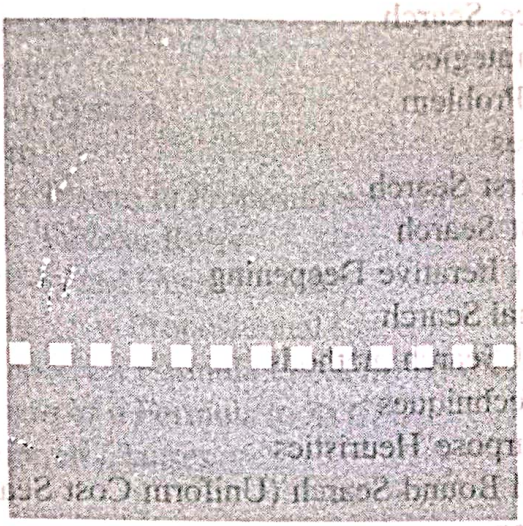
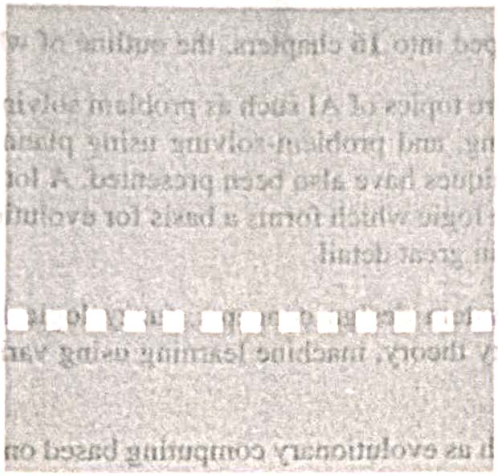
17BQ1A1259



Contents



Preface

Artificial Intelligence (AI) is a vast and truly universal field, the idea of which originated long before computers came into existence. However, tremendous growth has been observed in this field in the past two decades owing to valuable contributions from scientists from a variety of domains such as computers, medicine, philosophy, psychology, and linguistics.

This textbook *Artificial Intelligence* aims to provide comprehensive material on this subject to undergraduate and graduate students who possess an academic background of Computer Science. The book has been written keeping in mind the syllabi designed for courses on AI in various technical institutions and universities in India and abroad. The book can serve as a textbook for the first-level course for one full semester on AI since it covers all topics required for the study of the subject. It will also provide study material to computer professionals who wish to expand their knowledge by learning about AI.

The main topics covered in the book include problem-solving using intelligent searches and planning, knowledge representation techniques, game playing, first-order predicate logic and Prolog (programming in logic) programming language, uncertainty handling, expert systems. The language Prolog has been used throughout the book to write programs for problems to be solved using AI techniques. In addition, some advanced topics such as machine learning, fuzzy logic, artificial neural network, evolutionary computing, advanced knowledge representation techniques, agent technology and natural language processing have been included in detail. In order to derive maximum benefit from this book, reader should be familiar with concepts of data structure and programming in any language.

Each chapter in the book has been carefully developed with the help of several pedagogical features. A large amount of effort has been put in to ensure that every concept discussed in the book is explained with the help of examples as far as possible. Pseudo algorithms for various methods and techniques are included throughout the book to increase the comprehensibility of the topics and demonstrate their applications.

Contents and Structure

The book has been diligently developed into **16** chapters, the outline of which is provided below.

Chapters 1 to 7 include basic and core topics of AI such as problem solving using exhaustive and heuristic searches, techniques for game playing, and problem-solving using planning techniques. Various widely used knowledge-representation techniques have also been presented. A lot of emphasis has been placed on the discussion of first-order predicate logic which forms a basis for evolution of logic programming named Prolog which has also been covered in great detail.

Chapters 8 to 12 include expert system design concepts, fuzzy logic for capturing fuzzy knowledge, uncertainty handling using probability theory, machine learning using various models and artificial neural network.

Chapters 13 to 16 include topics such as evolutionary computing based on genetic algorithms, ants colony, and swarm intelligence. Knowledge representation using conceptual dependency, intelligent agents and their architectures and a detailed chapter on natural processing have also been included in this book.

Acknowledgements

First and foremost, I would like to thank GOD for giving me motivation and energy to write this book. Thanks are due to IIT Delhi authorities for giving me a sabbatical leave for a year for this purpose. During this period, I went to Maharishi University of Management, Fairfield, IOWA, US to teach a one-month crash course on AI to their MS (Computer Science) students and received lot of useful feedback and comments from the students. I would like to thank them all.

I would like to thank my late father who had always been a source of inspiration and was proud of me. I lost him during this period but always felt his constant blessings. In fact this book is dedicated to him.

Very special thanks are due to my loving husband and daughter. My husband has been a constant source of encouragement and moral support. Without the support of my family, this work would have not been possible.

I would also like to take this opportunity to express my gratitude to everyone who, directly or indirectly, contributed to the successful completion of the book.

SAROJ KAUSHIK

1

Introduction to Artificial Intelligence

1.1 Introduction

The foundation of Artificial Intelligence (AI) was laid with the development of Boolean theory and principles by a mathematician named Boole and others researchers. AI has come a long way from its early inception. Since the invention of the first computer in 1943, AI has been of interest to the researchers as they have always aimed to make machines more intelligent than humans and tried to simulate intelligent behaviour. Over the last six decades, AI has grown substantially starting from simple programs to intelligent programs such as programs for playing games, expert systems, intelligent robots, agents, etc. AI researchers generally use one of the two basic approaches, namely bottom–up and top–down, for creating intelligent machines. In bottom–up approach, the belief is to achieve artificial intelligence by building electronic replicas of the human brain which is a complex network of neurons. In the top–down approach, the attempt is to mimic the behaviour of brain with computer programs. There have been many studies and researches about parameters and factors that make people intelligent. People have believed that the mind gains its power from its ability to reason symbolically because of the fact that knowledge is stored in a form directly equivalent to a set of symbols, and reasoning means manipulation of these symbols.

AI currently comprises of a huge variety of subfields ranging from general-purpose areas such as perception, logical reasoning, to specific tasks such as game playing, theorem proving, diagnosing diseases, etc. Scientists of other fields use tools of artificial intelligence to systematize and automate the intellectual tasks of their fields. AI is engaged in two significantly different enterprises, namely a science of human intelligence and an engineering discipline concerned with building smarter physical systems. This field is truly a multi-disciplinary field and is based on the work done in different disciplines such as logic, cognition, linguistics, philosophy, psychology,

anthropology, computing etc. AI application problems can be found in all disciplines from software engineering to film theory. AI programs tend to be large, and it would have not been possible to work unless there is great advancement in speed and memory of computers.

In this chapter, we briefly introduce the subject by including brief history, introduction to intelligent systems, foundation and characteristics of AI system, various sub areas and applications of AI, current trends in AI followed by chapterwise layout of the book.

1.2 Brief History

The history of AI had cycles of success and failures but kept on introducing new creative approaches and systematically refining the best ones. There was no relation between human intelligence and machine till early 1950 even though philosophers in 400 B.C. had conceived the ideas that the mind operates on knowledge encoded in some internal language in the same ways as a machine does. Psychologists further strengthened the idea that humans and other living creatures can be considered to be information processing machines. Mathematicians provided tools to manipulate certain or uncertain logical statements of certainty as well as probabilistic statements. John McCarthy organized a conference on machine intelligence in 1956 and since then the field was known as Artificial Intelligence. In 1957, the first version of a new program named as General Problem Solver (GPS) was developed and tested. This program was also developed by Newell and Simon. The GPS was capable of solving to some extent the problems requiring common sense. Since then, many programs were developed, and McCarthy announced his new development called LISt Processing language (LISP) in 1958. LISP was adopted as the language of choice by most AI developers. Marvin Minsky of MIT demonstrated that computer programs could solve spatial and logic problems when confined to a specific domain. Another program, named STUDENT, was developed during late 1960 which could solve algebra story problems. Fuzzy set and logic was developed by L. Zadeh in 1960 that had the unique ability to make decisions under uncertain conditions. Also neural networks were considered as possible ways of achieving Artificial Intelligence. During the same time, the system named SHRDLU was developed by Terry Winograd at the MIT, AI Laboratory, as a part of the micro worlds project, which consisted of research and programming in small worlds. SHRDLU was a program that carried out a simple dialogue (via teletype) with a user in English, about a small world of objects (the BLOCK world) and this program was written in MACLISP. Frame theory was one of the new methods developed by Minsky during 1970 in the development of AI for storing structured knowledge to be used by AI programs. Another development during the same time was the PROLOG language that was initially proposed by R Kowalski of Imperial College, London.

Advent of the expert system appeared in 1970, when Expert systems were designed and developed to predict the probability of a solution under set conditions. An *expert system* is a program that uses logical rules that are derived from the knowledge of experts to answer the question or solves problems about a specific domain of knowledge. Since then, various expert systems for applications such as forecasting the stock market, aiding doctors with the ability to diagnose disease, and

instructing miners to promising mineral locations were developed. These would have been possible because of large storage capacity of computers at the time and ability to store conditional rules and information. Research organizations and the corporate sector during 1980 started developing AI systems at faster pace. Expert systems in particular were in great demand because of their efficiency. Companies such as Digital Electronics were using XCON, an expert system designed to program the large VAX computers. DuPont, General Motors, and Boeing relied heavily on expert systems. New theories about machine vision was proposed by David Marr, where it was possible to distinguish an image based on basic information such as shapes, colour, edge, texture, and the shading of an image. By 1985, over a hundred companies offered machine vision systems in the US. Researchers always dreamt of evolving machine similar to human evolution. Human brain works a billion times faster as compared to computer, but a computer still helps in doing things at a faster pace as compared to humans. Work on simulating human brain started from 1986 onwards. Neural networks were used to simulate brain functioning. Neural network provided a model for massive parallel computation. More recently, work in AI has started from agent point of view. Agents are viewed to be entities that receive percepts constantly from dynamic environment and perform actions. So, agents can be used in solving any problem which requires intelligence. Understanding of a problem and situations is embedded in agents and the problem is solved. Recent progress in understanding the theoretical basis for intelligence has gone hand in hand with improvements in the capabilities of real systems.

1.3 Intelligent Systems

As already mentioned earlier, AI is a combination of computer science, physiology, and philosophy. AI is a broad topic, consisting of different fields, from machine vision to expert systems. John McCarthy was one of the founders of AI field, who stated that AI is the science and engineering of making intelligent machines, especially intelligent computer programs. Further, he defines intelligence as the computational part of the ability to achieve goals in the world. Different people think of AI differently and there is no unique definition. Various authors have defined AI differently as given below.

- AI is the study of mental faculties through the use computational models (Charniak and McDermott, 1985).
- The art of creating machines that perform functions which require intelligence when performed by people (Kurzweil, 1990).
- AI is a field of study that seeks to explain and emulate intelligent behaviour in terms of computational processes (Schalkoff, 1990).
- AI is the study of how to make computers do things at which, at the moment, people are better (Rich and Knight, 1991).
- AI is the study of the computations that make it possible to perceive, reason, and act (Winston, 1992).

- AI is the branch of computer science that is concerned with the automation of intelligent behaviour (Luger and Stubblefield, 1993).

In brief, we can define AI as the study of making computers do things intelligently. It then requires the study and creation of computer systems that exhibit some form of intelligence or the characteristics which we associate with intelligence in human behaviour. Hence AI programs must have capability and characteristics of intelligence such as learning, reasoning, inferencing, perceiving and comprehending information.

Understanding of AI: It requires an understanding of related terms such as intelligence, knowledge, reasoning, cognition, learning, and a number of other computer-related terms. AI techniques and ideas seem to be harder to understand than most things in computer science. AI shows best on complex problems for which general principles do not help much, though there are a few useful general principles. AI is also difficult to understand by its content. The boundaries of AI are not well defined. Generally it is thought to be an advanced software engineering, sophisticated software techniques for hard problems that cannot be solved in any easy way. AI also involves working in domains for which the problem is often poorly understood. This gives rise to a need for flexible environments and languages capable of supporting rapid changes in knowledge representation schemes and alterations to the inference processes at work. AI programs like human are usually not perfect, and even make mistakes. As people cannot handle numbers well and need not necessarily provide the best solution AI programs also can be treated as non-numeric (common sense) ways of solving problems and need not be perfect but can give reasonably good solutions. There are two views of AI goals.

- The first view is that AI is about duplicating what the (human) brain DOES (Cognitive Science)
- The second view is that AI is about duplicating what the (human) brain SHOULD do, that is, doing things logically or rationally.

There are lot of challenges in building systems that mimic the behaviour of the human brain which is made up of billions of neurons. In order to classify systems as intelligent, it is necessary to define intelligence. The earliest best method to gauge the intelligence of a system was the *Turing test* that was proposed by Alan Turing in 1950. It was designed to provide a satisfactory operational definition of intelligence. Alan Turing defined intelligent behaviour as the ability to achieve human-level performance in all cognitive tasks, sufficient to fool an interrogator. He said that a system is called intelligent if it has an ability to pass the Turing test which is informally defined as follows:

- A system is said to have passed Turing test if a human questioner is unable to determine from repeated questions of any kind, whether he or she is talking to another person or to a machine/system.

The very first so-called intelligent system named *ELIZA* passed the Turing test which was written by Joseph Weizenbaum during the period from 1964 to 1966. This program created breakthrough at that time and researchers started aiming very high of making human-like machines. Before proceeding further, let us briefly describe Eliza in the following subsection.

1.3.1 ELIZA

Eliza was a program that conversed with user in English. People were amazed to see this program. The program was able to converse about any subject, because it stored subject information in data banks. Another feature of Eliza was its ability to pick up speech patterns from user's questions and provide responses using those patterns. In this mode, ELIZA mostly rephrased the user's statements as questions and posed those to the user. The following passage contains dialogue between Eliza and a teenage girl. The system initiates the conversation marked as > and user's responses are marked as *.

Passage of Eliza Conversation

```
> Hello, I am Eliza.
* Can you help me?
> Don't you believe that I can help you?
* Yes.
> I see.
* I feel depressed.
> How long have you been feeling depressed?
* Since last month.
> That is quite interesting.
* what is interesting in this?
> Have you asked anyone else?
* No
> Are you saying no just to be negative?
* No I did not ask anyone.
> Why no?
* I do not believe other people.
> Say, do you have any psychological problems?
* I do not know.
> Tell me more.
```

Since then lot many versions of Eliza have been developed and are available on the Internet. The basic philosophy and characteristics in all these programs are the same. The main characteristics of Eliza are briefly mentioned here:

Simulation of Intelligence: Eliza programs are not intelligent at all in real sense. They do not understand the meaning of utterance. Instead these programs simulate intelligent behaviour quite effectively by recognizing key words and phrases. By using a table lookup, one of a few ways of responding question is chosen.

Quality of Response: It is limited by the sophistication of the ways in which they can process the input text at a syntactic level. For example, the number of templates available is a serious limitation. However, the success depends heavily on the fact that the user has a fairly restricted notion of the expected response from the system.

Coherence: The earlier version of the system imposed no structure on the conversation. Each statement was based entirely on the current input and no context information was used. More complex versions of Eliza can do a little better. Any sense of intelligence depends strongly on the coherence of the conversation as judged by the user.

Semantics: Such systems have no semantic representation of the content of either the user's input or the reply. That is why we say that it does not have intelligence of understanding of what we are saying. But it looks that it imitates the human conversation style. Because of this, it passed Turing test.

1.3.2 Categorization of Intelligent Systems

In order to design intelligent systems, it is important to categorize these systems. There are four possible categories of such systems (Luger and Stubblefield, 1993), (Russell and Norvig, 2003).

- System that thinks like humans
- System that acts like humans
- System that thinks rationally
- System that acts rationally

System that thinks like human requires cognitive modelling approaches. Most of the time, it is a black box where we are not clear about our thought process. One has to know the functioning of the brain and its mechanism for processing information. It is an area of cognitive science. The stimuli are converted into mental representation and cognitive processes manipulate this representation to build new representation that is used to generate actions. Neural network is a computing model for processing information similar to brain.

System that acts like human requires that the overall behaviour of the system should be human like which could be achieved by observation. Turing test is an example.

System which thinks rationally relies on logic rather than human to measure correctness. For thinking rationally or logically, logical formulae and theories are used for synthesizing outcomes. For example, given *John is a human and all humans are mortal* then one can conclude logically that John is mortal. It should be noted that not all intelligent behaviours are mediated by logical deliberation.

System that acts rationally is the final category of intelligent system where by rational behaviour we mean doing the right thing. Even if the method is illogical, the observed behaviour must be rational.

To summarize, we can define intelligence to be that property of mind which encompasses many related mental abilities. Some of the capabilities are given as follows:

- Reason and draw meaningful conclusions
- Plan sequences of actions to complete a goal
- Solve problems
- Think abstractly
- Comprehend ideas and help computers to communicate in Natural Languages (NL)
- Store knowledge provided before or during interrogation
- Learn new ideas from environment and new circumstances
- Offer advice based on rules and situations
- Learn new concepts and tasks that require high levels of intelligence

1.3.3 Components of AI Program

AI techniques must be independent of the problem domain as far as possible. Any AI program should have *knowledge base*, and navigational capability which contains *control strategy* and *inference mechanism*.

Knowledge base: AI programs should be learning in nature and update its knowledge accordingly. Knowledge base generally consists of facts and rules and has the following characteristics:

- It is voluminous in nature and requires proper structuring.
- It may be incomplete and imprecise.
- It may be dynamic and keep on changing.

Control strategy: It determines which rule to be applied. To know this rule, some heuristics or thumb rules based on problem domain may be used.

Inference mechanism: It requires search through knowledge base and derives new knowledge using the existing knowledge with the help of inference rules.

1.4 Foundations of AI

Commonly used AI techniques and theories are rule-based, fuzzy logic, neural networks, decision theory, statistics, probability theory, genetic algorithms, etc. Since AI is interdisciplinary in nature, foundations of AI are in various fields such as:

- Mathematics
- Neuroscience
- Control theory
- Linguistics

Mathematics: AI systems use formal logical methods and Boolean logic (Boole, 1847), analysis of limits to what can be computed, probability theory, uncertainty that forms the basis for most modern approaches to AI, fuzzy logic, etc.

Neuroscience: This science of medicine helps in studying the functioning of brains. In early studies, injured and abnormal people were used to understand what parts of brain work. Now recent studies use accurate sensors to correlate brain activity to human thought. By monitoring individual neurons, monkeys can now control a computer mouse using thought alone. Moore's law states that the computers will have as many gates as humans have neurons in the year 2020. Researchers are working to know as to how to have a mechanical brain. Such systems will require parallel computation, remapping, and interconnections to a large extent.

Control theory: Machines can modify their behaviour in response to the environment (sense/action loop). Steam engine governor, thermostat and water-flow regulator are few examples of control theory. In 1950, control theory could only describe linear systems. AI largely rose as a response to this shortcoming. This theory of stable feedback systems helps in building systems that transition from initial state to goal state with minimum energy.

Linguistics: Speech demonstrates so much of human intelligence. Analysis of human language reveals thought taking place in ways not understood in other settings. Children can create sentences they have never heard before. Languages and thoughts are believed to be tightly intertwined.

1.5 Sub-areas of AI

As we have already mentioned earlier that AI is an interdisciplinary area having numerous diverse subfields. Each one of these field is an area of research in AI itself. Some of these are listed below:

- Knowledge representation models
- Theorem proving mechanisms
- Game playing methodologies
- Common sense reasoning dealing with uncertainty and decision making
- Learning models, inference techniques, pattern recognition, search and matching, etc.
- Logic (fuzzy, temporal, modal)
- Planning and scheduling
- Natural language understanding, speech recognition and understanding spoken utterances

- Computer vision
- Models for intelligent tutoring systems
- Robotics
- Data mining
- Expert problem solving
- Neural networks, AI tools, etc.
- Web agents

1.6 Applications

AI finds applications in almost all areas of real-life applications. Broadly speaking, business, engineering, medicine, education and manufacturing are the main areas.

- Business: financial strategies, give advice
- Engineering: check design, offer suggestions to create new product, expert systems for all engineering applications
- Manufacturing: assembly, inspection, and maintenance
- Medicine: monitoring, diagnosing, and prescribing
- Education : in teaching
- Fraud detection
- Object identification
- Space shuttle scheduling
- Information retrieval

Let us take an example of a two-player game named Tic–Tac–Toe. We will see different approaches that move toward being representations of AI techniques (Rich and Knight, 1991).

1.7 Tic–Tac–Toe Game Playing

Let us consider Tic–Tac–Toe game problem and three approaches for solving it. It is a two-player game with one player marking **O** and other marking **X**, at their turn in the spaces in a 3×3 grid. The player who succeeds in placing three respective marks in any horizontal, vertical or diagonal row wins the game. Here we are considering one human player and the other player to be a computer program. The objective to play this game using computer is to write a program which never loses. We present here three approaches to play this game which increase in

- Complexity
- Use of generalization

- Clarity of their knowledge
- Extensibility of their approach

It is assumed that X is the first player, which might either be human or computer.

1.7.1 Approach 1

Let us represent 3 × 3 board as nine elements vector. Each element in a vector can contain any of the following three digits:

- 0 – representing blank position
- 1 – indicating X player move
- 2 – indicating O player move

It is assumed that this program makes use of a move table as shown in Table 1.1 that consists of vector of 3^9 (19683) elements.

Table 1.1 Move Table

Index	Current Board Position	New Board Position
0	000000000	000010000
1	000000001	020000001
2	000000002	000100002
3	000000010	002000010
	⋮	

The entries of the table are carefully designed manually in advance keeping in mind that the computer should never lose. Each entry is indexed by decimal representation of current board position digits. Initially the board is empty and is represented by nine zeros. The best new board position (0 0 0 0 1 0 0 0 0) is obtained by putting 1 (representing move by X player) in the fifth cell. All possible board positions are stored in *Current Board Position* column along with its corresponding next best possible board position in *New Board Position* column. Once the table is designed, the computer program has to simply do the table lookup. Let us write algorithm for this version of a program as follows:

Algorithm

- View the vector (board) as a ternary number.
- Get an index by converting this vector to its corresponding decimal number.

- Get the vector from *New Board Position* stored at the index. The vector thus selected represents the way the board will look after the move that should be made.
- So set board position equal to that vector.

Disadvantages: This version of the program is very efficient in terms of time but has several disadvantages.

- It requires lot of memory space to store the move table.
- To specify entries in move table manually, lot of work is required.
- Creating move table is highly error prone as data to be entered is voluminous.
- This approach cannot be extended to 3D tic–tac–toe. In this case, 3^{27} board position are to be stored.
- This program is not intelligent at all as it does not meet any of AI requirements.

Let us develop second version of the program using approach 2 which makes use of human style of playing game. Here again the board is represented by 9 element vector. We hard code the fact that initially computer at its chance plays in the center, if possible, otherwise tries the various non-corner squares.

1.7.2 Approach 2

The board B[1..9] is represented by a nine-element vector. In this case we choose the following digits to represent blank, X player and O player moves. The choice of these digits will be clear in the strategy part given below.

- 2 – representing blank position
- 3 – indicating X player move
- 5 – indicating O player move

There are in all 9 moves represented by an integer 1 (first move) to 9 (last move). We will use the following three sub procedures.

- Go(n) – Using this function computer can make a move in square n.
- Make_2 – This function helps the computer to make valid 2 moves.
- PossWin(P) – If player P can win in the next move then it returns the index (from 1 to 9) of the square that constitutes a winning move, otherwise it returns 0.

Strategy:

The strategy applied by human for this game is that if human is winning in the next move then he/she plays in the desired square, else if human is not winning in the next move then one checks if the opponent is winning. If so then block that square, otherwise try making valid two in any row, column, or diagonal.

The function *PosWin* operates by checking, one at a time, for each of rows /columns and diagonals.

- If PossWin (P) = 0, then P cannot win. Find whether opponent can win. If so then block it. This can be achieved as follows:
  - If (3 * 3 * 2 = 18) then X player can win as there is one blank square in row, column, or diagonal.
  - If (5 * 5 * 2 = 50), then O player can win.

Let us represent computer by C and human by H. The player who is playing first will be called X player. Since computer can be first or second player, Table 1.2 consists of rules to be applied by computer for all nine moves. If C is the first player (playing X), then odd moves are to be chosen otherwise, if C is playing as second player (playing O) then even moves are the ones to be followed by C. Comments in the rules are enclosed in { }.

Table 1.2 Rules for Nine Moves

(C plays X, H plays O)	(H plays X, C plays O)
1 move : Go (5) / Go(1)	**2 move :** If B[5] is blank, then Go(5) else Go(1)
3 move : If B[9] is blank, then Go(9) else ***{make 2}*** Go(3)	**4 move:** *{By now human (playing X) has played 2 moves}:* If PossWin(X) then ***{block X}*** Go(PossWin(X)) else ***{make 2}*** Go(Make_2)
5 move : *{By now both have played 2 moves}:* If PossWin(X) then ***{X wins}*** Go(PossWin(X)) else if PossWin(O) ***{block O}*** then Go(PossWin(O)) else if B[7] is blank then Go(7) else Go(3)	**6 move :** {By now computer has played 2 moves}: If PossWin(O) then ***{O wins}*** Go(PossWin(O)) else if PossWin(X) ***{block X}*** then Go(PossWin(X)) else Go(Make_2)
7 & 9 moves : *{By now human (playing O) has played 3 chances}*: If PossWin(X) then ***{X wins}*** Go(PossWin(X)) else ***{block O}*** if PossWin(O) then Go(PossWin(O)) else Go(Anywhere)	**8 move :** *{By now computer has played 3 chances}*: If PossWin(O) then ***{O wins}*** Go(PossWin(O)) else ***{block O}*** if PossWin(X) then Go(PossWin(X)) else Go(Anywhere)

This version of the program is memory efficient and easier to understand as complete strategy has been determined in advance but has several disadvantages as listed below. It applies heuristics (thumb rules), so can be treated as one step towards AI approach.

Disadvantages:

- Not as efficient as first one with respect to time. Several conditions are checked before each move.
- Still cannot generalize to 3-D.

Third approach is same as second approach except for one change in the representation of the board which helps in simplified process of checking for a possible win.

1.7.3 Approach 3

In this approach, we choose board position to be a magic square of order 3; blocks numbered by magic number. The *magic square* of order n consists of n^2 distinct numbers (from 1 to n^2), such that the numbers in all rows, all columns, and both diagonals sum to the same constant. It is generated using the following method if n is odd. There might be various other methods for generating magic square. The one we have used is defined below:

- It begins by placing 1 in middle of top row, then incrementally placing subsequent numbers in the square diagonally up and right. If a filled square is encountered, move vertically down one square instead, then continue as before. The counting is wrapped around, so that falling off the top returns on the bottom and falling off the right returns on the left. One can easily show that the sum must be $n\,[(n^2+1)/2]$ for each row, column, and diagonal.

The magic square of order 3 is shown in Table 1.3. Sum of each row, column, and both diagonals is 15.

Table 1.3 Magic Square of Order 3

8	1	6
3	5	7
4	9	2

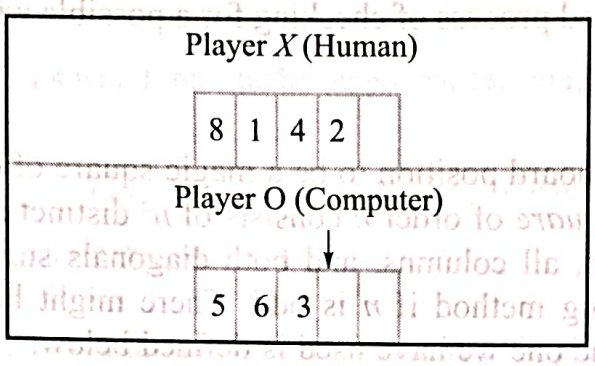
In this approach, we maintain a list of the blocks played by each player. For the sake of convenience, each block is identified by its number. The following strategy for possible win for a player is used. Obviously in our case, we are considering a computer to be a player for which the strategy is suggested as follows:

- Each pair of blocks a player owns is considered.
- Difference D between 15 and the sum of the two blocks is computed.
  - If $D < 0$ or $D > 9$, then these two blocks are not collinear and so can be ignored; otherwise if the block representing difference is blank (i.e., not in either list) then player can move in that block.
- This strategy will produce a possible win for a player.

It should be noted that first few moves are fixed as given in Table 1.2. To illustrate the method, consider the lists of both the players; say after seventh move assuming that the first player represented by 'X' is human (Table 1.4). Now it is the turn of computer at eighth move.

We choose a convention such that 'eighth block' refers to the block containing the number 8, that is, the first block of magic square.

Table 1.4 Status of Both Lists after Seventh Move



At this turn, computer checks if it can win. This checking is done by considering pair-wise blocks from its list and find a block which is collinear. Here $D = 15 - (5 + 3) = 7$ and since seventh block is not in either of the lists, computer can play in this block and declare itself as won. Let us see the execution of the program from the start and the strategy used by a computer.

Execution of Program 3: Assume that human is the first player.

- **Turn 1:** Suppose H plays in the eighth block.
- **Turn 2:** C plays in fifth block (fixed move, see from Table 1.2).
- **Turn 3:** H plays in first block.
- **Turn 4:** C checks if H can win or not.
  - Compute sum of blocks played by H
    - $S = 8 + 1 = 9$
    - Compute $D = 15 - 9 = 6$
  - The sixth block is a winning block for H and not there on either list. So, C blocks it and plays in sixth block. The sixth block is recorded in the list of computer.
- **Turn 5:** H plays in fourth block.
- **Turn 6:** C checks if C can win as follows:
  - Compute sum of blocks played by C
    - $S = 5 + 6 = 11$
    - Compute $D = 15 - 11 = 4$; Discard this block as it already exists in X list
  - Now C checks whether H can win.
    - Compute sum of pair of square from list of H which have not been used earlier
      - $S = 8 + 4 = 12$
      - *Compute $D = 15 - 12 = 3$*
    - Block 3 is free, so C plays in this block. The third block is recorded in the list of computer.
- **Turn 7:** If H plays in second or ninth block, then computer wins. Let us assume that H plays in second block.
- **Turn 8:** C checks if it can win as follows:
  - Compute sum of blocks played by C which has not been used earlier
    - $S = 5 + 3 = 8$;

- Compute $D = 15 - 8 = 7$
- Block 7 is free, so C plays in seventh block and wins the game.
- If H plays in seventh block at its **Turn 7**, then there is a draw.

The program will require much more time than other two as it must search a tree representing all possible move sequences before making each move. But the advantage is that this approach could be extended to handle three-dimensional tic–tac–toe. It could also be extended to handle games more complicated than tic–tac–toe.

Three-dimensional tic–tac–toe: This game is similar to the traditional game of tic–tac–toe, but is played on cube of order 3. Cube is created by stacking three grids each square of order 3. The goal is still to be the first to get 3-in-a-row, but now the 3-in-a-row can be on any of the three levels, or between levels. To win, a player must place three of their symbols on three squares that line up vertically, horizontally, or diagonally on a single grid, or spaced evenly over all three grids.

In this game one may use magic cube as shown in Table 1.5 which is three-dimensional equivalent of a magic square, that is, a numbers (from 1 to 27) are arranged in a $(3 \times 3 \times 3)$ pattern such that the sum of the numbers on each row, each column on six outer surfaces of a cube, each rows, each columns and two diagonal of middle grid and the four main space diagonals is equal to a single number (in this case 42) called a magic constant of the cube. It can be shown that a magic cube of order n has a magic constant equal to $n\,[(n^3 + 1)/2]$. One of such cubes is given in the Table 1.5.

Table 1.5 Magic Cube

Grid 1		
8	24	10
12	7	23
22	11	9

Grid 2		
15	1	26
25	14	3
2	27	13

Grid 3		
19	17	6
5	21	16
18	4	20

1.8 Development of AI Languages

AI languages have traditionally been those which stress on knowledge representation schemes; pattern matching, flexible search, and programs as data. The typical examples of such languages are LISP, Pop-2, ML, and Prolog. LISP is a functional language based on lambda calculus and Prolog is a logic language based on first-order predicate logic. Both languages are declarative languages where one is concerned about 'what to compute', and not 'how to compute'. Pop-2 (later extended to Pop11) is a stack-based language providing greater flexibility and has some similarity to LISP. Pop 11 is embedded in an AI Programming Environment called *Poplog*. It permits the mixed use of logic and functional languages like Prolog, LISP, and ML, and provides flexible implementation support. Other hybrid programming environments exist which focus on the domain level as opposed to the implementation level. For example, the KLONE family of languages support modelling at the knowledge level. Environments such as KEE, ART and CLIPS provide a variety of knowledge representation schemes which helps the AI programmer to integrate object-oriented representations with ones based on logic.

There are numerous descendents of such environments and many sophisticated programming languages. However, AI programming can exploit any language from BASIC through C to Smalltalk; though the ease with which languages can be exploited depends on what stage of development an AI system is at.

1.9 Current Trends in AI

Conventional computing (sometimes called hard computing) is based on the concept of precise modelling and analyzing to yield accurate results. *Hard computing* techniques work well for simple problems, but is bound by the NP-Complete set which include problems, often occurring in biology, medicine, humanities, management sciences, and similar fields. Such problems remain intractable to conventional mathematical and analytical methods as these problems are so large that it would take lifetime of the Universe to solve them, even at super computing speeds. AI community has started looking towards soft computing methodologies which are complementary rather than competitive. Furthermore, soft computing may be viewed as a foundation component for the emerging field of conceptual intelligence. Generally speaking, soft computing techniques resemble biological processes more closely than traditional techniques, which are largely based on formal logical systems. In effect, the role model for soft computing is the human mind. In many ways, soft computing represents a significant paradigm shift in the aims of computing—a shift which reflects the fact that the human mind, unlike present-day computers, possesses a remarkable ability to store and process information which is pervasively imprecise, uncertain and unstructured or uncategorized. Soft Computing, a formal Computer Science area of study from

the early 1990s refers to a collection of computational techniques in computer science, machine learning and some engineering disciplines, which study, model, and analyze very complex phenomena: those for which more conventional methods have not yielded low-cost, analytic, and complete solutions. Soft computing aims to surmount NP-complete problems by using inexact methods to give useful but inexact answers to intractable problems. Soft computing uses soft techniques contrasting it with classical artificial intelligence hard-computing techniques.

The successful applications of soft computing suggest that the impact of soft computing will be felt increasingly in the coming years. Soft computing is likely to play an especially important role in science and engineering, but eventually its influence may extend much farther. Components of soft computing include Neural networks, Fuzzy systems, Evolutionary algorithms, Swarm intelligence, etc.

Current trends in AI are basically towards the development of technologies which have origin and analogy with biological or behavioural phenomena related to human or animals such as evolutionary computation. Computer scientists have found ways to evolve solutions to complex problems. Evolutionary computation uses iterative progress, such as growth or development in a population, which is then selected in a guided random search using parallel processing to achieve the desired end. It is inspired by biological mechanisms of evolution. Evolutionary paths of AI and simulation have recently started to converge which has been possible by outgrowth research in cognitive psychology and mathematical logic. Until recently, the focus was on explaining the working of the human mind and developing general purpose problem-solving algorithms. But in contrast, simulation has been developed from the need to study and understand complex time varying behaviours exhibited by real physical systems. These technologies are helping a widely recognized goal of AI for creation of artificial scenarios that can emulate humans in their ability.

Neural networks have been developed based on functioning of the human brain. Attempts to model the biological neuron have led to the development of the field called *artificial neural network*. These systems have been developed in order to facilitate predicting features in advance based on the previous details available.

Evolutionary techniques mostly involve meta-heuristic optimization algorithms such as evolutionary algorithms (comprising genetic algorithms, evolutionary programming, etc.) and swarm intelligence (comprising ant colony optimization and particle swarm optimization).

Genetic algorithms based on Darwin theory of evolution were developed mainly by emulating the nature and behaviour of biological chromosome. Genetic algorithms are favoured for search problems which require the identification of a global optimal solution without getting stuck in local minima or maxima.

Ant colony algorithm was developed to emulate the behaviour of real ants and found suitable applications in multi-agent system to solve difficult combinatorial optimization problems. Behav-

iour of ants has applications in the discrete optimization problems. An ant algorithm is one in which a set of artificial ants (agents) cooperate to find the solution of a problem by exchanging information via pheromone deposited on graph edges.

Swarm intelligence (SI) is a type of artificial intelligence based on the collective behaviour of decentralized, self-organized systems. Social insects like ants, bees, wasps, and termites perform their simple tasks independent of other members of the colony. However, they are able to solve complex problems emerging in their daily lives by mutual cooperation. This emergent behaviour of self-organization by a group of social insects is known as *swarm intelligence*. Therefore, swarm intelligence utilizes the behaviour of non-human living entities for problem solving. Swarm intelligence is a specialization in the field of self-organizing systems (adaptation). Swarm intelligence was inspired by the social behaviour of birds flocking and fish schooling. In computational sense, a swarm has been defined as a set of (mobile) agents which are liable to communicate directly or indirectly (by acting on their local environment) with each other, and which collectively carry out a distributed problem solving. The agents follow very simple rules, and although there is no centralized control structure dictating how individual agents should behave, local, and to a certain degree random, interactions between such agents lead to the emergence of "intelligent" global behaviour, unknown to the individual agents. Advantages of such systems are: adaptability (self-organizing), robustness (ability to find a new solution if the current solution becomes invalid), reliability (agents can be added or removed without disturbing behaviour of the total system because of the distributed nature), simplicity and has no central control.

Expert system continues to remain an attractive field for its practical utility in all walk of real life.

Emergence of Agent technology as a subfield of AI is a significant paradigm shift for software development and breakthrough as a new revolution. Agents are generally situated in some environment and are capable of taking autonomous decisions while solving a problem. Multi-agent systems are designed using several independent and interacting agents to solve the problems of distributed nature. There are a wide variety of applications that range from industrial, commercial to medical fields.

1.10 Layout of the Book

The book is aimed as a textbook for a first-level full semester course on AI to undergraduate and graduate CS background students. It provides comprehensive material on the subject and can be used by computer professionals as well to learn about AI. The book includes all basic and advanced topics related to AI. The entire text of the book is divided into 16 chapters and each chapter contains number of solved examples to explain the concepts given in the text. The relevant exercises are given at the end of each chapter. The brief description of each chapter is as follows:

Chapter 1 is the current chapter consisting of an overview of AI, its evolution and history, foundation of AI and various characteristics for AI-based systems, transition from traditional program to AI-based program.

Chapter 2 is on problem solving using exhaustive and heuristic-based searches. Pseudo algorithms for almost all searches have been given for better understanding of the methods. Method for solving problem with constraints has also been included in this chapter.

Chapter 3 includes problem solving by decomposition of problem. Complex problems are decomposed into independent sub problems and are solved independently. The sub solutions are integrated at the end to find the final solution. The AO* algorithm is used for this purpose and has been explained in detail. Algorithms for various concepts are included. It also discusses various game playing strategies. Game playing is based on AND–OR graph search technique, The concept of alpha–beta pruning to improve the game playing efficiency has been explained with all possible algorithms. Finally, an example of playing NIM game has been discussed in great detail.

Chapter 4 is based on logic concepts and logic programming. An abstract system of Propositional Logic is introduced that captures certain aspects of human reasoning. In this system, a treatment is more syntactic in nature where we are concerned with the mathematical proofs and deductions of formulae. Various approaches such as Natural Deduction system, Axiomatic system, Semantic tableaux method, Resolution refutation system have been described in brief for theorem proving. Further, predicate logic that is an extension of propositional logic is included. Predicate Logic removes the limitations of Propositional Logic to a great extent. We have limited the scope of predicate logic to first-order predicate logic in which quantification is over simple variables only and not on predicates and functions. Then an evolution of logic programming, based on first-order predicate logic is included to make reader understand the logical evolution of programming language in logic named as PROLOG included in the next chapter.

Chapter 5 includes PROLOG that is an acronym for **PRO**gramming in **LOG**ic. It is based on special types of clauses called *Horn Clauses* that is either a program clause or a goal clause. Prolog has a unique capability of inferring facts from given rules and facts. We have discussed different features of programming in Prolog. Recursive programming is an important feature and execution control technique in Prolog. Number of Prolog programs has been included to explain each concept. Since Prolog's execution strategy imposes constraints on rule and goal orders, which affects efficiency and termination, the problems of termination, redundancy and goal and rule order are discussed. Handling of advanced data structures such as binary trees and objects is included to make Prolog comparable to any imperative language.

Chapter 6 is on advanced problem solving using planning. A number of planning algorithms such as logic-based planning, case-based planning, planning using means-ends analysis method have been described in this chapter. Broadly, planning falls under two categories: linear planning and

non-linear planning. The block world problem is chosen to describe working of planning methods. Linear planning is implemented using goal stack concept. Non-linear planning achieved by goal set, partial ordering, and constraint posting method is discussed in detail. Learning plans using triangle table method is also included.

Chapter 7 includes various knowledge representation techniques such as semantic network, extended semantic network, and Frames. Their capabilities of structuring knowledge and providing inheritance facility are the main features. Prototype implementations of these representations in Prolog along with inheritance functions are included. Extended semantic net combines the advantages of semantic net and predicate logic. This section is elaborated in detail with number of examples.

Chapter 8 discusses expert system design techniques and its applications. Phases in building expert system, its architecture and various sub modules of ES are discussed. Prototype of rule-based expert system in Prolog has been developed and included in this chapter. Various ES tools and shells are mentioned for the benefit of the reader. Blackboard system for handling diverse distributed knowledge and truth maintenance table are the other important features of this chapter.

Chapter 9 is on handling of uncertainty in the knowledge. Probability theory being the oldest gets special attention for uncertainty measure in this chapter. Rule-based expert system using probabilities in the rules and facts have been developed in Prolog. Bayesian belief network, certainty factor and Dempster–Shafer theories are other mechanisms to handle uncertainty in the knowledge. These have been described in detail.

Chapter 10 is on other important theory of handling fuzzy knowledge called as fuzzy logic which is based on fuzzy set. This chapter includes brief description of fuzzy set theory. Various membership functions are included in the chapter. Binary logic is extended to multi-valued logic and its relationship to fuzzy logic is established. Handling of fuzzy predicates, fuzzy truth values, linguistic variables and hedges are included in the chapter. Various types of fuzzy propositions and inference rules have also been discussed. An example of fuzzy system is included to give a feeling of fuzzy expert system where the rules are fuzzy in nature. A brief description of possibility theory, modal and temporal logic is included for the benefit of the reader to motivate him/her in this area.

Chapter 11 is on important area of AI named as machine learning. Various methods under supervised and unsupervised learning have been included. Inductive and deductive learning models are also described. Learning by clustering methods, support vector machines and case-based reasoning forms a substantial portion of the chapter.

Chapter 12 is on another learning model called artificial neural network. Various network architecture models namely, perceptron, multilayer feed forward network, radial basis function network, recurrent network along with Hopfield network are described in detail. Along with each type of network, their learning algorithms such as back propagation for FFNN, RBF network parameters learning have been included. Design issues such as data representation, network topology, network parameters for ANN are mentioned in the chapter. Boltzmann machines is also described in this chapter.

Chapter 13 is on very important and current area in AI. Evolutionary computation helps us solving optimization and NP-complete set of problems. Techniques under this category are called soft computing techniques which can be used where conventional computing or hard computing based on the concept of precise modelling and analyzing to yield accurate results do not work well. Evolutionary techniques mostly involve meta-heuristic optimization algorithms such as evolutionary algorithms (comprising genetic algorithms, evolutionary programming, etc.) and Swarm intelligence (comprising ant colony optimization and particle swarm optimization). We have included genetic algorithm and genetic programming concepts, swarm intelligence and ant colony optimization in detail.

Chapter 14 introduces intelligent agents. Overview of an agent and its working, different types of agents and their characteristics are described initially. Different architectures such as logic based, belief-desire-intension based and layered architectures have been included in the chapter. Agent communication languages are also described for the benefit of the reader. Multi-agent system is also included in the chapter which is very useful in distributed environment. Towards the end of this chapter, a multi-agent application for implementing bidding house framework proposed by one of the author's PhD student is also included to give feel to reader for building such system.

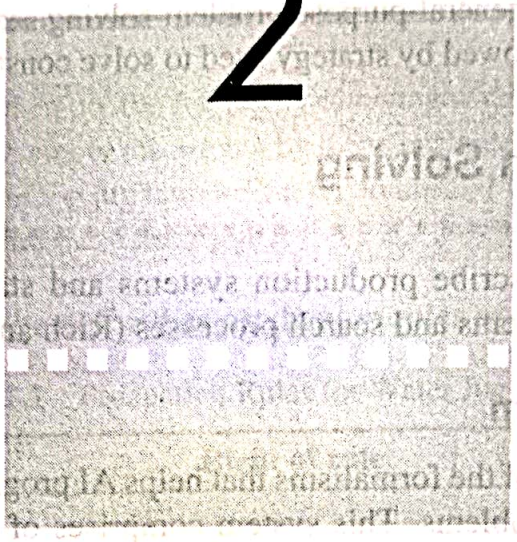
Chapter 15 is on conceptual dependency (CD)—an advanced level of knowledge representation at semantic level. Conceptual dependency primitives actions and category along with various rules for conceptualization blocks are dealt with in detail. CD is generally used for representing deep meaning of natural language sentences. Another method of storing knowledge is script structures which are predefined scenes of a particular situation. Case grammar that is intermediate representation of semantics of Natural Languages sentences is also included. Semantic web, resource description framework and ontology form integral and important part of this chapter.

Chapter 16 is the last chapter of the book which is on Natural language processing. It includes various syntactic models along with different types of parsers such as link parser, chart parser, recursive and augmented transition network (ATN) parsers. Definite clause grammar (DCG) which is a by product of Prolog has been described in detail. It helps us in capturing syntactic and semantic structures of Natural Language sentences. Various modules of DCG have been developed in Prolog. Case grammar is discussed in detail in this chapter. Universal Networking Language (electronic language) is the newest development that provides a uniform representation of

a natural language. It intermediates understanding among different natural languages and is represented in the form of logical expressions. UNL is similar to semantic networks for computers to express and exchange of information.

A bibliography is provided at the end of the book for the benefit of readers who wish to obtain further knowledge on the subject.

2



Problem Solving: State-Space Search and Control Strategies

2.1 Introduction

Problem solving is a method of deriving solution steps beginning from initial description of the problem to the desired solution. It has been conventionally one of the focus areas of Artificial Intelligence (AI) and can be characterized as a systematic search using a range of possible steps to achieve some predefined solution. The task is solved by a series of actions that minimizes the difference between the given situation and the desired goal. In AI, the problems are frequently modelled as a state space problem where the *state space* is a set of all possible *states* from start to goal states. The set of states form a graph in which two states are linked if there is an *operation* which can be executed to transform one state to other. While solving a problem, the state space is generated in the process of searching for its solution. There is difference between the state space search used in AI and the conventional computer science search methods. In case of the former, the states of the graph (search space) are generated as they are explored for the solution path and discarded thereafter, whereas in the traditional search method, searches are applied to the existing graphs. The typical state-space graph for solving problem is too large to be generated and stored in memory. The two types of problem-solving methods that are generally followed include general purpose and special-purpose methods. A *general-purpose method* is applicable to a wide variety of problems, whereas a *special-purpose method* is tailor-made for a particular problem and often exploits very specific features of the problem. The most general approach for solving a problem is to generate the solution and test it. For generating new state in the search space, an action/operator/rule is applied and tested whether the state is the goal state or not. In case the state is not the goal state, the procedure is repeated. The order of application of the rules to the current state is called *control strategy*. Since AI programs involve clean separation of computational components of operations, control, and data, it is useful to structure AI programs in such a way that it helps describe search process efficiently that forms the core of many intelligent processes.

This chapter introduces various general-purpose problem-solving search techniques, such as exhaustive and intelligent searches, followed by strategy used to solve constraint satisfaction problems.

2.2 General Problem Solving

The following subsections describe production systems and state space search methods that facilitate the modeling of problems and search processes (Rich and Knight, 2003).

2.2.1 Production System

Production system (PS) is one of the formalisms that helps AI programs to do search process more conveniently in state-space problems. This system comprises of start (initial) state(s) and goal (final) state(s) of the problem along with one or more databases consisting of suitable and necessary information for the particular task. Generally, knowledge representation schemes are used to structure information in these databases. PS consists of number of production rules in which each *production rule* has left side that determines the applicability of the rule and a right side that describes the action to be performed if the rule is applied. Left side of the rule is current state, whereas the right side describes the new state that is obtained from applying the rule. These production rules operate on the databases that change as these rules are applied.

PS also consists of control strategies (discussed later in detail) that specify the sequence in which the rules are applied when several rules match at once. One of the examples of PS is an Expert System which is used for expert opinion in a specific domain. In addition to usefulness of PS in describing search, following are other advantages of it as a formalism in AI:

1. It is a good way to model the strong state-driven nature of intelligent action.
2. New rules can be easily added to account for new situations without disturbing the rest of the system.
3. It is quite important in real-time environment and applications where new input to the database changes the behaviour of the system.

Let us consider an example of water jug problem and formulate production rules to solve this problem.

Water Jug Problem

Problem statement: We have two jugs, a 5-gallon (5-g) and the other 3-gallon (3-g) with no measuring marker on them. There is endless supply of water through tap. Our task is to get 4 gallon of water in the 5-g jug.

Solution: State space for this problem can be described as the set of ordered pairs of integers (X, Y) such that X represents the number of gallons of water in 5-g jug and Y for 3-g jug.

1. Start state is (0, 0)
2. Goal state is (4, N) for any value of $N \leq 3$.

The possible operations that can be used in this problem are listed as follows:

- Fill 5-g jug from the tap and empty the 5-g jug by throwing water down the drain
- Fill 3-g jug from the tap and empty the 3-g jug by throwing water down the drain
- Pour some or 3-g water from 5-g jug into the 3-g jug to make it full
- Pour some or full 3-g jug water into the 5-g jug

These operations can formally be defined as production rules as given in Table 2.1.

Table 2.1 Production Rules for Water Jug Problem

Rule No	Left of rule	Right of rule	Description
1	$(X, Y \mid X < 5)$	(5, Y)	Fill 5-g jug
2	$(X, Y \mid X > 0)$	(0, Y)	Empty 5-g jug
3	$(X, Y \mid Y < 3)$	(X, 3)	Fill 3-g jug
4	$(X, Y \mid Y > 0)$	(X,0)	Empty 3-g jug
5	$(X, Y \mid X + Y \leq 5 \wedge Y > 0)$	(X + Y, 0)	Empty 3-g into 5-g jug
6	$(X, Y \mid X + Y \leq 3 \wedge X > 0)$	(0, X + Y)	Empty 5-g into 3-g jug
7	$(X, Y \mid X + Y \geq 5 \wedge Y > 0)$	(5, Y – (5 – X)) until 5-g jug is full	Pour water from 3-g jug into 5-g jug
8	$(X, Y \mid X + Y \geq 3 \wedge X > 0)$	(X – (3 – Y), 3)	Pour water from 5-g jug into 3-g jug until 3-g jug is full

It should to be noted that there may be more than one solutions for a given problem. We have shown two possible solution paths as given in Table 2.2 and Table 2.3. We notice that solution-1 requires 6 steps as compared to solution-2 that requires 8 steps. In order to apply rules, we have to choose appropriate control strategy which is discussed later.

Table 2.2 Solution Path 1

Rule applied	5-g jug	3-g jug	Step No
Start state	0	0	
1	5	0	1
8	2	3	2
4	2	0	3
6	0	2	4
1	5	2	5
8	4	3	6
Goal state	4	–	

Table 2.3 Solution Path 2

Rule applied	5-g jug	3-g jug	Step No
Start state	0	0	
3	0	3	1
5	3	0	2
3	3	3	3
7	5	1	4
2	0	1	5
5	1	0	6
3	1	3	7
5	4	0	8
Goal state	4	–	

Let us consider another problem of 'Missionaries and Cannibals' and see how we can we solve this using production system.

Missionaries and Cannibals Problem

Problem Statement: Three missionaries and three cannibals want to cross a river. There is a boat on their side of the river that can be used by either one or two persons. How should they use this boat to cross the river in such a way that cannibals never outnumber missionaries on either side of the river? If the cannibals ever outnumber the missionaries (on either bank) then the missionaries will be eaten. How can they all cross over without anyone being eaten?

Solution: State space for this problem can be described as the set of ordered pairs of left and right banks of the river as (L, R) where each bank is represented as a list [nM, mC, B]. Here n is the number of missionaries M, m is the number of cannibals C, and B represents the boat.

1. Start state: ([3M, 3C, 1B], [0M, 0C, 0B]), 1B means that boat is present and 0B means it is absent.
2. Any state: ([n_1M, m_1C, _], [n_2M, m_2C, _]), with constraints/conditions at any state as $n_1 (\neq 0) \geq m_1$; $n_2 (\neq 0) \geq m_2$; $n_1 + n_2 = 3$, $m_1 + m_2 = 3$; boat can be either side.
3. Goal state: ([0M, 0C, 0B], [3M, 3C, 1B])

It should be noted that by no means, this representation is unique. In fact, one may have number of representations for the same problem. Table 2.4 consists of production rules based on the chosen representation. States on the left or right sides of river should be valid states satisfying the constraints given in (2) above.

One of possible solution path trace is given in the Table 2.5:

Table 2.4 Production Rules for Missionaries and Cannibals Problem

RN	Left side of rule	→	Right side of rule
	Rules for boat going from left bank to right bank of the river		
L1	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([(n_1-2)M, m_1C, 0B], [(n_2+2)M, m_2C, 1B])$
L2	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([(n_1-1)M,(m_1-1)C,0B],[(n_2+1)M,(m_2+1)C, 1B])$
L3	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([n_1M, (m_1-2)C, 0B], [n_2M, (m_2+2)C, 1B])$
L4	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([(n_1-1)M, m_1C, 0B],[(n_2+1)M, m_2C, 1B])$
L5	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([n_1M, (m_1-1)C, 0B], [n_2M, (m_2+1)C, 1B])$
	Rules for boat coming from right bank to left bank of the river		
R1	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([(n_1+2)M, m_1C, 1B], [(n_2-2)M, m_2C, 0B])$
R2	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([(n_1+1)M,(m_1+1)C,1B],[(n_2-1)M,(m_2-1)C, 0B])$
R3	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([n_1M, (m_1+2)C, 1B], [n_2M, (m_2-2)C, 0B])$
R4	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([(n_1+1)M, m_1C,1B],[(n_2-1)M, m_2C, 0B])$
R5	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([n_1M, (m_1+1)C, 1B], [n_2M, (m_2-1)C, 0B])$

Table 2.5 Solution Path

Rule number	**([3M, 3C, 1B], [0M, 0C, 0B]] ← Start State**
L2:	([2M, 2C, 0B], [1M, 1C, 1B])
R4:	([3M, 2C, 1B], [0M, 1C, 0B])
L3:	([3M, 0C, 0B], [0M, 3C, 1B])
R5:	([3M, 1C, 1B], [0M, 2C, 0B])
L1:	([1M, 1C, 0B], [2M, 2C, 1B])
R2:	([2M, 2C, 1B], [1M, 1C, 0B])
L1:	([0M, 2C, 0B], [3M, 1C, 1B])
R5:	([0M, 3C, 1B], [3M, 0C, 0B])
L3:	([0M, 1C, 0B], [3M, 2C, 1B])
R5:	([0M, 2C, 1B], [3M, 1C, 0B])
L3:	**([0M, 0C, 0B], [3M, 3C, 1B]) → Goal state**

2.2.2 State-Space Search

Similar to production system, state space is another method of problem representation that facilitates easy search. Using this method, one can also find a path from start state to goal state while solving a problem. A state space basically consists of four components:

1. A set S containing start states of the problem

2. A set G containing goal states of the problem
3. Set of nodes (states) in the graph/tree. Each node represents the state in problem-solving process.
4. Set of arcs connecting nodes. Each arc corresponds to operator that is a step in a problem-solving process.

A solution path is a path through the graph from a node in S to a node in G. The main objective of search algorithm is to determine a solution path in the graph. There may be more than one solution paths, as there may be more than one ways of solving the problem. One would exercise a choice between various solution paths based on some criteria of goodness or on some heuristic function. Commonly used approach is to apply appropriate operator to transfer one state of problem to another. It is similar to production system search method where we use production rules instead of operators. Let us consider again the problem of 'Missionaries and Cannibals'.

The possible operators that are applied in this problem are {2M0C, 1M1C, 0M2C, 1M0C, 0M1C}. Here M is missionary and C is cannibal. Digit before these characters indicates number of missionaries and cannibals possible at any point in time. These operators can be used in both the situations, i.e., if boat is on the left bank then, we write 'Operator →' and if the boat is on the right bank of the river, then we write "Operator ←".

For the sake of simplicity, let us represent state (L : R), where L = n_1M m_1C1B and R = n_2M m_2C0B. Here B represents boat with 1 or 0 indicating the presence or absence of the boat.

1. Start state: (3M3C1B : 0M0C0B) or simply (331:000)
2. Goal state: (0M0C0B : 3M3C1B) or simply (000:331)

Furthermore, we will filter out invalid states, illegal operators not applicable to some states, and some states that are not required at all. For example,

- An invalid state like (1M2C1B:2M1C0B) is not a possible state, as it leads to one missionary and two cannibals on the left bank.
- In case of a valid state like (2M2C1B:1M1C0B), the operator 0M1C or 0M2C would be illegal. Hence, the operators when applied should satisfy some conditions that should not lead to invalid state.
- Applying the same operator both ways would be a waste of time, since we have returned to a previous states. This is called *looping situation*. Looping may occur after few steps even. Such operations are to be avoided.

To illustrate the progress of search, we are required to develop a tree of nodes, with each node in the tree representing a state. The root node of the tree may be used to represent the start state. The arcs of the tree indicate the application of one of the operators. The nodes for which no operators

are applied, are called leaf nodes, which have no arcs leading from them. To simplify, we have not generated entire search space and avoided illegal and looping states. Depth-first or breadth-first strategy or some intelligent heuristic searches (explained later) is used to generate the search space. The search space generated using valid operators are shown in the Fig 2.1. The sequence of operators applied to solve this problem is given in Table 2.6.

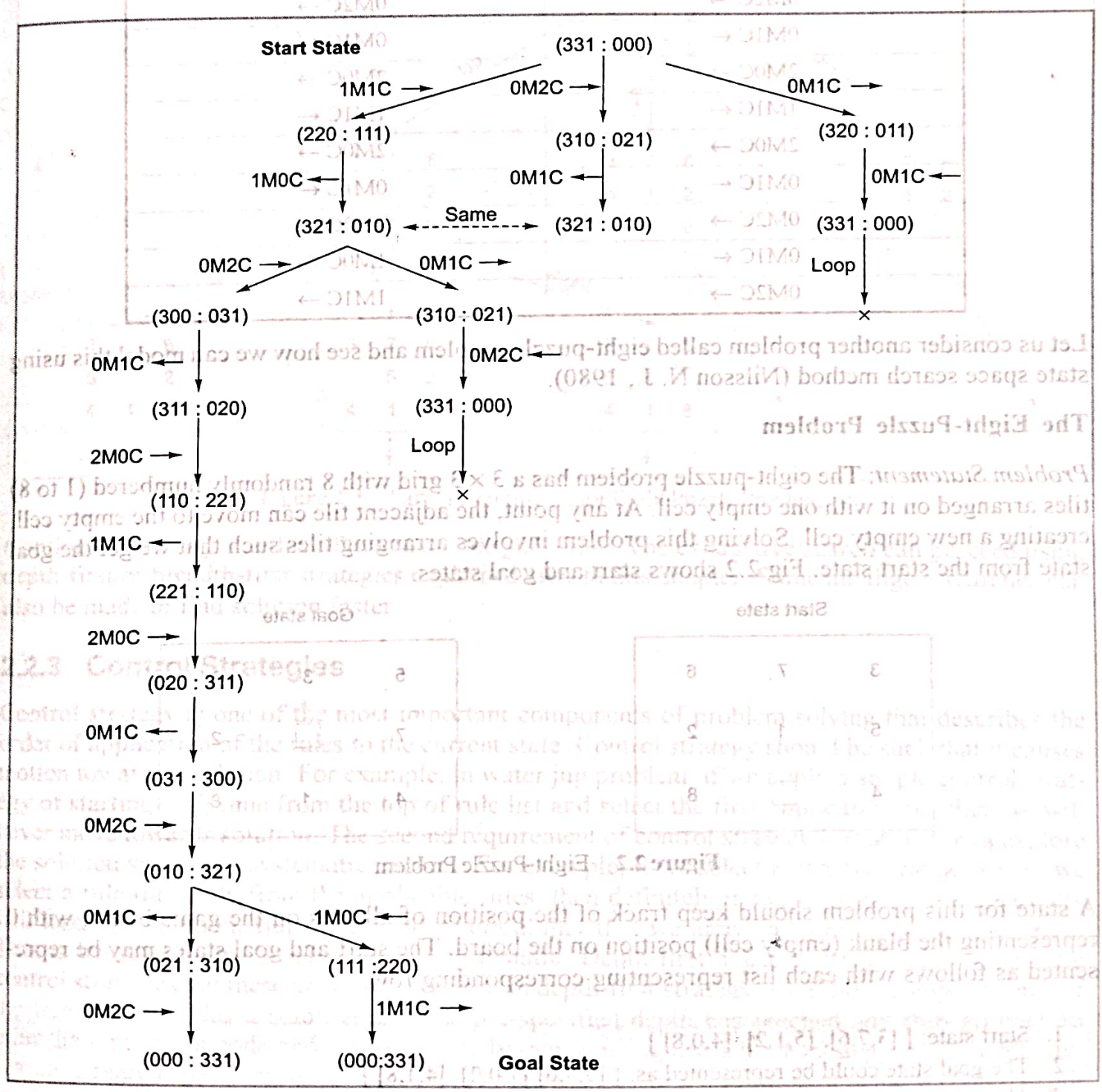


Figure 2.1 Search Space

Table 2.6 Two Solution Paths

Solution Path 1	Solution Path 2
1M1C →	1M1C →
1M0C ←	1M0C ←
0M2C →	0M2C →
0M1C ←	0M1C ←
2M0C →	2M0C →
1M1C ←	1M1C ←
2M0C →	2M0C →
0M1C ←	0M1C ←
0M2C →	0M2C →
0M1C ←	1M0C ←
0M2C →	1M1C →

Let us consider another problem called eight-puzzle problem and see how we can model this using state space search method (Nilsson N. J., 1980).

The Eight-Puzzle Problem

Problem Statement: The eight-puzzle problem has a 3 × 3 grid with 8 randomly numbered (1 to 8) tiles arranged on it with one empty cell. At any point, the adjacent tile can move to the empty cell, creating a new empty cell. Solving this problem involves arranging tiles such that we get the goal state from the start state. Fig 2.2 shows start and goal states.

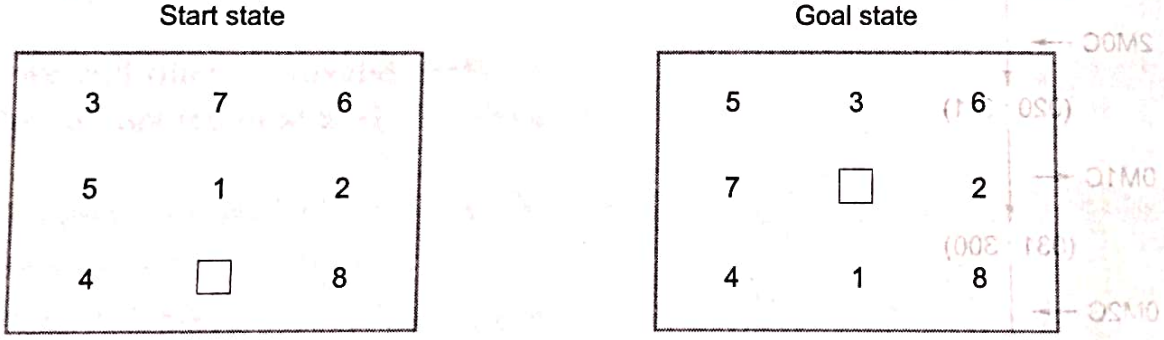


Figure 2.2 Eight-Puzzle Problem

A state for this problem should keep track of the position of all tiles on the game board, with 0 representing the blank (empty cell) position on the board. The start and goal states may be represented as follows with each list representing corresponding row:

1. Start state: [[3,7,6], [5,1,2], [4,0,8]]
2. The goal state could be represented as: [[5,3,6] [7,0,2], [4,1,8]]
3. The operators can be thought of moving {Up, Down, Left, Right}, the direction in which blank space effectively moves.

To simplify, a search tree up to level 2 is drawn as shown in Fig 2.3 to illustrate the use of operators to generate next state.

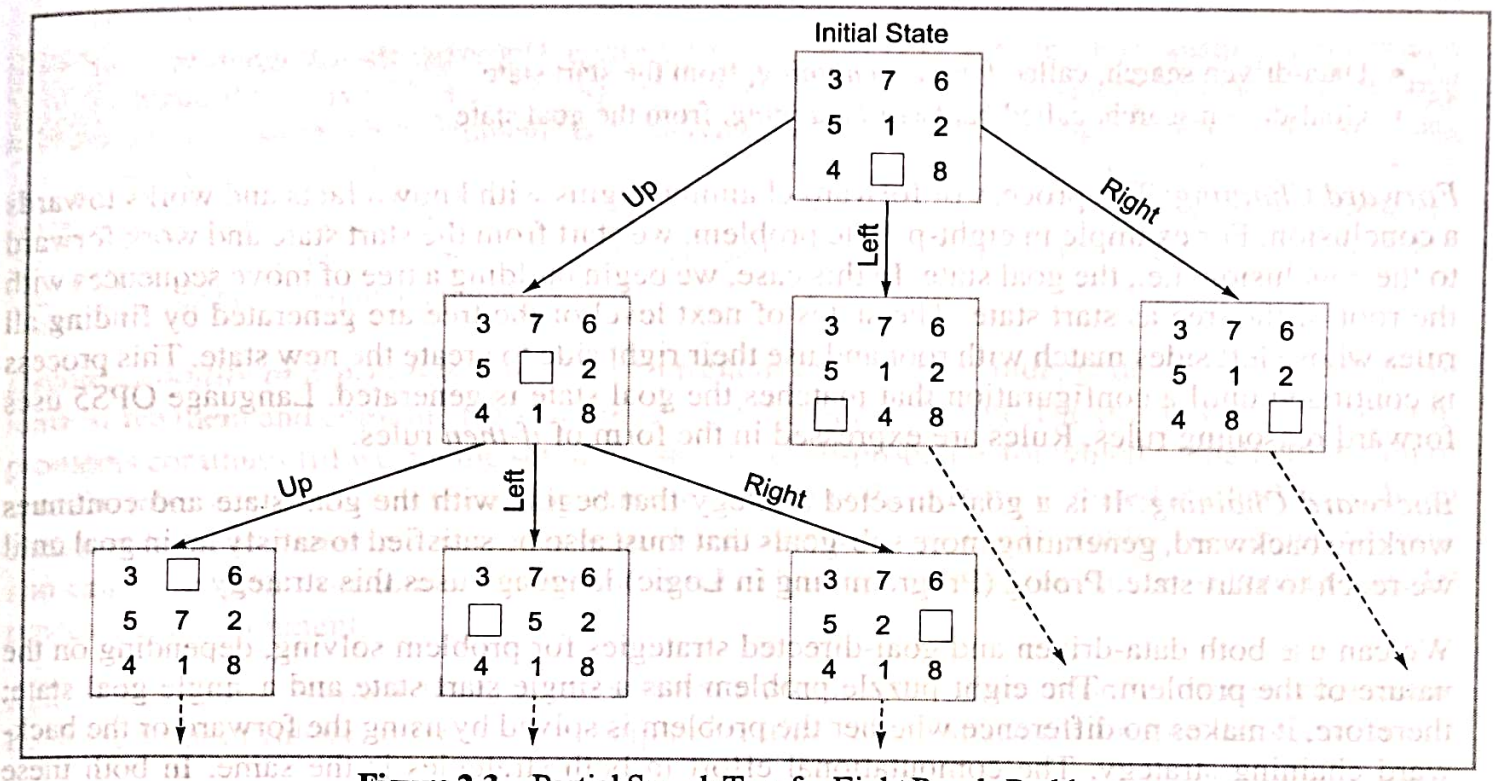


Figure 2.3 Partial Search Tree for Eight Puzzle Problem

Continue searching like this till we reach the goal state. The exhaustive search can proceed using depth-first or breadth-first strategies explained later in this chapter. Some intelligent searches can also be made to find solution faster.

2.2.3 Control Strategies

Control strategy is one of the most important components of problem solving that describes the order of application of the rules to the current state. Control strategy should be such that it causes motion towards a solution. For example, in water jug problem, if we apply a simple control strategy of starting each time from the top of rule list and select the first applicable one, then we will never move towards solution. The second requirement of control strategy is that it should explore the solution space in a systematic manner. For example, if we select a control strategy where we select a rule randomly from the applicable rules, then definitely it causes motion and eventually will lead to a solution. But there is every possibility that we arrive to same state several times. This is because control strategy is not systematic. Depth-first and breadth-first are systematic control strategies but these are blind searches. In depth-first strategy, we follow a single branch of the tree until it yields a solution or some pre-specified depth has reached and then go back to immediate previous node and explore other branches using depth-first strategy. In breadth-first search, a search space tree is generated level wise until we find a solution or some specified depth is reached. These strategies are exhaustive, uninformed, and blind searches in nature. If the problem is simple, then any control strategy that causes motion and is systematic will lead to a solution. However, to solve some real-world problems, effective control strategy must be used.

As mentioned earlier that the problem can be solved by searching for a solution. The main work in the area of search strategies is to find the correct search strategy for a given problem. There are two directions in which such a search could proceed.

- Data-driven search, called *forward chaining*, from the start state
- Goal-driven search, called *backward chaining*, from the goal state

Forward Chaining: The process of forward chaining begins with known facts and works towards a conclusion. For example in eight-puzzle problem, we start from the start state and work forward to the conclusion, i.e., the goal state. In this case, we begin building a tree of move sequences with the root of the tree as start state. The states of next level of the tree are generated by finding all rules whose left sides match with root and use their right side to create the new state. This process is continued until a configuration that matches the goal state is generated. Language OPS5 uses forward reasoning rules. Rules are expressed in the form of *if-then* rules.

Backward Chaining: It is a goal-directed strategy that begins with the goal state and continues working backward, generating more sub-goals that must also be satisfied to satisfy main goal until we reach to start state. Prolog (Programming in Logic) language uses this strategy.

We can use both data-driven and goal-directed strategies for problem solving, depending on the nature of the problem. The eight-puzzle problem has a single start state and a single goal state; therefore, it makes no difference whether the problem is solved by using the forward or the backward chaining strategy. The computational effort in both strategies is the same. In both these cases, same state space is searched but in different order. If there are large number of explicit goal states and one start state, then it would not be efficient to solve using backward chaining strategy, because we do not know which goal state is closest to the start state. So it is better to use forward chaining in such problems.

Therefore, the general observations are that move from the smaller set of states to the larger set of states and proceed in the direction with the lower branching factor (the average number of nodes that can be reached directly from single node). Let us consider an example to justify our argument. Suppose we have to prove a theorem in mathematics. We know that from small set of axioms, we can prove large number of theorems. On the other hand, the large number of theorems must go back to the small set of axioms. Here branching factor is significantly greater going forward from axioms to theorem rather than going from theorems to axioms. Therefore, proving theorem using backward strategy is more useful.

2.3 Characteristics of Problem

Before starting modelling the search and trying to find solution for the problem, one must analyze it along several key characteristics (Rich and Knight, 2003) initially. Some of these are mentioned below.

Type of Problem: There are three types of problems in real life, viz., Ignorable, Recoverable, and Irrecoverable.

- *Ignorable*: These are the problems where we can ignore the solution steps. For example, in proving a theorem, if some lemma is proved to prove a theorem and later on we realize that it is not useful, then we can ignore this solution step and prove another lemma. Such problems can be solved using simple control strategy.
- *Recoverable*: These are the problems where solution steps can be undone. For example, in water jug problem, if we have filled up the jug, we can empty it also. Any state can be reached again by undoing the steps. These problems are generally puzzles played by a single player. Such problems can be solved by backtracking, so control strategy can be implemented using a push-down stack.
- *Irrecoverable*: The problems where solution steps cannot be undone. For example, any two-player game such as chess, playing cards, snake and ladder, etc. are examples of this category. Such problems can be solved by planning process.

Decomposability of a problem: Divide the problem into a set of independent smaller sub-problems, solve them and combine the solutions to get the final solution. The process of dividing sub-problems continues till we get the set of the smallest sub-problems for which a small collection of specific rules are used. *Divide-and-conquer* technique is the commonly used method for solving such problems. It is an important and useful characteristic, as each sub-problem is simpler to solve and can be handed over to a different processor. Thus, such problems can be solved in parallel processing environment.

Role of knowledge: Knowledge plays an important role in solving any problem. Knowledge could be in the form of rules and facts which help generating search space for finding the solution.

Consistency of Knowledge Base used in solving problem: Make sure that knowledge base used to solve problem is consistent. Inconsistent knowledge base will lead to wrong solutions. For example, if we have knowledge in the form of rules and facts as follows:

If it is humid, it will rain. If it is sunny, then it is daytime. It is sunny day. It is nighttime.

This knowledge is not consistent as there is a contradiction because 'it is a daytime' can be deduced from the knowledge, and thus both 'it is night time' and 'it is a day time' are not possible at the same time. If knowledge base has such inconsistency, then some methods may be used to avoid such conflicts.

Requirement of solution: We should analyze the problem whether solution required is absolute or relative. We call solution to be *absolute* if we have to find exact solution, whereas it is *relative* if we have reasonably good and approximate solution. For example, in water jug problem, if there are more than one ways to solve a problem, then we follow one path successfully. There is no need to go back and find a better solution. In this case, the solution is absolute. In travelling salesman problem (discussed later), our goal is to find the shortest route. Unless all routes are known, it is difficult to know the shortest route. This is a best-path problem, whereas water jug is any-path problem. Any-path problem is generally solved in reasonable amount of time by using heuristics that suggest good paths to explore. Best-path problems are computationally harder compared with any-path problems.

2.4 Exhaustive Searches

Let us discuss some of systematic uninformed exhaustive searches, viz., breadth-first, depth-first, depth-first iterative deepening, and bidirectional searches, and present their algorithms.

2.4.1 Breadth-First Search

The breadth-first search (BFS) expands all the states one step away from the start state, and then expands all states two steps from start state, then three steps, etc., until a goal state is reached. All successor states are examined at the same depth before going deeper. The BFS always gives an optimal path or solution.

This search is implemented using two lists called OPEN and CLOSED. The OPEN list contains those states that are to be expanded and CLOSED list keeps track of states already expanded. Here OPEN list is maintained as a *queue* and CLOSED list as a *stack*. For the sake of simplicity, we are writing BFS algorithm for checking whether a goal node exists or not. Furthermore, this algorithm can be modified to get a path from start to goal nodes by maintaining CLOSED list with pointer back to its parent in the search tree.

Algorithm *(BFS)*

Input: START and GOAL states
Local Variables: OPEN, CLOSED, STATE-X, SUCCs, FOUND;
Output: Yes or No
Method:

- initialize OPEN list with START and CLOSED = ϕ;
- FOUND = false;
- while (OPEN $\neq \phi$ and FOUND = false) do
  {
  - remove the first state from OPEN and call it STATE-X;
  - put STATE-X in the front of CLOSED list {maintained as stack};
  - if STATE-X = GOAL then FOUND = true else
    {
    - perform EXPAND operation on STATE-X, producing a list of SUCCs;
    - remove from successors those states, if any, that are in the CLOSED list;
    - append SUCCs at the end of the OPEN list; /*queue*/

    }
  } /* end while */
- if FOUND = true then return ***Yes*** else return ***No***
- Stop

Let us see the search tree generation from start state of the water jug problem using BFS algorithm. At each state, we apply first applicable rule. If it generates previously generated state then cross it and try another rule in the sequence to avoid the looping. If new state is generated then expand this state in breadth-first fashion. The rules given in Table 2.1 for water jug problem are applied and enclosed in {}. Figure 2.4 shows the trace of search tree using BFS.

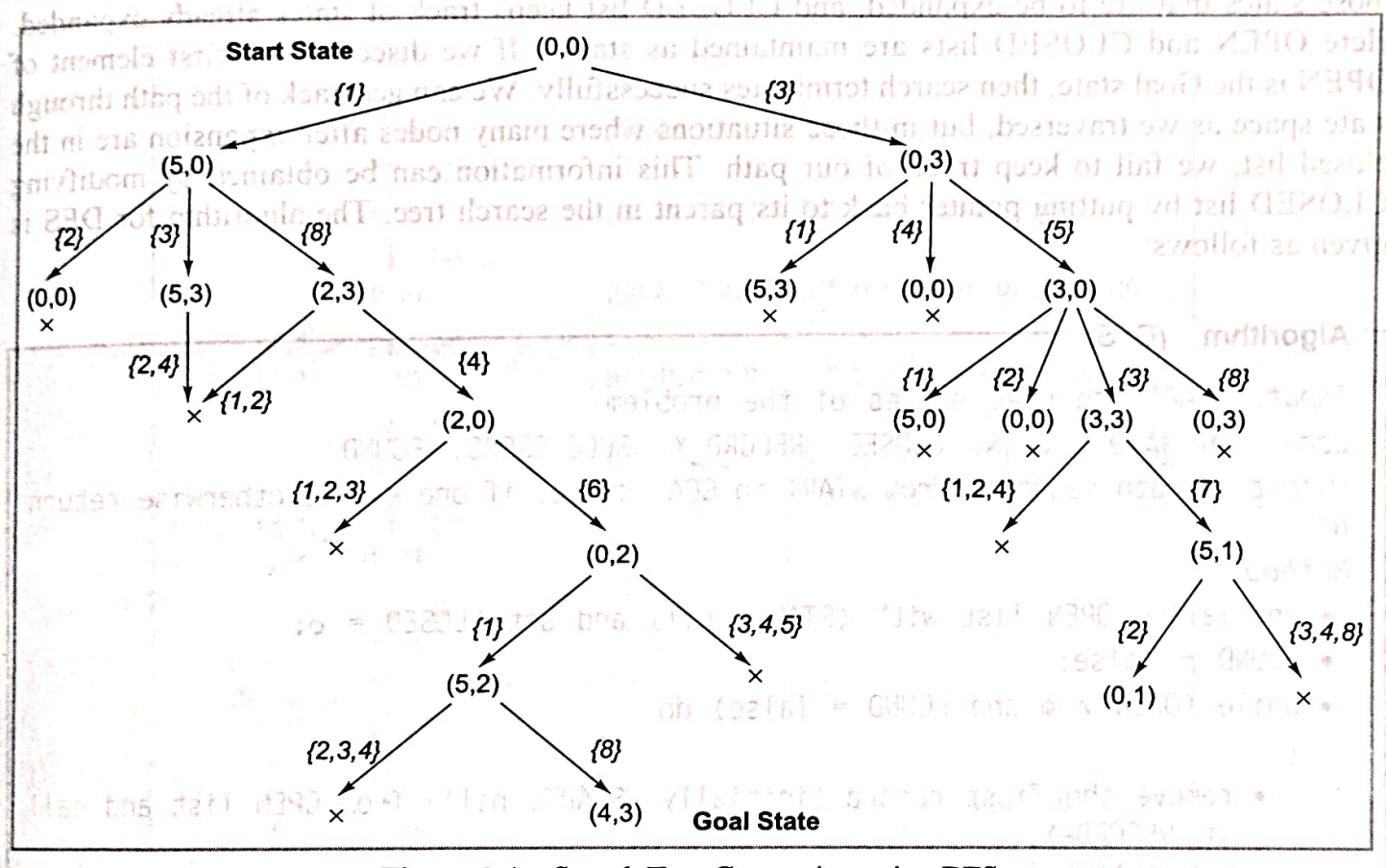


Figure 2.4 Search Tree Generation using BFS

Search tree is developed level wise. This is not memory efficient as partially developed tree is to be kept in the memory but it finds optimal solution or path. We can easily see the path from start to goal by tracing the tree from goal state to start state through parent link. This path is optimal and we cannot get a path shorter than this.

Solution path: (0,0) → (5,0) → (2,3) → (2,0) → (0,2) → (5,2) → (4,3)

The path information can be obtained by modifying CLOSED list in the algorithm by putting pointer back to its parent.

2.4.2 Depth-First Search

In the depth-first search (DFS), we go as far down as possible into the search tree/graph before backing up and trying alternatives. It works by always generating a descendent of the most

recently expanded node until some depth cut off is reached and then backtracks to next most recently expanded node and generates one of its descendants. DFS is memory efficient, as it only stores a single path from the root to leaf node along with the remaining unexpanded siblings for each node on the path.

We can implement DFS by using two lists called OPEN and CLOSED. The OPEN list contains those states that are to be expanded, and CLOSED list keeps track of states already expanded. Here OPEN and CLOSED lists are maintained as stacks. If we discover that first element of OPEN is the Goal state, then search terminates successfully. We can get track of the path through state space as we traversed, but in those situations where many nodes after expansion are in the closed list, we fail to keep track of our path. This information can be obtained by modifying CLOSED list by putting pointer back to its parent in the search tree. The algorithm for DFS is given as follows:

Algorithm *(DFS)*

Input: START and GOAL states of the problem
Local Variables: OPEN, CLOSED, RECORD_X, SUCCESSORS, FOUND
Output: A path sequence from START to GOAL state, if one exists otherwise return No
Method:

- initialize OPEN list with (START, nil) and set CLOSED = ϕ;
- FOUND = false;
- while (OPEN $\neq \phi$ and FOUND = false) do
  {
  - remove the first record (initially (START, nil)) from OPEN list and call it RECORD-X;
  - put RECORD-X in the front of CLOSED list (maintained as stack);
  - if (STATE_X of RECORD_X= GOAL) then FOUND = true else
  {
    - perform EXPAND operation on STATE-X producing a list of records called SUCCESSORS; create each record by associating parent link with its state;
    - remove from SUCCESSORS any record that is already in the CLOSED list;
    - insert SUCCESSORS in the front of the OPEN list /* Stack */

  }
  }/* end while */
- if FOUND = true then return the path by tracing through the pointers to the parents on the CLOSED list else return ***No***
- Stop

Let us see the search tree generation from start state of the water jug problem using DFS algorithm.

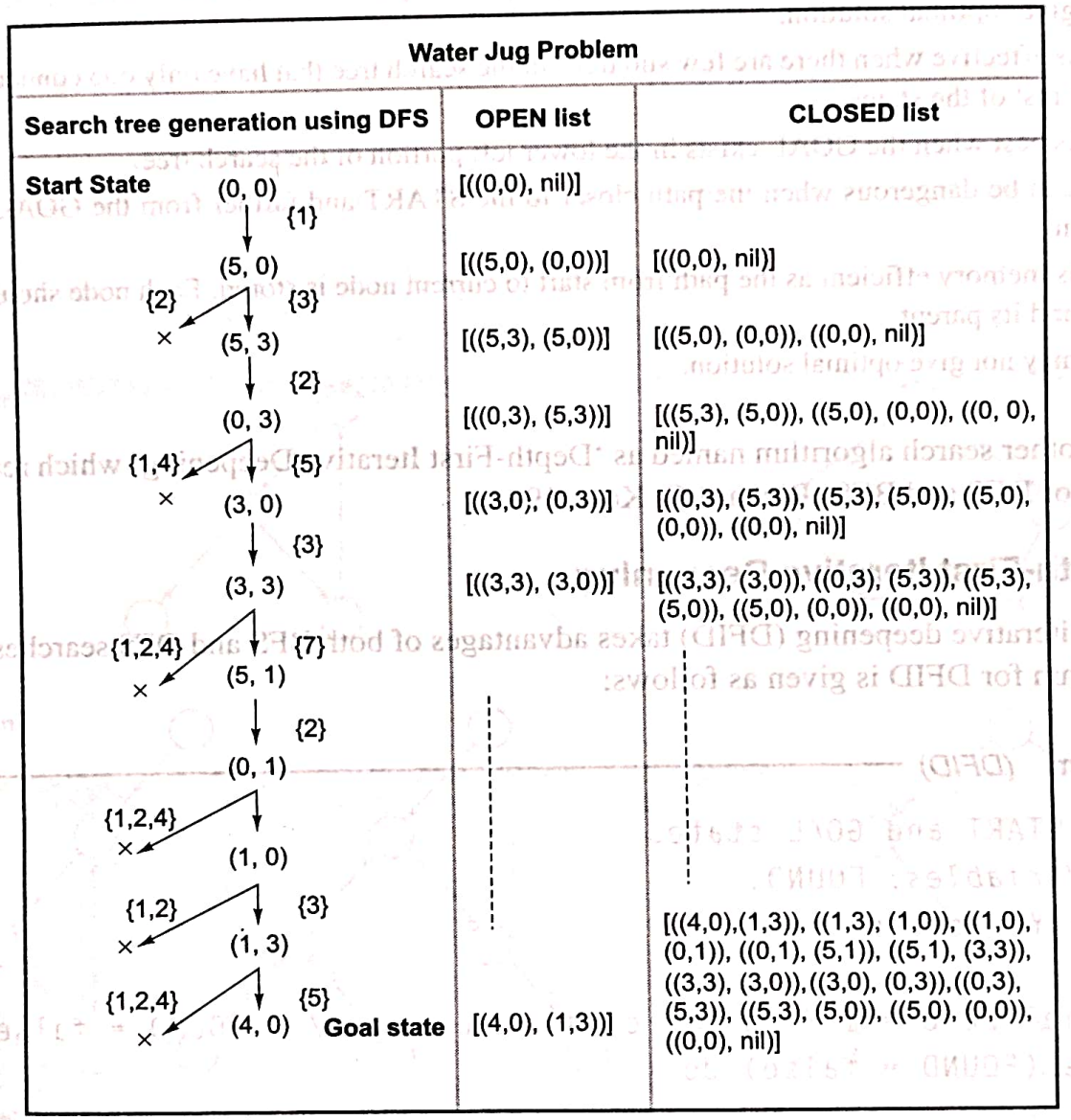


Figure 2.5 Search Tree Generation using DFS

The path is obtained from the list stored in CLOSED. The solution Path is

$(0,0) \rightarrow (5,0) \rightarrow (5,3) \rightarrow (0,3) \rightarrow (3,0) \rightarrow (3,3) \rightarrow (5,1) \rightarrow (0, 1) \rightarrow (1,0) \rightarrow (1,3) \rightarrow (4,0)$

Comparisons: Since these are unguided, blind, and exhaustive searches, we cannot say much about them but can make some observations.

- BFS is effective when the search tree has a low branching factor.
- BFS can work even in trees that are infinitely deep.

- BFS requires a lot of memory as number of nodes in level of the tree increases exponentially.
- BFS is superior when the GOAL exists in the upper right portion of a search tree.
- BFS gives optimal solution.
- DFS is effective when there are few sub trees in the search tree that have only one connection point to the rest of the states.
- DFS is best when the GOAL exists in the lower left portion of the search tree.
- DFS can be dangerous when the path closer to the START and farther from the GOAL has been chosen.
- DFS is memory efficient as the path from start to current node is stored. Each node should contain state and its parent.
- DFS may not give optimal solution.

There is another search algorithm named as 'Depth-First Iterative Deepening' which removes the drawbacks of DFS and BFS (Richard G. Korf, 1985)

2.4.3 Depth-First Iterative Deepening

Depth-first iterative deepening (DFID) takes advantages of both BFS and DFS searches on trees. The algorithm for DFID is given as follows:

Algorithm *(DFID)*

```
Input: START and GOAL states
Local Variables: FOUND;
Output: Yes or No
Method:
• initialize d = 1 /* depth of search tree */ , FOUND = false
• while (FOUND = false) do
  {
   • perform a depth first search from start to depth d.
   • if goal state is obtained then FOUND = true else discard the
     nodes generated in the search of depth d
   • d = d + 1
  } /* end while */
• if FOUND = true then return Yes otherwise return No
• Stop
```

Since DFID expands all nodes at a given depth before expanding any nodes at greater depth, it is guaranteed to find a shortest path or optimal solution from start to goal state. The working of DFID algorithm is shown in Fig. 2.6 as given below:

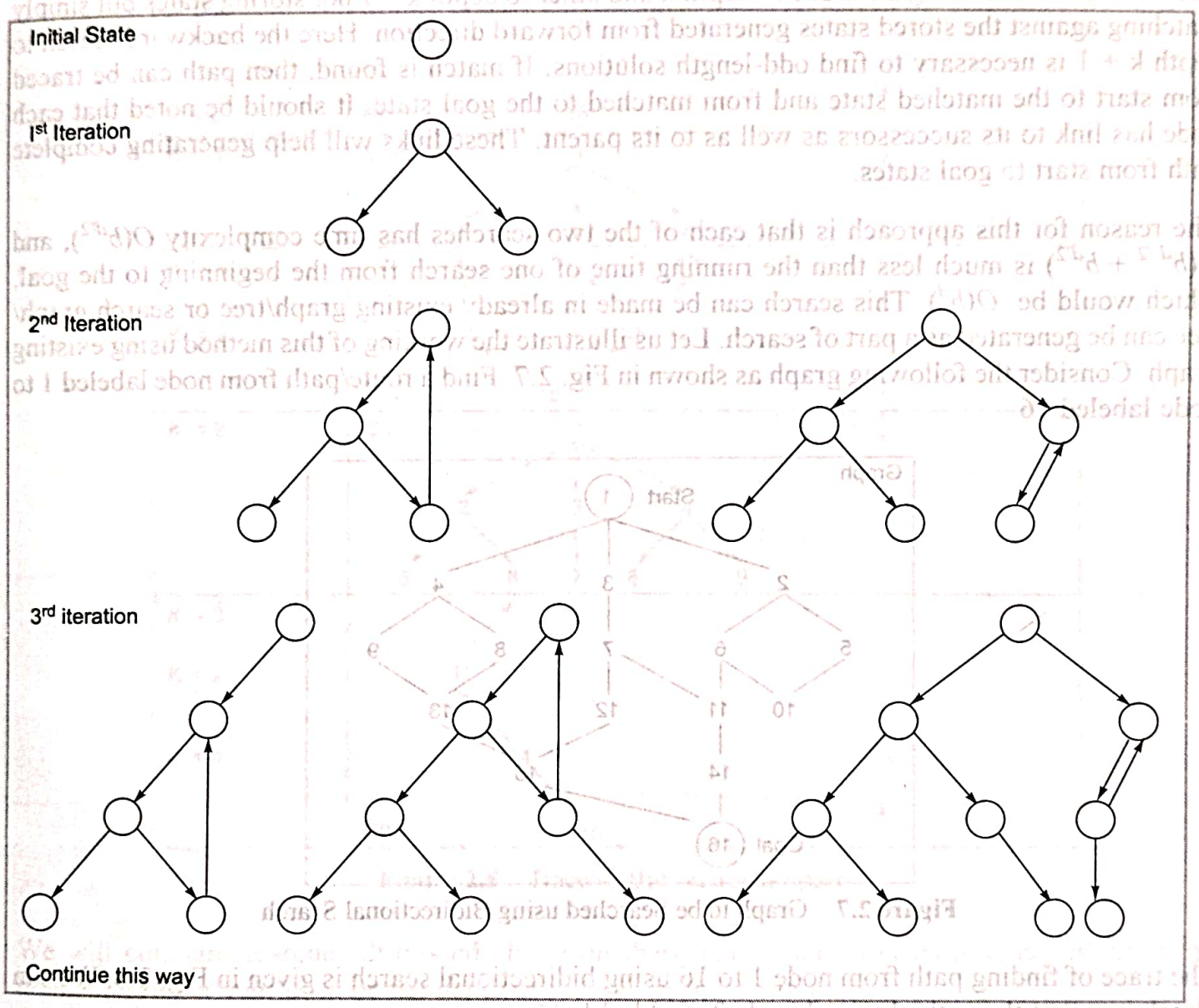


Figure 2.6 Search Tree Generation using DFID

At any given time, it is performing a DFS and never searches deeper than depth 'd'. Thus, the space it uses is $O(d)$. Disadvantage of DFID is that it performs wasted computation before reaching the goal depth.

2.4.4 Bidirectional Search

Bidirectional search is a graph search algorithm that runs two simultaneous searches. One search moves forward from the start state and other moves backward from the goal and stops when the

two meet in the middle. It is useful for those problems which have a single start state and single goal state. The DFID can be applied to bidirectional search for k = 1, 2, ... The kth iteration consists of generating all states in the forward direction from start state up to depth k using BFS, and from goal state using DFS one to depth k and other to depth k + 1 not storing states but simply matching against the stored states generated from forward direction. Here the backward search to depth k + 1 is necessary to find odd-length solutions. If match is found, then path can be traced from start to the matched state and from matched to the goal state. It should be noted that each node has link to its successors as well as to its parent. These links will help generating complete path from start to goal states.

The reason for this approach is that each of the two searches has time complexity $O(b^{d/2})$, and $O(b^{d/2} + b^{d/2})$ is much less than the running time of one search from the beginning to the goal, which would be $O(b^d)$. This search can be made in already existing graph/tree or search graph/tree can be generated as a part of search. Let us illustrate the working of this method using existing graph. Consider the following graph as shown in Fig. 2.7. Find a route/path from node labeled 1 to node labeled 16.

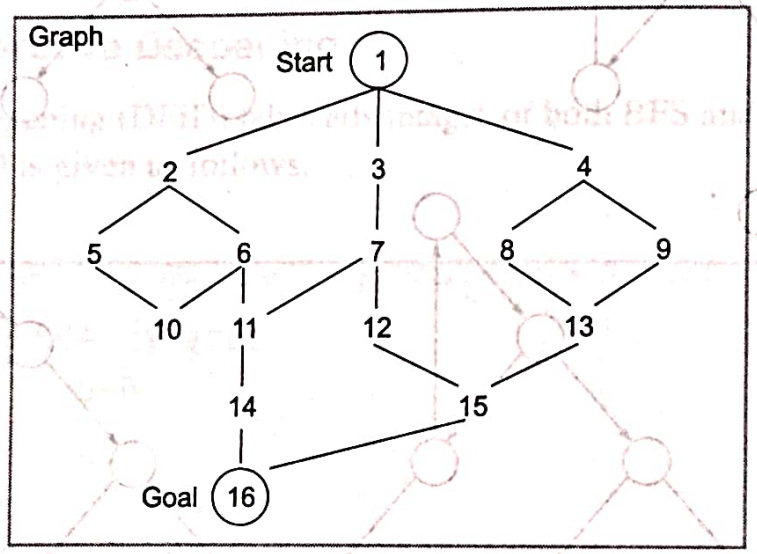


Figure 2.7 Graph to be Searched using Bidirectional Search

The trace of finding path from node 1 to 16 using bidirectional search is given in Fig. 2.8. We can clearly see that the path obtained is: 1, 2, 6, 11, 14, 16

2.4.5 Analysis of Search methods

Effectiveness of any search strategy in problem solving is measured in terms of:

- *Completeness*: Completeness means that an algorithm guarantees a solution if it exists.
- *Time Complexity*: Time required by an algorithm to find a solution.
- *Space Complexity*: Space required by an algorithm to find a solution.
- *Optimality*: The algorithm is optimal if it finds the highest quality solution when there are several different solutions for the problem.

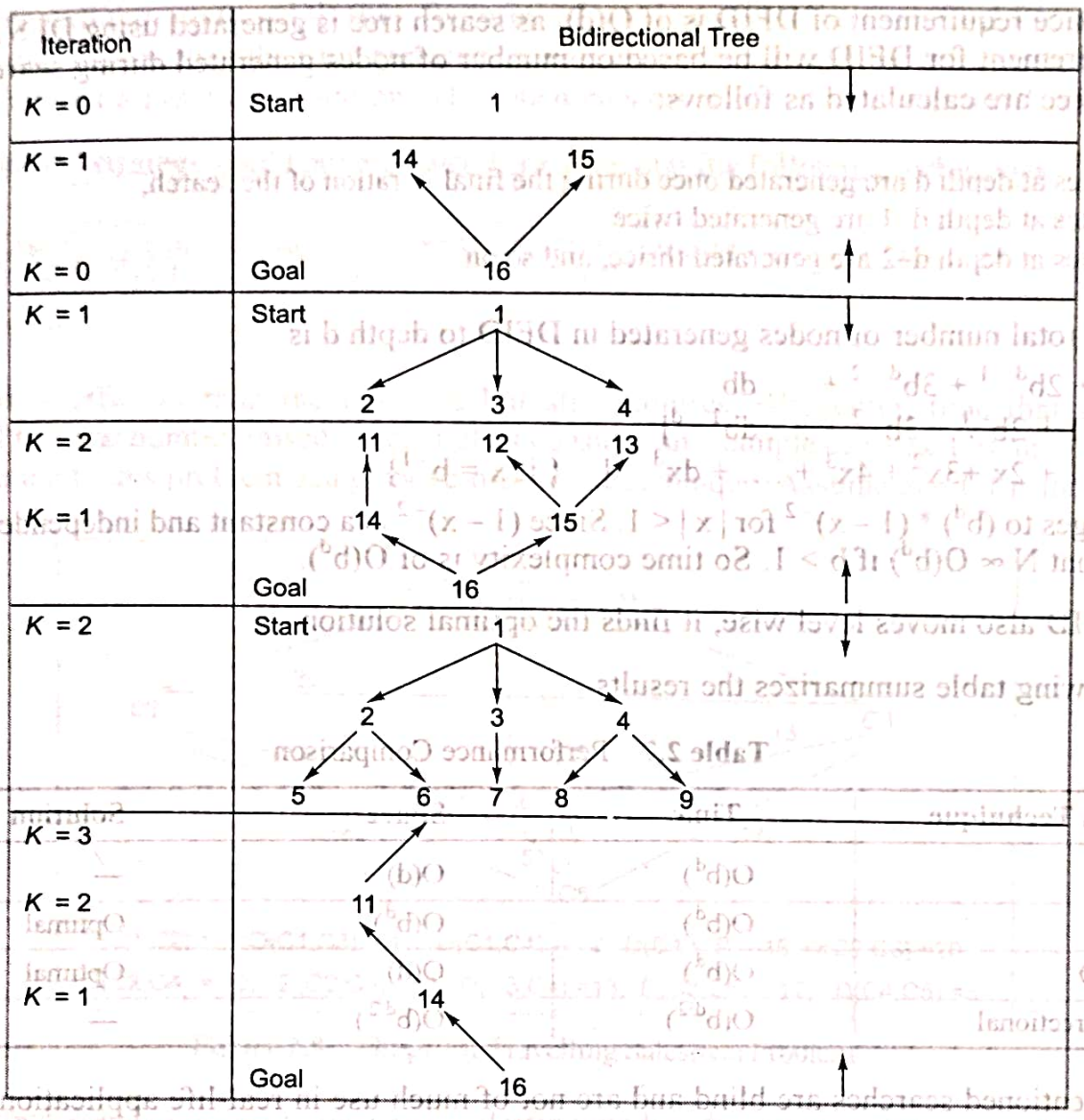


Figure 2.8 Trace of Bidirectional Space

We will compare searches discussed above on these parameters. Let us assume 'b' to be the branching factor and 'd' to be the depth of the tree in the worst case.

DFS: If the depth cut off is 'd', then the space requirement is of O(d). The time complexity of DFS to depth 'd' is of $O(b^d)$ in the worst case. The DFS requires some cut-off depth. If branches are not cut off and duplicates are not checked for, the algorithm may not even terminate. We can not say on optimality of solution.

BFS: In BFS, all the nodes up to depth '*d*' must be generated in the worst case. At level *i* there will be b^i nodes generated. So, total number of nodes generated in the worst case is

$$1 + b + b^2 + b^3 + \ldots + b^{d-1} \cong O(b^d)$$

Space complexity in the worst case is also $O(b^d)$. The solution obtained using BFS is optimal but it may take higher computational time. It will terminate and find solution if it exists.

DFID: Space requirement of DFID is of O(d), as search tree is generated using DFS method. The time requirement for DFID will be based on number of nodes generated during search. The total nodes in tree are calculated as follows:

- Nodes at depth d are generated once during the final iteration of the search,
- Nodes at depth d–1 are generated twice,
- Nodes at depth d–2 are generated thrice, and so on.

Thus, the total number of nodes generated in DFID to depth d is

$$N = b^d + 2b^{d-1} + 3b^{d-2} + \ldots. db$$
$$= b^d [1 + 2b^{-1} + 3b^{-2} + \ldots. db^{1-d}]$$
$$= b^d [1 + 2x + 3x^2 + 4x^3 + \ldots. + dx^{d-1}] \quad \{ \text{if } x = b^{-1} \}$$

N converges to $(b^d) * (1 - x)^{-2}$ for $|x| < 1$. Since $(1 - x)^{-2}$ is a constant and independent of 'd', we can say that $N \propto O(b^d)$ if $b > 1$. So time complexity is of $O(b^d)$.

Since DFID also moves level wise, it finds the optimal solution.

The following table summarizes the results.

Table 2.7 Performance Comparison

Search Technique	Time	Space	Solution
DFS	$O(b^d)$	O(d)	—
BFS	$O(b^d)$	$O(b^d)$	Optimal
DFID	$O(b^d)$	O(d)	Optimal
Bi-directional	$O(b^{d/2})$	$O(b^{d/2})$	—

Above-mentioned searches are blind and are not of much use in real-life applications. There are problems where combinatorial explosion takes place as the size of the search tree increases, such as travelling salesman problem. We need to have some intelligent searches which take into account some relevant problem information and finds solutions faster.

To illustrate the need of intelligent searches, let us consider a problem of travelling salesman.

Travelling Salesman Problem

Statement: In travelling salesman problem (TSP), one is required to find the shortest route of visiting all the cities once and returning back to starting point. Assume that there are 'n' cities and the distance between each pair of the cities is given.

The problem seems to be simple, but deceptive. The TSP is one of the most intensely studied problems in computational mathematics and yet no effective solution method is known for the general case.

In this problem, a simple motion causing and systematic control strategy could, in principle, be applied to solve it. All possible paths of the search tree are explored and the shortest path is

returned. This will require (n – 1)! (i.e., factorial of n – 1) paths to be examined for 'n' cities. If number of cities grows, then the time required to wait a salesman to get the information about the shortest path is not a practical situation. This phenomenon is called *combinatorial explosion.*

Above-mentioned strategy could be improved little bit using the following techniques.

- Start generating complete paths, keeping track of the shortest path found so far.
- Stop exploring any path as soon as its partial length becomes greater than the shortest path length found so far.

This method is efficient than the first one but still requires exponential time that is directly proportional to some number raised to 'n'. Let us consider an example of five cities and see how we can find solution to this problem using above-mentioned technique. Assume that C1 is the start city.

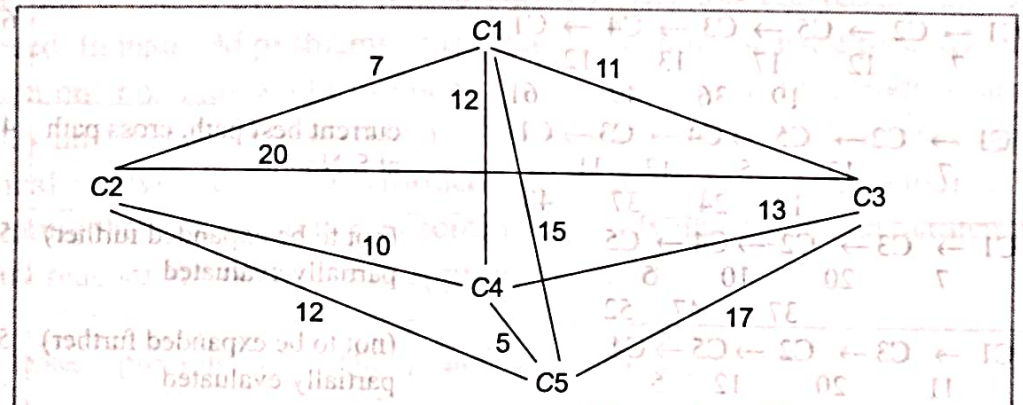


Figure 2.9 Graph for Travelling Salesman Problem

Table 2.8 shows some possible paths generated using modified approach upto some level. Some of the partial paths are pruned if the distance computed is less than minimum computed distance so far between any pair of cities. Initially, first complete path is taken to be the minimum and √ is put along with the distance, and if the distance (full or partial) is greater than the previously calculated minimum, then × is put to show pruning of that path.

Continue till all the paths have been explored. In this case, there will be 4! = 24 possible paths. We notice that out of 13 paths shown in Table 2.8, 5 paths are partially evaluated. But still it requires exponential time.

Therefore, some kind of thumb rules or heuristic techniques may be thought of and applied. Furthermore, there may be more than one ways of solving the problem and one would like to exercise a choice between various solution paths based on some criteria of goodness or on some thumb rule. The following sections describe various heuristic search techniques.

So we need some intelligent methods which can make use of problem knowledge and improves the search time substantially.

Table 2.8 Performance Comparison

Paths explored. Assume C1 to be the start city		Distance
1. C1 → C2 → C3 → C4 → C5 → C1 7 20 13 5 15 27 40 45 60	current best path	60 √ ×
2. C1 → C2 → C3 → C5 → C4 → C1 7 20 17 5 12 27 44 49 61		61 ×
3. C1 → C2 → C4 → C3 → C5 → C1 7 10 13 17 15 17 40 57 72		72 ×
4. C1 → C2 → C4 → C5 → C3 → C1 7 10 5 17 11 17 22 39 50	current best path, cross path at S.No 1.	50 √ ×
5. C1 → C2 → C5 → C3 → C4 → C1 7 12 17 13 12 19 36 49 61		61 ×
6. C1 → C2 → C5 → C4 → C3 → C1 7 12 5 13 11 19 24 37 48	current best path, cross path at S.No 4.	48 √
7. C1 → C3 → C2 → C4 → C5 7 20 10 5 37 47 52	(not to be expanded further) partially evaluated	52 ×
8. C1 → C3 → C2 → C5 → C4 11 20 12 5 37 49 54	(not to be expanded further) partially evaluated	54 ×
9. C1 → C3 → C4 → C2 → C5 → C1 11 13 10 12 15 24 34 46 61		61 ×
10. C1 → C3 → C4 → C5 → C2 → C1 11 13 5 12 7 24 29 41 48	same as current best path at S. No. 6.	48 √
11. C1 → C3 → C5 → C2 11 17 12 38 50	(not to be expanded further) partially evaluated	50 ×
12. C1 → C3 → C5 → C4 → C2 11 17 5 10 38 43 53	(not to be expanded further) partially evaluated	53 ×
13. C1 → C4 → C2 → C3 → C5 12 10 20 17 22 42 55	(not to be expanded further) partially evaluated	59 ×
Continue like this		

2.5 Heuristic Search Techniques

Heuristic technique is a criterion for determining which among several alternatives will be the most effective to achieve some goal. This technique improves the efficiency of a search process

possibly by sacrificing claims of systematic and completeness. It no longer guarantees to find the best solution but almost always finds a very good solution. Using good heuristics, we can hope to get good solution to hard problems (such as travelling salesman problem) in less than exponential time. There are two types of heuristics, namely,

- *General-purpose* heuristics that are useful in various problem domains.
- *Special purpose* heuristics that are domain specific.

2.5.1 General-Purpose Heuristics

A general-purpose heuristics for combinatorial problem is *nearest neighbor algorithms* that work by selecting the locally superior alternative. For such algorithms, it is often possible to prove an upper bound on the error. It provides reassurance that we are not paying too high a price in accuracy for speed. In many AI problems, it is often difficult to measure precisely the goodness of a particular solution. For real-world problems, it is often useful to introduce heuristics on the basis of relatively unstructured knowledge. It is impossible to define this knowledge in such a way that mathematical analysis can be performed. In AI approaches, behaviour of algorithms is analyzed by running them on computer as contrast to analyzing algorithm mathematically. There are at least many reasons for such ad hoc approaches in AI.

- It is a fun to see a program do something intelligent than to prove it.
- Since AI problem domains are usually complex, it is generally not possible to produce analytical proof that a procedure will work.
- It is not even possible to describe the range of problems well enough to make statistical analysis of program behaviour meaningful.

However, it is important to keep performance in mind while designing algorithms. One of the most important analysis of the search process is to find number of nodes in a complete search tree of depth 'd' and branching factor 'f', that is, f * d. This simple analysis motivates to look for improvements on the exhaustive searches and to find an upper bound on the search time which can be compared with exhaustive search procedures. The searches which use some domain knowledge are called *Informed Search Strategies*.

2.5.2 Branch and Bound Search (Uniform Cost Search)

In branch and bound search method, cost function (denoted by g(X)) is designed that assigns cumulative expense to the path from *start node* to the current node X by applying the sequence of operators. While generating a search space, a least cost path obtained so far is expanded at each iteration till we reach to goal state. Since branch and bound search expands the least-cost partial path, it is sometimes also called a uniform cost search. For example, in travelling salesman problem, g(X) may be the actual distance travelled from Start to current node X.

During search process, there can be many incomplete paths contending for further consideration. The shortest one is always extended one level further, creating as many new incomplete paths as there are branches. These new paths along with old ones are sorted on the values of cost function 'g' and again the shortest path is extended. Since the shortest path is always chosen for extension, the path first reaching to the goal is certain to be optimal but it is not guaranteed to find the solution quickly. The following algorithm is simple to give you an idea about how it works. Furthermore, it can be modified by putting parent links along with the node in the CLOSED list.

Algorithm *(Branch and Bound)*

Input: START and GOAL states
Local Variables: OPEN, CLOSED, NODE, SUCCs, FOUND;
Output: Yes or No
Method:

- initially store the start node with g(root) = 0 in a OPEN list; CLOSED = ϕ; FOUND = false;
- while (OPEN $\neq \phi$ and FOUND = false) do
  {
  - remove the top element from OPEN list and call it NODE;
  - if NODE is the goal node, then FOUND = true else
    {
    - put NODE in CLOSED list;
    - find SUCCs of NODE, if any, and compute their 'g' values and store them in OPEN list;
    - sort all the nodes in the OPEN list based on their cost-function values;

    }
  } /* end while */
- if FOUND = true then return *Yes* otherwise return *No*;
- Stop

In branch and bound method, if g(X) = 1 for all operators, then it degenerates to simple breadth-first search. From AI point of view, it is as bad as depth first and breadth first. This can be improved if we augment it by dynamic programming, that is, delete those paths which are redundant. We notice that algorithm generally requires generate solution and test it for its goodness. Solution can be generated using any method and testing might be based on some heuristics. Skeleton of algorithm for 'generate and test' strategy is as follows:

Algorithm *Generate and Test Algorithm*

```
Start
• Generate a possible solution
• Test if it is a goal.
• If not go to start else quit
End
```

2.5.3 Hill Climbing

Quality Measurement turns Depth-First search into Hill climbing (variant of generate and test strategy). It is an optimization technique that belongs to the family of local searches. It is a relatively simple technique to implement as a popular first choice is explored. Although more advanced algorithms may give better results, there are situations where hill climbing works well. Hill climbing can be used to solve problems that have many solutions but where some solutions are better than others. Travelling salesman problem can be solved with hill climbing. It is easy to find a solution that will visit all the cities, but this solution will probably be very bad compared to the optimal solution. If there is some way of ordering the choices so that the most promising node is explored first, then search efficiency may be improved. Moving through a tree of paths, hill climbing proceeds in depth-first order, but the choices are ordered according to some heuristic value (i.e., measure of remaining cost from current to goal state). For example, in travelling salesman problem, straight line (as the crow flies) distance between two cities can be a heuristic measure of remaining distance.

Algorithm *(Simple Hill Climbing)*

```
Input: START and GOAL states
Local Variables: OPEN, NODE, SUCCs, FOUND;
Output: Yes or No
Method:
  • store initially the start node in a OPEN list (maintained as stack); FOUND
    = false;
  • while (OPEN ≠ empty and Found = false) do
    {
    • remove the top element from OPEN list and call it NODE;
    • if NODE is the goal node, then FOUND = true  else
      • find SUCCs of NODE, if any;
      • sort SUCCs by estimated cost from NODE to goal state and add them to the
        front of OPEN list;
    } /* end while */
  • if FOUND = true then return Yes otherwise return No;
  • Stop
```

Problems with hill climbing

There are few problems with hill climbing. The search process may reach to a position that is not a solution but from there no move improves the situation. This will happen if we have reached a local maximum, a plateau, or a ridge.

Local maximum: It is a state that is better than all its neighbours but not better than some other states which are far away. From this state all moves looks to be worse. In such situation, backtrack to some earlier state and try going in different direction to find a solution.

Plateau: It is a flat area of the search space where all neighbouring states has the same value. It is not possible to determine the best direction. In such situation make a big jump to some direction and try to get to new section of the search space.

Ridge: It is an area of search space that is higher than surrounding areas but that cannot be traversed by single moves in any one direction. It is a special kind of local maxima. Here apply two or more rules before doing the test, i.e. moving in several directions at once.

2.5.4 Beam Search

Beam search is a heuristic search algorithm in which W number of best nodes at each level is always expanded. It progresses level by level and moves downward only from the best W nodes at each level. Beam search uses breadth-first search to build its search tree. At each level of the tree, it generates all successors of the states at the current level, sorts them in order of increasing heuristic values. However, it only considers a W number of states at each level. Other nodes are ignored. Best nodes are decided on the heuristic cost associated with the node. Here W is called *width* of beam search. If B is the *branching factor*, there will be only W * B nodes under consideration at any depth but only W nodes will be selected. If beam width is smaller, the more states are pruned. If W = 1, then it becomes hill climbing search where always best node is chosen from successor nodes. If beam width is infinite, then no states are pruned and beam search is identical to breath-first search. The beam width bounds the memory required to perform the search, at the expense of risking termination or completeness and optimality (possibility that it will not find the best solution). The reason for such risk is that the goal state potentially might have been pruned.

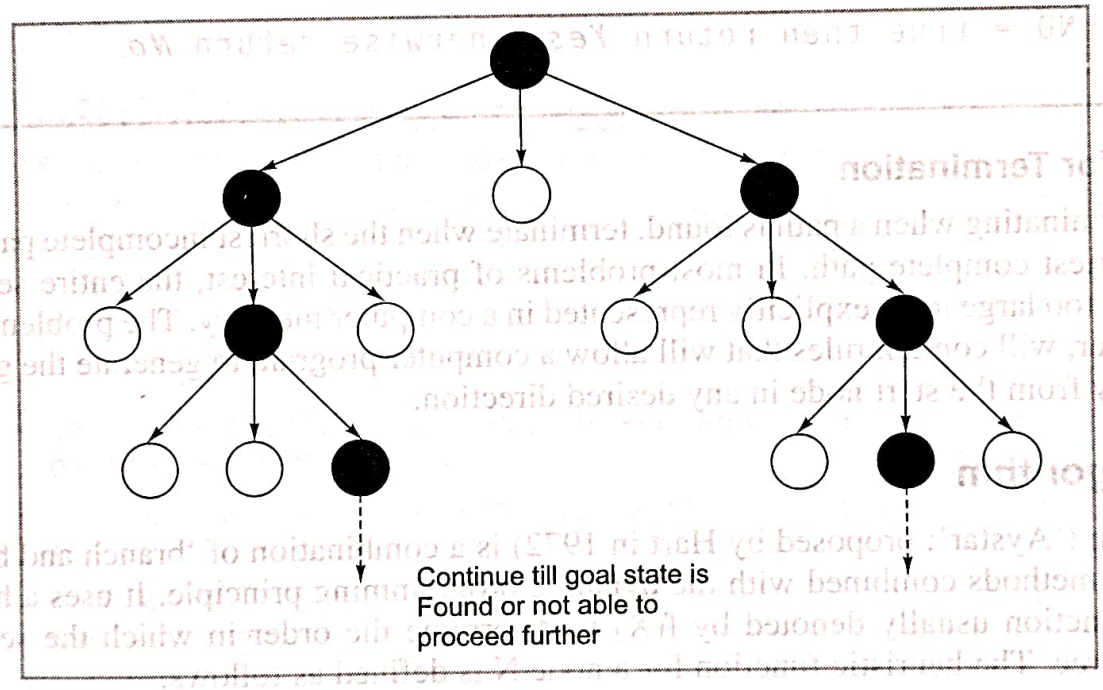
Algorithm *(Beam Search)*

Input: START and GOAL states
Local Variables: OPEN, NODE, SUCCs, W_OPEN, FOUND;
Output: Yes or No
Method:

- NODE = Root_node; Found = false;
- if NODE is the goal node, then *Found = true* else find SUCCs of NODE, if any with its estimated cost and store in OPEN list;
- while (FOUND = false and not able to proceed further) do
  {

```
• sort OPEN list;
• select top W elements from OPEN list and put it in W_OPEN list and empty
  OPEN list;
• for each NODE from W_OPEN list
  {
    • if NODE = Goal state then FOUND = true else find SUCCs of NODE, if any
      with its estimated cost and store in OPEN list;
  }
} /* end while */
• if FOUND = true then return Yes otherwise return No;
• Stop
```

The search tree generated using Beam search algorithm, assume W = 2 and B = 3 is given below. Here, black nodes are selected based on their heuristic values for further expansion.



2.5.5 Best-First Search

Best-first search is based on expanding the best partial path from current node to goal node. Here forward motion is from the best open node so far in the partially developed tree. The cost of partial paths is calculated using some heuristic.

If the state has been generated earlier and new path is better than the previous one, then change the parent and update the cost.

It should be noted that in hill climbing, sorting is done on the successors nodes, whereas in the best-first search, sorting is done on the entire list. It is not guaranteed to find an optimal solution, but generally it finds some solution faster than solution obtained from any other method. The performance varies directly with the accuracy of the heuristic evaluation function.

Algorithm *(Best-First Search)*

Input: START and GOAL states
Local Variables: OPEN, CLOSED, NODE, FOUND;
Output: Yes or No
Method:

- initialize OPEN list by root node; CLOSED = ϕ; FOUND = false;
- while (OPEN $\neq \phi$ and FOUND = false) do
  {
  - if the first element is the goal node, then FOUND = true else remove it from OPEN list and put it in CLOSED list;
  - add its successor, if any, in OPEN list;
  - sort the entire list by the value of some heuristic function that assigns to each node, the estimate to reach to the goal node;

  }
- if FOUND = true then return *Yes* otherwise return *No*;
- Stop

Condition for Termination

Instead of terminating when a path is found, terminate when the shortest incomplete path is longer than the shortest complete path. In most problems of practical interest, the entire search space graph will be too large to be explicitly represented in a computer memory. The problem specifications, however, will contain rules that will allow a computer program to generate the graph (tree) incrementally from the start node in any desired direction.

2.5.6 A* Algorithm

A* Algorithm ('Aystar'; proposed by Hart in 1972) is a combination of 'branch and bound' and 'best search' methods combined with the dynamic programming principle. It uses a heuristic or evaluation function usually denoted by f(X) to determine the order in which the search visits nodes in the tree. The heuristic function for a node N is defined as follows:

$$f(N) = g(N) + h(N)$$

The function *g* is a measure of the cost of getting from the start node to the current node N, i.e., it is sum of costs of the rules that were applied along the best path to the current node. The function *h* is an estimate of additional cost of getting from the current node N to the goal node. This is the place where knowledge about the problem domain is exploited. Generally, A* algorithm is called OR graph / tree search algorithm.

A* algorithm incrementally searches all the routes starting from the start node until it finds the shortest path to a goal. Starting with a given node, the algorithm expands the node with the lowest f(X) value. It maintains a set of partial solutions. Unexpanded leaf nodes of expanded nodes are stored in a queue with corresponding f values. This queue can be maintained as a priority queue.

Algorithm *(A*)*

Input: START and GOAL states
Local Variables: OPEN, CLOSED, Best_Node, SUCCs, OLD, FOUND;
Output: Yes or No
Method:

- initialization OPEN list with start node; CLOSED= ϕ; g = 0, f = h, FOUND = false;
- while (OPEN $\neq \phi$ and Found = false) do
  {
  - remove the node with the lowest value of f from OPEN list and store it in CLOSED list. Call it as a Best_Node;
  - if (Best_Node = Goal state) then FOUND = true else
    {
    - generate the SUCCs of the Best_Node;
    - for each SUCC do
      {
      - establish parent link of SUCC; /* This link will help to recover path once the solution is found */
      - compute g(SUCC) = g(Best_Node) + cost of getting from Best_Node to SUCC;
      - if SUCC $\in$ OPEN then /* already being generated but not processed */
        {
        - call the matched node as OLD and add it in the successor list of the Best_Node;
        - ignore the SUCC node and change the parent of OLD, if required as follows:
          - if g(SUCC) < g(OLD) then make parent of OLD to be Best_Node and change the values of g and f for OLD else ignore;

        }
      - If SUCC $\in$ CLOSED then /* already processed */
        {
        - call the matched node as OLD and add it in the list of the Best_Node successors;
        - ignore the SUCC node and change the parent of OLD, if required as follows:
          - if g(SUCC) < g(OLD) then make parent of OLD to be Best_Node and change the values of g and f for OLD and propogate the change to OLD's children using depth first search else ignore;

```
        }
      • If SUCC ∉ OPEN or CLOSED
        {
        • add it to the list of Best_Node's successors;
        • compute f(SUCC) = g(SUCC) + h(SUCC);
        • put SUCC on OPEN list with its f value
        }
      }
    }
  }    /* End while */
• if FOUND = true then return Yes otherwise return No;
• Stop
```

Let us consider an example of eight puzzle again and solve it by using A* algorithm. The simple evaluation function f(x) is defined as follows:

f(X) = g(X) + h(X), where

h(X) = the number of tiles not in their goal position in a given state X

g(X) = depth of node X in the search tree

Given

Start State

3	7	6
5	1	2
4	□	8

Goal State

5	3	6
7	□	2
4	1	8

The search tree using A* algorithm for eight-puzzle problem is given in Fig. 2.10. It should be noted that the quality of solution will depend on heuristic function. This simple heuristic may not be used to solve harder eight-puzzle problems (Nilsson N. J., 1980). Let us consider the following puzzle and try solving using earlier heuristic function. You will find that it cannot be solved.

Start State

3	5	1
2	□	7
4	8	6

Goal State

5	3	6
7	□	2
4	1	8

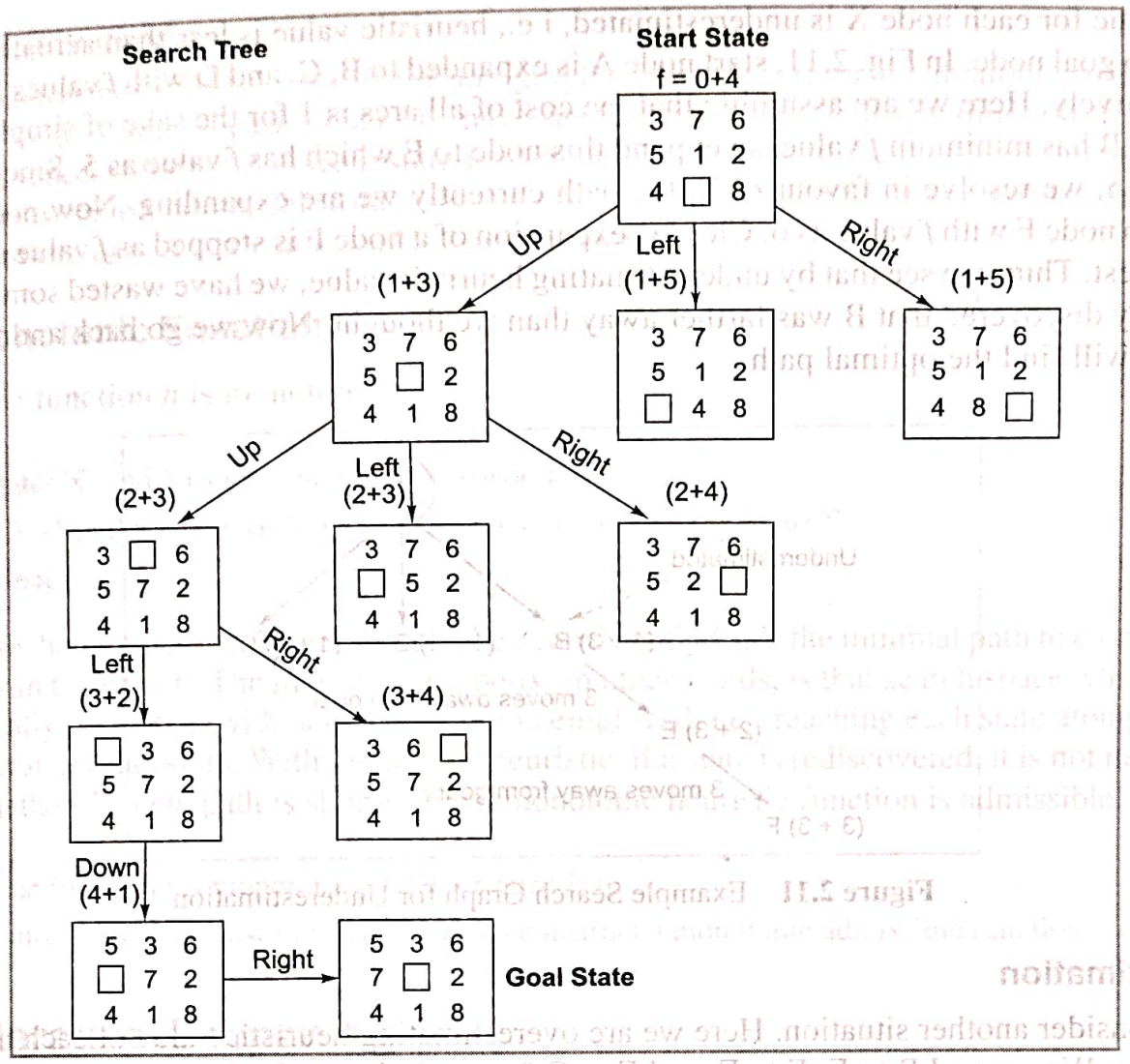


Figure 2.10 Search Tree

A better estimate of *h* function might be as follows. The function *g* may remain same.

h(X) = the sum of the distances of the tiles (1 to 8) from their goal position in a given state X.

Here start state has h(start_state) = 3 + 2 + 1 + 0 + 1 + 2 + 2 + 1 = 12

For the sake of brevity, search tree has been omitted.

2.5.7 Optimal Solution by A* Algorithm

A* algorithm finds optimal solution if heuristic function is carefully designed and is underestimated. We will support the argument using the following example (Rich and Knight, 2003).

Underestimation

If we can guarantee that *h* never overestimates actual value from current to goal, then A* algorithm ensures to find an optimal path to a goal, if one exists. Let us illustrate this by the following example shown in Fig. 2.11. Formal proof is omitted as it is not relevant here. Here we assume

that h value for each node X is underestimated, i.e., heuristic value is less than actual value from node X to goal node. In Fig. 2.11, start node A is expanded to B, C, and D with f values as 4, 5, and 6, respectively. Here we are assuming that the cost of all arcs is 1 for the sake of simplicity. Note that node B has minimum f value, so expand this node to E which has f value as 5. Since f value of C is also 5, we resolve in favour of E, the path currently we are expanding. Now node E is expanded to node F with f value as 6. Clearly, expansion of a node F is stopped as f value of C is now the smallest. Thus, we see that by underestimating heuristic value, we have wasted some effort but eventually discovered that B was farther away than we thought. Now we go back and try another path and will find the optimal path.

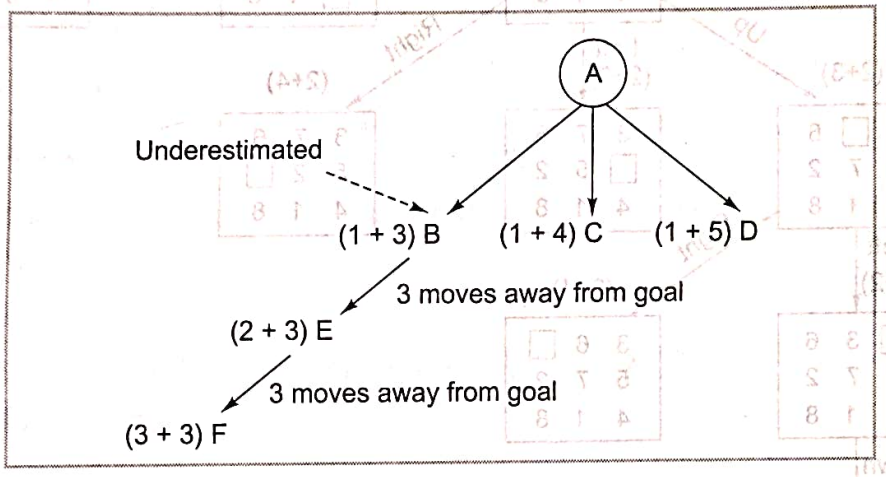


Figure 2.11 Example Search Graph for Underestimation

Overestimation

Let us consider another situation. Here we are overestimating heuristic value of each node in the graph/tree. We expand B to E, E to F, and F to G for a solution path of length 4. But assume that there is a direct path from D to a solution giving a path of length 2 as h value of D is also overestimated. We will never find it because of overestimating h(D).We may find some other worse solution without ever expanding D. So by overestimating h, we cannot be guaranteed to find the shortest path.

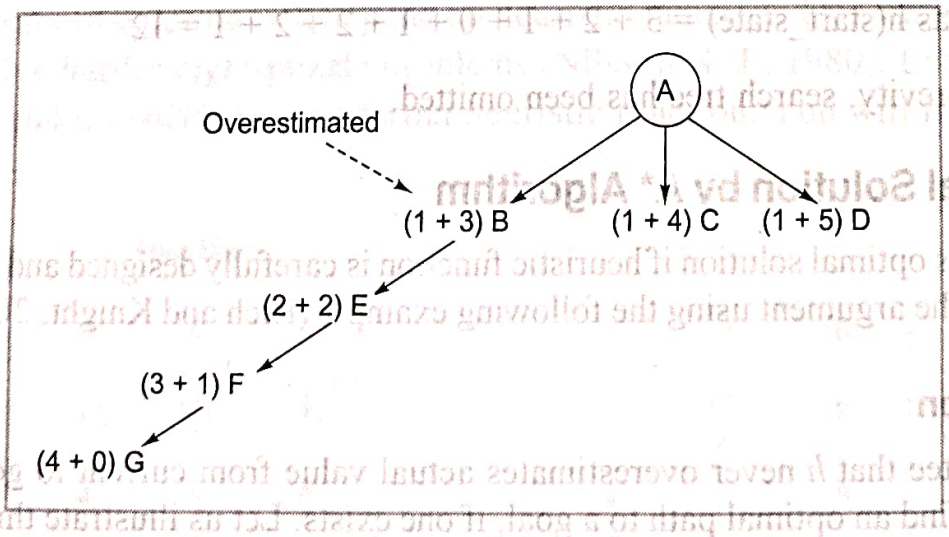


Figure 2.12 Example Search Graph for Overestimation

Admissibility of A*

A search algorithm is *admissible*, if for any graph, it always terminates in an optimal path from start state to goal state, if path exists. We have seen earlier that if heuristic function 'h' underestimates the actual value from current state to goal state, then it bounds to give an optimal solution and hence is called admissible function. So, we can say that A* always terminates with the optimal path in case *h* is an *admissible heuristic function*.

2.5.8 Monotonic Function

A heuristic function *h* is monotone if

1. ∀ states Xi and Xj such that Xj is successor of Xi
   h(Xi) – h(Xj) ≤ cost (Xi, Xj) i.e., actual cost of going from Xi to Xj
2. h (Goal) = 0

In this case, heuristic is locally admissible, i.e., consistently finds the minimal path to each state they encounter in the search. The monotone property, in other words, is that search space which is every where locally consistent with heuristic function employed, i.e., reaching each state along the shortest path from its ancestors. With monotonic heuristic, if a state is rediscovered, it is not necessary to check whether the new path is shorter. Each monotonic heuristic function is admissible.

- A cost function *f* is monotone if f(N) ≤ f(succ(N))
- For any admissible cost function *f*, we can construct a monotonic admissible function.

2.6 Iterative-Deepening A*

Iterative-Deepening A* (IDA*) is a combination of the depth-first iterative deepening and A* algorithm. Here the successive iterations are corresponding to increasing values of the total cost of a path rather than increasing depth of the search. Algorithm works as follows:

- For each iteration, perform a DFS pruning off a branch when its total cost (g + h) exceeds a given threshold.
- The initial threshold starts at the estimate cost of the start state and increases for each iteration of the algorithm.
- The threshold used for the next iteration is the minimum cost of all values exceeded the current threshold.
- These steps are repeated till we find a goal state.

Let us consider an example to illustrate the working of IDA* algorithm as shown in Fig. 2.13. Initially, the threshold value is the estimated cost of the start node. In the first iteration, Threshold = 5. Now we generate all the successors of start node and compute their estimated values as 6, 8, 4, 8, and 9. The successors having values greater than 5 are to be pruned. Now for next iteration,

we consider the threshold to be the minimum of the pruned nodes value, that is, threshold = 6 and the node with 6 value along with node with value 4 are retained for further expansion.

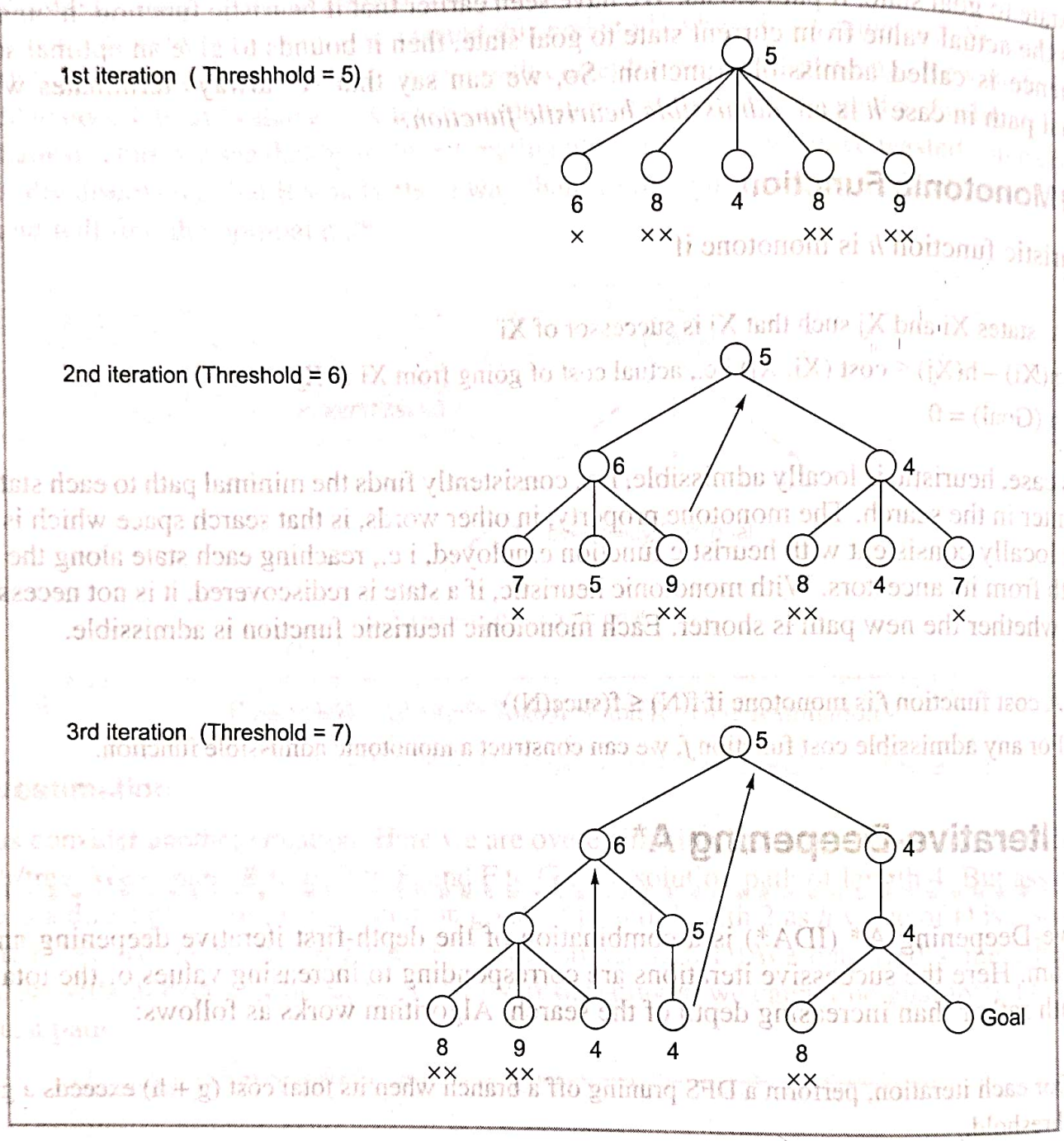


Figure 2.13 Working of IDA*

The IDA* will find a solution of least cost or optimal solution (if one exists), if an admissible monotonic cost function is used. IDA* not only finds cheapest path to a solution but uses far less space than A*, and it expands approximately the same number of nodes as that of A* in a tree search. An additional benefit of IDA* over A* is that it is simpler to implement, as there are no open and closed lists to be maintained. A simple recursion performs DFS inside an outer loop to handle iterations.

2.7 Constraint Satisfaction

Many AI problems can be viewed as problems of constraint satisfaction in which the goal is to solve some problem state that satisfies a given set of constraints instead of finding optimal path to the solution. Such problems are called *Constraint Satisfaction (CS) Problems*. Search can be made easier in those cases in which the solution is required to satisfy local consistency conditions. For example, some of the simple constraint satisfaction problems are cryptography, the n-Queen problem, map coloring, crossword puzzle, etc.

A cryptography problem: A number puzzle in which a group of arithmetical operations has some or all of its digits replaced by letters and the original digits must be found. In such a puzzle, each letter represents a unique digit. Let us consider the following problem in which we have to replace each letter by a distinct digit (0–9) so that the resulting sum is correct.

```
   B A S E
+  B A L L
---------
 G A M E S
---------
```

The n-Queen problem: The condition is that on the same row, or column, or diagonal no two queens attack each other.

A map colouring problem: Given a map, color regions of map using three colours, blue, red, and black such that no two neighboring countries have the same colour.

In general, we can define a *Constraint Satisfaction Problem* as follows:

- a *set of variables* {x1, x2, .., xn}, with each xi ∈ Di with possible values and
- a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables.

The problem is to find, for each i, $1 \leq i \leq n$, a value of xi ∈ Di , so that all constraints are satisfied. A CS problem is usually represented as an undirected graph, called *Constraint Graph* in which the nodes are the variables and the edges are the binary constraints. We can easily see that a CSP can be given an incremental formulation as a standard search problem.

- *Start state*: the empty assignment, i.e. all variables, are unassigned.
- *Goal state*: all the variables are assigned values which satisfy constraints.
- *Operator*: assigns value to any unassigned variable, provided that it does not conflict with previously assigned variables.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables. Furthermore, the search tree extends only to depth n and hence depth-first search algorithms are popular for CSPs.

Many design tasks can also be viewed as constraint satisfaction problems. Such problems do not require new search methods but they can be solved using any of the search strategies which can be augmented with the list of constraints that change as parts of the problem are solved. The following algorithm is applied for the CSP. This procedure can be implemented as a DF search (Rich and Knight, 2003).

Algorithm

- until a complete solution is found or all paths have lead to dead ends
  {
  - select an unexpanded node of the search graph;
  - apply the constraint inference rules to the selected node to generate all possible new constraints;
  - if the set of constraints contain a contradiction, then report that this path is a dead end;
  - if the set of constraint describes a complete solution, then report success;
  - if neither a contradiction nor a complete solution has been found, then apply the problem space rules to generate new partial solutions that are consistent with the current set of constraints. Insert these partial solutions into the search graph;

  }
- Stop

Let us solve the following crypt-arithmetic puzzle.

Crypt-Arithmetic Puzzle

Problem Statement: Solve the following puzzle by assigning numeral (0–9) in such a way that each letter is assigned unique digit which satisfy the following addition:

$$\begin{array}{r} B\ A\ S\ E \\ +\ B\ A\ L\ L \\ \hline G\ A\ M\ E\ S \\ \hline \end{array}$$

- Constraints: No two letters have the same value (the constraints of arithmetic).
- Initial Problem State
  G = ? ; A = ? ; M = ? ; E = ? ; S = ? ; B = ? ; L = ?
- Apply constraint inference rules to generate the relevant new constraints.
- Apply the letter assignment rules to perform all assignments required by the current set of constraints. Then choose another rules to generate an additional assignment, which will, in turn, generate new constrains at the next cycle.
- At each cycle, there may be several choices of rules to apply.
- A useful *heuristics* can help to select the best rule to apply first.

For example, if a letter that has only two possible values and another with six possible values, then there is a better chance of guessing right on the first than on the second.

C4	C3	C2	C1		← Carries
	B	A	S	E	
+	B	A	L	L	
G	A	M	E	S	

Constraints equations are:

$E + L = S \rightarrow C_1$
$S + L + C_1 = E \rightarrow C_2$
$2A + C_2 = M \rightarrow C_3$
$2B + C_3 = A \rightarrow C_4$
$G = C_4$

We can easily see that G has to be non-zero digit, so the value of carry C4 should be 1 and hence G = 1.

The tentative steps required to solve above crypt-arithmetic are given in Fig. 2.14.

Example: Let us solve another crypt-arithmetic puzzle.

C3	C2	C1		← Carries
	T	W	O	
+	T	W	O	
F	O	U	R	

Constraints equations:

$2O = R \rightarrow C_1$
$2W + C_1 = U \rightarrow C_2$
$2T + C_2 = O \rightarrow C_3$
$F = C_3$

The search tree using DF search approach for solving the crypt-arithmetic puzzle is given in Fig. 2.15. We get two possible solutions as {F = 1, T = 7, O = 4, R = 8, W = 3, U = 6} and {F = 1, T = 7, O = 5, R

W = 6, U = 3}. On backtracking we may get more solutions. All possible solutions are given as follows:

F	T	O	R	W	U
1	8	6	2	3	7
1	8	6	2	4	9
1	8	7	4	6	3
1	9	8	6	2	5

Crypt-Arithmetic Solution Trace

Constraints equations			*Initial State*
$G = C_4$			G = ? ; A = ? ; M = ? ; E = ? ;
$2B + C_3 = A$	→	C_4	S = ? ; B = ? ; L = ?
$2A + C_2 = M$	→	C_3	
$S + L + C_1 = E$	→	C_2	
$E + L = S$	→	C_1	

1. $G = C_4 \Rightarrow$ **G = 1**
2. $2B + C_3 = A \longrightarrow C_4$
   2.1 Since $C_4 = 1$, therefore, $2B + C_3 > 9 \Rightarrow$ B can take values from 5 to 9.
   2.2 Try the following steps for each value of B from 5 to 9 till we get a possible value of B.
   - If B = 5
     - if $C_3 = 0 \Rightarrow A = 0 \Rightarrow M = 0$ for $C_2 = 0$ or $M = 1$ for $C_2 = 1$ ×
     - if $C_3 = 1 \Rightarrow A = 1$ × (as G = 1 already)
   - For B = 6 we get similar contradiction while generating the search tree.
   - If **B = 7**, then for $C_3 = 0$, we get **A = 4** $\Rightarrow M = 8$ if $C_2 = 0$ that leads to contradiction later, so this path is pruned. If $C_2 = 1$, then **M = 9**
3. Let us solve $S + L + C_1 = E$ and $E + L = S$
   - Using both equations, we get $2L + C_1 = 0 \Rightarrow$ **L = 5** and $C_1 = 0$
   - Using L = 5, we get S + 5 = E that should generate carry $C_2 = 1$ as shown above
   - So S+5 > 9 ⇒ Possible values for E are {2, 3, 6, 8} (with carry bit $C_2 = 1$)
   - If E = 2 then S + 5 = 12 ⇒ S = 7 (as B = 7 already)
   - If E = 3 then S + 5 = 13 ⇒ S = 8.
   - Therefore **E = 3** and **S = 8** are fixed up
4. Hence we get the final solution as given below and on backtracking, we may find more solutions. In this case we get only one solution.

   G = 1 ; A = 4 ; M = 9; E = 3 ; S = 8 ;B = 7 ; L = 5

Figure 2.14 Working Steps for Crypt–Arithmetic Puzzle

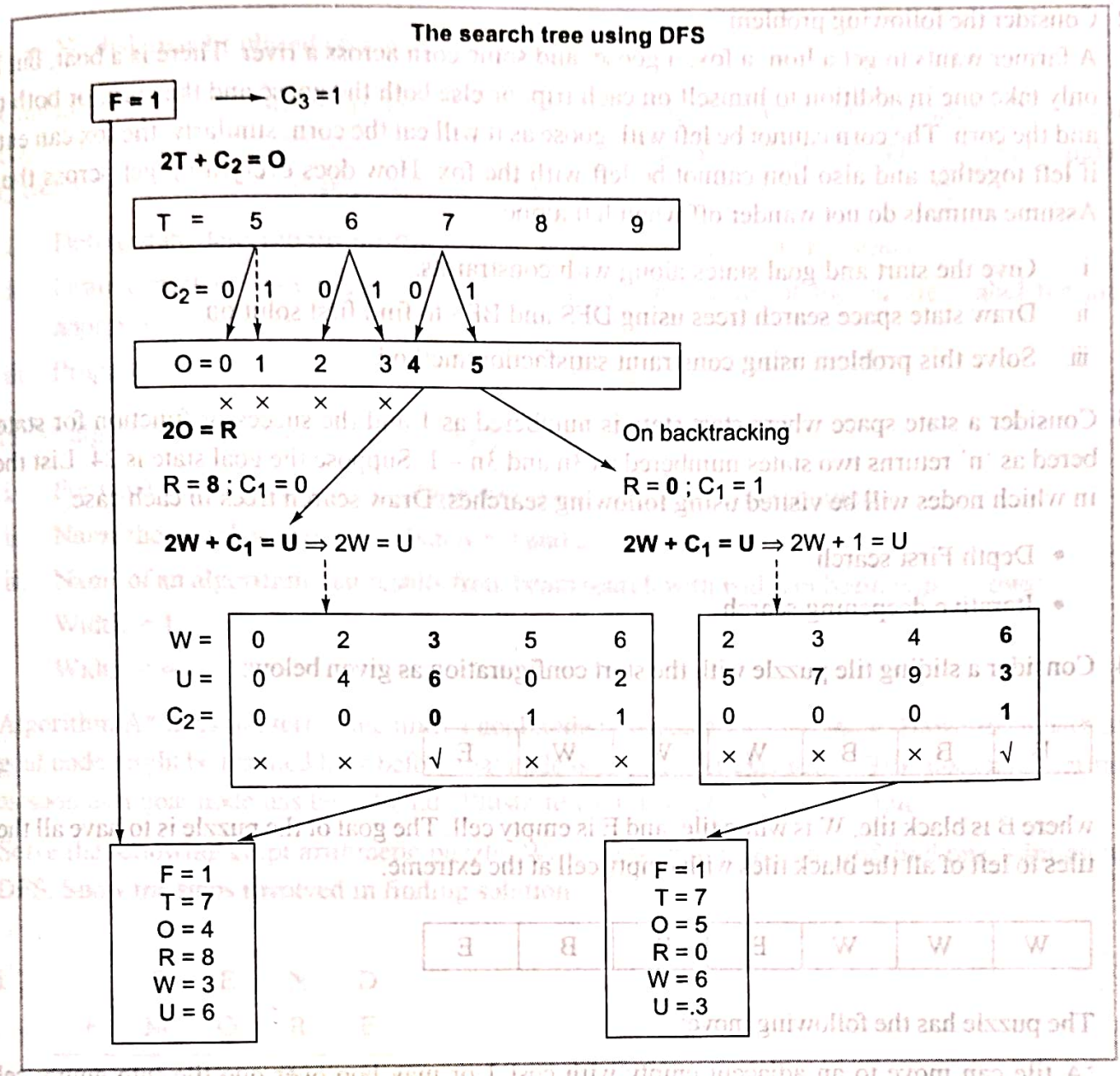


Figure 2.15 Search Tree for Crypt-Arithmetic Puzzle

Exercises

2.1 Consider the problem of driving a car from one place to another. Assume that route is not known. Consider this as a search problem with three operators:

R1: Travel to the next crossing and keep going straight, if possible

R2: Travel to the next crossing and turn to right, if possible

R3: Travel to the next crossing and turn to left, if possible

Which control strategy is better: depth first or breadth first?

2.2 Consider the following problem:
A farmer wants to get a lion, a fox, a goose, and some corn across a river. There is a boat, but he can only take one in addition to himself on each trip, or else both the goose and the corn, or both the fox and the corn. The corn cannot be left with goose as it will eat the corn; similarly, the fox can eat goose if left together and also lion cannot be left with the fox. How does everything get across the river? Assume animals do not wander off when left alone.

i. Give the start and goal states along with constraints.
ii. Draw state space search trees using DFS and BFS to find first solution.
iii. Solve this problem using constraint satisfaction method.

2.3 Consider a state space where start state is numbered as 1 and the successor function for state numbered as 'n' returns two states numbered as 3n and 3n – 1. Suppose the goal state is 24. List the order in which nodes will be visited using following searches. Draw search trees in each case.

- Depth First search
- Iterative deepening search

2.4 Consider a sliding tile puzzle with the start configuration as given below:

B	B	B	W	W	W	E

where B is black tile, W is white tile, and E is empty cell. The goal of the puzzle is to have all the white tiles to left of all the black tiles with empty cell at the extreme.

W	W	W	B	B	B	E

The puzzle has the following move:

'A tile can move to an adjacent empty with cost 1 or may hop over one tile into empty cell with cost 2'

i. Specify rules and control strategy for a production system.
ii. Find solution using A* algorithm.

2.5 The *Tower of Hanoi puzzle* consists of three pegs, and a number of disks of different sizes which can slide onto any peg. The puzzle starts with the disks neatly stacked in order of size on one peg, smallest at the top, thus making a conical shape. The objective of the game is to move the entire stack to another peg, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the pegs and sliding it onto another peg, on top of the other disks that may already be present on that peg.

- No disk may be placed on top of a smaller disk.

Consider three-discs Tower of Hanoi puzzle. Let the operators that describe actions be given by move (D, P, Q) (move disk D from peg P to peg Q), where D can be any of three discs D1, D2, or D3, and P and Q can be any pair of distinct pegs A, B, or C. Here size (D1) > size (D2) > size (D3).

i. Define state descriptions for this puzzle. Identify the start and goal states.

ii. Draw complete search space containing all possible states of the puzzle. Label the arcs by appropriate operators.

iii. Propose an admissible h function for this problem.

2.6 In A* algorithm we use heuristic function $f(n) = (2 - w)*g(n) + w*h(n)$.

i. For what values of 'w' will A* algorithm guarantee an optimal solution

ii. Name the search algorithms when w = 0 and 2

iii. Name of an algorithm that results from beam search with width of beam is as follows:

Width = 1

Width = ∞

2.7 Algorithm A* does not terminate until a goal node is selected for expansion. However, a path to the goal node might be reached long before that node is selected for expansion. Why does not it terminate as soon as a goal node has been found? Illustrate your answer with an example.

2.8 Solve the following crypt arithmetic puzzle. Write constraint equations and find one solution using DFS. Show the steps involved in finding solution.

i.

```
    S E N D
+   M O R E
-----------
M O N E Y
```

ii.

```
    R O A D S
+   C R O S S
-------------
D A N G E R
```

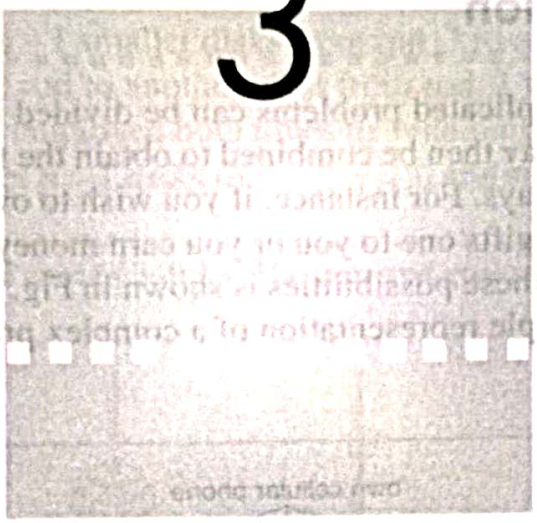
iii.

```
    D O N A L D
+   G E R A L D
---------------
R O B E R T
```

iv.

	L	O	G	I	C
+	L	O	G	I	C
P	R	O	L	O	G

3



Problem Reduction and Game Playing

3.1 Introduction

So far, we have only considered the search strategies for OR graphs. In this graph, arcs indicate the number of alternative ways in which a given problem may be solved. We may sometimes encounter certain real-life problems which are extremely complicated. An effective way of solving a complex problem is to reduce it to simpler parts and solve each part separately. The problem is automatically solved when we obtain solutions to all the smaller, simpler problems. This is the basic intuition behind the method of *problem reduction*. This method enables us to obtain an elegant solution to the problem. The structure called AND–OR graph (or tree) is useful for representing the solution of complicated problems. The decomposition of a complex problem generates arcs which we call AND arcs. One AND arc may point to any number of successors, all of which must be solved. The proposed structure is called AND–OR graph rather than simply AND graph.

In this chapter, we will discuss the concept of AND–OR graphs and its use in game playing. Game playing is one of the most direct applications of state space-search problem-solving paradigm. However, the search procedures employed in game playing are different from the ones used in state search problems as these are based on the concept of *generate and test philosophy*. In this method, the generator generates individual moves in the search space; each of these moves is then evaluated by the tester and the most promising one is chosen. The effectiveness of a search may be improved by improving the generate-and-test procedures used. The generate procedure should be such that it generates good moves (or paths), while the test procedure recognizes the best moves out of these and explores them first. In case of game playing, a change in state in state search space is solely caused by the actions from the point of view of one player. However, the points of view of other players having different goals also have to be taken into consideration. In this chapter, we will develop search procedures for two-player games as they are more common and easier to design and execute.

3.2 Problem Reduction

In real-world applications, complicated problems can be divided into simpler sub-problems; the solution of each sub-problem may then be combined to obtain the final solution. A given problem may be solved in a number of ways. For instance, if you wish to own a cellular phone then it may be possible that either someone gifts one to you or you earn money and buy one for yourself. The AND–OR graph which depicts these possibilities is shown in Fig. 3.1 (Rich & Knight, 2003). An AND–OR graph provides a simple representation of a complex problem and hence aids in better understanding.

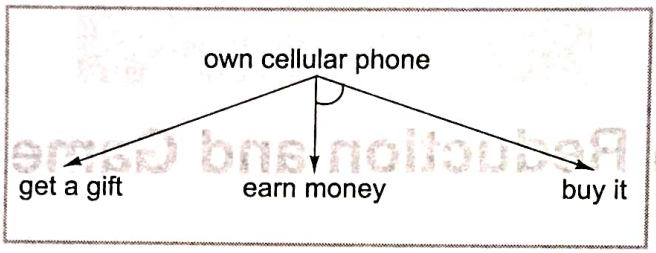


Figure 3.1 A Simple AND–OR Graph

Thus, this structure may prove to be useful to us in a number of problems involving real-life situations. To find a solution using AND–OR graphs, we need an algorithm similar to the A^* algorithm (discussed in Chapter 2) with an ability to handle AND arcs.

Let us consider a problem known as the *Tower of Hanoi* to illustrate the need of problem-reduction concept. It consists of three rods and a number of disks of different sizes which can slide onto any rod [Paul Brna 1996]. The puzzle starts with the disks being stacked in descending order of their sizes, with the largest at the bottom of the stack and the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod by using the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the uppermost disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

Consider that there are n disks in one rod (rod_1). Now, our aim is to move these n disks from rod_1 to rod_2 making use of rod_3. Let us develop an algorithm which shows that this problem can be solved by reducing it to smaller problems. Basically the method of recursion will be used to solve this problem. The game tree that is generated will contain AND–OR arcs. The solution of this problem will involve the following steps:

If $n = 1$, then simply move the disk from rod_1 to rod_2. If $n > 1$, then somehow move all the top smaller $n - 1$ disks in the same order from rod_1 to rod_3, and then move the largest disk from

rod_1 to rod_2. Finally, move $n-1$ smaller disks from rod_3 to rod_2. So, the problem is reduced to moving $n-1$ disks from one rod to another, first from rod_1 to rod_3 and then from rod_3 to rod_2; the same method can be employed both times by renaming the rods. The same strategy can be used to reduce the problem of $n-1$ disks to $n-2$, $n-3$, and so on until only one disk is left.

Example 3.1 Let us consider the case of 3 disks. The start and goal states are shown in Fig. 3.2.

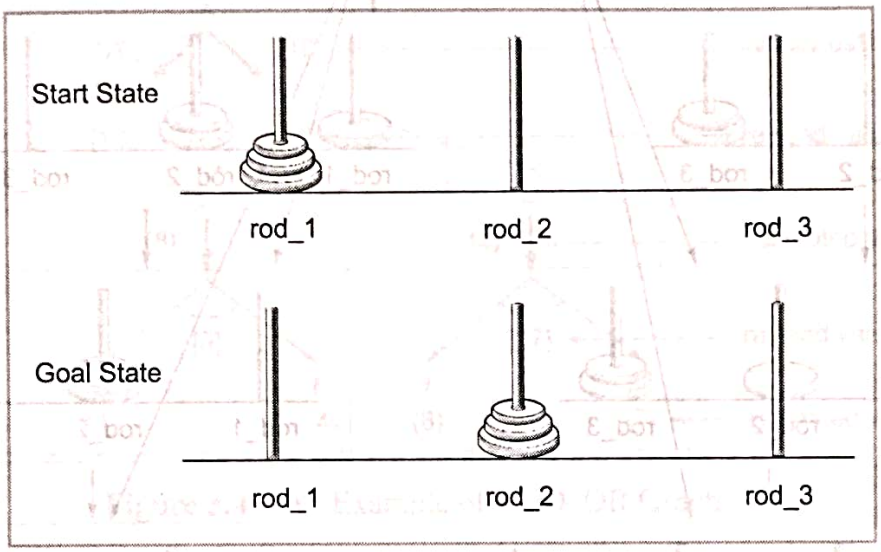


Figure 3.2 The Start and Goal States of a Three-disk *Tower of Hanoi* Problem

The search space graph (not completely expanded) shown in Fig. 3.3 is an AND–OR graph. Here, we have shown two alternative paths. One path is from root to state A and the other is from root to state A'. The path through A requires all the three states A, B, and C to be achieved to solve the problem. In order to achieve state A we need to expand it in a similar fashion, which will again require drawing of AND arcs. The process continues till we achieve the goal state, that is, state C is achived. It is important to note that the subtasks in the process are not independent of each other and therefore cannot be achieved in parallel. State B will be obtained after state A has been achieved, and state C will be obtained after state B has been achieved. The second path is from root to state A', then to state B', and then to state C'. This path is to be continued till we reach the goal state. The path from root to state A is optimal whereas the path from root to state A' is longer.

We will use the heuristic function f for each node in the AND–OR graph similar to the one used in the algorithm A^* to compute the estimated value. A given node in the graph may be either an OR node or an AND node. In an AND–OR graph, the estimated costs for all paths generated from the start node to level one are calculated by the heuristic function and placed at the start node itself. The best path is then chosen to continue the search further; unused nodes on the chosen best path are explored and their successors are generated. The heuristic values of the successor nodes are calculated and the cost of parent nodes is revised accordingly. This revised cost is propagated back to the start node through the chosen path. Let us explain this concept by considering a hypothetical example.

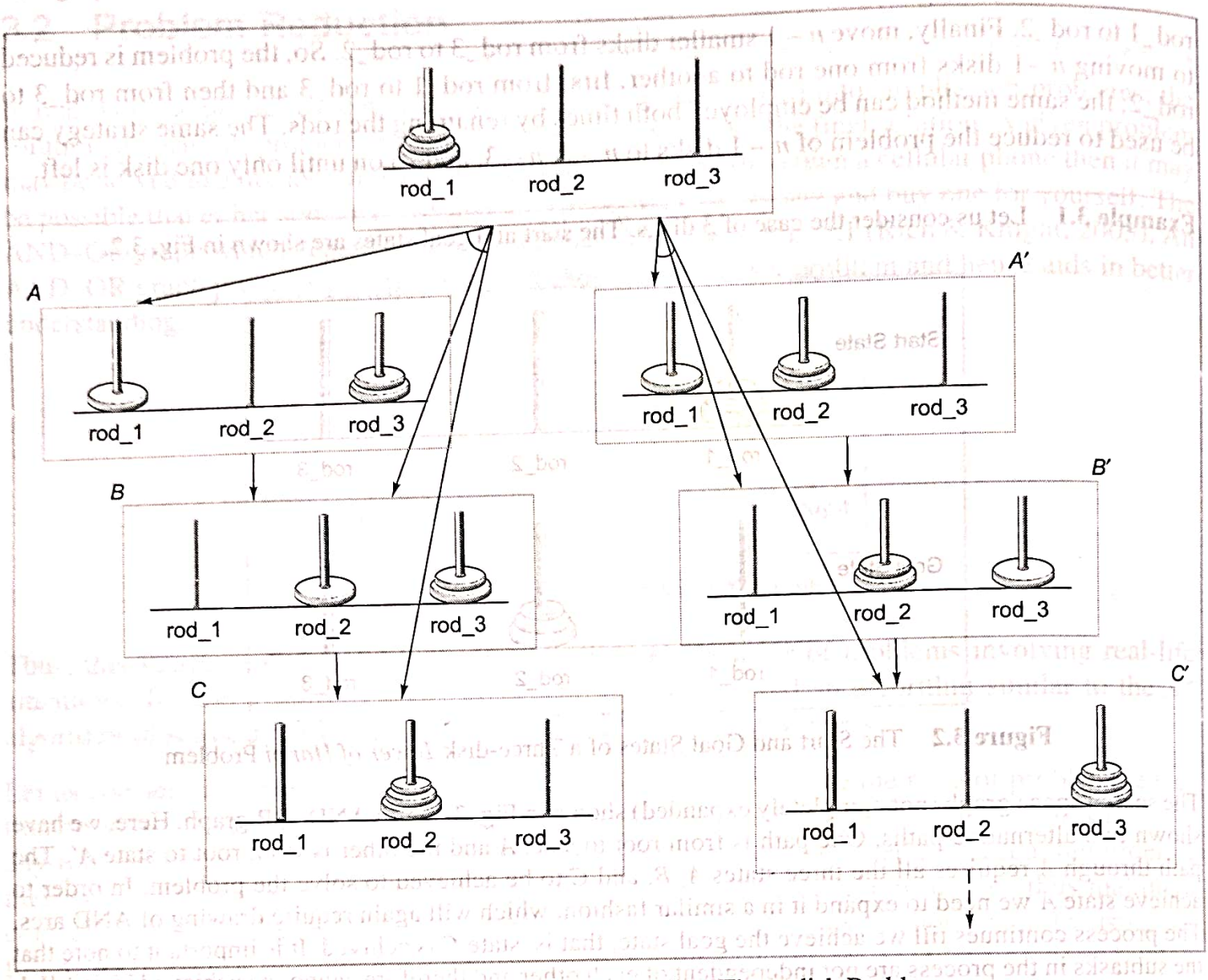


Figure 3.3 An AND–OR Graph for a Three-disk Problem

Consider an AND–OR graph (Fig. 3.4) where each arc with a single successor has a cost of 1; also assume that each AND arc with multiple successors has a cost of 1 for each of its components for the sake of simplicity. In the tree shown in Fig. 3.4, let us assume that the numbers listed in parenthesis, ( ), denote the estimated costs, while the numbers in the square brackets, [ ], represent the revised costs of path. Thick lines in the figure indicate paths from a given node. We begin our search from start node A and compute the heuristic values for each of its successors, say B and (C, D) as 19 and (8, 9) respectively. The estimated cost of paths from A to B is 20 (19 + cost of one arc from A to B) and that from A to (C, D) is 19 (8 + 9 + cost of two arcs, A to C and A to D). The path from A to (C, D) seems to be better than that from A to B. So, we expand this AND path by extending C to (G, H), and D to (I, J). Now, the heuristic values of G, H, I, and J are 3, 4, 8, and 7, respectively, which lead to revised costs of C and D as 9 and 17, respectively. These values are then propagated up and the revised costs of path from A to (C, D) is calculated as 28 (9 + 17 + cost of arcs A to C and A to D).

Note that the revised cost of this path is now 28 instead of the earlier estimation of 19; thus, this path is no longer the best path now. Therefore, choose the path from *A* to *B* for expansion. After expansion we see that the heuristic value of node *B* is 17 thus making the cost of the path from *A* to *B* equal to 18. This path is the best so far; therefore, we further explore the path from *A* to *B*. The process continues until either a solution is found or all paths lead to dead ends, indicating that there is no solution.

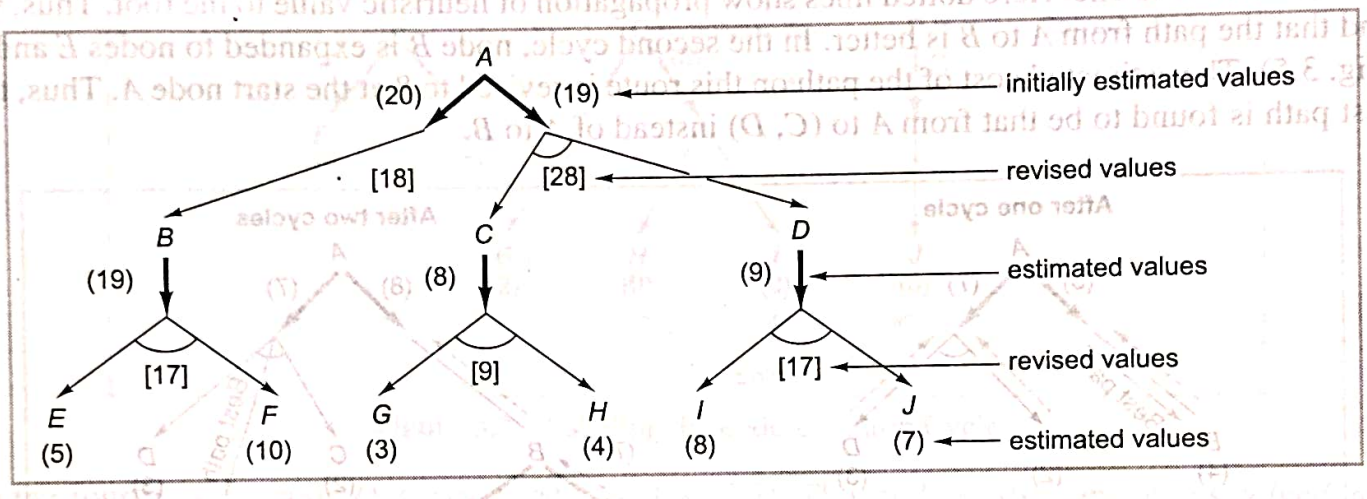


Figure 3.4 An Example of AND–OR Graph

It should be noted that the propagation of the estimated cost of the path is not relevant in *A** algorithm as it is used for an OR graph where there is a clear path from the start to the current node and the best node is expanded. In case of an AND–OR graph, there need not be a direct path from the start node to the current node because of the presence of AND arcs. Therefore, the cost of the path is recalculated by propagating the revised costs. For handling such graphs, a modified version of the algorithm *A** called *AO** algorithm is used. Before discussing *AO** algorithm, let us study the status labelling procedure of a node.

Node Status Labelling Procedure

At any point in time, a node in an AND–OR graph may be either a terminal node or a non-terminal AND/OR node. The labels used to represent these nodes in a graph (or tree) are described as follows:

- *Terminal node*: A terminal node in a search tree is a node that cannot be expanded further. If this node is the goal node, then it is labelled as *solved*; otherwise, it is labelled as *unsolved*. It should be noted that this node might represent a sub-problem.
- *Non-terminal AND node*: A non-terminal AND node is labelled as *unsolved* as soon as one of its successors is found to be unsolvable; it is labelled as *solved* if all of its successors are solved.
- *Non-terminal OR node*: A non-terminal OR node is labelled as *solved* as soon as one of its successors is labelled *solved*; it is labelled as *unsolved* if all its successors are found to be unsolvable.

Let us explain the labelling procedure with the help of an example. The AND–OR trees are generated levelwise as shown in Figs 3.5 to 3.7. The first two cycles are shown in Fig. 3.5.

In the first cycle, we expand the start node A to node B and nodes (C, D) (Fig. 3.5). The heuristic values at node B and nodes (C, D) are computed as 4 and (2, 3), respectively. The estimated costs of paths from A to B and from A to (C, D) are determined as 5 and 7, respectively assuming that the cost of each arc is one. Here dotted lines show propagation of heuristic value to the root. Thus, we find that the path from A to B is better. In the second cycle, node B is expanded to nodes E and F (Fig. 3.5). The estimated cost of the path on this route is revised to 8 at the start node A. Thus, the best path is found to be that from A to (C, D) instead of A to B.

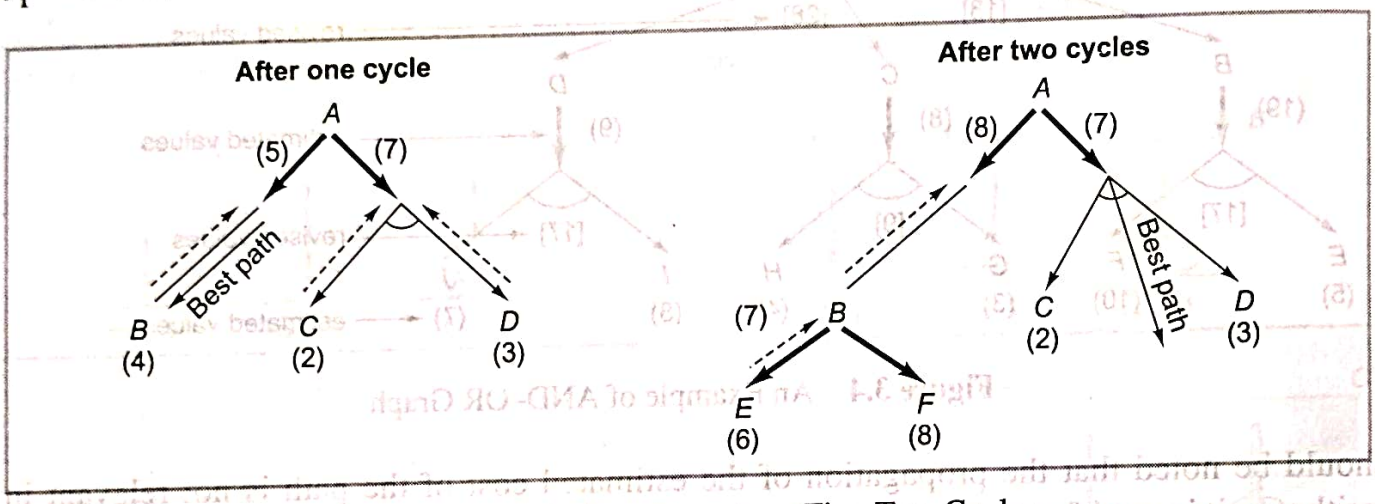


Figure 3.5 Labelling Procedure: First Two Cycles

In the third cycle, node C is expanded to $\{G, (H, I)\}$ (Fig. 3.6). Here we notice that the heuristic value at nodes H and I is 0 indicating that these are terminal solution nodes; H and I are thus labelled as *solved.* Node C also gets labelled as *solved* using the status labelling procedure.

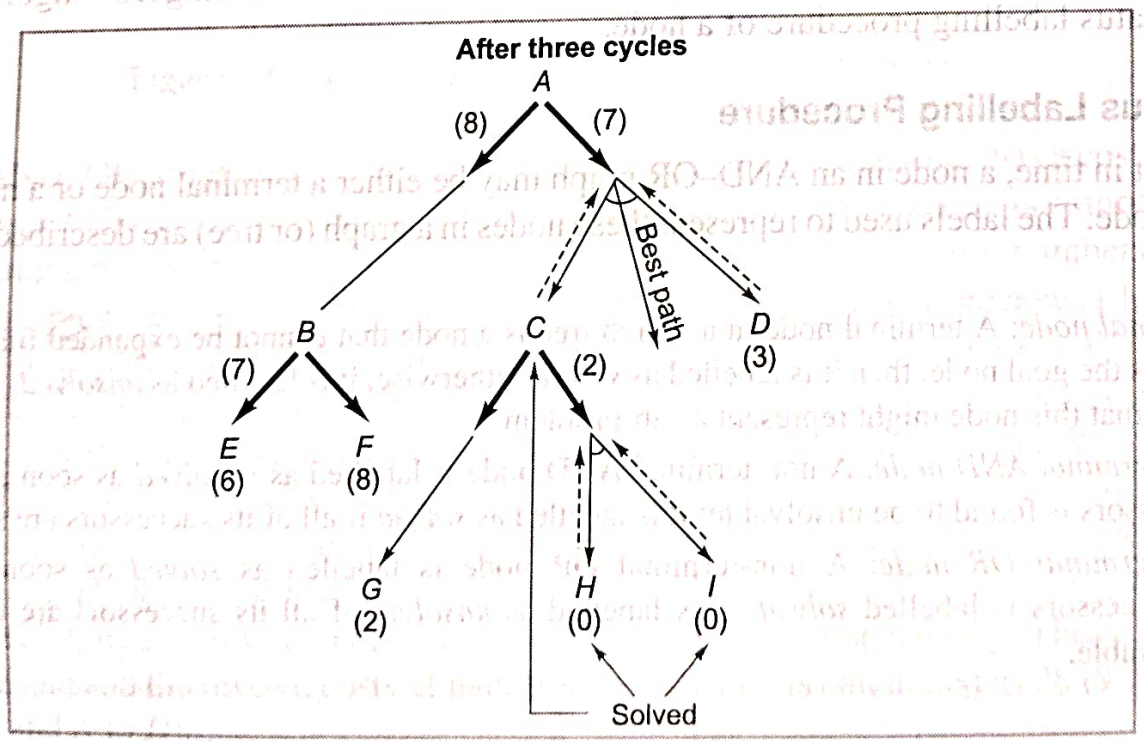


Figure 3.6 Labelling Procedure: Third Cycle

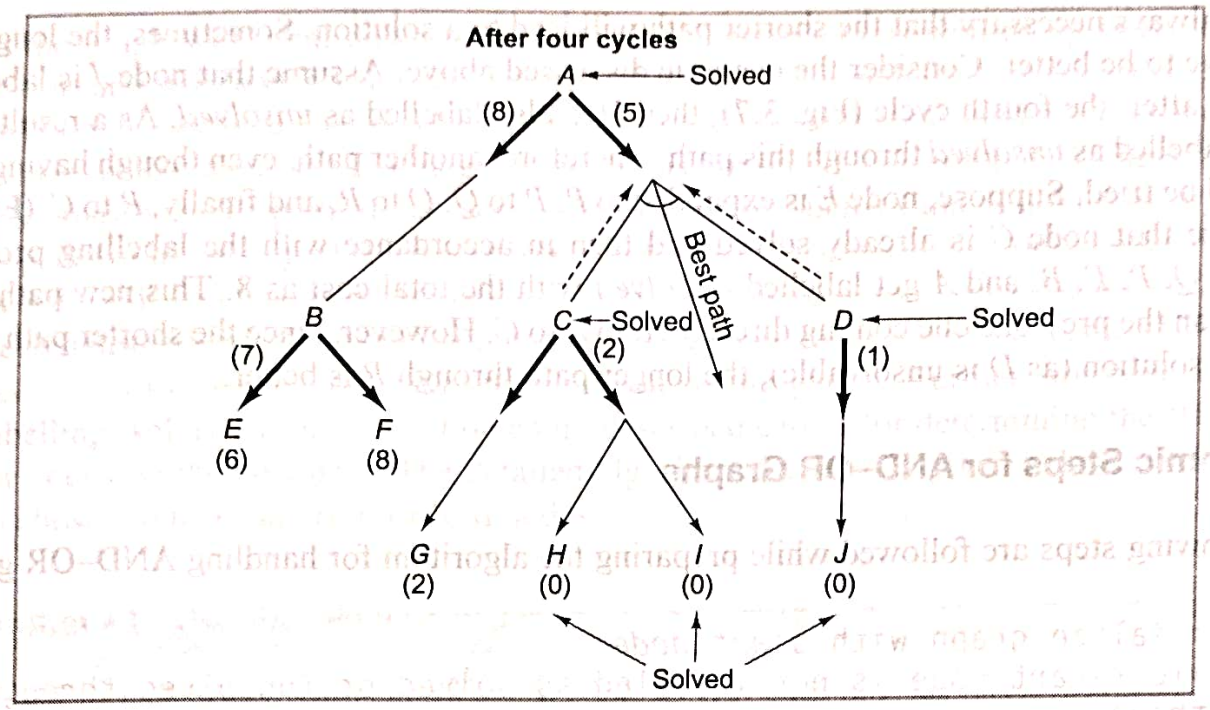


Figure 3.7 Labelling Procedure: Fourth Cycle

In the fourth cycle, node *D* is expanded to *J* (Fig. 3.7). This node is also labelled as *solved*, and subsequently node *D* attains the *solved* label. The start node *A* also gets labelled as *solved* as *C* and *D* both are labelled as solved. Along with the labelling status, the cost of the path is also propagated. In this example, the solution graph with minimal cost equal to 5 is obtained by tracing down through the marked arrows as A → (C → (H, I), D → J).

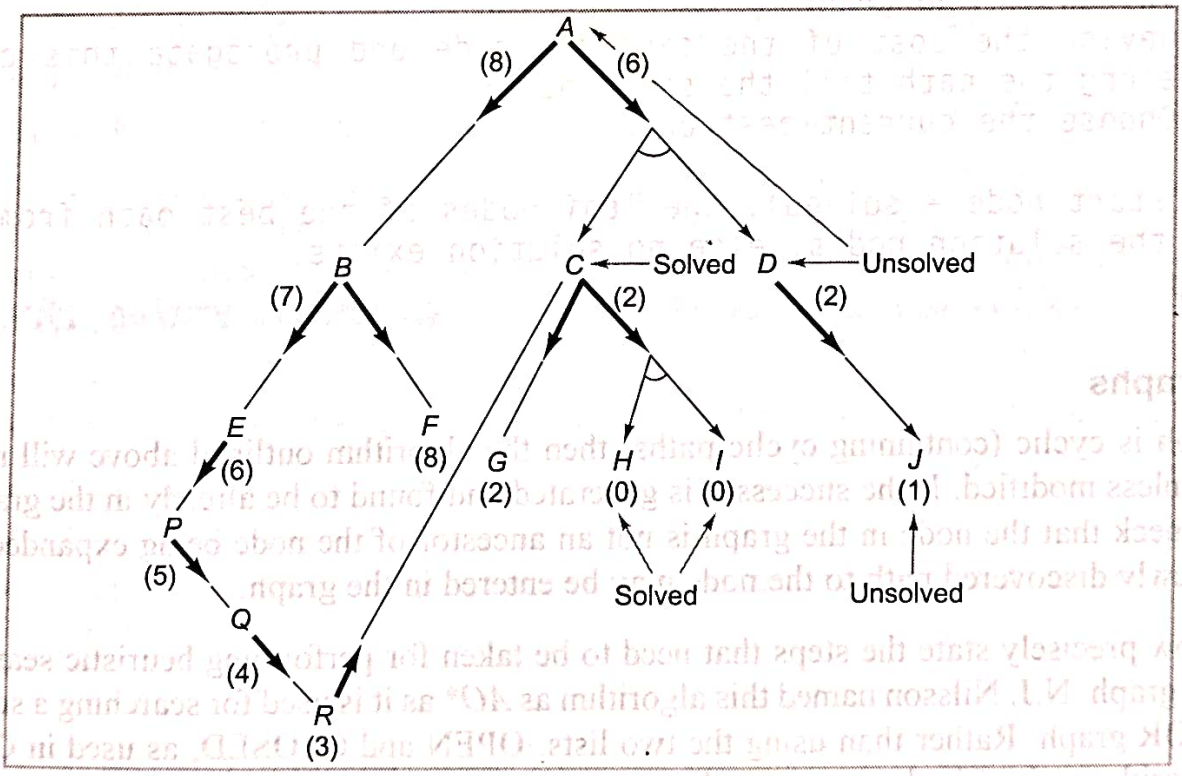


Figure 3.8 Non-Optimal Solution

It is not always necessary that the shorter path will lead to a solution. Sometimes, the longer path may prove to be better. Consider the example discussed above. Assume that node J is labelled as *unsolved* after the fourth cycle (Fig. 3.7), then D is also labelled as *unsolved*. As a result, A will also be labelled as *unsolved* through this path. Therefore, another path, even though having higher cost, will be tried. Suppose, node E is expanded to P, P to Q, Q to R, and finally, R to C (Fig. 3.8). We notice that node C is already solved and then in accordance with the labelling procedure, nodes R, Q, P, E, B, and A get labelled as *solved* with the total cost as 8. This new path to C is longer than the previous one coming directly from A to C. However, since the shorter path will not lead to a solution (as D is unsolvable), the longer path through R is better.

Algorithmic Steps for AND–OR Graphs

The following steps are followed while preparing the algorithm for handling AND–OR graphs:

- Initialize graph with *start node*.
- While (start node is not labelled as solved or (unsolved through all paths))
  {
  - Traverse the graph along the best path and expand all unexpanded nodes on it;
  - If node is terminal and the heuristic value of the node is 0, label it as *solved* else label it as *unsolved* and propagate the status up to the start node;
  - If node is non terminal, add its successors with the heuristic values in the graph;
  - Revise the cost of the expanded node and propagate this change along the path till the start node;
  - Choose the current best path

  }
- If (start node = solved), the leaf nodes of the best path from root are the solution nodes, else no solution exists;
- Stop

Cyclic Graphs

If the graph is cyclic (containing cyclic paths) then the algorithm outlined above will not work properly unless modified. If the successor is generated and found to be already in the graph, then we must check that the node in the graph is not an ancestor of the node being expanded. If not, then the newly discovered path to the node may be entered in the graph.

We can now precisely state the steps that need to be taken for performing heuristic search of an AND–OR graph. N.J. Nilsson named this algorithm as *AO** as it is used for searching a solution in an AND–OR graph. Rather than using the two lists, OPEN and CLOSED, as used in OR graph search algorithms given in the previous chapter, a single structure called graph G is used in *AO** algorithm. This graph represents the part of the search graph generated explicitly so far. Each

node in the graph will point down to its immediate successors as well as up to its immediate predecessor, and will have h value (an estimate of the cost of a path from current node to a set of solution nodes) associated with it. The value of g (cost from start to current node) is not computed at each node, unlike the case of A^* algorithm, as it is not possible to compute a single such value since there may be many paths to the same state. Moreover, such a value is not necessary because of the top–down traversing of the best-known path which guarantees that only nodes that are on the best path will be considered for expansion. So, h will be a good estimate for an AND–OR graph search instead of f. While propagating the cost upward to the parent node, the value of g will be added to h in order to obtain the revised cost of the parent. Further, we have to use the node-labelling (solved or unsolved) procedure described earlier for determining the status of the ancestor nodes on the best path. The detailed algorithm for AO^* is given below. The threshold value is chosen to take care of unsolved nodes.

Algorithm 3.1 *AO* Algorithm*

- Initially graph G consists of the start node. Call it START;
- Compute h(START);
- While (START is not labelled as either *solved* OR h(START) > threshold) DO

{

- Traverse the graph through the current best path starting from START;
- Accumulate the nodes on the best path which have not yet been expanded;
- Select one of those unexpanded nodes. Call it NODE and expand it;
- Generate successors of the NODE. If there are none, then assign *threshold* as the value of this NODE (to take care of nodes which are unsolvable) else for each SUCC which is not an ancestor of NODE do the following:

{

- Add SUCC to the graph G and compute h value for each SUCC;
- If h(SUCC) = 0 then it is a solution node and label it as *solved*;
- Propagate the newly discovered information up the graph (described below)

}

}

(Contd)

(Contd)

- If (START = *solved*) then path containing all the solved nodes (obtained from the graph) is the solution path else if h(START) > threshold, then solution cannot be found;
- Stop

The algorithm for propagation of the newly discovered information up to the graph is given below:

Algorithm 3.2 *Propagation of Information Up the Graph*

- Initialize L with NODE;
- While (L ≠ ϕ) DO
  {
  - Select a node from L, such that the selected node has no ancestor in G occurring in L /* this is to avoid cycle */ and call it CURRENT;
  - Remove the selected node from L;
  - Compute the cost of each arcs emerging from CURRENT
    - Cost of AND arc = sum of [h of each of the nodes at the end of the arc] + cost of arc itself;
  - Assign the minimum of the costs as revised value of CURRENT;
  - Mark the best path out of CURRENT (with minimum cost). Mark CURRENT node as *solved* if all of the nodes connected to it on the selected path have been labelled as *solved*;
  - If CURRENT has been marked *solved* or if the cost of CURRENT was just changed, then new status is propagated back up the root of the graph.
  - Add all the ancestors of CURRENT to L;
  }

Interaction between Sub-Goals

The *AO** algorithm discussed above fails to take into account an interaction between sub-goals which may lead to non-optimal solution. Let us explain the need of interaction between sub-goals.

In the graph shown in Fig. 3.9, we assume that both *C* and *D* ultimately lead to a solution. In order to solve *A* (AND node), both *B* and *D* have to be solved. The *AO** algorithm considers the solution of *B* as a completely separate process from the solution of *D*. Node *B* is expanded to *C* and *D* both of which eventually lead to solution. Using the *AO** algorithm, node *C* is solved in order to solve *B* as the path $B \rightarrow C$ seems to be better than path $B \rightarrow D$. We note that it is necessary to solve *D* in order to solve *A*. But we realize that node *D* will also solve *B* and hence there would be no need to solve *C*. We can clearly see that the cost of solving *A* through the path $A \rightarrow B \rightarrow D$ is 9, whereas in case of solving *B* through *C*, the cost of *A* comes out to be 12. Since *AO** does not consider such interactions, we may fail to find the optimal path for this problem.

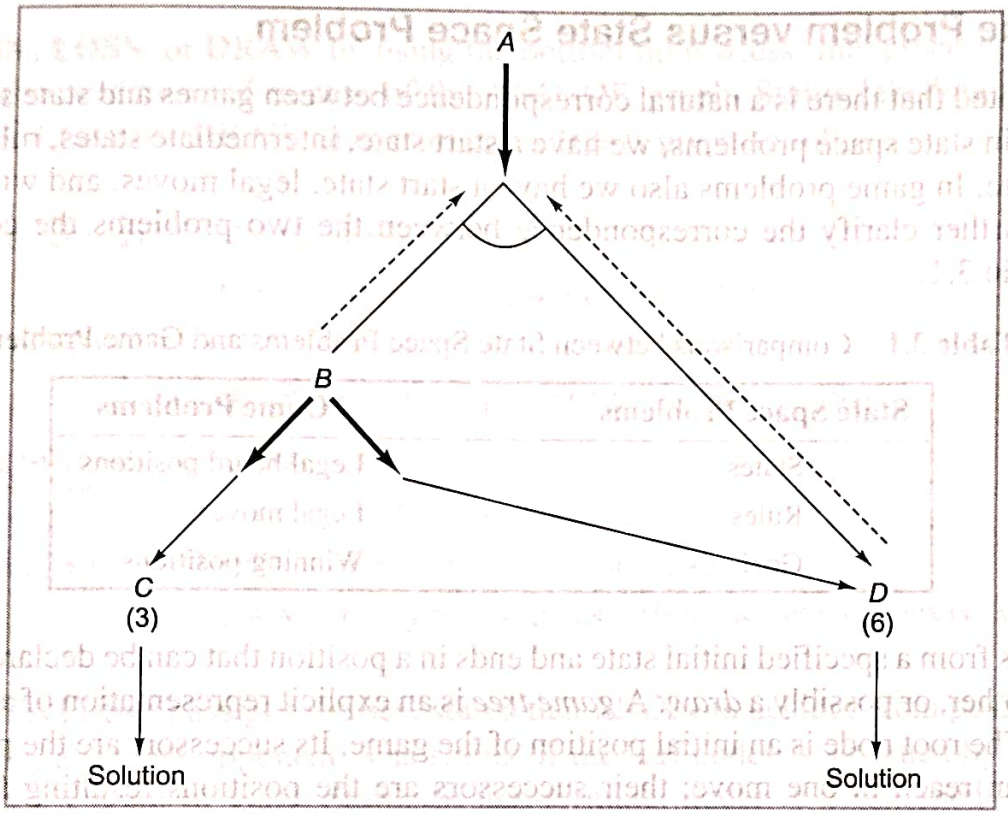


Figure 3.9 Interaction between Sub-Goals

3.3 Game Playing

Since the beginning of the AI paradigm, game playing has been considered to be a major topic of AI as it requires intelligence and has certain well-defined states and rules. A *game* is defined as a sequence of choices where each choice is made from a number of discrete alternatives. Each sequence ends in a certain outcome and every outcome has a definite value for the opening player. Playing games by human experts against computers has always been a great fascination. We will consider only two-player games in which both the players have exactly opposite goals. Games can be classified into two types: *perfect information games* and *imperfect information games*. Perfect information games are those in which both the players have access to the same information about the game in progress; for example, Checker, Tic–Tac–Toe, Chess, Go, etc. On the other hand, in imperfect information games, players do not have access to complete information about the game; for example, games involving the use of cards (such as Bridge) and dice. We will restrict our study to discrete and perfect information games. A game is said to be *discrete* if it contains a finite number of states or configurations.

In the following sections, we will develop search procedures for two-player games as they are common and easier to design. A typical characteristic of a game is to *look ahead* at future positions in order to succeed. Usually, the optimal solution can be obtained by exhaustive search if there are no constraints on time and space, but for most of the interesting games, such a solution is usually too inefficient to be practically used (Rich & Knight, 2003).

3.3.1 Game Problem versus State Space Problem

It should be noted that there is a natural correspondence between games and state space problems. For example, in state space problems, we have a start state, intermediate states, rules or operators, and a goal state. In game problems also we have a start state, legal moves, and winning positions (goals). To further clarify the correspondence between the two problems the comparisons are shown in Table 3.1.

Table 3.1 Comparisons between State Space Problems and Game Problems

State Space Problems	Game Problems
States	Legal board positions
Rules	Legal moves
Goal	Winning positions

A game begins from a specified initial state and ends in a position that can be declared a *win* for one, a *loss* for the other, or possibly a *draw*. A *game tree* is an explicit representation of all possible plays of the game. The root node is an initial position of the game. Its successors are the positions that the first player can reach in one move; their successors are the positions resulting from the second player's moves and so on. Terminal or leaf nodes are represented by WIN, LOSS, or DRAW. Each path from the root to a terminal node represents a different complete play of the game.

There is an obvious correspondence between a *game tree* and an AND–OR tree. The moves available to one player from a given position can be represented by the OR nodes, whereas the moves available to his opponent are the AND nodes. Therefore, in the game tree, one level is treated as an OR node level and other as AND node level from one player's point of view. On the other hand, in a general AND–OR tree, both types of nodes may be on the same level.

Game theory is based on the philosophy of minimizing the maximum possible loss and maximizing the minimum gain. In game playing involving computers, one player is assumed to be the computer, while the other is a human. During a game, two types of nodes are encountered, namely, MAX and MIN. The MAX node will try to maximize its own game, while minimizing the opponent's (MIN) game. Either of the two players, MAX and MIN, can play as the first player. We will assign the computer to be the MAX player and the opponent to be the MIN player. Our aim is to make the computer win the game by always making the best possible move at its turn. For this, we have to look ahead at all possible moves in the game by generating the complete game tree and then decide which move is the best for MAX. As a part of game playing, game trees labelled as MAX level and MIN level are generated alternately.

3.3.2 Status Labelling Procedure in Game Tree

We label each level in the game tree according to the player who makes the move at that point in the game. The leaf nodes are labelled as WIN, LOSS, or DRAW depending on whether they represent a win, loss, or draw position from MAX's point of view. Once the leaf nodes are assigned their WIN–LOSS–DRAW status, each non-terminal node in the game tree can be

labelled as WIN, LOSS, or DRAW by using the bottom-up process; this process is similar to the status labelling procedure used in case of the AND–OR graph. Status labelling procedure for a node with WIN, LOSS, or DRAW in case of game tree is given as follows:

- If j is a non-terminal MAX node, then

$$\text{STATUS}(j) = \begin{cases} \text{WIN}, & \text{if any of } j\text{'s successor is a WIN} \\ \text{LOSS}, & \text{if all } j\text{'s successor are LOSS} \\ \text{DRAW}, & \text{if any of } j\text{'s successor is a DRAW and none is WIN} \end{cases}$$

- If j is a non-terminal MIN node, then

$$\text{STATUS}(j) = \begin{cases} \text{WIN}, & \text{if all } j\text{'s successor are WIN} \\ \text{LOSS}, & \text{if any of } j\text{'s successor is a LOSS} \\ \text{DRAW}, & \text{if any of } j\text{'s successor is a DRAW and none is LOSS} \end{cases}$$

The function STATUS(j) assigns the best status that MAX can achieve from position j if it plays optimally against a perfect opponent. The status of the leaf nodes is assigned by the rules of the game from MAX's point of view. Status of non-terminal nodes is determined by the labelling procedure discussed above.

Solving a game tree implies labelling the root node with one of labels, namely: WIN (W), LOSS (L), or DRAW (D). There is an optimal playing strategy associated with each root label, which tells how that label can be guaranteed regardless of the way MIN plays. An optimal strategy for MAX is a sub-tree in which all nodes, starting from first MAX, are WIN.

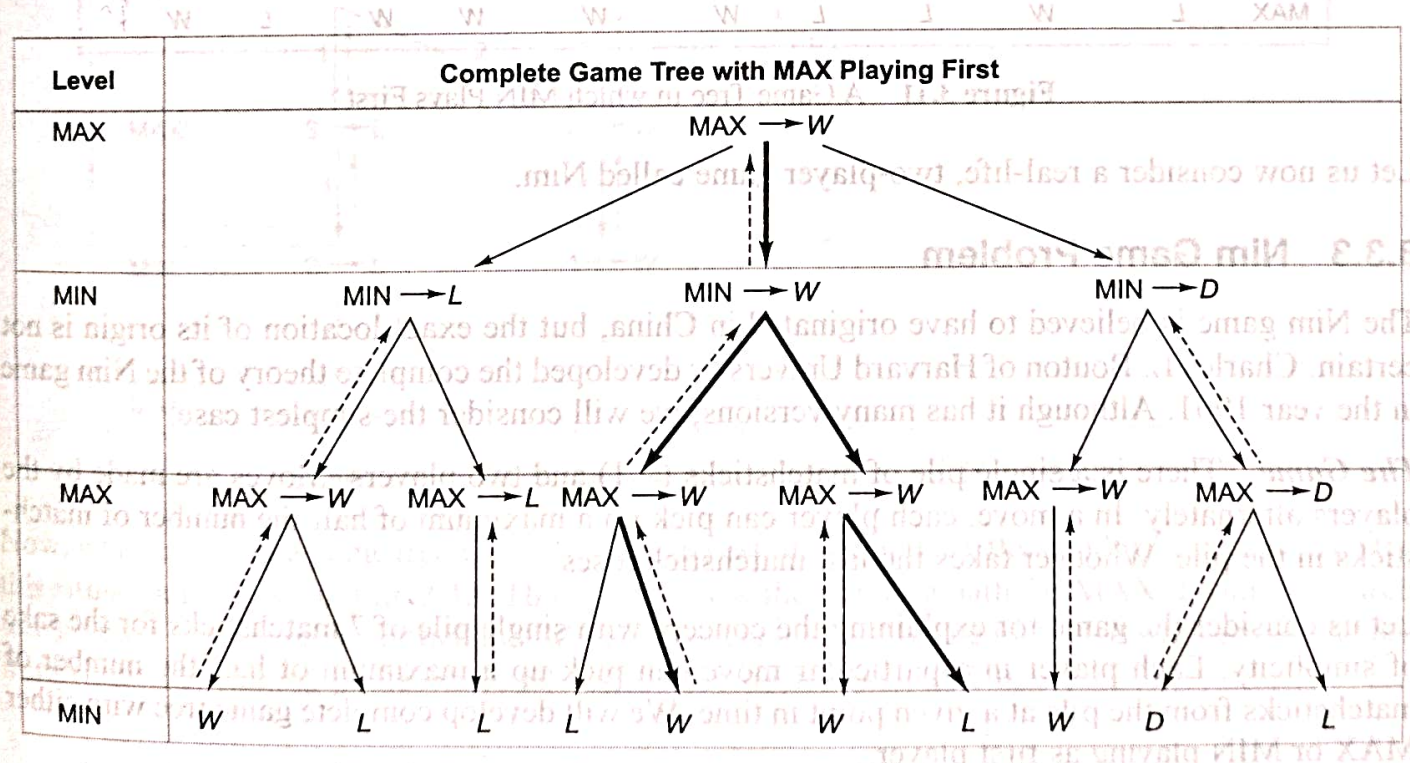


Figure 3.10 A Game Tree in which MAX Plays First

The hypothetical game tree shown in Fig. 3.10 is generated when the MAX player plays first. As mentioned earlier, the status of the leaf nodes is calculated in accordance with the rules of the game as *W*, *L*, or *D* from MAX's point of view. The status labelling procedure is used to propagate the status to non-terminal nodes till root and is shown by attaching the status to the node. Thick lines in the game tree show the winning paths for MAX, while dotted lines show the status propagation to the root node. It should be noted that all the nodes on the winning path are labelled as *W*.

The game tree shown in Fig. 3.11 is generated with the MIN player playing first. Here, MAX may lose the game if MIN chooses the first path.

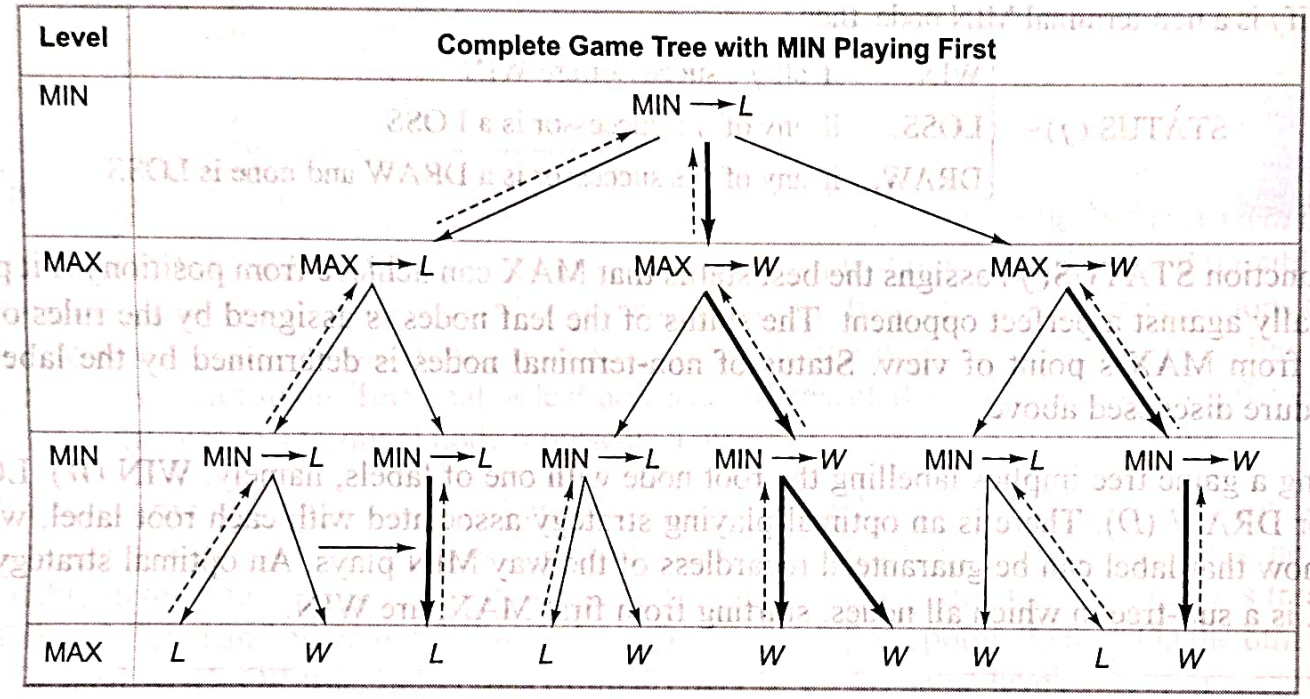


Figure 3.11 A Game Tree in which MIN Plays First

Let us now consider a real-life, two-player game called Nim.

3.3.3 Nim Game Problem

The Nim game is believed to have originated in China, but the exact location of its origin is not certain. Charles L. Bouton of Harvard University developed the complete theory of the Nim game in the year 1901. Although it has many versions, we will consider the simplest case.

The Game There is a single pile of matchsticks (> 1) and two players. Moves are made by the players alternately. In a move, each player can pick up a maximum of half the number of matchsticks in the pile. Whoever takes the last matchstick loses.

Let us consider the game for explaining the concept with single pile of 7 matchsticks for the sake of simplicity. Each player in a particular move can pick up a maximum of half the number of matchsticks from the pile at a given point in time. We will develop complete game tree with either MAX or MIN playing as first player.

The convention used for drawing a game tree is that each node contains the total number of sticks in the pile and is labelled as W or L in accordance with the status labelling procedure. The player who has to pick up the last stick loses. If a single stick is left at the MAX level then as a rule of the game, MAX node is assigned the status L, whereas if one stick is left at the MIN level, then W is assigned to MIN node as MAX wins. The label L or W have been assigned from MAX's point of view at leaf nodes. Arcs carry the number of sticks to be removed. Dotted lines show the propagation of status. The complete game tree for Nim with MAX playing first is shown in Fig. 3.12. We can see from this figure that the MIN player always wins irrespective of the move made by the first player.

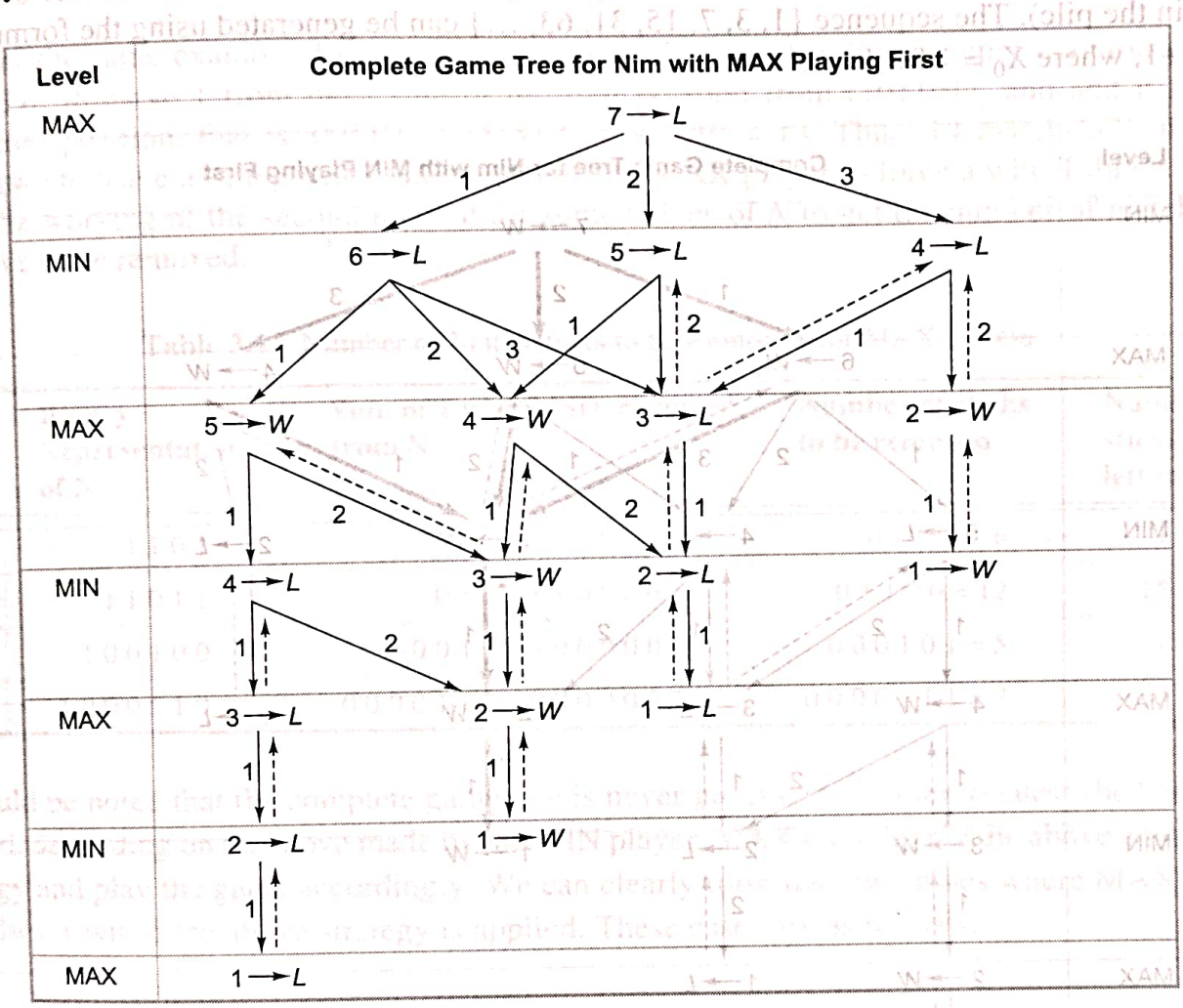


Figure 3.12 Game Tree for Nim in which MAX Plays First

Now, let us consider a game tree with MIN as the first player and see the results. The game tree for this situation is shown in Fig. 3.13. Thick lines show the winning path for MAX. From the search tree given in the figure, we notice that MAX wins irrespective of the moves of MIN player. Thick lines show the winning paths where all nodes have been labelled as W.

From the trees given in Figs 3.12 and 3.13, we can infer that the second player always wins regardless of the moves of the first player in this particular case.

Since the game is played between computer and a human being, we will now be discussing certain game-playing strategies with respect to a computer. In this case, MAX player is considered to be a computer program. Let us formulate some strategy for MAX player so that MAX can win the game.

Strategy If at the time of MAX player's turn there are N matchsticks in a pile, then MAX can force a win by leaving M matchsticks for the MIN player to play, where $M \in \{1, 3, 7, 15, 31, 63, \ldots\}$ using the rule of game (that is, MAX can pick up a maximum of half the number of matchsticks in the pile). The sequence $\{1, 3, 7, 15, 31, 63, \ldots\}$ can be generated using the formula $X_i = 2X_{i-1} + 1$, where $X_0 = 1$ for $i > 0$.

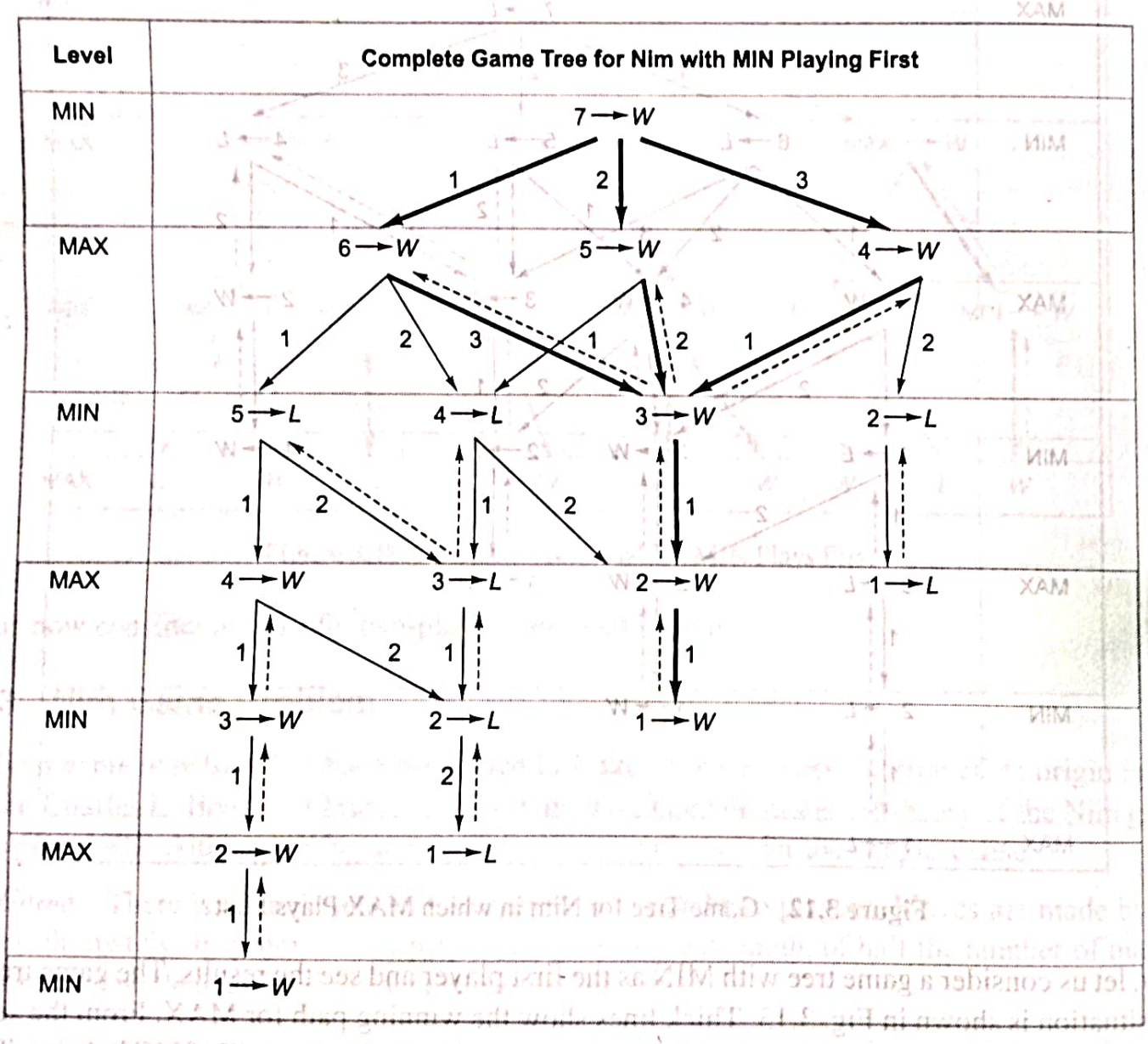


Figure 3.13 Game Tree for Nim in which MIN Plays First

Now we will formulate a *method* which will determine the number of matchsticks that have to be picked up by MAX player. There are two ways of finding this number:

- The first method is to look up from the sequence {1, 3, 7, 15, 31, 63, ...} and figure out the closest number less than the given number N of matchsticks in the pile. The difference between N and that number gives the desired number of sticks that have to be picked up. For example, if $N = 45$, the closest number to 45 in the sequence is 31, so we obtain the desired number of matchsticks to be picked up as 14 on subtracting 31 from 45. In this case we have to maintain a sequence {1, 3, 7, 15, 31, 63, ...).
- The second method is a simple one, in which the desired number is obtained by removing the most significant digit from the binary representation of N and adding it to the least significant digit position.

Consider the same example discussed above, where $N = 45$. The binary representation of 45 is $(101101)_2$. Remove 1 from most significant digit position from $(101101)_2$ and add it to least significant position, that is, 001101 + 000001 = 001110 = 14. Thus, 14 matchsticks must be withdrawn to leave a safe position and to enable the MAX player to force a win. Table 3.2 illustrates the working of the second method for some values of N to get the number of matchsticks that have to be removed.

Table 3.2 Number of Matchsticks to be Removed for MAX to Win

N	Binary Representation of N	Sum of 1 with MSD removed from N	Number of sticks to be removed	Number of sticks to be left in pile
13	1101	0101+0001	0110= 6	7
27	11011	01011+00001	01100=12	15
36	100100	000100+000001	000101=5	31
70	1000110	0000110+0000001	0000111=7	63

It should be noted that the complete game tree is never generated in order to guess the best path; instead, depending on the move made by the MIN player, MAX has to apply the above-mentioned strategy and play the game accordingly. We can clearly formulate two cases where MAX player will always win if the above strategy is applied. These cases are as follows:

- **CASE 1** MAX is the first player and initially there are $N \notin \{3, 7, 15, 31, 63, ...\}$ matchsticks.
- **CASE 2** MAX is the second player and initially there are $N \in \{3, 7, 15, 31, 63, ...\}$ matchsticks.

Validity of Cases for Winning of MAX Player

Let us show the validity of the cases mentioned above by considering suitable examples.

CASE 1 If MAX is the first player and $N \notin \{3, 7, 15, 31, 63, ...\}$, then MAX will always win.

Consider a pile of 29 sticks and let MAX be the first player. The complete game tree for this case is shown in Fig. 3.14. From the figure, it can be seen that MAX always wins. This case can be validated for any number of sticks $\notin \{3, 7, 15, 31, ...\}$. Thus, in this case, we can conclude by observing the figure that MAX is bound to win irrespective of how MIN plays.

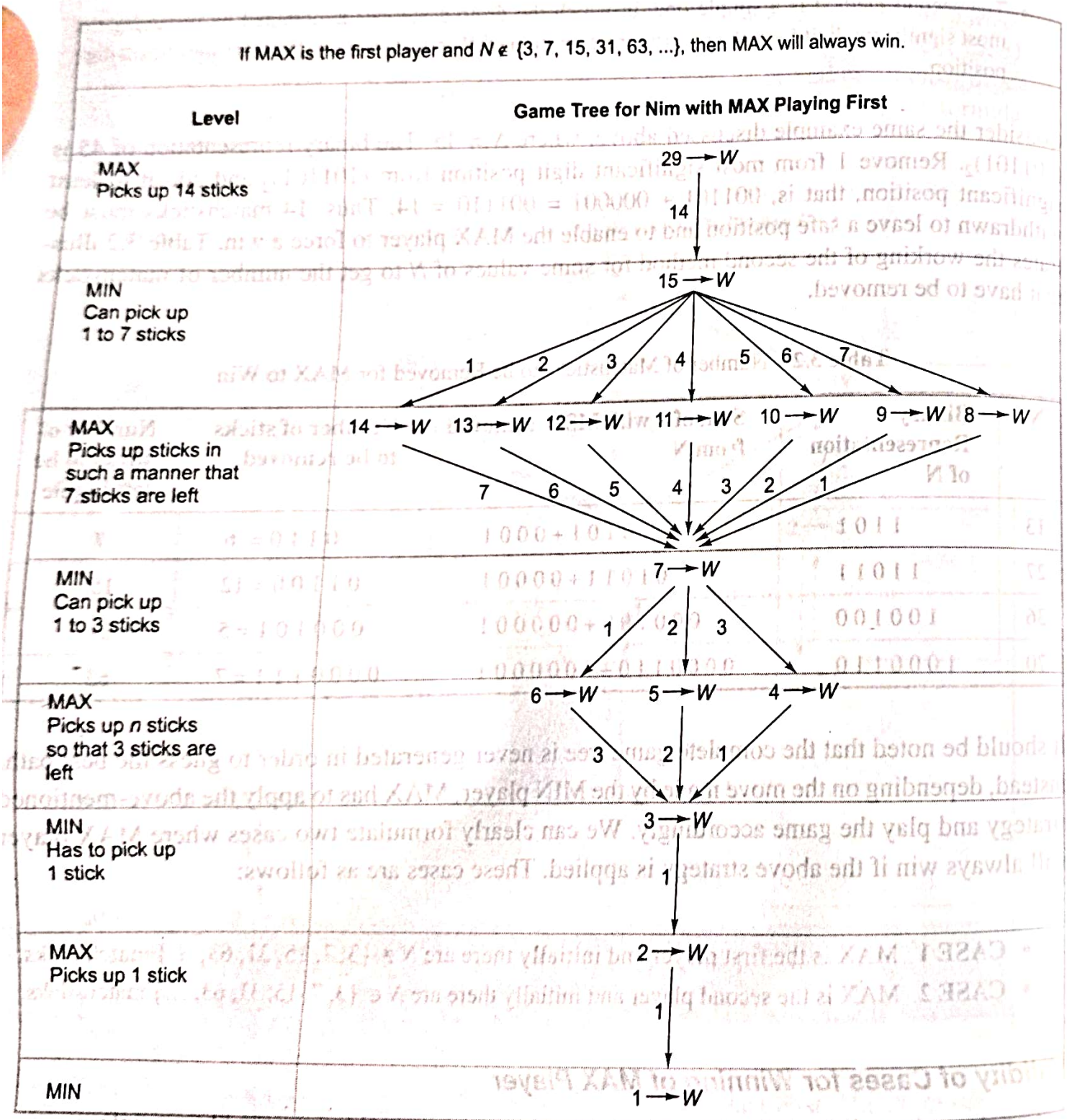


Figure 3.14 Validity of Case 1 (Example for $N = 29$)

CASE 2 If MAX is the second player and $N \in \{3, 7, 15, 31, 63, ...\}$, then MAX will always win.

Consider a pile of 15 sticks and let MAX be the second player. The complete game tree for this case is shown in Fig. 3.15. From the figure, it can be observed that MAX always wins. This case can be validated for any number of sticks $\in \{3, 7, 15, 31, 63, ...\}$.

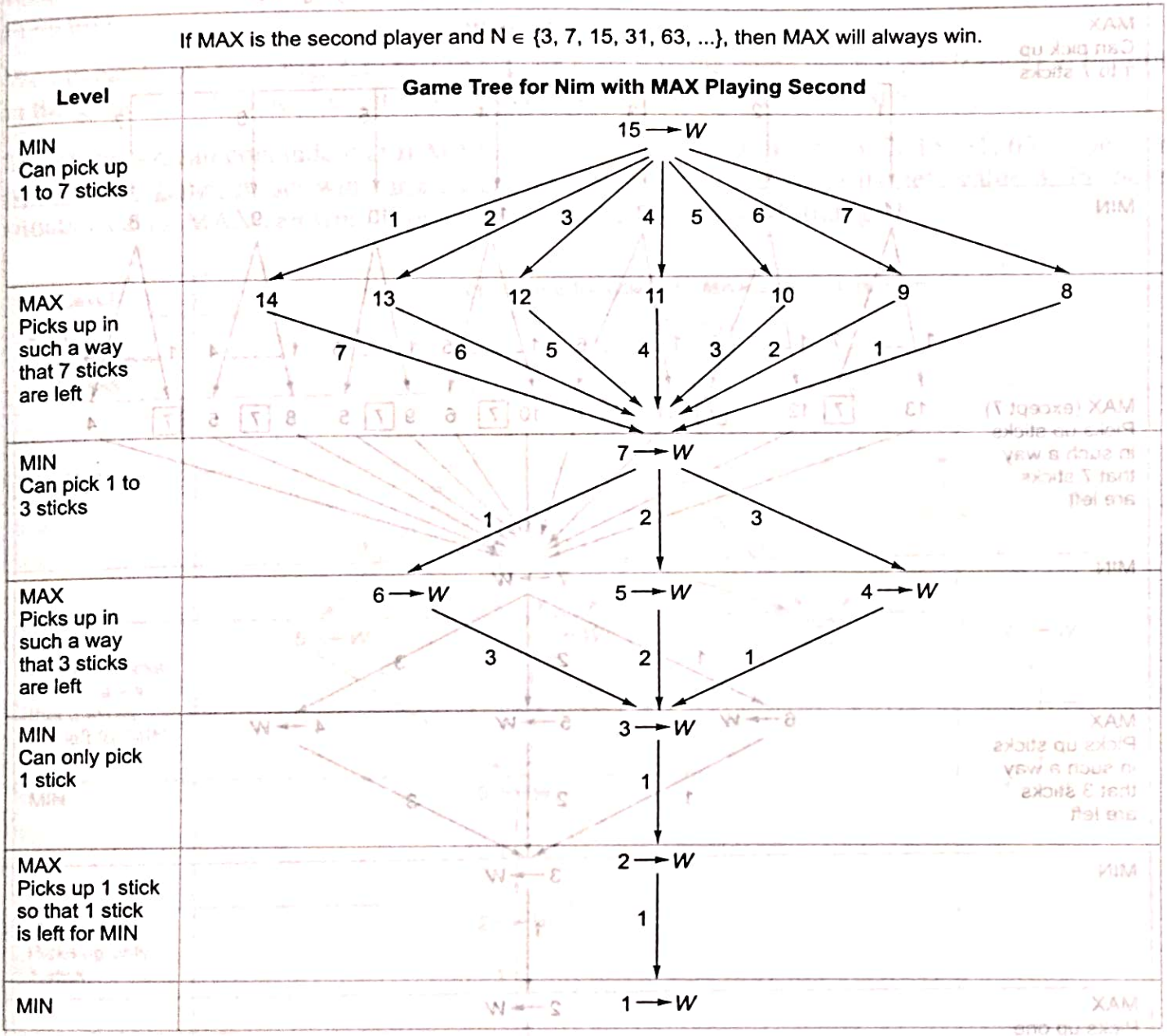


Figure 3.15 Validity of Case 2 (Example for $N = 15$)

There are other two cases where MAX can force a win if MIN is not playing optimally. These cases have been discussed below with examples. We do not have any clear strategy for these cases except that whenever possible MAX should leave M matchsticks for MIN to play, where $M \in \{1, 3, 7, 15, 31, 63, ...\}$.

If MAX is the first player and $N \in \{3, 7, 15, 31, 63, ...\}$ at root of the game, then MAX can force a win if MAX can leave any number out of the sequence {3, 7, 15, 31, 63, ...} for MIN. MAX *might* lose only in the case of getting 7.

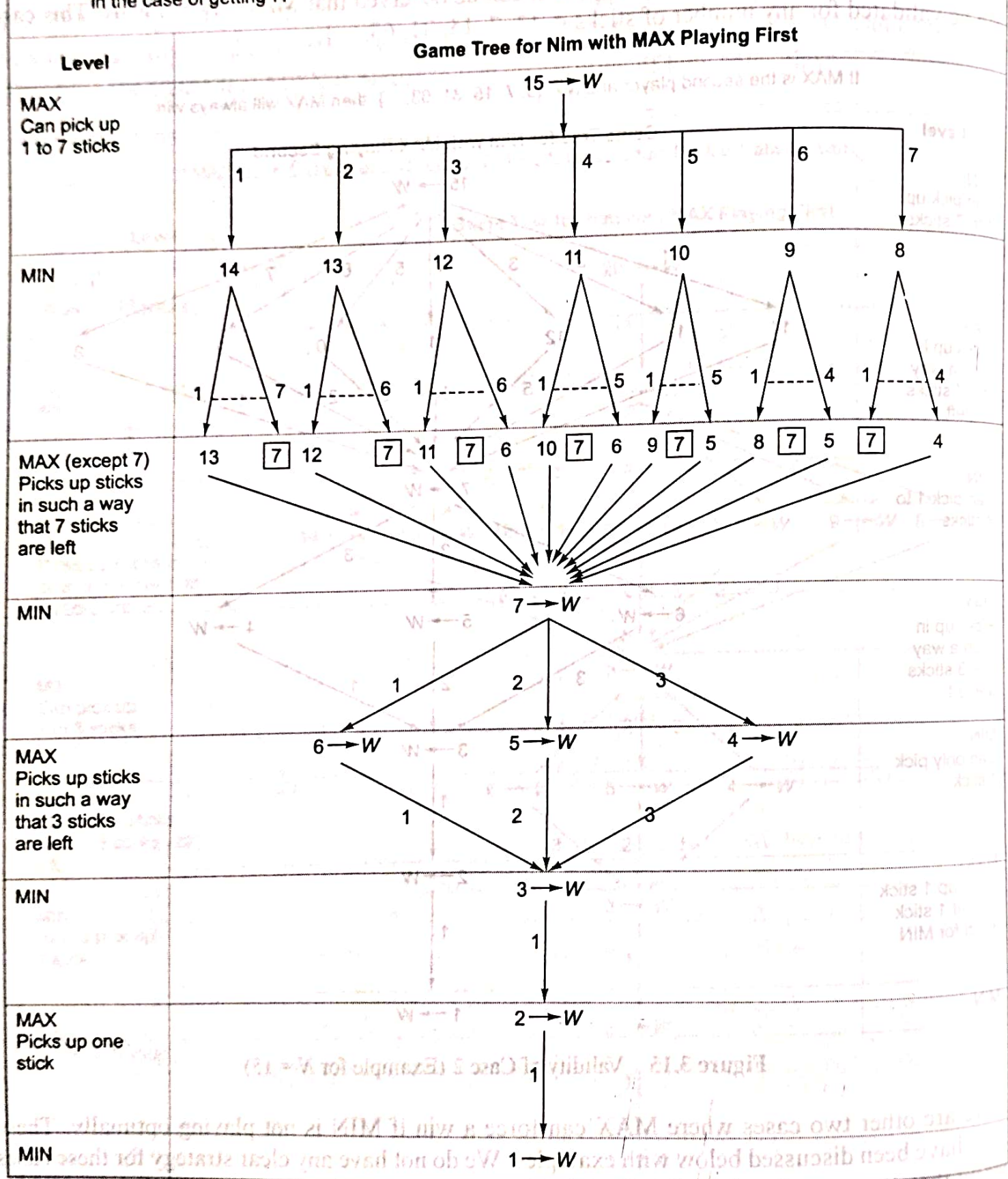


Figure 3.16 Validity of Case 3 (Example for $N = 15$)

CASE 3 If MAX is the first player and $N \in \{3, 7, 15, 31, 63, ...\}$ at root of the game, then MAX can force a win using the strategy mentioned above in all cases except when MAX gets a number from the sequence {3, 7, 15, 31, 63, ...} at its turn.

Assume that $N = 15$. Fig. 3.16 shows that MAX wins in all cases except when it gets 7 matchsticks in its turn.

We can easily see from Fig. 3.17 that MAX can even win the game when it gets 7 sticks at its turn in the game for all values except when it gets 3 sticks at its turn in the game.

Therefore, we can conclude that if MAX is playing with M sticks $\in$ {3, 7, 15, 31, 63,...} at any point in the game, it can win for all cases except for the case when its gets value 3. In these situations also, MAX can win if opponent is playing without any strategy.

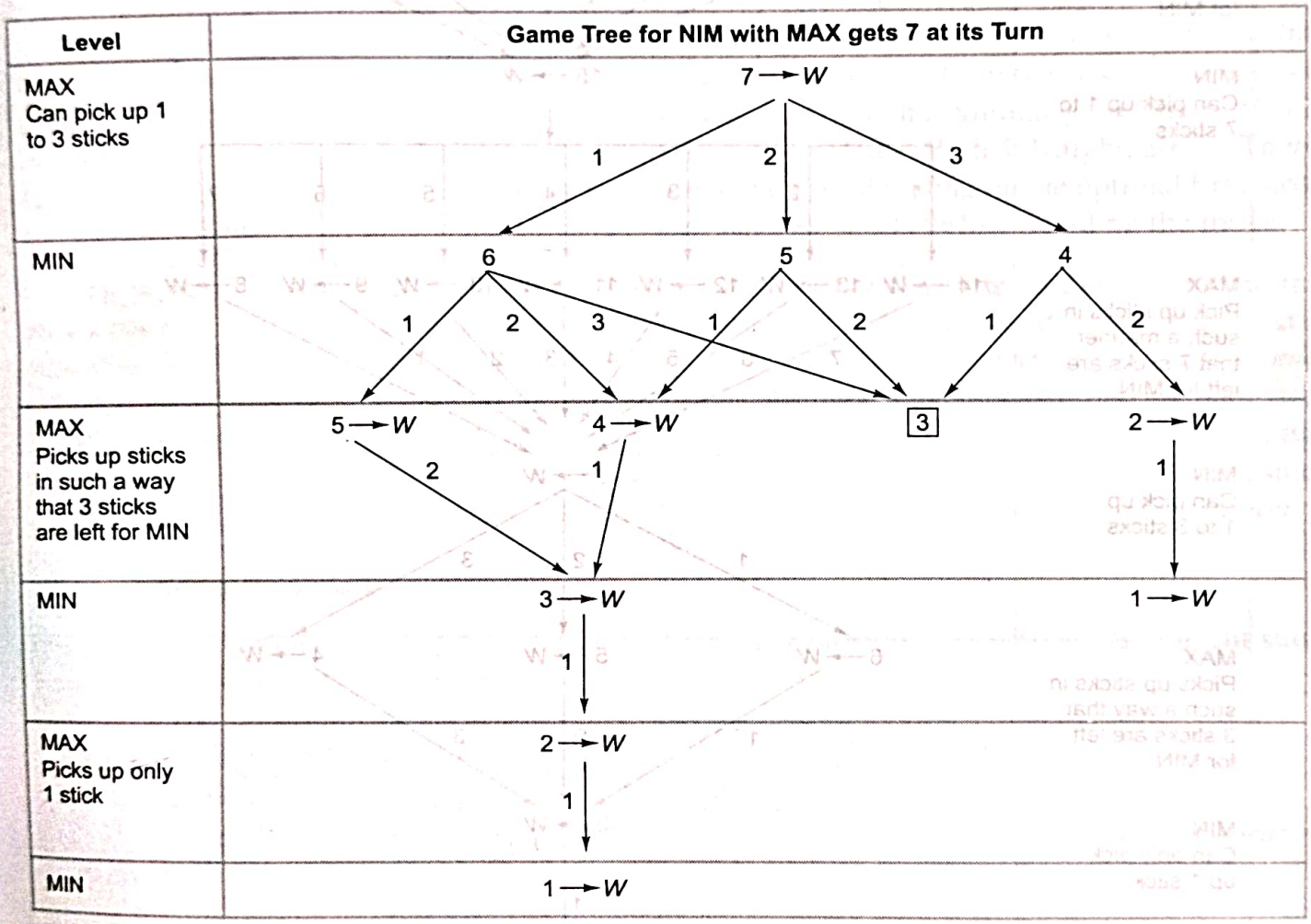


Figure 3.17 Game Tree

CASE 4 If MAX is the second player and $N \notin \{3, 7, 15, 31, 63, ...\}$ then MAX can force a win using the above-mentioned strategy in all cases except when it gets a number from the sequence {3, 7, 15, 31, 63, ...} at its turn.

If MAX is the second player and $N \notin \{3, 7, 15, 31, 63, ...\}$ then MAX can force a win in all cases except when it gets a number from the sequence {3, 7, 15, 31, 63, ...} at its turn.

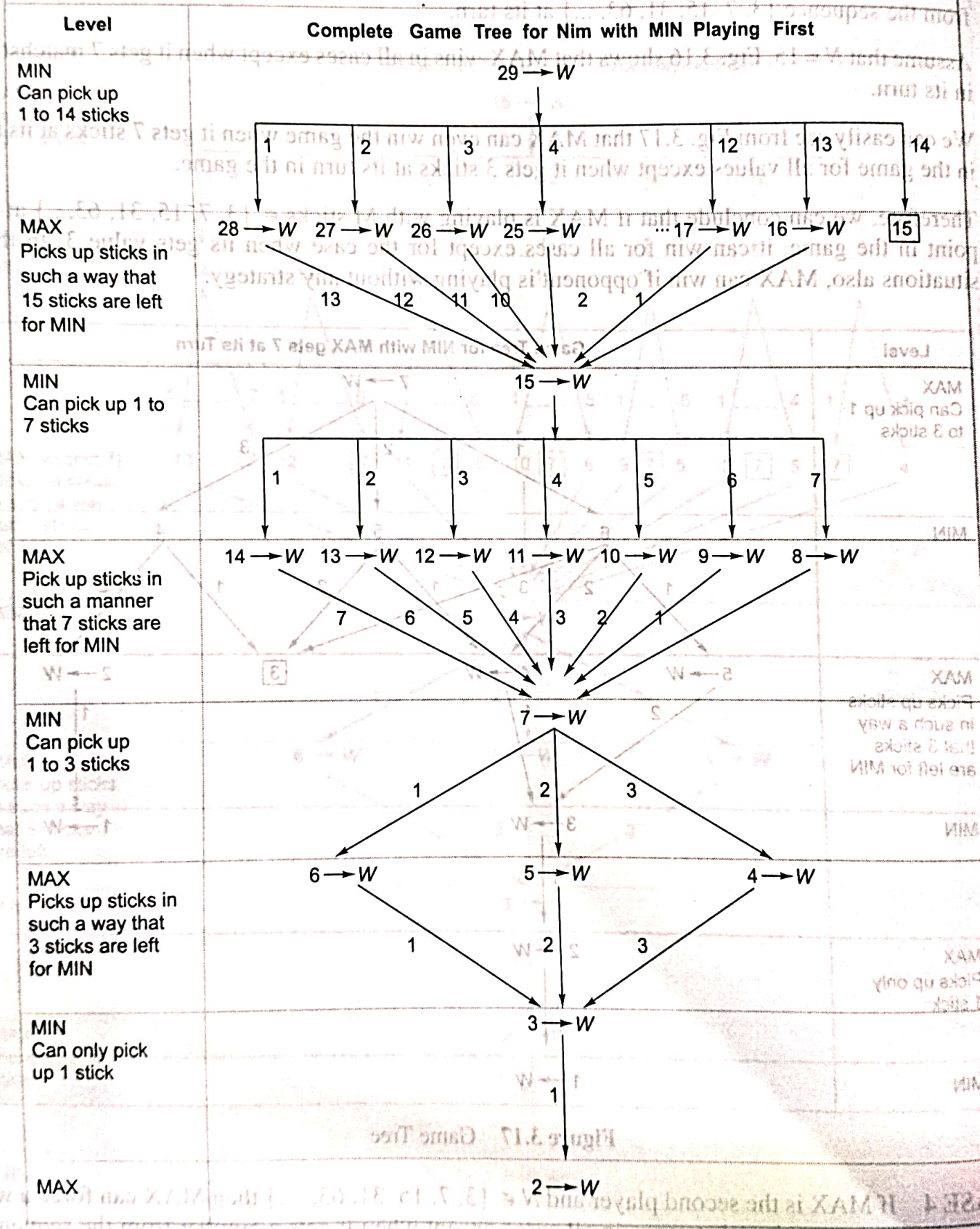


Figure 3.18 Validity of Case 4 (Example for $N = 29$)

Let us consider an example where $N = 29$ and let MIN be the first player. Figure 3.18 shows that MAX wins in all cases except when it gets 15 matchsticks at its turn. MAX might lose in case of it getting 15.

3.4 Bounded Look-Ahead Strategy and Use of Evaluation Functions

In all the examples discussed in the previous section, complete game trees were generated and with the help of the status labelling procedure, the status is propagated up to the root. Therefore, status labelling procedure requires the generation of the complete game tree or at least a sizable portion of it. In reality, for most of the games, trees of possibilities are too large to be generated and evaluated backward from the terminal nodes to root in order to determine the optimal first move. For example, in the game of Checkers, there are 1040 non-terminal nodes and we will require 1021 centuries if 3 billion nodes are generated every second. Similarly, in Chess, 10,120 non-terminal nodes are generated and will require 10,101 centuries (Rich & Knight, 2003). Therefore, this approach of generating complete game trees and then deciding on the optimal first move is not practical. One may think of looking ahead up to a few levels before deciding the move.

If a player can develop the game tree to a limited extent before deciding on the move, then this shows that the player is looking ahead; this is called *look-ahead* strategy. If a player is looking ahead n number of levels before making a move, then the strategy is called n-move *look-ahead*. For example, a look-ahead of 2 levels from the current state of the game means that the game tree is to be developed up to 2 levels from the current state. The game may be one -ply (depth one), two -ply (depth two), and so on. In this strategy, the actual value of a terminal state is unknown since we are not doing an exhaustive search. Hence, we need to make use of an *evaluation function*.

3.4.1 Using Evaluation Functions

The process of evaluation of a game is determined by the structural features of the current state. The steps involved in the evaluation procedure are as follows:

- The first step is to decide which features are of value in a particular game.
- The next step is to provide each feature with a range of possible values.
- The last step is to devise a set of weights in order to combine all the feature values into a single value.

In the absence of a practical way of evaluating the exact status of successor game states, we have to resort to heuristic approximation. It is important to understand that certain features in a game position contribute to its strength, while others tend to weaken it. A proper static evaluation function (heuristic) can convert all judgements about board situations into a single overall quality number. Therefore, the purpose of evaluation function is to provide the best judgement regarding

a position in the game in terms of the probability that the MAX player has a greater chance of winning from this position relative to other similar positions. The best evaluation functions are based on the experience of experts who are well-versed with the game.

Evaluation functions represent estimates of a given situation in the game rather than accurate calculations. This function provides numerical assessment of how favourable the game state is for MAX. We can use a convention in which a positive number indicates a good position for MAX, while a negative number indicates a bad position. The general strategy for MAX is to play in such a manner that it maximizes its winning chances, while simultaneously minimizing the chances of the opponent. The heuristic values of the nodes are determined at some level and then the status is accordingly propagated up to the root. The node which offers the best path is then chosen to make a move. For the sake of convenience, let us assume the root node to be a MAX node. Consider the one-ply and two-ply games shown in Fig. 3.19; the score of leaf nodes is assumed to be calculated using evaluation functions. The values at the nodes are backed up to the starting position.

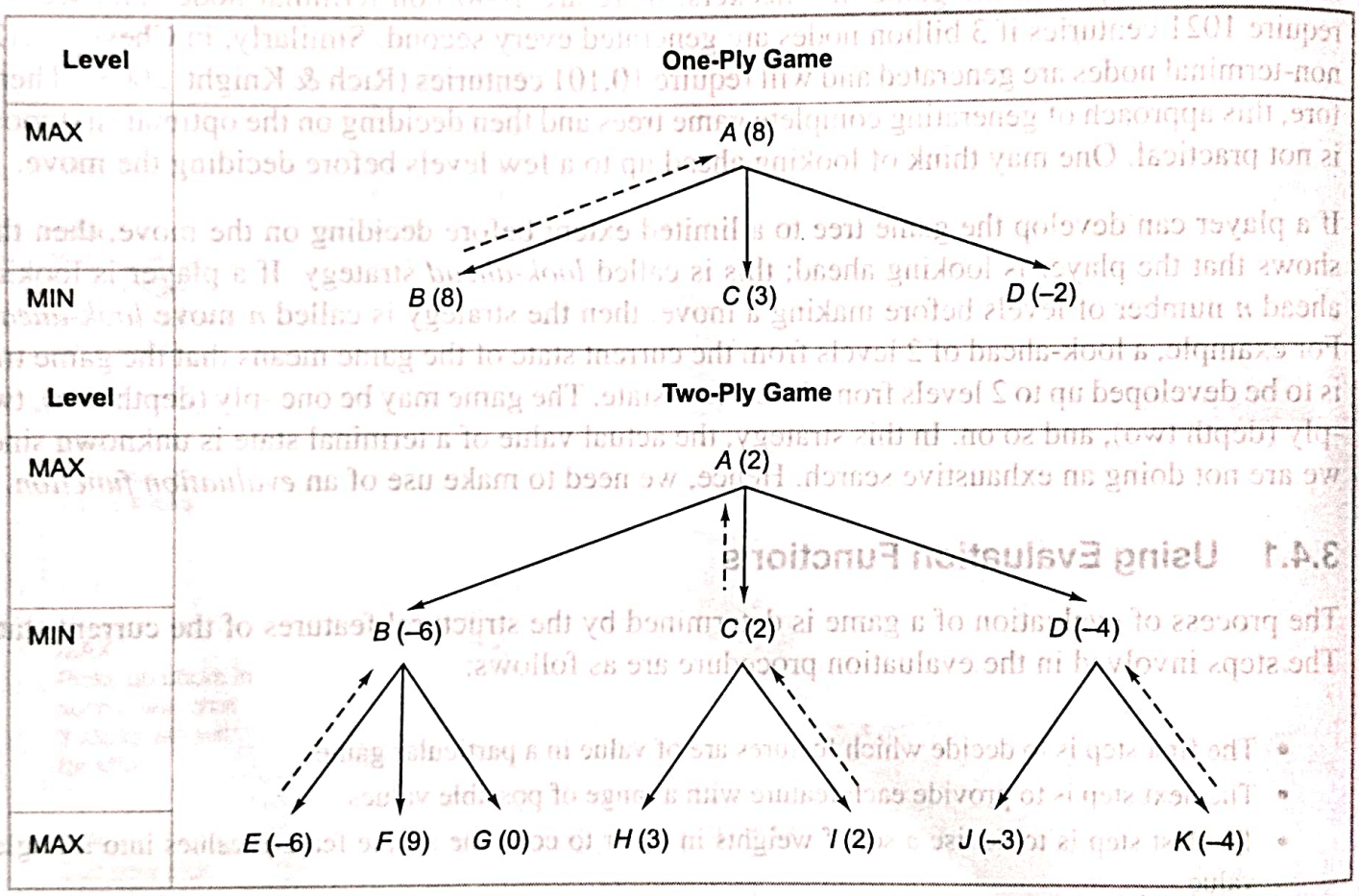


Figure 3.19 Using Evaluation Functions in One-Ply and Two-Ply Games

The procedure through which the scoring information travels up the game tree is called the MINIMAX procedure. This procedure represents a recursive algorithm for choosing the next move in two-player game. In this, a value is associated with each position or state of the game; this value is computed using an evaluation function and it denotes the extent to which it would be favourable for a player to reach that position. The player is then required to make a move which

maximizes the minimum value of the position resulting from the opponent's possible following moves. The MINIMAX procedure evaluates each leaf node (up to some fixed depth) using a heuristic evaluation function and obtains the values corresponding to the state. By convention of this algorithm, the moves which lead to a win of the MAX player are assigned a positive number, while the moves that lead to a win of the MIN player are assigned a negative number. MINIMAX procedure is a depth-first, depth-limited search procedure.

For a two-player, perfect-information game, the MINIMAX procedure can solve the problem provided there are sufficient computational resources for the same. This procedure assumes that each player takes the best option in each step. MINIMAX procedure starts from the leaves of the tree (which contain the final scores with respect to the MAX player) and then proceeds upwards towards the root. In the following section, we will describe MINIMAX procedure in detail.

3.4.2 MINIMAX Procedure

Lack of sufficient computational resources prevent the generation of a complete game tree; hence, the search depth is restricted to a constant. The estimated scores generated by a heuristic evaluation function for leaf nodes are propagated to the root using MINIMAX procedure, which is a recursive algorithm where a player tries to maximize its chances of a win while simultaneously minimizing that of the opponent. The player hoping to achieve a positive number is called the *maximizing player*, while the opponent is called the *minimizing player*. At each move, the MAX player will try to take a path that leads to a large positive number; on the other hand, the opponent will try to force the game towards situations with strongly negative static evaluations. A game tree shown in Fig. 3.20 is a hypothetical game tree where leaf nodes show heuristic values, whereas internal nodes show the backed-up values. This game tree is generated

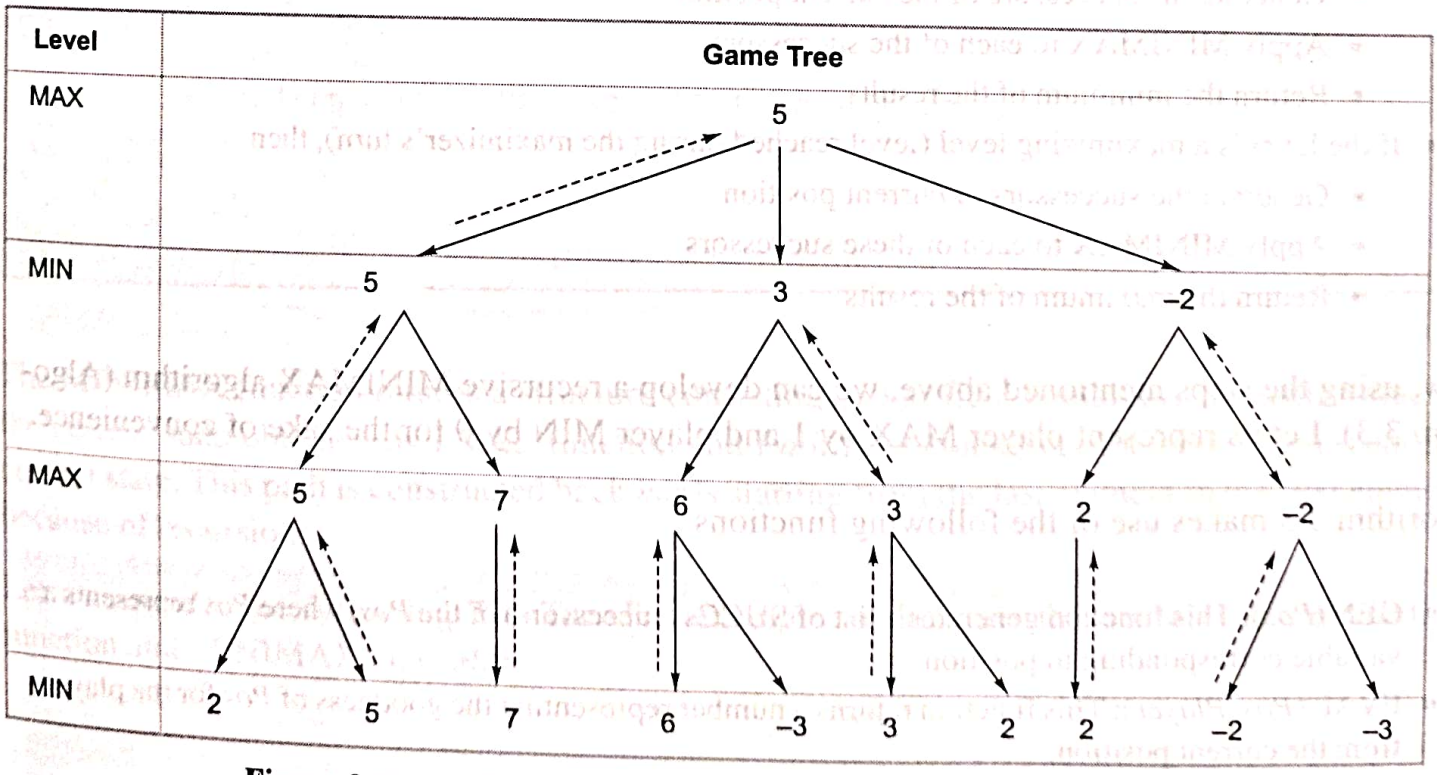


Figure 3.20 A Game Tree Generated using MINIMAX Procedure

using MINIMAX procedure up to a depth of three. At MAX level, maximum value of its successor nodes is assigned, whereas at MIN level, minimum value of its succesor nodes is assigned.

The MAX node moves to a state that has a score of 5 in the example considered above. After this, MIN will get a chance to play a move. Whenever MAX gets a chance to play, it will generate a game tree of depth 3 from the state generated by MIN player in order to decide its next move. The process will continue till the game ends with, hopefully, MAX player winning. The algorithmic steps of a MINIMAX procedure can be written in the following manner [Rich & Knight 2003]:

MINIMAX Procedure

The algorithmic steps of this procedure may be written as follows:

- Keep on generating the search tree till the limit, say depth *d* of the tree, has been reached from the current position.
- Compute the static value of the leaf nodes at depth *d* from the current position of the game tree using evaluation function.
- Propagate the values till the current position on the basis of the MINIMAX strategy.

MINIMAX Strategy

The steps in the MINIMAX strategy are written as follows:

- If the level is minimizing level (level reached during the minimizer's turn), then
  - Generate the successors of the current position
  - Apply MINIMAX to each of the successors
  - Return the minimum of the results
- If the level is a maximizing level (level reached during the maximizer's turn), then
  - Generate the successors of current position
  - Apply MINIMAX to each of these successors
  - Return the maximum of the results

Now, using the steps mentioned above, we can develop a recursive MINIMAX algorithm (Algorithm 3.3). Let us represent player MAX by 1 and player MIN by 0 for the sake of convenience.

Algorithm 3.3 makes use of the following functions:

- GEN (*Pos*): This function generates a list of SUCCs (successors) of the *Pos*, where *Pos* represents a variable corresponding to position.
- EVAL (*Pos, Player*): This function returns a number representing the goodness of *Pos* for the player from the current position.

- DEPTH (*Pos, Depth*): It is a Boolean function that returns *true* if the search has reached the maximum depth from the current position, else it returns *false*.

Algorithm 3.3 *MINIMAX Algorithm*

```
MINIMAX(Pos, Depth, Player)
{
  • If DEPTH(Pos, Depth) then return ({Val = EVAL(Pos, Player), Path =
    Nil})
  Else
  {
  • SUCC_List = GEN(Pos) ;
  • If SUCC_List = Nil then return ({Val = EVAL(Pos, Player), Path =
    Nil})
    Else
    {
      • Best_Val = Minimum value returned by EVAL function;
      • For each SUCC ∈ SUCC_List DO
      {
        • SUCC_Result = MINIMAX(SUCC, Depth + 1, ~Player);
        • NEW_Value = - Val of SUCC_Result ;
          • If NEW_Value > Best_Val then
          {
            • Best_Val = NEW_Value;
            • Best_Path = Add(SUCC, Path of SUCC_Result);
          };
      };
      • Return ({Val = Best_Val, Path = Best_Path});
    }
  }
}
```

The MINIMAX function returns a structure consisting of *Val* field containing heuristic value of the current state obtained by EVAL function and *Path* field containing the entire path from the current state. This path is constructed backwards starting from the last element to the first element because of recursion.

Let us consider the following Tic–Tac–Toe example to illustrate the use of static evaluation function and MINIMAX algorithm.

Tic–Tac–Toe Game

Tic–tac–toe is a two-player game in which the players take turns one by one and mark the spaces in a 3×3 grid using appropriate symbols. One player uses 'o' and other uses '×' symbol. The player who succeeds in placing three respective symbols in a horizontal, vertical, or diagonal row wins the game.

Let us define the static evaluation function f to a position P of the grid (board) as follows:

- If P is a win for MAX, then
  $f(P) = n$, (where n is a very large positive number)
- If P is a win for MIN, then
  $f(P) = -n$
- If P is not a winning position for either player, then
  $f(P)$ = (total number of rows, columns, and diagonals that are still open for MAX)
  − (total number of rows, columns, and diagonals that are still open for MIN)

Consider the symbol × for MAX and the symbol o for MIN and the board position P at some given point in time (Fig. 3.21). Assume that MAX starts the game. After MIN has played, it is the turn of MAX at board position P.

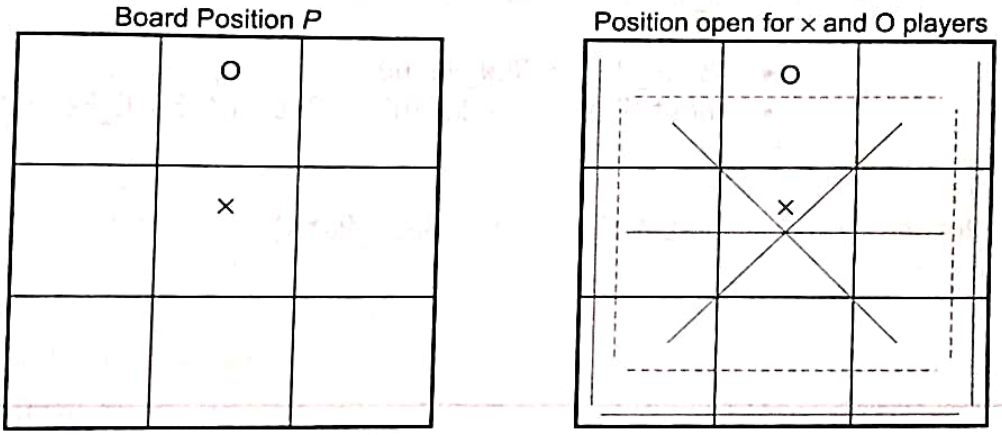


Figure 3.21 Board Position for a Game of Tic–Tac–Toe (Example)

Grey lines in Fig. 3.21 represent the positions that are open for × (MAX) and dotted grey lines represent those for o (MIN). We notice that both the diagonals, last two rows, and first and third columns are still open for MAX player (i.e., 6 lines are available for the symbol ×). On the other hand, the first and third rows and columns are open for MIN (i.e., 4 lines are available for the symbol o). Thus,

- Total number of rows, columns, and diagonals still open for MAX (thick lines) = 6
- Total number of rows, columns, and diagonals still open for MIN (dotted lines) = 4

$f(P)$ = (total number of rows, columns, and diagonals that are still open for MAX) – (total number of rows, columns, and diagonals that are still open for MIN) = 2

Therefore, the board position P has been evaluated by static evaluation function and assigned the value 2. Similarly, all the board positions after MIN player has played are evaluated and the move by MAX player to the best-secored board position is made.

Using the fact that MINIMAX algorithm is a depth-first process, we can improve its efficiency by using a *dynamic branch-and-bound* technique; in this technique, partial solutions that appear to be clearly worse than known solutions are abandoned. Under certain conditions, some branches of the tree can be ignored without changing the final score of the root.

3.5 Alpha–Beta Pruning

The strategy used to reduce the number of tree branches explored and the number of static evaluation applied is known as *alpha–beta pruning*. This procedure is also called *backward pruning*, which is a modified depth-first generation procedure. The purpose of applying this procedure is to reduce the amount of work done in generating useless nodes (nodes that do not affect the outcome) and is based on common sense or basic logic.

The alpha–beta pruning procedure requires the maintenance of two threshold values: one representing a *lower bound* (α) on the value that a maximizing node may ultimately be assigned (we call this alpha) and another representing *upper bound* (β) on the value that a minimizing node may be assigned (we call it beta). Each MAX node has an alpha value, which never decreases and each MIN node has a beta value, which never increases. These values are set and updated when the value of a successor node is obtained. The search is depth-first and stops at any MIN node whose beta value is smaller than or equal to the alpha value of its parent, as well as at any MAX node whose alpha value is greater than or equal to the beta value of its parent.

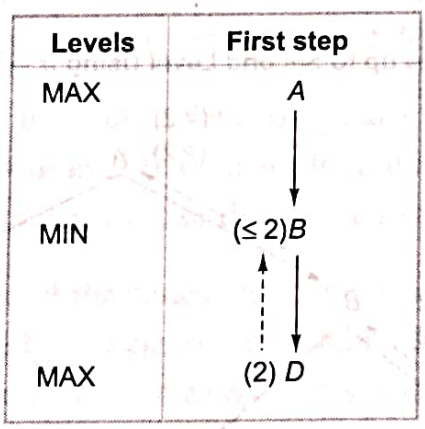


Figure 3.22 α–β Pruning Algorithm: Step 1

Let us consider the systematic development of a game tree and propagation of α and β values using alpha–beta (α–β) pruning algorithm up to second level stepwise in depth-first order. In Fig. 3.22, the MAX player expands root node A to B and suppose MIN player expands B to D.

Assume that the evaluation function generates $\alpha = 2$ for state D. At this point, the upper bound value $\beta = 2$ at state B and is shown as ≤ 2.

After the first step, we have to backtrack and generate another state E from B in the second step as shown in Fig. 3.23. The state E gets $\alpha = 7$ and since there is no further successor of B (assumed), the β value at state B becomes equal to 2. Once the β value is fixed at MIN level, the lower bound $\alpha = 2$ gets propagated to state A as ≥ 2.

In the third step, expand A to another successor C, and then expand C's successor to F with $\alpha = 1$. From Fig. 3.24 we note that the value at state C is ≤ 1 and the value of a root A cannot be less than 2; the path from A through C is not useful and thus further expansion of C is pruned. Therefore, there is no need to explore the right side of the tree fully as that result is not going to alter the move decision. Since there are no further successors of A (assumed), the value of root is fixed as 2, that is, $\alpha = 2$.

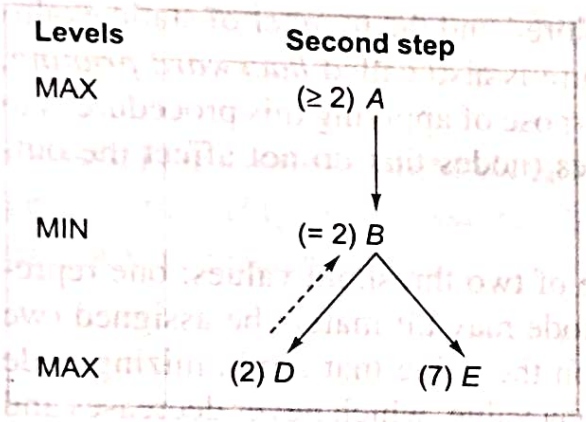


Figure 3.23 α–β Pruning Algorithm: Step 2

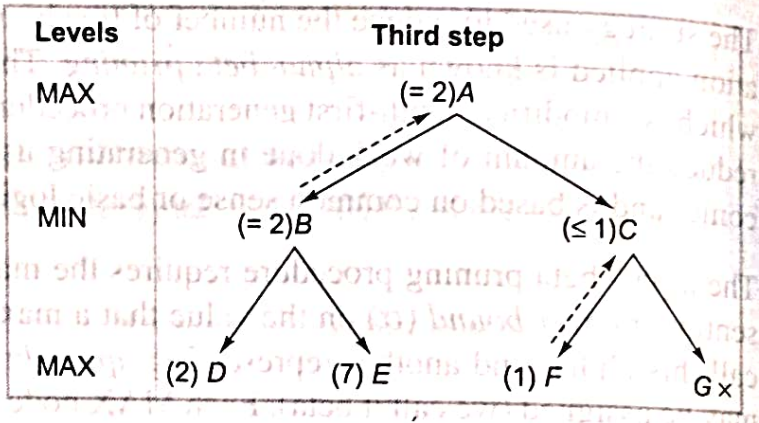


Figure 3.24 α–β Pruning Algorithm: Step 3

The complete diagram of game tree generation using (α–β) pruning algorithm is shown in Fig. 3.25 as follows:

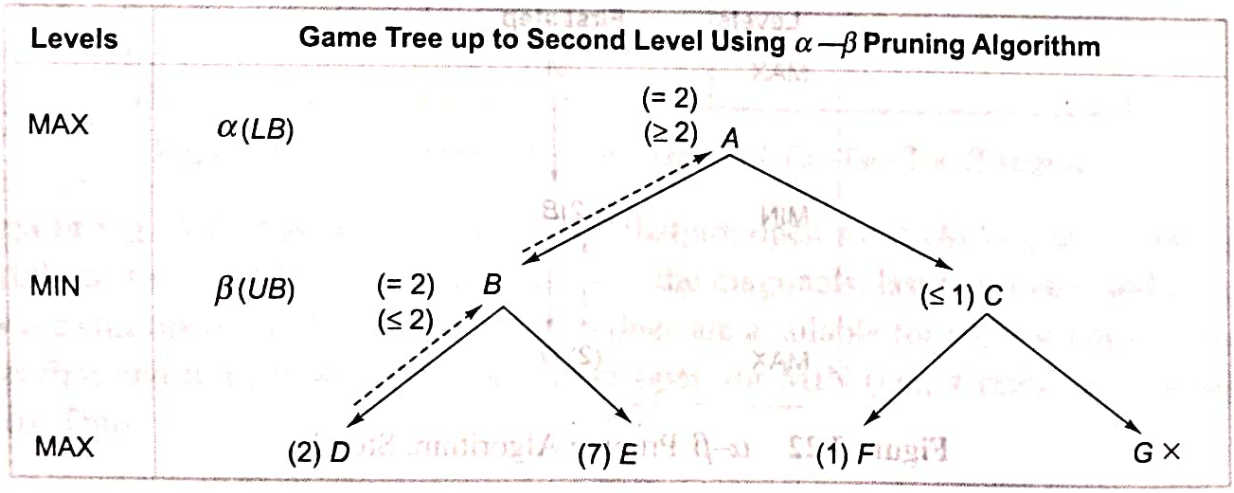


Figure 3.25 Game Tree Generation using α–β Pruning Algorithm

Let us consider an example of a game tree of depth 3 and branching factor 3 (Fig. 3.26). If the full game tree of depth 3 is generated then there are 27 leaf nodes for which static evaluation needs to be done. On the other hand, if we apply the α–β pruning, then only 16 static evaluations need to be made.

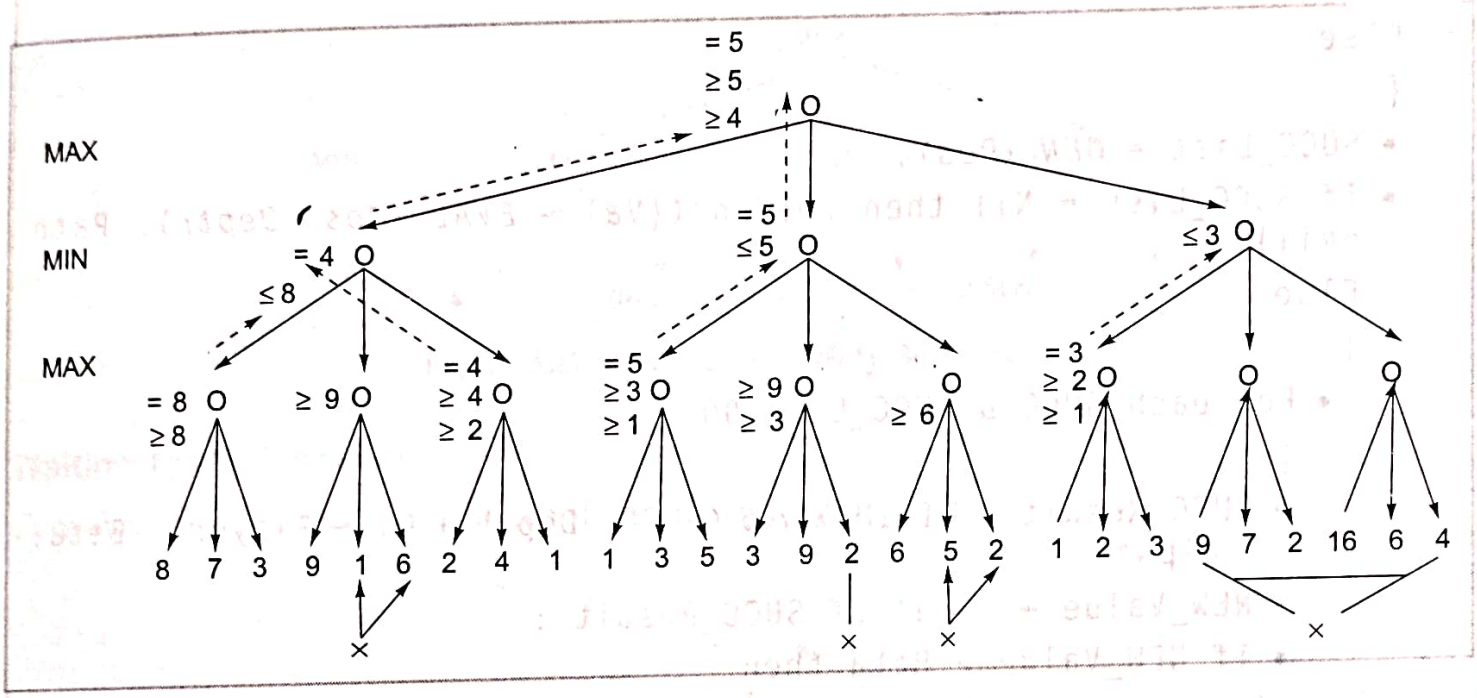


Figure 3.26 A Game Tree of Depth 3 and Branching Factor 3

Let us write the MINIMAX algorithm using α–β pruning concept (Algorithm 3.4). We notice that at the maximizing level, we use β to determine whether the search is cut-off, while at the minimizing level, we use α to prune the search. Therefore, the values of α and β must be known at maximizing or minimizing levels so that they can be passed to the next levels in the tree. Thus, each level should have both values: one to use and the other to pass to the next level. This procedure will therefore simply negate these values at each level.

The effectiveness of $\alpha - \beta$ pruning procedure depends greatly on the order in which the paths are examined. If the worst paths are examined first, then there will be no cut-offs at all. So, the best possible paths should be examined first, in case they are known in advance.

It has been shown by researchers that if the nodes are perfectly ordered, then the number of terminal nodes considered by search to depth d using $\alpha - \beta$ pruning is approximately equal to twice the number of nodes at depth $d/2$ without α–β pruning. Thus, the doubling of depth by some search procedure is a significant gain.

Algorithm 3.4 *MINIMAX algorithm using α–β pruning concept*

```
MINIMAX_αβ (Pos, Depth, Player, Alpha, Beta)
{
• If DEPTH (Pos, Depth) then return ({Val = EVAL (Pos, Depth), Path =
  Nil})
  Else
  {
  • SUCC_List = GEN (Pos).
  • If SUCC_List = Nil then return ({Val = EVAL (Pos, Depth), Path
    =Nil})
    Else
    {
     • For each SUCC μ SUCC_List 00
       {
       • SUCC_Result = MINIMAX_αβ (SUCC, Depth + 1, ~ Player, -Beta,
         -Alpha):
       • NEW_Value = - Val of SUCC_Result ;
       • If NEW_Value > Beta then
        {
          • Beta = NEW_Value;
          • Best_Path = Add(SUCC, Path of SUCC_Result);
        };
       • If Beta ³ Alpha then Return ({Val = Beta, Path = Best_Path)};
       } ;
   • Return ({Val = Beta, Path = Best_Path});
    }
  }
}
```

3.5.1 Refinements to α–β Pruning

In addition to α–β pruning, a number of modifications can be made to the MINIMAX procedure in order to improve its performance. One of the important factors that need to be considered during a search is when to stop going deeper in the search tree. Further, the idea behind α–β pruning procedure can be extended by cutting-off additional paths that appear to be slight improvements over paths that have already been explored (Rich & Knight, 2003).

Pruning of Slightly Better Paths

In Fig. 3.27, we see that the value 4.2 is only slightly better than 4, so we may terminate further exploration of node *C*. Terminating exploration of a sub-tree that offers little possibility for improvement over known paths is called *futility cut off*.

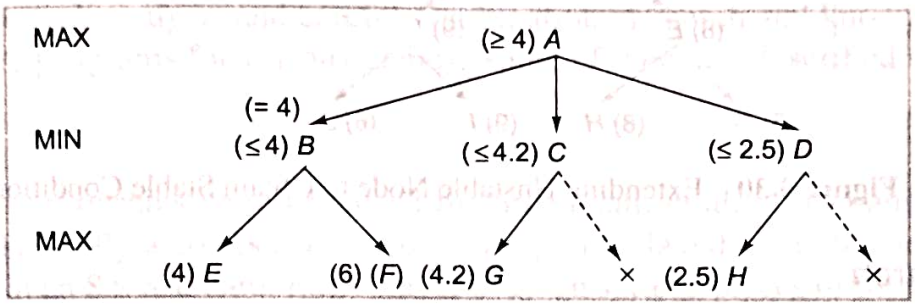


Figure 3.27 Pruning of Slightly Better Paths

Waiting for Quiescence

Consider a one-level deep game tree shown in Fig. 3.28.

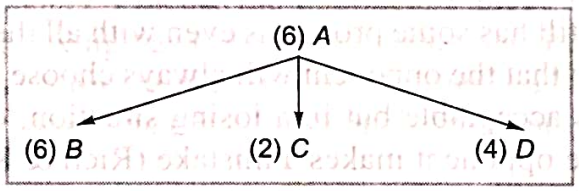


Figure 3.28 A One–Level Deep Game Tree

If node *B* is extended to one more level, we obtain a tree as shown in Fig. 3.29.

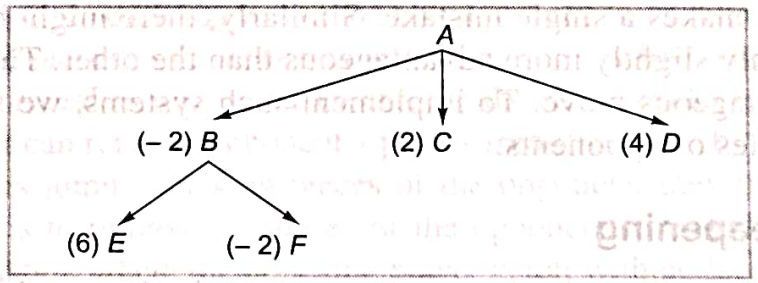


Figure 3.29 Expansion of Node *B*

From Fig. 3.29, we can see that our estimate of the worth of node *B* has changed. We can stop exploring the tree at this level and assign a value of – 2 to *B*, and therefore, decide that *B* is not a good move. Such short-term measures do not unduly influence our choice of moves, we should continue the search further until no such drastic change occurs from one level to the next or till the condition is stable. Such situation is called waiting for *quiescence* (Fig. 3.30). Thus, we should keep going deeper into the game tree till the condition is stable and then make a decision regarding the move. On following this method, we observe that *B* again looks like a reasonable move.

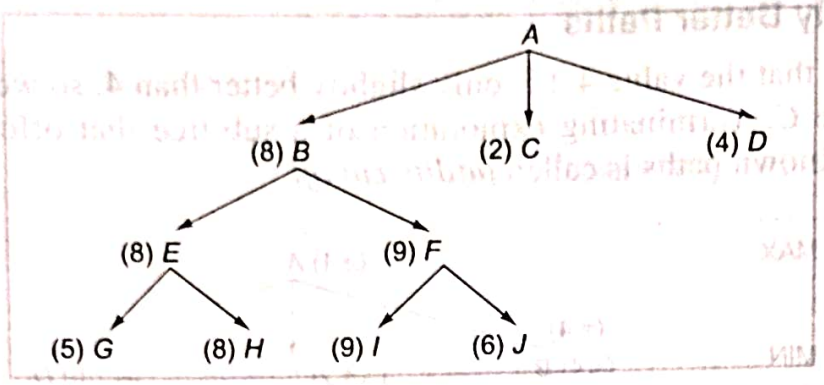


Figure 3.30 Extending Unstable Node to Obtain Stable Condition

Secondary Search

To provide a double check, explore a game tree to an average depth of more ply and on the basis of that, choose a particular move. The chosen branch is then to be further expanded up to two levels to make sure that it still looks good. This technique is called *secondary search.*

3.5.2 Alternative to α–β pruning MINIMAX Procedure

The MINIMAX procedure still has some problems even with all the refinements discussed above. It is based on the assumption that the opponent will always choose an optimal move. In a winning situation, this assumption is acceptable but in a losing situation, one may try other options and gain some benefit in case the opponent makes a mistake (Rich & Knight, 2003).

Suppose, we have to choose one move out of two possible moves, both of which may lead to bad situations for us if the opponent plays perfectly. MINIMAX procedure will always choose the bad move out of the two; however, here we can choose an option which is slightly less bad than the other. This is based on the assumption that the less bad move could lead to a good situation for us if the opponent makes a single mistake. Similarly, there might be a situation when one move appears to be only slightly more advantageous than the other. Then, it might be better to choose the less advantageous move. To implement such systems, we should have a model of individual playing styles of opponents.

3.5.3 Iterative Deepening

Rather than searching till a fixed depth in a given game tree, it is advisable to first search only one-ply, then apply MINIMAX to two-ply, then three-ply till the final goal state is searched (CHESS 5 uses this procedure). There is a good reason why iterative deepening is popular in case of games such as chess and others programs. In competitions, there is an average amount of time allowed per move. The idea that enables us to conquer this constraint is to do as much look-ahead as can be done in the available time. If we use iterative deepening, we can keep on increasing the look-ahead depth until we run out of time. We can arrange to have a record of the best move for a given look-ahead even if we have to interrupt our attempt to go one level deeper. This could not be done using (unbounded) depth-first search. With effective ordering, α–β pruning MINIMAX algorithm can prune many branches and the total search time can be decreased.

3.6 Two-Player Perfect Information Games

Even though a number of approaches and methods have been discussed in this chapter, it is still difficult to develop programs that can enable us to play difficult games. This is because every game requires thorough analysis and careful combination of search and knowledge. AI researchers have developed programs for various games. Some of them are described as follows:

Chess

The first two chess programs were proposed by Greenblatt, et al. (1967) and Newell & Simon (1972). Chess is basically a competitive two-player game played on a chequered board with 64 squares arranged in an 8×8 square. Each player is given sixteen pieces of the same colour (black or white). These include one king, one queen, two rooks, two knights, two bishops, and eight pawns. Each of these pieces moves in a unique manner. The player who chooses the white pieces gets the first turn. The objective of this game is to remove the opponent's king from the game. The player who fulfils this objective first is declared the winner. The players get alternate chances in which they can move one piece at a time. Pieces may be moved to either an unoccupied square or a square occupied by an opponent's piece; the opponent's piece is then captured and removed from the game. The opponent's king has to be placed in such a situation where the king is under immediate attack and there is no way to save it from the attack. This is known as *checkmate*. The players should avoid making moves that may place their king under direct threat (or check).

Checkers

Checkers program was first developed by Arthur Samuel (1959, 1967); it had a learning component to improve the performance of players by experience. Checkers (or draughts) is a two-player game played on a chequered 8×8 square board. Each player gets 12 pieces of the same colour (dark or light) which are placed on the dark squares of the board in three rows. The row closest to a player is called the *king row*. The pieces in the king row are called *kings*, while others are called *men*. Kings can move diagonally forward as well as backward. On the other hand, *men* may move only diagonally forward. A player can remove opponent's pieces from the game by diagonally jumping over them. When *men* pieces jump over *king* pieces of the opponent, they transform into kings. The objective of the game is to remove all pieces of the opponent from the board or by leading the opponent to such a situation where the opposing player is left with no legal moves.

Othello

Othello (also known as Reversi) is a two-player board game which is played on an 8×8 square grid with pieces that have two distinct bi-coloured sides. The pieces typically are shaped as coins, but each possesses a light and a dark face, each face representing one player. The objective of the game is to make your pieces constitute a majority of the pieces on the board at the end of the game, by turning over as many of your opponent's pieces as possible. Advanced computer programs for Othello were developed by Rosenbloom in 1982 and subsequently Lee & Mahajan in 1990 leading to it becoming a world championship level game.

Go

It is a strategic two-player board game in which the players play alternately by placing black and white stones on the vacant intersections of a 19 × 19 board. The object of the game is to control a larger part of the board than the opponent. To achieve this, players try to place their stones in such a manner that they cannot be captured by the opposing player. Placing stones close to each other helps them support one another and avoid capture. On the other hand, placing them far apart creates an influence across a larger part of the board. It is a strategy that enables players to play a defensive as well as an offensive game and choose between tactical urgency and strategic planning. A stone or a group of stones is captured and removed if it has no empty adjacent intersections, that is, it is completely surrounded by stones of the opposing colour. The game is declared over and the score is counted when both players consecutively pass on a turn, indicating that neither side can increase its territory or reduce that of its opponent's.

Backgammon

It is also a two-player board game in which the playing pieces are moved using dice. A player wins by removing all of his pieces from the board. Although luck plays an important role, there is a large scope for strategy. With each roll of the dice a player must choose from numerous options for moving his checkers and anticipate the possible counter-moves by the opponent. Players may raise the stakes during the game. Backgammon has been studied with great interest by computer scientists. Similar to chess, advanced backgammon software has been developed which is capable of beating world-class human players. Backgammon programs with high level of competence were developed by Berliner in 1980 and by Tesauro & Sejnowski in 1989.

Exercises

3.1 Figure 3.31 shows a partial AND–OR graph with static evaluation values shown along with leaf nodes. Which is the best path from the root node *A*?

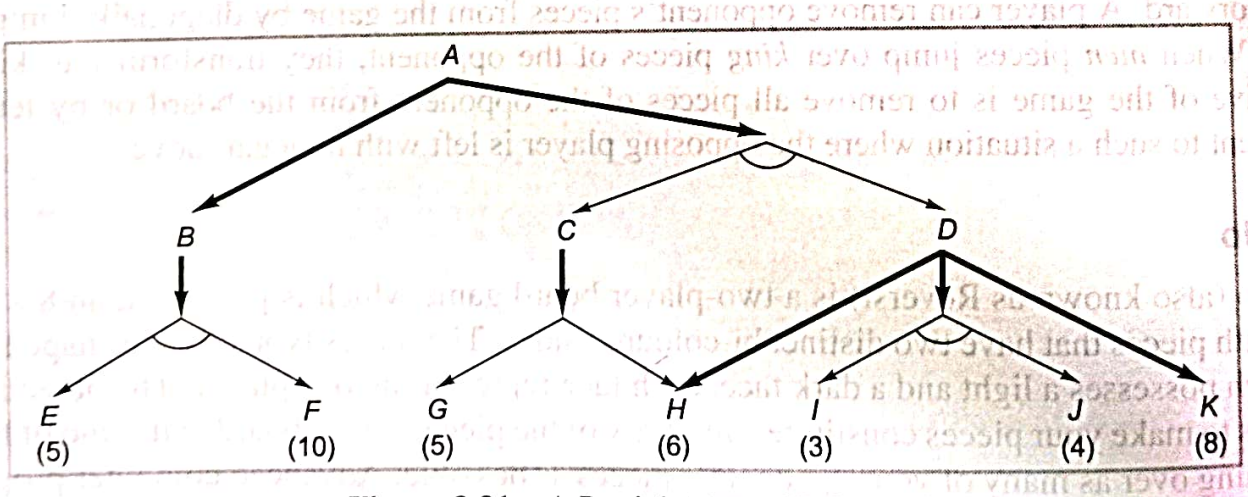


Figure 3.31 A Partial AND–OR Graph

3.2 The algorithm *AO** may not find the optimal solution. Show with the help of an example.

3.3 Use *AO** algorithm to transform the numeral 6 into a string of 1's using the following rules. Assume that the cost of *k*-connectors is *k*-units and that the value of a static evaluation function at a node labelled by numeral 1 is zero and at a node labelled by *n* is *n*. Draw AND–OR graphs for the following rules:

i. $6 \rightarrow 5, 1$; $6 \rightarrow 4, 2$; $5 \rightarrow 4, 1$; $5 \rightarrow 3, 2$;
$4 \rightarrow 3, 1$; $4 \rightarrow 2, 2$; $3 \rightarrow 2, 1$; $2 \rightarrow 1, 1$;

ii. $6 \rightarrow 3, 3$; $6 \rightarrow 4, 2$; $4 \rightarrow 2, 2$; $4 \rightarrow 3, 1$;
$3 \rightarrow 2, 1$; $2 \rightarrow 1, 1$;

3.4 Implement an AND–OR graph algorithm for a generic problem where static evaluation function generates random values within a specified range. A node with 0 value is assumed to be solved.

3.5 Show the computation for first three-player moves in tic-tac-toe game using α–β pruning.

3.6 Consider the game tree given in Fig. 3.32 in which static scores are shown along the leaf nodes from the first player's (MAX) point of view.

i. What move should the first player choose?

ii. What nodes would need to be examined using α–β pruning algorithm assuming that nodes are examined in left to right order?

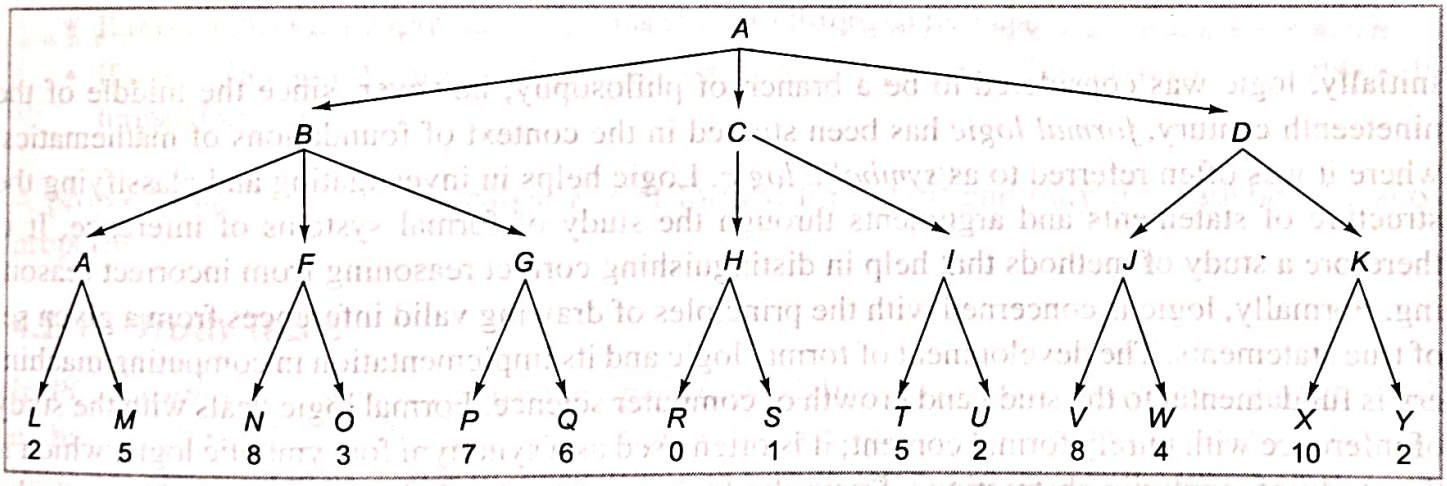


Figure 3.32 Game Tree

3.7 Consider a Nim game with 16 sticks in the pile. Draw game trees with

i. MAX as the first player

ii. MIN as the first player

3.8 Develop an algorithm for Nim game explained in the chapter.

3.9 Write an algorithm for iterative deepening for a hypothetical two-player game. Use static evaluation function to be random function.

3.10 Collect detailed information about the following games and implement them:

i. Othello

ii. Go

iii. Backgammon

iv. Checker

4

Logic Concepts and Logic Programming

4.1 Introduction

Initially, logic was considered to be a branch of philosophy; however, since the middle of the nineteenth century, *formal logic* has been studied in the context of foundations of mathematics, where it was often referred to as *symbolic logic*. Logic helps in investigating and classifying the structure of statements and arguments through the study of formal systems of inference. It is therefore a study of methods that help in distinguishing correct reasoning from incorrect reasoning. Formally, logic is concerned with the principles of drawing valid inferences from a given set of true statements. The development of formal logic and its implementation in computing machinery is fundamental to the study and growth of computer science. Formal logic deals with the study of inference with purely formal content; it is often used as a synonym for symbolic logic, which is the study of symbolic abstractions. Symbolic logic is often divided into two branches, namely, *propositional logic* and *predicate logic*. In the study of logic, *a proposition* refers to a declarative statement that is either true or false (but not both) in a given context. One can infer a new proposition from a given set of propositions in the same context using logic. An extension to symbolic logic is *mathematical logic*, which is particularly concerned with the study of proof theory, set theory, model theory, and recursion theory. The field of logic is also concerned with core topics such as the study of validity, consistency, and inconsistency. Logical systems should possess properties such as consistency, soundness, and completeness. Consistency implies that none of the theorems of the system should contradict each other; soundness means that the inference rules shall never allow a false inference from true premises. If a system is sound and its axioms are true then its theorems are also guaranteed to be true. Completeness means that there are no true sentences in the system that cannot be proved in the system.

In this chapter, the concepts of propositional calculus and logic are introduced along with four formal methods concerned with proofs and deductions. The concept of propositional logic has also been extended to first-order predicate logic followed by the evolution of logic programming which forms the basis of the logic programming language called PROLOG (this language has been described in detail in the next chapter) (Kaushik S., 2002).

4.2 Propositional Calculus

Propositional calculus (PC) refers to a language of propositions in which a set of rules are used to combine simple propositions to form compound propositions with the help of certain logical operators. These logical operators are often called *connectives*; examples of some connectives are *not* (~), *and* (Λ), *or* (V), *implies* (→), and *equivalence* (↔). In PC, it is extremely important to understand the concept of a well-formed formula. A well-formed formula is defined as a symbol or a string of symbols generated by the formal grammar of a formal language. The following are some important properties of a well-formed formula in PC:

- The smallest unit (or an atom) is considered to be a well-formed formula.
- If α is a well-formed formula, then $\sim\alpha$ is also a well-formed formula.
- If α and β are well-formed formulae, then $(\alpha \Lambda \beta)$, $(\alpha \text{ V } \beta)$, $(\alpha \rightarrow \beta)$, and $(\alpha \leftrightarrow \beta)$ are also well-formed formulae.

A propositional expression is called a well-formed formula if and only if it satisfies the above properties.

4.2.1 Truth Table

In PC, a *truth table* is used to provide operational definitions of important logical operators; it elaborates all possible truth values of a formula. The logical constants in PC are *true* and *false* and these are represented as T and F, respectively in a truth table. Let us assume that *A*, *B*, *C*, ... are propositioned symbols. The meanings of above-mentioned logical operators are given in a truth table (Table 4.1) as follows:

Table 4.1 Truth Table for Logical Operators

A	B	~A	A Λ B	A V B	A → B	A ↔ B
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

The truth values of well-formed formulae are calculated by using the truth table approach. Let us consider the following example.

Example 4.1 Compute the truth value of $\alpha : (A \vee B) \wedge (\sim B \rightarrow A)$ using truth table approach.

Solution Using the truth table approach, let us compute truth values of $(A \vee B)$ and $(\sim B \rightarrow A)$ and then compute for the final expression $(A \vee B) \wedge (\sim B \rightarrow A)$ (as given in Table 4.2).

Table 4.2 Truth Table for α

A	B	A ∨ B	~B	~B → A	α
T	T	T	F	T	**T**
T	F	T	T	T	**T**
F	T	T	F	T	**T**
F	F	F	T	F	**F**

Definition: Two formulae α and β are said to be logically equivalent ($\alpha \equiv \beta$) if and only if the truth values of both are the same for all possible assignments of logical constants (T or F) to the symbols appearing in the formulae.

4.2.2 Equivalence Laws

Equivalence relations (or laws) are used to reduce or simplify a given well-formed formula or to derive a new formula from the existing formula. Some of the important equivalence laws are given in Table 4.3. These laws can be verified using the truth table approach.

Table 4.3 Equivalence Laws

Name of Relation	Equivalence Relations
Commutative Law	$A \vee B \equiv B \vee A$ $A \wedge B \equiv B \wedge A$
Associative Law	$A \vee (B \vee C) \equiv (A \vee B) \vee C$ $A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$
Double Negation	$\sim(\sim A) \equiv A$
Distributive Laws	$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
De Morgan's Laws	$\sim(A \vee B) \equiv \sim A \wedge \sim B$ $\sim(A \wedge B) \equiv \sim A \vee \sim B$

(Contd.)

Table 4.3 (Contd.)

Name of Relation	Equivalence Relations
Absorption Laws	$A \vee (A \wedge B) \cong A$ $A \wedge (A \vee B) \cong A$ $A \vee (\sim A \wedge B) \cong A \vee B$ $A \wedge (\sim A \vee B) \cong A \wedge B$
Idempotence	$A \vee A \cong A$ $A \wedge A \cong A$
Excluded Middle Law	$A \vee \sim A \cong$ T (True)
Contradiction Law	$A \wedge \sim A \cong F$ (False)
Commonly used equivalence relations	$A \vee F \cong A$ $A \vee T \cong T$ $A \wedge T \cong A$ $A \wedge F \cong F$ $A \rightarrow B \cong \sim A \vee B$ $A \leftrightarrow B \cong (A \rightarrow B) \wedge (B \rightarrow A)$ $\cong (A \wedge B) \vee (\sim A \wedge \sim B)$

Let us verify the absorption law $A \vee (A \wedge B) \cong A$ using truth table approach as shown in Table 4.4.

Table 4.4 Verification of $A \vee (A \wedge B) \cong A$

A	B	A ∧ B	A ∨ A ∧ B
T	T	T	T
T	F	F	T
F	T	F	F
F	F	F	F

We can clearly see that the truth values of $A \vee (A \wedge B)$ and A are same; therefore, these expressions are equivalent.

4.3 Propositional Logic

Propositional logic (*or prop logic*) deals with the validity, satisfiability (also called consistency), and unsatisfiability (inconsistency) of a formula and the derivation of a new formula using equivalence laws. Each row of a truth table for a given formula α is called its *interpretation* under which the value of a formula may be either *true* or *false*. A formula α is said to be a *tautology* if and only if the value of α is true for all its interpretations. Now, the validity, satisfiability, and unsatisfiability of a formula may be determined on the basis of the following conditions:

- A formula α is said to be *valid* if and only if it is a *tautology*.
- A formula α is said to be *satisfiable* if there exists at least one interpretation for which α is true.
- A formula α is said to be *unsatisfiable* if the value of α is false under all interpretations.

Let us consider the following example to explain the concept of validity:

Example 4.2 Show that the following is a valid argument:
If it is humid then it will rain and since it is humid today it will rain

Solution Let us symbolize each part of the above English sentence by propositional atoms as follows:

A : It is humid
B : It will rain

Now, the formula (α) corresponding to the given sentence:

If it is humid then it will rain and since it is humid today it will rain

may be written as

$$\alpha : [(A \rightarrow B) \wedge A] \rightarrow B$$

Using the truth table approach (as given in Table 4.5), one can see that α is true under all interpretations and hence is a *valid argument*.

Table 4.5 Truth Table for $[(A \rightarrow B) \wedge A] \rightarrow B$

A	B	A → B = (X)	X ∧ A = (Y)	Y → B
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

The truth table approach is a simple and straightforward method and is extremely useful at presenting an overview of all the *truth* values in a given situation. Although it is an easy method for evaluating consistency, inconsistency, or validity of a formula, the limitation of this method lies in the fact that the size of truth table grows exponentially. That is, if a formula contains n atoms, then the truth table will contain 2^n entries. Moreover, it may be possible in some cases that all entries of a truth table are not required. In such situations, the construction of a truth table becomes a futile exercise.

For example, if we have to show that a formula $\alpha : (A \wedge B \wedge C \wedge D) \rightarrow (B \vee E)$ is *valid* using the truth table approach, then we need to construct a table containing 32 rows and compute the truth values of α for all 32 interpretations. In this example, we notice that the value of $(A \wedge B \wedge C \wedge D)$ is false for 30 out of the 32 entries and is true for 2 entries only. Since we know that $(X \rightarrow Y)$ is

true in all cases except the one in which $X = T$ and $Y = F$, it is clear that α is true for these 30 cases where $(A \wedge B \wedge C \wedge D)$ is *false*. Hence, we are left to verify whether α is true or not for 2 entries only by checking an expression on the right side of $\rightarrow$. If this expression is true then α is valid, otherwise it is not valid.

Use of the truth table approach in such situations proves to be a wastage of time. Therefore, we require some other methods which can help in proving the validity of the formula directly. Some other methods that are concerned with proofs and deductions are as follows:

- Natural deduction system
- Axiomatic system
- Semantic tableau method
- Resolution refutation method

All these methods have been discussed in the following sections.

4.4 Natural Deduction System

Natural deduction system (NDS) is thus called because of the fact that it mimics the pattern of natural reasoning. This system is based on a set of deductive inference rules. Assuming that $A_1, \ldots, A_k$, where $1 \leq k \leq n$, are a set of atoms and α_j, where $1 \leq j \leq m$, and β are well-formed formulae, the inference rules may be stated as shown in the following NDS rules table (Table 4.6).

Table 4.6 NDS Rules Table

Rule Name	Symbol	Rule	Description
Introducing $\wedge$	$(I{:}\wedge)$	If $A_1, \ldots, A_n$ then $A_1 \wedge \ldots \wedge A_n$	If $A_1, \ldots, A_n$ are true, then their conjunction $A_1 \wedge \ldots \wedge A_n$ is also *true*.
Eliminating $\wedge$	$(E{:}\wedge)$	If $A_1 \wedge \ldots \wedge A_n$ then A_i $(1 \leq i \leq n)$	If $A_1 \wedge \ldots \wedge A_n$ is *true*, then any A_i is also *true*.
Introducing V	$(I{:}\mathrm{V})$	If any A_i $(1 \leq i \leq n)$ then $A_1 \mathrm{V} \ldots \mathrm{V} A_n$	If any A_i $(1 \leq i \leq n)$ is *true*, then $A_1 \mathrm{V} \ldots \mathrm{V} A_n$ is also *true*.
Eliminating V	$(E{:}\mathrm{V})$	If $A_1 \mathrm{V} \ldots \mathrm{V} A_n, A_1 \rightarrow A, \ldots, A_n \rightarrow A$ then A	If $A_1 \mathrm{V} \ldots \mathrm{V} A_n, A_1 \rightarrow A, A_2 \rightarrow A, \ldots$, and $A_n \rightarrow A$ are *true*, then A is *true*.
Introducing $\rightarrow$	$(I : \rightarrow)$	If from $\alpha_1, \ldots, \alpha_n$ infer β is proved then $\alpha_1 \wedge \ldots \wedge \alpha_n \rightarrow \beta$ is proved	If given that $\alpha_1, \alpha_2, \ldots$, and α_n are *true* and from these we deduce β then $\alpha_1 \wedge \ldots \wedge \alpha_n \rightarrow \beta$ is also *true*.

(Contd.)

Table 4.6 (Contd.)

Rule Name	Symbol	Rule	Description
Eliminating $\rightarrow$	(*E*: $\rightarrow$)	If $A_1 \rightarrow A$, A_1, then A	If $A_1 \rightarrow A$ and A_1 are *true* then A is also *true*. This is called *Modus Ponen* rule.
Introducing $\leftrightarrow$	(*I*: $\leftrightarrow$)	If $A_1 \rightarrow A_2$, $A_2 \rightarrow A_1$ then $A_1 \leftrightarrow A_2$	If $A_1 \rightarrow A_2$ and $A_2 \rightarrow A_1$ are *true* then $A_1 \leftrightarrow A_2$ is also *true*.
Elimination $\leftrightarrow$	(*E*: $\leftrightarrow$)	If $A_1 \leftrightarrow A_2$ then $A_1 \rightarrow A_2$, $A_2 \rightarrow A_1$	If $A_1 \leftrightarrow A_2$ is *true* then $A_1 \rightarrow A_2$ and $A_2 \rightarrow A_1$ are *true*
Introducing ~	(*I*: ~)	*If from A infer* $A_1 \wedge \sim A_1$ is proved then ~*A* is proved	If from *A* (which is *true*), a contradiction is proved then truth of ~*A* is also proved
Eliminating ~	(*E*: ~)	*If from* ~*A infer* $A_1 \wedge \sim A_1$ is proved then *A* is proved	If from ~*A*, a contradiction is proved then truth of *A* is also proved

A theorem in the NDS written as *from* $\alpha_1, \ldots, \alpha_n$ *infer* β leads to the interpretation that β is deduced from a set of hypotheses $\{\alpha_1, \ldots, \alpha_n\}$. All hypotheses are assumed to be true in a given context and therefore the theorem β is also true in the same context. Thus, we can conclude that β is consistent. A theorem that is written as *infer* β implies that there are no hypotheses and β is true under all interpretations, i.e., β is a *tautology* or *valid*. Let us consider the following example and show the proof using Natural deduction systems. The conventions used in such a proof are as follows:

- The 'Description' column consists of rules applied on a subexpression in the proof line.
- The second column consists the subexpression obtained after applying an appropriate rule.
- The final column consists the line number of subexpressions in the proof.

Example 4.3 Prove that $A \wedge (B \vee C)$ is deduced from $A \wedge B$.

Solution The theorem in NDS can be written as *from* $A \wedge B$ *infer* $A \wedge (B \vee C)$ in NDS. We can prove the theorem (Table 4.7) as follows:

Table 4.7 Proof of the Theorem for Example 4.3

Description	Formula	Comments
Theorem	*from* $A \wedge B$ *infer* $A \wedge (B \vee C)$	*To be proved*
Hypothesis (given)	$A \wedge B$	1
E: $\wedge$ (1)	A	2
E: $\wedge$ (1)	B	3
I: $\vee$ (3)	$B \vee C$	4
I: $\wedge$ (2, 4)	$A \wedge (B \vee C)$	Proved

If we assume that $\alpha \rightarrow \beta$ is *true*, then we can conclude that β is also *true* if α is *true*. It can be represented in the form of a theorem of NDS as *from* α *infer* β and if we can prove the theorem then we can conclude the truth of $\alpha \rightarrow \beta$. The converse of this is also true. Let us state formally the deduction theorem in NDS.

Deduction Theorem To prove a formula $\alpha_1 \wedge \ldots \wedge \alpha_n \rightarrow \beta$, it is sufficient to prove a theorem *from* $\alpha_1, \ldots, \alpha_n$ *infer* β. Conversely, if $\alpha_1 \wedge \ldots \wedge \alpha_n \rightarrow \beta$, is proved then the theorem *from* $\alpha_1, \ldots, \alpha_n$ *infer* β is assumed to be proved.

Let us consider the following example to show the use of deduction theorem.

Example 4.4 Prove the theorem *infer* $[(A \rightarrow B) \wedge (B \rightarrow C)] \rightarrow (A \rightarrow C)$.

Solution The theorem *infer* $[(A \rightarrow B) \wedge (B \rightarrow C)] \rightarrow (A \rightarrow C)$ is reduced to the theorem *from* $(A \rightarrow B), (B \rightarrow C)$ *infer* $(A \rightarrow C)$ using deduction theorem. Further, to prove '$A \rightarrow C$', we will have to prove a sub-theorem *from A infer C*. The proof of the theorem is shown in Table 4.8.

Table 4.8 Proof of the Theorem *from* $(A \rightarrow B), (B \rightarrow C)$ *infer* $(A \rightarrow C)$

Description	Formula	Comments	
Theorem	*from* $A \rightarrow B, B \rightarrow C$ *infer* $A \rightarrow C$	To be proved	
Hypothesis 1	$A \rightarrow B$	1	
Hypothesis 2	$B \rightarrow C$	2	
Sub-theorem	*from A infer C*	3	
Hypothesis	A		3.1
E: $\rightarrow$ (1, 3.1)	B		3.2
E: $\rightarrow$ (2, 3.2)	C		3.3
I: $\rightarrow$ (3)	$A \rightarrow C$	Proved	

4.5 Axiomatic System

The *axiomatic system* is based on a set of three axioms and one rule of deduction. Although minimal in structure, it is as powerful as the truth table and NDS approaches. In axiomatic system, the proofs of the theorems are often difficult and require a guess in selection of appropriate axiom(s). In this system, only two logical operators *not* (~) and *implies* ($\rightarrow$) are allowed to form a formula. It should be noted that other logical operators, such as $\wedge$; V, and $\leftrightarrow$, can be easily expressed in terms of ~ and $\rightarrow$ using equivalence laws stated earlier. For example,

$$A \wedge B \equiv \sim(\sim A \vee \sim B) \equiv \sim(A \rightarrow \sim B)$$
$$A \vee B \equiv \sim A \rightarrow B$$
$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A) \equiv \sim[(A \rightarrow B) \rightarrow \sim(B \rightarrow A)]$$

In axiomatic system, there are three axioms, which are always true (or valid), and one rule called *modus ponen* (MP). Here, α, β, and γ are well-formed formulae of the axiomatic system. The three axioms and the rule are stated as follows:

Axiom 1 $\alpha \rightarrow (\beta \rightarrow \alpha)$

Axiom 2 $[\alpha \rightarrow (\beta \rightarrow \gamma)] \rightarrow [(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)]$

Axiom 3 $(\sim\alpha \rightarrow \sim\beta) \rightarrow (\beta \rightarrow \alpha)$

Modus Ponen Rule *Hypotheses:* $\alpha \rightarrow \beta$ and α; *Consequent:* β

Interpretation of Modus Ponen Rule: Given that $\alpha \rightarrow \beta$ and α are hypotheses (assumed to be *true*), β is inferred (i.e., *true*) as a consequent.

Let $\Sigma = \{\alpha_1, ..., \alpha_n\}$ be a set of hypotheses. The formula α is defined to be a *deductive consequence* of Σ if either α is an *axiom* or a *hypothesis* or is *derived* from α_j, where $1 \leq j \leq n$, using modus ponen inference rule. It is represented as $\{\alpha_1, ..., \alpha_n\} \vdash \alpha$ or more formally as $\Sigma \vdash \alpha$. If Σ is an empty set and α is deduced, then we can write $\vdash \alpha$. In this case, α is deduced from axioms only and no hypotheses are used. In such situations, α is said to be a *theorem*. To illustrate the concepts clearly let us consider the following example:

Example 4.5 Establish that $A \rightarrow C$ is a deductive consequence of $\{A \rightarrow B, B \rightarrow C\}$, i.e., $\{A \rightarrow B, B \rightarrow C\} \vdash (A \rightarrow C)$.

Solution We can prove the theorem as shown below in Table 4.9.

Table 4.9 Proof of the theorem $\{A \rightarrow B, B \rightarrow C\} \vdash (A \rightarrow C)$

Description	Formula	Comments
Theorem	$\{A \rightarrow B, B \rightarrow C\} \vdash (A \rightarrow C)$	*Prove*
Hypothesis 1	$A \rightarrow B$	1
Hypothesis 2	$B \rightarrow C$	2
Instance of Axiom 1	$(B \rightarrow C) \rightarrow [A \rightarrow (B \rightarrow C)]$	3
MP (2, 3)	$[A \rightarrow (B \rightarrow C)]$	4
Instance of Axiom 2	$[A \rightarrow (B \rightarrow C)] \rightarrow [(A \rightarrow B) \rightarrow (A \rightarrow C)]$	5
MP (4, 5)	$(A \rightarrow B) \rightarrow (A \rightarrow C)$	6
MP (1, 6)	$(A \rightarrow C)$	*Proved*

Hence, we can conclude that $A \rightarrow C$ is a deductive consequence of $\{A \rightarrow B, B \rightarrow C\}$.

Deduction Theorem Given that Σ is a set of hypotheses and α and β are well-formed formulae. If β is proved from $\{\Sigma \cup \alpha\}$, then according to the deduction theorem, $(\alpha \rightarrow \beta)$ is proved from Σ. Alternatively, we can write $\{\Sigma \cup \alpha\} \vdash \beta$ implies $\Sigma \vdash (\alpha \rightarrow \beta)$.

Converse of Deduction Theorem The converse of the deduction theorem can be stated as: Given $\Sigma \vdash (\alpha \rightarrow \beta)$, then $\{\Sigma \cup \alpha\} \vdash \beta$ is proved.

Useful Tips

The following are some tips that will prove to be helpful in dealing with an axiomatic system:

- If α is given, then we can easily prove $\beta \rightarrow \alpha$ for any well-formed formulae α and β.
- If $\alpha \rightarrow \beta$ is to be proved, then include α in the set of hypotheses Σ and derive β from the set $\{\Sigma \cup \alpha\}$. Then, by using deduction theorem, we can conclude that $\alpha \rightarrow \beta$.

Example 4.6 Prove $\vdash \sim A \rightarrow (A \rightarrow B)$ by using deduction theorem.

Solution If we can prove $\{\sim A\} \vdash (A \rightarrow B)$ then using deduction theorem, we have proved $\vdash \sim A \rightarrow (A \rightarrow B)$. The proof is shown in Table 4.10.

Table 4.10 Proof of $\{\sim A\} \vdash (A \rightarrow B)$

Description	Formula	Comments
Theorem	$\{\sim A\} \vdash (A \rightarrow B)$	*Prove*
Hypothesis 1	$\sim A$	1
Instance of Axiom 1	$\sim A \rightarrow (\sim B \rightarrow \sim A)$	2
MP (1, 2)	$(\sim B \rightarrow \sim A)$	3
Instance of Axiom 3	$(\sim B \rightarrow \sim A) \rightarrow (A \rightarrow B)$	4
MP (3, 4)	$(A \rightarrow B)$	*Proved*

4.6 Semantic Tableau System in Propositional Logic

In both natural deduction and axiomatic systems, forward chaining approach is used for constructing proofs and derivations. In this approach, we start proofs or derivations from a given set of hypotheses or axioms. In axiomatic system, we often require a guess for the selection of appropriate axiom(s) in order to prove a theorem. Although the forward chaining approach is good for theoretical purposes, its implementation in derivations and proofs is difficult. Two other approaches may be used: *semantic tableau* and *resolution refutation* methods; in both cases, proofs follow backward chaining approach. In semantic tableau method, a set of rules are applied systematically on a formula or a set of formulae in order to establish consistency or inconsistency.

Semantic tableau is a binary tree which is constructed by using semantic tableau rules with a formula as a root. These rules and building proofs using this method are discussed in detail in the following subsection.

4.6.1 Semantic Tableau Rules

The semantic tableau rules are given in Table 4.11 where α and β are two formulae.

Table 4.11 Semantic Tableau Rules for α and β

Rule No.	Tableau tree	Explanation
Rule 1	$\alpha \wedge \beta$ *is true if both* α *and* β *are true* $\alpha \wedge \beta$ | α | β	A tableau for a formula $(\alpha \wedge \beta)$ is constructed by adding both α and β to the same path (branch)
Rule 2	$\sim(\alpha \wedge \beta)$ *is true if either* $\sim\alpha$ *or* $\sim\beta$ *is true* $\sim(\alpha \wedge \beta)$ $\sim\alpha$ $\quad$ $\sim\beta$	A tableau for a formula $\sim(\alpha \wedge \beta)$ is constructed by adding two new paths: one containing $\sim\alpha$ and the other containing $\sim\beta$
Rule 3	$\alpha \vee \beta$ *is true if either* α *or* β *is true* $\alpha \vee \beta$ α $\quad$ β	A tableau for a formula $(\alpha \vee \beta)$ is constructed by adding two new paths: one containing α and the other containing β
Rule 4	$\sim(\alpha \vee \beta)$ *is true if both* $\sim\alpha$ *and* $\sim\beta$ *are true* $\sim(\alpha \vee \beta)$ | $\sim\alpha$ | $\sim\beta$	A tableau for a formula $\sim(\alpha \vee \beta)$ is constructed by adding both $\sim\alpha$ and $\sim\beta$ to the same path
Rule 5	$\sim(\sim\alpha)$ *is true then* α *is true* $\sim(\sim\alpha)$ | α	A tableau for $\sim(\sim\alpha)$ is constructed by adding α on the same path
Rule 6	$\alpha \rightarrow \beta$ *is true then* $\sim\alpha \vee \beta$ *is true* $\alpha \rightarrow \beta$ $\sim\alpha$ $\quad$ β	A tableau for a formula $\alpha \rightarrow \beta$ is constructed by adding two new paths: one containing $\sim\alpha$ and the other containing β
Rule 7	$\sim(\alpha \rightarrow \beta)$ *true then* $\alpha \wedge \sim\beta$ *is true* $\sim(\alpha \rightarrow \beta)$ | α | $\sim\beta$	A tableau for a formula $\sim(\alpha \rightarrow \beta)$ is constructed by adding both α and $\sim\beta$ to the same path

(Contd.)

Table 4.11 (Contd.)

Rule No.	Tableau tree	Explanation
Rule 8	$\alpha \leftrightarrow \beta$ *is true then* $(\alpha \wedge \beta) \vee (\sim \alpha \wedge \sim \beta)$ *is true* $\alpha \leftrightarrow \beta$ $\alpha \wedge \beta$ $\quad$ $\sim\alpha \wedge \sim\beta$	A tableau for a formula $\alpha \leftrightarrow \beta$ is constructed by adding two new paths: one containing $\alpha \wedge \beta$ and other $\sim\alpha \wedge \sim\beta$ which are further expanded
Rule 9	$\sim(\alpha \leftrightarrow \beta)$ is *true* then $(\alpha \wedge \sim\beta) \vee (\sim\alpha \wedge \beta)$ *is true* $\sim(\alpha \leftrightarrow \beta)$ $\alpha \wedge \sim\beta$ $\quad$ $\sim\alpha \wedge \beta$	A tableau for a formula $\sim(\alpha \leftrightarrow \beta)$ is constructed by adding two new paths: one containing $\alpha \wedge \sim\beta$ and the other $\sim\alpha \wedge \beta$ which are further expanded

Let us consider an example to illustrate the method of constructing semantic tableau for a formula. The convention used in this construction is self-explanatory. The first column consists of rule number applied on line number. The second column contains the derivation of semantic tableau and last column contains the line number.

Example 4.7 Construct a semantic tableau for a formula $(A \wedge \sim B) \wedge (\sim B \rightarrow C)$.

Solution The construction of the semantic tableau for the given formula $(A \wedge \sim B) \wedge (\sim B \rightarrow C)$ is shown in Table 4.12.

Table 4.12 Semantic Tableau for Example 4.7

Description	Formula	Line number
Tableau root	$(A \wedge \sim B) \wedge (\sim B \rightarrow C)$	1
Rule 1 (1)	$A \wedge \sim B$	2
	$\sim B \rightarrow C$	3
Rule 1 (2)	A	4
	$\sim B$	5
Rule 6 (3)	$\sim(\sim B)$ $\quad$ C	6
Rule 3 (6)	B $\quad$ $\surd$(open)	
	× (closed) $\{B, \sim B\}$	

Paths in a tableau tree extend from the root to the leaf nodes. There are two paths in the tree as shown in Table 4.12 starting from the root to leaf nodes ending at B and C. It is observed that the first path from root to B becomes closed because of the presence of complementary atoms B and $\sim B$, while the other path remains open.

The thumb rule to construct a semantic tableau is to apply non-branching rules (such as rules 1, 4, and 7) before branching rules.

4.6.2 Satisfiability and Unsatisfiability

Before we proceed any further, it is important to become familiar with certain terms that are used in the study of a tableau. For this, consider α to be any formula (Kaushik S 2002).

- A path is said to be *contradictory* or *closed* (finished) whenever complementary atoms appear on the same path of a semantic tableau. This denotes inconsistency.
- If all paths of a tableau for a given formula α are found to be closed, it is called a *contradictory tableau*. This indicates that there is no interpretation or model that satisfies α.
- A formula α is said to be *satisfiable* if a tableau with root α is not a contradictory tableau, that is, it has at least one open path. We can obtain a model or an interpretation under which the formula α is evaluated to be true by assigning T (true) to all atomic formulae appearing on the open path of semantic tableau of α.
- A formula α is said to be *unsatisfiable* if a tableau with root α is a contradictory tableau.
- If we obtain a contradictory tableau with root $\sim\alpha$, we say that the formula α is *tableau provable*. Alternatively, a formula α is said to be *tableau provable* (denoted by $\vdash \alpha$) if a tableau with root $\sim\alpha$ is a contradictory tableau.
- A set of formulae $S = \{\alpha_1, \alpha_2, ..., \alpha_n\}$ is said to *unsatisfiable* if a tableau with root $(\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n)$ is a contradictory tableau.
- A set of formulae $S = \{\alpha_1, \alpha_2, ..., \alpha_n\}$ is said to be *satisfiable* if the formulae in a set are simultaneously true, that is, if a tableau for $\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n$ has at least one open (or non-contradictory) path.
- Let S be a set of formulae. The formula α is said to be tableau provable from S (denoted by $S \vdash \alpha$) if there is a contradictory tableau from S with $\sim\alpha$ as a root.
- A formula α is said to be a *logical consequence* of a set S if and only if α is tableau provable from S.
- If α is tableau provable ($\vdash \alpha$) then it is also valid ($\models \alpha$) and vice versa.

Now we consider a few examples to illustrate the terminology discussed above.

Example 4.8 Show that a formula $\alpha : (A \wedge \sim B) \wedge (\sim B \rightarrow C)$ is satisfiable.

Solution The semantic tableau for $(A \wedge \sim B) \wedge (\sim B \rightarrow C)$ has been drawn in Table 4.12 and we observe that there are two paths in it of which one path is closed, while the other is open. This shows that the formula $(A \wedge \sim B) \wedge (\sim B \rightarrow C)$ is satisfiable. In order to find its model (that is, the interpretation under which the formula is true), we assign T (true) to all atomic formulae appearing on the open path. Therefore, $\{A = T; \sim B = T; C = T\}$ or alternatively $\{A = T; B = F; C = T\}$ is a model under which α is true. We can verify this using truth table approach also.

Example 4.9 Show that $\alpha : (A \wedge B) \wedge (B \rightarrow \sim A)$ is unsatisfiable using the tableau method.

Solution It can be proven that α: $(A \wedge B) \wedge (B \rightarrow \sim A)$ is unsatisfiable as shown in Table 4.13.

Table 4.13 Tableau Method for Example 4.9

Description	Formula	Line number
Tableau root	$(A \wedge B) \wedge (B \rightarrow \sim A)$	1
Rule 1 (1)	$A \wedge B$	2
	$B \rightarrow \sim A$	3
Rule 1 (2)	A	4
	B	5
Rule 6 (3)	$\sim B$ | $\times \{B, \sim B\}$; $\sim A$ | $\times \{A, \sim A\}$	

Example 4.10 Consider a set $S = \{\sim(A \vee B), (C \rightarrow B), (A \vee C)\}$ of formulae. Show that S is unsatisfiable.

Solution Consider the conjunction of formulae in the set as a root of semantic tableau. We see from Table 4.14 that such a tableau is contradictory; hence, S is unsatisfiable.

Table 4.14 Tableau Method For Example 4.10

Description	Formula	Line number
Tableau root	$\sim(A \vee B) \wedge (C \rightarrow B) \wedge (A \vee C)$	1
Rule 1 (1)	$\sim(A \vee B)$	2
	$(C \rightarrow B)$	3
	$(A \vee C)$	4
Rule 4 (2)	$\sim A$	
	$\sim B$	
Rule 3 (4)	A $\quad$ C	
	$\times \{A, \sim A\}$	
Rule 6 (3)	$\sim C$ $\quad$ B	
	$\times \{C, \sim C\}$ $\quad$ $\times \{B, \sim B\}$	

Example 4.11 Show that a set $S = \{\sim(A \vee B), (B \rightarrow C), (A \vee C)\}$ is consistent.

Solution The set S can be shown to be consistent (Table 4.15) as follows:

Table 4.15 Tableau Method for Example 4.11

Description	Formula	Line number
Tableau root	$\sim(A \vee B) \wedge (B \rightarrow C) \wedge (A \vee C)$	1
Rule 1 (1)	$\sim(A \vee B)$	2
	$(B \rightarrow C)$	3
	$(A \vee C)$	4
Rule 4 (2)	$\sim A$	
	$\sim B$	
Rule 3 (4)	A $\quad$ C	
	$\times \{A, \sim A\}$	
Rule 6 (3)	$\sim B$ $\quad$ C	
	$\surd$ $\quad$ $\surd$	

Since the tableau of conjunction of formulae of S has open paths, S is satisfiable. Further, we can construct a model for S by assigning truth value T to each literal on open path, that is, $\{\sim A = T; \sim B = T; C = T\}$ or $\{A = F; B = F; C = T\}$.

Example 4.12 Show that B is a logical consequence of $S = \{A \rightarrow B, A\}$.

Solution Let us include $\sim B$ as a root with S in the tableau tree.

Table 4.16 Tableau Method for Example 4.12

Description	Formula	Line number
Tableau root	$\sim B$	1
Premise 1	$A \rightarrow B$	2
Premise 2	A	
Rule 6 (2)	$\sim A$ | B	3
	$\times \{A, \sim A\}$ $\times \{B, \sim B\}$	

We see from Table 4.16 that B is tableau provable from S, that is, $\sim B$ as root gives contradictory tableau; thus B is a logical consequence of S.

Example 4.13 Show that $a : B \vee \sim(A \rightarrow B) \vee \sim A$ is valid.

Solution In order to show that α is valid, we have to show that α is tableau provable, that is, the tableau tree with $\sim\alpha$ is contradictory. Table 4.17 shows that α is a valid formula.

4.7 Resolution Refutation in Propositional Logic

Another simple method that can be used in propositional logic to prove a formula or derive a goal from a given set of clauses by contradiction is the *resolution refutation method*. The term *clause* is used to denote a special formula containing the boolean operators ~ and V. Any given formula can be easily converted into a set of clauses. The method to do this is explained later in this section. Resolution refutation is the most favoured method for developing computer-based systems that can be used to prove theorems automatically. It uses a single inference rule, which is known as *resolution based on modus ponen inference rule*. It is more efficient in comparison to NDS and Axiomatic system because in this case we do not need to guess which rule or axiom to apply in development of proofs. Here, the negation of the goal to be proved is added to the given set of clauses, and using the resolution principle, it is shown that there is a refutation in the new

set. During resolution, we need to identify two clauses: one with a positive atom (P) and the other with a negative atom ($\sim P$) for the application of resolution rule. Before discussing resolution of clauses, let us desribe a method for converting a formula into a set of clauses.

Table 4.17 Tableau Method for Example 4.13

Description	Formula	Line number
Tableau root	$\sim(B \vee \sim(A \rightarrow B) \vee \sim A)$	1
	|	
Rule 4 (1)	$\sim B$	2
	|	
	$\sim[\sim(A \rightarrow B) \vee \sim A]$	3
Rule 4 (3)	$\sim[\sim(A \rightarrow B)]$	4
	|	
	$\sim(\sim A)$	5
	|	
Rule 5 (5)	A	6
	|	
Rule 5 (4)	$A \rightarrow B$	
Rule 6 (7)	$\sim A$ | $\times\{A, \sim A\}$; B | $\times\{B, \sim B\}$	7

4.7.1 Conversion of a Formula into a Set of Clauses

In propositional logic, there are two normal forms, namely, *disjunctive normal form* (DNF) and *conjunctive normal form* (CNF). A formula is said to be in its *normal form* if it is constructed using only natural connectives $\{\sim, \wedge, \vee\}$. In DNF, the formula is represented as disjunction of conjunction, that is, in the form $(L_{11} \wedge \ldots \wedge L_{1m}) \vee \ldots \vee (L_{p1} \wedge \ldots \wedge L_{pk})$, whereas in CNF, it is represented as conjunction of disjunction, that is, in the form $(L_{11} \vee \ldots \vee L_{1m}) \wedge \ldots \wedge (L_{p1} \vee \ldots \vee L_{pk})$, where all L_{ij} are literals (positive or negative atoms). We can easily write the CNF form of a given formula as $(C_1 \wedge \ldots \wedge C_n)$, where each $C_k (1 \leq k \leq n)$ is a disjunction of literals and is called a *clause*. Formally, a *clause* is defined as a formula of the form $(L_1 \vee \ldots \vee L_m)$. Therefore, if a

given formula is converted to its equivalent CNF as $(C_1 \wedge \ldots \wedge C_n)$, then the set of clauses is nothing but a set of each conjunct of CNF, that is, $\{C_1, \ldots, C_n\}$. For example, the set $\{A \vee B, \sim A \vee D, C \vee \sim B\}$ represents a set of clauses $A \vee B$, $\sim A \vee D$, and $C \vee \sim B$. The procedure by which such clauses for a given formula can be obtained is discussed in the following subsection.

4.7.2 Conversion of a Formula to its CNF

Any formula in propositional logic can be easily transformed into its equivalent CNF representation by using the equivalence laws described below. The following steps are taken to transform a formula to its equivalent CNF.

- Eliminate double negation signs by using
$$\sim(\sim A) \equiv A$$
- Use De Morgan's Laws to push ~ (negation) immediately before the atomic formula
$$\sim(A \wedge B) \equiv \sim A \vee \sim B$$
$$\sim(A \vee B) \equiv \sim A \wedge \sim B$$
- Use distributive law to get CNF
$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$
- Eliminate $\rightarrow$ and $\leftrightarrow$ by using the following equivalence laws:
$$A \rightarrow B \equiv \sim A \vee B$$
$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$$

The method of conversion will become clearer with the help of the following examples:

Example 4.14 Convert the formula $(\sim A \rightarrow B) \wedge (C \wedge \sim A)$ into its equivalent CNF representation.

Solution The given formula $(\sim A \rightarrow B) \wedge (C \wedge \sim A)$ can be transformed into its CNF representation in the following manner:

$$(\sim A \rightarrow B) \wedge (C \wedge \sim A) \equiv (\sim(\sim A) \vee B) \wedge (C \wedge \sim A) \quad \{\text{as } \sim A \rightarrow B \equiv (\sim A) \vee B\}$$
$$\equiv (A \vee B) \wedge (C \wedge \sim A) \quad \{\text{as } \sim(\sim A) \equiv A\}$$
$$\equiv (A \vee B) \wedge (C \wedge \sim A)$$

The set of clauses in this case is written as $\{(A \vee B), C, \sim A\}$

4.7.3 Resolution of Clauses

Two clauses can be resolved by eliminating complementary pair of literals, if any, from both; a new clause is constructed by disjunction of the remaining literals in both the clauses. Therefore, if two clauses C_1 and C_2 contain a complementary pair of literals $\{L, \sim L\}$, then these clauses may be resolved together by deleting L from C_1 and $\sim L$ from C_2 and constructing a new clause by the disjunction of the remaining literals in C_1 and C_2. This new clause is called *resolvent* of C_1 and C_2. The clauses C_1 and C_2 are called *parent clauses* of the resolved clause. The resolution tree is an inverted binary tree with the last node being a resolvent, which is generated as a part of the resolution process. The process of resolution of clauses will become clearer with the help of the following examples:

Example 4.15 Find resolvent of the clauses in the set $\{A \vee B, \sim A \vee D, C \vee \sim B\}$.

Solution The method of resolution is shown in Fig. 4.1.

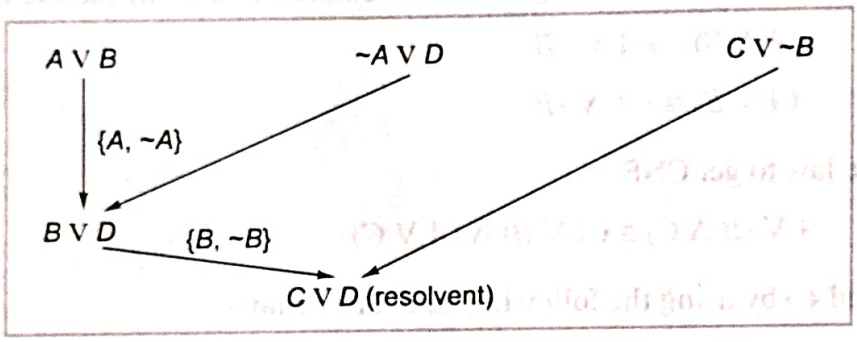


Figure 4.1 Resolution of clauses of Example 4.15

We can clearly see that $C \vee D$ is a resolvent of the set $\{A \vee B, \sim A \vee D, C \vee \sim B\}$.

Now that we are familiar with the resolution process, we can state a few results.

- If C is a resolvent of two clauses C_1 and C_2, then C is called a logical consequence of the set of the clauses $\{C_1, C_2\}$. This is known as *resolution principle.*
- If a contradiction (or an empty clause) is derived from a set S of clauses using resolution then S is said to be *unsatisfiable.* Derivation of contradiction for a set S by resolution method is called a *resolution refutation* of S.
- A clause C is said to be a *logical consequence* of S if C is derived from S.
- Alternatively, using the resolution refutation concept, a clause C is defined to be a *logical consequence* of S if and only if the set $S' = S \cup \{\sim C\}$ is unsatisfiable, that is, a contradiction (or an empty clause) is deduced from the set S', assuming that initially the set S is satisfiable.

Example 4.16 Using resolution refutation principle show that $C \vee D$ is a logical consequence of $S = \{A \vee B, \sim A \vee D, C \vee \sim B\}$.

Solution To prove the statement, first we will add negation of the logical consequence, that is, $\sim(C \vee D) \equiv \sim C \wedge \sim D$ to the set S to get $S' = \{A \vee B, \sim A \vee D, C \vee \sim B, \sim C, \sim D\}$. Now, we can show that S' is unsatisfiable by deriving contradiction using the resolution principle (Fig. 4.2).

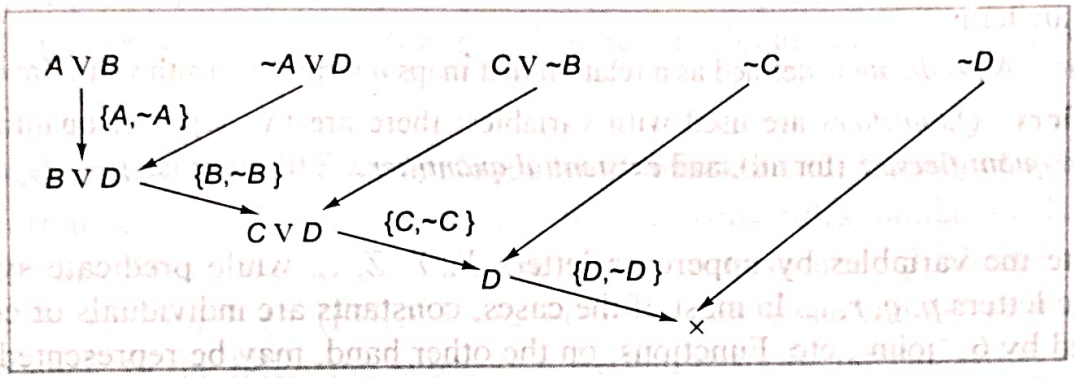


Figure 4.2 Resolution of Clauses for Example 4.16

Since we get contradiction from S', we can conclude that $(C \vee D)$ is a logical consequence of $S = \{A \vee B, \sim A \vee D, C \vee \sim B\}$.

4.8 Predicate Logic

In the preceding sections, we have discussed about propositional logic and various methods that can be used to show validity, unsatisfiability, etc., of a given proposition or a set of propositions. However, propositional logic has many limitations. For example, the facts *John is a boy*, *Paul is a boy*, and *Peter is a boy* can be symbolized by *A*, *B*, and C, respectively, in propositional logic but we can not draw any conclusions about the similarities between *A*, *B*, and *C*, that is, we cannot conclude that these symbols represent boys. Alternatively, if we represent these facts as boy(John), boy(Paul), and boy(Peter), then these statements give prima facie information that *John*, *Paul*, and *Peter* are all boys. These facts can be easily generated from a general statement boy(*X*), where the variable *X* is bound with *John*, *Paul*, or *Peter*. These facts are called instances of boy(*X*), while the statement boy(*X*) is called a *predicate statement or expression*. Here, *boy* is a predicate symbol and *X* is its argument. When a variable *X* gets bound to its actual value, then the predicate statement boy(*X*) becomes either *true* or *false*, for example, boy(Peter) = *true*, boy(Mary) = *false*, and so on.

Further, statements like *All birds fly* cannot be represented in propositional logic. Such limitations are removed in predicate logic. The predicate logic is a logical extension of propositional logic, which deals with the validity, satisfiability, and unsatisfiability (inconsistency) of a formula along with the inference rules for derivation of a new formula. Predicate calculus is the study of predicate systems; when inference rules are added to predicate calculus, it becomes predicate logic.

4.8.1 Predicate Calculus

Predicate calculus has three more logical notions in addition to propositional calculus. These are described as follows:

- **Term** A *term* is defined as either a variable, or constant or n-place function. A *function* is defined as a mapping that maps n terms to a single term. An n-place function is written as $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms.
- **Predicate** A *predicate* is defined as a relation that maps n terms to a truth value {*true*, *false*}.
- **Quantifiers** *Quantifiers* are used with variables; there are two types of quantifiers, namely, *universal quantifiers*, $\forall$ (for all), and *existential quantifiers*, $\exists$ (there exists)

We can denote the variables by uppercase letters X, Y, Z, ... while predicate symbols can be denoted by the letters p, q, r, ... In most of the cases, constants are individuals or concepts which can be denoted by 6, 'john', etc. Functions, on the other hand, may be represented by lowercase letters such as f, g, h, etc.

Well-formed formula In predicate calculus, well-formed formula (or simply formula) is defined as follows:

- Atomic formula $p\ (t_1, \ldots, t_n)$ (also called an atom) is a well-formed formula, where p is a predicate symbol and $t_1, \ldots, t_n$ are the terms.
- If α and β are well-formed formulae, then $\sim(\alpha)$, $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$, $(\alpha \rightarrow \beta)$, and $(\alpha \leftrightarrow \beta)$ are well-formed formulae.
- If α is a well-formed formula and X is a free variable in α, then $(\forall X)\alpha$ and $(\exists X)\alpha$ are both well-formed formulae. Here, α is in scope of quantifier $\forall$ or $\exists$. Scope of the variable X is defined as that part of an expression where all occurrences of X have the same value.
- Well-formed formulae may be generated by applying the rules described above a finite number of times.

The following are some examples of atomic formulae:

- A statement *X is brother of Y* can be easily represented in predicate calculus as brother(X, Y) which maps it to *true* or *false* when X and Y get instantiated to actual values. Here, brother is a predicate name.
- A statement *Peter loves his son* is represented as love("Peter", son("Peter")). Here, *son* is a function that maps Peter to his son and *love* is a predicate name which takes two terms and maps them to *true* or *false* depending on the values of its terms.
- A statement *Every human is mortal* is translated into predicate calculus formula as follows: Let us represent the statement *X is a human* as human(X) and *X is mortal* as mortal(X). The corresponding formula can then be written as $(\forall X)$ (human(X) $\rightarrow$ mortal(X))

4.8.2 First-Order Predicate Calculus

If the quantification in predicate formula is only on simple variables and not on predicates or functions then it is called *first-order predicate calculus*. On the other hand, if the quantification is over first-order predicates and functions, then it becomes *second-order predicate calculus*. For example, $\forall p\ (p(X)) \leftrightarrow p(Y))$ is a second-order predicate statement, whereas $\forall X\ \forall Y\ (p(X) \leftrightarrow p(Y))$ is a first-order predicate statement. Similarly, higher order predicate calculus allows quantification over higher types. Since we will not be dealing with higher order predicates presently, we need not go into the details.

The first-order predicate calculus is a formal language in which a wide variety of statements are expressed. The formulae in predicate calculus are formed using rules similar to those used in propositional calculus. When inference rules are added to first-order predicate calculus, it becomes *first-order predicate logic* (FOL). Using inference rules, one can derive new formulae from the existing ones. In the following subsection we will describe interpretation of predicate formula which is quite different from interpretation of propositional formula.

4.8.3 Interpretations of Formulae in FOL

In propositional logic, an *interpretation* simply refers to an assignment of truth values to atoms. Since variables are involved in FOL, we need to do more than a simple assignment of values. An interpretation of a formula α in FOL is not just restricted to assigning truth values; it consists of a non-empty domain D and involves assignment of values to each constant, function symbol and also an assignment of truth values to each predicate atom. Each formula α is evaluated to be *true* or *false* under a given interpretation I over a given domain D.

The following results hold true for any interpretation I over a domain D:

- $(\forall X)\ p(X)$ = *true* if and only if $p(X)$ = *true*, $\forall X \in D$ otherwise it is *false*.
- $(\exists X)\ p(X)]$ = *true* if and only if $\exists\ c \in D$ such that $p(c)$ = *true*, otherwise it is *false*.

Example 4.17

(i) Evaluate the truth value of an FOL formula α: $(\forall X)\ (\exists Y)\ p(X, Y)$ under the following interpretation I:

- $D = \{1, 2\}$
- $p(1, 1) = F, p(1, 2) = T, p(2, 1) = T,\ p(2, 2) = F$

Solution Let us denote *true* by T and *false* by F. For $X = 1$, then $\exists\ 2 \in D$ such that $p(1, 2) = T$ and for $X = 2$, then $\exists\ 1 \in D$ such that $p(2, 1) = T$. Hence, α is *true* under interpretation I.

(ii) Evaluate $\alpha: (\forall X)\,[p(X) \rightarrow q(f(X), c)]$ under the following interpretation:

- $D = \{1, 2\}$
- $c = 1$ (c is a constant from the domain D)
- $f(1) = 2, f(2) = 1$
- $p(1) = F, p(2) = T$
- $q(1, 1) = T, q(1, 2) = T, q(2, 1) = F, q(2, 2) = T$

Solution For $X = 1$

$$p(1) \rightarrow q(f(1), 1) \equiv p(1) \rightarrow q(2, 1) \equiv F \rightarrow q(2, 1) \equiv T$$

For $X = 2$

$$p(2) \rightarrow q(f(2), 1) \equiv p(2) \rightarrow q(1, 1) \equiv T \rightarrow q(1, 1) \equiv T \rightarrow T = T$$

We can easily say that α is true for all values of $X \in D$ under the interpretation I.

4.8.4 Satisfiability and Unsatisfiability in FOL

To study FOL in detail we need to be familiar with the following definitions for a given formula α.

- A formula α is said to be *satisfiable* if and only if there exists an interpretation I such that α is evaluated to be *true* under I. Alternatively, we may say that I is a *model* of α or I satisfies α.
- A formula α is said to be *unsatisfiable* if and only if $\exists$ no interpretation that satisfies α or $\exists$ no model for α.
- A formula α is said to be *valid* if and only if for every interpretation I, α is *true*.
- A formula α is called a *logical consequence* of a set of formulae $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ if and only if for every interpretation I, if $\alpha_1 \wedge ... \wedge \alpha_n$ is evaluated to be *true*, then α is also evaluated to be *true*.

Since there are finite interpretations in propositional logic, it is possible to verify validity, satisfiability, and unsatisfiability of a formula; however, in FOL, there are infinite number of domains and consequently infinite number of interpretations of a formula. Therefore, it is not possible to verify validity and unsatisfiability of a formula by evaluating it under infinite interpretations. We can easily solve this problem in predicate logic by using resolution refutation method, which is similar to propositional logic. In this method, we work with clauses similar to propositional logic; however, obtaining clauses in FOL is not as straightforward as it is in the case of propositional logic, where we convert a given formula to its equivalent CNF notation, each conjunct of which is a clause. In addition to DNF and CNF notations, there is another notation called the *prenex normal form* (PNF), which is used for obtaining clauses from an FOL formula.

4.8.5 Transformation of a Formula into Prenex Normal Form

A formula is said to be in *closed form* if all the variables appearing in it are quantified and there are no free variables.

Prenex Normal Form A closed formula α in FOL is said to be in PNF if and only if α is represented as $(Q_1 X_1)\ (Q_2 X_2)\ \ldots\ (Q_n X_n)\ M$, where Q_k are quantifiers ($\forall$ or $\exists$), X_k are variables, for $1 \leq k \leq n$, while M is a formula free from quantifiers. The list of quantifiers $[(Q_1 X_1)\ \ldots\ (Q_n X_n)]$ is called *prefix* and M is called the *matrix* of a formula α. Here, M is assumed to be represented in CNF notation. For example, $(\exists X)\ (\forall Y)\ [p(X) \vee q(X, Y)]$ is in PNF notation, whereas $(\forall X)\ [p(X) \rightarrow (\exists Y)\ q(X, Y)]$ is not in PNF notation.

Conversion of Formula into PNF Notation

A formula can be easily transformed or converted into PNF using various equivalence laws. The following are some of the conventions that will be used to understand the concept clearly:

- α — FOL formula α without a variable X
- $\alpha[X]$ — FOL formula α which contains a variable X
- Q — Quantifier ($\forall$ or $\exists$).

Equivalence Laws

Although a number of equivalence laws of propositional logic have been studied earlier, the following pairs of logically equivalent formulae need to be discussed in addition. Here, the symbol * represents $\wedge$ or $\vee$.

Law 1 $(QX)\ \alpha[X] * \beta \equiv (QX)\ (\alpha[X] * \beta)$

Law 2 $\alpha * (QX)\ \beta[X] \equiv (QX)\ (\alpha * \beta[X])$

Law 3 $\sim(\forall X)\ \alpha[X] \equiv (\exists X)\ (\sim\alpha[X])$

Law 4 $\sim(\exists X)\ \alpha[X] \equiv (\forall X)\ (\sim\alpha[X])$

Any formula that does not contain X can be brought into the scope of the quantifier Q on X. Therefore, the first two equivalences hold true; these equivalences can be easily proved. Let us prove the last equivalence for the sake of clarity.

Result $\sim(\exists X)\ \alpha[X] \equiv (\forall X)\ (\sim\alpha[X])$

Proof Let I be any interpretation over a domain D. To prove the equivalence law $\sim(\exists X)\alpha[X] \equiv (\forall X)(\sim\alpha[X])$, we have to prove that $\sim(\exists X)\ \alpha[X]$ is *true* if and only if $(\forall X)\ (\sim\alpha[X])$ is *true* under any interpretation I over any domain D.

Let us assume that $\sim(\exists X)\ \alpha[X]$ is *true* under I over D and prove that $(\forall X)\ (\sim\alpha[X])$ is also *true* under I over D. Since $\sim(\exists X)\ \alpha[X]$ is *true* (assumption), it implies that there does not exist any X for which $\alpha[X]$ is *true*, that is, for all X, $\alpha[X]$ is *false*.

Therefore,

$(\forall X)\,(\alpha[X]) = false$

that is, for all X

$\sim\alpha[X]$ is *true*

$\Rightarrow \quad (\forall\alpha)\,(\sim\alpha[X]) = true$

Hence, the result follows. The converse of this law can be proved similarly.

The following are some more equivalence laws that prove to be useful.

Law 5 $(\forall X)\ \alpha[X] \wedge (\forall X)\ \beta[X] \equiv (\forall X)\ (\alpha[X] \wedge \beta[X])$

Law 6 $(\exists X)\ \alpha[X] \vee (\exists X)\ \beta[X] \equiv (\exists X)\ (\alpha[X] \vee \beta[X])$

It should be noted that although quantifiers $\forall$ and $\exists$ may be distributed over $\wedge$ and $\vee$, respectively, they cannot distribute over $\vee$ and $\wedge$, respectively, that is

- $(\forall X)\,\alpha[X] \vee (\forall X)\,\beta[X] \neq (\forall X)\,(\alpha[X] \vee \beta[X])$
- $(\exists X)\,\alpha[X] \wedge (\alpha X)\,\beta[X] \neq (\exists X)\,(\alpha[X] \wedge \alpha[X])$

The following example will make the statement clear to the readers.

Example 4.18 Show that $(\exists X)\,p(X) \wedge (\exists X)\,q(X) \neq (\exists X)\,(p(X) \wedge q(X))$ under the following interpretation:

- $D = \{1, 2\}$
- $p(1) = T, p(2) = F$
- $q(1) = F, q(2) = T$

Solution We can easily see that the formula $(\exists X)\,p(X) \wedge (\exists X)\,q(X)$ is *true* under the interpretation I, whereas $(\exists X)\,(p(X) \wedge q(X))$ is *false* under I. Therefore, we can conclude that

$$(\exists X)\,\alpha(X) \wedge (\exists X)\,\beta(X) \neq (\exists X)\,(\alpha(X) \wedge \beta(X))$$

Similarly

$$(\forall X)\,\alpha(X) \vee (\forall X)\,\beta(X) \neq (\forall X)\,(\alpha(X) \vee \beta(X))$$

can be shown using the same interpretation.

4.8.6 Conversion of PNF to its Standard Form (Skolemization)

The prenex normal form of a given formula can be further transformed into a special form called *skolemization* or *standard form*. This form provides clauses which can then be used in resolution process. The process of eliminating existential quantifiers from the prefix of a PNF notation and

replacing the corresponding variable by a constant or a function is called *skolemization*; such a constant or a function is called *skolem constant* or *skolem function*, respectively. The procedure used to obtain the standard form is described below.

Skolemization Procedure

Let $(Q_1X_1)\ (Q_2X_2)\ \ldots\ (Q_nX_n)M$ be the PNF notation corresponding to some formula, where $(Q_1X_1)\ (Q_2X_2)\ \ldots\ (Q_nX_n)$ represent the prefix, while M denotes a matrix. The steps that need to be followed in the skolemisation procedure are as given:

- Scan prefix from left to right till we obtain the first existential quantifier
  - If Q_1 is the first existential quantifier then choose a new constant $c \notin$ {set of constants in M}. Replace all occurrence of X_1 appearing in matrix M by c and delete (Q_1X_1) from the prefix to obtain new prefix and matrix;
  - If Q_r is the first existential quantifier and $Q_1 \ldots Q_{r-1}$ are universal quantifiers appearing before Q_r, then choose a new $(r-1)$ place function symbol $f \notin$ {set of functions appearing in M}. Replace all occurrence of X_r in M by $f(X_1, \ldots, X_{r-1})$ and remove (Q_rX_r) from prefix;
- Repeat the process till all existential quantifiers are removed from M

Therefore, we observe that any formula α in FOL can be transformed into its standard form. Matrix of the standard formula is in CNF notation and the prefix is free from existential quantifiers. Therefore, the formula in standard form can be expressed as

$$(\forall X_1) \ldots (\forall X_p)\ (C_1 \wedge \ldots \wedge C_m)$$

where each C_k $(1 \leq k \leq m)$ is a formula in disjunctive normal form.

Since all the variables in the prefix are universally quantified in standard form, we can safely omit the prefix from the standard form for the sake of convenience and can write the standard form as $M = (C_1 \wedge \ldots \wedge C_m)$ with the assumption that all variables appearing in M are universally quantified. Now we can easily see that C_k $(1 \leq k \leq m)$ is a clause that has all variables universally quantified. The following subsection contains a formal definition of a clause and statements of some important results.

4.8.7 Clauses in FOL

A *clause* is defined as a closed formula written in the form $(L_1 \vee \ldots \vee L_m)$, where each L_i is a literal and all variables occurring in $L_1, \ldots, L_m$ are universally quantified. The scope of variables appearing in a clause is the clause itself.

Let $S = \{C_1, \ldots, C_m\}$ be a set of clauses that represents a standard form of a given formula α. Then, the following definitions hold true:

- A formula α is said to be *unsatisfiable* if and only if its corresponding set S is unsatisfiable.
- S is said to be *unsatisfiable* if and only if there $\exists$ no interpretation that satisfies all the clauses of S simultaneously.
- S is said to be *satisfiable* if and only if each clause is satisfiable, i.e., $\exists$ an interpretation that satisfies all the clauses of S simultaneously.
- Alternatively, an interpretation I is said to model S if and only if I models each clause of S.

4.8.8 Resolution Refutation Method in FOL

Resolution refutation method in FOL is used to test unsatisfiability of a set (S) of clauses corresponding to the predicate formula. It is an extension of the resolution refutation method described earlier for propositional logic. A deduction of a contradiction from a set S of clauses is called a *resolution refutation* of S. The resolution principle basically checks whether a contradiction is contained in or derived from S.

Resolution for the clauses containing no variables is simple and is similar to that used in propositional logic, but it becomes complicated when clauses contain variables. In such cases, before resolution, two complementary literals are resolved after proper substitutions so that both the literals have same arguments. Let us consider the following example:

Example 4.19 Find the resolvent of two clauses CL_1 and CL_2, where p, q, and r are predicate symbols, X is a variable and f is a unary function.

$$CL_1 = p(X) \vee q(X)$$

$$CL_2 = \sim p(f(X)) \vee r(X)$$

Solution If we substitute $f(a)$ for X in CL_1 and a for X in CL_2, where a is a new constant from the domain, then we obtain

$$CL_3 = p(f(a)) \vee q(f(a))$$

$$CL_4 = \sim p(f(a)) \vee r(a)$$

We observe that clauses CL_3 and CL_4 have complementary literals as $p(f(a))$ and $\sim p(f(a))$. The resolvent is generated by taking the disjunction of literals from parent clauses after eliminating pair of complementary literals. Therefore, we get resolvent of CL_3, and CL_4 as $CL = q(f(a)) \vee r(a)$.

Here we notice that CL_3 and CL_4 do not have variables. These are called ground instances of CL_1 and CL_2. In general, if we substitute $f(X)$ for X in CL_1, then we get another clause as follows:

$$CL_1' = p(f(X)) \vee q(f(X))$$

Then, CL' = Resolvent $(CL_1', CL_2) = [q(f(X)) \vee r(X)]$

We notice that CL is an instance of CL', which is a general form, and can be obtained from CL' by substituting a for X.

Assuming that set of clauses is given by $S = \{C_1, \ldots, C_m\}$ and L is some predicate formula, we can state some important results as given below.

- L is a logical consequence of S if and only if $\{S \cup \sim L\} = \{C_1, \ldots, C_m, \sim L\}$ is unsatisfiable. Alternatively, we can say that the goal L is deduced from S. Here $\sim L$ may give rise to more than one clause. For example, if $L = p(X) \vee (q(X) \wedge r(X))$, then

  $$\begin{aligned}\sim L &= \sim[p(X) \vee (q(X) \wedge r(X))] \\ &= \sim p(X) \wedge \sim(q(X) \wedge r(X)) \\ &= \sim p(X) \wedge (\sim q(X) \vee \sim r(X))\end{aligned}$$

  Therefore, $\sim L$ is converted to two clauses $\{\sim p(X), \sim q(x) \vee \sim r(X)\}$
- Soundness and completeness of resolution theorem states that there is a *resolution refutation* of S if and only if S is *unsatisfiable*. A deduction of a contradiction from a set S of clauses is called a *resolution refutation* of S.
- Alternatively, L is said to be a logical consequence of S if and only if there is a resolution refutation of $S \cup \{\sim L\}$.

Using the results given above, we can state that L is a logical consequence of the set of formulae $\{\alpha_1, \ldots, \alpha_n\}$ if and only if $S \cup \{\sim L\}$ is unsatisfiable, where S is a set of all the clauses obtained from each formula α_k.

In conclusion, if we have to show that L is a logical consequence of a given set of formulae $\alpha_1, \ldots, \alpha_n$, then the procedure described below needs to be used:

Logical Consequence Procedure

- Obtain a set S of all the clauses by converting each formula α_k to its corresponding standard form. Further, clauses can be obtained from standard form as explained in Section 4.8.6.
- Show that a set $S \cup \{\sim L\}$ is unsatisfiable by deducing a contradiction using resolution method.
- If a contradiction is obtained then conclude that L is a logical consequence of S and subsequently of the formulae $\alpha_1, \ldots, \alpha_n$, otherwise L is not a logical consequence.

Choice of Clauses to be Resolved

It is important to choose the clauses to be resolved reasonably. If the choice of clauses that need to be resolved at each step is made in a systematic way, the resolution algorithm will surely find a contradiction, if one exists. There are various heuristics for making the right choice that can speed up the process considerably. The steps can be listed as follows:

- As the first step, choose a clause from the negated goal clauses as one of the parents to be resolved; this is done because the contradiction, if it exists, might be occurring due to the goal we are trying to prove.
- Use the resolvent for further resolution with an existing clause. Both the clauses should contain one pair of complementary literals, if possible. If parent clauses contain more than one pair of complementary literals, then resolvent is always true.
- If such clauses do not exist, then resolve any pair of clauses that may contain complementary literals.
- Resolve using clauses with a single literal whenever possible. Such resolutions generate new clauses with fewer literals than the larger of their parent clauses; thus, the algorithm may most probably terminate faster.
- Eliminate tautologies as soon as they are generated.

Example 4.20 Show that the formula $\alpha : (\forall X)\ (p(X) \wedge \sim[q(X) \rightarrow p(X)])$ is unsatisfiable.

Solution To prove the above statement, we need to convert α into a set of clauses with the help of equivalence laws.

$$\begin{aligned} p(X) \wedge \sim[q(X) \rightarrow p(X)] &\equiv p(X) \wedge \sim[\sim q(X) \vee p(X)] \\ &\equiv p(X) \wedge \sim \sim q(X) \wedge \sim p(X) \\ &\equiv p(X) \wedge q(X) \wedge \sim p(X) \end{aligned}$$

The set of clauses is written as $S = \{p(X), q(X), \sim p(X)\}$. Since there is a contradiction in S itself [because of $p(X)$ and $\sim p(X)$], S is unsatisfiable and consequently α is unsatisfiable.

Example 4.21 Show that $q(a)$ is a logical consequence of formulae α and β where

$\alpha : (\forall X)\ [p(X) \rightarrow q(X)]$

$\beta : p(a)$

Solution To prove the statement given in Example 4.21, we need to convert α and β into a set of clauses by using equivalence laws. The set of clauses, S, may be written as

$S = \{\sim p(X) \vee q(X), p(a)\}$

When we add the negation of goal, that is, $\sim q(a)$ to S, we get the new set, S', where $S' = \{\sim p(X) \vee q(X), p(a), \sim q(a)\}$. The resolution tree is given in Fig. 4.3.

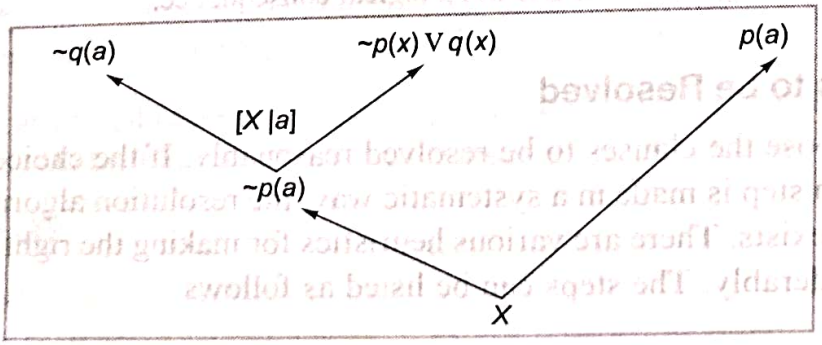


Figure 4.3 Resolution Tree for Example 4.21

6

Advanced Problem-Solving Paradigm: Planning

6.1 Introduction

One of the key abilities of intelligent systems is planning. Planning refers to the process of computing several steps of problem-solving before executing any of them. Planning increases the autonomy and flexibility of systems by constructing sequences of actions that help them in achieving their goals. Planning is an area of current interest within AI. One of the reasons for this is that it combines two major areas of AI: *search* and *logic*. Planning involves the representation of actions and world models, reasoning about the effects of actions, and techniques for efficiently searching the space of possible plans. A planner may therefore be either a program that searches for a solution or one that proves the existence of a solution. Planning helps in controlling combinatorial explosion while solving problems. In order to solve non-trivial problems, it is necessary to combine basic problem-solving strategies and knowledge representation mechanisms. Planning involves further to integrate partial solutions for decomposable problems into a complete solution at the end. Planning proves to be useful as a problem-solving technique in case of non-decomposable problems. In general, problem-solving systems, elementary techniques are required to perform the following functions (Rich & Knight, 2003).

Choosing the Best Rule (Based on Heuristics) to be Applied For selecting appropriate rules, the most widely used technique is to first isolate a set of differences between the desired goal state and current state. Then, those rules are identified that are relevant for reducing this difference. If more than one rule is found, then heuristic information is applied to choose an appropriate rule.

Applying the Chosen Rule to Obtain a New Problem State In simple-problem-solving systems, it is easy to apply rules as each rule specifies the problem state that would result from its application. However, in complex problems, we have to deal with rules that only specify a small part of the complete problem state.

Detecting When a Solution is Found By the time the required goal state is reached, the solution of each sub problem is found. The solutions so obtained are then combined to give the final solution of the problem.

Detecting Dead Ends so that New Directions may be Explored When a situation arises where we are not able to proceed further from a particular state, which is not the desired goal state, we can declare the state to be a *dead end* and can proceed further through a new direction.

In complex systems, we have to explore techniques to detect when an almost correct solution is found and then employ special techniques to make it absolutely correct. Planning can also be done by proving a theorem in situation calculus which states that:

> Given the axioms of the start and successor states, which describe the effects of actions, the goal will be true in a situation that results from a certain action sequence.

Planning techniques are applied in a variety of tasks including robotics, process planning, web-based information gathering, autonomous agents, spacecraft mission control, and so on. Three phases of advanced problem-solving are recognized: *planning, execution*, and *learning*. The functions carried out in each phase are described as follows:

- **Planning phase** In this phase, a set of actions called a *plan* is generated, which transforms a given start state to the goal state.
- **Execution phase** In this phase, each action of the plan generated in the preceding phase is performed starting with the start state.
- **Learning phase** In this phase, plan generation and plan execution are learned through experience.

Generally, for a large number of discrete planning problems, the state space is enormous. Thus, there is a need to construct *implicit* encodings of problems so that the entire state space need not be explored by the algorithm for solving a given problem. Logic-based representations have been popularly used for constructing implicit representations; they are convenient for producing output, which logically explains the steps involved in arriving at some goal. A prominent shortcoming of logic-based representations is that they are difficult to generalize. This is why state-space representation is generally used. In this chapter, we will discuss various techniques of plan generation for both linear and non-linear planning.

6.2 Types of Planning Systems

The proper representation of planning problems is an important issue and needs to be discussed at length. Planning problems involve, representing states, actions, and goals. An ideal language would be one which is expressive enough to describe a wide variety of problems, but restrictive enough to allow efficient algorithms to operate over it. The different components may be represented in the following manner:

Representation of States For representation of states, planners decompose the world into logical conditions and then represent a state as a conjunction of predicate atoms (positive literals).

Representation of Goals A goal is a partially specified state and is represented as a conjunction of predicate atoms (ground literals).

Representation of Actions An action is specified in terms of preconditions (that must hold true before the action can be executed) and effects (that ensue when the action has been executed).

A number of formulations have been used so far that attempt to solve the planning problems, such as operator-based planning, case-based planning, logic-based planning, constraint-based planning, distributed planning, etc.

6.2.1 Operator-Based Planning

Before we attempt to solve a problem with the help of planning, we need to have a start state, a set of actions, a goal state, and a database of logical sentences about the start state. Once these are specified, the *planner* will try to generate a plan, which when executed by the *executor* in a state S (start state), satisfying the description of the start state, will result in the state G (goal state), satisfying the description of the final state. Here, actions are represented as *operators*. This approach, also known as the STRIPS approach (explained later), utilizes various *operator schemas* and *plan representations*. The major design issues and concepts are given as follows and have been explained in the following sections appropriately.

- **Operator schema** Add–delete–precondition lists, procedural vs declarative representations, etc.
- **Plan representations** Linear plans, non-linear plans, hierarchical plans, partial-order plans, conditional plans, etc.
- **Planning algorithms** Planning as search, world-space vs plan-space, partial-order planning, total-order planning, progression, goal-regression, etc.
- **Plan generation** Plan reformulation, repair, total-ordering, etc.

6.2.2 Planning Algorithms

Search techniques for planning involve searching through a search space. We now introduce the concept of *planning as a search* strategy. In search technique method, there are basically two approaches: searching a *world* (or *state*) space or searching a *plan* space. The concepts of world (or state) space and plan space may be defined as given below.

World-Space In world space, the *search space* constitutes a set of states of the world, an *action* is defined as transitions between states, and a *plan* is described as a path through the state space. In state space, it is easy to determine which sub goals are achieved and which actions are applicable; however, it is hard to represent concurrent actions.

Plan-Space In plan space, the *search space* is a set of plans (including partial plans). The start state is a null plan and transitions are plan operators. The order of the search is not the same as plan execution order. A shortcoming of the plan space is that it is hard to determine what is true in a plan. Both the approaches are discussed in detail as follows:

Searching a World Space

Each node in the state search graph denotes a *state* of the world, while arcs in the graph correspond to the execution of a specific action. The planning problem is to find a path from a given start state to the desired goal state in the search graph. For developing planning algorithms, one of the following two approaches may be used:

i. **Progression** This approach refers to the process of finding the goal state by searching through the states generated by actions that can be performed in the given state, starting from the start state. It is also referred to as the *forward chaining approach*. Here at a given state, an action (may be non-deterministic) is chosen whose preconditions are satisfied. The process continues until the goal state is reached.

ii. **Regression** In this approach, the search proceeds in the backward direction, that is, it starts with the goal state and moves towards the start state. This is done by finding actions whose effects satisfy one or more of the posted goals. Posting the preconditions of the chosen action as goals is called *goal regression*. It is also known as *backward chaining approach*. Here, we choose an action (which may be non-deterministic) that has an effect that matches an unachieved sub goal. Unachieved preconditions are added to the set of sub goals and this process is continued till the set of unachieved sub goals becomes empty. In regression, the focus is on achieving goals and is thus often more efficient.

It is to be noted that algorithms based on both the approaches are sound and complete. An algorithm is said to be *sound* if the plan generated succeeds in completing the desired job, and it is said to be *complete* if it guarantees to find a plan, if one exists. However, in most situations, regression is found to be a better strategy.

Searching a Plan Space

Each node in plan space graph represents *partial plans*, while arcs denote *plan refinement operations*. One can search for either a plan with a totally-ordered sequence of actions or a plan with a partially-ordered set of actions. A partial-order plan has the following three components:

i. ***Set of actions*** As an example of a set of actions, we can consider some of the activities from our daily routine, such as *go-for-morning-walk*, *wake-up*, *take-bath*, *go-to-work*, *go-to sleep*, and so on.

ii. ***Set of ordering constraints*** In the actions mentioned above, the action *wake-up* is performed before *go-for-morning-walk*; therefore, we can represent it as [wake-up ← go-for-morning-walk]. Some of the partial ordering constraints in the set of actions given above may be written as follows:

wake-up	←	go-for-morning-walk
wake-up	←	take-bath
wake-up	←	go-to-sleep
wake-up	←	go-to-work
go-for-morning-walk	←	go-to-work
go-for-morning-walk	←	go-to-sleep
take-bath	←	go-to-sleep
go-to-work	←	go-to-sleep

While most activities will have to follow a certain order, some may not necessarily obey the constraints. For example, [take-bath ← go-to-work] may not have this strict ordering as one may go to work without taking bath or one may take bath after returning from work. The ordering [go-for-morning-walk ← go-to-work] has to be in constraint as specified as *go-for-morning-walk* activity cannot follow *go-to-work* activity.

iii. ***Set of causal links*** We observe that *awake* is a link from the action *wake-up* to the action *go-for-morning-walk* with an awakening state of the person between them represented as '*wake-up—awake—go-for-morning-walk*'. This denotes a *casual link*. When the action *wake-up* is added to the generated plan, the above causal link is also recorded along with the ordering constraint [wake-up ← go-for-morning-walk]. This is because the effect of the action *wake-up* is that the individual is awake; this is a precondition of the action *go-for-morning-walk*. The second action cannot occur until this precondition is satisfied. Causal links help in detecting inconsistencies whenever a partial plan is refined.

6.2.3 Case-Based Planning

In case-based planning, for a given case (consisting of start and goal states) of a new problem, the library of cases in case base is searched to find a similar problem, with similar start and goal states. The retrieved solution is then modified or tailored according to the new problem. Thus,

case-based planning helps in utilizing *specific* knowledge obtained by previous experience. This approach is based on human methodology of tackling a problem. Humans also obtain solutions to their problems by learning from their experience (Althoff & Aamodt, 19960) and solve new problem by finding a similar case handled in the past. So, in case-based planning, a new problem is matched against the cases stored in the case base (past experience) and one or more similar cases are *retrieved*. A solution suggested by the matched cases is then *reused* and tested for success. Unless the retrieved case is a close match, the solution will have to be *revised*, which produces a new case that can then be *retained* as a part of learning. An initial description of a problem defines a *new case*. This planning procedure is described as a cyclical process consisting of the following steps:

Retrieve	The most similar cases are retrieved from the case base using various methods.
Reuse	The cases are reused in an attempt to solve a new problem;
Revise	The proposed solution of the retrieved case is revised and modified, if necessary, and
Retain	The new solution is retained as a part of learning if it is very different from the retrieved case.

Case-based reasoning (CBR) systems will be discussed in detail in Chapter 11.

6.2.4 State-Space Linear Planning

In the process of linear planning, only one goal is solved at a time. It requires a simple search strategy that uses a stack of unachieved goals. The advantages and disadvantages of state-space linear planning are discussed as follows:

Advantages of Linear Planning

- Since the goals are solved one at a time, the search space is considerably reduced.
- Linear planning is especially advantageous if goals are (mainly) independent.
- Linear planning is sound.

Disadvantages of Linear Planning

- It may produce sub-optimal solutions (based on the number of operators in the plan).
- Linear planning is incomplete.
- In linear planning, the planning efficiency depends on ordering of goals.

6.2.5 State-Space Non-Linear Planning

In non-linear planning, the sub goals may be solved in any order and they may be interdependent. In this process, the basic idea is to use a goal set instead of a goal stack. All possible sub-goal orderings are included in the search space and the goal interactions are handled by interleaving. The advantages and disadvantages of state-space non-linear planning are discussed as follows:

Advantages of Non-Linear Planning

- Non-linear planning is sound and complete.
- It may be optimal with respect to plan length (depending on the search strategy employed).

Disadvantages of Non-Linear Planning

- Since all possible goal orderings may have to be considered, a larger search space is required in non-linear planning.
- Non-linear planning requires a more complex algorithm and a lot of book-keeping.

In the following section, we will discuss planning methods using a specific example of block world problem and explain the procedures of generating plans to solve goals using linear and non-linear methods.

6.3 Block World Problem: Description

A block world problem basically consists of handling blocks and generating a new pattern from a given pattern (Rich and Knight, 2003). This problem closely resembles a game of block arrangement and construction usually played by children. Our aim is to explain strategies that may lead to a plan (or steps), which may help a robot to solve such problems. For this, let us consider the following assumptions:

- All blocks are of the same size (square in shape).
- Blocks can be stacked on each other.
- There is a flat surface (table) on which blocks can be placed.
- There is a robot arm that can manipulate the blocks. The arm can hold only one block at a time.

In the block world problem, a state is described by a set of predicates, which represent the facts that are true in that state. For every action, we describe the changes that the action makes to the state description. In addition, some statements regarding the things which remain unchanged by applying actions are also to be specified. For example, if a robot picks up a block, the colour of the block will not change. This is the simplest possible approach and is described below. Table 6.1 provides descriptions of the operators or actions used for this problem.

Actions (Operations) Performed by Robot

To understand the concept of block world problem, let us use the following convention:

- Capital letters X, Y, Z, ..., are used to denote variables
- Lowercase letters a, b, c, ... are used to represent specific blocks

The description of various operators or actions used in this problem is given in Table 6.1.

Table 6.1 Description of Operators or Actions Used in Block World Problem

Operators	Short Form	Description
UNSTACK(X, Y)	US(X, Y)	Pick up block X from block Y (current position is that X is on Y). The arm must be empty and top of X must be clear.
STACK(X, Y)	ST(X, Y)	Put block X on the top of block Y. The arm must be holding block X. Top of Y should be clear.
PICKUP(X)	PU(X)	Pick up block X from the table and hold it. Initially the arm must be empty and top of X should be clear.
PUTDOWN(X)	PD(X)	Put down block X on the table. The arm must be holding block X before putting down.

In block world problem, certain predicates are used to represent the problem states for performing the operations given in Table 6.1. These predicates are described in Table 6.2.

Table 6.2 Predicates Used in Block World Problem

Predicates	Description
ON(X, Y)	Block X is on the block Y
ONTABLE(X)	Block X is on the table
CLEAR(X)	Top of X is clear
HOLDING(X)	Robot arm is holding X
ARMEMPTY	Robot arm is empty

In the following sections, we will discuss logic-based planning, linear planning, and non-linear planning.

6.4 Logic-Based Planning

We will use the block world problem explained in the preceding section to explain the concept of logic-based planning. In this approach, we have to explicitly state all possible logical statements that are true in the block world problem. Some of these logical statements are described as follows:

- If the robot arm is holding an object, then arm is not empty
  $(\exists X)\,\text{HOLDING}(X) \rightarrow \sim \text{ARMEMPTY}$
- If the robot arm is empty, then arm is not holding anything
  $\text{ARMEMPTY} \rightarrow \sim (\exists X)\,\text{HOLDING}(X)$
- If X is on a table, then X is not on the top of any block
  $(\forall X)\,\text{ONTABLE}(X) \rightarrow \sim (\exists Y)\,\text{ON}(X, Y)$

- If X is on the top of a block, then X is not on the table
  $(\forall X)(\exists Y)\, ON(X, Y) \rightarrow \sim ONTABLE(X)$
- If there is no block on top of block X, then top of block X is clear
  $(\forall X)(\sim(\exists Y)\, ON(X, Y)) \rightarrow CLEAR(X)$
- If the top of block X is clear, then there is no block on the top of X
  $(\forall X)\, CLEAR(X) \rightarrow \sim(\exists Y)\, ON(Y, X)$

Further, the axioms reflecting the effect of the operations mentioned in Table 6.2 have to be described on a given state. Let us assume that a function named GEN generates a new state from a given state as a result of the application of some operator/action. For example, if the action OP is applied on a state S and a new state S1 is generated then S1 is written as S1= GEN(OP, S).

- The effect of UNSTACK(X, Y) in state S is described by the following axiom.
  $[CLEAR(X, S) \wedge ON(X, Y, S) \wedge ARMEMPTY(S)] \rightarrow$
  $[HOLDING(X, S1) \wedge CLEAR(Y, S1)]$

Here, S1 is a new state obtained after performing the UNSTACK operation in state S. If we execute UNSTACK(X, Y) in S, then we can prove that HOLDING(X, S1) $\wedge$ CLEAR (Y, S1) holds true. Here state is introduced as an argument of an operator.

Since the interpretation of the axioms showing effect of the following operations is self explanatory, we will omit these now onwards.

- The effect of STACK(X, Y) in state S is described as follows.
  $[HOLDING(X, S) \wedge CLEAR(Y, S)] \rightarrow$
  $[ON(X, Y, S1) \wedge CLEAR(X, S1) \wedge ARMEMPTY(S1)]$
- The effect of PU(X) in state S is described by the following axiom.
  $[CLEAR(X, S) \wedge ONTABLE(X, S) \wedge ARMEMPTY(S)] \rightarrow$
  $[HOLDING(X, S1)]$
- The effect of PD(X) in state S is described by as follows.
  $[HOLDING(X, S)] \rightarrow$
  $[ONTABLE(X, S1) \wedge CLEAR(X, S1) \wedge ARMEMPTY(S1)]$

It should be noted here that after any operation is carried out on a given state, we cannot comment about other situations in the new state S1. For example, after the operation UNSTACK(X, Y), we cannot make a statement regarding the current position of Y, that is, whether Y is still on the table or on another block. Similarly, we cannot say that properties such as colour or weight of any block in the new state are same as the previous state. Therefore, there might be many such properties which do not change with change in state. So, we have to provide a set of rules known as *frame axioms* for the properties that do not get affected in the new state with the application of each

operator. Table 6.3 lists a few such situations which will not get affected by using the UNSTACK operator. We can define frame axioms for other operators in a similar manner.

Table 6.3 Frame Axioms for Operator UNSTACK

Current State S0	implies	State S1 = GEN(UNSTACK (X, Y), S0))
If block Z or Y is on a table in state S0, then UNSTACK(X, Y) in state S0 will not affect block Z or Y in the state S1		
ONTABLE(Z, S0)	→	ONTABLE(Z, S1)
ONTABLE(Y, S0)	→	ONTABLE(Y, S1)
If block Z is on block W or Y is on any block U, then UNSTACK(X, Y) in state S0 will not affect blocks Z and Y in the state S1		
ON(Z, W, S0)	→	ON(Z, W, S1)
ON (Y, U, S0)	→	ON(Y, U, S1)
If colour of blocks X, Y, or any block Z is same (say, red) in state S0 then it remains the same in state S1 after UNSTACK(X, Y) operation is performed		
COLOR(X, C, S0)	→	COLOR(X, C, S1)
COLOR(Y, C, S0)	→	COLOR(Y, C, S1)
COLOR(Z, C, S0)	→	COLOR(Z, C, S1)
ON relation is not affected by UNSTACK operator if the blocks involved in ON relation are different from those involved in UNSTACK operation		
ON(Z, W, S0) A NE (Z, X)	→	ON(Z, W, S1)

The advantage of this approach is that only a simple mechanism of resolution needs to be performed for all the operations that are required on the state descriptions. On the other hand, the disadvantage of this approach is that in case of complex problems, the number of axioms becomes very large as we have to enumerate all those properties which are not affected, separately for each operation. Further, if a new attribute is introduced into the problem, it becomes necessary to add a new axiom for each operator.

For handling the complex problem domain, we need a mechanism that does not require a large number of frame axioms to be specified explicitly. Such mechanism was used in early robot problem-solving system known as STRIPS (STanford Research Institute Problem Solver), which was developed by Fikes and Nilsson in 1971. Each operator in this approach is described by a list of new predicates that become true and a list of old predicates that become false after the said operator is applied. These are called ADD and DEL (delete) lists, respectively. There is another list called PRE (preconditions), which is specified for each operator; these preconditions must be true before an operator is applied. Any predicate which is not included on either ADD or DEL list of an operator is assumed to remain unaffected by it. Frame axioms are specified implicitly in STRIPS; this greatly reduces the amount of information stored.

STRIPS-Style Operators

We observe that by making the frame axioms implicit, we have greatly reduced the amount of information that needs to be provided for each operator. Now, we only need to specify the required effect; the unaffected attributes are not included. Therefore, in this representation, if a new attribute is added, the operator lists do not get changed. Henceforth, we will use short forms of predicates as given in Table 6.4 for the sake of convenience.

Table 6.4 Short Forms of Predicates

Predicates	ON(X, Y)	ONTABLE(X)	CLEAR(X)	HOLDING(X)	ARMEMPTY
Short Form	O(X, Y)	T(X, Y)	C(X)	H(X)	AE

The three lists (PRE, DEL, and ADD) required for each operator are given in Table 6.5.

Table 6.5 Operator with PRE, DEL and ADD Lists

Operator	PRE list	DEL list	ADD list
ST(X, Y)	C(Y) ∧ H(X)	C(Y) ∧ H(X)	AE ∧ O(X, Y)
US(X, Y)	O(X,Y) ∧ C(X) ∧ AE	O(X, Y) ∧ AE	H(X) ∧ C(Y)
PU(X)	T(X) ∧ C(X) ∧ AE	T(X) ∧ AE	H(X)
PD(X)	H(X)	H(X)	T(X) ∧ AE

The state description is updated after each operation by deleting those predicates from the state that are present in the DEL list and adding those predicates to the state that are present on the ADD list of the operator applied. If an incorrect sequence is explored accidentally, then it is possible to return to the previous state so that a different path may be tried.

6.5 Linear Planning Using a Goal Stack

The goal stack method is one of the earliest methods that were used for solving compound goals, which may not interact with each other. This approach was used by STRIPS systems. In this system, the problem solver makes use of a single stack containing goals as well as operators that have been proposed to satisfy these goals. In goal stack method, individual sub goals are solved linearly and then, at the final stage, the conjoined sub goals are solved. Plans generated by this method contain complete sequence of operations for solving the first goal followed by complete sequence of operations for the next one, and so on. The problem solver uses database that describes the current state and the set of operators with PRE, ADD, and DEL lists.

6.5.1 Simple Planning using a Goal Stack

In this section, we will discuss the method of simple planning using a goal stack. Let us explain the method by using hypothetical goals. Consider the following goal that consists of sub goals G1, G2, ..., Gn.

GOAL = G1 ∧ G2 ∧ ... ∧ Gn

The sub goals G1, ..., Gn are stacked (in any order) with a compound goal G1 ∧ ...∧ Gn at the bottom of the stack. The status of stack is shown in Fig. 6.1.

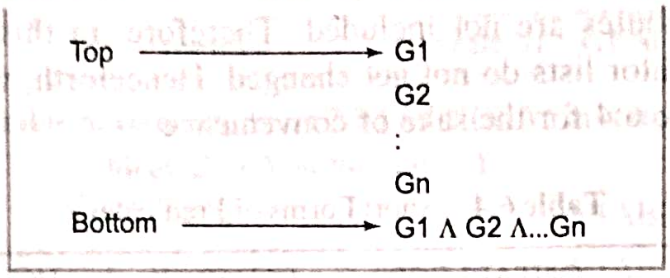


Figure 6.1 Status of Stack

The algorithm that is required to solve the goal is given below.

Algorithm 6.1 *GOAL_Stack*

```
{
• Push conjoined sub goals and individual sub goals onto Stack;
• flag = true;
While (Stack ≠ ϕ and flag = true) do
{ If top element of stack is an operator then
   {
   • POP it and add to PLAN_QUEUE of operations to be performed in a
     plan;
   • Generate new state from current state by using ADD and DEL
     lists of an operator
   }
   Else
   { If top of the stack is sub goal and is true in the current
     state then POP it
     Else
     {
        • Identify operator(s) that satisfies top sub goal of the stack;
        • If (no operator exists) then set flag=false
          Else
          {
             • Choose one operator that satisfies the sub goal (use
               some heuristic);
             • POP sub goal and PUSH chosen operator along with its
               preconditions in the stack;
          }
     }
   }
• If (flag = false) then problem solver returns no plan else returns
  the plan stored in PLAN_QUEUE for the problem;
}
}
```

6.5.2 Solving Block World problem using Goal Stack method

To illustrate the working of Algorithm 6.1, consider the following example, where the start and goal states of block world problem are shown in Fig. 6.2. Here a, b, c, and d are specific blocks.

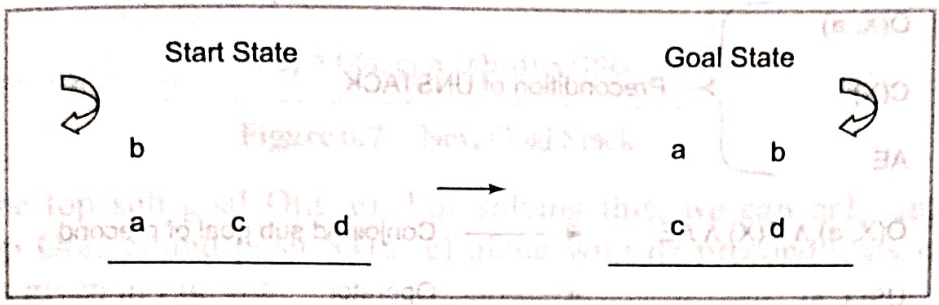


Figure 6.2 Start and Goal States of a Block World Problem

The logical representations of start and goal states may be written as

Start State: O(b, a) Λ T(a) ΛT(c) Λ T(d) Λ C(b) Λ C(c) Λ C(d) Λ AE
Goal State: O(a, c) Λ O(b, d) Λ T(c) Λ T(d) Λ C(a) Λ C(b) Λ AE
PLAN_QUEUE = ϕ

We notice that (T(c) Λ T(d) Λ C(b) Λ AE) is true in both start and goal, states. Hence, for the sake of convenience, we can represent it by CSG (conjoined sub goals present in both the states). We will work to solve sub goals O(a, c), O(b, d), and C(a) and while solving these sub goals, we will make sure that CSG remains true. We will first put these sub goals in some order in the stack. Let the initial status of the stack be as shown in Fig. 6.3.

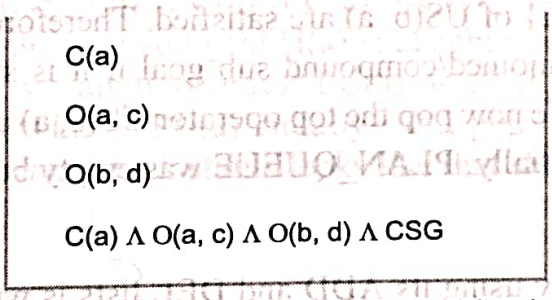


Figure 6.3 Initial Status of Goal Stack

Now, we need to identify an operator that can solve C(a). We notice that the operator US(X, a) can only be applied, where X gets bound to the actual block on top of 'a'. Here pop C(a) and-push US(X, a) in the goal stack. The status of the goal stack changes as shown in Fig. 6.4.

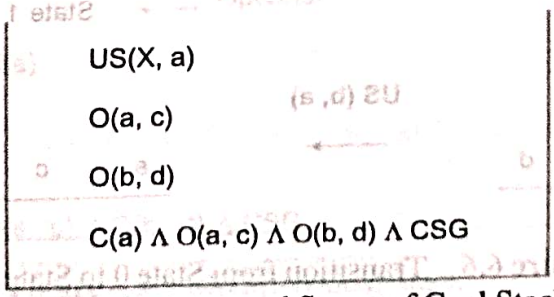


Figure 6.4 Changed Status of Goal Stack

Since stack operator US(X, a) can be applied only if its preconditions are true, therefore, we add its preconditions on top of the stack. The changed status of stack now looks as shown in Fig. 6.5.

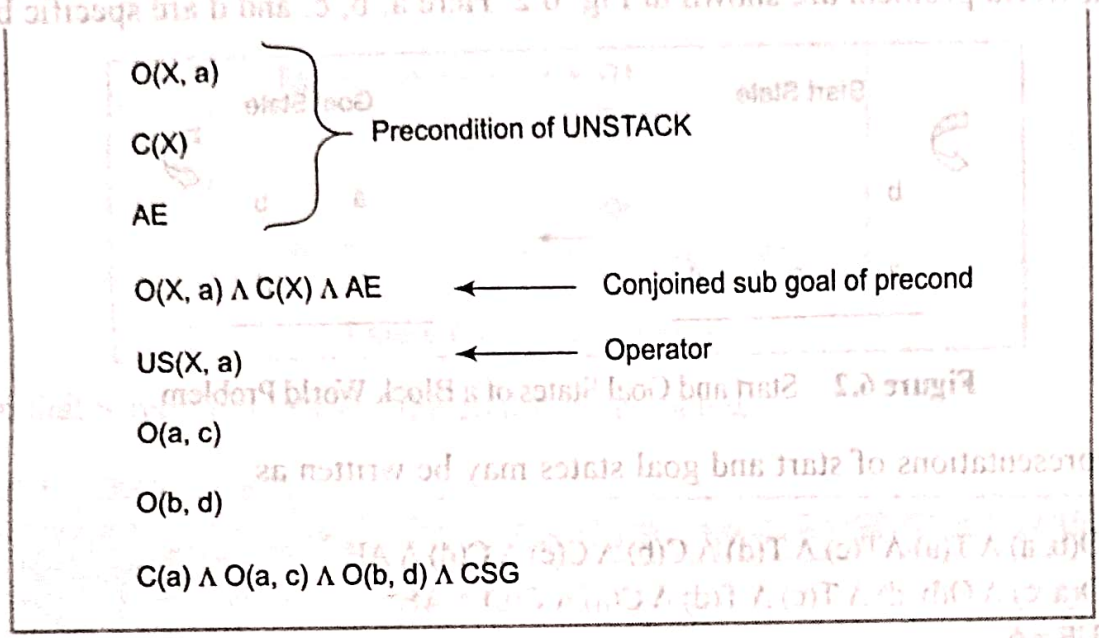


Figure 6.5 Status of Stack on Addition of Preconditions

The start state of the problem may be written as State 0

State 0 (Start state) $O(b, a) \wedge T(a) \wedge T(c) \wedge T(d) \wedge C(b) \wedge C(c) \wedge C(d) \wedge AE$

From State 0, we find that 'b' is on top of 'a', so the variable X is unified with block 'b'. Now, all preconditions {O(b, a), C(b), AE} of US(b, a) are satisfied. Therefore, the next step is to pop these preconditions along with its conjoined/compound sub goal if it is still true. In this case, we find compound sub goal to be true. We now pop the top operator US(b, a) and add it in a PLAN_QUEUE of the sequence of operators. Initially, PLAN_QUEUE was empty but now it contains US(b, a).

PLAN_QUEUE = US(b, a)

A new state State 1 produced by using its ADD and DEL lists is written as

State 1 $T(a) \wedge T(c) \wedge T(d) \wedge H(b) \wedge C(a) \wedge C(c) \wedge C(d)$

The transition from State 0 to State 1 is shown in Fig. 6.6.

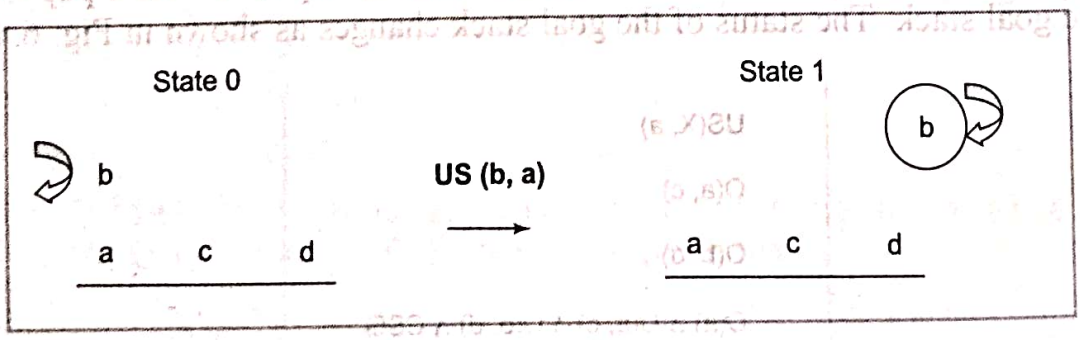


Figure 6.6 Transition from State 0 to State 1

The new goal stack is shown in Fig. 6.7.

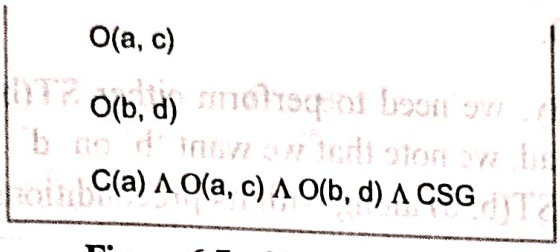


Figure 6.7 New Goal Stack

Now, let us solve the top sub goal O(a, c). For solving this, we can only apply the operator ST(a, c). So, we pop O(a, c) and push ST(a, c) along with its preconditions in the stack. The changed goal stack is given in Fig. 6.8.

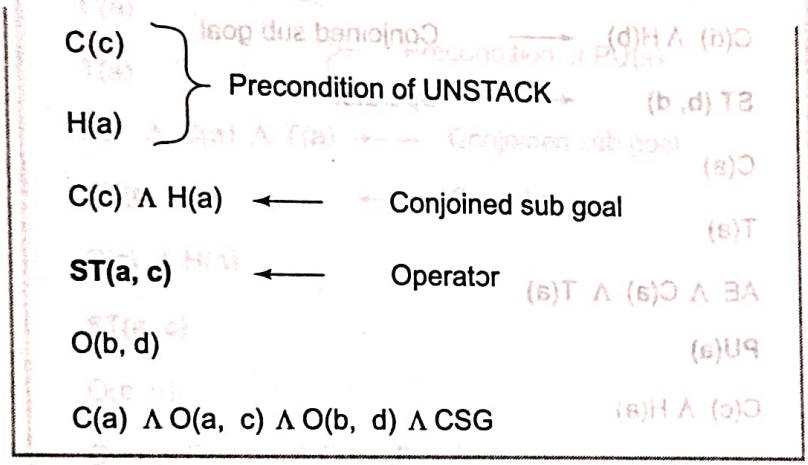


Figure 6.8 Changed Goal Stack

From *State 1*: {T(a) ΛT(c) Λ T(d) Λ H(b) Λ C(a) Λ C(c) Λ C(d)}, we notice that C(c) is true, so we pop it. Then, we observe that the next sub goal H(a) is unachieved (not true), so we will solve this. For making H(a) to be true, we apply operator PU(a) or UN(a, X). In fact, any of the two operators can be applied but let us choose PU(a) initially. Now pop H(a) and push PU(a) with its preconditions to the stack. The current stack status looks like that shown in Fig. 6.9.

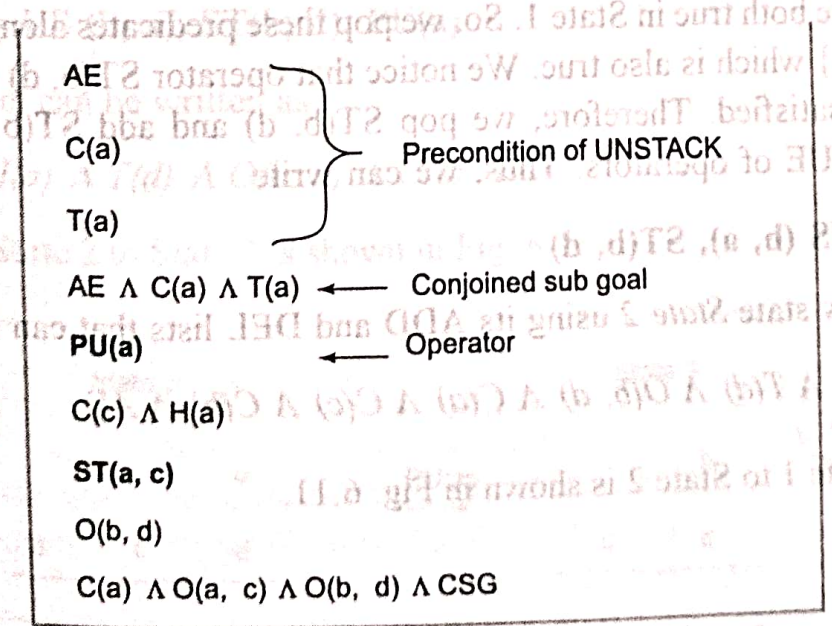


Figure 6.9 Modified Goal Stack

Now top sub goal AE above stack is to be solved. We notice that AE is not true as the arm is holding 'b' (as given in State 1).

In order to make the arm empty, we need to perform either ST(b, X) or PD(b). Let us choose ST(b, X). If we look a little ahead, we note that we want 'b' on 'd'. Therefore, we need to unify X with 'd'. So, we replace AE by ST(b, d) along with its preconditions. The goal stack now changes to that shown in Fig. 6.10.

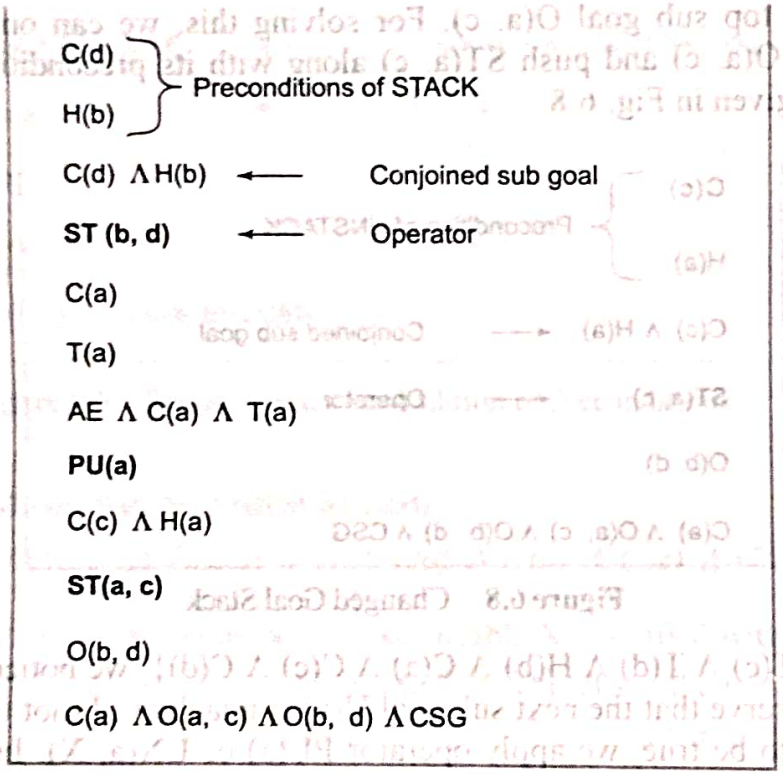


Figure 6.10 New Goal Stack

Now, C(d) and H(b) are both true in State 1. So, we pop these predicates along with the compound sub goal {C(d) Λ H(b)} which is also true. We notice that operator ST(b, d) can be applied as all its preconditions are satisfied. Therefore, we pop ST(b, d) and add ST(b, d) in the queue of sequence PLAN_QUEUE of operators. Thus, we can write

PLAN_QUEUE = US (b, a), ST(b, d)

Now we produce a new state *State 2* using its ADD and DEL lists that can be written as

State 2 *T(a) Λ T(c) Λ T(d) Λ O(b, d) Λ C(a) Λ C(c) Λ C(b) Λ AE*

The transition from State 1 to State 2 is shown in Fig. 6.11.

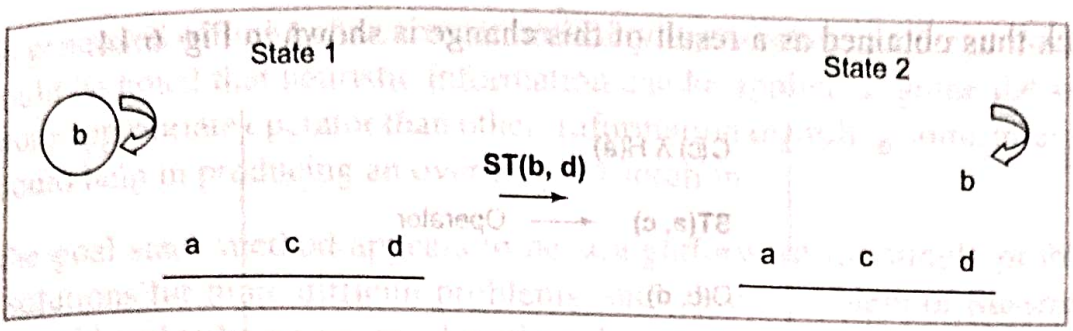


Figure 6.11 Transition from State 1 to State 2

The goal stack obtained as a result of this transition is shown in Fig. 6.12.

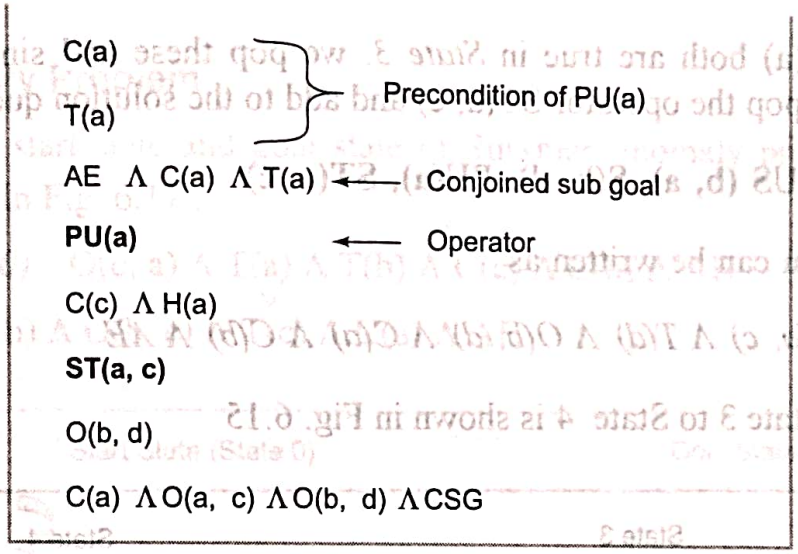


Figure 6.12 New Goal Stack

Now, AE, C(a) and T(a) are true (from State 2); hence, preconditions of PU(a) are satisfied. As a result, the operation PU(a) can be performed. Therefore, we pop it and add it in the queue of the sequence of operators and generate new state *State 3*. We can now write

PLAN_QUEUE = US (b, a), ST(b, d), PU(a)

State 3 of the problem can be written as

State 3: $T(c) \Lambda H(a) \Lambda T(d) \Lambda O(b, d) \Lambda C(c) \Lambda C(b)$

The transition from State 2 to State 3 is shown in Fig. 6.13.

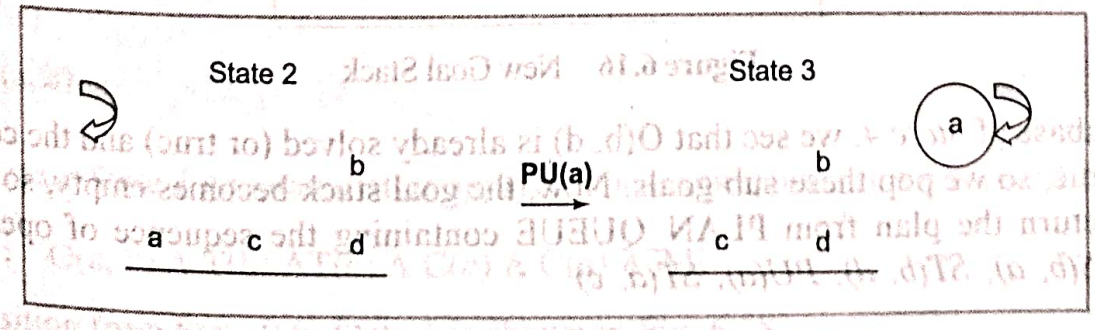


Figure 6.13 Transition from State 2 to State 3

The goal stack thus obtained as a result of this change is shown in Fig. 6.14.

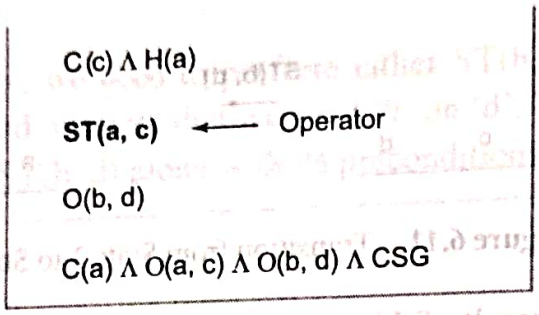


Figure 6.14 Changed Goal Stack

Further, C(c) and H(a) both are true in *State 3*, we pop these and since all preconditions of ST(a, c) are met, we pop the operator ST(a, c) and add to the solution queue. Thus, we can write

PLAN_QUEUE = US (b, a), S(b, d), PU(a), ST(a, c)

State 4 of the problem can be written as

State 4 *T(c) Λ O(a, c) Λ T(d) Λ O(b, d) Λ C(a) Λ C(b) Λ AE*

The transition from State 3 to State 4 is shown in Fig. 6.15

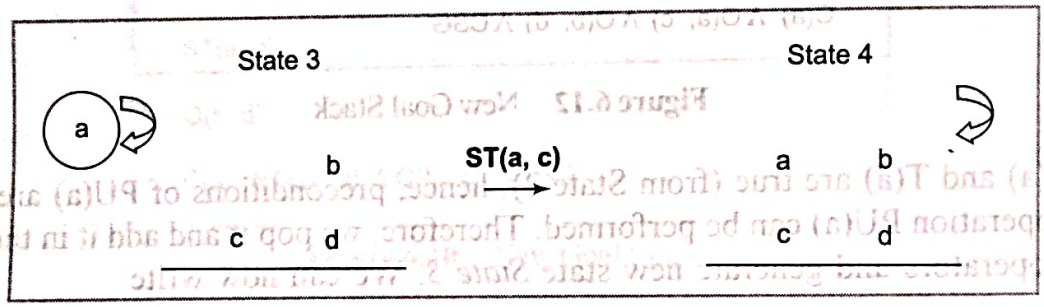


Figure 6.15 Transition from State 3 to State 4

The goal stack obtained as a result of this change is shown in Fig. 6.16.

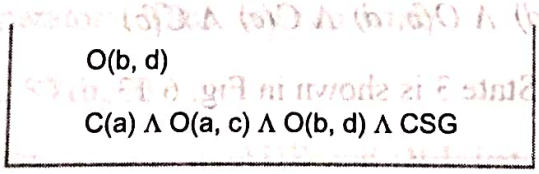


Figure 6.16 New Goal Stack

From the database of *State 4*, we see that O(b, d) is already solved (or true) and the conjoined sub goal is also true, so we pop these sub goals. Now, the goal stack becomes empty, so the problem solver will return the plan from PLAN_QUEUE containing the sequence of operators to be applied as *US(b, a), ST(b, d), PU(a), ST(a, c)*

This plan is generated offline before execution and given to robot to change the start state to goal state. It should be noted that heuristic information can be applied to guide the search process by selecting more appropriate operator than other. Information regarding some interaction among the sub goals could help in producing an overall good solution.

Although the goal stack method appears to be straightforward for simple problems, it may not give good solutions for more difficult problems, such as the problem of Sussman anomaly, and thus is not considered to be a very good method. Let us consider Sussman anomaly problem and solve it using the goal stack method to show the inefficiency of this method. For the sake of simplicity, we will show the major steps to illustrate the fact that this method obtains redundant solutions.

Sussman Anomaly Problem

To begin with, the start state and goal state of Sussman anomaly problem may be written as follows and shown in Fig. 6.17 .

Start State (State 0) O(c, a) Λ T(a) Λ T(b) Λ C(c) Λ C(b) Λ AE

Goal State O(a, b) Λ O(b, c) Λ T(c) Λ C(a) Λ AE

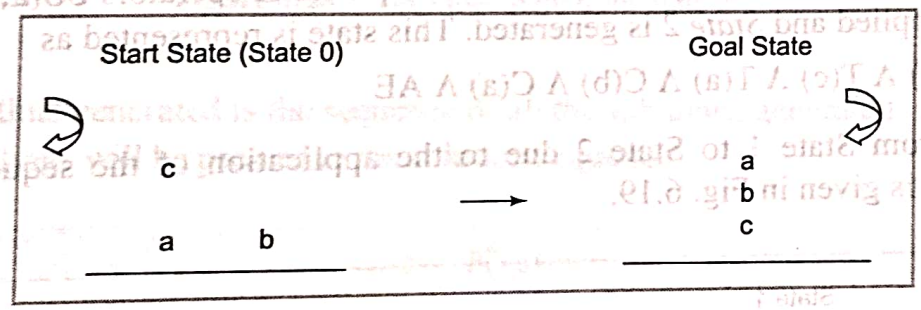


Figure 6.17 Start and Goal States of Sussman Anomaly Problem

For solving sub goal O(a, b), we have to apply the following operators obtained using the goal stack method:

- **US(c, a)**
- **PD(c)**
- **PU(a)**
- **ST(a, b)**

The new state *State 1* generated after applying these operations is represented as

State 1: O(a, b) Λ T(b) ΛT(c) Λ C(c) Λ C(a) Λ AE

The transition from State 0 to State 1 is shown in Fig. 6.18.

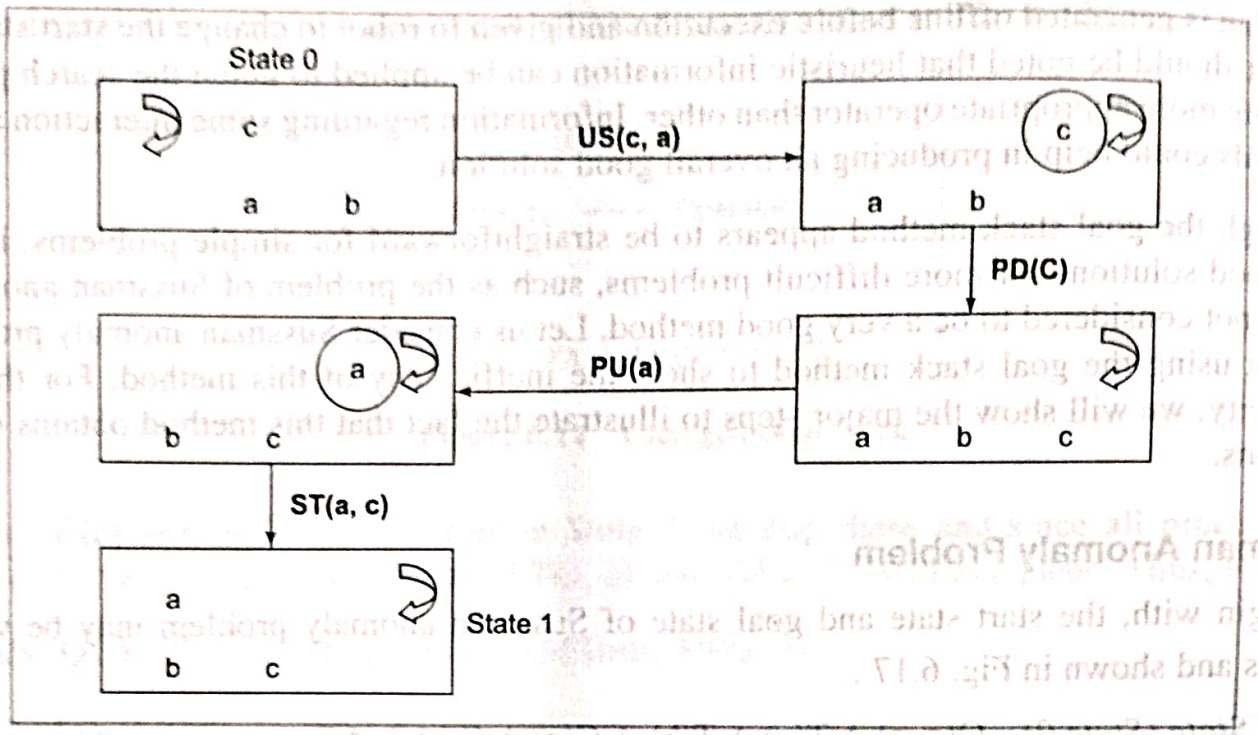


Figure 6.18 Transitions from State 0 to State 1

Now, our aim is to satisfy the sub goal O(b, c). The sequence of operators **US(a, b), PD(a), PU(b),** and **ST(b, c)** is applied and *State 2* is generated. This state is represented as

State 2 O(b, c) Λ T(c) Λ T(a) Λ C(b) Λ C(a) Λ AE

The transition from State 1 to State 2 due to the application of the sequence of operators mentioned above is given in Fig. 6.19.

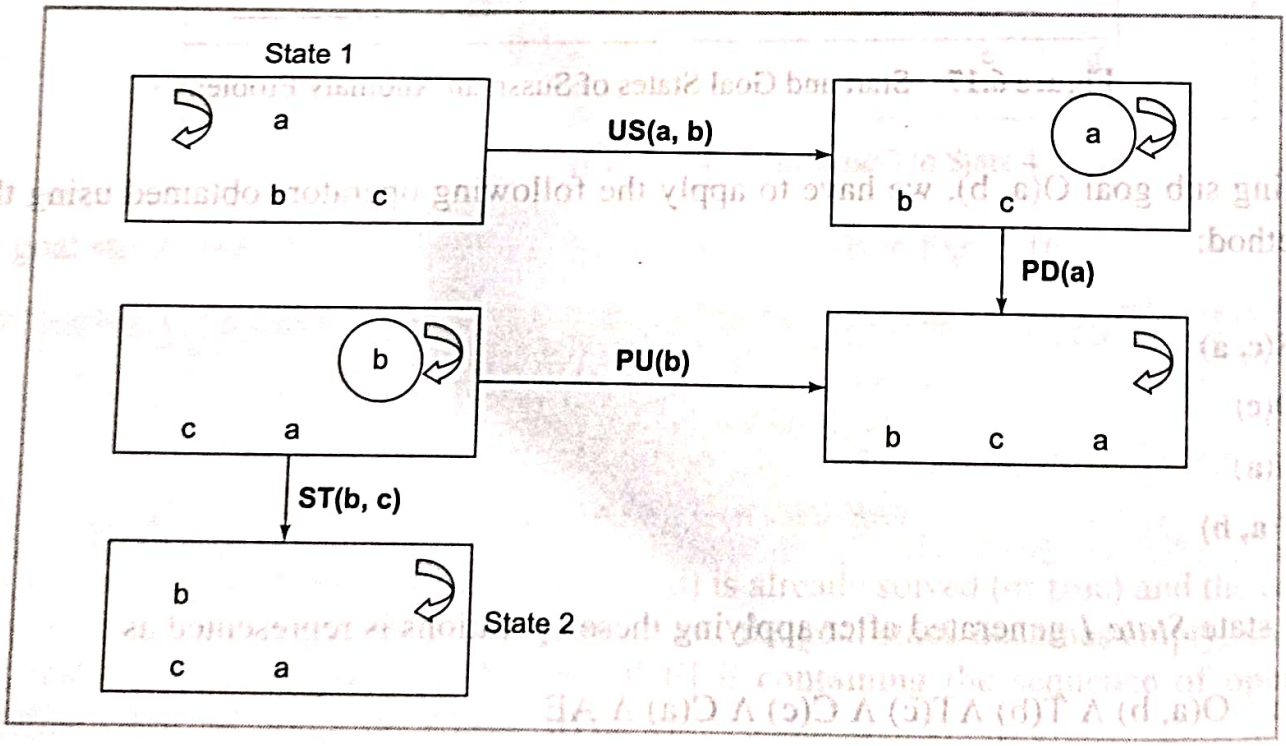


Figure 6.19 Transitions from State 1 to State 2

Finally, we need to satisfy the conjoined goal O(a, b) Λ O(b, c) Λ T(c) Λ C(a) Λ AE. We notice that while satisfying O(b, c), we have undone the already solved sub goal O(a, b). In order to solve O(a, b) again, we apply the operations **PU(a)** and **ST(a, b)**. The conjoined goal is checked again and it is found to be satisfied now. We obtain the goal state, and therefore, can now collect the total plan. The transition from State 2 to the goal state is shown in Fig. 6.20.

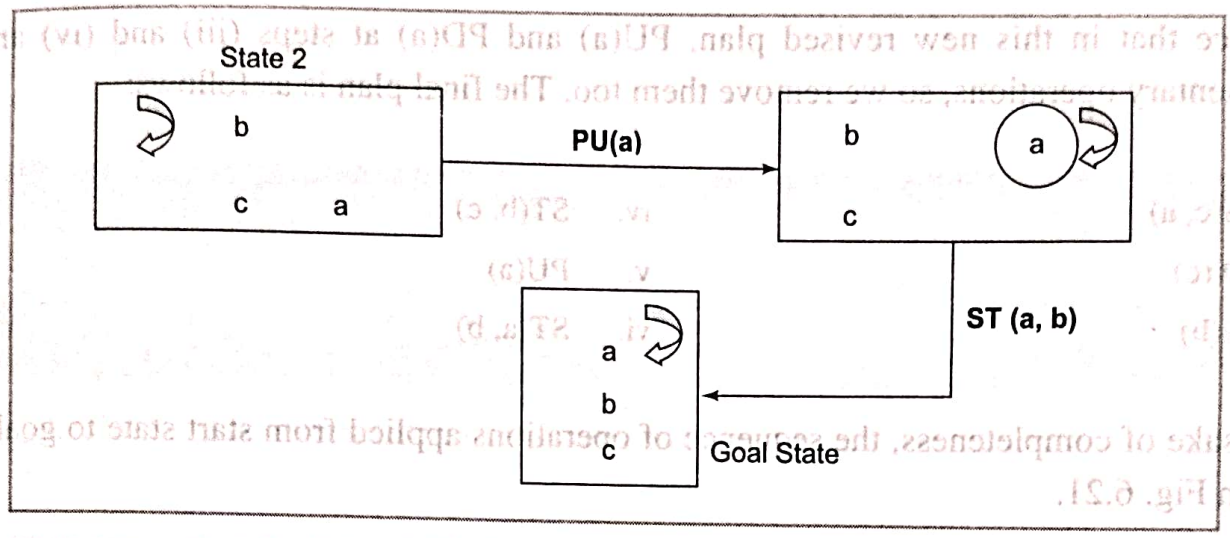


Figure 6.20 The Goal State from State 2

The complete plan thus generated is the sequence of all the sub plans generated above. Therefore, the following operations will be present in the solution sequence:

i. US(c, a)
ii. PD (c)
iii. PU(a)
iv. ST(a, b)
v. US(a, b)
vi. PD(a)
vii. PU(b)
viii. ST(b, c)
ix. PU(a)
x. ST(a, b)

Although this plan eventually achieves the desired goal, it is not considered to be efficient because of the presence of a number of redundant steps, such as stacking and unstacking of the same blocks, one performed immediately after the other. We can get an efficient plan from this plan simply by repairing it. Repairing is done by looking at those steps where operations are done and undone immediately, such as ST(X, Y) and US(X, Y) or PU(X) and PD(X). In the above plan, we notice that stacking and unstacking are done at steps (iv) and (v). By removing these complimentary sub goals, we obtain the new plan as follows:

i. US(c, a)
ii. PD (c)
iii. PU(a)
iv. PD(a)
v. PU(b)
vi. ST(b, c)
vii. PU(a)
viii. ST(a, b)

We notice that in this new revised plan, PU(a) and PD(a) at steps (iii) and (iv) are again complimentary operations, so we remove them too. The final plan is as follows:

i. US(c, a)
ii. PD (c)
iii. PU(b)
iv. ST(b, c)
v. PU(a)
vi. ST(a, b)

For the sake of completeness, the sequence of operations applied from start state to goal state is shown in Fig. 6.21.

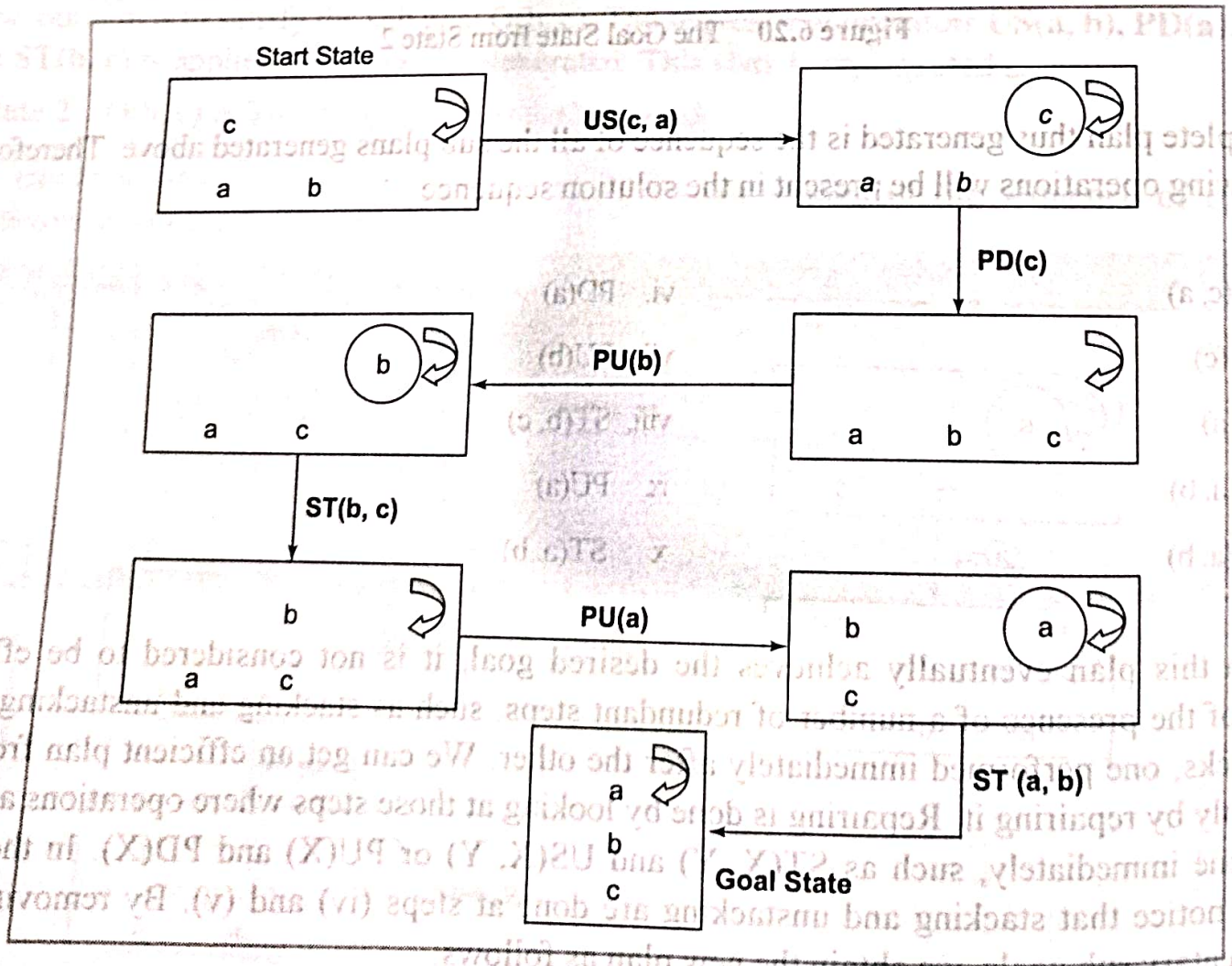
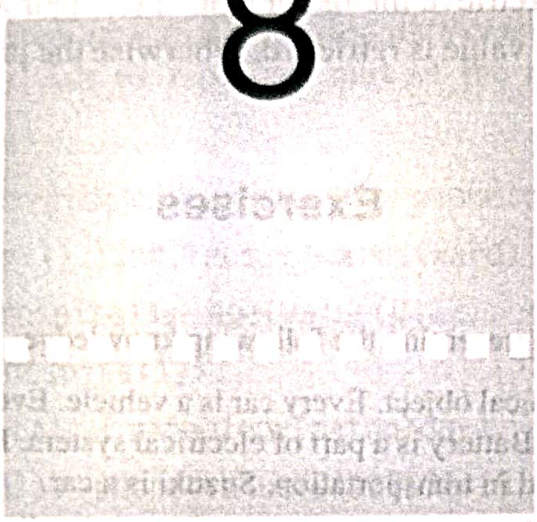


Figure 6.21 Complete Sequence of Operations for Sussman Anomaly

8



Expert System and Applications

8.1 Introduction

One of the goals of AI is to understand the concept of intelligence and develop intelligent computer programs. An example of a computer program that exhibits intelligent behaviour is an *expert system* (ES). Expert systems are meant to solve real-world problems which require specialized human expertise and provide expert quality advice, diagnoses, and recommendations. An ES is basically a software program or system that tries to perform tasks similar to human experts in a specific domain of the problem. It incorporates the concepts and methods of symbolic inference, reasoning, and the use of knowledge for making these inferences. Expert systems may also be referred to as *knowledge-based expert systems*. Expert systems represent their knowledge and expertise as data and rules within the computer system; these rules and data can be called upon whenever needed to solve problems. It should be noted that the term *expert systems* is often reserved for programs whose knowledge base contains the knowledge provided by human experts in contrast to knowledge gathered from textbooks or non-experts. The earliest examples of ES are briefly mentioned below:

- The system called MYCIN was developed using the expertise of best diagnosticians of bacterial infections whose performance was found to be better than the average clinician.
- In another real-world case, at a chemical refinery, a knowledge engineer was assigned to produce an ES to reproduce the expertise of an experienced retired employee to save the company from incurring the loss of the valued knowledge asset that the employee possessed.

An important characteristic of an ES is that the sequence of steps taken to reach a conclusion is dynamically synthesized with each new case. The sequence is not explicitly programmed during the development of the system. Expert systems can process multiple values for any problem parameter. This causes them to pursue more than one line of reasoning hence leading to results of incomplete (not fully determined) reasoning being presented. Problem solving by these systems is accomplished by applying specific knowledge rather than specific technique. This is the key idea in ES technology. It reflects the belief that human experts do not process their knowledge differently from others, but they do possess different knowledge. Therefore, the power of an ES lies in its store of knowledge regarding the problem domain; the more knowledge a system is provided, the more competent it becomes. Expert systems may or may not possess learning components; however, once they are fully developed, their performance is evaluated by subjecting them to real-world problem-solving situation (Luger & Stubblefield, 1993).

In this chapter, we will describe some important concepts pertaining to expert systems such as expert system architecture, its various components, implementation of rule-based expert system in Prolog and various expert system shells and tools available.

8.2 Phases in Building Expert Systems

Building an ES initially requires extracting the relevant knowledge from a human domain expert; this knowledge is often based on useful thumb rules and experience rather than absolute certainties. Usually experts find it difficult to express concretely the knowledge and rules used by them while solving a problem. This is because their knowledge is almost subconscious or appears so obvious that they do not bother mentioning it. After extracting knowledge from domain experts, the next step is to represent this knowledge in the system. Representation of knowledge in a computer is not straight forward and requires special expertise. A knowledge engineer handles the responsibility of extracting this knowledge and building the ES's *knowledge base*. This process of gathering knowledge from a domain expert and codifying it according to the formalism is called *knowledge engineering*. This phase is known as *knowledge acquisition*, which is a big area of research. A wide variety of techniques have been developed for this purpose. Generally, an initial prototype based on the information extracted by interviewing the expert is developed. This prototype is then iteratively refined on the basis of the feedback received from the experts and potential users of the ES. Refinement of a system is possible only if the system is scalable and modifiable and does not require rewriting of major code. The developed system should be able to explain its reasoning to its users and answer questions about the solution process. Moreover, updating the system should just involve adding or deleting localized regions of knowledge [Waterman 1986].

A simple ES primarily consists of a knowledge base and an inference engine, while features such as reasoning with uncertainty and explanation of the line of reasoning enhance the capabilities of ES. Since an ES uses uncertain or heuristic knowledge just like humans, its credibility is often questionable. In real-life situations whenever we obtain an answer to a problem which seems questionable, we want to know the rationale behind it, and we believe the answer only if the

rationale seems plausible to us. Similar is the case with expert systems; most expert systems have the ability to answer questions of the form 'Why the answer X?' explanations can be generated by tracing the line of reasoning used by the inference engine.

To be more precise, the different interdependent and overlapping phases involved in building an ES may be categorized as follows:

- **Identification Phase** In this phase, the knowledge engineer determines important features of the problem with the help of the human domain expert. The parameters that are determined in this phase include the type and scope of the problem, the kind of resources required, and the goal and objective of the ES.
- **Conceptualization Phase** In this phase, knowledge engineer and domain expert decide the concepts, relations, and control mechanism needed to describe the problem-solving method. At this stage, the issue of granularity is also addressed, which refers to the level of details required in the knowledge.
- **Formalization Phase** This phase involves expressing the key concepts and relations in some framework supported by ES building tools. Formalized knowledge consists of data structures, inference rules, control strategies, and languages required for implementation.
- **Implementation Phase** During this phase, formalized knowledge is converted to a working computer program, initially called *prototype* of the whole system.
- **Testing Phase** This phase involves evaluating the performance and utility of prototype system and revising the system, if required. The domain expert evaluates the prototype system and provides feedback, which helps the knowledge engineer to revise it.

8.2.1 Knowledge Engineering

The whole process of building an ES is often referred to as knowledge engineering. It typically involves a special form of interaction between ES builder, or the knowledge engineer, one or more domain experts, and potential users. Although there are different ways and methods of knowledge engineering, the basic approach remains the same. The tasks and responsibilities of a knowledge engineer involve the following:

- Ensuring that the computer has all the knowledge needed to solve a problem.
- Choosing one or more forms to represent the required knowledge.
- Ensuring that the computer can use the knowledge efficiently by selecting some of the *reasoning methods.*

Figure 8.1 shows the interaction between the knowledge engineer and the domain expert to produce an ES.

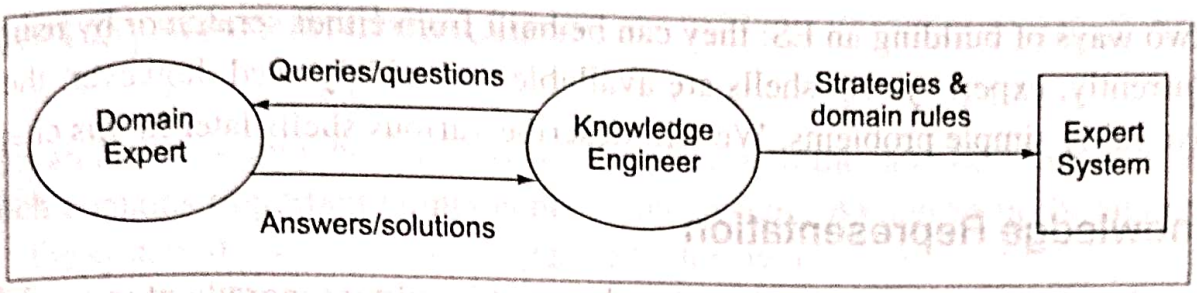


Figure 8.1 Interaction between Knowledge Engineer and Domain Expert for Creating an ES

The main role of the knowledge engineer begins only once the problem of some domain for developing an ES is decided. The job of the knowledge engineer involves close collaboration with the domain expert(s) and the end user(s). The knowledge engineer may or may not have any knowledge of the application domain initially; however, he/she must become familiar with the problem domain by reading introductory texts or literature and talking to the domain expert(s). The next step of the process involves a more systematic interviewing of the expert. The knowledge engineer will then extract general rules from the discussion and interview held with expert(s) and get them checked by the expert(s) for correctness. The engineer then translates the knowledge into a computer-usable language and designs an inference engine, which is a reasoning structure that uses the knowledge appropriately. He/she also determines the mechanism to integrate the use of uncertain knowledge in the reasoning process, and should know the kinds of explanation that may be useful to the end user.

The domain knowledge, consisting of both formal, textbook knowledge and experiential knowledge (obtained by the expert' experiences), is entered into the program piece by piece. In the initial stages, the knowledge engineer may encounter a number of problems such as the inference engine may not be right, the form of knowledge representation may not be appropriate for the kind of knowledge needed for the task, or the expert may find the pieces of knowledge incorrect. As all these are discovered and modified, the ES gradually gains competence.

The development of ES would remain incomplete if it did not involve close collaboration with end users. The basic development cycle should include the development of an initial prototype and iterative testing and modification of that prototype by both experts (for checking the validity of the rules) and users (for checking the performance of the system and explanations for the answers). In order to develop the initial prototype, the knowledge engineer will have to take provisional decisions regarding appropriate knowledge representation (e.g., rules, semantic net or frames, etc.) and inference methods (e.g., forward chaining or backward chaining or both). To test these basic design decisions, the first prototype may be so designed that it only solves a small part of the overall problem. If the methods used seem to work well for that small part, then that would imply that it is worth investing effort in representing the rest of the knowledge in the same form. During the initial years of ES development era, there were unrealistic expectations about the potential benefits of these systems. But now it has been realized that building expert systems for very complicated problems is not successful and may not fulfil expectations.

There are two ways of building an ES: they can be built from either scratch or by using ES shells or tools. Currently, expert system shells are available and widely used; however, they are often used to solve fairly simple problems. We will describe various shells later in this chapter.

8.2.2 Knowledge Representation

It must have already become clear by now that the most important ingredient in any ES is knowledge. The power of expert systems resides in the specific, high-quality knowledge they contain regarding problem domains. Therefore, the heart of ES is the powerful corpus of knowledge that is accumulated during the system-building phase; accumulation and codification of knowledge is one of the most important aspects of ES. Expert knowledge of the problem domain is organized in such a way that this knowledge is separated from other knowledge possessed by the system such as knowledge about user's interaction, general knowledge about how to solve a problem, etc. The collection of domain knowledge is called *knowledge base*, while the general problem-solving knowledge may be called inference engine, user interface, etc. The most common knowledge representation scheme for expert systems consists of *production rules*, or simply *rules*; they are of the form *if–then*, where the *if* part contains a set of conditions in some logical combination. The piece of knowledge represented by the production rule is relevant to the line of reasoning being developed when *if* part of the rule is satisfied; consequently, the *then* part can be concluded. Expert systems in which knowledge is represented in the form of rules are called *rule-based systems*. The rules may have certain conclusions or may have some degree of uncertainty; statistical techniques (such as probability) are used to handle such rules. The concept of certainty has been discussed separately in Chapter 9. Rule-based systems are easily modifiable and helpful in giving traces of the system's reasoning. These traces can be used in providing explanations of how the system solved the problem.

Another widely used representation in ES is called the *unit* (also known as *frame, semantic net,* etc.), which is based upon a more passive view of knowledge. The unit is an assemblage of associated symbolic knowledge about an entity to be represented. Typically, a unit consists of a list of properties of an entity and associated values for those properties. These have already been described in Chapter 7. Since every task domain consists of many entities that occur in various relations, properties can also be used to specify relations. The values of these properties represent the names of other units that are linked according to relations. One unit can also represent knowledge that is a *special case* of another unit, or some units can be *parts of* another unit.

AI researchers continue to explore and add to the current list of knowledge representation techniques and reasoning methods. We now understand that knowledge is an integral part of ES. However, the current knowledge acquisition methods are slow and tedious. Thus, the future of ES depends on developing better methods of knowledge acquisition and in codifying and representing a large knowledge infrastructure.

8.3 Expert System Architecture

Expert system architecture may be effectively described with the help of a diagram as given in Fig. 8.2, which contains important components of the system. As shown in the figure, the user interacts with the system through a *user interface* which may use menus, natural language, or any other style of interaction. Then, an *inference engine* is used to reason with the *expert knowledge* as well as the data specific to the problem being solved. *Case-specific data* includes both data provided by the user and partial conclusions along with certainty measures based on this data. In a simple forward-chaining rule-based system, case-specific data will be included in *working memory*. Generally, all expert systems possess an *explanation subsystem*, which allows the program to explain its reasoning to the user. Some systems also have a *knowledge acquisition module* that helps the expert or knowledge engineer to easily update and check the knowledge base. Each of these components is briefly described in the following subsection.

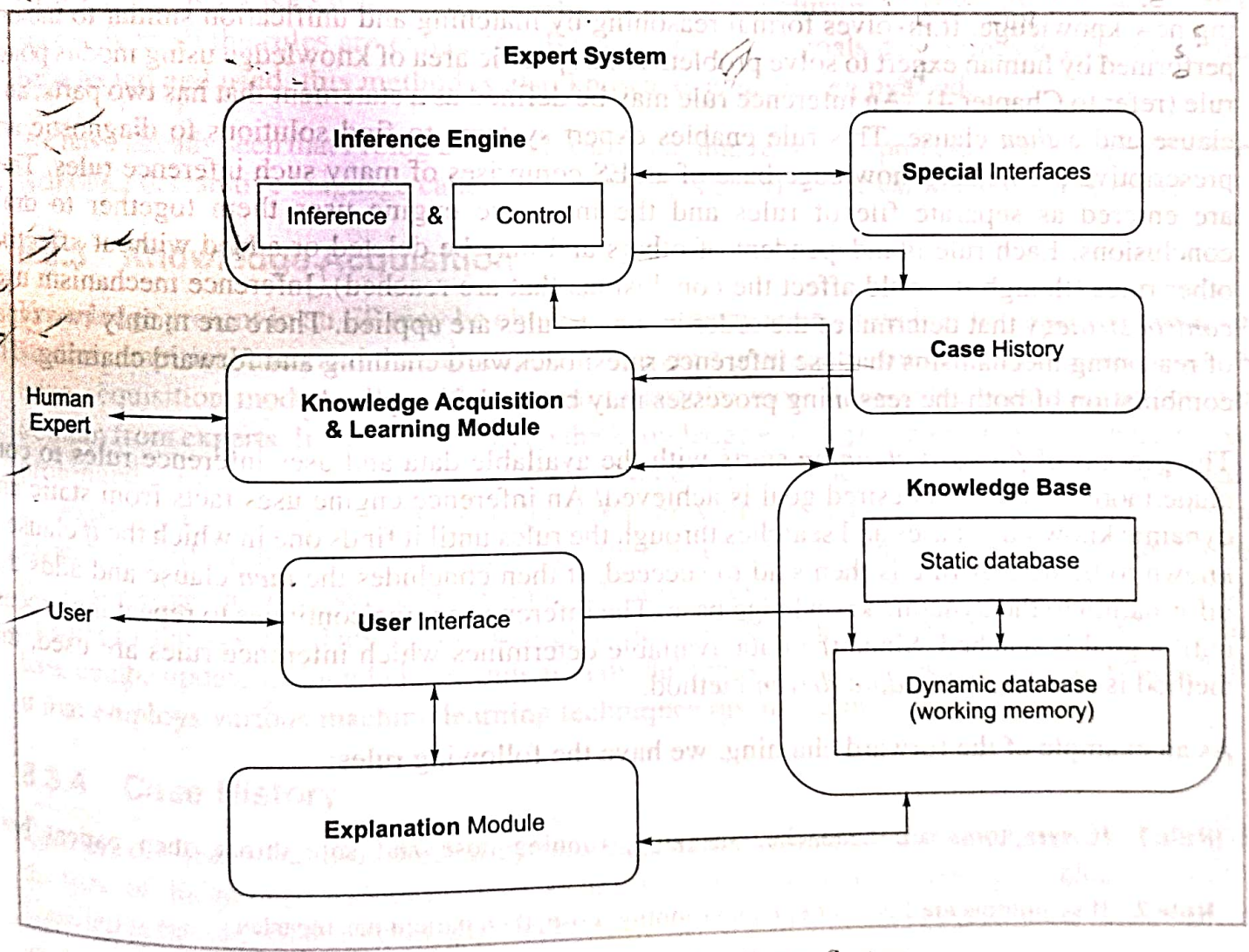


Figure 8.2 Architecture of an Expert System

8.3.1 Knowledge Base

Knowledge base of an ES consists of knowledge regarding problem domain in the form of static and dynamic databases. *Static knowledge* consists of rules and facts, or any other form of knowledge representation which may be compiled as a part of the system and does not change during the execution of the system. On the other hand, *dynamic knowledge* consists of facts related to a particular consultation of the system collected by asking various questions to the user who is consulting the ES. At the beginning of the consultation, the dynamic knowledge base (often called working memory) is empty. As the consultation progresses, dynamic knowledge base (in the form of facts only) grows and is used in decision making along with static knowledge. Working memory is deleted at the end of consultation of the system.

8.3.2 Inference Engine

An inference engine developed for an ES consists of inference mechanism as well as control strategy. The term *inference* refers to the process of searching through knowledge base and deriving new knowledge. It involves formal reasoning by matching and unification similar to the one performed by human expert to solve problems in a specific area of knowledge using modus ponen rule (refer to Chapter 4). An inference rule may be defined as a statement that has two parts, an *if* clause and a *then* clause. This rule enables expert systems to find solutions to diagnostic and prescriptive problems. Knowledge base of an ES comprises of many such inference rules. They are entered as separate file of rules and the inference engine uses them together to draw conclusions. Each rule is independent of others and may be deleted or added without affecting other rules (though it would affect the conclusions that are reached). Inference mechanism uses *control strategy* that determines the order in which rules are applied. There are mainly two types of reasoning mechanisms that use inference rules: backward chaining and forward chaining. The combination of both the reasoning processes may be used, if required.

The process of *forward chaining* starts with the available data and uses inference rules to conclude more data until a desired goal is achieved. An inference engine uses facts from static and dynamic knowledge bases and searches through the rules until it finds one in which the *if* clause is known to be true. A rule is then said to succeed. It then concludes the *then* clause and adds this information to the dynamic knowledge base. The inference engine continues to repeat the process until a goal is reached. Since the data available determines which inference rules are used, this method is also known as *data driven* method.

As an example of the forward chaining, we have the following rules:

Rule 1 If symptoms are headache, sneezing, running_nose and sore_throat, then patient has cold.

Rule 2 If symptoms are fever, cough and running_nose, then patient has measles.

Facts are generated in working memory by asking questions to the user whether he has fever, running_nose, cough, etc.

Thus, in forward chaining, we start with the facts given by the user and try to find an appropriate rule whose *if* part is satisfied and subsequently the *then* part is concluded.

The other important inference mechanism that we are familiar with is backward chaining. *Backward chaining* starts with a list of goals and works backwards to see if there is data which will allow it to conclude any of these goals. An inference engine using backward chaining would search the inference rules until it finds one whose *then* part matches a desired goal. If the *if* part of that inference rule is not known to be true, then it is added to the list of goals.

Consider the same example discussed above. In order to satisfy a goal called *cold*, the inference engine will select a rule with conclusion as *cold* and will try to find the facts in the *if* part of the rule whether the user has headache, sneezing, running nose, and sore throat. If *yes*, then *cold* is established otherwise it tries other rule for goal, if it exists. If we are not able to satify all the rules with the goal cold then other goals such as *measles* will be tried. Using rule 2, if the symptoms of the user are fever, running nose, and cough, then *measles* is concluded. The inference engine using backward chaining tries to prove conclusion of the rules one by one till it succeeds or all the rules are exhausted. Because the list of goals determines which rules would be selected and used, this method is also known as *goal-driven* method.

We have already seen that Prolog uses backward-chaining reasoning process (refer Chapter 5). There is another declarative language called OPS5 which is based on forward-chaining reasoning.

8.3.3 Knowledge Acquisition

Knowledge present in an ES may be obtained from many sources such as textbooks, reports, case studies, empirical data, and domain expert which are a prominent source of knowledge. A knowledge acquisition module allows the system to acquire more knowledge regarding the problem domain from experts. Interaction between the knowledge engineer and the domain expert involves prolonged series of intense systematic interviews or using a questionnaire (carefully designed to get expertise knowledge). The knowledge engineer working on a system should be able to extract expert methods, procedures, strategies, and thumb rules for solving the problem at hand. Later, the knowledge can be updated (insertion, deletion, or updation) by using knowledge acquisition module of the system. This system will give all these facilities to expert so that the knowledge base can be updated. Knowledge acquisition module may also have a learning module attached to it that employs various machine learning techniques discussed in Chapter 11.

8.3.4 Case History

Case history stores the files created by inference engine using the dynamic database (created at the time of different consultation of the system) and is used by the learning module to enrich its knowledge base. Different cases with solutions are stored in Case Base system and these cases are used for solving the problem using Case Base Reasoning (CBR). This will be discussed in detail in Chapter 11 of this book.

8.3.5 User Interfaces

User interface of an ES allows user to communicate with the system in an interactive manner and helps the system in creating working knowledge for the problem that has to be solved. Knowledge may be entered using some editor or specialized designed user interface. The function of the user interface is to present questions and information to the user and supply the responses of user to the inference engine. Any values provided by the user need to be received and interpreted by the user interface. Responses of the user may or may not be restricted to a set of possible legal answers. The user interface checks all responses to ensure that they are of the correct data type. Any responses that are restricted to a legal set of answers are compared against the legal answers. So, whenever the user enters an illegal answer, the user interface informs the user that the answer is invalid and prompts the user to provide the correct answer.

Therefore, the end-user usually sees an ES through an interactive dialogue module. We observe from the dialogue module given in Table 8.1 how the system leads the user through a set of questions, whose purpose is to determine a set of symptoms. Note that the dialogues are not pre-planned in expert systems and there is no fixed control structure. They are generated from the current information present in the system and the knowledge contained in the knowledge base. The answers to questions shown in Table 8.1 are binary in *yes* or *no* form for simplification. An advanced interface may accept answers in natural language and then it employs natural language processing techniques to extract out relevant information from the response. Responses of user may be fuzzy also. Lot of research is in progress in this area.

Table 8.1 Dialogue Module of a User Interface

System	Do you have fever?
User	Yes
System	Do you have sore throat?
User	No
System	Do you have cough?
User	Yes
System	Are you suffering from running nose?
User	Yes
System	Are you suffering from headache?
User	No

8.3.6 Explanation Module

Most of the ES has explanation facilities that enable users to query the system about why it asked some questions and how it reached some conclusion; these modules are called *How* and *Why*. The sub-module *How* tells users regarding the process through which the system has reached a particular solution, while the *Why* sub-module explains to the user, the reasoning behind arriving

at a solution to any particular problem. These questions are answered by referring to the system goals, the rules being used, and any existing problem data. An ES can generate an explanation by retracing the steps of its reasoning. The response of the ES to the question *why* may provide an insight into the underlying knowledge structure. The knowledge structure of an ES consists of a rule, a set of antecedent conditions, which, if true, allows the assertion of a consequent. To illustrate the use of explanation facilities, the Table 8.2 shows explanation for *why* and *how* questions.

Table 8.2 Explanation Module

Invoke	Why/How modules
System	Do you have running nose?
User	Yes
System	I am trying to prove that you have cold. Do you have headache?
User	Yes
System	Do you have sore throat?
User	No
System	Then you do not have cold. Do you have fever?
User	Yes
System	Now I am looking at the possibility of measles. Do you have cough?
User	Yes
System	I can infer measles using the rule "If symptoms are fever, cough, running_nose, then patient has measles". Measles is concluded.
User	**How?**
System	Since you have fever, running_nose, and cough and there is a rule "If symptoms are fever, cough, running_nose, then patient has measles". So, measles is concluded for you.

Providing explanation facility in a system involves, at least, recording the rules that are used in drawing conclusions; these records are then used to compose the explanations. Providing simple explanations as those given in the above table is not very difficult (because the answers always have a simple set form); these explanations prove to be quite useful at times. However, it is very difficult to implement a general explanation system in a traditional computer program.

8.3.7 Special Interfaces

Special interfaces may be used in ES for performing specialized activities, such as handling uncertainty in knowledge. These interfaces form a major area of expert system research which involves methods for reasoning with uncertain data and uncertain knowledge. A point to be kept in mind regarding knowledge is that it is generally incomplete and uncertain. To deal with uncertain knowledge, a *confidence factor* or a weight may be associated with a rule. The set of methods for using uncertain knowledge in combination with uncertain data in the reasoning process is called *reasoning with uncertainty*. Probability theory is the oldest method used to determine these

certainties. Another important subclass of methods for reasoning with uncertainty is called *fuzzy logic* and the systems that use them are known as *fuzzy systems*. Uncertainty and fuzzy logic have been discussed in Chapters 9 and 10, respectively.

8.4 Expert Systems versus Traditional Systems

The basic difference between an ES and a traditional system is that an ES manipulates knowledge, whereas a traditional system manipulates data. The distinction between these systems lies in the manner in which the problem-related expertise is coded into them. In traditional applications, problem expertise is encoded in program as well as in the form of data structures. On the other hand, in the ES approach, all problem-related expertise is encoded in data structures only and not in the programs. Traditional computer programs perform tasks using conventional decision-making logic, which is often embedded as a part of the code in the form of a basic algorithm containing little knowledge for solving that specific problem. Hence, if the knowledge changes, the program has to be rebuilt. However, in expert systems, small fragments of human experience are collected into a knowledge base which are used to reason through a problem. A different problem, within the domain of the knowledge base, can be solved using the same program without having to reprogram the system.

Another advantage of expert systems over traditional systems is that they allow the use of confidences or certainty factors. This is similar to human reasoning where one cannot always conclude things with 100% confidence. For example, consider the statement *If weather is humid, then it might probably rain*. The use of words such as *if*, *then*, *might*, *probably*, etc., indicate that there is some uncertainty involved in the statement. This type of reasoning can be imitated by using numeric values called *confidences* in ES. For example, if it is known that the weather is humid, it might be concluded with 0.9 confidences that it rains. Although, these numbers are similar to probabilities, they are not the same. They are meant to imitate the confidences humans use in reasoning rather than to follow the mathematical definitions used in calculating probabilities. Therefore, the ability of ES to explain the reasoning process through back-traces and to handle levels of confidence and uncertainty provides an additional feature as compared to conventional or traditional programs.

Further, conventional programs are designed to always produce correct answers, whereas expert systems are designed to behave like human experts and may sometimes produce incorrect results. In the following subsections, we will discuss some important characteristics of expert systems, their evaluations, advantages and disadvantages, and the languages that can be used for their development.

8.4.1 Characteristics of Expert Systems

Some key characteristics that every ES must possess are described as follows:

- **Expertise** An ES should exhibit expert performance, have high level of skill, and possess adequate robustness. The high-level expertise and skill of an ES aids in problem solving and makes the system cost effective.
- **Symbolic reasoning** Knowledge in an ES is represented symbolically which can be easily reformulated and reasoned.
- **Self knowledge** A system should be able to explain and examine its own reasoning.
- **Learning capability** A system should learn from its mistakes and mature as it grows. Flexibility provided by the ES helps it grow incrementally.
- **Ability to provide training** Every ES should be capable of providing training by explaining the reasoning process behind solving a particular problem using relevant knowledge.
- **Predictive modelling power** This is one of the important features of ES. The system can act as an information processing model of problem solving. It can explain how new situation led to the change, which helps users to evaluate the effect of new facts and understand their relationship to the solution.

8.4.2 Evaluation of Expert Systems

Evaluation of an ES consists of performance and utility evaluation. Performance evaluation consists of answering various questions such as the ones given below.

- Does the system make decisions that experts generally agree to?
- Are the inference rules correct and complete?
- Does the control strategy allow the system to consider items in the natural order that the expert prefers?
- Are relevant questions asked to the user in proper order (otherwise it will be an irritating process)?
- Are the explanation given by the ES adequate for describing *how* and *why* conclusions?

Utility evaluation consists of answering the following questions:

- Does the system help user in some significant way?
- Are the conclusions of the system organized and ordered in a meaningful way?
- Is the system fast enough to satisfy the user?
- Is the user interface friendly enough?

8.4.3 Advantages and Disadvantages of Expert Systems

One has to justify the need for developing an ES as it involves a lot of time and money. In order to avoid costly and embarrassing failures, one should evaluate whether a problem is suitable for an ES solution using the following guidelines:

- **Specialized knowledge problems** If there is a rare specialized knowledge involved in some special domain (say, oil exploration and medicine), then it is worth developing an ES to solve problems pertaining to the domain as human experts are scarce and unavailable. This implies that we typically need to develop ES for problems that require highly specialized expertise, which is likely to be lost due to personnel changes or retirement.
- **High payoff** An ES may be developed if the task to be performed has a very high payoff. The company may need similar expertise at a large number of different physical locations.
- **Existence of cooperative experts** For an ES to be successfully developed, it is essential that the experts are willing to help and provide their expertise. We also need potential users to be involved who will be using the system once it is fully developed.
- **Justification of cost involved in developing ES** There must be a realistic assessment of the costs and benefits involved.
- **The type of problem** The problem for which an ES is to be developed must be structured and it does not require common sense knowledge as common sense knowledge is hard to capture and represent. It is easier to deal with ES developed for highly technical fields.

If there exists a good algorithmic solution to a problem, there is no need to build an ES. The problems that cannot be easily solved using more traditional computing methods are potential problems for ES. It should be clear that only a small range of problems are appropriate for ES technology. However, given a suitable problem, expert systems can bring enormous benefits. The advantages and disadvantages of ES are listed as follows:

Advantages

- Helps in preservating scarce expertise.
- Provides consistent answers for repetitive decisions, processes, and tasks.
- Fastens the pace of human professional or semi-professional work.
- Holds and maintains significant levels of information.
- Provides improved quality of decision making.
- Domain experts are not always able to explain their logic and reasoning unlike ES.
- Encourages organizations to clarify the logic of their decision-making.
- Leads to major internal cost savings within companies.
- Causes introduction of new products.
- Never *forgets* to ask a question, unlike a human.

Disadvantages

- Unable to make creative responses as human experts would in unusual circumstances.
- Lacks common sense needed in some decision making.