Chapter 1

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

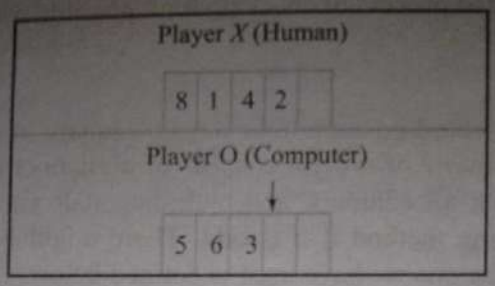
In this approach, we maintain a list of the blocks played by each player. For the sake of convenience, each block is identified by its number. The following strategy for possible win for a player is used. Obviously in our case, we are considering a computer to be a player for which the strategy is suggested 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 $D \leq 0$ 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
    - $S = 5 + 3 = 8$;

Chapter 2

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	1
3	0	3	2
5	3	0	3
3	3	3	4
7	5	1	5
2	0	1	6
5	1	0	7
3	1	3	8
5	4	0	
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 R

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:

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 operator

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.

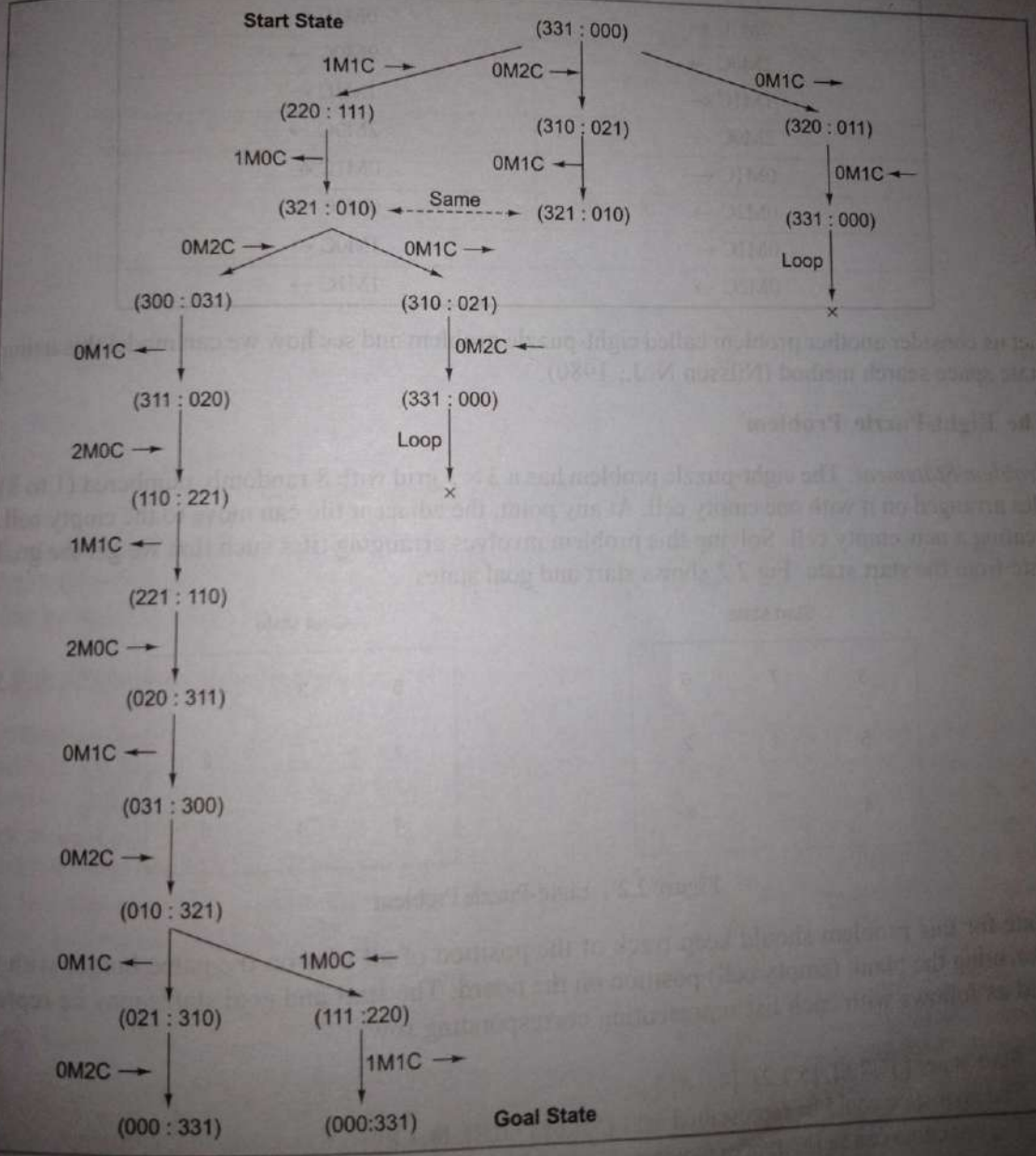


Figure 2.1 Search Space

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.

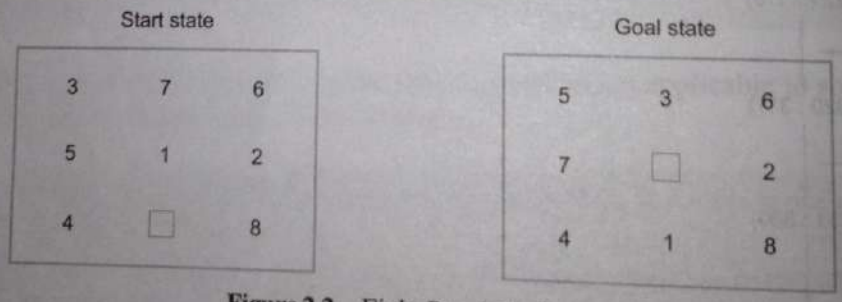


Figure 2.2 Eight-Puzzle Problem

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.

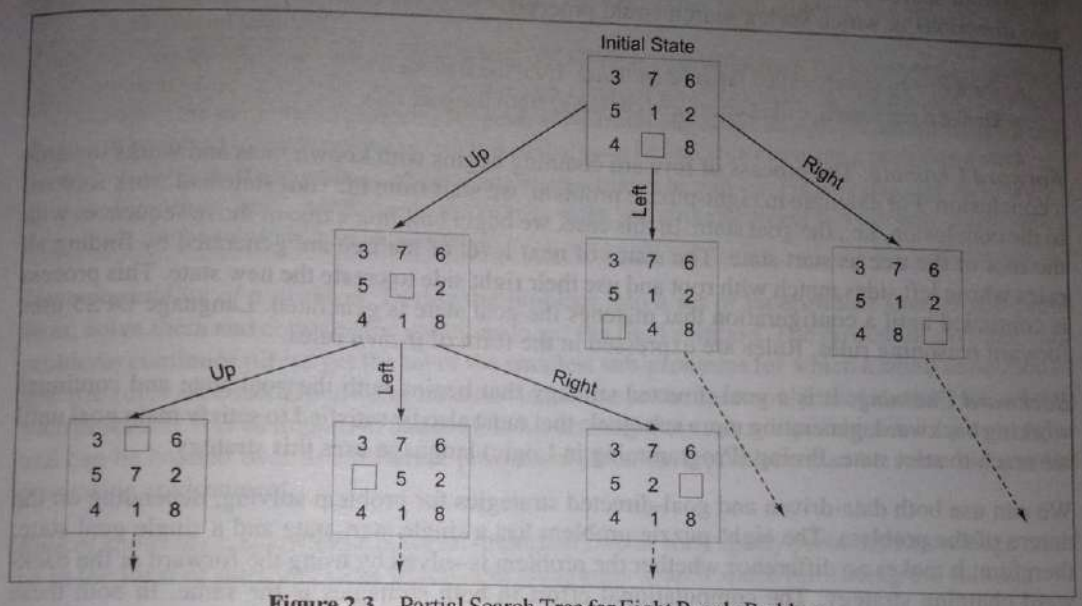


Figure 2.3 Partial Search Tree for Eight Puzzle Problem

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 ≠ ϕ 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.

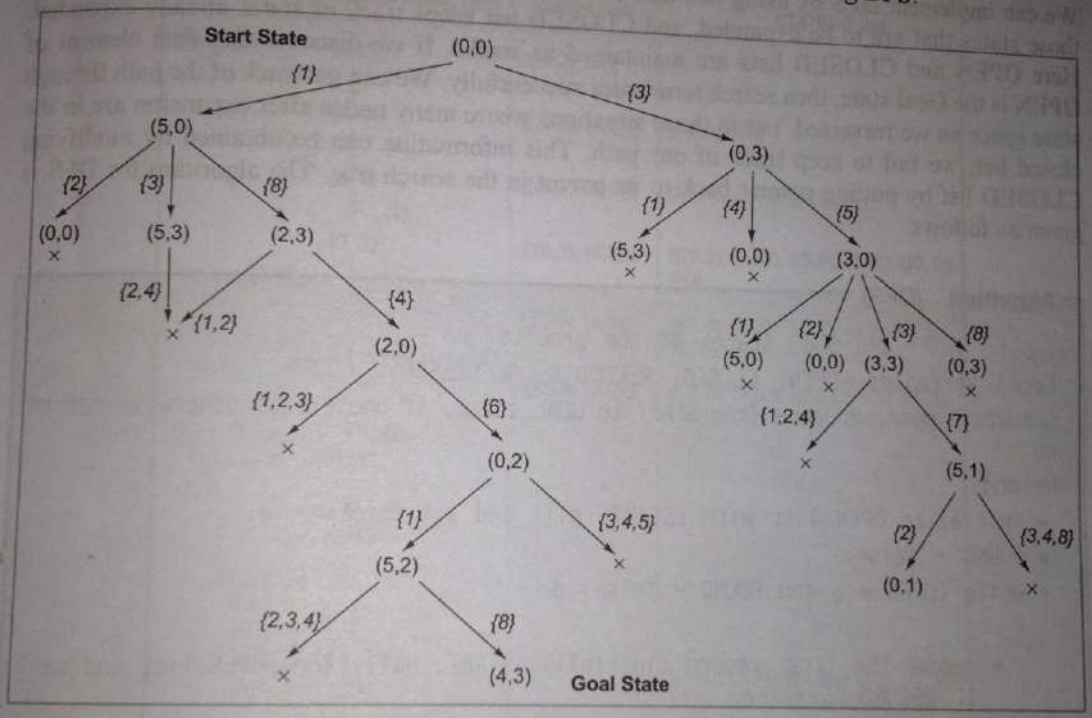


Figure 2.4 Search Tree Generation using BFS

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: $(0,0) \rightarrow (5,0) \rightarrow (2,3) \rightarrow (2,0) \rightarrow (0,2) \rightarrow (5,2) \rightarrow (4,3)$

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.

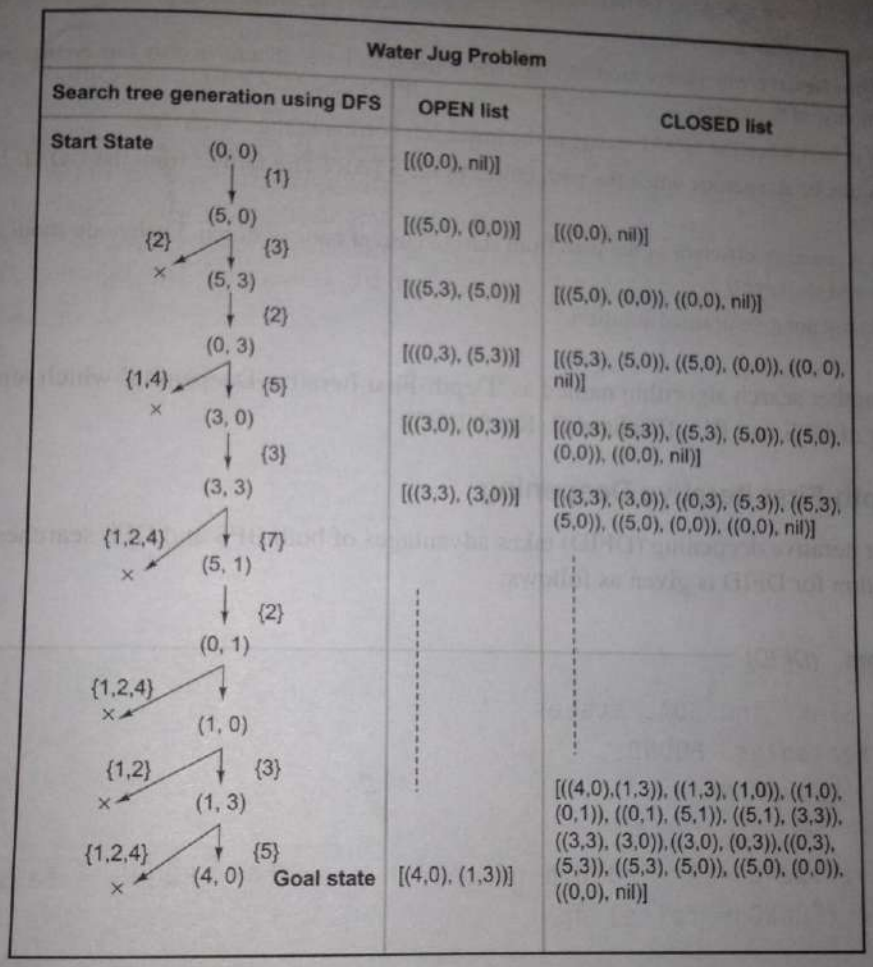


Figure 2.5 Search Tree Generation using DFS

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:

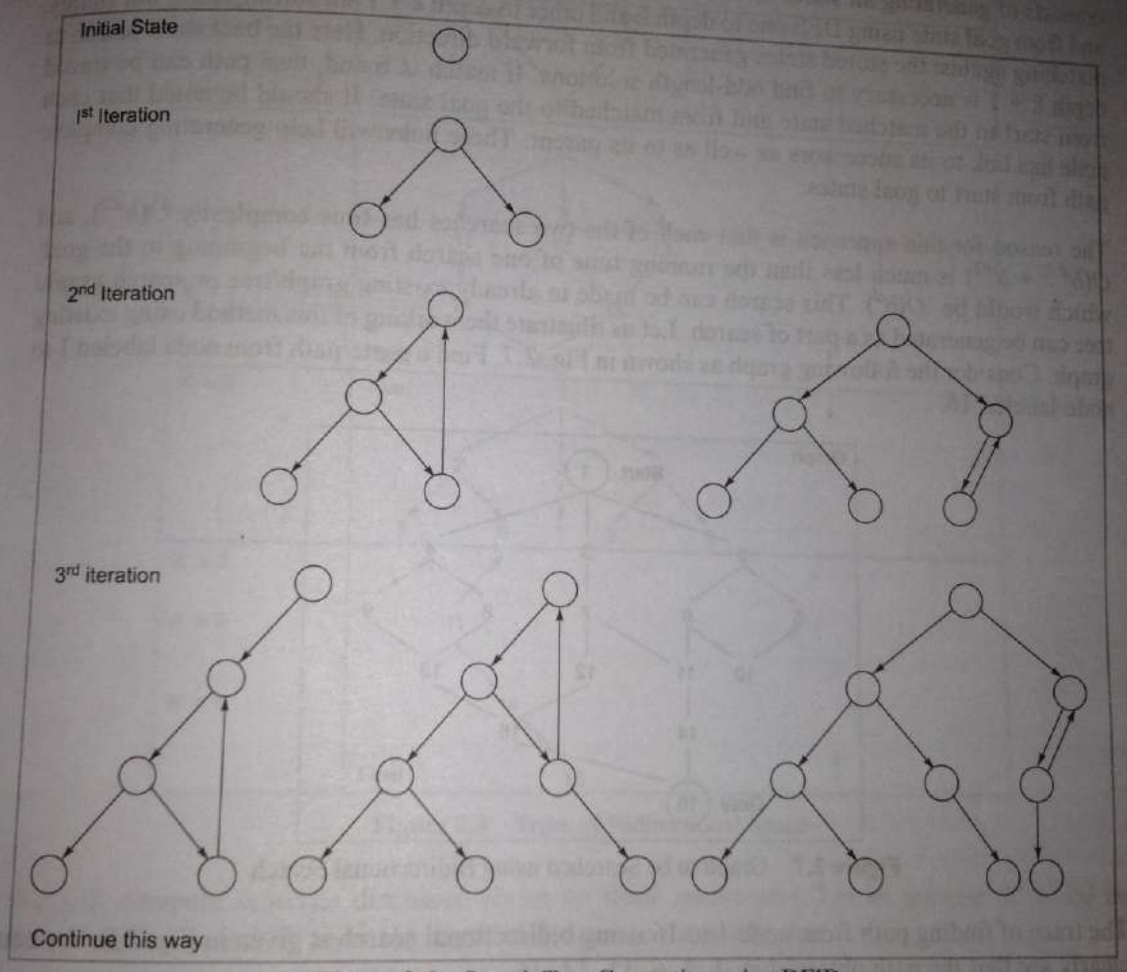


Figure 2.6 Search Tree Generation using DFID

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.

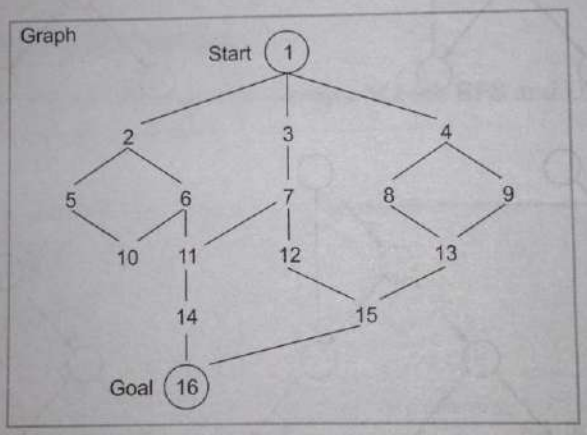


Figure 2.7 Graph to be Searched using Bidirectional Search

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.

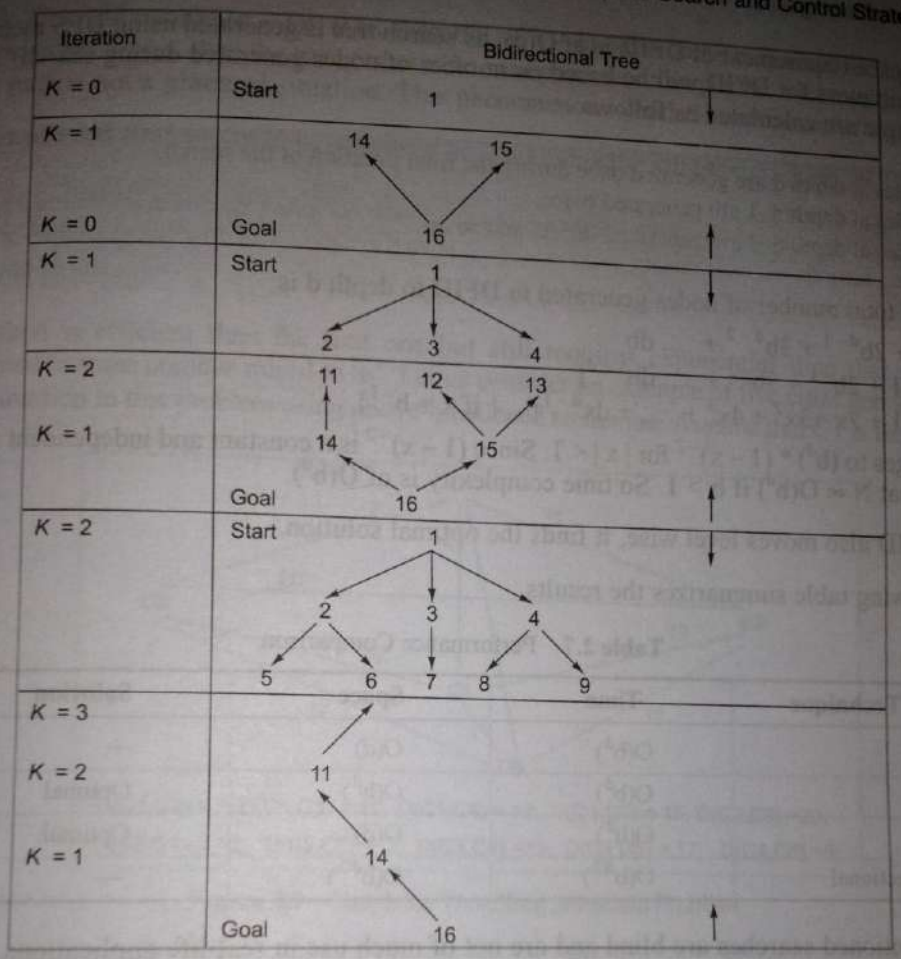


Figure 2.8 Trace of Bidirectional Space

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

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

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.

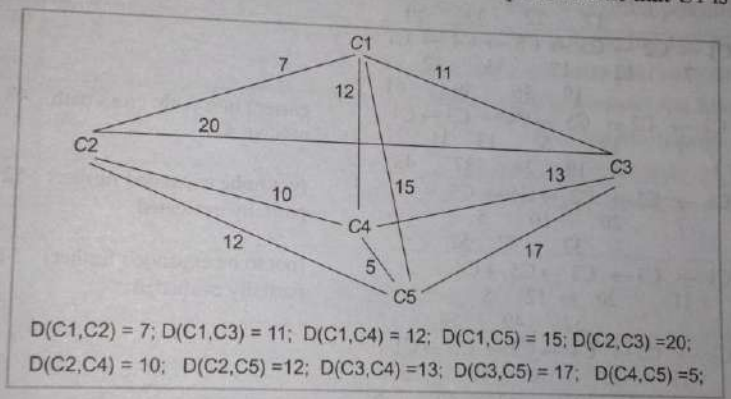


Figure 2.9 Graph for Travelling Salesman Problem

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

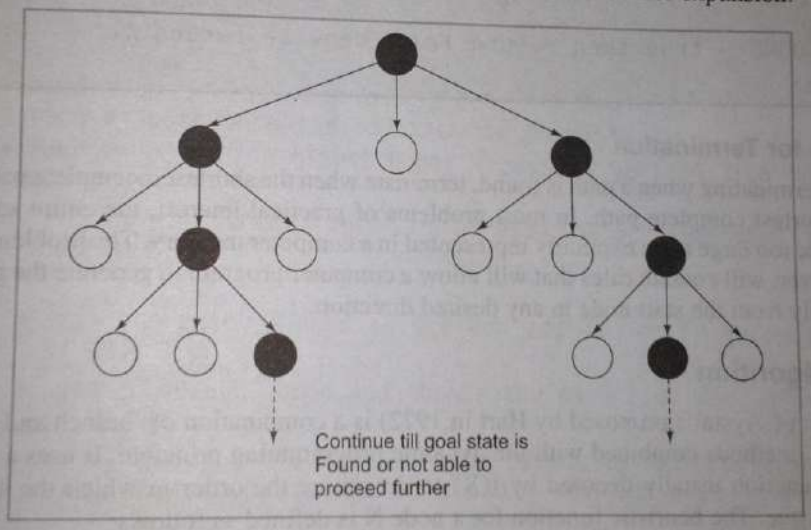
Beam search is a heuristic search algorithm in which W number of best nodes at each level is always expanded. It progresses level by level and moves downward only from the best W nodes at each level. Beam search uses breadth-first search to build its search tree. At each level of the tree, it generates 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

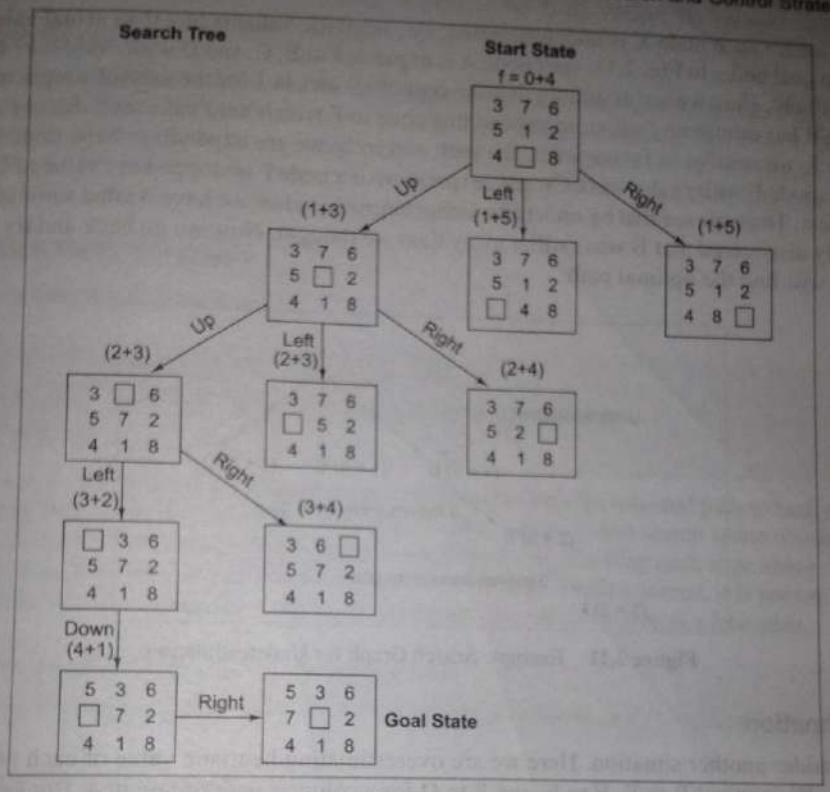


Figure 2.10 Search Tree

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.

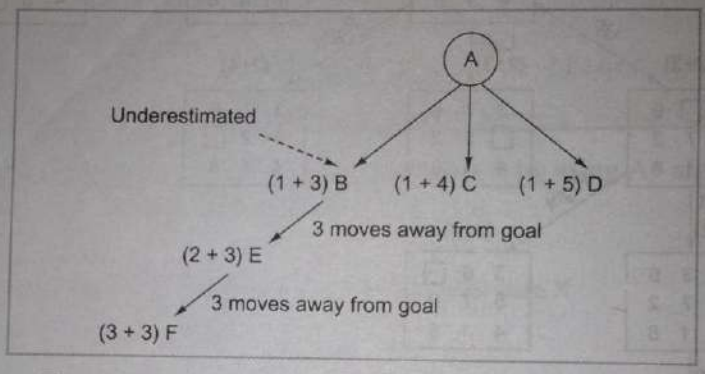


Figure 2.11 Example Search Graph for Underestimation

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.

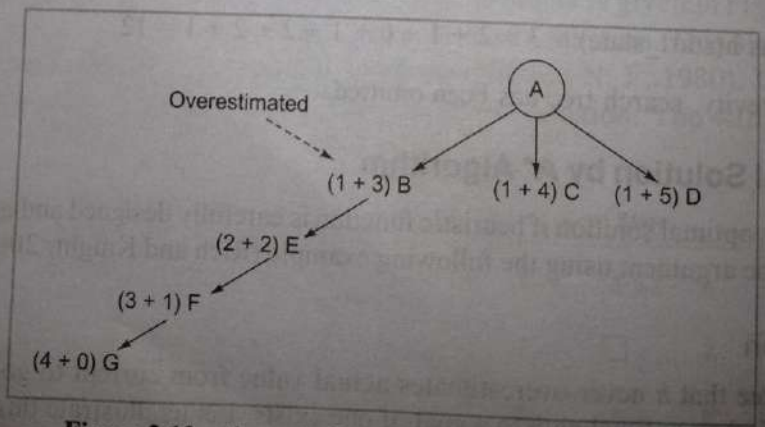


Figure 2.12 Example Search Graph for Overestimation

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.

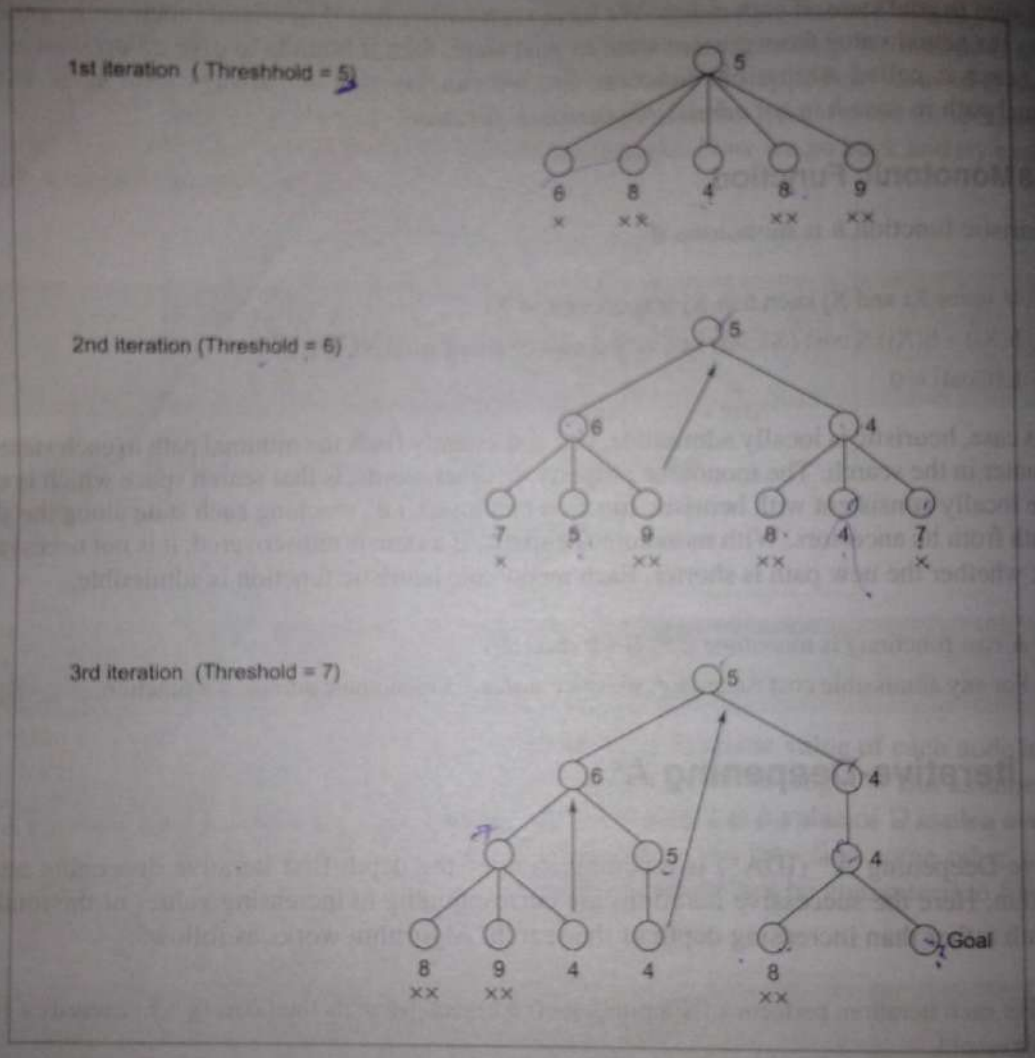


Figure 2.13 Working of IDA*

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:

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

- 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,

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

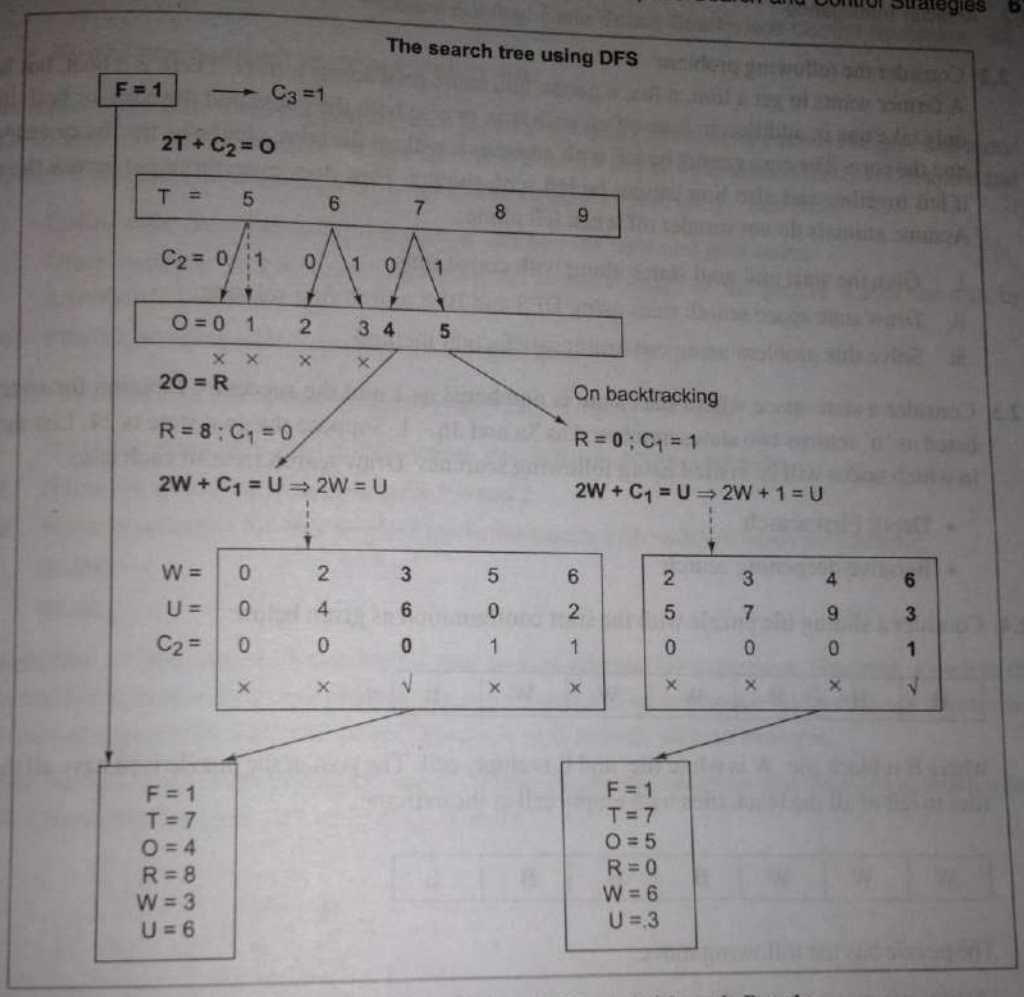


Figure 2.15 Search Tree for Crypt-Arithmetic Puzzle

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?

Chapter 3

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.

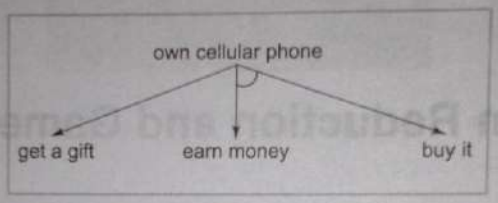


Figure 3.1 A Simple AND–OR Graph

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.

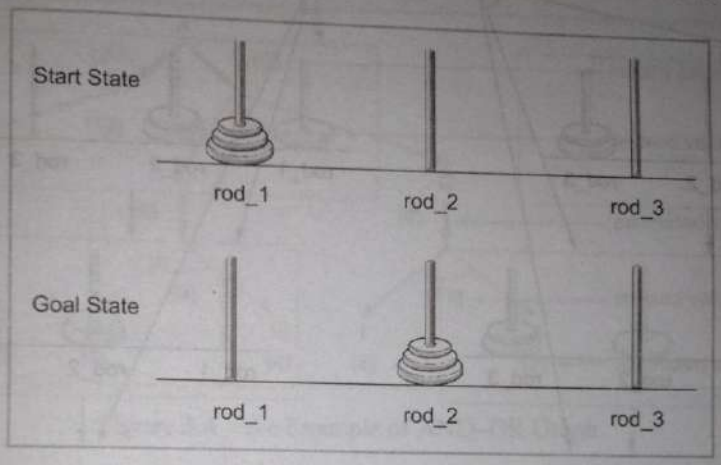


Figure 3.2 The Start and Goal States of a Three-disk *Tower of Hanoi* Problem

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.

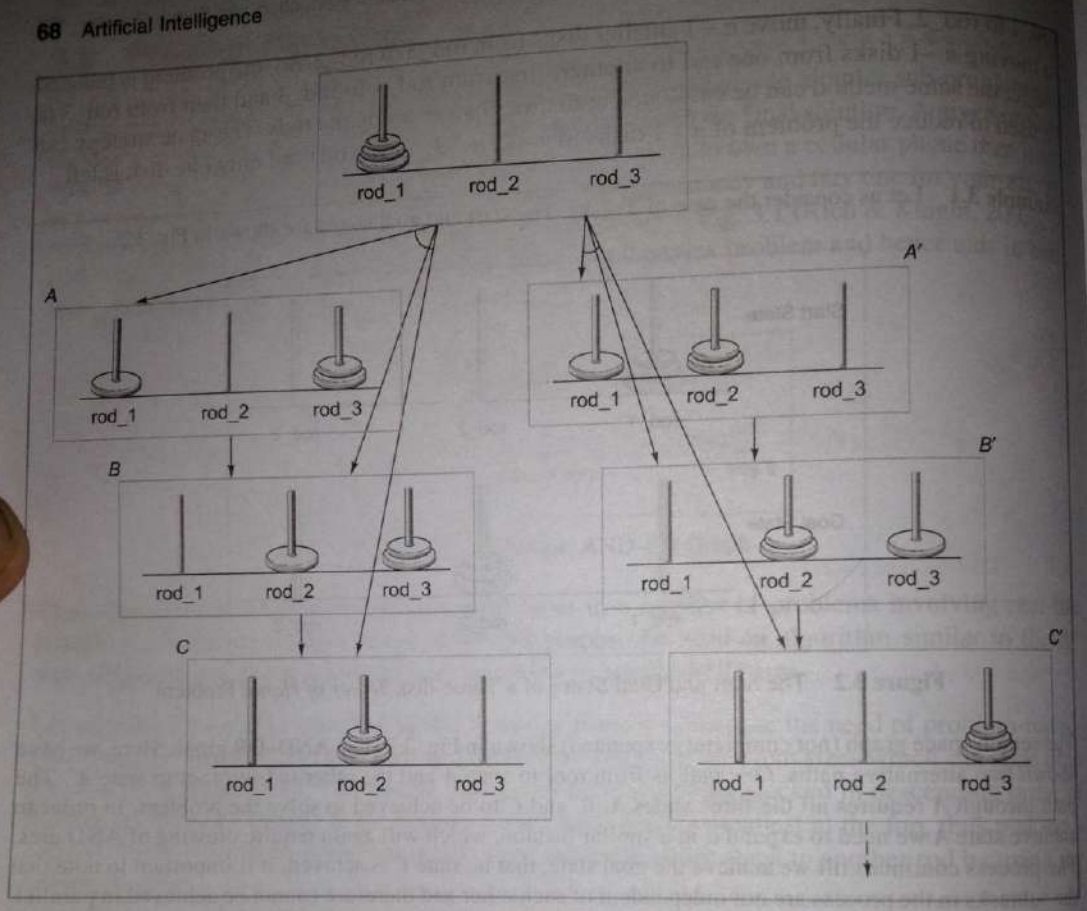


Figure 3.3 An AND–OR Graph for a Three-disk Problem

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.

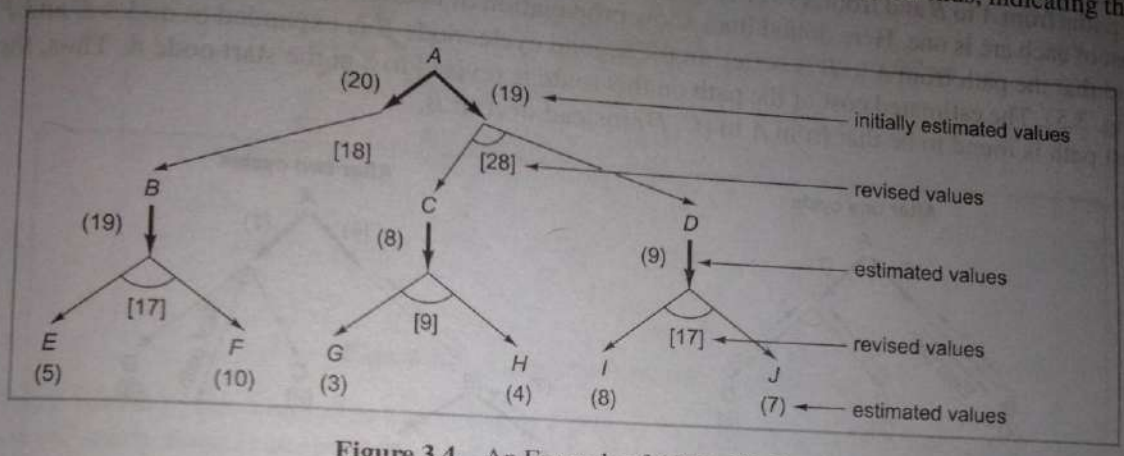


Figure 3.4 An Example of AND–OR Graph

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.

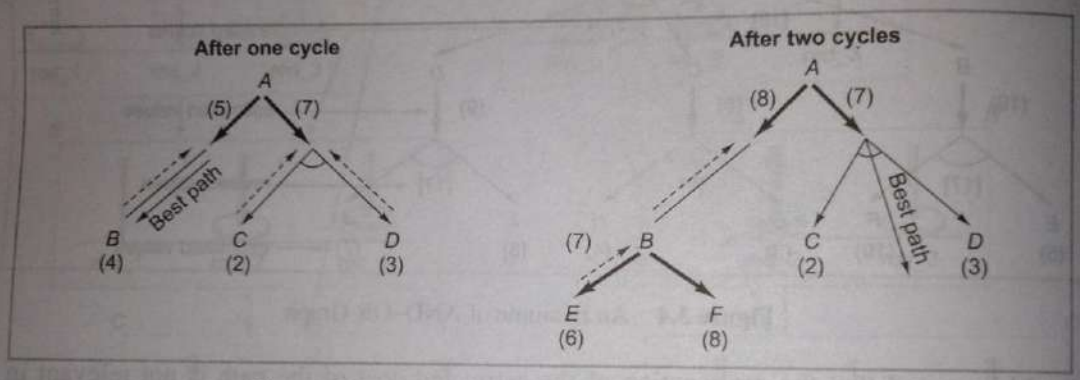


Figure 3.5 Labelling Procedure: First Two Cycles

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.

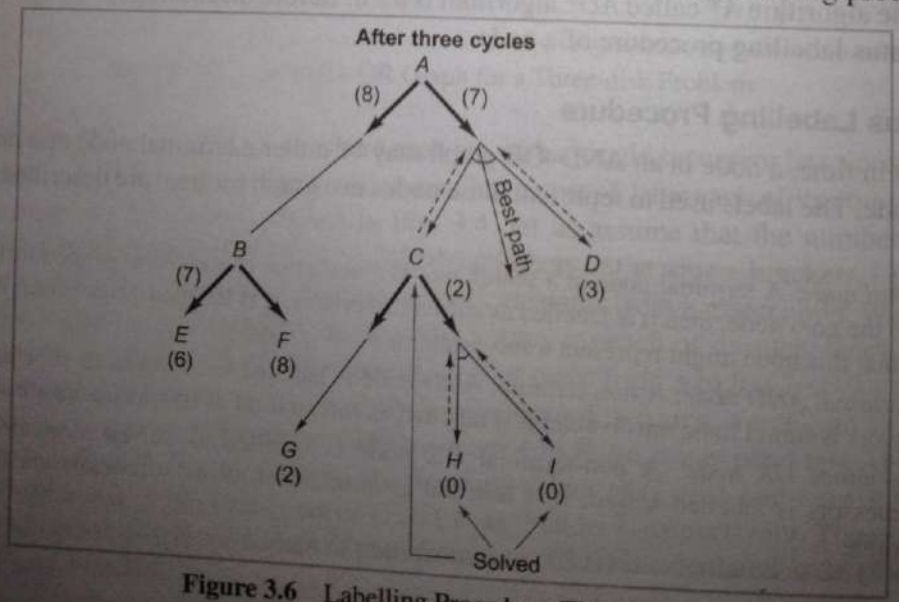


Figure 3.6 Labelling Procedure: Third Cycle

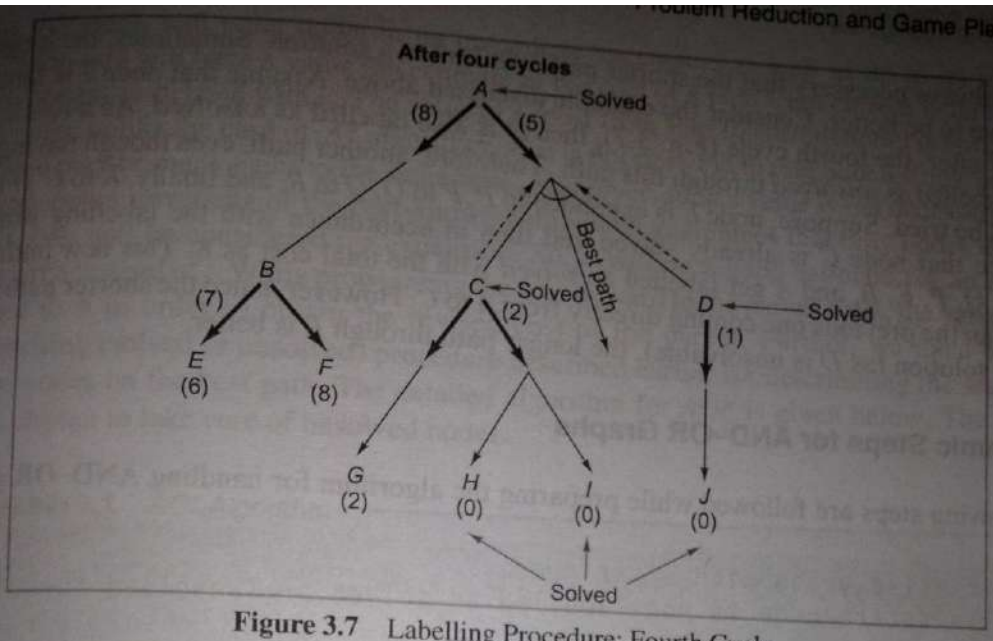


Figure 3.7 Labelling Procedure: Fourth Cycle

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).

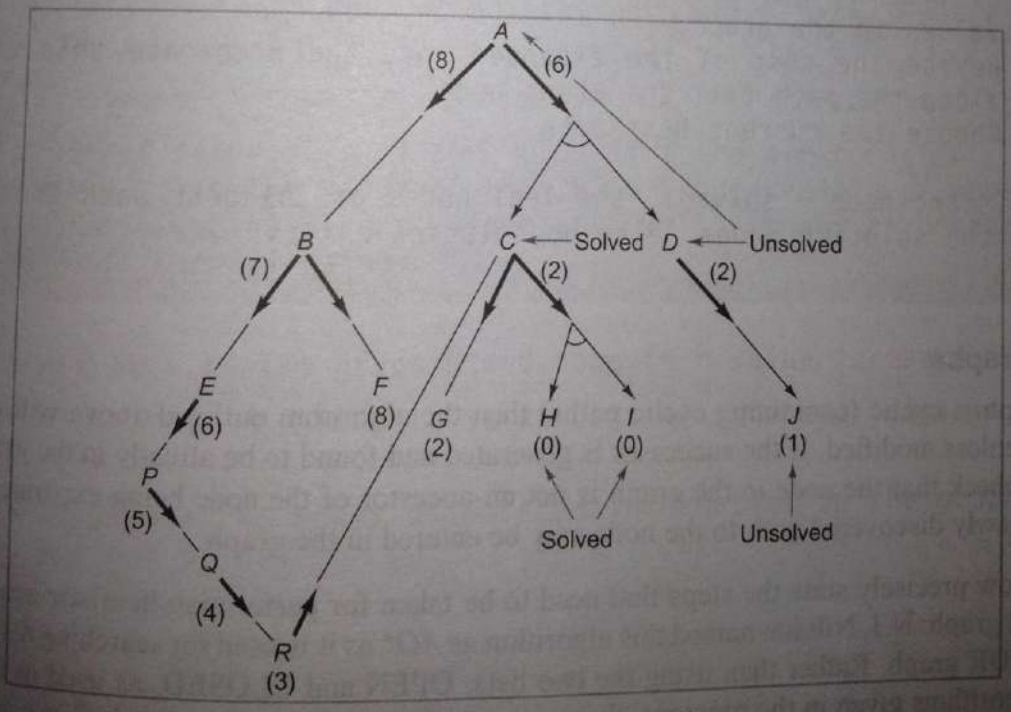


Figure 3.8 Non-Optimal Solution

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.

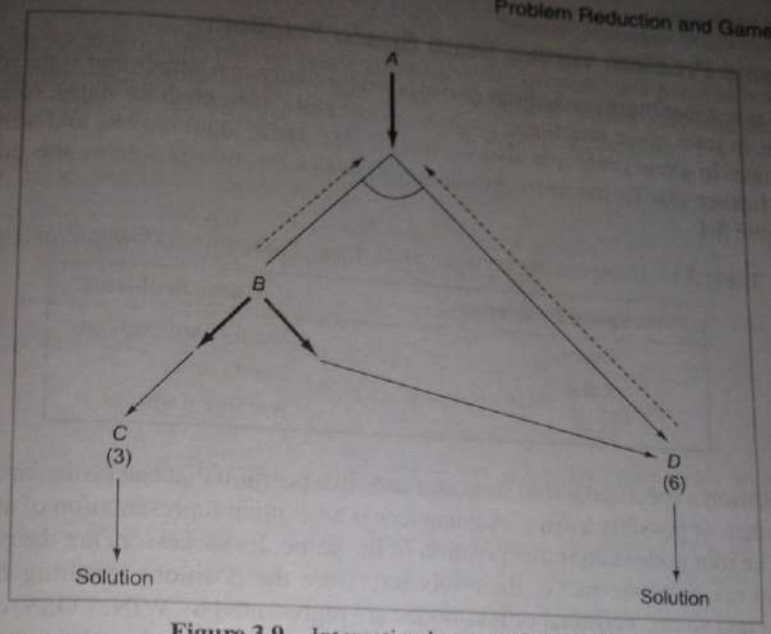


Figure 3.9 Interaction between Sub-Goals

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.

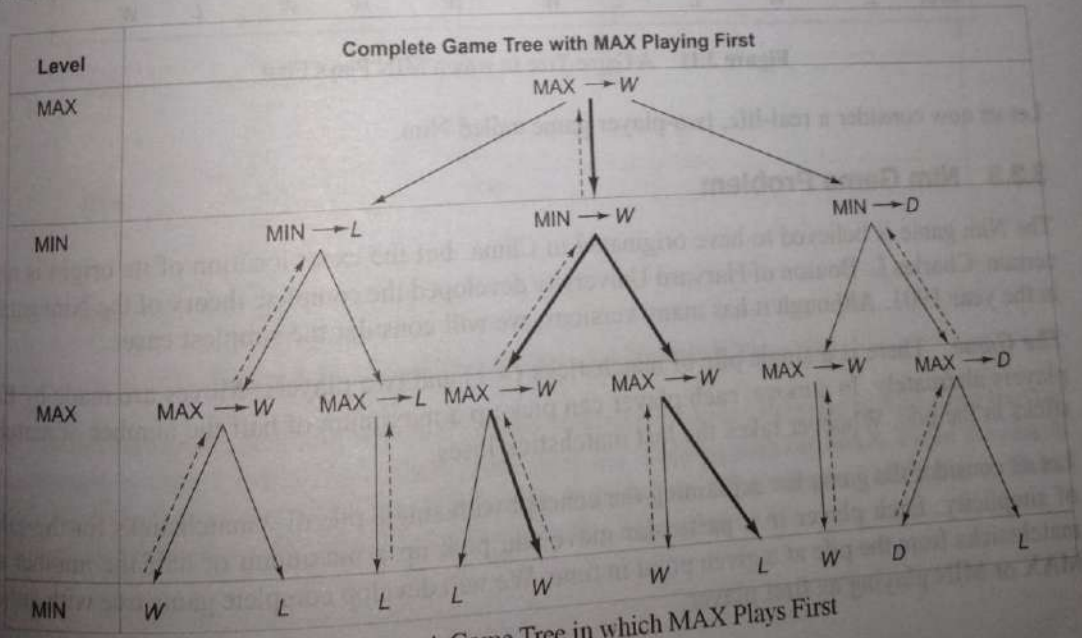


Figure 3.10 A Game Tree in which MAX Plays First

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.

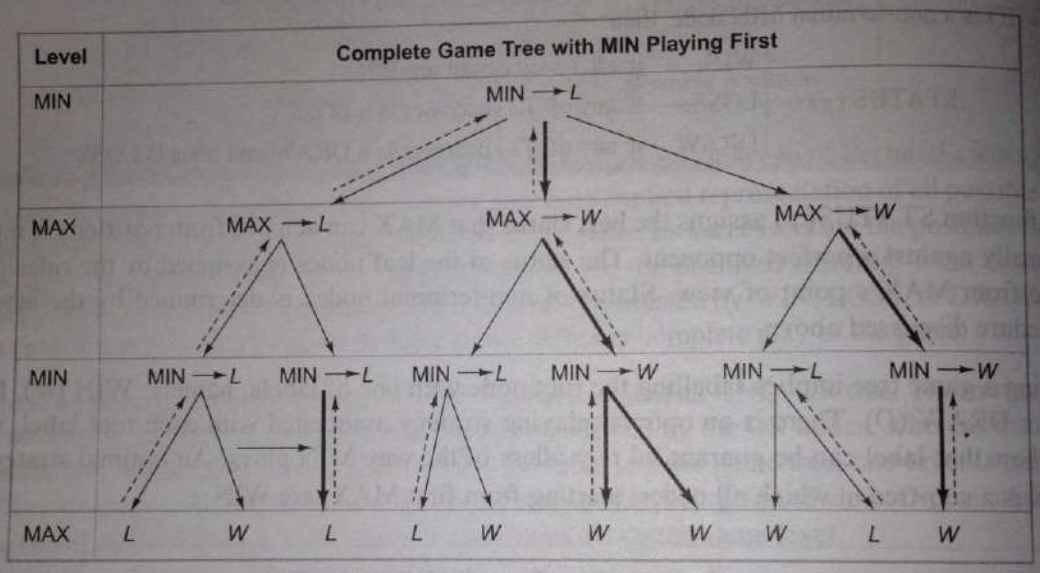


Figure 3.11 A Game Tree in which MIN Plays First

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.

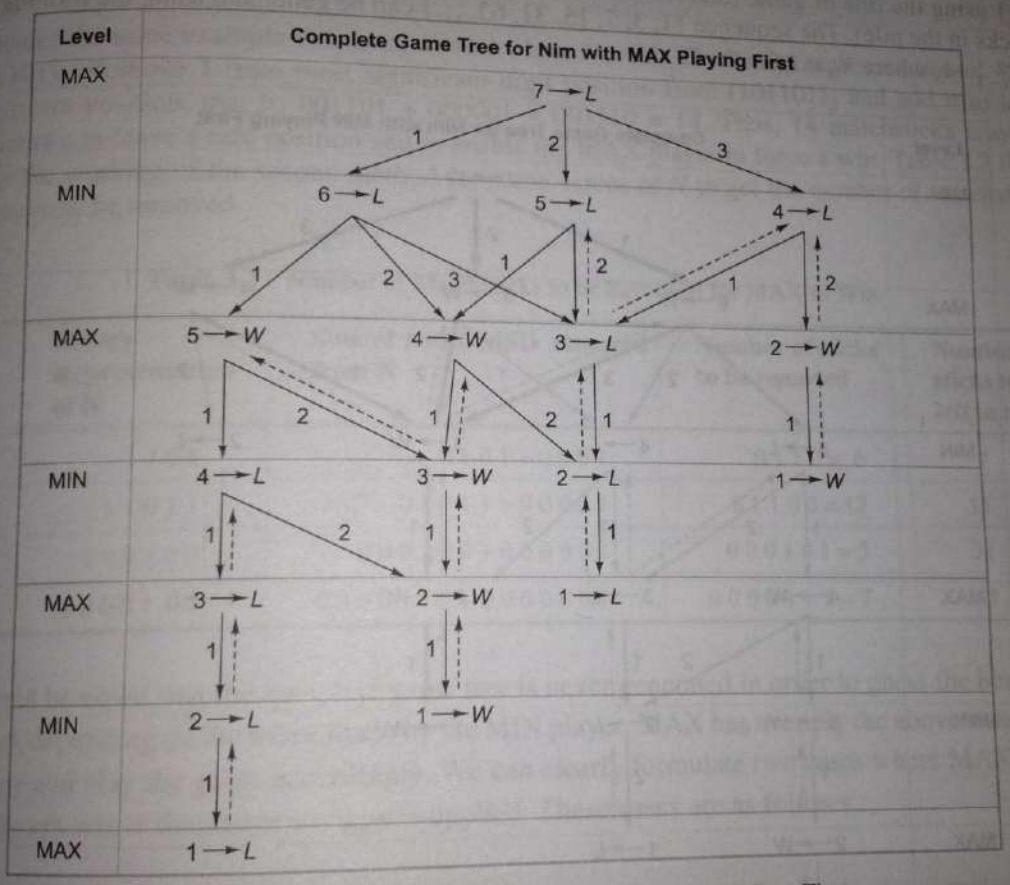


Figure 3.12 Game Tree for Nim in which MAX Plays First

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 $X_i = X_{i-1} + 1$, where $X_0 = 1$ for $i > 0$.

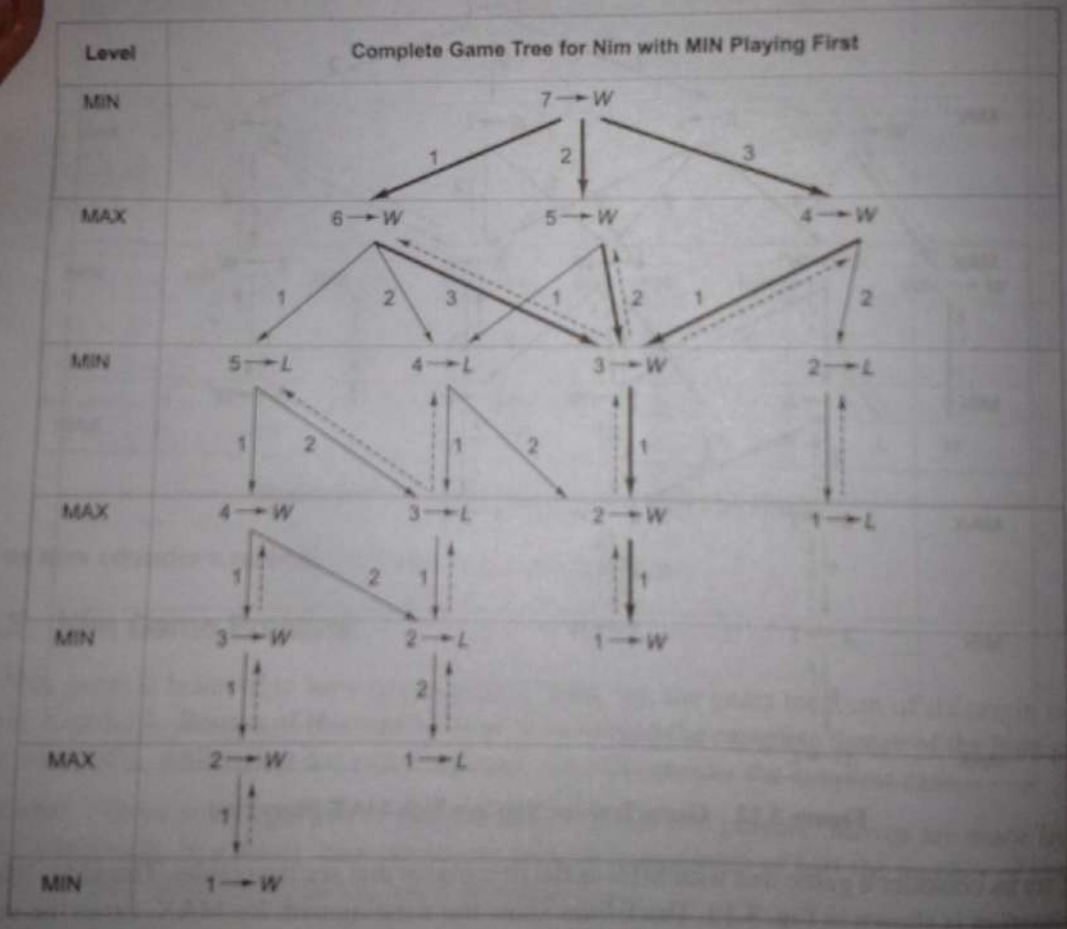


Figure 3.13 Game Tree for Nim in which MIN Plays First

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.

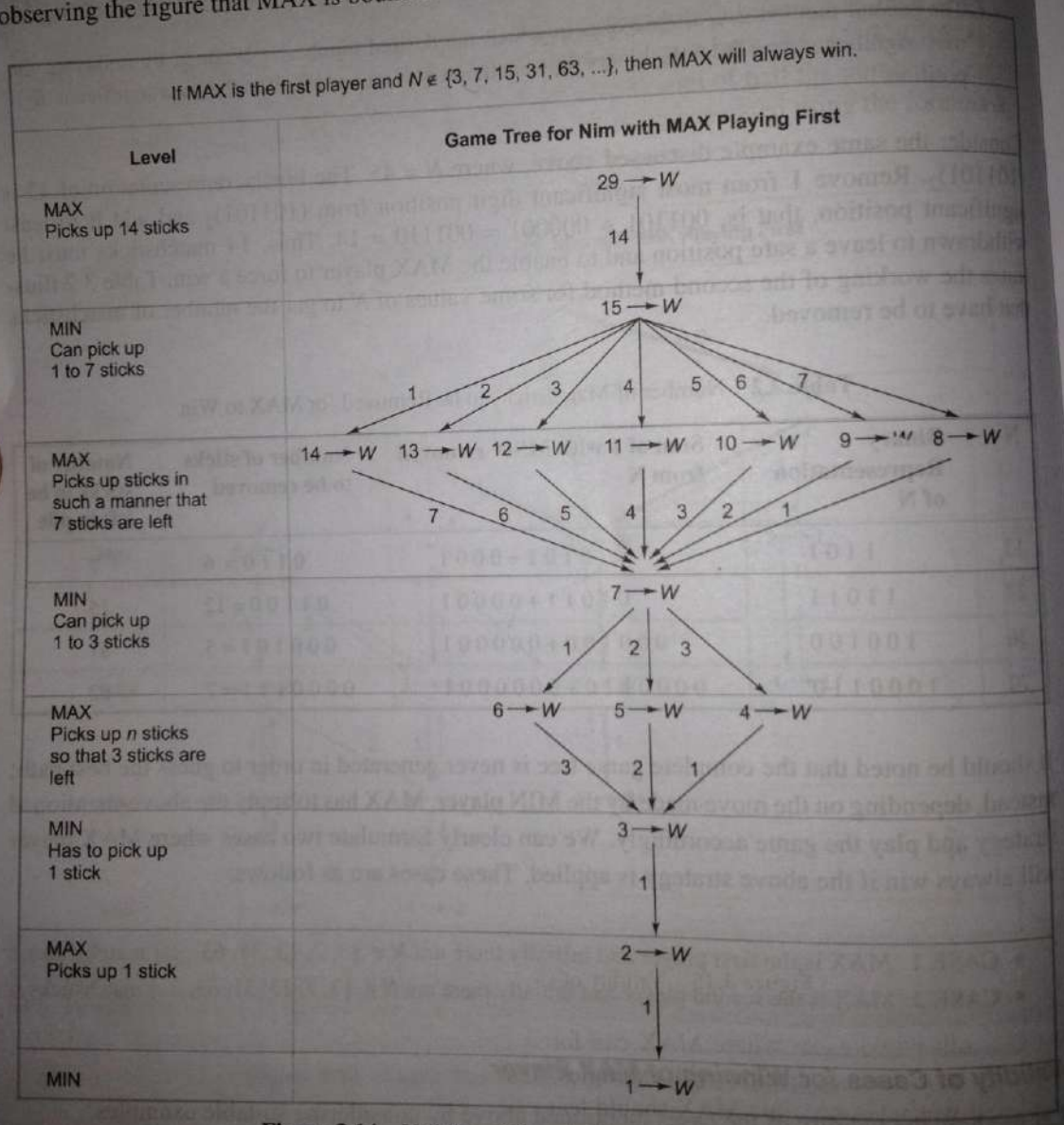


Figure 3.14 Validity of Case 1 (Example for $N = 29$)

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 $\in \{3, 7, 15, 31, 63, ...\}$.

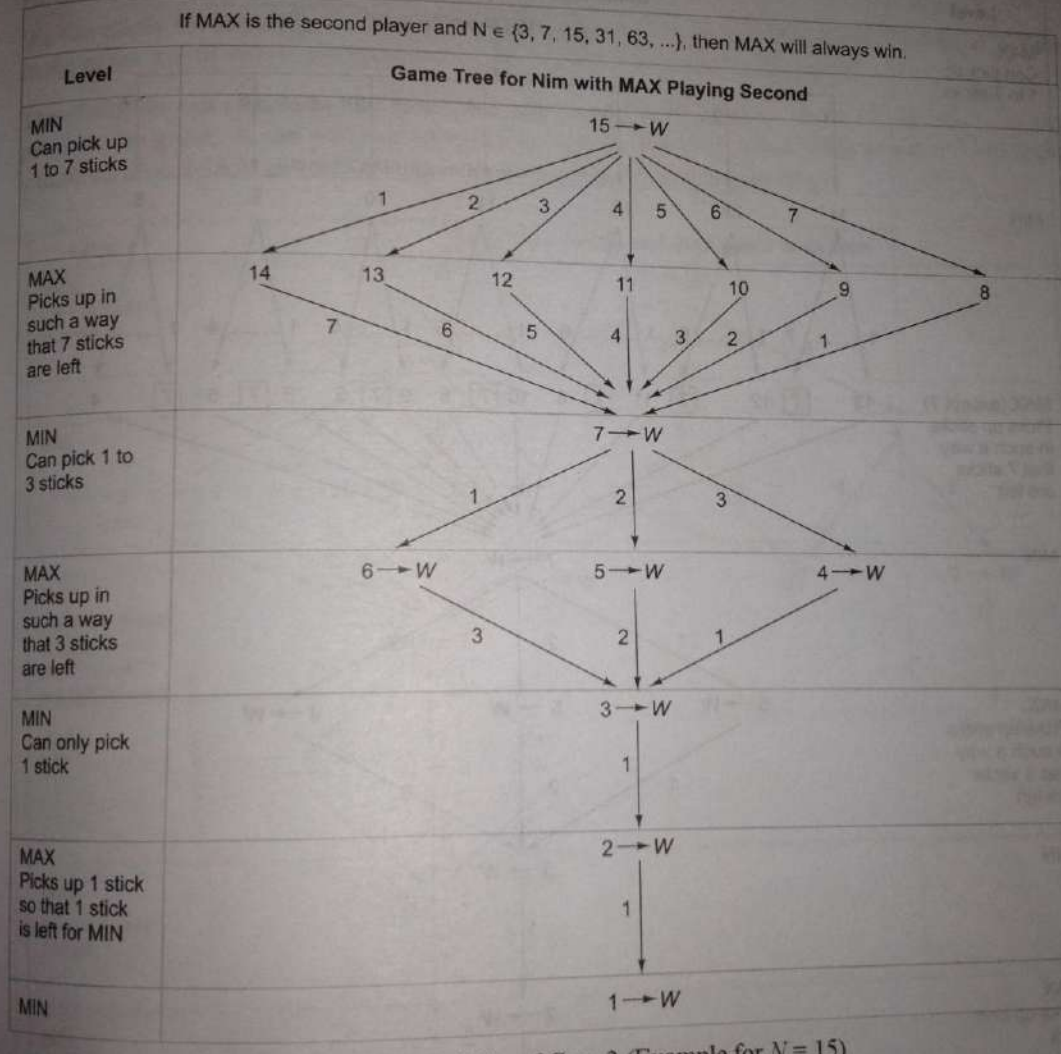


Figure 3.15 Validity of Case 2 (Example for $N = 15$)

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.

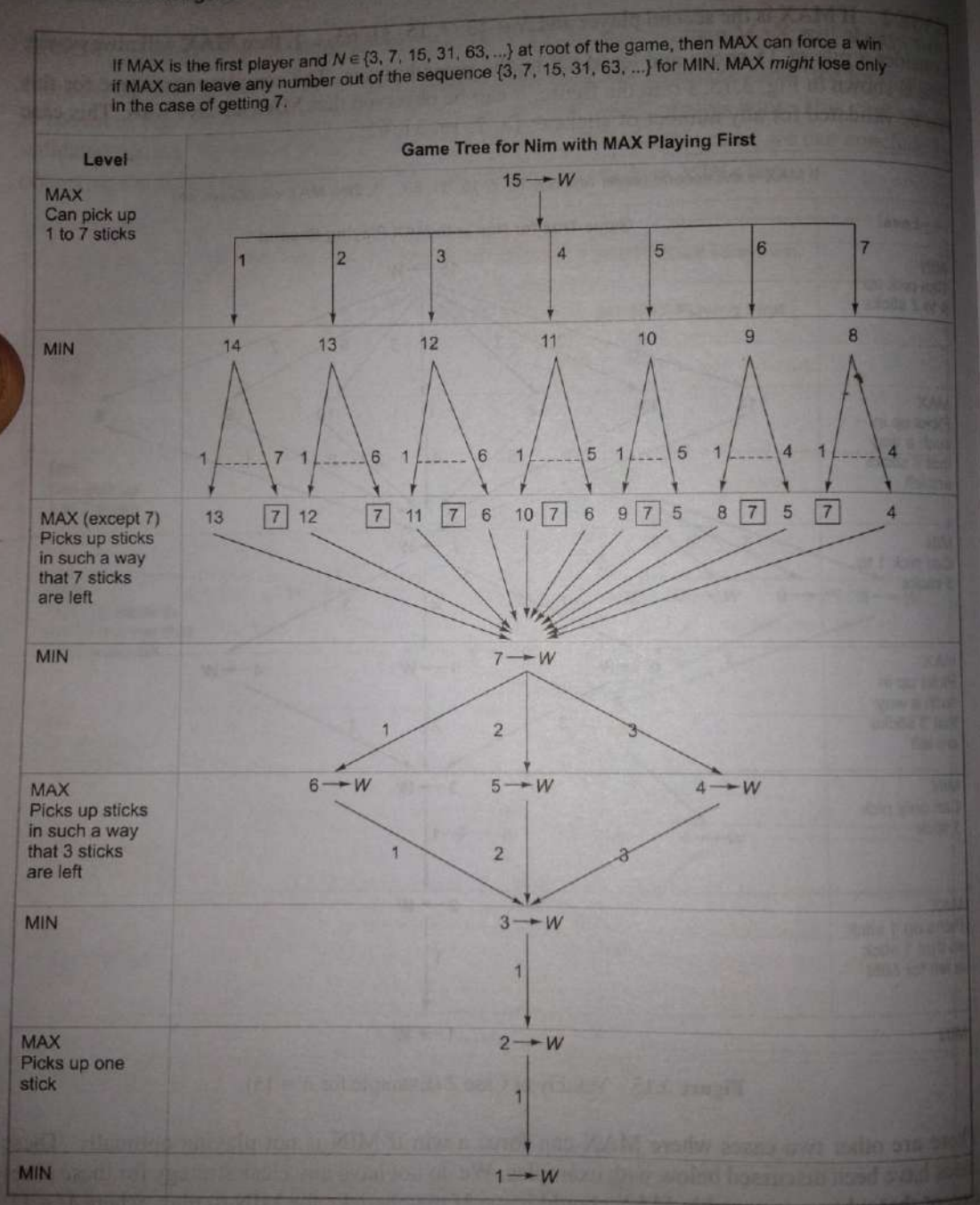


Figure 3.16 Validity of Case 3 (Example for $N = 15$)

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.

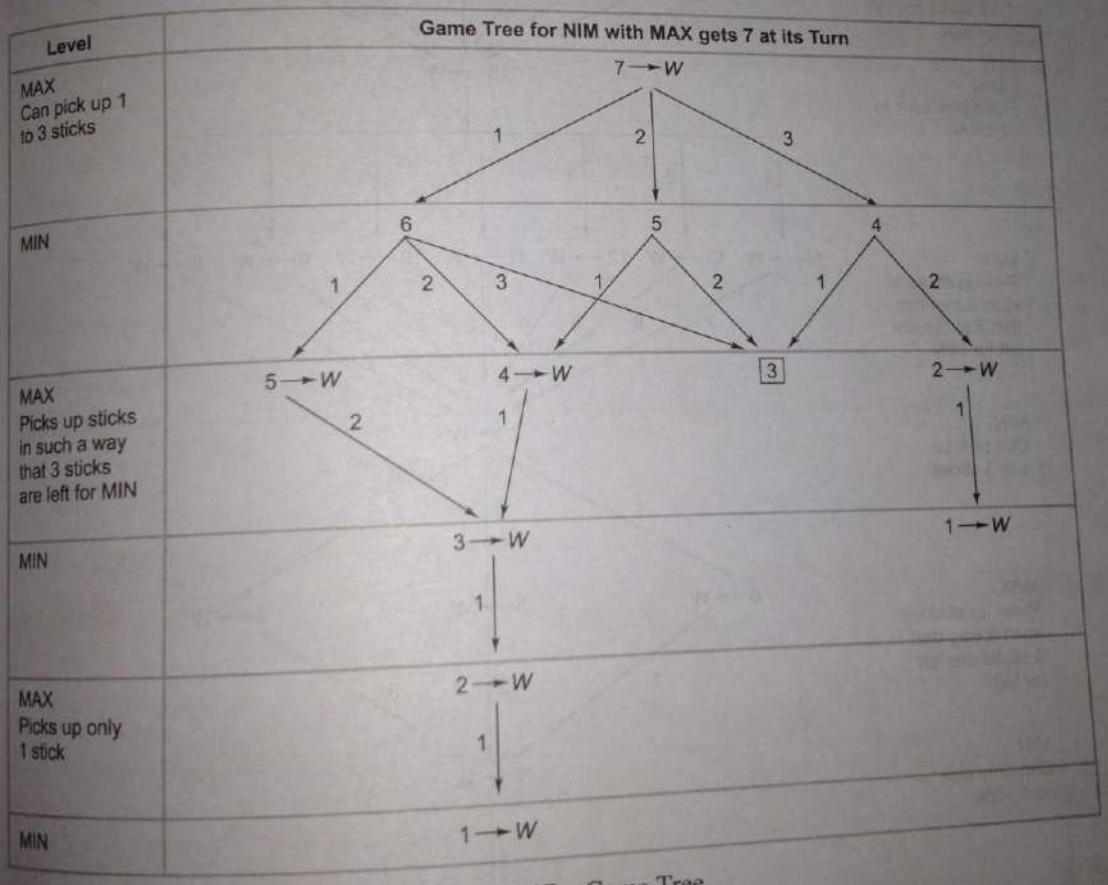
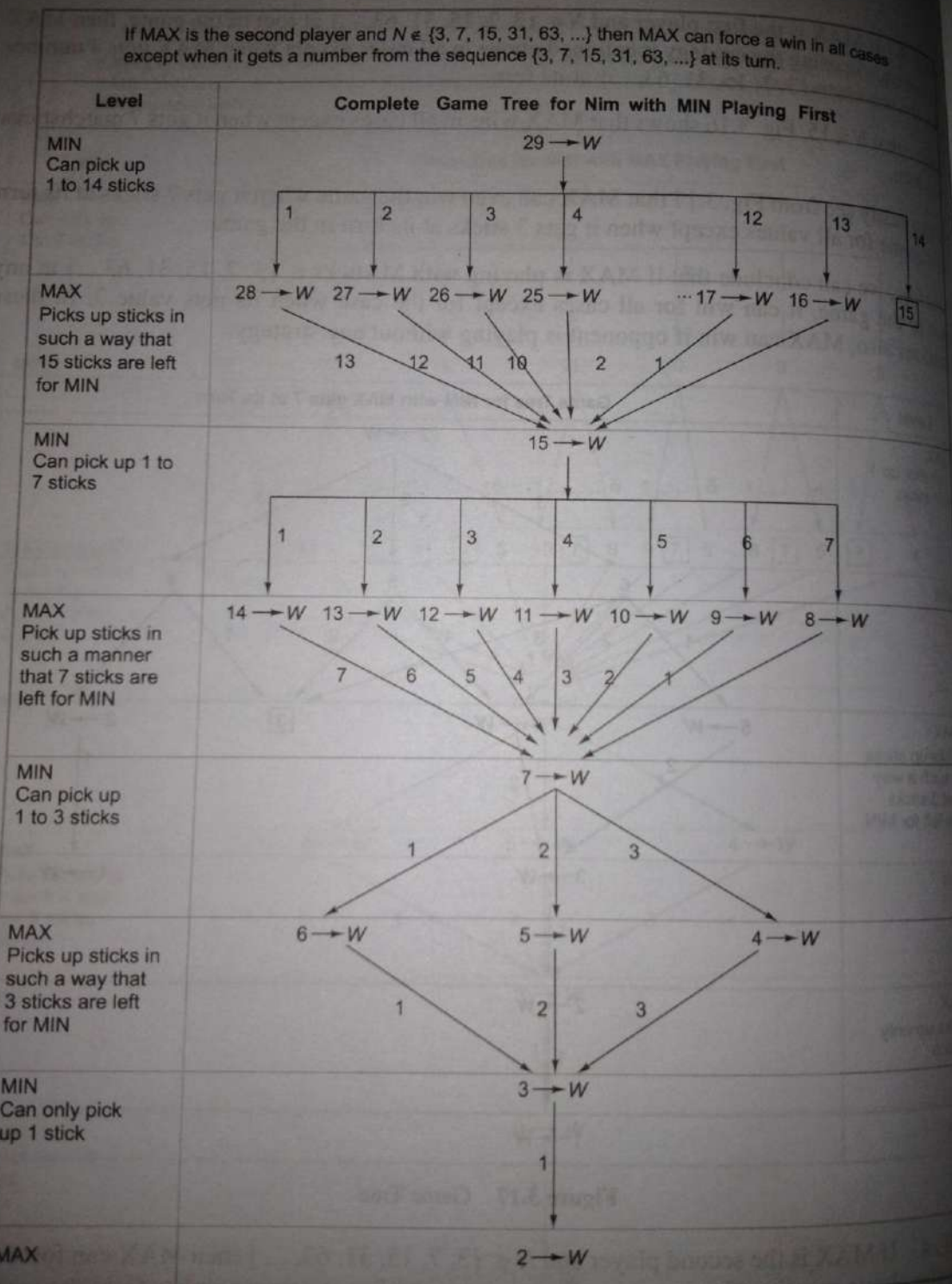


Figure 3.17 Game Tree

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.



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.

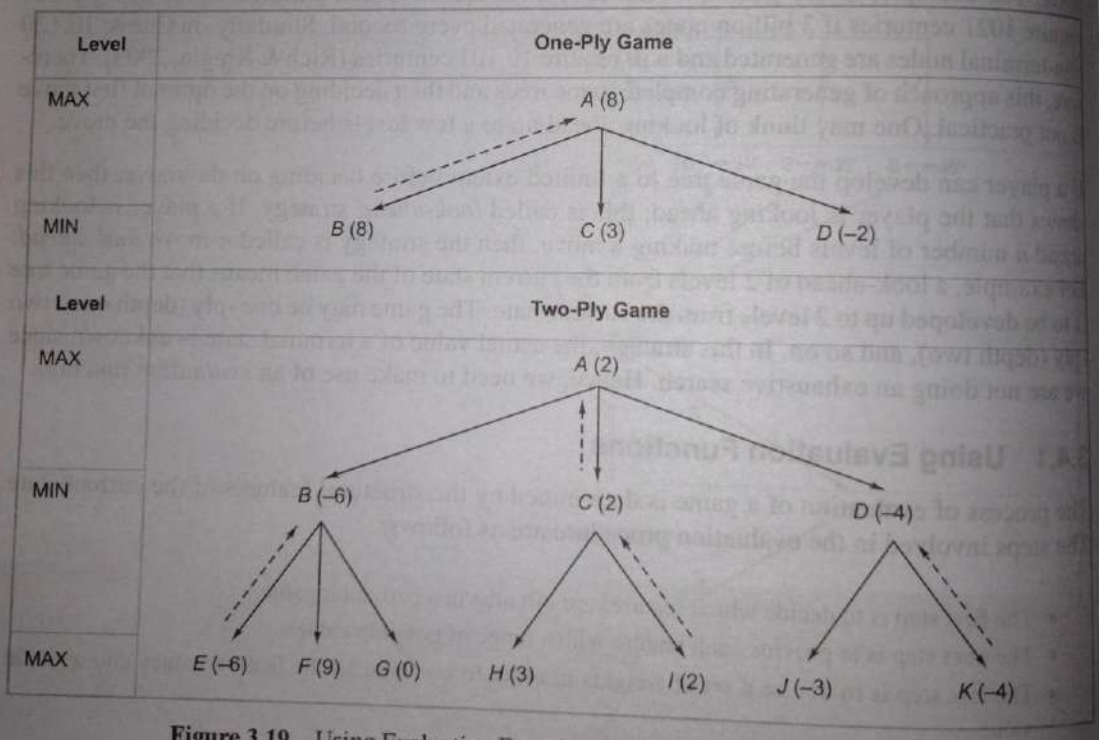


Figure 3.19 Using Evaluation Functions in One-Ply and Two-Ply Games

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

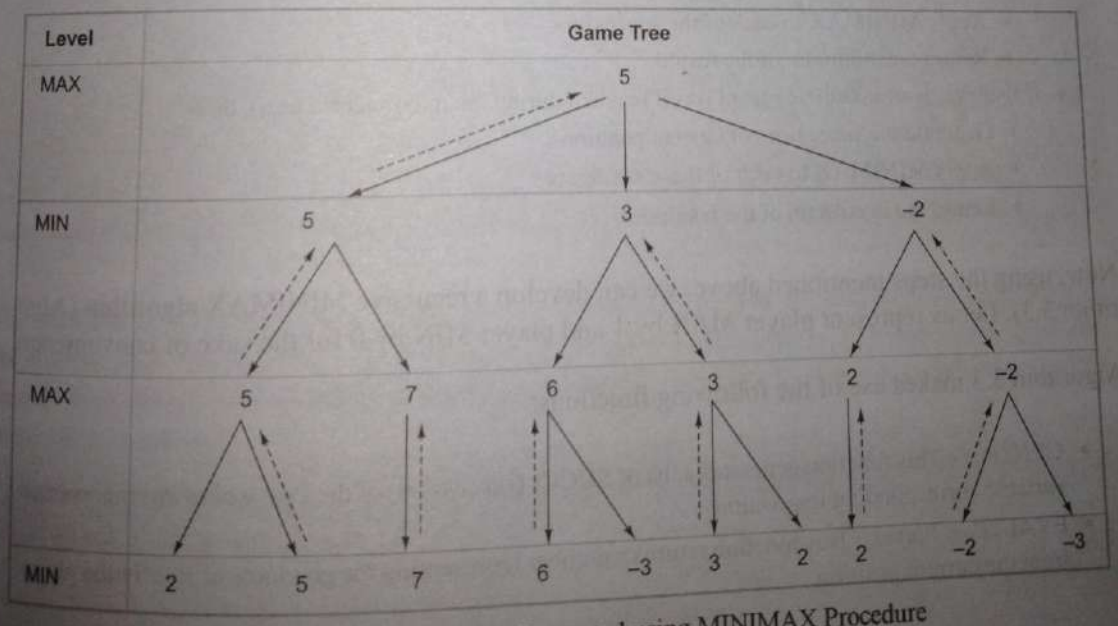


Figure 3.20 A Game Tree Generated using MINIMAX Procedure

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.

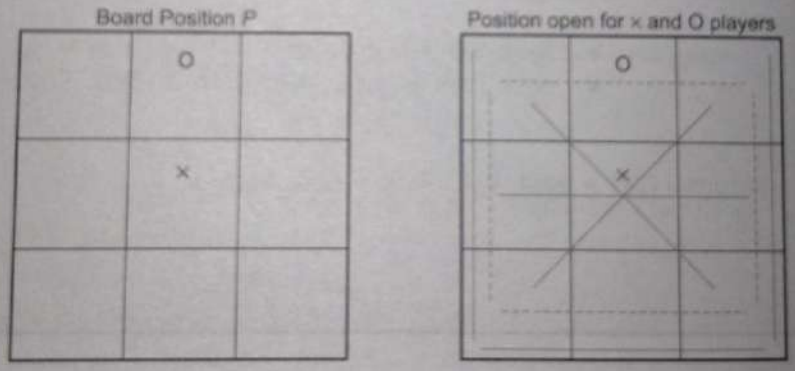


Figure 3.21 Board Position for a Game of Tic–Tac–Toe (Example)

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.

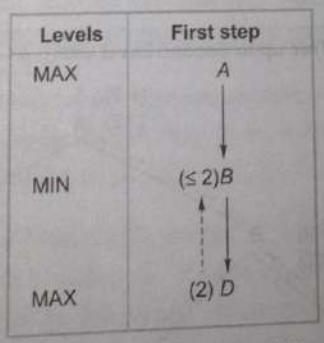


Figure 3.22 α–β Pruning Algorithm: Step 1

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, $\alpha = 2$.

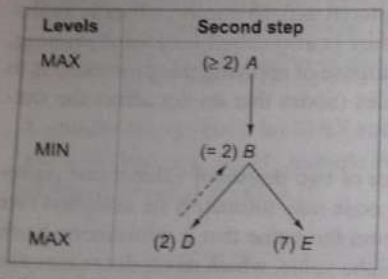


Figure 3.23 α–β Pruning Algorithm: Step 2

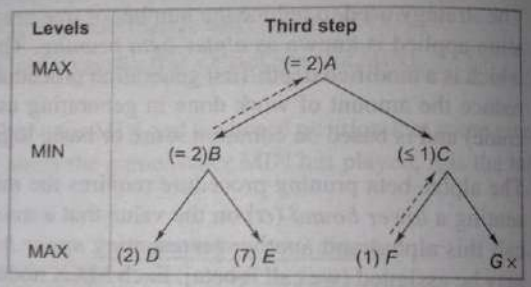


Figure 3.24 α–β Pruning Algorithm: Step 3

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

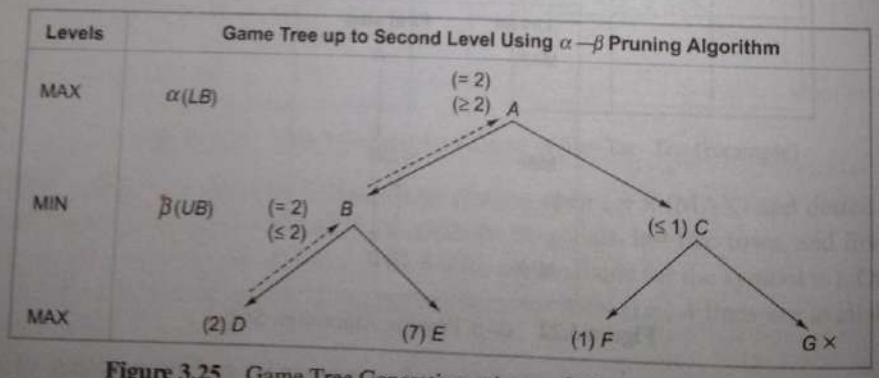


Figure 3.25 Game Tree Generation using α–β Pruning Algorithm

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.

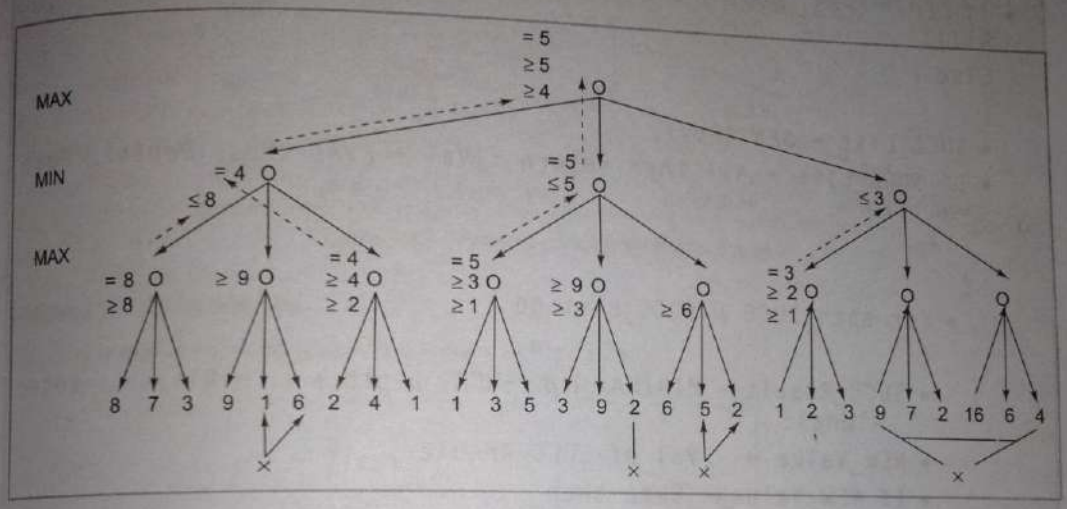


Figure 3.26 A Game Tree of Depth 3 and Branching Factor 3

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 DO
       {
       • 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*.

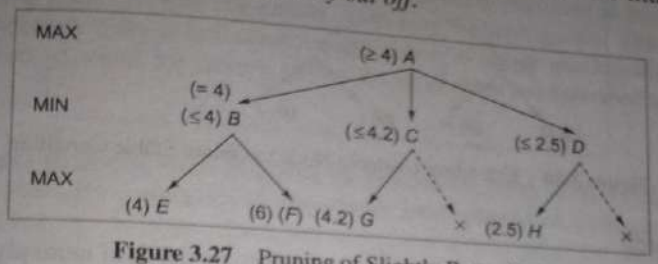


Figure 3.27 Pruning of Slightly Better Paths

Waiting for Quiescence

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

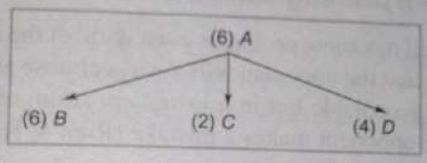


Figure 3.28 A One–Level Deep Game Tree

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

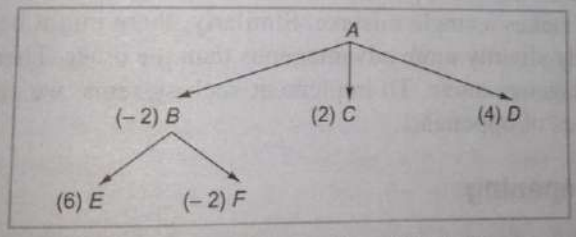


Figure 3.29 Expansion of Node *B*

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.

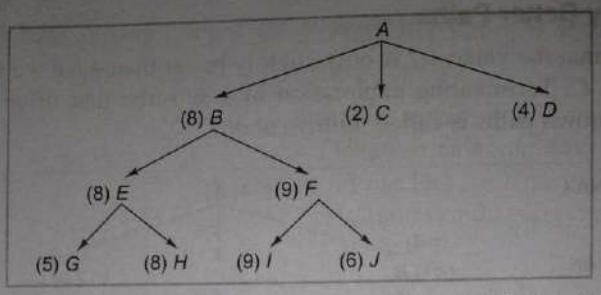


Figure 3.30 Extending Unstable Node to Obtain Stable Condition

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.

Chaper 4

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 $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \rightarrow \beta)$, 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 $\alpha : (A \vee B) \wedge (\neg B \rightarrow A)$ using truth table approach.

Solution Using the truth table approach, let us compute truth values of $(A \vee B)$ and $(\neg B \rightarrow A)$ and then compute for the final expression $(A \vee B) \wedge (\neg B \rightarrow A)$ (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	$\neg(\neg A) \equiv A$
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	$\neg(A \vee B) \equiv \neg A \wedge \neg B$ $\neg(A \wedge B) \equiv \neg A \vee \neg B$

Table 4.3 (Contd.)

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

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

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

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	$A \rightarrow B = (X)$	$X \wedge A = (Y)$	$Y \rightarrow B$
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 $\vee$	$(I{:}\vee)$	If any $A_i\ (1 \leq i \leq n)$ then $A_1 \vee \ldots \vee A_n$	If any $A_i\ (1 \leq i \leq n)$ is *true*, then $A_1 \vee \ldots \vee A_n$ is also *true*
Eliminating $\vee$	$(E{:}\vee)$	If $A_1 \vee \ldots \vee A_n, A_1 \rightarrow A, \ldots, A_n \rightarrow A$ then A	If $A_1 \vee \ldots \vee A_n, A_1 \rightarrow A, A_2 \rightarrow A, \ldots,$ 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 →	(*E*: →)	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 ↔	(*I*: ↔)	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 ↔	(*E*: ↔)	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 $\sim A$ is proved	If from A (which is *true*), a contradiction is proved then truth of $\sim A$ is also proved
Eliminating ~	(*E*: ~)	*If from* $\sim A$ *infer* $A_1 \wedge \sim A_1$ is proved then A is proved	If from $\sim A$, 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: ∧ (1)	A	2
E: ∧ (1)	B	3
I: ∨ (3)	$B \vee C$	4
I: ∧ (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* ($\sim$) and *implies* ($\rightarrow$) are allowed to form a formula. It should be noted that other logical operators, such as $\wedge$, $\vee$, and $\leftrightarrow$, can be easily expressed in terms of $\sim$ 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 $(\neg\alpha \rightarrow \neg\beta) \rightarrow (\beta \rightarrow \alpha)$

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 $\vdash \neg A \rightarrow (A \rightarrow B)$ by using deduction theorem.

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

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

Description	Formula	Comments
Theorem	$\{\neg A\} \vdash (A \rightarrow B)$	*Prove*
Hypothesis 1	$\neg A$	1
Instance of Axiom 1	$\neg A \rightarrow (\neg B \rightarrow \neg A)$	2
MP (1, 2)	$(\neg B \rightarrow \neg A)$	3
Instance of Axiom 3	$(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$	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 $\quad$ √(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 $\alpha : B \vee \sim(A \rightarrow B) \vee \sim A$ 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 $\{(A \vee B), C, \sim A\}$

$(A \vee B) \wedge (\sim A \wedge \sim A$

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.

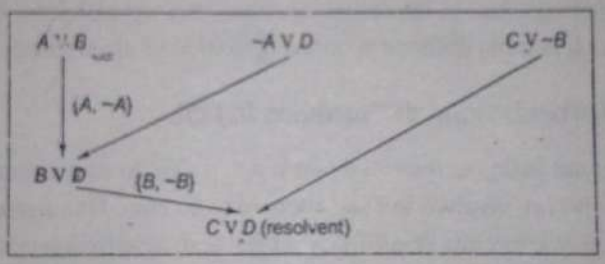


Figure 4.1 Resolution of clauses of Example 4.15

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 $S = \{A \vee B, \neg A \vee D, C \vee \neg B\}$.

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

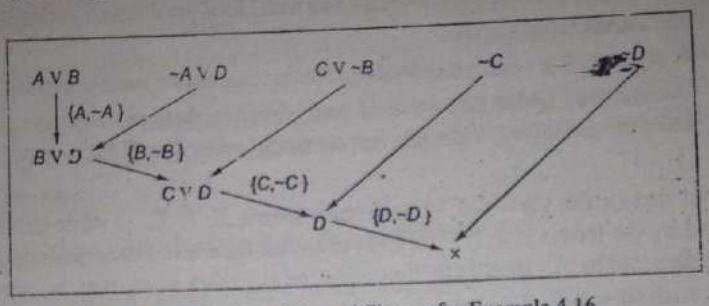


Figure 4.2 Resolution of Clauses for Example 4.16

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

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.

```
PL = {
 grandmother(X,Y) V ~mother(X, Z) V ~parent(Z, Y).
 parent(X, Y) V ~father(X, Y).
 parent(X,Y) V ~mother(X, Y).
 mother('Mary', 'John').
 mother('Tina', 'Kittu').
 mother('Kittu', 'Mita').
 father('John', 'Mike')
   }
```

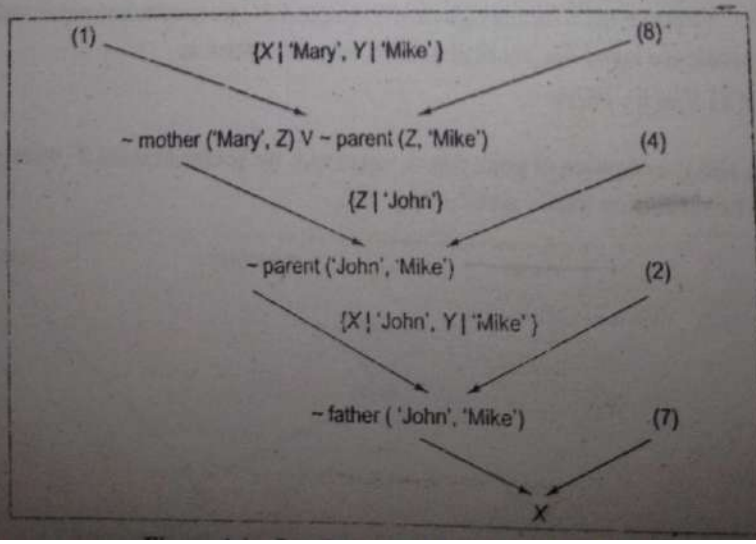
We can check whether the goals mentioned above are proved or not as shown in the following examples.

Example 4.22 Check whether the ground goal ← *grandmother*('*Mary*', '*Mike*') is true.

Solution Add ~ of the goal to the set *PL* to get a new set *S*, where *S* is defined as $PL \cup \{\sim G\}$

1. grandmother(X, Y) V ~mother (X, Z) V ~parent (Z, Y)
2. parent(X, Y) V ~father(X, Y)
3. parent(X, Y) V ~mother(X, Y)
4. mother('Mary', 'John')
5. mother('Tina', 'Kittu')
6. mother('Kittu', 'Mita')
7. father('John', 'Mike')
8. ~grandmother('Mary', 'Mike')

The resolution tree is generated as shown in Fig. 4.4. Since a contradiction is deduced, we can say that the goal is proved (or true).



4.7 Determine whether the following formulae are consistent or inconsistent using tableau method.

i. $(A \wedge \sim B) \wedge (\sim A \wedge B)$
ii. $(A \vee B) \wedge (\sim A \wedge \sim B)$
iii. $(\sim A \vee B) \rightarrow (A \rightarrow B)$
iv. $(A \rightarrow B) \leftrightarrow (\sim B \rightarrow \sim A)$
v. $(A \rightarrow B) \leftrightarrow (\sim C \rightarrow B) \wedge (C \vee B)$

4.8 Show that the following formulae are valid by giving tableau proof of each of these.

i. $A \rightarrow (B \rightarrow A)$
ii. $\sim(A \vee B) \leftrightarrow (\sim A \wedge \sim B)$
iii. $(\sim A \vee B) \leftrightarrow (A \rightarrow B)$
iv. $(A \wedge (B \vee C) \leftrightarrow [(A \wedge B) \vee (A \wedge C)]$
v. $((A \rightarrow (B \rightarrow C)) \rightarrow [(A \wedge B) \rightarrow C]$

4.9 You are given a set of propositional formulae. Show that the following are logical consequence (LC) of the given set using resolution refutation method.

i. $(B \vee C)$ LC $\{A \wedge \exists, \sim A \vee C\}$
ii. $(A \vee C)$ LC $\{A, B \rightarrow C, B\}$
iii. $(C \rightarrow A)$ LC $\{B \wedge C) \rightarrow A, B\}$
iv. $(A \vee \sim B)$ LC $\{A \vee C, \sim B \vee \sim C\}$
v. $(\sim U \wedge S)$ LC $\{A \vee C, \sim C \rightarrow B, \sim B, A \rightarrow S, \sim U\}$

4.10 Evaluate the truth values of the following formulae. Define your own interpretation.

i. $(\exists X)\ (p[f(X)] \wedge q[X, f(c)])$
ii. $(\exists X)\ [p(X) \wedge q(X, c)]$
iii. $(\exists X)\ [p(X) \rightarrow q(X, c)]$
iv. $(\forall X)\ [p(X) \wedge (\exists Y)\ q(X, Y)]$
v. $(\forall X)\ (p(X) \rightarrow (\exists Y)\ q[f(c), Y])$

4.11 Transform the following formulae into PNF and then into Skolem Standard Form.

i. $(\forall X)\ (p(X) \rightarrow (\exists Y)\ q[f(c), Y])$
ii. $(\forall X)\ (\exists Y)\ [q(X, Y) \rightarrow p(X)]$
iii. $(\forall X)\ \{(\exists Y)\ p(X, Y) \rightarrow \sim[(\exists z)\ q(z) \wedge c(X)]\}$
iv. $(\forall X)\ (\exists Y)\ p(X, Y) \rightarrow ((\exists Y)\ p(X, Y)$
v. $(\forall X)\ [(\exists Y)\ p(X, Y) \wedge \{(\exists z)\ q(z) \wedge c(X)\}]$

4.12 Consider the following English sentence:

"Anything anyone eats is called food. Mita likes all kinds of food. Burger is a food. Mango is a food. John eats pizza. John eats everything Mita eats."

Translate these sentences into formulae in predicate logic and then to program clauses. Use resolution algorithm to answer the following goals

i. What food does John eat?
ii. Does Mita like pizza?
iii. Which food does John like?
iv. Who likes what foods?
v. Prove the statement "Mita likes pizza and burger" using resolution.