, then we can write it as

$$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n = w. \qquad (19.1)$$

A derivation of a sentence is thus roughly analogous to a proof with $V$ taken as axiom system and $P$ as the rules of inferences. We say that $L$ is a *derivable language* if $L$ is the set of strings that are derivable from grammar $G$, and say that $L$ is a terminal language if it is the set of terminal strings from some system $(V, \Sigma, S, P)$.

As a simple example of a system of this form, consider the following small part of English grammar:

$$
\begin{aligned}
S &\to NP \ \ VP \\
VP &\to Verb \ \ NP \\
NP &\to the \ \ man \mid the \ \ book \\
Verb &\to took
\end{aligned}
\qquad (19.2)
$$

Among the derivations from (19.2) we have, in particular:

$$
\begin{aligned}
D_1 : S &\to NP \ \ VP \\
&\to NP \ \ Verb \ \ NP \\
&\to the \ \ man \ \ Verb \ \ NP \\
&\to the \ \ man \ \ took \ \ NP \\
&\to the \ \ man \ \ took \ \ the \ \ book
\end{aligned}
$$

Also, we have:

$$
\begin{aligned}
D_2 : S &\to NP \ \ VP \\
&\to NP \ \ Verb \ \ NP \\
&\to the \ \ man \ \ Verb \ \ NP \\
&\to the \ \ man \ \ Verb \ \ the \ \ book \\
&\to the \ \ man \ \ took \ \ the \ \ book.
\end{aligned}
$$

These derivations $D_1$ and $D_2$ are evidently equivalent; they differ only in the order in which the rules are applied. We can represent this equivalence graphically by constructing diagrams that correspond, in an obvious way, to derivations. Both $D_1$ and $D_2$ reduce to the diagram as shown in Fig. 19.3. The diagram gives the phrase structure of terminal sentence "the man took the book." Note that "the man", "took", "the book", and "took the book" are phrases as these are traceable back to some nodes. However, "man took" is not a phrase as it is not traceable back to any node.

**Example 19.1**  Consider that various tuples of a grammar are given as follows:

**Fig. 19.3** Phrase structure of "the man took the book"
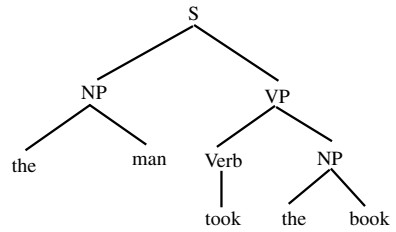


$$V = \{S, NP, N, VP, V, Art\}$$
$$\Sigma = \{boy, icecream, dog, bites, likes, ate, the, a\}$$
$$P = \{S \rightarrow NP \ VP,$$
$$NP \rightarrow N,$$
$$NP \rightarrow Art \ N,$$
$$VP \rightarrow V \ NP,$$
$$N \rightarrow boy \mid icecream \mid dog,$$
$$V \rightarrow ate \mid likes \mid bites,$$
$$Art \rightarrow the \mid a\}$$

Using the above grammar, we can generate the following sentences:

*The dog bites boy.*
*Boy bites the dog.*
*Boy ate Icecream.*
*The dog bites the boy.*

Note that, to generate any sentence, rules from the set $P$ are applied sequentially starting from the first rule. However, a grammar does not guarantee the generation of meaningful sentences, but generate only those that are structurally correct as per the rules of the grammar. □

Structural Representation

It is convenient to represent the sentences as tree or a graph to help expose the structure of the constituent parts, called parts-of-speech (POS). For example, the sentence, "The boy ate an Icecream" can be represented as a tree shown in Fig. 19.4.

For the purpose of computation, a tree must also be represented as a record, or a list, or some similar data structure. For example, the tree above in Fig. 19.4 can be represented as a list:

$$\text{(S (NP ((Art the)}$$
$$\text{(N boy))}$$
$$\text{(VP (V ate) (NP (Art an) (N Icecream)))))}$$

**Fig. 19.4** A Syntax Tree of
"The boy ate an Icecream"
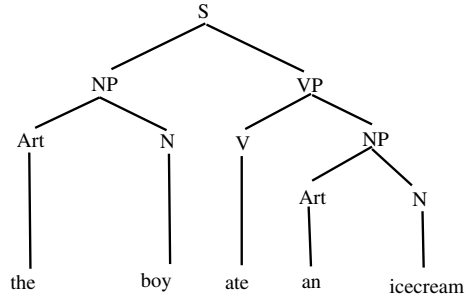


A more extensive English grammar can be obtained with the addition of other
constituencies such as prepositional phrases (PP), prepositions (Prep), adjectives
(ADJ), determiners (DET), adverbs (ADV), auxiliary verbs (AUX), and many other
features. Correspondingly, the additional rewrite rules are as follows:

$$PP \rightarrow Prep\ NP$$
$$VP \rightarrow V\ ADV\ |\ V\ PP\ |\ V\ NP\ PP\ |\ AUX\ V\ NP$$
$$Det \rightarrow Art\ ADJ\ |\ Art$$

These extensions allow the increase in complexity of the sentences generated
through this expanded grammar, along with its expression power, as in the following
sentences.

The(Art) cruel(Adj) man(N) locked(V) the(Art) dog(N) in(prep) the(Art) cage(N).
The(Art) laborious(Adj) man(N) worked(V) to(Aux) make(V) extra(Adj) money(N).

## 19.6   Classification of Grammars

The grammars are classified in many ways: when they are based on their generating
rules they are called *Chomskian grammars*; when a grammar transforms the passive-
voice sentences to active voice, it is *transformational grammar*; and when a sentence
generated by a grammar has more than one meaning(sense), it is called *ambiguous
grammar*. The following describes the theory of each in brief.

### 19.6.1   Chomsky Hierarchy of Grammars

In fact, it is not always possible to formally characterize the natural languages with
a simple grammar as discussed above. The grammars are defined by *Chomsky Hier-
archy*, as type 0, 1, 2, 3, with simplest as type-3 and most complex as type-0. The

type-3, called *Regular Grammars* have rewrite rules as:

$$S \rightarrow aS$$
$$S \rightarrow a.$$

We note that the left-hand side is a variable symbol, and right hand is only terminal symbol, or a terminal followed with variable symbol.

The type-2 grammars, called *context-free grammars* (CFG) have left-hand side symbol of variable, and right-hand side has any combination of variables and terminal symbols. Following all the productions belong to type-2 grammars, where $S$, $A$, $B$ are variable symbols; $a$, $b$ are terminals, and $\alpha$ is any combination of variables and terminals.

$$S \rightarrow aS \mid aSb \mid aB \mid aAB \mid \alpha$$
$$A \rightarrow a$$
$$B \rightarrow b$$

The typical rewrite rules for type-0, called *unrestricted grammar* and type-1 called context-sensitive grammar (CSG), are are follows:

$$S \rightarrow aS \mid aAB$$
$$AB \rightarrow BA$$
$$aA \rightarrow ab \mid aa$$
$$\alpha \rightarrow \beta$$

In the above rules, $\alpha \in (\Sigma \cup V)^* \cup V$, and $\beta \in (\Sigma \cup V)^*$. In all these rules, left-hand sides are both variables and terminals and same applies for right-hand symbols. The only difference between type-0 and type-1 grammars is that, if $\alpha \rightarrow \beta$ is production, then in type-1, $|\alpha| \leq |\beta|$.

Since, the grammars are called unrestricted, context-sensitive, context-free, and regular grammars, the corresponding languages are also called unrestricted, context-sensitive, context-free, and regular languages, respectively. The natural languages are mostly based on the type-2 languages, as the type-0 and type-1 are not much understood yet, and are difficult to implement.

### 19.6.2  Transformational Grammars

Transformational Grammar involves the use of defined operations called transformations, to produce new sentences from existing ones. The grammars discussed above

produce different structures for different sentences, even though they have the same meaning. For example,

> *Ram gave Shyam a book.*
> *A book was given by Ram to Shyam.*

In the above, the subject and object roles in second sentence are switched. In the first, subject is "Ram" and object is "Book", while in second sentence they are other way round. This is an undesirable feature for machine processing of a language. In fact, sentences having the same meaning should map to the same internal structures.

However, by adding some extra components, we can produce a single representation for sentences having the same meaning, through a series of transformations. This extended grammar is called *Transformational grammar*. In addition, *semantic* and *phonological* components are components, which are added as new, helps in interpreting the output of the syntactic components, as meaning and sound sequences. These transformations are tree manipulation rules, which are taken from dictionary, where words contain semantic featuring each of the lexica.

Using transformational generative grammar, a sentence is analyzed in two stages: (1) basic structure of the sentence is analyzed to determine the grammatical constitutional parts, which provides the structure of the sentence. (2) This structure is transformed into another form, where a deeper semantic structure is determined.

The application of transformations is to produce a change from passive voice form of the sentence into active voice, which changes a question to declarative form, handle negations, and provide subject–verb agreement. Figure 19.5a–c shows the three stages of conversion, from passive voice to active voice of a sentence.

However, the transformational grammars are now rarely used as computational models for language processing [14].



**Fig. 19.5** Transformational Grammar
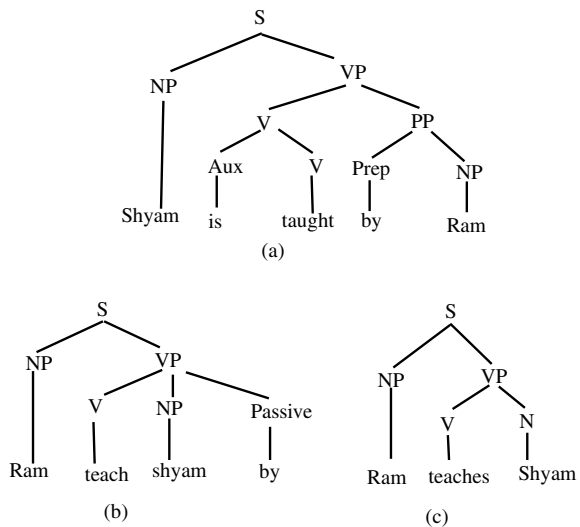
### 19.6.3  Ambiguous Grammars

An ambiguous grammar is a context-free grammar for which there exists a sentence that can have more than one parse tree, while an unambiguous grammar is a context-free grammar for which every sentence has a unique parse-tree. For computer programming languages, the reference grammar is often ambiguous, due to issues such as the dangling "Else" problem. If present, some of the ambiguities can be resolved by adding precedence rules or other context-sensitive parsing rules, so the overall phrase grammar is unambiguous. Note that, to find out if general grammar is ambiguous, is a NP-hard problem [4].

The ambiguous grammars have more than one parse-tree for the same sentence. For example, consider the sentence "He drove down the hill in the Car." A process for constructing a parse-tree is grouping the words to realize the structure in the sentence. The parse-tree given in Fig. 19.6a for this sentence shows the grouping of words and Fig. 19.6b shows the parse-tree for this sentence. Similarly, Fig. 19.7a
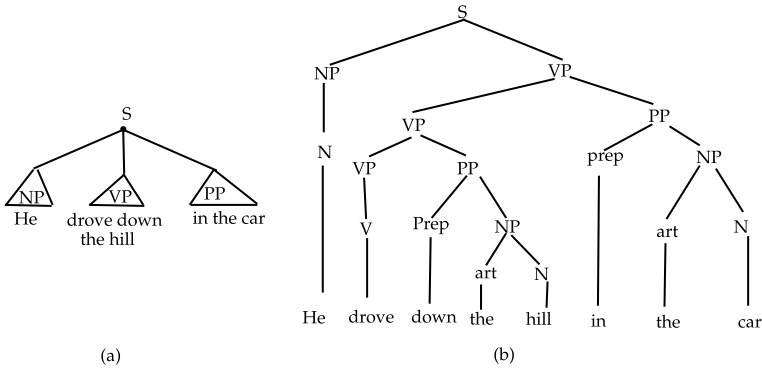


**Fig. 19.6** **a** Grouping the words, **b** Parse-tree I: "He drove down the hill in the car"
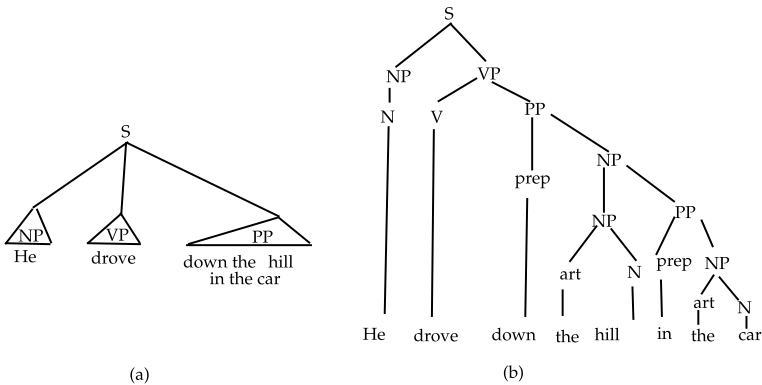


**Fig. 19.7** **a** Grouping the words, **b** Parse-tree II: "He drove down the hill in the car"

shows another grouping for the same sentence and the corresponding parse-tree is shown in Fig. 19.7b. Since this sentence can be generated through two different parse-trees, the sentence is ambiguous, and hence its grammar is also ambiguous.

## 19.7   Prepositions in Applications

Prepositions, as well as prepositional phrases (PPs) are important to the precise understanding of any language, which may be useful for certain applications, e.g., in Information Extraction. However, they have been ignored on the grounds of being syntactically promiscuous, and semantically having no significant meaning, hence they were considered in the rank of "stop words" [20].

As a part of its argument structure, property of a preposition being subcategorized or specified by the governor is called *selection* property. The governor is usually a verb. An example of selection preposition is, "with", like, in the phrase, *dispense* with *introductions*. Here *dispense* is verb and *introductions* is the object of this verb. But this object is realized in a prepositional phrase, which is headed by *with*. Conventionally, the prepositions of *selection* type are uniquely specified. The following are other examples with well-defined structures.
*chuckle over/at*

It is clear from the above examples the use of prepositions have no definite semantics.

The selection proposition is useful in NLP applications; it operates at the *syntax-semantics* interface, i.e., those application which openly translates surface strings onto semantic representations, or vice versa. Examples of such translation are: Information Extraction, and Machine Translation using some form of interlingua.

PP Attachment

To find a PP attachment is nothing but to find the governor for a given PP. Consider the sentence,
*Sujata eats[1] kheer with spoon.*

This sentence[1] consists a syntactic ambiguity, as in one case the PP "with spoon" is governed by noun *kheer* (i.e., as part of the NP *kheer with spoon* (see Fig. 19.8a). As an alternative sense the PP "with spoon", is governed by the verb *eats* (i.e., as a modifier of the verb, as indicated in Fig. 19.8b. Of these, the later case of verb attachment (i.e., Fig. 19.8b) is, of course, the correct analysis.

The much interest in PP *attachment* comes from the PP being a common phenomenon when parsing the languages such as English, which is a major cause of parser errors. The ambiguity due to PP is called *attachment ambiguity*.

The syntactic preferences do not provide the solutions in dealing with PP attachment, and they are not very effective for predicting the difference in PP attachment

---

[1]Kheer: A sweet dish, like porridge, popular among the Indians.
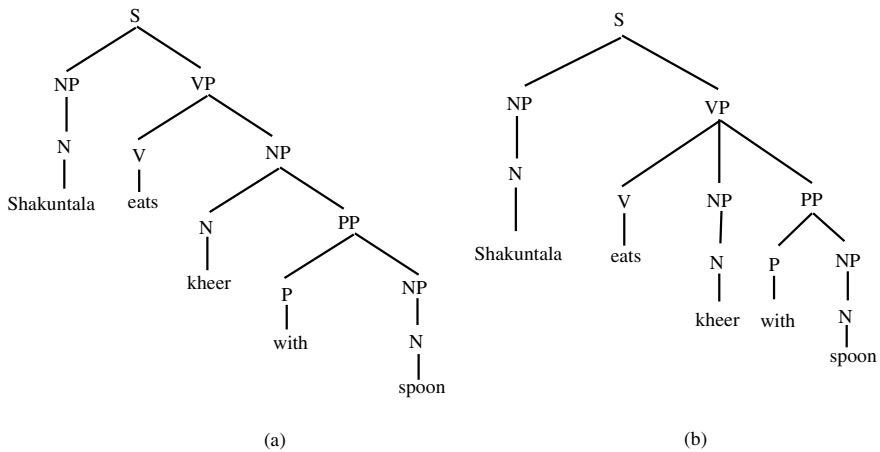
**Fig. 19.8** The prepositional phrase "with spoon," being governed by: **a** noun *kheer*, **b** verb *eats*

between noun attachment and verb attachment. This difficulty led to a shift from syntactic approaches in the 1980s toward AI-based approaches, the later used world knowledge for resolving PP attachment ambiguity. Figure 19.8a shows that, due to the availability of the knowledge that *spoon* is an eating implement, it would suggest a preference for verb attachment. In a similar way, the knowledge that they are not a foodstuff, it would suggest a preference against noun attachment.

An algorithm that resolves the attachment ambiguity, defines constraints that are derived from *semantic* and world knowledge. These constraints are used for composing the meaning of the child unit, which is attached to the meaning of each of the possible parent's syntactic units. This would be helpful for attaching the child unit to the parent units (see Fig. 19.8). These constraints are called selectional constraints, and usually represented in the form of the permissible range of *fillers* in the slots in *frames* that represent the meanings. The potential filler, which is meaning of a child unit, is compared against the constraint by a *fuzzy match* function. A popular way to do this fuzzy match is to compute a weighted distance between the two meanings in a *semantic* or *ontological network* of concepts. The closer the two are in the network, higher is the score assigned to the particular choice.

## 19.8 Natural Language Parsing

Parsing is carried out to compute the structural description of a sentence. This structural description is assigned by the grammar of the language, and as a precondition it is assumed that sentence is well-formed, i.e., grammatically correct. The process of parsing consists of at least the following activities:

- Mathematical characterization of derivations in a grammar, and those associated with parsing algorithms.

- Computing the time and space complexities of parsing algorithms, with length of input as the length of the input sentence.
- Comparing different grammar formalism and showing equivalences among them whenever possible.
- Combining grammatical and statistical information for improving the efficiency of parser, and ranking the parsers.

The structural descriptions provided by a grammar depend on the formalism to which the grammar belongs. For the well-known context-free grammar (CFG), the structural description is phrase structure grammar.

### 19.8.1   Parsing with CFGs

A parse-tree describe the structure of a sentence, and is also the record of the sequence of steps of derivation of the sentence. The parse-trees are useful due to following reasons:

- In carrying out semantic analysis, the parsing is an important intermediate stage,
- Grammatically checking the sentence, and
- The parsing is useful in following:

  - Mechanical translation,
  - Question answering, and
  - Information Extraction.

A *syntactic* parser can be thought of as searching through the space of all possible parse-trees to find the correct parse-tree for the given sentence. Before we go through the steps of parsing, let us consider the following rules for grammar.

$S \rightarrow NP\ VP$ ;a start symbol for sentence can be replaced by
      noun phrase followed by verb phrase

$S \rightarrow Aux\ NP\ VP$ ;Aux stands for auxiliary, e.g., do, does

$S \rightarrow VP$

$NP \rightarrow Det\ Nom$ ;Det is determiners, Nom is for Nomial

$Nom \rightarrow Noun\ Nom$

$Nom \rightarrow N$

$NP \rightarrow proper\text{-}N$ ;for proper noun

$VP \rightarrow V$

$VP \rightarrow V\ NP$

$VP \rightarrow VP\ PP$ ;PP is prepositional phrase

$PP \rightarrow Prep\ NP$ ;Prep is preposition, PP is prepositional phrase

**Fig. 19.9** Parsing tree of
sentence: "Book that flight"
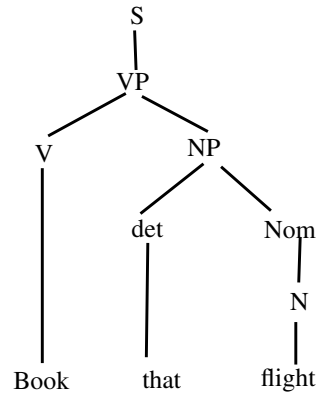


$$Det \rightarrow a \mid an \mid the$$
$$N \rightarrow book \mid flight \mid meal$$
$$V \rightarrow book \mid include \mid proper$$
$$Aux \rightarrow Does$$
$$prep \rightarrow from \mid to \mid on$$
$$Proper\text{-}N \rightarrow Mumbai$$
$$Nom \rightarrow Nom \ \ PP$$

The parse-tree for the sentence "Book that flight" is shown in Fig. 19.9.

**Example 19.2** Set of production rules ($P$) and parsed sentences for different forms
of verb-phrase.

$$S \rightarrow NP \ \ VP \ ; \text{I prefer a morning flight}$$
$$VP \rightarrow V \ \ NP \ ; \text{prefer a morning flight}$$
$$VP \rightarrow V \ \ NP \ \ PP \ ; \text{leaves Bombay in the morning}$$
$$VP \rightarrow V \ \ PP \ ; \text{leaving on Tuesday}$$
$$PP \rightarrow prep \ \ NP \ ; \text{from New Delhi.}$$

A NP can be location, date, time, or others. Following are other examples of parts of
speech (POS).

$$N \rightarrow flights \mid breeze \mid trip \mid morning$$
$$V \rightarrow is \mid prefer \mid like \mid need \mid want \mid fly$$
$$Det \rightarrow a \mid an \mid the \mid this \mid these \mid those$$

**Fig. 19.10** Parse-Tree of
sentence: "I prefer morning
flight"
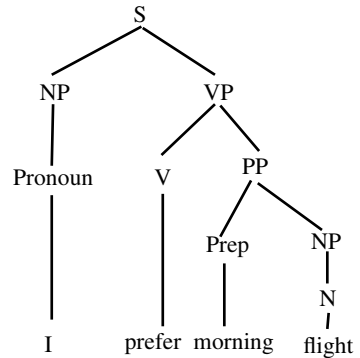


$$Pronoun \rightarrow me \mid I \mid you \mid it$$
$$Proper\text{-}N \rightarrow Mumbai \mid Delhi \mid India \mid USA$$
$$Adj \rightarrow cheapest \mid non\text{-}stop \mid first \mid latest \mid other$$
$$Prep \rightarrow from \mid to \mid on \mid near$$
$$Conj \rightarrow and \mid or \mid but$$

The following examples show the substitution rules and with values for each parts-
of-speech to be substituted.

$$NP \rightarrow Pronoun(I) \mid proper\text{-}N\ (Mumbai) \mid det\ Nomial$$
$$(a\ flight) \mid N\ (flight)$$
$$VP \rightarrow V\ (do) \mid V\ NP(\text{want a flight}) \mid V\ NP\ PP$$
$$(\text{leaves Delhi in Morning})$$
$$PP \rightarrow Prep\ NP(\text{from Delhi})$$

Making use of above rules, Fig. 19.10 demonstrates the parsing of sentence "I prefer
morning flight."

### 19.8.2  Sentence-Level Constructions

A sentences can be classified as *declarative*, *imperative*, and *pragmatic*, as follows.

- *Declarative Sentences.* These sentences have structure: $S \rightarrow NP\ VP$.
- *Imperative Sentences:* These sentences begin with "VP". For example, "Show
  the lowest fare," "List all the scores," etc. Following are the production rules for
  imperative sentences.

$$S \rightarrow VP$$
$$VP \rightarrow V \ \ NP$$

The other substitutions for verb are already discussion in above.

- *Pragmatic Sentences:* The examples of pragmatic sentences are the following:

*Can you give me the some information*?
*Do all these flights have stops*?
*What flights do you have from Delhi to Mumbai*?
*What Airlines fly from Delhi*?

The pragmatic sentences are governed by substitution rule

$$S \rightarrow Aux \ \ NP \ \ VP.$$

Corresponding to the word at begin of a sentence e.g., "What", the production rule is

$$Wh\text{-}NP \rightarrow What.$$

Hence, for the sentence,

"*What flights do you have from Delhi to Mumbai*?",

can be generated by the production rule,

$$S \rightarrow Wh\text{-}NP \ Aux \ NP \ VP.$$

Many times, the longer sentences are conjuncted together using connectives, e.g., *I will fly to Delhi and Mumbai*. The corresponding rule is

$$VP \rightarrow VP \ and \ VP.$$

### 19.8.3  Top-Down Parsing

In top-down parsing, the searching is carried out from the root node, substitutions are carried out at every step, and the progressing sentence is compared with the input text sentence to determine whether the sentence generated progressively matches with the original. Figure 19.11 demonstrates the steps of top-down parsing for the sentence "Book that flight".

To carry out the top-down parsing, we expand the tree at each level as shown in the figure. At each level, the trees whose leaves fail to match the input sentence,
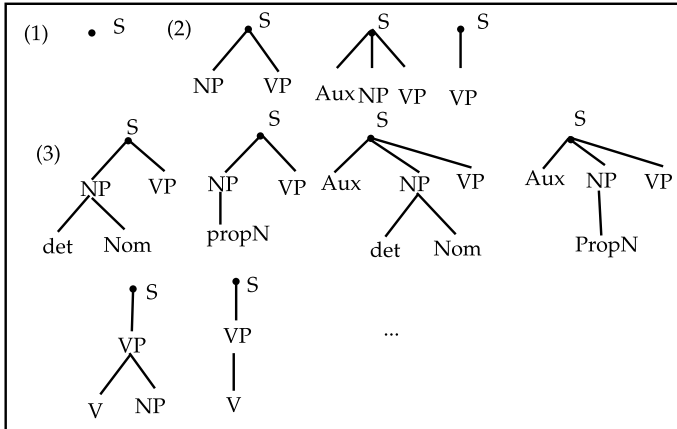
**Fig. 19.11**  Top-down parsing for the sentence, "Book that flight"

are rejected, leaving behind the trees that represent the successful parses. Going this way, we ultimately construct the original sentence: "Book that flight."

Figure 19.11 shows that, first, root node is created as $S$, then in step (2), it is expanded to generate three possible sub-trees,

$$S \rightarrow NP \ VP,$$
$$S \rightarrow Aux \ NP \ VP,$$
$$S \rightarrow VP.$$

First of these is expanded at level 2, in step (3), by rule "NP → det Nom", which, we know will not match. Neither, when NP is expanded by NP → PropN in the next sub-tree. So, we need to expand the next choice "S → Aux NP VP" of stage (2). This also does not work. Finally, what works is as follows:

$$S \rightarrow VP$$
$$VP \rightarrow V \ NP$$
$$V \rightarrow Book$$
$$NP \rightarrow Det \ N$$
$$Det \rightarrow that$$
$$N \rightarrow flight.$$

And, ultimately generates the sentence, "Book that flight."

The bottom-up parsing is other way round, starting from the sentence "Book that flight," and reducing it to start symbol $S$. Every time we perform a reduction, we reduce the right-hand side of the production $A \rightarrow \alpha$, i.e., $\alpha$ by a variable $A$.

### *19.8.4  Probabilistic Parsing*

In the discussion above, we drew a sharp line between a correct and an incorrect parses, which was based on whether a terminal node either matched or did not match the next word in the sentence. According to this parsing, a phrase is either acceptable or not. There are situations under which we can relax these requirements, e.g., when we cannot afford to try alternatives exhaustively. The examples are: analysis of connected speech, text segmentation, and identification of words can never be done with complete certainty. At best, one can say that certain sound is more probable than the other. Consequently, one may associate a fractional number with each terminal node, indicating the probability or quality of nodes below that, in respect of forming grammatically correct sentence [5].

In the probabilistic parsing, the non-terminal nodes will be assigned some values, which are sophisticated enough to realize that syntactic and semantic restrictions are taken care of. Hence, a parser that aims to deliver the best analysis even if every analysis violates some constraint, must associate a measure of being grammatical (i.e., acceptable) with the analysis of portions of the sentence. The sentence ultimately associates a measure with the analyses of the complete sentence. As a matter of rule, it is possible to generate an analysis of every sentence with a nonzero acceptability or matching probability, and then select the one having the best analysis.

One simple parser of this type is "best-first" parser, which is based on the modified form of standard *top-down serial* parsing algorithm for context-free grammars. The standard algorithm tries to generate one possible parse-tree until it gets stuck (i.e., on generation a terminal node which does not match the next word in the sentence). In case of stuck, it "backs up" to try another alternative. A *best-procedure*, is like a best-first search that tries all alternatives in parallel. A measure in the numerical value is associated with each alternative path that indicates the likelihood, that this analysis matches the sentence processed so far, and it can be extended to the complete sentence analysis. At each progressive step, the path with the highest likelihood is extended. In the process, if the "measure" of current path falls below that of some other path, the parser shifts its attention to that of other paths.

A CFG (context-free grammar) consists of

> terminal words $w^1, w^2, ..., w^V$,
> non-terminals $N^1, N^2, ..., N^n$,
> start symbol $N^1$, and
> production rules $N^i \rightarrow \alpha^j$,

where $\alpha^j$ is sequence of terminals and non-terminals. Given this, we can define a generative PCFG (probabilistic context-free grammar) as

> terminal words $w^1, w^2, ..., w^V$,
> non-terminals $N^1, N^2, ..., N^n$,
> start symbol $N^1$, and
> production rules $N^i \rightarrow \alpha^j$,

where $\alpha^j$ is sequence of terminals and non-terminals. And can define the rule probabilities as

$$\forall_i \sum_j P(N^i \to \alpha^j) = 1, \tag{19.3}$$

which shows that for each set of productions having same left-side variable ($N^i$), the sum of probabilities is unity.

We consider that sentence is represented as sequence of words $w_1 w_2 ... w_m$, and $w_{ab} = w_a ... w_b$ is a subsequence. Let non-terminal $N^i$ dominates subsequence $w_a ... w_b$ is represented by $N^i_{ab}$, .i.e., $N_i$ is root of sub-tree having children sequence as $w_a ... w_b$. Let $N^i \Rightarrow^* \alpha$. We represent the probability of sentence $w_{1n}$ as

$$P(w_{1n}) = \sum_t P(w_{1n,t}) \tag{19.4}$$

where $t$ is parse-tree of sentence $w_{1n}$.

**Example 19.3**  Construct a correct parse-tree for the sentence "I saw an astronomer with telescope," for the following PCFG:

| Rule | Probability | Rule | Probability |
|------|-------------|------|-------------|
| $S \to NP\ VP$ | 1.0 | $NP \to NP\ PP$ | 0.20 |
| $PP \to Prep\ NP$ | 1.0 | $NP \to N$ | 0.45 |
| $VP \to V\ NP$ | 0.7 | $NP \to Art\ N$ | 0.35 |
| $VP \to VP\ PP$ | 0.3 | $N \to telescope$ | 0.25 |
| $Prep \to with$ | 1.0 | $N \to astronomer$ | 0.15 |
| $Art \to an$ | 1.0 | $N \to I$ | 0.60 |
| $V \to saw$ | 1.0 | | |

In this problem, the symbols are as follows:

- Terminals: I, saw, an, astronomer, with, telescope.
- Non-terminals: *S, NP, VP, PP, V, Prep, Art, N*
- Start Symbol: *S*

The sentence above can be generated using two different parse-trees, say $T_1$ (see Fig. 19.12) and $t_2$ (see Fig. 19.13).

Probability for parse-tree $t_1$ is the product of all the probabilities of rules used, which starting from top, and going down through all the levels is given by

$$P(t_1) = 1.0 \times 0.45 \times 0.7 \times 0.60 \times 1.0 \times 0.20 \times 0.35 \times 1.0 \times 1.0 \times 0.45 \times 0.25$$
$$= 0.001488375$$

**Fig. 19.12** Probabilistic parse-tree $t_1$ for the sentence "I saw an astronomer with telescope"
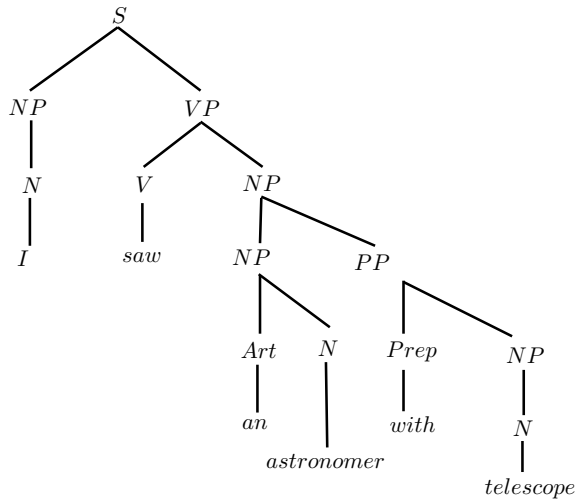


**Fig. 19.13** Probabilistic parse-tree $t_2$ for the sentence "I saw an astronomer with telescope"
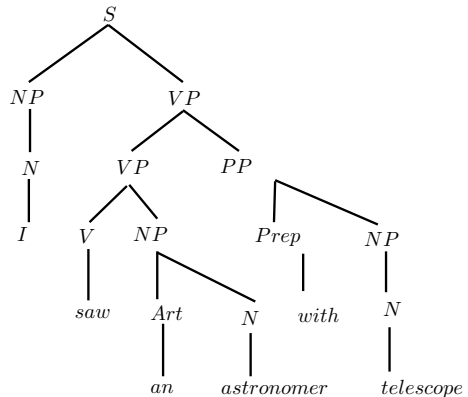


Similarly, the probability for parse-tree $t_2$ is given by

$$P(t_2) = 1.0 \times 0.45 \times 0.3 \times 0.60 \times 1.0 \times 0.35 \times 1.0 \times 0.45 \times 1.0 \times 0.15 \times 0.25$$
$$= 0.00047840625$$

Naturally, the parse-tree $t_1$ is correct, as the probability of its construction is higher. The tree $t_1$ indicates that the observer (I) is seeing an astronomer carrying a telescope, while $t_2$ has semantics which indicates that observer (I) is seeing an astronomer with the help of telescope, which obviously is incorrect.

However, the accuracy of the probability of each tree depends on the accuracy of the probabilities of rules used. The probabilities associated with the rules are made

available from the statistics of semantics from a large corpus of natural language
text.                                                                                                    □

The probabilistic parsing provides a solution to ambiguity in language and gram-
mar, as it is conclusive from the above example. But, this is not necessarily complete,
but in partial. The other benefit is that it gives a *probabilistic language model*.

## 19.9   Information Extraction

We consider the following small text, which we would like to use for *Information
Extraction*.

> Firm XYZ is a full service advertising agency specializing in direct and interactive market-
> ing. Located in Bigtown CA, Firm XYZ is looking for an Assistant Account Manager to help
> manage and coordinate interactive marketing initiatives for a marquee automotive account.
> Experience in online marketing, automotive and/or the advertising field is a plus. Assistant
> Account Manager responsibilities ensures smooth implementation of programs and initia-
> tives, helps manage the delivery of projects and key client deliverables …Compensation:
> 50, 000 − 80, 000, Hiring Organization: Firm XYZ.

Given the above text, the extracted information may be

This information can be loaded into a database and can be queried any number of
times to find useful information, at a convenience [4].

The general architecture of an IE (Information Extraction) system is "a cascade
of transducers or modules such that, at each step structure is added to the document
and, sometimes, it filter relevant information by application of rules" (see Fig. 19.14).

| INDUSTRY | Advertising |
|----------|-------------|
| POSITION | Assistant Account Manager |
| LOCATION | Bigtown, CA |
| COMPANY  | Firm XYZ |
| SALARY   | 50,000–80,000 |

The majority of current systems follow this general architecture, although specific
systems are characterized by their own set of modules. In general, the combination
of such modules allow of the functionalities for IE system, discussed below [21].

### 19.9.1   Document Preprocessing

Preprocessing of the documents can be achieved by a variety of modules such as:
text segmenters, filters, tokenizers, lexical analyzers, stemmers, and disambiguators
(POS taggers, semantic taggers, etc.)
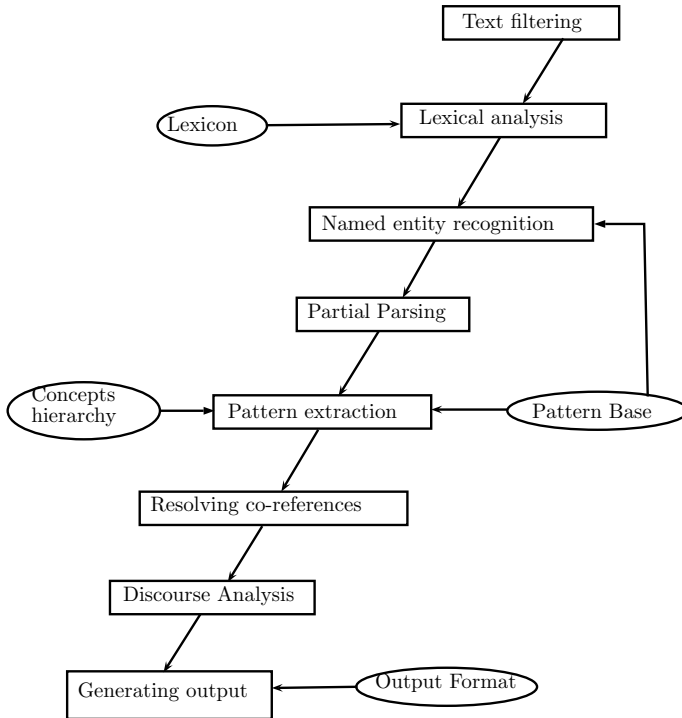
**Fig. 19.14**   Typical architecture of an IE system

The most relevant document processing activity to IE is Named-Entity recognition, i.e., recognition of proper nouns. The process of named-entity recognition may be performed using finite-state transducers (FST), together with dictionary lookup. These dictionaries are domain-specific, or they are terminological databases.

### 19.9.2   Syntactic Parsing and Semantic Interpretation

For performing the Information Extraction (IE), a traditional architecture based on Natural Language understanding was used. This method needed full parsing of sentences, and then followed by a semantic interpretation of syntactic structure obtained through parsing. The discourse analysis was carried out as the final step.

A new approach to IE is radically different from traditional, where only the concepts that are within the scope of extraction, are required to be found out in the documents. This leads to simplification of syntactic and semantic analysis due to more restricted, deterministic, and collaborative process. The new approach has the strategy that replaces the traditional parsing, interpretation, and discourse (module)

with a simple *phrasal parser* (parsing based on phrases). The later finds the local phrases. In addition, the discourse module does the job of *event pattern* matching, and merging the of templates. The above approach is useful because, the full parsing is expensive, not robust, hence produces ambiguous results, and cannot manage off-vocabulary conditions.

The current IE systems make use of partial parsing, such that the process of finding the constituent partial structure of a sentence consists one or more cascaded steps onto the text fragments. The generated constituents are tagged as parts-of-speech, like noun/verb, or phrases like prepositional or others. After parsing of the constituents, the system resolves domain-specific dependencies based on the semantic restrictions imposed by the extraction environment. The dependencies are resolved using the following two alternatives.

Pattern Matching

Majority of IE systems follow the approach based on pattern matching, called *named extraction* patterns or IE rules, to resolve the dependencies. In this approach, simplification of the syntax helps in reducing the semantic processing, and that leads ultimately to simplify the pattern matching task. The use of patterns is scenario-specific to recognize both, modifier and argument dependencies between constituents.

Grammatical relations

For representation of the grammatical relations, a graph is constructed (similar to dependency grammars) using the general rules of interpretation, such that previously detected chunks are constructed as nodes, and relations among these nodes are the labeled arcs. Such graphs are useful for reasoning about IE. There are three different types of grammatical relations for IE, as follows.

- First type is defined in the form of general relations of subject, object, and modifier.
- Second type is specialized *modifier relations*, which are specified as temporal relations, and relations concerning to locations.
- Third type is *mediated relations*, i.e., they are mediated by prepositional phrases.

### 19.9.3   *Discourse Analysis*

The task of the majority of IE systems is to represent the extracted information from the text in the form of partially filled templates or in some logical forms. Once represented/stored in a specified format, this information can be queried later, like we query a database. Since such information may be incomplete some times, it will result in partially incomplete templates. To recover the missing parts of the information to some extent, merging procedures are used to merge the partial templates, to explore the recovery of missing information.

However, when working with logical forms the IE systems can use traditional semantic interpretation methods in this phase.

### *19.9.4    Output Template Generation*

The output template generation phase is the final stage of IE systems. This stage maps the extracted pieces of information to the required output format. Due to the domain-specific restrictions in the output structure, some inferences can be drawn in this phase. The inference drawing can happen in the following situations:

1. a value from a predefined set is taken as it is by the output slots;
2. a forced instantiation of output slots;
3. when an extracted information generates multiple output templates, it results in inference;
4. some times, the *normalization* of the output slots produces inferences, e. g., when date in the coded form is normalized, it produces, say day or day of month or week, etc. Similarly, when products list is normalized we may get name of items instead of its code, etc.

## 19.10    NL-Question Answering

The Natural Language Question Answering (QA) derives the common features from NLP (Natural Language Processing) and Information Retrieval (IR). The QA (i.e., NL-QA) promises to deliver "answers", unlike the "hits" by IR. Most of QA is focused questions based on facts like "When did first human land on Moon?", "Who invented the paper clip?", and "How far is the Moon from earth?" These are called "factoid" questions, which can be typically answered using the *named entities* such as people, organizations, measures, dates, locations, etc. [7].

The commonly used approach for answering the factoid questions of open-domain requires the technique of named-entity recognition, together with IR. Typically, major steps of a "traditional" ontology-driven question answering system that is primarily based on IR and named-entity recognition technologies, are shown in Fig. 19.15. Usually, the large ontologies drive the process of QA, which relates the question types to semantic classes of answers. The detailed description of steps for QA is as follows:

1. *Question Analysis.* The first step of QA focus is on the analysis of the question, where knowledge resources are explored to find out the expected semantic type of the answer.
2. Document retrieval. Next, in the *document retrieval* phase, the candidates' documents containing the terms from the questions are retrieved. This retrieval is often done using off-the-shelf IR engines.
3. *Identification of passages.* In the next stage, the system *identifies passages* within these restricted candidate documents that contain a concentration of terms from the question–these regions are likely to contain the answer.
4. *Answer extraction.* Finally, in the *answer extraction* stage, named-entity recognizers identify candidates of the correct semantic type [9].

Question

Question Analysis

Document Retrieval

Passage Retrieval

Answer Extraction
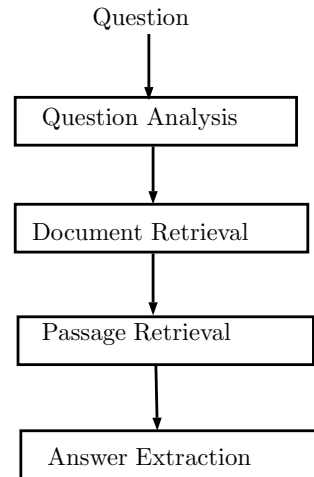
However, there are challenges of the QA approach discussed above. One is due to complex many-to-many mapping between question types and answer types. As an example, the answer to a "who" question can be name of a person (e.g., "Who invented the light bulb?") or an organization's name (e.g., "Who won the World Series in 2004?"), and there are other possibilities also. On the other side, different question types may map onto the single semantic answer type. For example, the questions – "Where was the world book fair held in 1900?" and "What city hosted the world book fair in 1900?", would map to the same answer. To overcome these challenges of multiple questions with single answer, a system must have elaborate ontological resources which explicitly encode semantic relationships between questions and answers. More advanced techniques such as *abductive inferencing*, *feedback loops*, and *fuzzy matching* of syntactic relations are used, but the factoid question answering is still driven by information retrieval and named-entity recognition technologies, and only on large ontologies.

## 19.10.1   Data Redundancy Based Approach

The data redundancy based approach depends on statistical regularities, using which it is possible to extract "easy" answers to factoid questions from web. For answering a question, the system makes some connection between the question and the passages containing the answers. Having done this connection at the lexical level, the rest of the job is simple. The meaning of the lexical connection between the question and likely answer carrying passage is the presence of large degree of overlap between these texts. But, in reality, it is not the case very often. This is because, the richness of natural language, and its expressive power provides the facility to create varying

textual forms which have the same meaning (semantics). But, in that case also, there is a good degree of semantics resemblance between the question and answer carrying text. To appreciate this scenario, consider the following question, with two different answers, which though lexically different but have the same semantics.

*When did Sikkim become the state of India?*

1. Sikkim became a state on 16 May 1975.
2. Sikkim was admitted to India on May 16, 1975.

In the above cases, both the passages contain correct answers, however, it seems obvious that for computer, it would be easier to extract the answer from passage 1 than from passage 2. We note that the task of answering the factoid questions will be far easier for the computer if the answer passage carries the same words as that in the question.

From the point of view of text processing algorithms, answers can be expressed in a variety of different ways. Within text collection, it is possible that the answer to above question lies in passage 2. Though in that text, the answer is not obviously stated. In such cases, since the answers share few words common with the question, a sophisticated natural language processing may be required to find out the relation between them. These processing may typically comprise the following types of domains [7].

> Recognizing syntactic alternations,
> Collapsing of paraphrases,
> Resolving the anaphora,[2]
> Making commonsense inferences, and
> Performing date calculations.

### 19.10.2 Structured Descriptive Grammar-Based QA

A special grammar, called SDG (Structured Description Grammar) can be used for information representation and extraction. The basic approach here is data redundancy based. In this, a text sentence ($S$) is mapped into a transition graph or state diagram, as shown in Fig. 19.16. It shows seven types of structurally different sentences labeled as 1, 2, …, 7, based on the number of *wh*-pronouns and their positions, which can be mapped into this structure. The structure of each sentence explicitly indicates the position of interrogating *wh*-pronouns, *who*, *what*, *where*, *when* and *why*. The positions of these wh-pronouns are invariant [3].

Following examples demonstrate that English-language sentences structures map into the transition diagram of SDG. It can be easily verified that the sentences (a) to (d) are of types 7, 7, 3, 4 respectively.

---

[2]Anaphora: making use of a pronoun or similar word instead of repeating a word used earlier.
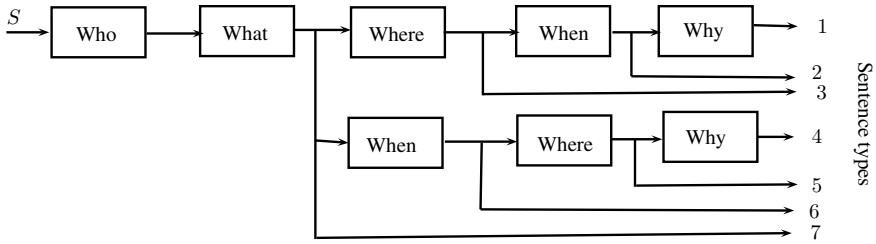
**Fig. 19.16** Transition diagram of sentence structure

(a) Akbar | followed a liberal policy for religion.
   Who | *What*
(b) Jahangir | Married Nur Jahan.
   *Who | What*
(c) Red fort | is located | at Delhi.
   *Who | What | Where*
(d) Shah Jahan | built Taj Mahal | during his rule | at Agra | in the memory of queen Mumtaj.
   *Who | What | When | Where | Why*

It may be noted that the structures of the sentences in SDG are simple English-language sentences. Further, in SDG, i) a question is allowed to carry more than one subquestion, as a sentence has more than one position for wh-pronouns, ii) sub-answers for a question which are distributed in a single document can be aggregated to fill up the slot-like structures in the transition diagram to generate a single answer, and iii) sub-answers for a question which are distributed in multiple passages can also be aggregated to fill up the slot-like structures in the transition diagram to generate a single answer.

## 19.11   Commonsense-Based Interfaces

To make our computers easier to use, it is required to make them understand the meaning of what we tell them. Usually these efforts are failed because meaning is not one thing but a combination of many. This is because, the activities of human thought engage an enormous collection of different structures and processes [10].

In human understanding, what some thing $X$ means to us depends on how representations of $X$ (in our brain) connects to, that is, having the links to other things we know. If we understand something in only one way then we know it very little, because when something goes wrong, we are left no links to connect that object (sense) with others. But if we use several features, each integrated with its set of related pieces of knowledge, then if one of them fails we can switch to another. This we implement by turning our ideas around in our mind to examine the other pieces

of knowledge from different perspectives, and carry on this process until we find one that works.

When our goal is that our computers must understand us, we will need to equip them with this kind of facility, i.e., they should connect each object/action (sense) to many other senses. For example, the sense "cat" is connected with other senses: color, size, height, no. of legs, tails, eyes, ears, leopard shape, etc. Those ways computers must be equipped with knowledge, like human.

Present Limitations of Computers

For computers to understand like humans, there should be a program with common sense like humans. This is a difficult task, because a typical program has only one way to deal with a problem, so if something goes wrong, the program is totally stuck. However, as humans, we search for alternatives in the event of failure of one approach. The limitation with present-day computers is that they always start from scratch. To make the computers of more worth, we need to supply them a great library of *commonsense knowledge*, which are common even with young children.

The present-day computers are not designed with commonsense in them, nor they have capability to learn from experience. The programs can solve difficult problems in specialized subjects, but even today, computers cannot do the things, which even young children do so easily. The current programs behaves in a limited and inflexible way. Some of the critical differences between the capabilities of computers and human are distinctly explained below.

*Vitalist*

Computers can do only what they are programmed to do so. The computers have been programmed by people to speak, but in fact the computers do not know what those words mean. The meaning is an intuitive thing, and it cannot be reduced to zeros and ones. The present computers can only do logical things, and meanings are not necessarily logical.

*Humanist*

The machines are not humanist, as machines have no goals or hopes, nor any fears or phobias. They do not even know that exist, nor they have a sense of accomplishment.

*Logician*

Since we want that computers should have commonsense, it is more important to understand as what the commonsense is? We need to define it more clearly and precisely. That policy seems alright, but it is wrong when it comes to psychology. Of course, we do not like to be imprecise, but some times, the definitions are not sufficient. So instead, we will take a different approach, and try to design (as opposed to defining) machines that can do the things we want.

### 19.11.1  Commonsense Thinking

When we say "commonsense thought," we are indicating to things that most people can accomplish, even not knowing many times that they are doing them. Thus, when we hear a sentence like: "Bill told the waiter he wanted some chips," we will infer all possible inferences. Some of them are [10] the following:

– The word "he" means Bill, and not the waiter.
– This event possibly took place in a restaurant, where Bill was a customer, and waiter was close to him at the time. The waiter was working there, waiting for Bill's meal order at that time.
– Bill wants potato chips, not memory or IC chips. No count of chips is mentioned. But, it may be 20–30, and in thousands.
– Bill will start eating the chips soon after he gets them.
– Bill and waiter are both human beings, Bill is old enough to talk, and the waiter is old enough to work.
– This event happened after the date of invention of potato chips (i.e., year 1853).
– It is Bill's assumption that waiter also infers all those the abovementioned things.

Every child learns to use thousands of words at a very young age, but no computer knows even the meaning of some of those, so no computer can even understand a casual conversation. For example, if you mention about "string," any child would know what it means, because it knows that many places are there it can be used, for example, it can be used to tie a cart and pull it. With this, the child would also know things like these:

– An object can be pulled by a string, and cannot be pushed by it.
– It is not good for eating, and you cannot construct a cart using a string.
– Before we put a string into a box, the box needs to be opened.
– And so, on.

Let us find out the size of networks of commonsense knowledge, i.e., network of, say 1000 words, each word having links with other knowledge structures, in different ways. Some links are for objects, while others are for processes. Each such link will, in turn, lead to other links (all in a semantic network), so that whenever some process gets stuck, we usually can find some alternative. Language is not the only feature with such abilities, but each expert skill a person possesses, there must be other similar order of structures, say, for vision, for hearing, for perception, for speech, and for all kinds of physical manipulations, and also for various kinds of social knowledge. So, it may be that we possess millions/billions of units of knowledge.

### 19.11.2  Components of Commonsense Reasoning

The commonsense thinking we do every day involves a large number of hard-earned ideas, which includes masses of factual knowledge about the problems we are trying

to solve. Not only the learning, but we must also adopt efficient ways to retrieve and apply the relevant knowledge. Many processes gets engaged in the activities of imagining, planing, predicting, and deciding, which also make use of many exceptions and rules. For these, it requires knowledge about how to think, i.e., how to organize and control those processes, also even if the representations are different, it can describe the same situation. In addition, there must be a facility to convert the new experiences into suitable memory structures to represent them. Some of the possible structures are: property lists, frames, frame-arrays, database query languages, explanation-based reasoning, logic programming, rule-based systems, semantic networks, scripts, and stories.

The first step in knowledge representation is to select a representation. Using any specific representation will shortly lead to limitations, which may quickly accumulate, leading the reasoning to ultimately halt. It is usually required to use several different representations for each fragment of the commonsense knowledge about each *idea*, *thing*, or *relationship*. We swiftly change from one method of representation to other methods. And, that depends on which methods are better than other methods for solving problems.

Finally, the following three are the important capabilities we need for characterizing the problems for making use of commonsense reasoning.

*Negative Expertise*

This deals with ways through which we recognize each of the methods as when they are going to fail. If it becomes possible to recognize the way particular things went wrong, it becomes a clue for deciding what action should be taken next. By knowing the way how each of the methods is likely to fail, it can be used to control the higher activity, e.g., by brain we control the mental activity.

*Knowledge Retrieval*

Retrieving the relevant information from the commonsense knowledge networks requires appropriate methods to identify, as what problems or situations in the past are having maximum resemblance to the context of present problem. This means, the systems need ways to describe what we are trying to do, and then reason about those descriptions. This is based on the skillful use of analogies.

*Self-reflection*

Our computers, when implementing commonsense reasoning, are required to keep records which describe *acts* and *thinking* they have recently done, so that they are able to *reflect* on the results of what they tried to do. Also, they must be aware of what they are doing. This is what we call with human as *consciousness*.

How the programs like playing chess or proving theorem are different from commonsense reasoning? Is the reasoning process in a child's brain for fixing blocks of different colors is a game, simpler than that of chess or theorem proving? The answer is No. There is a fundamental difference between these programs, like for chess game or theorem proving versus playing these block games. In fact, the programs of expert systems, like chess or theorem prover, require much less knowledge

skills. However, the children make use of thousands of skills, they manipulate all kinds of things, irrespective of their texture and shapes; they stack the things up and then knock down then, and learn the dynamics of how the stack got scattered. Even for building a small toy house of blocks, one needs to mix and match the knowledge of many kinds: that is, about shapes and colors, speed, space and time, support and balance, stress and strain, and the economics of self-management.

Do computers lack in learning all these things? Not likely. These limitations persist because we have learned to program only in certain ways. Our decades-old approach to solve some problem $X$ has been: find the best way to represent $X$, find the best way to represent the knowledge needed to solve $X$, and find the best way to solve $X$, and so on! These steps lead to write *specialized programs* that cope with solving only that type of problems—called *brittle programs*! This has resulted to millions of specialized programs for solving only that kind of problems, such as playing chess, playing cards, or designing a bridge, banking, etc. In the computer programs, price for best was sacrificed in limiting the resources.

To make the machines deal with commonsense reasoning, we must use multiple ways to represent the knowledge, acquire huge knowledge in that, and find common-sense ways to reason with this. Consider the following example.

> Mary was invited to Jack's party.
> She wondered if he would like a kite.

What leads us to infer that Mary was thinking to take a kite as a gift, but there was no mention of "birthday" or "present" in the first sentence. There should be a suitable representation of phrase "invited to party", which could help to infer that she is thinking of a kite as a suitable gift for Jack.

### 19.11.3   Representation Structures

It is not well known as which structure of representation is best suitable for any specific purpose. On the contrary, the brain represents a problem in several ways, as well as represents the required knowledge. When one method fails to solve a problem, we quickly switch over to another approach.

Now, to reply to the above question, we do a small analysis. Consider that we are traveling by electric train, and there is a possibility of it being halted in any of the stations due to electricity failure. Assuming that there are 10 railway stations, and it may stuck at any one of them. Therefore, it is not difficult to plan for food and shelter, as there can be a maximum 10 different solutions. So, given a cause of halting of train, there are a maximum of 10 possible effects.

Consider another example, a possibility of meteorite strike on earth of sufficient size, and depending on where it strikes, there are different rescue operations and strategies needed depending on where it strikes. Naturally, for a single cause, there are hundreds or thousands of possible different measures needed(effects), and even more. Naturally, this second problem is more difficult to solve.

**Fig. 19.17** Causal-diversity matrix
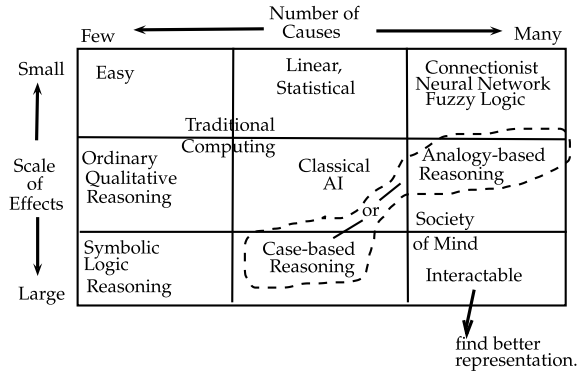


The cause-effect matrix (shown in Fig. 19.17) illustrates the relation "*cause × effect*" to reasoning process, which can be used to arrive at the solution. In the cases, when the causes are only a few, and each cause having a small number of effects as shown in the top-left corner cells of the matrix, such the problems are easy to solve even by exhaustive search methods, or some times there is no need to search, but simply answer is recalled [10].

When the causes are many, each with a small effect, then statistical methods and neural networks may work well, as indicated by the top-right corner cell of the matrix. But, such systems are likely to breakdown if those causes have different characters that interact in hard-to-predict ways.

The symbolic or logical reasoning, shown at the bottom-left corner cell of the matrix, can work well when the causes are few, but corresponding effects are in large number, except that the search may exponential in the case of sequential processes.

It is a rare situation, when many causes correspond to a large number of effects, indicated in the right-bottom of the matrix, such systems are interactable, unless the system is linear. Sometimes it is possible to arrive at solutions by reformulating those difficult problems by switching to different representations, that emphasize fewer and more pertinent features, so that we can work with simpler descriptions.

Except in the right corners of the matrix, there are multiple causes with modest number of effects, the heuristic programs succeed in these conditions, using knowledge combined with controllable search. This is the region of "classical AI" research—the causes and effects are both moderate. Consider the adjacent cells to the bottom and to the right of classical AI. In these regions, analytical methods are usually not helpful, but it may be possible to use use our knowledge and experience to construct and apply analogies, plans, or case-based reasoning methods. Such methods may not work perfectly, but they are frequently useful for practical purposes.

The conclusion of the above discussion is that there cannot be a single type of approach possible for all types of problems. Accordingly, one should not seek a uniform way to represent commonsense knowledge. In fact, there is frequent need to use several representations when we face a difficult problem and then we will need additional knowledge about it. The *causal-diversity* method may help, but eventually

**Fig. 19.18** Architecture of
representations
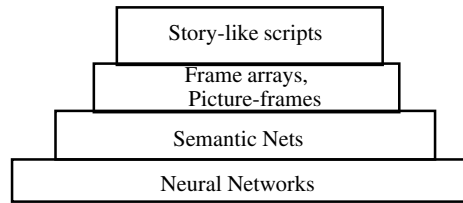


it must be replaced by more resourceful, *knowledge based* systems that can generate
new representations.

In fact, there is no best way to represent the knowledge. The present limitations of
machine intelligence are largely due to our quest for seeking unified theories capable
of reasoning well in all situations. The versatility human can emerge from a large-
scale architecture of representation, where each of several different representations
can help overcome the deficiencies of other representations.

Consequently, the commonsense knowledge in the machines be built, which rep-
resents knowledge about so many things like strings, houses, clothing, roads, books;
in other words, everything that most children know. For such commonsense knowl-
edge base, we will need ways to link each unit of knowledge to the use, or functions
that each unit knowledge can serve. Figure 19.18 shows some typical representation
levels, which have been covered in previous chapters.

## 19.12  Tools for NLP

There are a number of tools that exist, either as open-source or research tools created
by some research laboratories, as well as closed source for natural language process-
ing, and speech processing. These tools have built-in functions to perform a number
of commonly used tasks, which can be directly called, or using these scripts can be
written to perform more complex jobs of NLP and speech processing. For example,
for NLP, they can tokenize the given text, can do stemming and POS (parts of speech
tagging), can find out word frequencies in given documents, parsing of NL sentences,
etc. These inputs can help to compute, for example, $tf \times idf$, which can be helpful
in IR (information retrieval), IE (information extraction), text classification, etc. In
the following part, we discuss some such tools, which are either open source or they
can be obtained on request from respective research laboratories.

### 19.12.1  NLTK

The NLTK (Natural Language Toolkit) is a collection of Python libraries and pro-
grams for *symbolic* and *statistical natural language processing*. It is suited to practi-
tioners as well those who are learning natural language processing (NLP), those who

are conducting research in NLP or areas close to NLP, like, empirical linguistics, cognitive science, AI, IR, and machine learning.

The NLTK has been also used as a teaching tool, as a study tool for individuals, and as a prototyping and building platform for research systems. Python language has been chosen due to its shallow learning curve, its transparent syntax and semantics, and due to its extraordinary capability for handling strings. Python is an interpreted language, which facilitates interactive exploration. It is an object-oriented language, allows data and methods to be encapsulated and reused easily. Python is also available with an extensive standard library, that includes tools for speech processing, natural language processing, numerical processing, and graphical programming [12].

Consider that tasks of stemming and parts-of-speech (POS) tagging are independent, and both operate on sequences of tokens. If the stemming task is done first, the information required for POS tagging is lost. If tagging task is performed first, the stemming process must be able to skip over the tags. If these two tasks are done independent of each other, it becomes difficult to align the resultant texts. Hence, as the combinations of tasks increase, it becomes extremely difficult to manage the data. To address this problem, NLTK version 1.4 onward comes with a new architecture where tokens are based on Python's native dictionary datatype, such that the tokens can have an arbitrary number of *named properties*. The *Tag* and *Stem* are examples of these properties. The NLTK allows for even whole sentence and document to be represented as a single token with *Sub-tokens* attribute that holds sequences of smaller tokens.

A *parse-tree* can also be treated as a token, which has special property/attribute of *Children*. The benefit of this type of architecture in NLTK is that it unifies many different data types, and allows distinct tasks to be run independently. Of course, this architecture comes with an overhead for programmers, because the program needs to keep track of a growing number of property names.

### *19.12.2   NLTK Examples*

**Example 19.4**   Tokenization of natural language text.

The following commands in Python, with NLP tool NLTK installed, demonstrate the tokenization of a given text into sentence tokens and word tokens. Since there is only one sentence, the sentence token is one only.

```
$ python
Python 2.7.14 (default, Sep 23 2017, 22:06:14)
[GCC 7.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more
                                             information.
>>> from nltk.tokenize import sent_tokenize, word_tokenize
>>> text="Fundamentals of Artificial Intelligence."
>>> print(sent_tokenize(text))
```

```
    ['Fundamentals of Artificial Intelligence.']

>>> print(word_tokenize(text))
    ['Fundamentals', 'of', 'Artificial', 'Intelligence', '.']
>>>
```

□

**Example 19.5**  Stemming of given set of words.

Following are the commands for stemming a set of words to reduce them to their stem words. This makes use of the Porter Stemmer algorithm.

```
>>> from nltk.stem import PorterStemmer
>>> ps=PorterStemmer()
>>> words=["python", "pythoning", "pythonize", "pythonly"]
>>> for w in words: print(ps.stem(w))
...
python
python
python
pythonli
>>>
```

□

The parts-of-speech tagging (grammatical tagging) or disambiguation of word-category, is a process of marking-up word in a text (corpus) corresponding to a particular POS. This is carried out based on its definition as well as its context.[3] Various POS in the English language are: noun, verb, adjective, adverb, pronoun, preposition, conjunction, and interjection.

The POS tagging is carried out as part of computational linguistics using some algorithms. These algorithms associate discrete terms, as well as hidden parts of speech, in accordance to a set of descriptive tags. POS-tagging algorithms fall into two distinctive categories: *rule-based* and *stochastic* based. For example, Brill's tagger, one of the first and most widely used English POS taggers, makes use of rule-based algorithms. The following example demonstrates the POS tagging using NLTK.

**Example 19.6**  Parts-of-speech tagging.

```
>>> import nltk
>>> text=nltk.word_tokenize("Part of speech tagging and POS
                                                    tagger")
```

---

[3]Context: Relationship with adjacent and related words in a sentence, or phrase, or a paragraph.

```
>>> text
 ['Part', 'of', 'speech', 'tagging', 'and', 'POS', 'tagger']
>>> nltk.pos_tag(text)
 [('part', 'NN'), ('of', 'IN'), ('speech', 'NN'),
 ('tagging', 'NN'), ('and', 'CC'), ('POS', 'NNP'), ('tagger',
                                                     'NN')]
```

□

## 19.13   Summary

Natural language processing (NLP) is an academic, and technology-based research domain comprising a range of computational techniques for representation and automatic analysis of human languages—a field that is motivated by theory. Automatic analysis of text requires a deep understanding of natural language by machines. However, we are still far away from machines that have this capability. To develop a program that can understand that natural language is a challenge, because most of the natural languages are large and complex, which can have infinitely large number of sentences. In addition, there is ambiguity in natural languages, as many words have more than one meaning, such as *can*, *bear*, *fly*, *orange*, and many others. That gives different meanings to the same sentence, when used in different contexts. Due to this problem, creating programs that understand the NL correctly is a difficult task.

Right from its start, the field of NLP focused on areas like machine translation, IR, IE, question answering, text summarization, topic modeling, and more recently, on opinion mining. Although the *semantic* problems and the actual requirements of NLP were apparent right from the beginning. However, for the more direct applicability of *machine learning techniques*, the strategy adopted was to tackle *syntax* first. This was due to the fact that semantics was considered a more challenging problem.

One of the early representation strategies in NL was FOPL (First-Order Predicate Logic)—a deductive system comprising *axioms* and *rules of inferences*. The FOPL supported to good degree of handling the syntax, but limited order of semantics and *pragmatics*. The syntax is concerned with the well-formedness of the expressions, while semantics specifies what the well-formed expressions mean. The handling of pragmatics is more challenging, they specify how the contextual information can be used to provide a better correlation between different semantics. Properly specifying of pragmatics is essential for the tasks such as WSD (Word-Sens Disambiguation).

*Production rule* based language model is one of the popular models for the description of Natural language (Chomsky, 1956). Other important models for NLP are Ontology Web Language (OWL), which uses XML-based vocabulary for ontology representation, e.g., definition of classes and relation between them, classes' properties, and constraints on the properties, and constraints on the relations. Network-based models are also common for NLP, like Bayesian networks, which provides joint probability distribution over many interrelated hypotheses, and semantic networks—a

graphical notation for representing knowledge in patterns of interrelated interconnected nodes and arcs.

Some of the common applications of NLP are: Classification of text into categories, Index and search large texts, Automatic translation, Information extraction, Automatic summarization, Question answering, Knowledge acquisition, and Text generations/dialogues.

Syntax of natural language is checked by generating using its grammar, called *phrase structure grammar*. The generating process is a derivation, similar to rigorous proof. Different grammars are classified as per Chomsky hierarchy of grammars, called type 0, 1, 2, and 3, with last one most simple, and first as the most complex.

A sentence of natural language is represented by a syntax-tree, which shows the relations between various constituents of the sentence. Every sentence is supposed to have a corresponding one and only one syntax tree. If a sentence can be generated using more than one syntax-tree, then the language of that sentence as well the grammar are said to be ambiguous—having more than one meaning of the sentence.

Parsing a sentence is the process of computing the structural description of the sentence assigned by the grammar, assuming that the sentence is well-formed. Parsing consists of : Mathematical characterization of derivations in the grammar using a specified algorithm. A parse-tree describe the structure of a sentence, and is also the record of the history of derivation of the sentence. The parse-trees are useful for grammar checking of the sentence, which is an important intermediate stage in semantic analysis, and it plays an important role in applications of language translation, question answering, and information extraction.

Parsing can be top-down—a sentence is generated using start symbol, or it can be bottom-up—a sentence is reduced to start symbol through a sequence of steps. Further, the parsing can be *deterministic*—each production rule has equal weight, in comparison to probabilistic parsing—there is a probability weight associated with each production rule. Usually, in parsing, a sentence is split into NP (noun phrase), and VP (verb phrase). More extensive English grammar is obtained by with the addition of other constituents, such as PP (prepositional phrase), ADJ (adjectives), DET (determiners), ADV (adverb), AUX (auxiliary verb), etc.

Information extraction (IE) systems extract the important information from natural text, and load into databases, that can be queried later a number of times to find useful information, and at convenience.

The general architecture of an IE system is defined as "a cascade of transducers or modules that, at each step, add structure to the documents and, sometimes, filter relevant information, by means of applying rules." Most existing IE systems are based on partial parsing. Generally, the process of finding constituents consists of using a cascade of one or more parsing steps against fragments. The resulting constituents are tagged as noun, verb, prepositional phrases, etc.

Syntactical parsing for IE defines some grammatical relations (subject, object, and modifier), and some specialized modifier relations (temporal, and location). This is made possible using a dependency graph built following general rules of interpretation for the grammatical relations, with previously detected chunks as nodes, and relations. The other jobs performed by IE are: discourse analysis, output template generation, and NL-language question answering.

Question-answering is possible using a special format of grammar called structured descriptive grammar. It is possible to map a sentence with a sequence of *wh-pronouns* (*who, what, when*,...), and consequently, determine the value of a missing wh-pronoun, thus, answering a question.

Preprocessing, is a common phase in NLP applications discussed above. It is carried out on documents using the program modules such as: text zoners, segmenters, filters, tokenizers, lexical analyzers, disambiguators (POS taggers, and semantic taggers), stemmers, etc.

To make our computers to understand us, we need to equip them with adequate knowledge. To help this work, the computer must know what our jobs are. To entertain us they will need to know what their audiences like or need. This requires to create commonsense reasoning in computers. However, it is not well known as which structure of representation is best suitable for any specific purpose. On the contrary, the brain represents a problem in several ways, as well as represents the required knowledge. When one method fails to solve a problem, we quickly switch over to another approach. That means, commonsense reasoning is that we should not seek one uniform way to represent commonsense knowledge. We will frequently need to use several representations when we face a difficult problem and then we will need additional knowledge about it.

## Exercises

1. What are the challenges of NLP?
2. Give one example of the following ambiguities:

   a. Phonetic
   b. Syntactic
   c. Pragmatic

3. What are the applications of NLP?
4. Develop the parse-tree to generate the sentence "Rajan slept on the bench" using following rewrite rules:
5. Draw the tree for the following phrases:

   a. after 5 pm.
   b. on Tuesday.
   c. From Delhi.
   d. Any delay at Mumbai.

6. Draw the tree structures for the following sentences:

   a. I would like to fly on Air India.
   b. I need to fly between Delhi and Mumbai.
   c. Please repeat again.
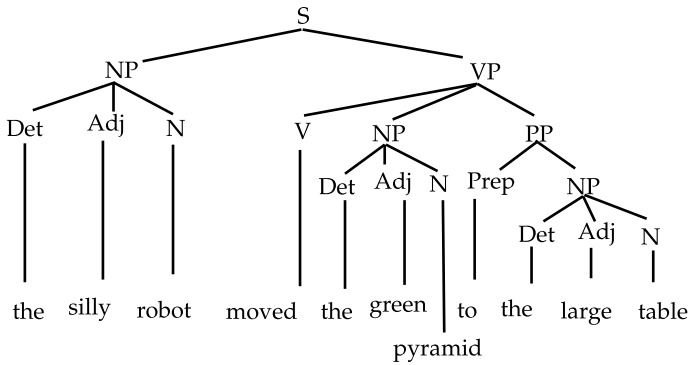
**Fig. 19.19** Parse-tree

7. Convert the following passive voice to active voice. Construct the necessary trees. Also write the steps.

   The village was looted by dacoits.

$$S \rightarrow NP \ VP$$
$$NP \rightarrow N$$
$$NP \rightarrow Det \ N$$
$$VP \rightarrow V \ PP$$
$$PP \rightarrow Prep \ NP$$
$$N \rightarrow Rajan \mid bench$$
$$Det \rightarrow the$$
$$prep \rightarrow on$$

8. Given the parse-tree in Fig. 19.19, construct the grammar for this.
9. Construct the grammars and parse-tree for the following sentences.

   a. The boy who was sleeping was awakened.
   b. The boy who was sleeping on the table was awakened.
   c. Jack slept on the table.

10. Construct the parse-trees and resolve the ambiguities in the following sentences using "selectional constraint". Also, specify whether the ambiguities are syntactic, semantic, or some other?

    a. "He saw the man with the horse."
    b. "he saw the man with gun."
    c. "He saw the man with binocular."

11. What are the different types of ambiguities in natural language, like English?

# References

1. Cambria E, White B (2014) Jumping NLP curves: a review of natural language processing research. IEEE Computat Intell Mag 5:48–57
2. Chomsky N (1956) Three models for the description of language. IRE Trans Inform Theory 2(3):113–124
3. Chowdhary KR, Bansal VS (2006) Information Extraction from Natural Language Texts. *J of the Institution of Engineers(India)*, 87:14–19
4. Chowdhary KR (2004) Natural Language Processing for Word Sense Disambiguation and Information Extraction. PhD Thesis, JNV University, Jodhpur (India)
5. Jurafsky D, Martin J H (2002) Speech and Language Processing - An Intro to Natural Language Processing, Computational Linguistics, and Speech Recognition. Pearson Education Asia, ISBN 81-7808-594-1
6. Lenat D, Guha R (1989) Building Large Knowledge-Based Systems: Representation and Inference. The Cyc Project. Addison-Wesley, Boston, MA
7. Lin J (2007) An Exploration of the Principles Underlying Redundancy-Based Factoid Question Answering. ACM Trans on Inf Sys 25(2):1–55
8. Miller AM (1995) Wordnet: A Lexical Database for English. Communications of the ACM 38(11):39–41
9. Minsky M (1968) Semantic Information Processing. MA, MIT Press, Cambridge
10. Minsky M (2000) Commonsense-based Interfaces. Communications of the ACM 43(8):67–73
11. Mueller E (1998) Natural Language Processing with Thought-Treasure. Erik T. Mueller, New York
12. https://www.nltk.org/, Cited 19 Dec 2017
13. Pearl J (1985) Bayesian networks: A model of self-activated memory for evidential reasoning. UCLA comput Sci, Irvine, CA: *Tech Rep CSD-850017*
14. Ralph G (1994) Computational Linguistics - An Introduction, Studies in Natural Language Processing, Cambridge Univ Press
15. Reiter R (1980) A logic for default reasoning. Artificial Intelligence 13:81–132
16. Ronald C et al (1997) Survey of the state of art in human language technology – Studies in Natural Language Processing, Cambridge Univ Press
17. Schank R (1975) Conceptual Information Processing. Elsevier Sc Inc, Amsterdam, The Netherlands
18. Singh P (2002) The open mind common sense project. http://www.kurzweilai.net/ Cited 19 Dec 2017
19. Sowa J (1987) Semantic networks. *Encyclopedia of Artificial Intelligence*, S. Shapiro edn, Wiley, New York
20. Baldwin T (2009) et al (2009) Prepositions in Applications: A Survey and Introduction to the Special Issue. *Computational Linguistics*, Vol 35, 2:119–150
21. Turmo J et al (2006) Adaptive Information Extraction. ACM Computing Surveys 38(2):1–47

# Chapter 20
# Automatic Speech Recognition

**Abstract** There are basically two application modes for automatic speech recognition (ASR): using speech as spoken input or as knowledge source. Spoken input addresses applications like dictation systems and navigation (transactional) systems. Using speech as a knowledge source has applications like multimedia indexing systems. The chapter presents the stages of speech recognition process, resources of ASR, role and functions of speech engine—like, Julius speech recognition engine, voice-over web resources, ASR algorithms, language model and acoustic models—like HMM (hidden Markov models). Many open-source tools like—Kaldi speech recognition toolkit, CMU-Sphinx, HTK, and Deep speech tools' introduction, and guidelines for their usages are presented. These tools have interfaces with high-level languages like C/C++ and Python. The is followed with chapter summary and set of exercises.

**Keywords** Automatic speech recognition · ASR · Multimedia indexing · ASR resources · Language model · Acoustic model · Julius · Kaldi · CMU-Sphinx · HTK · Deep tools

## 20.1 Introduction

Speech has long been viewed as the future of computer interfaces, promising significant improvements in ease of use over the traditional keyboard and mouse. There are basically two application modes that exist for speech recognition: 1. Speech as *spoken input* to computers, and 2. The speech is used as *data* or *knowledge* source. The first application mode comprises the potential applications as, dictation systems, navigation, and transactional systems.

In the application of *dictation*, a system transcribes the spoken words into written text, and for dictating letters, reports, business correspondence, or e-mail messages, to the computer/machine.

The speech can be used in the form of commands to *navigate* around the applications, for example, selection of main application, then its one of the sub-applications, and sub-sub-application, till you reach to the command to execute the final application.

The speech can be used for *transactional* applications, i.e., to use speech in the form of command or command sequence to cause a transaction to be performed. For example, the speech-based transactions can be purchase of stock, book a flight ticket, reserve an itinerary for tour, or doing the fund transfer.

The second mode of speech application, i.e., speech as a knowledge source has applications like meeting capture, and knowledge management. These applications are basically multimedia indexing systems that use speech recognition to transcribe words verbatim from an audio file into text. Subsequently, the IR (Information Retrieval) techniques are applied to the transcript to create an index with time offsets into the audio. Users can access the index using text keywords to search a collection of audio or video documents.

We understand both written and spoken language, and also know that skill of reading is learned much later. We now focus on spoken language. The problem of understanding the spoken language can be divided into major parts discussed below [6].

*Phonological*

The phonological processing step relates sounds to the words we recognize. Phone is smallest unit of sound, and the phones are aggregated into word sounds.

*Morphological*

This is lexical knowledge, which relates to word construction from basic units called morphemes. A morpheme is the smallest unit of meaning, for example, the construction of *friendly* from *friend* and *ly*.

*Syntactic*

It is knowledge about how the words are organized to construct meaningful and correct sentences.

*Pragmatics*

It is a high-level knowledge about how to use sentences in different contexts and how the contexts effect the meanings of the sentences.

*World*

It is useful in understanding the sentence and carry out the conversation. It includes the other person's beliefs and goals.

Speech recognition by machine is important, as many problems get solved if our computer/laptop can recognize the spoken works. We know, in the present days it has become possible to do search in some of the search engines by speaking rather than entering the keywords. In the following, we discuss some of the potential applications of automatic speech recognition (ASR).

**Learning Outcomes of this Chapter:**

1. Distinguish the goals of sound recognition, speech recognition, and speaker recognition and identify how the raw audio signal will be handled differently in each of these cases. [Familiarity]

2. Implement a feature-extraction algorithm on real data, e.g., an edge or corner detector for images describing a short slice of audio signal. [Usage]
3. Language model and acoustic models for speech recognition. [usage]
4. Implement an algorithm combining features into higher level percepts, e.g., phoneme hypotheses from an audio signal. [Usage]
5. Evaluate the performance of the underlying feature extraction, relative to at least one alternative possible approach (whether implemented or not) in its contribution to the classification task. [Assessment]
6. Tools for speech recognition system. [Usage]

## 20.2   Automatic Speech Recognition Resources

At the simplest level, the programs that are speech driven can be characterized by the words or phrases we say to a given application and how that application interprets them. An application's active vocabulary is what it listens for, determines what it understands. The speech recognition process makes use of a speech engine, which is language-independent, and what it recognizes can be from several domains. A domain comprises a vocabulary set, pronunciation models, word-usage models that are associated with specific speech applications. An acoustic component is also present in the speech recognition engine, as part of voice models used by the speech engine during the recognition. The voice models can be speaker independent, or may be unique to the speaker.

Figure 20.1 shows the resources used by a typical speech engine during the process of speech recognition. The domain-specific resources can vary dynamically during



**Fig. 20.1** Speech recognition resources
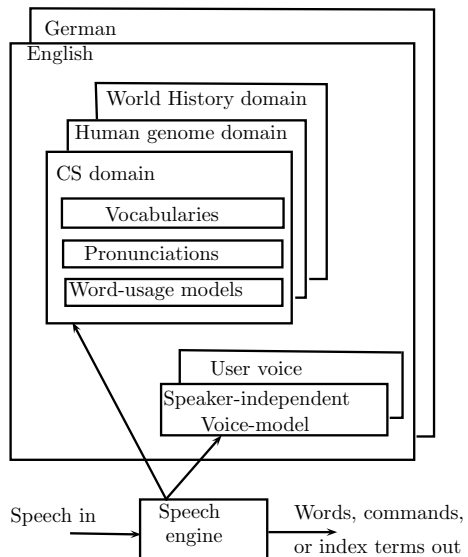
a given recognition session. The vocabulary is one of the domain-specific resources. Some of the major applications are as follows [7].

1. *Dictation application.* It transcribes spoken input directly into the document's text content.
2. *Transaction application.* It facilitates a dialogue leading to a transaction, and
3. *Multimedia indexing application.* This application can generate words, which act as index terms into the multimedia.

As far as application development is concerned, speech engines typically offer a combination of programmable APIs (Application Programming Interfaces) and tools that are helpful to create and define vocabularies and pronunciations for the words they contain. A transactional application may use a smaller, task-specific vocabulary of a few hundred words, but a dictation or multimedia indexing application may use a predefined large vocabulary, some times as large as 100,000 words or so.

A small size of vocabularies is enough for some applications. However, they pose usability problems in the system as their size grows. This is because, the system requires a strict enumeration of the phrases, which must be recognizable by any of the given states in the application. To overcome this limitation, *speech grammars* for specific tasks are defined in the transaction-based applications, which provide an extension to the single words or phrases a vocabulary supports. The speech grammars are helpful in constructing structured collection of words and phrases bound together by rules that define the set of speech streams the speech engine can recognize at a given time. For example, using these grammars, the developers can define a grammar that permits flexible ways of speaking a date, a dollar currency, a number, etc. The prompts that cue users on what they can say next, are an important aspect of defining and using grammar. Further, the speech grammars can serve as a critical component of enabling the voice over the Web, discussed in the next section.

## 20.3   Voice Web

To have voice facility over the web, a group of industry organizations, which included AT&T, IBM, Lucent, and Motorola, had established the VoiceXML Forum, in March 1999 to develop and promote a new language—the *Voice extensible Markup Language* (http://www.w3.org/Voice/). The VoiceXML forum was established with the objective, to bring the content delivery to interactive voice response applications, using Web-based developments [7, 8].

The VoiceXML (or VXML) is a digital document standard for specifying interactive media and voice dialogues between humans and machines. This language has applications for developing audio and voice response applications, such as for banking systems, customer services, and for automated customer service portals. VoiceXML applications are commonly used in industries and segments of commerce, some of the examples are as follows.

Speech synthesis,
Recognize spoken and touch-tone key input,
Digitize audio,
Order inquiry,
Package tracking,
Flight tracking,
Voice access to email,
Emergency notification,
Audio news magazines,
Can record spoken input,
Like voice dialing, and
Directory assistance.

VoiceXML applications are developed and deployed in all the above major fields. These applications are analogous to how a web browser interprets and visually renders the Hypertext Markup Language (HTML) it receives from a web server. In similar ways, the VoiceXML documents are interpreted by a voice browser, and users interact with voice browsers via the public network like Internet. Like HTML, the VoiceXML text provides *tags* that instruct the voice browser to provide the functions of automatic speech recognition, speech synthesis, dialogue management, and audio playback.

Figure 20.2 shows the architecture of VoiceXML model, which makes use of standard client–server paradigm that integrates voice and data services. A voice service consists of sequence of voice interaction dialogues between a user and an implementation platform. A *document server*, which can be external to the implementation platform, provide the dialogues. The overall service logic is maintained by the document servers or web servers, which also perform database and legacy system operations, and produce dialogues.

A VoiceXML document specifies each interaction dialogue that is conducted by the VoiceXML interpreter. The following is an example of a VoiceXML document:



**Fig. 20.2** VoiceXML architectural model
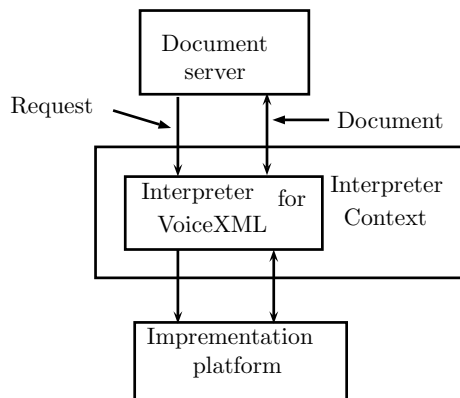
```
<vxml version="3.0" xmlns="https://www.w3.org/TR/voicexml30/">
  <form>
    <block>
      <prompt>
        This is a VXML Code!
      </prompt>
    </block>
  </form>
</vxml>
```

A user's input affects dialogue interpretation, and the system collects this information in the form of requests, which it submits to a document server. The later can reply with another VoiceXML document to continue the user's session with other dialogues. The grammar-based recognition vocabularies are commonly used to support voice services in VoiceXML applications.

## 20.4   Speech Recognition Algorithms

The initial attempts for speech recognition were targeted to use expert knowledge of speech production and perception processes. Soon it was found that such knowledge was inadequate for capturing the complexities of continuous speech. Currently, the statistical modeling techniques trained using hours of speech have provided most speech recognition advancements [4].

The process of speech recognition starts with a sampled speech signal. This signal has a good deal of redundancy because the physical constraints on the articulators that produce speech—the glottis, tongue, lips, and so on—prevent them from moving quickly. Consequently, the ASR (Automatic Speech Recognition) system can compress information by extracting a sequence of acoustic feature vectors from the signal.

Typically, the system extracts a single multidimensional feature vector every 10 ms that consists of 39 parameters. These feature vectors, which contain information about the local frequency content in the speech signal, are called *acoustic observations* because they represent the quantities the ASR system actually observes. The system attempts to infer the spoken word sequence that could have produced the observed acoustic sequence.

To simplify the design, we assume that speaker's vocabulary is known to the ASR system. Having adopted this approach, it is helpful in restricting the search for the possible word sequences only within the words listed in the ASR lexicon. The ASR lists the vocabulary and provides *phonemes*—a set of basic units of words, which are usually individual speech sounds to pronounce each word.

The commercially available lexicons usually include tens of thousands of words. The length of the word sequence uttered by the speaker is not necessarily be known, for the same word by different speakers, as well as by the same speaker at two dif-

ferent times. Consider that length of the word sequence is $N$. If $V$ is taken as the size of the lexicon, the ASR system can hypothesize $V^N$ possible word sequences. The language constraints dictate that these word sequences are not equally likely to occur. For example, the word sequence "please call me" is more likely to occur than the sequence "please me call." In addition, the acoustic feature vectors extracted from the speech signal can provide important clues about the phonemes which produced them. The sequence of phonemes that corresponds to the acoustics observations, can imply the word sequences that could have produced the sequence of these sounds. Hence, the acoustic observations experienced provide an important source of information that can help further narrow down the space of possible word sequences. The ASR systems use the acoustic observations information to compute the probability that these observed acoustic feature vectors have been produced when the speaker uttered a particular word sequence. Essentially, the system efficiently computes these probabilities and outputs the most probable sequence of words as a *decoded hypothesis*.

The most successful speech recognition systems of today, use a *generative probabilistic model*, shown as Eq. 20.1. The speech recognizer tries to find the probability of word sequence $\hat{w}_1^N$ (of $N$ words) that maximizes the word sequence's probability, having given some observed acoustic sequence $y_1^T$. This approach makes use of Bayes' theorem to compute the conditional probability of $p(w_1^N)$ given $y_1^T$. When Bayes equation is expanded in second line of equation (20.1), it ignores the denominator term $(p(y_1^T))$, common for all possible word sequences. This equation maximizes the product of two terms: the probability of the acoustic observations given the word sequence $(p(y_1^T \mid w_1^N))$ and the probability of the word sequence itself $(p(w_1^N))$.

$$
\begin{aligned}
\hat{w}_1^N &= \underbrace{argmax}_{w_1^N}\ p(w_1^N \mid y_1^T) \\
&= \underbrace{argmax}_{w_1^N}\ p(y_1^T \mid w_1^N) p(w_1^N).
\end{aligned}
\tag{20.1}
$$

Figure 20.3 shows the process described by Eq. 20.1 as a block diagram. The lexicon, language model, and acoustic model components construct hypotheses for
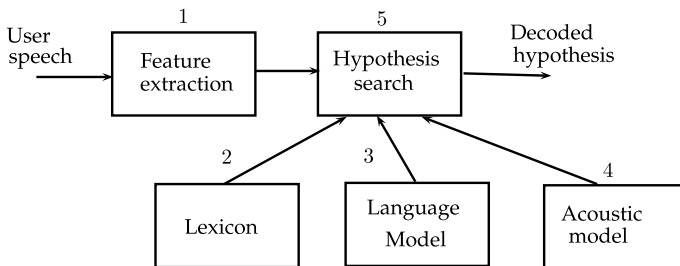


**Fig. 20.3** Speech recognition system block diagram

interpreting a speech sample. Block 1, extracts multidimensional features from the sampled speech signal. In Block 5, hypothesis search is carried out, where the search hypothesizes a probable word sequence based on the observation of features, as well the input from three models—lexicon, language, and acoustic. The other components drive the hypothesis search as follows:

- Lexicon in Block 2 defines the possible words that the search can hypothesize, where each word is a linear sequence of phonemes;
- Language model of Block 3 models the linguistic structure (sequence of words i.e., $p(w_1^N)$), but does not contain any knowledge about the relationship between the feature vectors and the words, and
- The acoustic model in Block 4 models the relationship between the feature vectors and the phonemes ($p(y_1^T|w_1^N)$), which might have produced the sounds.

Getting the best performance for feature extraction and hypothesis searches requires customizing the ASR system for individual speakers. The following section explains the hypothesis in detail.

## 20.5  Hypothesis Search in ASR

Three basic components comprise the hypothesis search: a lexicon, a language model, and an acoustic model. Each one is described in detail in the following [4].

### 20.5.1  Lexicon

A typical lexicon is shown in Table 20.1 with each lexicon's possible pronunciations constructed from phonemes. English language has 44 phonemes. Despite there being just 26 letters in the language, there are 44 unique sounds (phonemes). These sounds are helpful in distinguishing one word or meaning from another. An individual word can have multiple pronunciations, for example, the word "the" has two pronunciations as shown in Table 20.1. These multiple pronunciations complicate the process of recognition. The hypothesis search chooses the lexicon on the basis of task, trading off vocabulary size with word coverage. Although a search can easily find phonetic representations for commonly used words in various sources, task-dependent jargon often requires writing out pronunciations by hand.

### 20.5.2  Language Model

The search for the most likely word sequence corresponding to the speech features sampled, requires the computation of terms, $p(y_1^T|w_1^N)$ and $p(w_1^N)$ in Eq. 20.1. The

**Table 20.1** Typical lexicon

| Lexicon | Phonetic representation |
|---------|------------------------|
| The | dhah |
| The | dhiy |
| Cat | kaet |
| Pig | pihg |
| Two | tuw |

term $p(w_1^N)$ is called the *language model*. The computation requires the assignment of probability to a sequence of words $w_1^N$. A simplest way we can imagine to determine such a probability, is to compute the relative frequencies of different word sequences, like we discussed earlier, that "Please call me" is more probable than "Please me call." Note that, the total number of different sequences can grow exponentially with the length of the sequence, making this approach computationally infeasible. Therefore, there is a need of approximations.

A typical approximation used assumes the probability of current word in the sequence as depending on previous two words only, called 2-grams, against $n$-grams. When this is considered, the computation can approximate the probability of the word sequence as follows:

$$p(w_1^N) \approx p(w_1)p(w_2|w_1) \prod_{i=3}^{i=N} p(w_i|w_{i-1}, w_{i-2}).  \qquad (20.2)$$

The term $p(w_i|w_{i-1}, w_{i-2})$ can be estimated through computation by counting the relative frequencies of word *trigrams*, or triplets:

$$p(w_i|w_{i-1}, w_{i-2}) \approx \frac{N(w_i, w_{i-1}, w_{i-2})}{N(w_{i-1}, w_{i-2})}.  \qquad (20.3)$$

In the above, $N$ is the associated event's relative frequency. Typically, training such a language model requires using hundreds of millions of words to estimate $p(w_i|w_{i-1}, w_{i-2})$ for different word sequences. Even then, many trigrams do not occur in the training text, so the computation must smooth the probability estimates to avoid zeros in the probability assignment.

### 20.5.3   Acoustic Models

An acoustic model computes the probability of feature vector sequences ($y_1^T$) under the assumption that a particular word sequence ($w_1^N$) produced the vectors. In other words, an acoustic model is $P(y_1^T|w_1^N)$.

**Fig. 20.4** Hidden Markov
model for a phoneme
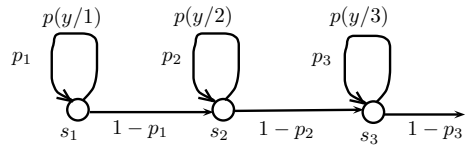


Due to inherently stochastic nature of the speech, a speaker never utters a word exactly the same way twice. The variation in a word's or phoneme's pronunciation manifests itself in two ways: duration and *spectral contents*, also known as *acoustic observations*. In addition, a particular phoneme's spectral content are effected due to phonemes in surrounding context, a phenomenon called *co-articulation* effect. It is, therefore, necessary that acoustic models used should take into account these co-articulation effects. One of the popular acoustic models is based on HMM (Hidden Markov Model).

Hidden Markov Models

A hidden Markov model offers a natural choice for modeling speech's stochastic aspects. HMMs function as probabilistic finite-state machines—the model has a set of states, and its topology specifies the allowed transitions between them. At every time frame, an HMM makes a probabilistic transition from one state to another and emits a feature vector with each transition.

Figure 20.4 represents a *phoneme*'s transitions using a HMM. The transitions in the waveform of a phoneme correspond to state transitions in the HMM. We may think of a HMM as a finite automata with transitions governed by probabilities. Accordingly, we take a set of state transition probabilities in the HMM as, $p_1$, $p_2$, and $p_3$, due to which the possible transitions between the states of the HMM are governed. The probabilities specify the probabilities of going from one state at time $t$ to another state at time $t + 1$. The feature vectors emitted while making a particular transition in the speech waveform, represent the spectral characteristics of the speech at that point. These feature vectors vary corresponding to varying pronunciations of the phoneme. A probability distribution or probability density function can model this variation. In Fig. 20.4, the functions $p(y|1)$, $p(y|2)$, $p(y|3)$, could be different for different transitions. Typically, these distributions are modeled as parametric distributions, which are a mixture of multidimensional Gaussian distributions.

The HMM in Fig. 20.4 has three states, representing the pronunciation of a phoneme starting at state $s_1$. Then, the phoneme corresponds to a sequence of transitions, and terminating at state $s_3$. Duration of a phoneme is equal to the number of time frames required to complete the transition sequence. The transition probabilities $p_1...p_3$ implicitly indicate probability distribution that governs the duration of the phoneme. If any of these transitions exhibits high *self-loop*[1] probabilities, the model spends more time in that state, consequently consuming a longer time

---

[1]for example, the word "speech" can be also pronounced as "spee...ech", repeating the sound of 'e', which creates a self-loop.

to go from the first to the third state. Note that, how time duration a self-loop may repeat, in unknown and varies from speaker to speaker. However, a self-loop (a state) may repeat on itself few times, typically 2-5. However, some words' phoneme(s) may have exceptionally long loop, for example, chanting of *Aum*.[2] The probability density functions associated with these three transitions govern the feature vector output.

A fundamental task required to be performed through an HMM is computation of the likelihood that it produces a given sequence of acoustic feature vectors. For example, assume that the system extracted *T feature vectors* from speech corresponding to the pronunciation of a single phoneme, now the system seeks to infer which phoneme from a set of, say, 50 was spoken, given these feature vectors. The procedure for inferring the phoneme first assumes that the $i$th phoneme was spoken, then finds the likelihood that the HMM for this phoneme produced the observed feature vectors. The system then hypothesizes that the spoken phoneme model is the one which has the highest likelihood of matching the observed sequence of feature vectors.

If the sequence of HMM states is known, we can easily compute the probability of a set of feature vectors. For this, the system computes the likelihood of the $t$th feature vector $y_t$ using the probability density function for the HMM state at time $t$. Having done this, the likelihood that set of all the $T$ feature vectors has occurred is simply the product of all the individual likelihoods $y_t$. Usually, it is not possible to know the actual sequence of state transitions, for the computation of likelihood, all possible state sequences are summed. Given that the HMM dependencies are local, it is possible to derive efficient formulas for performing these calculations recursively.

Parameter Estimation

It is necessary that in advance to using of an HMMs to compute the likelihood values of feature vector sequences, the HMMs needs to be trained to estimate the model's parameters. The training process requires the availability of a large volume of training data in the form of mappings of "spoken word sequences" versus "feature vectors" extracted from the corresponding speech signals. The commonly used process to find a particular correspondence is, estimation of *maximum likelihood* (*ml*) function ($\hat{\theta}_{ml}$). Given that a correct word sequence is known corresponding to the feature vector sequence, the maximum likelihood computation process tries to choose those HMM parameters, which maximize the likelihood of training feature vector. The computation of feature vectors also keeps target for obtaining the correct word sequence. Consider that $y_1^T$ is the representation for stream of $T$ acoustic observations, and let $w_1^N$ represents the correct word sequence; for these, the maximum likelihood function estimate represented by $\hat{\theta}_{ml}$ is,

$$\hat{\theta}_{ml} = \underbrace{argmax}_{\theta} \; \log[p_\theta(y_1^T | w_1^N)]. \tag{20.4}$$

---

[2]chanting of *Aum* is a common practice during meditation and yoga (IPA:/ɐwm/), where sound of IPA ''m' is repeated.

In the beginning, an HMM is constructed for a correct word sequence to start the training process. Then, for each next word, the HMM is constructed by concatenating the HMMs for the phonemes that constitute the next word. The word HMMs are concatenated to construct the HMM for the complete utterance. As an example, the words "we" and "were" have corresponding phonemes as "W IY" and "W ER", respectively. Hence, HMM for the utterance "we were" would consist of the concatenation of HMMs of four phonemes "W IY W ER".

The training phase of an HMM assumes that it is possible to obtain the acoustic observations $y_1^T$ by the system, by traversing HMM from initial state to final state in total $T$ time-frames. However, we know that system cannot trace the actual state sequence, e.g., due to the loops. Therefore, *ml* estimation assumes that this state sequence is hidden, and thus, what is best possible is to average out all the state sequence values. The system can express the maximization of Eq. 20.4 in terms of the HMM's hidden states $s_t$ at time $t$, as follows:

$$\underbrace{argmax}_{\theta} \sum_{t=1}^{T} \sum_{s_t} p_\theta(s_t|y_1^T) \log[p_{\hat{\theta}}(y_t|s_t)]. \qquad (20.5)$$

An iterative process is used to solve Eq. 20.5, with each iteration having two steps: 1. An expectation step, and 2. A maximization step. The expectation step computes the posterior probability $p_\theta(s_t|y_1^T)$, or count of a state. The posterior probability is conditioned on all the acoustic observations. The system makes use of current parameter estimate of HMM, and the computation is performed using forward–backward algorithm. The parameter $\hat{\theta}$ is chosen by the maximization step to maximize Eq. 20.5. For Gaussian nature of the probability function, the computation can derive closed-form expressions for this step [4].

## 20.6   Automatic Speech Recognition Tools

Before we proceed to understand some of the commonly available tools for speech recognition, let us try to understand some of the important terminology of speech recognition.

Automatic speech recognition (ASR) or simply the speech recognition, or computer-based speech recognition, is a process of converting the speech into a sequence of words, using some algorithms which have been implemented as programs. A standard way of doing this is to first split the utterances in the speech waveform with respect to certain parameters of speech recognition. Some of these are: presence of voice activity, duration, pitch, voice quality, voice intensity, S/N (signal to noise) ratio, and the strength of *Lombard effect*.[3] This is followed with

---

[3]*Lombard effect*: Involuntary tendency of speakers to increase their vocal effort particularly when speaking in loud background to enhance the audibility of their voice. Due to the Lombard effect, not

recognition of each utterance. The important factors to be considered during the recognition are detailed in the following paragraphs.

*Concept of Features*

The parameters or values in a complete waveform signal for a given speech is very large. Therefore, to optimize it, a given speech is divided into a large number of *frames*, such that each frame is of small length, typically 10 milliseconds. Each such frame is used to extract 39 different *features*, i.e., 39 different numerical values representing speech optimized to 1 frame. These values of one frame are called *feature vector*. Each such vector that corresponds to a speech segment of ten milliseconds is numerical representation of the speech of that duration.

*Concept of Model*

We need a mathematical model, called *concept model*, that has a representation to gather common attributes of spoken words. Such a model needs to be evaluated for various characteristics, like, how adaptive it is for the changing situations, how well this model fits into practice, and how well it can be configured?

*Concept of Matching Process*

It would require a lot of time for comparing the feature vectors with all the models. Hence, the search should be optimized for choosing the best matching variant.

### 20.6.1 Automatic Speech Recognition Engine

Julius is a high- performance continuous speech recognition software, based on word $N$-grams. It is able to perform recognition at the sentence level with a vocabulary of the order of tens of thousands of words. Julius can realize high-speed speech recognition on ordinary laptop/PC, can perform the speech recognition at near real time, and can achieve typically a recognition rate of greater than 90% using a vocabulary of 20, 000-words, for dictation tasks. Julius is multipurpose, i.e., by recombining the pronunciation dictionary, language, and acoustic models, one can build task-specific systems. Its code is open source, so one can recompile the system for other platforms or alter the code for specific needs [2].

Figure 20.5 shows the structure of *Julius* speech recognition system. The language model of Julius uses $N$-gram mode, and context-dependent HMM (Hidden Markov Model) is used as Acoustic model. As shown in the figure, the input speech is processed through two passes: first pass is 2-gram frame synchronous beam-search (a high-speed approximate search), and second pass is 3-gram $N$-best stack decoding, which is a high precision technique. It can do online recognition using PC/laptop's

---

only the loudness increases, but also the other acoustic features such as pitch, rate, and duration of syllables. The Lombard effect also results in an increase in the signal-to-noise ratio of the speaker's signal.
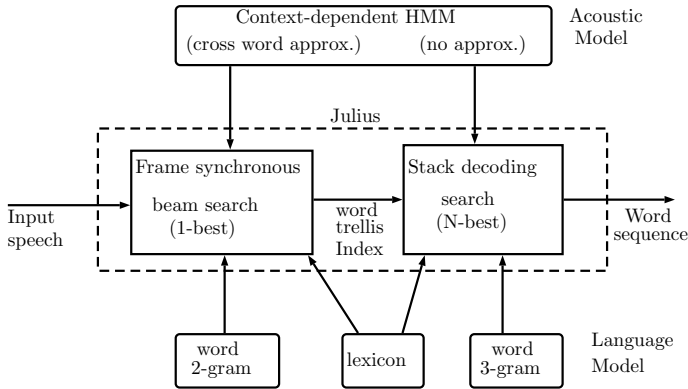
**Fig. 20.5** Julius speech recognition system

microphone or can use any audio device. The first pass of Julius segments input with short pauses, and the second pass sequentially decodes these segments and slots to the results. During the first pass, when a short pause has the maximum likelihood at a certain point of time, a break is placed at that point and second pass is executed on that utterance segment. Due to this process, word constraints are preserved as the context within a utterance segment, and first pass may continue over to the next utterance. Using the above sequence of steps, an input speech file with multiple sentences can be decoded.

### 20.6.2   Tools for ASR

Speech Recognition is available in English, and many other languages, and this feature is common in most smart phones, laptops, and PCs. In the following, we discuss basic tools available as open source.

Kaldi Speech Recognition Toolkit

Kaldi is an open-source speech recognition toolkit, written in C++ language, and works under the Apache platform.

It is a finite-state transducer (FST) based framework, with linear algebra support. Figure 20.6 shows different components of Kaldi: the library modules are grouped together, which depend on two types of libraries, 1. linear algebra libraries (the numerical algebra libraries) and 2. OpenFST (finite-state framework). These two external libraries are also freely available as open source [3, 5].

Access to the library functionalities is provided through command-line tools written in C++. These tools are called from a scripting language, for building and running a speech recognizer. Each tool has a very specific functionality with a small set of command-line arguments: for example, there are separate commands to be executed

**Fig. 20.6** Kaldi speech
recognition toolkit
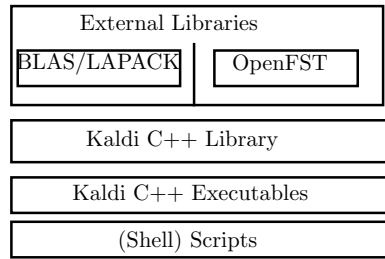


for accumulating statistics, summing accumulators, and updating a GMM-based
(Gaussian Mixture Models) acoustic model. For language modeling (LM), Kaldi
uses FST-based framework, hence, in principle it can use any language model that
can be represented as FST.

CMU-Sphinx

Since its release as an open-source code in 1999, CMU-Sphinx provides a platform
for building speech recognition applications. It is used in desktop control software,
telephony platforms, intelligent houses, computer-assisted language learning tools,
information retrieval, and mobile applications. Traditionally, CMU-Sphinx provides
support for low-resource and underdeveloped languages. It is a speech recognition
toolkit with tools to build speech applications, which makes use of technologies
such as C, cross-platform, HMM (Hidden Markov Models), JavaScript, and Python.
CMU-Sphinx contains a number of packages for different tasks and applications.
The following is the list:

*Sphinx4*—adjustable, modifiable recognizer written in Java, and
*Sphinxtrain*—acoustic model training tools.
*Pocketsphinx*—lightweight recognizer library written in C, and
*Sphinxbase*—support library required by Pocketsphinx.

Deep Speech Tool

It is a simple end-to-end deep learning based speech system, which when combined
with a language model, achieves higher performance than traditional methods on hard
speech recognition tasks. The deep tool is realized by training a large recurrent neural
network (RNN) that uses multiple GPUs and thousands of hours of data. Due to which
the system learns directly from data, and there is no need for specialized components
for *speaker adaption*, like in other systems, neither it needs noise filtering. The more
traditional systems use acoustic models and Hidden Markov Models(HMM) [1].

   The RNN, which is core of this system, is trained to speech spectrograms to gener-
ate English text transcriptions. A training set $\chi = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ...\}$ is used
to sample a single utterance $x$ and a label $y$, where each utterance $x^{(i)}$ is a time series of
length $T^{(i)}$. Here every time slice is a vector of audio features, $x_t^{(i)}, t = 1, ..., T^{(i)}$. The
role of RNN is to convert the input sequence $x$ into a sequence of character probabil-
ities for the transcription $y$, with $\hat{y}_t = p(c_t|x)$, where $c_t \in \{a, b, c, ..., x, space, \prime\}$.

The RNN has five layers of hidden units, such that for an input $x$, the hidden units at layer $l$ are denoted as $h^{(l)}$, with input as $h^{(0)}$. The first three layers are not kept as recurrent layers. At each time $t$, for the first layer, the output depends on the spectrogram frame $x_t$ along with a context of $C$ frames on each side. The remaining non-recurrent layers operate on independent data for each time step. Thus, for each time $t$, the first three layers are computed by

$$h_t^{(l)} = g(W^{(l)} h_t^{(l-1)} + b^{(l)}), \tag{20.6}$$

where $g(z) = min\{max\{0, z\}, 20\}$ is the clipped rectified-linear activation function and $W^{(l)}, b(l)$ are weight matrix and bias parameters for layer $l$.

HTK Tool

The hidden Markov model toolKit (HTK) is a portable toolkit for building and manipulating hidden Markov models. It is primarily used for *speech recognition* research, and *speech synthesis*, as well for character recognition and DNA sequencing. HTK consists of set of library modules and tools available in ANSI C source form. These tools provide sophisticated facilities for speech analysis, HMM training, testing, and result analysis.

The statistical speech models use here the context-dependent hidden Markov models. Probabilities of word sequences is based on N-gram, which finds the most probable word sequence using language model (refer Eqs. 20.1, 20.2) and acoustic model (refer Eq. 20.1).

## 20.7   Summary

Automatic speech recognition (ASR) is another domain of human–machine interface, where machine is to recognize human speech, that is, transform some kind of frequency signal to text. This is a complex process, and requires many processes like phonological, morphological, syntactic, pragmatics, and world. Speech recognition is considered as the future of computer interface. There are basically two application modes for speech recognition: 1. Using speech as input, or 2. As data or knowledge. The application of "speech as input" addresses applications like dictation systems, navigation or transaction systems (like purchasing stocks). Using speech as knowledge has applications like meeting capture.

An application of speech recognition can be implemented using "Voice extensible Markup Language" (http://www.w3.org/Voice/). Developers can use *VoiceXML* to create audio dialogues that feature synthesized speech, recognition of spoken and touch-tone key input, digitized audio, recording of spoken input, telephony, and mixed-initiative conversations. The VoiceXML's architecture uses client–server paradigm to integrate voice services with data services. A voice service is a sequence

of interaction dialogues between a user and an implementation platform, and a document server, which can be external to the implementation platform, and can provide the dialogues.

The modern speech recognition algorithms are based on statistical modeling techniques trained from hours of speech. The process of speech recognition starts with a sampled speech signal, which has a good deal of redundancy due to the physical constraints on the articulators that produce speech. Consequently, the ASR system can compress information by extracting a sequence of acoustic feature vectors from the signal. A system extracts a single multidimensional feature vector every 10 ms that consists of 39 parameters. The system seeks to infer the spoken word sequence that could have produced the observed acoustic sequence, using Bayesian probabilistic approach, which basically, is a process of hypothesis search. Models used for AR are: Language model (searching the most likely word sequence), Acoustic models (compute the probability of feature vector components), and Hidden Markov models (probabilistic finite-state machines).

There are a number of software tools, most as open source, for speech recognition, speech synthesis, as well for research in automatic speech recognition. These software tools take input as sound-wave signal (a file) and split the waveform based on the utterances by the speaker, sample the speech input intervals of about 10 milliseconds, extract the features of input speech. Then using the various models of probability theory, estimate the probable text, which most likely would have produced these utterances. These tools were developed (mostly) as research projects. Among these are: Julius, Kaldi, CMU-Sphinx, Deep Speech tools, and HTK (Hidden Markov Model).

## Exercises

1. Consider alphabet set $\Sigma = \{a, b, c, d\}$. Create finite automata (recognizers) for following strings.

   a. All strings which start with letter $a$.
   b. All strings which end with letter $d$.
   c. All strings where every $c$ is followed letter $d$.
   d. All strings which have odd number of $c$'s.

2. Answer followings in brief, giving suitable examples.

   a. What is the difference between *phoneme* and *morpheme*?
   b. What is the difference between *language* and *dialect*?

3. Write an equation to compute trigram probability.

4. The text processing algorithms are usually written in Python, while the ASR algorithms, which produce the same text, are written in C/C++. Explain what could have been the reason behind this?
5. What is the fundamental difference between the *language* model and *acoustic* model? Why are they the same so?

# References

1. Hannun A et al (2014) Deep Speech: Scaling up end-to-end speech recognition. https://arxiv.org/abs/1412.5567. Accessed Dec 19, 2017
2. http://julius.osdn.jp/book/Julius-3.2-book-e.pdf. Accessed Dec 19, 2017
3. http://kaldi.sf.net/. Accessed Dec 19, 2017
4. Padmanabham M, Picheny M (2002) Large-vocabulary speech recognition algorithms. Computer 4:42–50
5. Provey D (2011) The Kaldi speech recognition toolkit. IEEE workshop on automatic speech recognition and understanding. US IEEE Signal Processing Society, Hawaii
6. Ronald C et al (1997) Survey of the state of art in human language technology. Studies in Natural Language Processing, Cambridge University Press
7. Savitha S, Eric B (2002) Is speech recognition becoming mainstream? Computer 4:38–41
8. http://www.w3.org/Voice/. Accessed Dec 19, 2017

# Chapter 21
# Machine Vision

**Abstract** The goal of computer vision is to extract information from images—a method that can produce a structure from motion, can recover a three-dimensional model of an object from a sequence of views, e.g., grasping by robot, medical imaging, and graphical modeling. This chapter presents the machine vision applications, the basic principle of vision, cognition and classification, and cognitive architecture. The cognition is—going from image-to-scene, which can be achieved through inversion by fixing scene parameters, inversion by restricting the problem domain, and inversion by acquiring additional images. The machine vision techniques are presented here for low-level, middle-level, and high-level vision. The last one requires indexing of images which can be searched through geometric hashing. One of advanced areas of vision—the object tracking, is presented in-depth, which requires the sequences—subtraction of image from background, segmentation of the image, and learning, followed with tracking. Finally, the axioms of vision, tools for computer vision, chapter summary, and practice exercises are presented.

**Keywords** Computer vision goals · 3D models · Vision principles · Vision application · Cognition · Object classification · Cognitive architecture · Machine vision techniques · Low-level vision · Middle-level vision · High-level vision · Image segmentation · Computer vision tools · Vision axioms · Computer vision tools

## 21.1 Introduction

The field of Machine Vision (also called Computer Vision) is growing rapidly. It is concerned with analysis, modification, and understanding of images. The objective of this field is to understand what is happening in front of a camera, and use that understanding to control a computer or robotic system, or making use of these images provide people new images that are more informative or aesthetically more pleasing than the original images. The machine vision has vast applications, some of the important application areas of machine vision are the following: automotive systems, photography, video surveillance, biometrics, movie productions, Web search,

medicines, medical and health sciences, augmented reality, gaming, new user interfaces with computers, and in fact, there cannot be an exhaustive list of all possible areas, as they are continuously increasing in numbers.

As part of advances in machine vision, there are cameras that can focus automatically on our face, and can trigger the shutter when we smile. On the other hand, using an optical text-recognition system one can transform scanned documents into text, which can be analyzed or read aloud by a voice synthesizer.

It is common to have driver-assistance systems in the newly built cars that help the driver to park the car, or give warning signal to the driver in potentially dangerous situations. Intelligent video surveillance systems exist, that can monitor the security of public areas. As new smartphones come progressively equipped with more and more processing power and better resolution of cameras, these phones are becoming fertile field for potential computer vision applications. These devices have built-in capability to merge number of photographs into a high-resolution panorama, they can also read a quick response code, can recognize it and retrieve information about the product from Internet. The mobile computer vision technology is moving fast toward touch interface. However, still the computer vision is computationally expensive to achieve all these in ideal sense.

The computer vision applications have real use only when they can be executed in real time, for which, it is necessary that processing of each frame needs to be completed within $30 - 40$ msec duration. This is a challenging task, particularly when the computation takes place, e.g., in a smartphone or some other embedded computing environment. Generally, it is possible to trade quality for speed or vice versa. For example, in the *panorama stitching* algorithm, if time is provided more liberally, it is possible to find out more matches in source images, and one can synthesize an image of higher quality. To meet the constraints of time and computations to be carried out, one can compromise either on quality or spend longer time optimizing the code for a given hardware architecture.

Further, some of the goals of *computer vision* or *machine vision* can be to extract information from images. For example, an algorithm that produces a *structure* from a *motion*, can recover a 3D model of an object from a sequence of views. As other examples, from a robot grasping an object it is possible to construct a 3D view; construction of a 3D object through medical imaging; and graphical modeling to produce a 3D view. Model-based recognition methods can determine the best matches of stored objects for image data, for use in visual inspection and image database searches. Through visual motion analysis, it is possible to recover image motion patterns for use in vehicle guidance and processing digital video.

The field of computer vision has a close relation to the field of image processing, as follows: In image processing, the focus is on transformation of images, but in computer vision, the focus is on extracting information from image data. To extract a 3D from 2D images is the case of computer vision, and not of an image processing. However, image filtering and color enhancement are the tasks of image processing.

Though many other computer science areas (computation), and board game playing (chess, and tic-tac-toe), have progressed enough and have speed far higher than human, the area of computer vision heavily lags behind compared to human capa-

bilities. The successful example of visual information processing is recognizing of printed text. However, the Optical Character Recognition (OCR) systems make mistakes that even school level students do not make. The problem with *computer vision*[1] systems is their properties of brittleness—a small change in input causes a big change in the output.

**Learning Outcomes of this Chapter**:

1. Summarize the importance of image and object recognition in AI and indicate several significant applications of this technology. [Familiarity]
2. List at least three image-segmentation approaches, such as thresholding, edge-based, and region-based algorithms, along with their defining characteristics, strengths, and weaknesses. [Familiarity]
3. Implement 2D object recognition based on contour-and/or region-based shape representations. [Usage]
4. Provide at least two examples of a transformation of a data source from one sensory domain to another, e.g., data interpreted as single-band 2D image to 3D object. [Familiarity]
5. Describe an algorithm combining features into higher level percepts, e.g., a contour or polygon from visual primitives. [Usage]
6. Describe a classification algorithm that segments input percepts into output categories and quantitatively evaluates the resulting classification. [Usage]
7. Evaluate the performance of the underlying feature extraction, relative to at least one alternative possible approach in its contribution to the classification task. [Assessment]
8. Describe at least three classification approaches, their prerequisites for applicability, their strengths, and their shortcomings. [Familiarity]
9. Tools for Computer Vision. [Usage]

## 21.2 Machine Vision Applications

Originally the computer vision systems were designed with the aim to be used in industrial and military applications. Some of the common military applications of computer vision are: recognition of far-off targets, visual-based guidance systems for autonomous vehicles, and interpretation of images. In the recent times, many computer vision applications have emerged in medical imaging and multimedia systems, like preoperative scans of patients in the operating room, complex surgery using robots, etc. Computer vision techniques are also used for virtual reality applications, and image database retrieval systems. The successful applications of computer vision exists as a consequence to two important effects: 1. A technical force push toward limiting the domain that have sufficient structure or constraint that the many-to-one

---

[1]In AI literature, the words: "computer vision" and "machine vision" are interchangingly used, hence in this chapter also, both the terms mean the same.

inversion problem can be made tractable, and 2. Economic force push toward having a problem of sufficient base to be financially viable. The possible solution is an intersection of these two conditions. The following is a brief account of machine vision applications [4].

*Visual Inspection*

One of the most successful applications of computer vision is in industrial quality control, for example, in automation of visual inspection tasks. This is in replacement for human visual inspection, which is crucial to quality control across a broad spectrum of manufacturing, that ranges from low-volume custom products to high-volume and bulk products.

*Assembly and Material Handling*

In automatic assembly of products, the central driver is the uncertainty in the factory. To solve this problem, robot vision is typically used. The robot vision is based on computer vision algorithms, which have applications in navigation and path control of robot vehicles and robot manipulators. This application is governed by economic drive as well as quality control.

*Automatic Target Recognition (ATR)*

Due to the progress of sophistication of weapon systems, it has become necessary to provide rapid and accurate identification and tracking of targets. The main drive force toward the progress in this direction is accuracy of recognition, with tracking of multiple targets at the same time.

*Photo Interpretation*

This work requires the digitization of image data, and interpret them using vision algorithms.

*Extraction of 3D Structure*

There are many engineering applications, that require construction of complex 3D geometric models and terrains, such applications are: engineering simulation, numerically controlled machines, virtual reality, and advanced cartography. The input to such models are: drawings, intensity images, and 3D sensors like laser triangulation. The algorithms used in these applications derive automatically a 3D slid model of an object that drives a finite element or graphic simulation program.

## 21.3   Basic Principles of Vision

One of the most basic and essential characteristics of living beings is the ability to recognize and identify objects. All the higher animals depend on this ability for their survival. Without this they are unable to function even in a static and unchanging environment. Sight is our most impressive sense. It gives us, without conscious
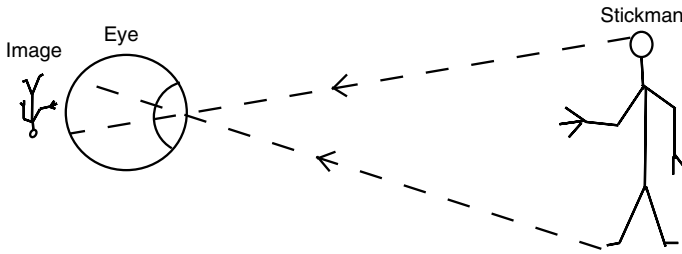
**Fig. 21.1** Image of a "Stickman"

efforts, the detailed information about the shape of the world around us. It is also the challenging area in AI.

The most natural place to begin our study is *transduction* of physical phenomena into internal representation. Vision starts with an eye—a device for capturing and focusing the light that bounces off the objects. Every point on an object (other than a mirror) scatters incident light in all directions. An eye has a lens that directs the light from one point on an object to just one point on retina (see Fig. 21.1). The image on retina is upside down, but human mind corrects it. We note that image is 2D, while the object is 3D [1].

**Definition 21.1** *(Vision Problem)* A vision problem may be stated as, given a 2D image, infer the objects that produced it, as well as infer their shapes, position, sizes, and colors.

However, one imprecision exists in the above definition, it is—how all these characteristics counts for object. For monochrome images, an image may be thought as a function, giving a gray-level at every point on the image plane. The gray-level varies from 0 (maximum black) to 1 (maximum bright). The latter is corresponding to the maximum response the eye can make. We assume that surface on which image is constructed is flat in $x$ and $y$ coordinates, which is appropriate for computer vision also. We approximate the gray-level function by a 2D array *image*, which breaks the image into squares, and records average gray-level over each square, with a pixel at a point, say $(i, j)$, is cell $image(i, j)$. The value of gray-level function at each pixel is in the range [0, 1].

It is challenge to reason back from image to the scene that created that. Figure 21.1 suggests that each spot in image corresponds to one spot in the real world, namely, the piece of surface the light bounced off in order to make the image spot. The recognition of images is a process of pattern recognition. The recognition requires learning, which means mapping the images patterns to some internal representation—this is scene transformation into images. However, recognition of objects is inversion of this process, i.e., reconstruction of scene features from the images, which is a complex task.

Though it is not yet fully established, but it is hypothesized that human follow the following process for identification or classification of objects:

New objects are introduced to a human through activation of sensor stimuli. The sensors, depending on their physical properties, are sensitive in varying degrees to certain attributes, serve to characterize the objects, and the sensor output tends to be proportional to the more prominent attributes. Having perceived a new object, a cognitive model is formed from the stimuli patterns and stored in the memory. Recurrent experiences in perceiving the same object or similar objects strengthen and refine the similarity patterns. Repeated perceptions result in the generalized models of objects classes which become later useful in matching, and hence recognition of similar objects.

The *recognition* is, in fact, a process of establishing a close match between some new stimulus and the previously stored stimulus pattern. Object recognition is the task of finding and labeling parts of a 2D image of scene that corresponds to objects in the scene. For example, from an aerial photo, to identify the various objects, like building, roads, forests, etc. It is a task as it might be carried out by a human observer with marking pen, to mark and label the objects.

Object *classification* is closely related to recognition. The ability to classify various objects or group of objects accordingly to some commonly shared features is a form of class recognition. Classification is essential for decision-making, learning, and many other cognitive acts. Like recognition, classification depends on the ability to discover common patterns among objects. This ability must be acquired through some learning process. For the learning to take place, the prominent feature patterns that characterize classes of objects must be discovered, generalized, and stored for subsequent recall and comparison.

To recognize an object, it is required to first establish a general description of each object to be recognized. Most often, a model includes texture, shape, and the context knowledge about the occurrence of such objects in a scene. For instance, a mathematical description of a set of shaded rectangles may be used to generate buildings as objects. A 3D building object could be modeled as a set of rectangular solids. Texture information might include colors or knowledge about the layout of a building's window.

Corresponding to each occurrence or instance of a model in the image, a label is attached, called *model label*, that can be thought of as a tag pinned to an area in the image that we consider showing an instance of the corresponding object model. The "roads" and "buildings/houses" in Fig. 21.2, shown by arrows, are examples of model labels. Note that a model may be $2D$ or $3D$, but the labels always show $2D$ model instances.

There are many important distinctions about the kind of information we are interested within a digital image, corresponding the scene of that image. The most elementary type of information is called *syntactic* information, which is concerned only with pixel values, and their meanings. The other information, called *semantic information*, deals with knowledge and meaning of the information. Hence, a syntactic image operator's role is to blindly apply an algorithm to the pixel values, without
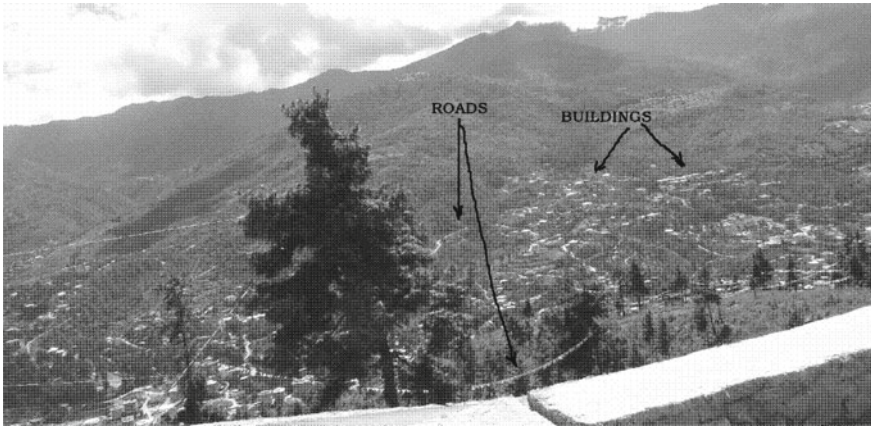
**Fig. 21.2**  An aerial image of a Hilly Town with labels attached to buildings and roads

concerned to the meaning associated with those pixels. An example is, a procedure that assembles group of adjacent pixels which have, say, a high contrast with respect to their neighbors. On the contrary, a semantic operator uses models of the scene and the image production process. They incorporate symbolic knowledge about how the image is organized, such as "a part may be lying on top of one another."

## 21.4  Cognition and Classification

For a process of mechanized recognition requires learning new objects carried out through the following steps:

1. The stimuli produced by objects is perceived by the sensory devices. The more prominent attributes, like size, shape, color, and texture produce the strong stimuli. The values of these attributes and their relations are used to characterize an object in the form of a *pattern vector* as a string. This string is represented as a classification tree, a description graph, or some other form. The range of characteristic attribute value is known as the *measurement space*.
2. A subset of attributes whose values provide cohesive object grouping or clustering, consistent with some goals associated with the object classifications, are selected. The range of the subset of attribute values is *feature space*.
3. Using the selected attribute values, the object or class characterization models are learned by forming generalized prototype description, classification rule, or decision functions. These models are stored for subsequent recognition. The range of decision function values or classification rules is *decision space*.

   Recognition of familiar objects is achieved through application of the rules learned by step 3 above, by comparison and matching of object features with the stored
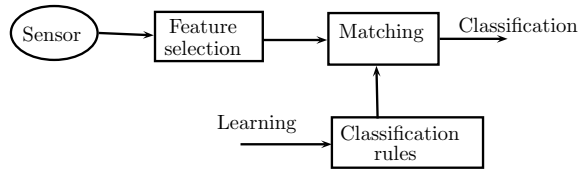
models. Refinements and adjustments can be performed continually thereafter to improve the quality and speed of recognition. These process steps are shown in Fig. 21.3.

Conceptual View of Cognitive Architecture

For an artificial system it is a primary requirement that it should build a rich internal representation of the external environment. This internal representation should be such that it is supportive to draw the inferences, make the decisions, and allows to perform the reasoning process in general related to its own tasks. One approach for representation is *classical logic*, where symbols are given the meaning by relating them to abstract entities as per semantics of model-theoretic approach. However, this approach turns out to be incomplete for the requirement of machine vision, as it requires to find out two things, 1. The meanings of its symbols within the internal representation, and 2. Its interaction with the external world.

A machine vision system requires a cognitive architecture with effective internal representation of the environment. This environment is built through the processes that are defined over suitable intermediate level. The processes act as intermediary between sensory data input and the internal symbolic level representation. One approach to model the visual perception through a process, such that the representation of *information* and *knowledge*, as well as the processing of both, take place at different levels of abstractions. The two levels of abstraction are: 1. The lowest level, which is directly related to features of stimuli and 2. The highest level, where knowledge is in symbolic form, that is concerned with the perceived object. A general assumption of computer vision is that a vision process concludes with 3D reconstruction of shapes using some *geometric primitives*.

For the designs related to cognitive architectures, there are three cognitive representation levels, are as follows [2]:

- *Sub-symbolic level*. The information at this level is related only to sensory data and to nothing else.
- *Linguistic level*. This level uses the information represented using symbolic forms.
- *Intermediate level*. It is also called prelinguistic conceptual level. The characteristic of information at this level (in terms of metric spaces) is defined by a number of cognitive dimensions, that is language independent.

The *intermediate* level representation is useful for generating the essential representation of external environment of a cognitive agent. In addition, the intermediate level provides a precise interpretation of the linguistic level, where interpretation of

conceptual categories is concerned with some standard problems. One such problem is that, perceptual commonsense concepts never correspond to any clear classic category which can be described in terms of necessary and sufficient conditions. For example, the membership in perceptive concepts categories is never in 0 and 1 classes, but usually there is need to consider a prototype of the category.

The information available in a cognitive system strictly depend on data acquired through measurement process. Hence, the knowledge derived through this information at conceptual level will also be affected due to measurement errors. To solve this problem, a model mapping is carried out between conceptual and linguistic levels using a *connectionist*[2] device. The use of Neural Networks (NN) eliminate the need for exhaustive description of conceptual categories at the symbolic level. The latter becomes possible because, a prototype is built based on associative mapping taking place during the training phase of NNs. A measure of similarity between the prototype and the given object is implicit in the behavior of the NN, which is determined during the learning phase of this network.

## 21.5  From Image-to-Scene

For a creature, of type either *biological* or *mechanical* (in case of latter, e.g., robot), for effective interaction with its environment, it needs to know *what* are the objects, and *where* they exists? The computer vision provides the basic methods to understand as how to make intelligent decisions about the environment of interactions, on the basis of sensory inputs it receives. To support the intelligent interaction with the environment, the vision system must know the information about objects in the world (with which it interacts). This knowledge about the objects and their positions become known to the vision system only on the basis of the measurements of inputs received in the form of reflected brightness. To convert this brightness into the world of measurements, first it is necessary to know how the objects are mapped into the image brightness. The magnitude of light recorded by an individual sensor is the result complex interaction of the following parameters:

- Orientation and position of the object, as well as the position and orientation of the light sources with respect to the sensor,
- $3D$ shape of the object,
- The reflective properties of the object's surface (i.e., the rules of physics that govern the reflection of light rays from surface of the object),
- Spectral sensitivity and quantum efficiency of the sensor, and
- Spectral properties of the light sources.

For generating an image from a given set of scene parameters (i.e., mapping from scene to image) is a well-defined process. However, to invert an image to compute the scene parameters that gave rise to the image is an ill-posed and challenging

---

[2]Labeling of unsegmented sequence of data with recurrent neural networks.

problem. This is because, it requires an inversion of a single number to deduce many parameters. If an image is treated as a set of independent brightness values, there can be an infinite number of scenes that could have produced this image ! Hence, deciding the precise scene that might have given rise to so and so image is a challenging task. However, as human beings (and other living creatures), we generally do not have any trouble in concluding the true original scene by interpreting any given image. Thus, the image brightness is generally not independent, and goal of computer vision is to find out the sufficient additional constraints to invert the brightness into scene parameters. In the following discussions, we present major approaches to object recognition through the inversions.

### 21.5.1   Inversion by Fixing Scene Parameters

One approach to perform inversion from image-to-scene that created the image is through fixing some scene parameters. For example, if we have information about the *surface reflectivity*[3] of an isolated object, and about the position of light source, there is a nonlinear equation available that directly relates image brightness at a point to the object's local surface orientation at that point. Since there are two unknown variables for each measured brightness, we need more constraints, to determine the scene.

   One method, called *shape-from-shading*, assumes that the surface that is locally smooth is sufficient to provide solution for the object's shape. The other methods are called *photometric stereo methods*. Their working is based on taking several images with a single position of camera, but with different light sources, and a set of brightnesses are used corresponding to a single point, to determine the local surface shape.

### 21.5.2   Inversion by Restricting the Problem Domain

There is a class of methods for going from image-to-scene, i.e., image inversion, that is based on restricting the problem domain. Consider the case of printed character recognition, like Optical Character Recognition (OCR), where domain consists of $2D$ objects, with known shapes and restricted range of orientation and positions. The inversion problem, in this case, is basically separating individual objects (letters, digits, and punctuation marks) in the presence of noisy brightnesses, and then matching the shapes of those objects against the canonical models. There are in fact many successful commercial tools available for recognizing the printed characters, but the problem becomes more complex if we allow in the range of characters, all the cursive scripts with all their possible variants!

---

[3]The fraction of radiant energy that is reflected from a surface, also called coefficient of reflection.

Consider the problem of identification and locating planar objects in cluttered scenes. The image inversion in this case typically requires finding the invariants in the image forming process. That is, to infer a sudden change in a scene parameter, like edge of an object, based on the observation of a sudden change in a recorded brightness in the image. The features like this have the advantage of being insensitive to variations in light sources and camera positions, hence they are more reliable indicators of scene features.

After having extracted the hypothesized instances of the object features, the next step to recover the scene is to match the geometric shapes possessing such features against the known models. However, in the following, we present an inversion technique that is based on acquiring additional images.

### 21.5.3   *Inversion by Acquiring Additional Images*

Yet another method for image inversion states that, it is possible to achieve image inversion by acquiring additional images. For example, using *stereo vision*, which obtains two views of a scene, it is possible to extract $3D$ shapes of objects. The principle of stereo vision is as follows: if one can determine a point in two images that are projections of a single point in the original scene, and if relative orientation of two cameras is known, then the method of triangulation can produce a $3D$ reconstruction of the shapes of the objects that are existing in the real world. A similar principle holds for motion sequence of images, where either the camera or the objects in the real world move between the images. This process again involves a matching problem, this time there is matching between features from two images, instead of an image and a model. Most of the machine vision algorithms depend on matching schemes that measure the similarity between features in the two images, and make use of multi-resolution methods to control the complexity of such matchings.

Once the $3D$ shape of the scene is extracted, recognition of the scene is made possible by matching $3D$ shape descriptions, while the navigation in the $3D$ world is supported by identifying the collision-free paths through the reconstruction world.

A large variety of matching schemes exists, that also includes the methods that directly search the space, of all possible pairings of models and image features, called, *correspondence space*. There are other methods, that directly search the space of all possible poses of the objects, and there are hybrid methods that can combine both the type of spaces. Most of the methods rely on geometric constraints to reduce the complexity of search, the constraints that typically encode information about objects' shape. The constraints are also used to measure the feasibility of matching the data features to model features, such that they are consistent with the legal transformations of the object. The following are typically the difficult parts of this problem: 1. efficiently deciding about the image features that belong to a single object in the presence of scene clutter and occlusion, and 2. effectively match the partial description of the object in the presence of uncertainty in sensor input.

From the above discussions, we find it common that, vision methods attempt to extract scene parameters, like surface material type and object shape given the observed brightnesses, and then to use these extracted parameters to match against known-object models to support the recognition.

## 21.6 Machine Vision Techniques

The machine vision systems operates on *digital images*, i.e., quantized collection of discrete values in 2D space with varying intensity. The methods used in machine vision are also classified as low-level vision, middle-level vision, and high-level visions. Though, this is not the only classification for vision systems, but it provides a useful way of classifying the computer vision problems [9].

The *Low-level vision* techniques operate directly on images and produce the output images in the same coordinate system as that of the input. For example, an *edge detection algorithm* may take an image with different intensity values as input, and produce a binary output that indicates the edges in the picture.

The *Middle-level vision* techniques take an image or the output produced by low-level vision algorithm as input, and may produce the output something other than pixels of the image coordinate system. For example, the stereo-vision system may produce an output shape in $3D$ using input as two $2D$ images. As another example, a *structure-from-motion* algorithm takes as input a set of image features and produce at output $3D$ coordinate of these features.

The *high-level vision* techniques take as input the results of low- or middle-level vision algorithms and produce in the output some abstract data structures. For example, a *model-based recognition* system may take a set of image features as input, and provide at output the geometric transformations that map the models in its database to the respective locations in the image.

### 21.6.1 Low-Level Vision

The low-level vision computations operate directly on images and produce pixel-based outputs, that remains within the original image coordinates only. The examples of these computations are: finding the intensity edges in an image, smoothing images using filters, computing visual motion fields, and analyzing color information in images [9].

In the following, we discuss edge detection and smoothing of image, in more detail.

## 21.6.2  *Local Edge Detection*

The edge detection of a real-world object is carried out to find out the geometry of the image and to be used in higher level image processing. There are some physical events that cause intensity changes, or appearance of an edge in the images. For example, the object boundaries produce intensity changes which are caused due to discontinuity in the depth or difference in surface color, or difference in texture. The surface boundaries produce intensity changes due to a difference in the surface orientation. However, there are some phenomena of intensity changes that do not reflect directly on the geometry of the object, the examples are shadows and interreflections [9].

As an example, we shall represent gray-level image by $A(x, y)$, as an intensity of a function of the image coordinate system $(x, y)$. The intensity of edges in the image correspond to sudden changes in the value of function $A(x, y)$. For this image, we compute the *gradient magnitude* using the expression,

$$\|\nabla A\|^2 = \left(\frac{\partial A}{\partial x}\right)^2 + \left(\frac{\partial A}{\partial y}\right)^2. \tag{21.1}$$

In simple words, where the squared gradient magnitude (i.e., $\|\nabla A\|^2$) is large, it is an indication of an edge. The Laplacian operator ($\nabla^2$) is another local differential operator that is used for edge detection, expressed by

$$\nabla^2 A = \frac{\partial^2 A}{\partial x^2} + \frac{\partial^2 A}{\partial y^2} \tag{21.2}$$

This second-derivative operator retains the information as which side of the edge is brighter. The zero crossing of $\nabla^2 A$ indicates that the intensity edges of the image, and the sign on each side of a zero crossing indicates which side is brighter.

The Machine Vision Systems

The images input to machine vision systems are digitized with respect to both the space and intensity, and are available as an array $A[x, y]$ of discrete intensity values. The approximations based on finite difference are used to estimate the derivatives for computing the local differential operators. A discrete 1D sampled function is represented as a vector $\mathbf{f}[i]$. The first-order derivative of this vector is $d\mathbf{f}/dx$, which can be approximated as $\mathbf{f}[i + 1] - \mathbf{f}[i]$. And, the second-order derivative is computed as an approximation $\mathbf{f}[i - 1] - 2\mathbf{f}[i] + \mathbf{f}[i + 1]$. Hence, the gradient magnitude can be approximated as

$$\left(\frac{\partial A}{\partial x}\right)^2 + \left(\frac{\partial A}{\partial y}\right)^2 \approx (A[j + 1, k + 1] - A[j, k])^2 + (A[j, k + 1] - A[j + 1, k])^2. \tag{21.3}$$

Image Smoothing and Filtering

Filtering of images in computer vision is carried out using *convolution* operation. Consider a function $h(x, y)$ defined in terms of $f(x, y)$, and $g(x, y)$, as

$$h(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x - \xi, y - \eta) \ g(\xi, \eta) d_\xi d_\eta \qquad (21.4)$$

In the above, $h$ is called the convolution of $f$ and $g$, and expressed as $h = f \otimes g$, and the value of $h$ at any given point $(x, y)$ depend on the values of $f$ and $g$ at all points. However, this complex looking computation can be simplified, as convolution is *commutative* and *associative*, hence the computations can be rearranged in any order it is convenient, to compute them efficiently.

To carry out the discrete approximation for a convolution, the sum of products can be represented as four nested loops over two arrays representing the sampled functions $f$ and $g$. Let the array $g[i, j]$ be $m \times m$, and the array $f[x, y]$ be $n \times n$, for $n > m$. The initial values of indexes are 0. Algorithm 21.1 computes the discrete convolution of sampled functions $f$ and $g$.

The role of convolution is to smooth an image (i.e., to act as low-pass filter), to handle the problem of high-frequency variations. The latter is indicated by sudden change of intensity from a pixel to next pixel. Gaussian method is used for representation here. The Gaussian function $G_\sigma$ in 1D is expressed by

---

**Algorithm 21.1** Compute discrete convolution of $f$ and $g$

1: **for** $x \leftarrow xmin \ to \ xmax$ **do**
2:    **for** $y \leftarrow ymin \ to \ ymax$ **do**
3:       Let $s = 0$
4:       **for** $i \leftarrow 0 \ to \ m - 1$ **do**
5:          **for** $j \leftarrow 0 \ to \ m - 1$ **do**
6:             $s \leftarrow s + g[i, j] \ f[x - \lfloor m/2 \rfloor + i, y - \lfloor m/2 \rfloor + j]$
7:          **end for**
8:       **end for**
9:       $h[x, y] = s$
10:    **end for**
11: **end for**

---

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-x^2}{2\sigma^2}}, \qquad (21.5)$$

which is a canonical bell-shaped distribution, also called *normal distribution*. The maximum value of this function is obtained at $x = 0$, and the area of the function under the curve is 1. The parameter $\sigma$ controls the width of function $G_\sigma$, the larger the $\sigma$, the wider the bell. The Gaussian function in 2D $(x, y)$ is given by the following function.

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma} e^{\frac{-(x^2+y^2)}{2\sigma^2}}. \qquad (21.6)$$

In discrete-based system, the values of integral steps over some range are used as approximations, which are generally at $\pm 4\sigma$ steps. These values are normalized so that they sum to 1, just like in the case of continuous values, with integral as 1.

### 21.6.3 Middle-Level Vision

The middle-level vision techniques receives the images or results from the output of low-level vision algorithms, and produce some output other than pixels, within the coordinate systems of the original image. One goal of the middle-level vision is to extract the 3D geometric information from the images, which are in fact of 2D—usually referred to as *shape-from x*, i.e., recovering $3D$ structure from $2D$ images. In fact, presence of a shading in an image reveals information about $3D$ objects. An example is, perceiving the shape of a sphere from its image, which we conclude as a solid rather than a circular disk. This perception is due to the uniform change taking place in brightness away from the light source [9].

Information about 3D shapes of objects is perceived due to the mirror-like reflections from the objects. Yet, one more source of 3D shape information is due to change in location of an object in an image, from one image to another image of the same object's views. The commonly used techniques for extracting image shapes from multiple images are: 1. stereopsis and 2. structure-from-motion. Apart from extraction of shape-from images, another goal of middle-level vision is to extract structural description of images. What relations exist between these structures can be found out using grouping methods, called perceptual grouping. The latter is responsible for recovering non-accidental alignments of image primitives, like colinear line segments or co-circular arcs.

Stereopsis

Computer stereo vision is the extraction of 3D information from digital images. By comparing the information about same object about a scene from two panels. The human beings compare two images, based on their being superimposed in a stereoscopic device, such that the image from the right camera being shown to the observer's right eye and that from the left camera is shown to the left eye. In a basic stereo-vision system, two cameras are maintained for observing a scene. The idea behind this system is that objects closer to camera will appear more displaced between two views than those which are farther away. By comparing the two images, it is possible to find out the relative depth information in the form of a map. This map encodes the difference in horizontal coordinates of corresponding image points.

For any given point in a scene, the amount of this point's motion between two views is called *disparity*. If the camera system is calibrated in world coordinates, then the actual distance to a point can be found out using the magnitude of its disparity. Given any two images of a scene, the disparity between two is often the intensity image having bright points corresponding to larger intensities, i.e., the things that are closer to the cameras.

**Example 21.1** Finding out the dept using stereo vision.

Let a simple pinhole camera model has a focal point (optical center) $o$, such that all the rays of light pass through this, and are projected onto an image plane $A(x, y)$ (see Fig. 21.4).

Note that in case of human eyes the image plane is retina of eye. In case of camera, it is medium (i.e., the $2D$ surface) inside the camera where image is formed. The *optical axis* (parallel to axis $z$ in this figure) of cameras is perpendicular to the image plane $A(x, y)$ and passes through the focal point $o$. The *focal length $f$* (shown as $(o_l, o'_l)$ and $(o_r, o'_r)$) is the distance from the optical center $o$ to the image plane $A(x, y)$.[4] We will consider a simple stereo camera geometry where the optical axes of the two cameras are parallel to one another, and perpendicular to the *baseline $b$*, that connects the two cameras' centers $o_l$ and $o_r$. We assume that the both the cameras have equal focal lengths, and the origin of the world coordinate frame is placed along the baseline, at the point which is in equal distance between the two camera centers (at distance $b/2$ from each $o_l$ and $o_r$).

Let the origin of the coordinate system for the left image plane $L$ be the projection of its optic axis, i.e., $o'_l$. Similarly, the origin of the right image plane $R$ is at $o'_r$. Each of the $L$ and $R$ planes are $(x, y)$ planes.

We consider a point $p = (x, y, z)$ (see Fig. 21.4) in the world coordinate, that is projected onto plane $L$ at location $p'_l = (x'_l, y'_l)$ and onto plane $R$ at location $p'_r = (x'_r, y'_r)$. From the geometry of two cameras, we have

$$\frac{x'_l}{f} = \frac{x + b/2}{z}, \tag{21.7}$$

$$\frac{x'_r}{f} = \frac{x - b/2}{z}, \tag{21.8}$$
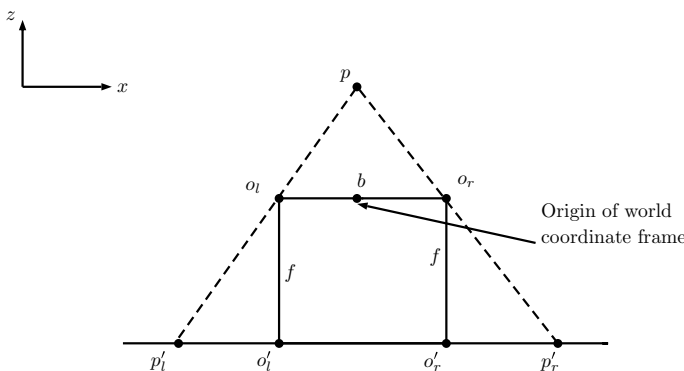


**Fig. 21.4** A simple stereo camera geometry in image plane $A(x, y)$

---

[4]The plane $(x, y)$ is formed by lines $(o'_l, o'_r)$, and axis $y$, the latter is perpendicular to plane of paper.

$$\frac{y_l'}{f} = \frac{y_r'}{f} = \frac{y}{z}. \qquad (21.9)$$

Note that in this simple camera geometry, only the $x$ location ($x_l'$ and $x_r'$) of a projected point differs between the left and right images, and the $y$ location ($y_l'$ and $y_l'$) of a given point in space is the same for both images. The disparity is distance between $p_l'$ and $p_r'$, is just $x_l' - x_r'$. We have from Eqs. 21.7 and 21.8,

$$\frac{x_l' - x_r'}{f} = \frac{b}{z}. \qquad (21.10)$$

Hence, if $b$ and $f$ are known, depth $z$ of the point $p$ can be computed from the disparity, giving the third dimension, hence creating a 3D view using 2D image. If $b$ and $f$ are unknown, the relative depths of points can be determined, and we can grasp the relative $3D$ views of two images. But, in this case, their absolute distance from the camera cannot be determined.  $\square$

### *21.6.4  High-Level Vision*

The High-level vision systems are used to make abstract decisions, or based on the visual data they perform classification of objects. These methods usually make use of the outputs produced by low-level or middle-level vision algorithms discussed above. The high-level vision systems have main applications as object recognition systems, or as object tracking systems.

- *Object recognition systems.* These systems are used to find out whether or not a particular objects is present in the scene, and also determine the locations of other objects with respect to camera;
- *Object tracking systems.* As the name implies, the object tracking systems follow a moving object in a video sequence, and hence can guide a mobile robot or an autonomous vehicle.

In the following, we will discuss the principles used in the first application.

Object Recognition Systems

The Object recognition systems compare new image to stored object models to determine whether any of the models is present in the image. Many object recognition systems perform both the recognition and localization tasks, i.e., identifying what is the object present in the image, and also recovering its location in the image or world coordinate systems. The location of an object is called its *pose*; it is generally specified using a transformation that maps the model coordinate system to the image or world coordinate system.

The object recognition systems can be classified based on the type of problems they solve. A simple recognition task requires identification of 2D objects that are

fully un-occluded (all are fully visible), they appear in a uniform background, and the lighting conditions are controlled, i.e., there are no reflections or shadows in the view. The large majority of industrial inspections fall in this category of object recognition, and can be handled accurately using the commercially available systems.

The object recognition problems are difficult, in the following conditions:

Background of the objects is highly textured,
Lighting conditions are unknown and uncontrolled,
Scene comprises too many objects,
Number of objects models are very large in the storage, and
Some objects may be touching and occluding other objects.

In our discussions, we will consider methods to handle images with multiple objects, the objects are partially occluded, and there is some amount of cutter in the background. The methods that can solve the object recognition with the presence of these challenges, first extract geometric information, like intensity edges from an image. Then, the next step is to compare 2D geometry with 3D geometry.

An alternative method is computing invariant representations of the image geometry, i.e., it remains unchanged in spite of changes in the viewpoints. Such representations can be used to create an index of object models in a library of collections of object models. One of the challenges in geometric recognition is to find out which portion of the image corresponds to a given object from the library collections. The *recognition problem* is often defined as recovering a correspondence between local features of an image and an object model. There are three classes of approaches as how to search possible match between a model and image feature.

1. *Transformation space methods.* These methods use the space of possible transformations that map the model to the image.
2. *Correspondence methods.* These methods use the space of possible corresponding features to match with the image.
3. *Hypothesize and test.* The hypothesize and test methods consider $k$-tuples of model and data features for matching with the image.

We will be discussing the correspondence-based methods in more detail.

Correspondence Search

It is a tree search to find out which model's features match with the image features. Considering a set of *image features* as $A = \{a_1, \ldots, a_n\}$, and a set of *model features* as $M = \{m_1, ..., m_r\}$, an interpretation of the image can be expressed as a set of pairs, $N = \{(m_i, a_j), ...\}$. The interpretations states, which model features correspond to the image features.

If the model features are unclear (hidden or occluded), and some features are outside the image features, in that case the set $N$ is any subset of all "pairs of model and image features." A search method used, called *interpretation tree approach* makes use of pruned search in otherwise exponential space of possible interpretations. In these interpretations, the method makes use of all possible ways of pairings of

models and image features. In simple terms, the search method makes use of pairwise relations between features to prune a tree of possible model and image feature pairs. A traversal path in this tree corresponds to a sequence of interpretations.

To limit the number of pairs $(m_i, a_j)$ and $(m_p, a_q)$ in the interpretations, the distances $\|m_p - m_i\|$ and $\|a_q - a_j\|$ must be equal, i.e., are within the allowable error limits.

The search for interpretations of an image, a pruned-tree search algorithm is used, which works as follows:

- Each level of the tree, other than root node, corresponds to an image.
- Each branch at a node corresponds to a *model feature* or there is special branch called *null face*.
- Accordingly, each node of the tree specifies a pair: ⟨image feature at that level, model feature taken from the previous level⟩. This model feature may be null also.
- The three search carried out is Depth-First search (DFS). Any given node is expanded only when it is pairwise consistent with all the nodes along the path from the current node back to the root of the tree. That is, a given node is paired with each node along the path back to the root, and for each such pair, if the distance constraint is satisfied, the node is expanded otherwise not.

In the above, the null face branch is considered as always consistent.

A path from root to leaf node of the search tree indicates that zero or more model features exists. However, a null branch path does not account for a model feature. A path that accounts for $k$ model features is called *k-interpretation*. Consider that a reference level to filter out any hypothesis that does not account for enough model features, is set at a threshold, say $k_0$. In this case, a matcher algorithm reports only those $k$ interpretations whose threshold is $k > k_0$. All these $k$-interpretations are guaranteed to be pairwise consistent. Hence, an additional step, to verify the model is carried out. This step estimates the best transformation for each $k$-interpretation and checks that this transformation brings each mode feature within some error range of each corresponding image feature.

## 21.7  Indexing and Geometric Hashing

The significance of indexing in vision is to match the images efficiently with the models stored in the repository. Having this facility, it is possible, in principle, to search an object in a library of stored models, with time complexity, that is independent of number of models stored. To carry out the indexing, it requires finding the *geometric invariants* in model recognition—those attributes of images that do not change due to changes of viewpoints of objects. One key difference of this model indexing with hashing is that, various instances of the same object in different images will not generate exactly the same key due to sensing uncertainty.

There are number of methods to use invariants in object recognition. These are categorized based on: geometric differential, photometric, and thermal properties

of images. The indexing methods for structures make use of combinations of simple features such as points and segments. Other geometric techniques use invariant properties of curves and co-planar conics. The advantage of using curve features is lower combinatorial complexity, but it has disadvantage of requiring to extract features from noisy and cluttered images. However, the intensity information from images provides their richer description than the geometric features, like points and line segments.

The *geometric hashing* is used to explain the invariants in recognition. For this 2D *affine transformation* is used, with geometric hashing using three points to define a coordinate system, and with respect to these points, the other points are encoded in invariant manner. The geometric hashing comprises the following two basic steps:

a. Construction of a model hash table, and
b. Matching of model to image.

The hash table stores redundant and the representation of each object that are transformation-invariant. For any given model, each ordered triple, $m_1, m_2, m_3$ forms an affine basis having origin as $o = m_1$, and axes $x = m_2 - m_1$, $y = m_3 - m_1$. For each such basis every additional model point $m_i$ is expressed as $(\alpha_i, \beta_i)$, such that $m_i - o = \alpha_i x + \beta_i y$. The basis triple $(o, x, y)$ and the point $m_i$ are then stored in a hash table using affine invariant indices $(\alpha_i, \beta_i)$. For $r$ number of model points, a table results with $O(r^4)$ entries, i.e., space complexity is combinatorial. The table is formed using buckets instead of purely a hashing scheme. This is because, due to the uncertainty in sensing in real-life data makes it difficult to use exact values for retrieval purpose.

At the time of recognition, the hash table is searched to find out what models are present in the image. The algorithm steps for recognition of an object are as follows.

i First, the image points are rewritten in terms of an *image basis*.
ii As the next step, corresponding to an instance of a model that model basis will be retrieved from the table.
iii An image basis is formed using each ordered triple of image points $s_1, s_2, s_3$, such that origin is $O = s_1$, and axes are $X = s_2 - s_1$, $Y = s_3 - s_1$.
iv For this image basis, each additional image point $s_i$ is written as $(\alpha_i', \beta_i')$, such that $s_i - O = \alpha_i' X + \beta_i' Y$.
v The indices $(\alpha_i', \beta_i')$ are used for retrieving matching model basis from the hash table. Along with retrieving each model basis, corresponding counter is incremented in a histogram.
vi When all the retrieved points for the model basis have been considered for an image basis, the histogram contains the votes for those model bases which could match to the current image basis $(O, X, Y)$.
vii If peak in the histogram for a given model basis $(o, x, y)$ is above a threshold, the corresponding basis is selected as a potential match. When a next image basis is chosen, the histogram counts reset.

## 21.8   Object Representation and Tracking

The availability of inexpensive high-quality video cameras and the increasing demand for automated video analysis, has resulted to lot of interest for *object tracking*. The video analysis comprises three key steps: 1. detection of moving objects, 2. tracking of these objects from one image frame to another frame, and 3. analysis of the object tracks to recognize the object behavior, and sometimes its geometry also.

The goal of object tracking is to generate *trajectory* of an object by locating its positions in every frame of the video, over a time. Object tracking also provides information about the region occupied by the object at every time instant in the image. The two tasks, i.e., detecting the object, and establishing correspondence between the object instances from frame to frame, can either be performed separately or jointly. When only one job is required to be performed, i.e., to detect an object, the possible object regions in every frame are obtained using an algorithm for object detection. Then, as next step, the tracker can find the correspondence in frame sequences.

In the second approach, when detection and tracking are done together, the object regions and the correspondence is jointly estimated through iteratively updating the object location and region information. The information about location and region are obtained from the previous frames. What type of model is selected to represent an object's shape can decide the type of motion or deformation it can undergo. For instance, if an object is represented as a point, then, only a translational model can be used for this object [10].

In situations where a geometric shape representation, e.g., an ellipse is used for the object, parametric motion models like *affine*[5] or transformations like *projective* are preferred. The motion of any rigid object in a scene can be approximated by these representations. For nonrigid objects, *silhouette* or *contour*-based approach is preferred, being the most descriptive form of representation. Both the parametric as well as nonparametric models can be used to specify their motion using these approaches.

It is possible to simplify the object tracking job by restricting the motion of the object or its appearance, or both of these; for example, assuming that object motion is uniform, i.e., constant velocity, or constant acceleration. The objects can be represented in many ways, but some representations are considered as more suitable than others. For instance, fish in an aquarium, boats in the river, vehicles on a road, and birds in the air, are the sets of objects that may be important to track in a specific domain. Depending on the type of objects, they may be represented by their appearances and shapes.

- *Points.* An object is represented by a point as its centroid (see Fig. 21.5a), or can be represented by a set of points, as shown in Fig. 21.5b.

---

[5]Affine transformation is a linear mapping method in geometry, such that it preserves points, straight lines, and planes. For example, after an affine transformation, a set of parallel lines remain parallel, say, a transformation from parallelogram to rectangle and vice versa. Affine transformation is typically used to correct for geometric distortions or deformations that occur with nonideal camera angles.
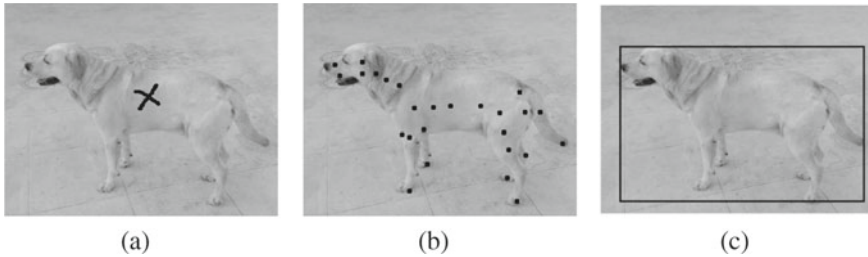
**Fig. 21.5** Object representations-I: **a** Centroid **b** Multiple points **c** Rectangular patch
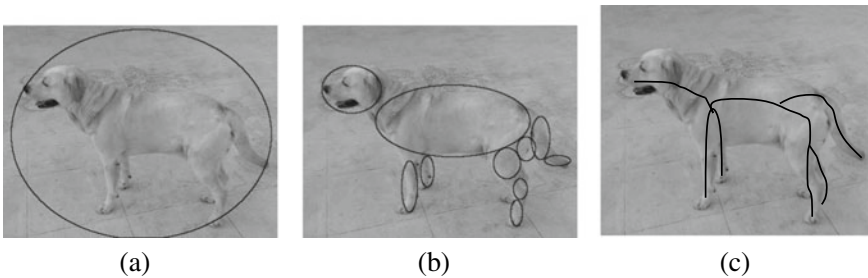


**Fig. 21.6** Object representations-II: **a** Elliptical patch **b** Part-based multiple patches **c** Object skeleton

- *Primitive geometric shapes.* An Object can be represented using a shape, e.g., a rectangle, or an ellipse (see Fig. 21.5c, 21.6a).
- *Articulated shape models.* The articulated objects comprise body parts, which are held together with the help of joints. For example, a pet's body is an articulated object with hands, legs, head, feet, and the torso connected together by joints (see Fig. 21.6b).
- *Skeletal models.* In these models, the object skeleton can be extracted by applying medial axis transform to the object silhouette (see Fig. 21.6c).
- *Object contour.* The contour representation defines the boundary of an object (see Fig. 21.7a, b).
- *Object silhouette.* The region inside the contour is called the *silhouette* of the object (see Fig. 21.7c).

The object tracking is carried out to estimate the trajectory of an object in the plane of the image, as the object moves in the scene. During the tracking, the tracker assigns labels to the tracked object to mark its positions in different frames of a video. Apart from this, depending on what is the domain used, a tracker can also provide object-centric information, such as area, shape of the object, and about orientation of the object. However, the tracking process sometimes turn out to be complex, due to reasons mentioned below.
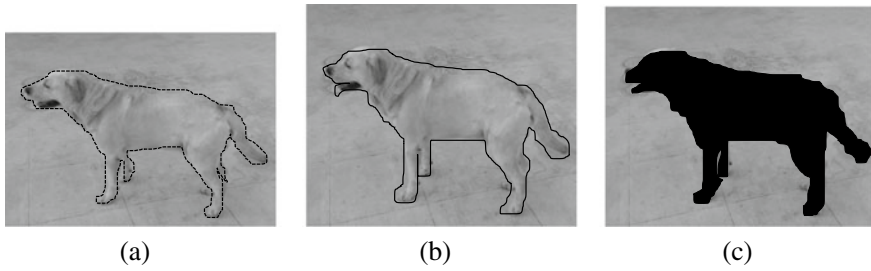
**Fig. 21.7** Object representations-III: **a** complete object contour (dotted lines) **b** control points on object contour **c** object silhouette
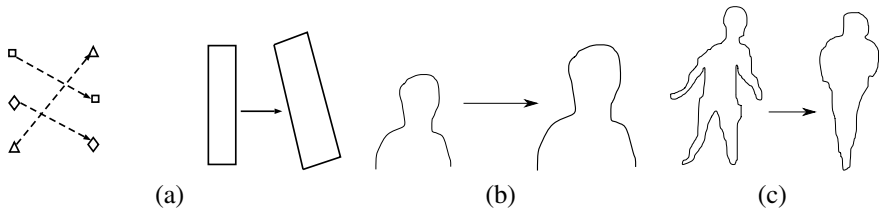


**Fig. 21.8** Object tracking. **a** (i) Multipoint correspondence (ii) Kernel Tracking **b** Evolution of contours **c** Silhouette Tracking and Evolution of contours

Complex object motions,
Articulated or nonrigid nature of objects,
Occlusion of objects in full or partially,
Complex shapes of objects,
Illumination changes in the scene,
Need of real-time processing of the scenes,
Loss of information due to projection of $3D$ world on a $2D$ images, and
Image noise.

The following is brief introduction to the commonly used tracking categories:

Point Tracking

In point tracking, objects detected in consecutive frames are represented by points. The association between consecutive points is established based on the previous state of the object, which may comprise object position and its motion. This approach needs a separate external mechanism to detect the objects in every frame. An example of correspondence between the objects is shown in Fig. 21.8a, with mapping through broken lines.

Kernel Tracking

A kernel is concerned with the object shape and its appearance. For example, a kernel can be a rectangular template or an elliptical shape with an associated histogram (see Fig. 21.8a, with rectangles). The objects are tracked by computing the motion of

the kernel in consecutive frames. The motion is usually in the form of a parametric transformation, like rotation, translation, or affine.

Silhouette Tracking

Object tracking can also be performed by estimating the object's region in each frame. Information encoded inside an object's region is used by the Silhouette tracking, which is either in the form of appearance density or shape models (i.e., in the form of edge maps). Further, given an object model, the silhouettes are tracked by matching the shapes (Fig. 21.8b), or they can be matched by contour evolution (Fig. 21.8c). Both of these methods can be used for object segmentation, which is applied in the temporal domains.

   The following are major applications of object tracking.

- *Video indexing.* It comprises automatic annotation and retrieval of the videos in multimedia databases.
- *Motion-based recognition.* Motion-based recognition has applications in identification of humans based on gait and in automatic object detection.
- *Automated surveillance.* It is concerned with monitoring a scene to detect suspicious activities or unlikely events.
- *Human–computer Interaction.* The HCI is concerned with applications, like gesture recognition and eye-gaze tracking, for data input to computers.
- *Vehicle navigation.* It is used in video-based path planning and obstacle avoidance capabilities.
- *Traffic monitoring.* It is concerned with real-time gathering of traffic statistics to direct the traffic flow.

## 21.9   Feature Selection and Object Detection

The feature selection is a process of deciding those attributes of a picture that help in identifying the object in the picture frame, while object detection or object identification is a process to identify the object in the frame, based on these features. Therefore, it is necessary that the features decided should give a strong evidence of the corresponding object.

Feature Selection

To efficiently distinguish the objects in the feature space, it is desired that the visual features be unique. The selection of feature is related closely with the objects representation. For example, for histogram-based representations, the color is a required feature, and for contour-based representations, object edges are important features. Many times, the features with various combinations are useful. Some examples of features as given below [10].

Color

The color, in appearance of an object, is primarily influenced by two physical features of the object: (1) light source's spectral power distribution, and (2) object's surface reflection properties. The color space used for an object is RGB (red, green, blue). Since, the RGB does not have uniform perceptions, HSV (hue, saturation, value)—more closer to uniform color space—is preferred as a representation to obtain uniform perception.

Edges

Since the object boundaries are the cause of strong changes in image intensities, the edge detection is used to identify these changes. The edges intensities are also less sensitive to the level of illumination. Hence, the algorithms used for tracking boundaries of the objects, make use of edges as the important representative features of the image.

Texture

Texture of an object surface causes an intensity variation in the reflected light from its surface, hence quantifies properties of the object surface such as smoothness and regularity. The texture requires an additional processing step to generate the descriptors.

An object's representative features usually depend on the application domain; however, the automatic feature selections is in common practice. The features are either *filter* based or they are *wrapper* based. The methods that are filter based, make use of general criteria, e.g., the features need to be correlated. On the other hand, the wrapper methods select the features based on their usefulness in the concerned problem domain, for example, classifying the performance using the subset of features. An example of filter method, called Principle Component Analysis (PCA), is useful in reduction of number of features. The PCA is based on transformation of possibly a number of correlated variables into a smaller number of variables that are not correlated, called principal components. The first principal component chosen should account for maximum possible variability of the data, and each succeeding component should account for most of the remaining variability of the components possible. *Adaboost* algorithm is an example of wrapper-based method for selection of discriminatory features for tracking a class of objects. This method identifies a strong classifier using a combination of moderately less accurate and weak classifiers. Having given a large set of features, it is possible to train one such classifier for each feature [3].

Among all the features used for object tracking, color is the one most widely used feature. The colors' histogram can be used to represent the object's appearance. In spite of the popularity of color as an important feature, large majority of color bands are sensitive to variation in illumination. Thus, in situations where this effect cannot be stopped, other features are considered to model the object's appearance.

### 21.9.1    Object Detection

A tracking method always requires a mechanism for object detection, either in every frame or when the object appears in the video for the first time. The commonly used approach for object detection is to use the information from a single frame. However, some method makes use of temporal information computed from a sequence to reduce the false detection of objects. The temporal information is in the form of *frame differencing*, i.e., it highlights the changes in regions from frame to frame. Based on the object regions available in an image, the trackers establish correspondence from one frame to next frame of same object, and generate the tracks [10].

Some commonly used methods for object detection as discussed below.

Point Detectors

The point detectors have application in finding the *interest points* in an image that have expressive texture in their localities. An important characteristic of interest point is its invariance with respect to changes in the illumination and camera viewpoint. A procedure, called *Moravec's* operator-based method, is used for finding the interest points, which computes the variation in image intensity in a $4 \times 4$ patch in horizontal, vertical, diagonal, and anti-diagonal directions, and selects the minimum of the four variations as representative values for the window. A point in this case is declared as interest point if the intensity variation in a local maxima is $12 \times 12$ patch.

Image Background Subtraction

It is possible to build a representation of a scene, called *background model*, and then to find the deviation from the model for the incoming frame, to detect the object. If there is a significant change in the image region from the background model, then it indicates that the object is moving. This approach can also be used to detect if a new object has been introduced into the scene, or some object has escaped from the scene. In this method, the pixels representing the regions that undergone the change are marked for further processing. In such cases, it is common to use connected component algorithm to obtain the required objects in the image. The process used, as it appears, is called *background subtraction*.

Figure 21.9 shows background subtraction results: The part (a) of figure shows an input image of highway with moving vehicles, pedestrians, and nearby houses as objects, the part (b) is reconstructed image after projecting input image onto the eigenspace, and part (c) is an image obtained as difference of two images.

Image Segmentation

Aim of an image-segmentation algorithm is to partition an image into perceptually similar regions. A segmentation comprises solution of two problems: 1. What is criteria for good partition, and 2. What is procedure for achieving efficient partitioning? In the following we discuss some of commonly used segmentation techniques that are useful for object tracking.
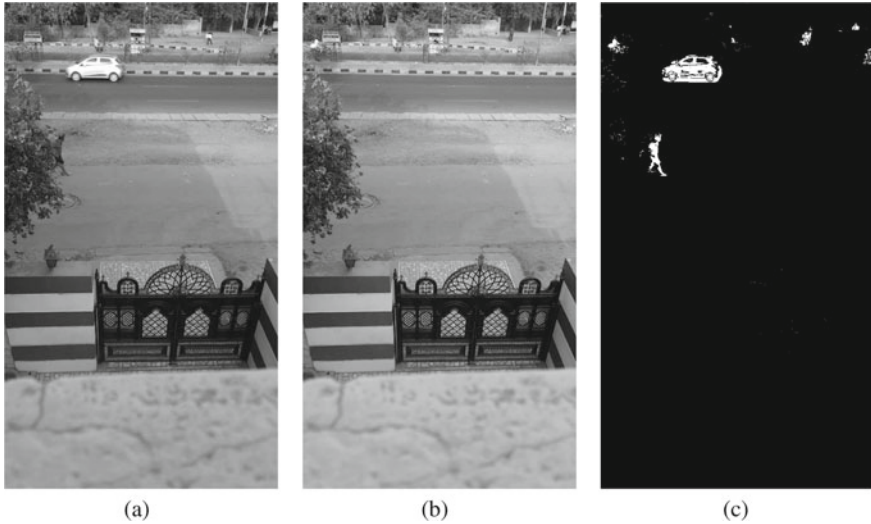
**Fig. 21.9** Background subtraction. **a** input image with objects **b** reconstructed image **c** difference image

*Image segmentation using mean-shift clustering*

This approach for image segmentation finds the clusters in the joint color+spatial space, $[u, v, w, x, y]$, where $[u, v, w]$ represents color, and $[x, y]$ represents the spatial location in $2D$ system. The color may be either RGB or HSV. The algorithm steps are as given below.

i. For any given image, large number of random hypothetical cluster centers are initialized, which are chosen from a given data set;
ii. Each cluster center is moved to the mean of the data lying inside a multidimensional ellipsoid whose center is the cluster center;
iii. Using the old and new cluster centers a vector, called as *mean-shift vector*, is defined;
iv. This mean-shift vector is iteratively computed until the cluster centers are stabilized (i.e., they do not change their positions).

*Image segmentation using graph cuts*

The image-segmentation problem can also be solved like partitioning of a graph. Consider that $\mathbf{V} = \{v_1, v_2, ..., v_n\}$ be vertices (pixels) in a graph (image), and the graph is represented by $G = (\mathbf{V}, \mathbf{E})$. Let us assume that this graph is partitioned into $N$ disjoint subgraphs, with regions $A_1, ..., A_N$, by its weighted edges $\mathbf{E}$. The partitioning is subject to the condition, that

$$\bigcup_{i=1}^{N} A_i = \mathbf{V}, \tag{21.11}$$

and

$$A_i \cap A_j = \phi, \ \text{for} \ i \neq j. \tag{21.12}$$

In any graph, the total weight of the pruned edges between its two subgraphs is called a *cut*. The weights are typically computed based on color of the image, its brightness, and texture similarity between the vertices. In one approach, minimum cut (*mincut*) criteria is used, with the aim to find out the partition that minimizes a cut. The weights in this approach are defined based on color similarity. However, this approach has disadvantage of bias toward *over-segmenting* the image. This over-segmenting is caused due to increase in cost of a cut with number of edges going across the partitioned segments.

The problem of over-segmentation can be solved by using a *normalized cut*, where the cut depends on the sum of edge weights in the cut, like before. In addition, the cut also depends on the ratio of, total connection weight of the vertices in each partition to weight of all vertices of the graph taken together. For the segmentation based on images, weights between the vertices are defined by the product of the *spatial proximity* and *color similarity*. Once the weights are computed between each pair of vertices, a *weight matrix* $\mathbf{W}$ and a *diagonal matrix* $\mathbf{D}$ are computed, such that

$$\mathbf{D}_{i,i} = \sum_{i=1}^{N} \mathbf{W}_{i,j}. \tag{21.13}$$

The segmentation is performed first by computing the eigenvectors and the eigenvalues of the generalized eigensystem, as per the expression,

$$(\mathbf{D} - \mathbf{W})\mathbf{y} = \lambda \mathbf{D}\mathbf{y}. \tag{21.14}$$

Next, the second-smallest eigenvector is used for dividing the image into two segments. For each of the new segment, this process is performed recursively until a threshold is reached.

In a segmentation that is based on normalized cut, solution to the generalized eigensystem for large images can be expensive in terms of memory requirements as well as time requirements. However, this method requires far less number of manually selected parameters, compared to that of *mean-shift* clustering-based method.

## 21.10   Supervised Learning for Object Detection

The *supervised learning* can be used to automatically learn different object views from a set of examples to perform the object detection. Learning of different object views exempts the need of storing the complete set of templates. The supervised learning makes use of a set of examples to generate a function which maps any input to desired output. A standard formulation of supervised learning is in the form

of a classification problem. A learner in the supervised learning approximates the behavior of a function by generating an output either as continuous value or class labels. When output is continuous value, it is called *regression* model, and when the output is class labels, it is called *classification model*. For carrying out the object detection, the learning examples comprise pairs: ⟨object feature, object class⟩, with both quantities manually defined.

Selection of proper features is important for effectiveness of the classification, only those features need to be used that are able to discriminate one class from the other class. Apart from the features, like, color and texture, other features are also possible to use, such as area of the object, its orientation, and appearance. These features can be used in the form of density function, for example, in histogram. Once the features are finalized, all required appearances of an object can be learned through supervised learning only. Some of the learning approaches to supervised learning are decision trees, neural networks, and support vector machines. The learning methods compute a *hypersurface*, which separates one object class from the other in a high dimensional space.

For supervised learning to be useful, it requires a large collection of samples from each object class, where each collection has been manually labeled. Along with the supervised learning, *co-training* is also carried out to reduce the size of manually labeled data. The co-training approach trains two classifiers using a small set of labeled data, such that features used for each classifier are independent. Once training is complete, each classifier assigns unlabeled data to the training set of other classifiers. Using this method, we start with a small set of labeled data with two sets of statistically independent features, and the co-training can provide accurate classification rule. In the following, we discuss two approaches, i.e., *adaptive boosting* and *support vector machines* for object detection.

Adaptive Boosting

This method combines many base and moderately accurate classifiers to produce a more accurate classifier. The steps (of algorithm) of this classifier are as follows.

1. The training phase of the algorithm constructs an initial distribution of weights over a training set;
2. A base classifier having least error is selected by a boosting mechanism. This error is proportional to the weights of the misclassified data;
3. The weights associated with data, that are misclassified by the selected base classifier, are increased;
4. The iteration step is repeated after selecting a new classifier that performs better on the misclassified data, and the process is repeated, until there is no misclassified data.

The simple operators can act as weak classifiers for object detection, e.g., a set of thresholds. These classifiers are applied to the object features extracted from the image. In an approach where adaptive-boosting framework is used, say, for detecting the pedestrians in the presence of other traffic, perceptrons can be chosen to act as the weak classifiers. These perceptrons can be trained on the image features extracted

through a combination of temporal and spatial operators. Operators in the temporal domains will be in the form of frame differencing, and they encode some form of motion information (a required feature of pedestrians). When temporal domain makes use of the frame differencing operator, there is less likelihood of false detection, because it enforces the object detection in the region where the motion occurs.

Support Vector Machines

When used as a classifier, a Support Vector Machine (SVM)[6] is used for classifying the data into two classes by finding a maximum marginal hyperplane which separates one class from the other. The margin of this hyperplane is maximized, as defined by the distance between the hyperplane and its closest data points. The data points that lie on the boundary of the margin of this hyperplane are called *support vectors*, hence the name of this technique.

For detecting an object, these classes, separated by the hyperplane, correspond to the object class (positive samples) and the non-object class (negative samples). Using manually generated training examples, labeled as object and non-object, one hyperplane is computed from among an infinite number of possible hyperplanes using a method called *quadratic programming*.[7]

In spite of being a linear classifier, the SVM can also be used as a nonlinear classifier by application of a technique, called *kernel-trick*,[8] to the feature vector extracted from the input. The kernel-trick is applied to a set of data that is not linearly separable, to transform them to a higher dimensional space so that it becomes separable.

## 21.11   Axioms of Vision

The term axiom (also called *primitives*) is considered from the mathematical usage, using these as the basic building blocks it is possible to derive new constructs. This approach has the advantage of abstracting the representation, the type, and the algorithm details, within those components so that it is possible to combine them together. In the vision axioms, each axiom performs a subset of the tasks such that these tasks can be grouped together to accomplish a more complex task. The axioms are low-level enough so that the majority of the subsets of vision problems can be represented by them, and after combining them, it results in high-level solutions [5].

The axioms share common elements defined globally, such as images and elementary data types. The time, images, and colors are treated as continuous signals in the axioms of vision. For example, color is RGB, with each channel represented

---

[6]For more about SVM, refer p. 515 in Chap. 17.

[7]*Quadratic Programming* is process of solving a linearly constrained quadratic optimization problem (minimizing or maximizing) a quadratic function of several variables subject to linear constraints on these variables.

[8]The technique of *kernel-trick* does not require explicit mapping used for linear learning algorithms to learn a nonlinear function.

in the interval of [0, 1]. The width and height of an image are represented in similar way, with additional value for the aspect ratio. Since time is treated as a continuous signal, it allows different inputs to the vision system.

The classes of common axioms for vision are: mathematical axioms, source axioms, model axioms, and construct axioms. These are based in a problem-centric taxonomy. Each axiom is further divided into three parts: a. description of its tasks, b. its representations, and c. the processing it would perform when implemented.

### 21.11.1   Mathematical Axioms

The functionality is encapsulated in these axioms, such as to perform transformations, optimizations, and other necessary formulations in machine vision.

*Optimize*

This axiom comprises optimization functions of dynamic programming, linear programming, graph cuts, and greedy methods.

*Transform*

The necessary mathematical descriptions are contained in this axiom to carry out the transformation of vision objects. It includes operations of linear algebra, geometry, and other formulations necessary for computer vision. The transformation may comprise the operations, like matrix multiplications and projections.

*Similarity*

The similarity axiom is used for recognition, correspondence, and tracking. It comprises various matrices for evaluating similarity.

### 21.11.2   Source Axioms

The source axioms are used for encapsulating the source information. Thus, they also describe the acquired data or the source of the acquired data, such as cameras, images, and properties of images like noise and illuminations in the images.

*Noise*

This axiom provides the description of noise in the data.

*Light*

The lighting models are required to approximate the lighting in a scene. It may have simple systems like in computer graphics, or variation of intensity across images, to add the effect of flash, or combining of the images, etc.

*Camera*

The camera axioms provide the general description of the camera, and also include other models and specific parameters for calibration, like focal length, principal point, etc.

*Image*

The image comprises descriptions for low-level image processing techniques, and representation of image, like resize and filter operations.

*Blur*

The blur is due to motion of camera or the object. The focus is deliberately used for creative effect or measurement of depth from focus.

### *21.11.3 Model Axioms*

The model axioms are used for representing abstract concepts used within computer vision, such that they are accessible.

*Model*

This axiom contains descriptions of models used in the vision, like geometric model, probabilistic model, or example-based model. A model unit can provide conversion from one format to another format.

*Object*

This axiom provides types and the descriptions of objects.

*Shape*

It finds the similarly shaped regions within an image, which is carried out in conjunction with matching and similarity axioms.

*Texture*

It performs synthesis and analysis with the help of texture descriptor. It also finds regions of similar texture, with the help of matching and similarity axioms.

### *21.11.4 Construct Axioms*

The construct axioms represent constructs used within vision for the purpose of modeling and used as output to other modeling packages.

*Mesh*

The mesh axiom is for centralized description of a mesh, using triangles, quads, etc. A mesh unit provides an input–output system, for construction of meshes from an image base or from an arbitrary basis in $3D$, for reconstruction of output.

*Grid*

The grid axiom is used for providing $N$-dimensional grids' description, for use in vision. For example, a $2D$ grid would be a discretized image, and a $3D$ grid would be a set of voxels,[9] etc. A grid can be indexed at intersections of grid lines. A grid unit also provides methods of grid construction, conversion, and for IO methods.

*Algorithm Composition*

The vision axioms can be combined together through loose coupling to form more sophisticated algorithms. The basic axioms already encapsulate the sophisticated concepts and algorithms; and combining these we can create higher level systems such as correspondence systems, tracking, $3D$ reconstructions, etc. As an example, for creating a simple $3D$ reconstruction such as *visual hull*, a camera can provide calibrations, grid can provide volumetric constructs, transformation will give projected grid points, and mesh can create a model for output from volumetric grid.

It is possible to create a simple tracking system with direct use of axioms: initialization of this system is done using *object*, *matching*, and *similarity*. And then, an object can be tracked through a sequence of images through use of *move* and *optimization* axioms.

However, the algorithm composition is not suited to be used by developers, because it requires expert knowledge of vision. Instead of that use, we look at top most level, which is problem itself. This is because the problem is decomposed into smaller level parts, and we use this as a basis for solving the problem. This simplifies the implementation of computer vision using the axioms.

## 21.12 Computer Vision Tools

The open source computer vision library, the OpenCV is available under a Berkley Software Distribution (BSD) license and it is free for use both by academic and commercial users. The OpenCV has interfaces with C++, C, Python, and Java, and it is supported by all major Operating Systems. OpenCV was designesd keeping the computational efficiency as one of the goals, and with a strong focus on realtime applications. The OpenCV is written in optimized C/C++ code, the library can take advantage of the availability *multicore processing* in the computer being used for running these library programs. The OpenCV is enabled with OpenCL, and it can take advantage of the hardware acceleration of the underlying heterogeneous

---

[9]*Voxels:* Each array of elements of 3D volumetric space.

computer systems. Usage of OpenCV range from interactive art, to mines inspection, to stitching maps on the web through advanced robotics [6].

The OpenCV is designed with other goals of building tools for solving computer-vision problems. It comprises a mixture of low-level image processing functions and high-level algorithms for the applications of, face recognition, face detection, pedestrian detection, feature matching, and object tracking.

The following is an example of a simple program in OpenCV to load and display a given image file, whose name is provided as input at Linux command line [7].

**Example 21.2   Load and display an image file.**

```
/* loaddisp.cpp, OpenCV ver 3.2.0 */
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/core/core.hpp>
#include <iostream>
using namespace std;
using namespace cv;
int main(int argc, char** argv)
{
    if(argc != 2)
    {
     cout << "error" << std::endl;
     return -1;
    }

    Mat imagemat;
    // Read image file
    imagemat = imread(argv[1], CV_LOAD_IMAGE_COLOR);
    // Check for invalid input
    if(! imagemat.data)
     {
        cout <<  "error" << std::endl;
        return -1;
    }
    //Create a display window
    namedWindow("Display Window", WINDOW_AUTOSIZE);
    //Show the image inside window
    imshow("Display Window", imagemat);
    //press any key to terminate the program
    waitKey(0);
    return 0;
}
```

The above program is compiled at command level in Ubuntu 18.04.1 by

```
g++ loaddisp.cpp -ggdb ⌐pkg-config --cflags --libs opnecv⌐
```

and run by command

```
./a.out krc.jpg
```

On run, the screen will display the image in file krc.jpg. □

All modern smartphones and most tablets also contain one or more cameras, and OpenCV is also available on both Android and iOS operating systems. With all these components, it is possible to create vision applications for mobile platform also.

## 21.13 Summary

The vision system has significance due to the fact that, it is one of the essential characteristics of living beings, and specifically, all the higher animals depend on this ability for their survival. The vision gives us detailed information about and around us, without much conscious efforts.

The goal of *computer vision* or *machine vision* is to extract the information from images, so that one can produce a structure from image, e.g., a 3D view from source images that are 2D. Having this available, it can recover the 3D model of an object, which in turn can be useful in medical imaging to recover 3D objects, say tumor. The methods can find best matches for stored objects of image data, for example, we can search some individual person in a collection of library of photographs, or in collection of video.

Initially, the vision systems were designed for military applications, like visual guidance for autonomous vehicles, target recognition, etc. However, in recent many applications have emerged in medical, for example, for preoperative scan for patients, virtual reality, image-based data retrieval, assembly and material handling, visual inspection, and photo interpretation.

Every object, other than mirror, scatters the light incident on it in all directions, this reflected light is projected on the retina of our eye through a lens, and we are able to recognize the object, through the interpretation by brain. The vision problem, which is related to this statement is, given the 2D image, how to infer the shape of the object that produced this image. However, it is a challenge to reason back from image to the scene that created it. The recognition requires learning, i.e., mapping image patterns to some internal representation—transformation of scene to image. This requires construction of models, i.e., general description of each object that is to be recognized. A model includes shape, texture, and context knowledge.

Going from image-to-scene is a challenging task. A vision system receives only the reflected brightness as input in different intensities. To convert these brightnesses into world measurements, we first need to understand how objects are mapped into image

brightnesses. The light received by individual sensor call is a complex interaction of: position and object's orientation, its 3D shape, reflection properties of its surface, properties of light source as well as its spectral sensitivity.

For vision, what is required is inversion of brightness into scene parameters, using any of the following methods: inversion by fixing scene parameters, by restricting problem domain, or by acquiring additional images.

A general approach to computer vision is that a vision process ends with 3D reconstruction of shapes by means of *geometric primitives* at symbolic level, linguistic level, and intermediate. However, the interpretation of the conceptual categories at linguistic levels involves problems as the concepts hardly correspond to classic categories that can be described in terms of necessary and sufficient conditions. In fact, the membership of the categories is not 0 and 1. Hence, knowledge in conceptual categories is affected by measurement errors. The neural networks make it possible to avoid the exhaustive description of conceptual categories, accordingly, they are better fit as models.

Depending on the complexities involved, the vision is classified as low-level vision (image and output are in same coordinate system), medium level vision (input is low-level vision, and output is something other than pixels of the image, say 3D image), and high-level vision (input is low and middle-level vision algorithms, and output is abstract data structures).

One of the applications of vision system is *indexing* of images, or scenes in a video. Having this facility, it is possible to lookup an object in a library of stored models, and that too in time that is independent of number of models in storage. To carry out the indexing, it requires finding the *geometric invariants* in model recognition, i.e., attributes of images that do not change due to change of viewpoints of the object. For indexing, we make use of geometric hashing approach, which has two basic steps: 1. construction of a hash table for models, and 2. matching of model to the current image.

One important area of vision systems is *object tracking*. Due to lot of demand for automated video analysis, interest in video analysis has increased. The video analysis has three steps: a. detecting the moving objects, b. object tracking from frame to frame, and c. objects' analysis to recognize their behavior. The major applications of object tracking are: automated surveillance, video indexing, motion-based recognition, traffic monitoring, Human–Computer Interaction (HCI), and vehicle navigation.

The tracking process is a complex task due to loss of information caused by projections, noise in images, partial and full object occlusions, complex shapes of the objects, and the requirements of real-time processing.

To detect an object, point detectors are used to: find points of interest, subtraction of image background, and segmentation of the image. Object detection is performed by learning different views from a set of examples using supervised learning. A standard formulation of supervised learning is: classification problem, such that the learner approximates the behavior of a function by generating an output in the form of either continuous value, called *regression*, or labeling of classes, called, *classification*.

The *support vector machines* and *adaptive boosting* are the examples of supervised learning.

The complexity of vision systems, where basic primitives are identical, for example, in 3D objects, and in dynamics of objects, can be simplified by defining and constructing *axioms* of vision. These are the basic building blocks from which we derive new constructs. The axioms are ways for representations, as well as correspond to certain algorithms within these, and are useful for common subtasks in vision systems.

The common axioms in vision are: mathematical axioms (transformation, optimization, linearity), source axioms (camera, image, light), model axioms (object, shape, texture), and construct axioms (mesh, grid, algorithm).

One of the important tool for Vision system research is the open source computer vision library, OpenCV released under a BSD license, is free for academics as well as commercial use. It provides C++, C, Python, and Java interfaces, and has been designed for efficiency of computation with a focus mainly on real-time applications.

## Exercises

1. Suggest some experimental observations about human vision that support the idea that *vision is graphics* — what we see is explicable only partly by the optical image itself, but is more strongly determined by top-down knowledge, model-building, and inference processes.
2. Explain in your own words, with supporting geometry, as how the human vision system inverts the image formed on the retina of eye(s) to visualize the object.
3. How the vision is a learning and cognition process? Justify.
4. Why it is deterministically not possible to recognize an object given an image?
5. We can always map the coordinates from an object to its image, but mapping the image coordinates to object is challenging. Why?
6. Suggest any five new applications of machine vision, which are not discussed in this chapter.
7. Most smartphones available now have feature of face detection. Why this feature is important? Explain the process of face detection in your own words.
8. What are the *syntax* and *semantics* information in a vision? Explain in brief.
9. How the axioms of vision are helpful in machine vision?
10. What is image inversion? Explain some of the techniques for this, and the significance of each of them.
11. Give proper examples of low-level, middle-level, and high-level visions.
12. Like, two stereo cameras, our eyes are able to find out the depth of vision, e.g., how far roughly it is from the eyes. Write an algorithm, to compute this depth of vision.
13. Does the width between the centers of human eyes, anyway make difference in computing accuracy of depth of vision? Give a proof for this, based on some computations.

14. Given any image having number of human faces, write an algorithm, to count the number of faces in the given image.
15. Write an algorithm for panorama-stitching for given *n* number of images.
16. Suggest various approaches to speedup the processing of image transformations.

# References

1. Charniak E, McDermott D (1998) Introduction to artificial intelligence, Addison-Wesley publication
2. Chella A et al (1997) A cognitive architecture for artificial vision. Artif Intell 89:73–111
3. Felzenszwalb P et al (2013) Visual object detection with deformable part models. Commun ACM 56(9):97–105. https://doi.org/10.1145/2494532
4. Grimson WEL, Mundy JL (1994) Computer vision applications. Commun ACM 37(3):44–51
5. Miller G et al (2011) A conceptual structure for computer vision. In: Canadian conference on computer and robot vision. https://doi.org/10.1109/CRV.2011.29
6. https://opencv.org/. Cited on Dec 2017
7. Pulli K et al (2012) Real-time computer vision with OpenCV. ACMQUEUE 10(4):1–17
8. Suetens P et al (1992) Computational strategies for object recognition. ACM Comput Surv 24(1):5–61
9. Tucker AB Jr (1997) The computer science and engineering handbook, CRC Press
10. Yilmaz A et al (2006) Object tracking: a survey. ACM Comput Surv 38(4):1–45

# Further Readings

1. Auger A, Doerr B (2011) Theory of randomized search heuristics—foundations and developments. Series vol 1, Theoretical Computer Science, World Scientific
2. Briggs R (1985) Knowledge representation in Sanskrit and artificial intelligence. AI Mag AAAI 6(1):32–39
3. Chakbarti S (1999) Mining Web's link structure. Computer 4:60–67
4. Golding AR et al (1996) Improving accuracy by combining rule-based and case-based reasoning. Artif Intell 87:215–254
5. Jansen BJ, Spink A (2007) Sponsored search: is money a motivator for providing relevant results? Computer 5:52–57
6. Jansen BJ et al (2009) The components and impact of sponsored search. Computer 98–101
7. Joseph M et al (1999) Interactive data analysis: the control project. Computer 4:51–58
8. Koehler J (1996) Planning from second principles. Artif Intell 87:145–186
9. McKinly R (2013) Proof nets for Herbrand's theorem. ACM Trans Comput Logic 14(1)
10. Minsky M (1970) Form and content in computer science. J ACM 17(2):197–215
11. Natalia D, Iguez iR et al (2014) A survey on ontologies for human behavior recognition. ACM Comput Surv 46(4)
12. Ritchie GD, Hanna FK (1984) AM: a case study in AI methodology. Artif Intell 23:249–268
13. Ruspini EH (1991) On the semantics of fuzzy logic. Int J Approx Reason 5:45–88
14. Shankar N (2009) Automated deduction for verification. ACM Comput Surv 41(4). https://doi.org/10.1145/1592434.1592437
15. Sowa JF (1997) Knowledge representation: logical, philosophical, and computational foundations. PWS Publishing, Boston, Mass
16. Vishnupriya R, Devi T (2014) Speech recognition tools for mobile phone—a comparative study. In: IEEE International Conference on Intelligent Computing Applications (ICICA). https://doi.org/10.1109/ICICA.2014.93

# Index