UNIT - 2

Problem Solving: State Space Search and Control Strategies

Introduction:-

→ problem solving is a method of deriving solution steps beginning from initial description of the problem to the desired solution.

→ The task is solved by a series of actions that minimizes the difference b/w the given situation and the desired goal.

→ In AI the problems are frequently modelled as a state prspace problem where the state space is a set of all possible states from start to goal states.

→ The set of states form a graph in which two states are linked if there is an Operation which can be executed to transform one state to other.

→ while solving a problem, the state space is generated in the process of searching for its solution.

→ There is a difference b/w the state space search used in AI and the conventional computer science search methods.

→ The two types of problem-solving methods methods that are generally followed include general purpose and special purpose methods.

→ A general-purpose method is applicable to a wide variety of problems, whereas a special-purpose method is tailor-made for the particular problem and often exploits very specific features of the problem.

→ The most general approach for solving a problems is to generate the solution and test it.

→ for generating new state in the search space, an action/operator/rule is applied and tested whether the state is the goal state/not.

→ Incase the state is not the goal state, the procedure is repeated.

→ The order of application of the rules to the current state is called Control strategy.

General problem Solving:-

The following subsections describe production systems and state space search methods that facilitate the modeling of problems and search processes.

production systems(ps) is one of the formalisms that helps AI programs to do search process more conveniently in state-space problems. This system comprises of start/Initial states and Goal/final states of the problem along with one/more databases consisting of suitable & necessary info for the particular task.

→ Generally knowledge representation schemes are used to structure information in these databases.

→ PS consists of number of production rules in which each production rule has left side that determines the applicability of the rule, and the right side that describes the action to be performed if the rule is applied.

→ Left side of the rule is current state, whereas the right side describes the new state that is obtained from applying the rule.

Water-Jug Problem:-

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

Solution:- state space for this problem can be described as the set of ordered pairs of Intergers (x, y) such that x - represents the no of gallons of water in 5-g jug and y for 3-g jug.

(1) Start state is (0, 0)

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

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

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

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	$(x, y \mid x+y \geq 3 \wedge x > 0)$	$(x-(3-y), 3)$ until 3-g jug is full.	Pour water from 5-g jug into 3-g jug.

fig: production Rules

It should to be noted that there may be more than one solutions for a given problem. We have shown two possible solutions paths as below.

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	-	

fig: Solution path-1

Rule applied	5-g jug	3-g jug	step-No
start state	0	0	
3	0	3	1
5	3	0	2
3	3	3	3
7	5	1	4
2	0	1	5
5	1	0	6
3	1	3	7
5	4	0	8
Goal state	4		

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 & right banks of the river as (L, R) where each bank is represented as a list [nM, mC, B]. Here n is the no. 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 : ($[n_1 M, m_1 C, \_], [n_2 M, m_2 C, \_]$), 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 the possible solution path trace is given in the Table 2.5 :

Table 2.4 :- Production rules for missionaries and cannibals Problem

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

Table 2.5 :- solution Path

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

2.2.2 State-space search

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

1. A set of S containing start states of the problem
2. A set of 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 are 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 solutions 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

$L = n_1 M m_1 C_1 B$ and

$R = n_2 M m_2 C O B$

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 areas of the tree indicate the application of one of the operators. The nodes for which no operators are applied, are called leaf nodes, which have no arcs leading from them. To simplify, we have not generated entire search space and avoided illegal and looping states. Depth-first- or breadth-first strategy or some illegal heuristic searches is used to generate the search space.

The space for searching can be generated using operators which are valid, are shown in the fig 2.1. The sequence of operators applied to solve this problem is given in the table 2.6

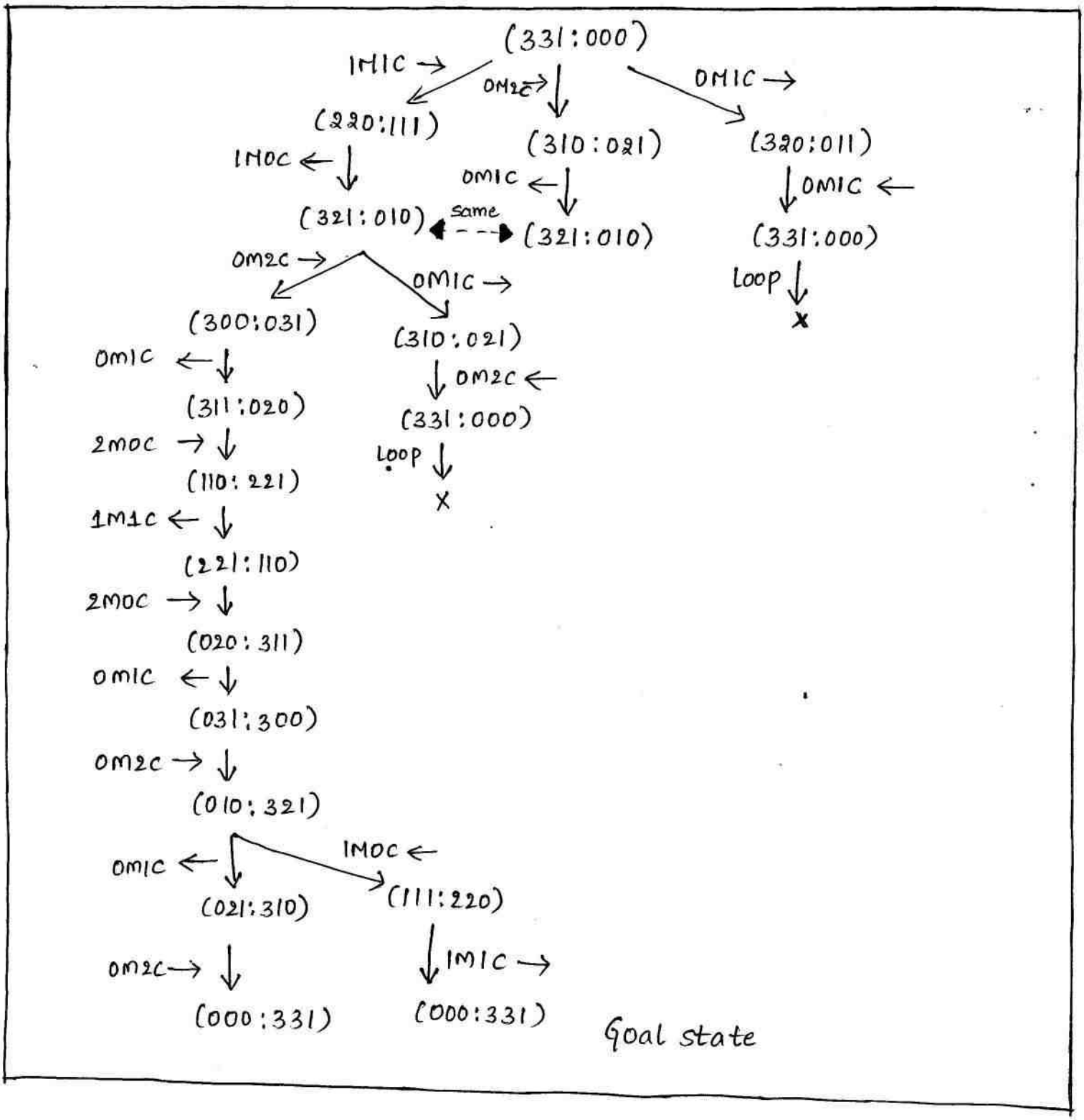


Fig :- Search Space

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.

The Eight-Puzzle Problem

Problem statement:- The eight-puzzle problem has a 3x3 grid with 8 randomly numbered (1 to 8) tiles arranged on it with one empty cell. At any point, the adjustro 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

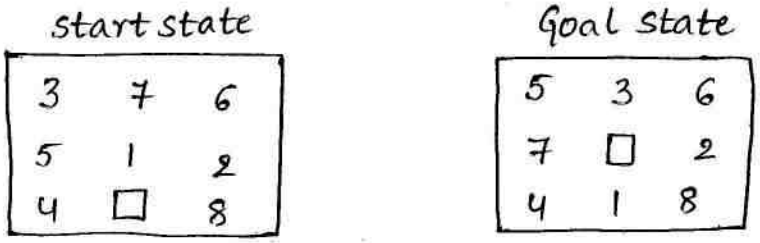


Fig 2.2 :- Eight-puzzle problem

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 should 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.

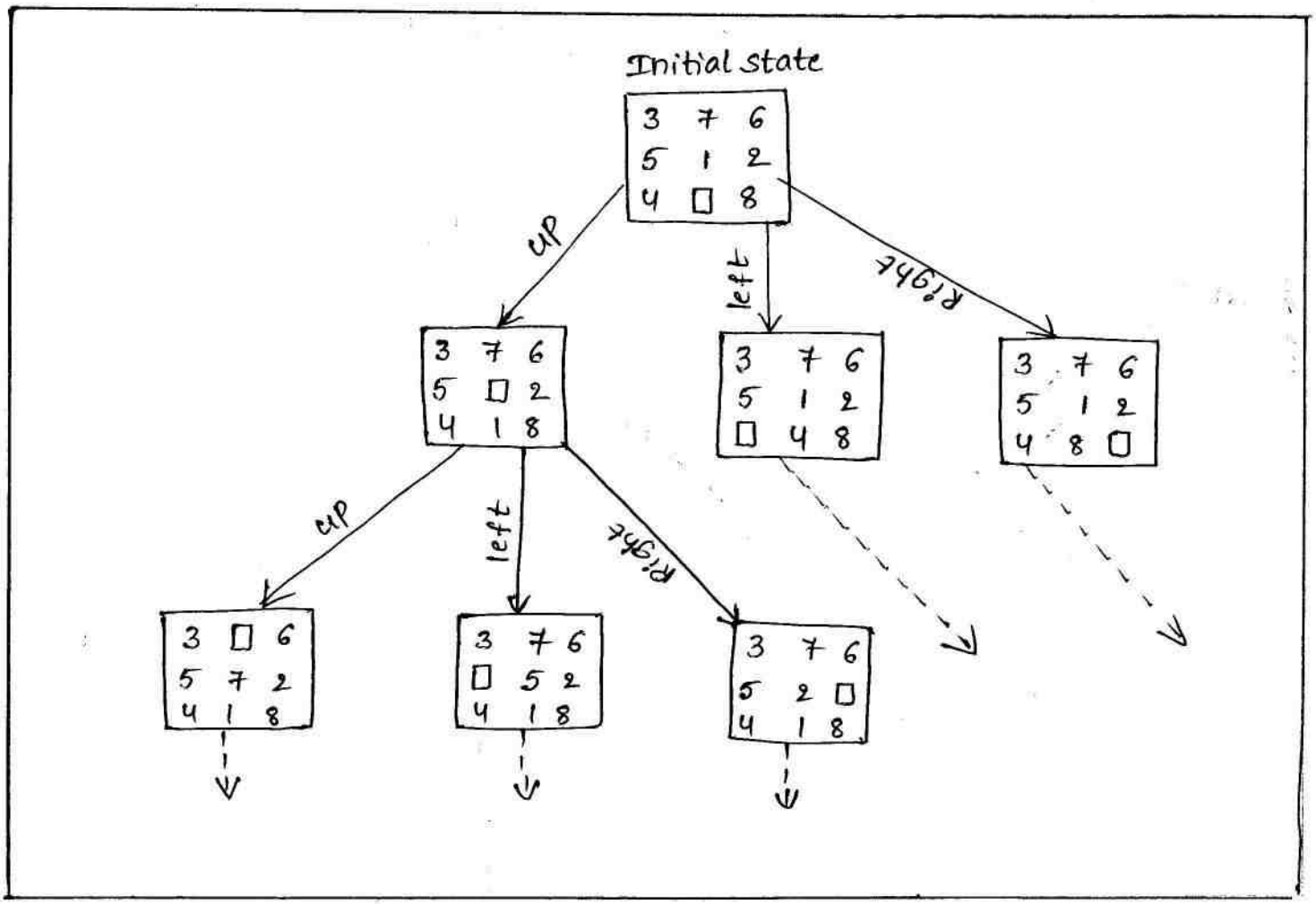


Figure 2.3 : Partial Search Tree for Eight Puzzle Problem

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 and systematic control strategies but these are blind searches. In depth-first strategy, we follows 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, uniformed, 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 strategies 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 chining, from the start state
- Goal-Driven search, called backward chaining, from the goal state

Forward chining:- 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 opss 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.

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 initially. some of these are given below.

Type of Problem:- There are three types of problems in real life.

(a) Ignorable (b) Recoverable (c) Irrecoverable

(a) 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.

(b) 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.

(c) 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, each such problems can be solved in Parallel processing environment.

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

Consistency of knowledge based 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 night time.

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 appropriate 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, our goal is to find out 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 uniformed exhaustive searches like 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 are examined at the same depth before going deeper. the BFS always gives an optimal path or solution.

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

Algorithm (BFS)

Input: START and GOAL states
Local variables : OPEN, CLOSED, STATE-X, SUCCS, FOUND:
output : Yes or No
method :

- initialize OPEN list with START and CLOSED = ϕ;
- FOUND = false ;
- while (OPEN $\neq \phi$ and FOUND = false) do
  {
  - remove the first state from OPEN and call it STATE-X;
  - Put STATE-X in the front of CLOSED list (maintained as stack);
  - if STATE-X = GOAL then FOUND = true else
    {
    - perform EXPAND operation on STATE-X, producing a list of SUCCS;

- remove from successors those states. if any, that are in the CLOSED list;
- append succs at the end of the OPEN list; /*queue*/

}

} /*end while*/

- if FOUND = true then return Yes else return No
- stop

→ Let us see the search tree generation from start state of the water jug problem using BFS algorithm. At each state, we apply first applicable rule. If it generates previously generatinoed 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.

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

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

(Refer fig 2.4)

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 descendents. 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 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 ≠ ϕ 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

Water Jug problem		
Search tree generation using DFS	OPEN list	CLOSED list
Start state (0,0) ↓ {1} (5,0) {2} ← X, {3} ↓ (5,3) ↓ (0,3) ←	{((0,0), nil)} [((5,0),(0,0))] [((5,3),(5,0))]	 [((0,0), nil)] [((5,0),(0,0)), ((0,0), nil)]

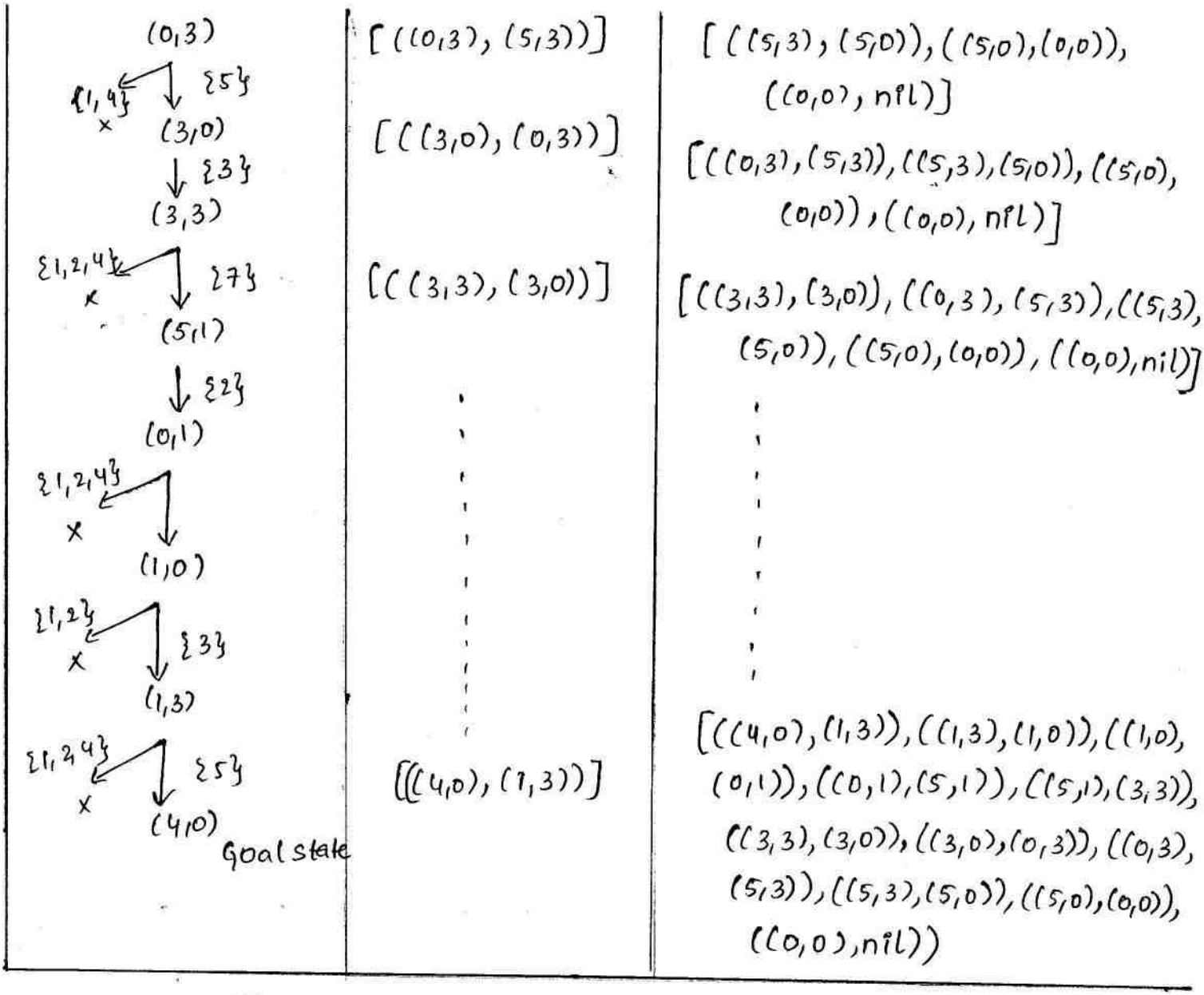


Fig:- 2.5 Search Tree Generation Using DFS

The path is obtained from the list stored in CLOSED. The solution path is $(0,0) \rightarrow (5,0) \rightarrow (5,3) \rightarrow (0,3) \rightarrow (3,0) \rightarrow (3,3) \rightarrow (5,1) \rightarrow (0,1) \rightarrow (1,0) \rightarrow (1,3) \rightarrow (4,0)$

Comparisions:- since these are unguided, blind, and exhaustive searches, we cannot say much about them but can make some observation.

- 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 the 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 state 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.

2.4.3 Depth-First Iterative Deepening :-

Depth-first iterative deepening (DFID) takes advantages of both 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 , FOUND = false
- while (FOUND = false) do
  {
  - Perform a depth first search from start to depth d.
  - if goal state is obtained from 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:

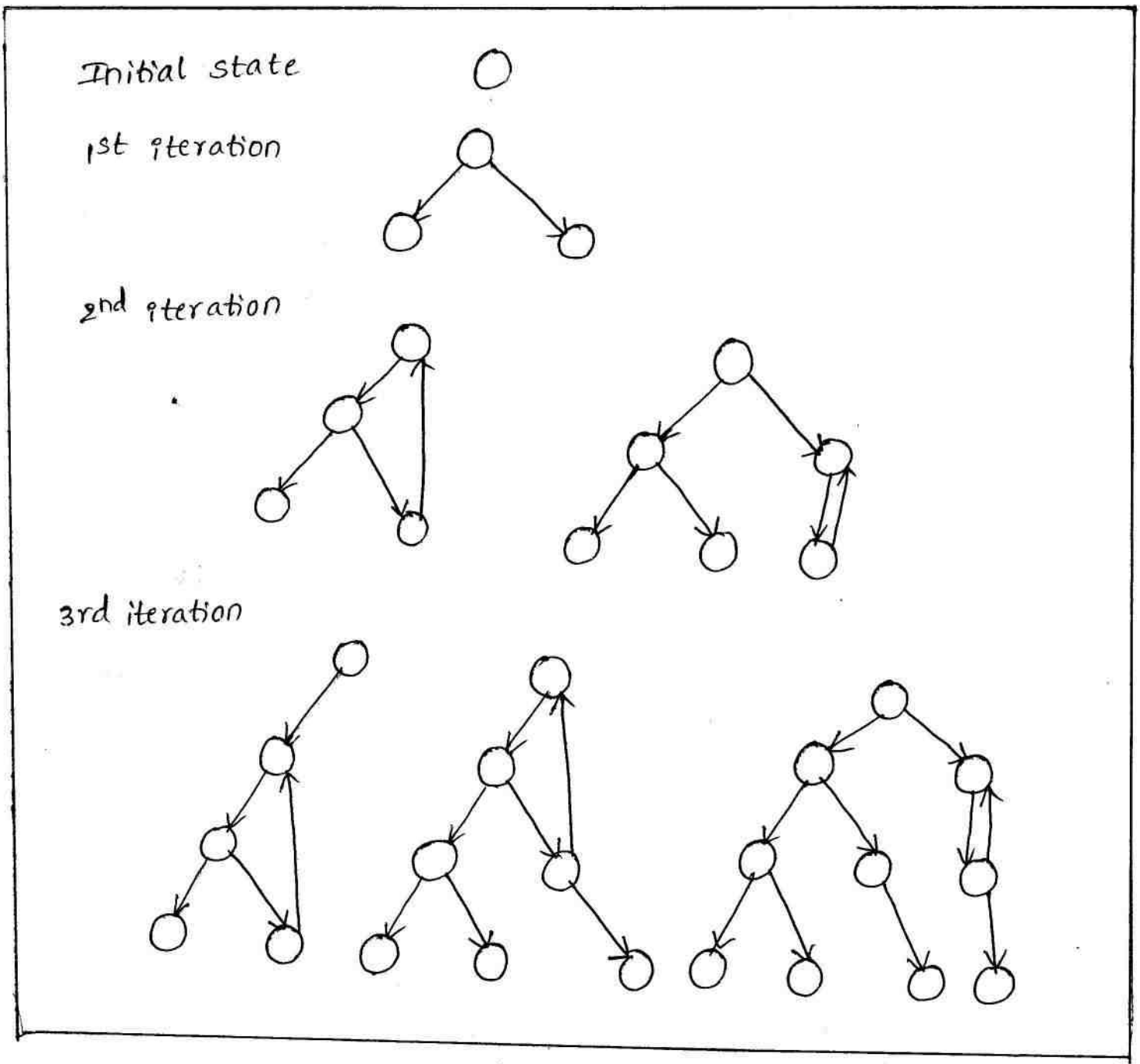


fig 2.6 :- Search tree Generation using DFID

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 noticed 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 $o(d^{b/2})$, and o($O(b^{d/2})$ 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.

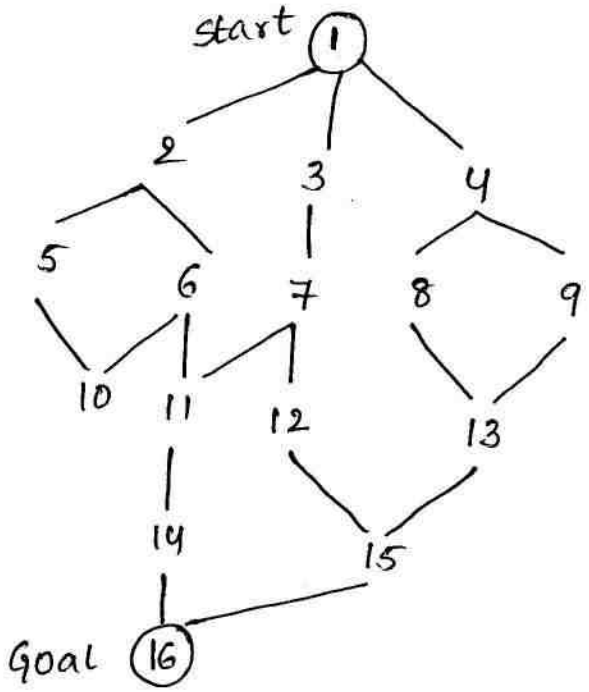


Fig:- 2.7 Graph to be searched using Birectional search

The trace of finding path from node 1 to 16 using birectional 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 gurantees a solution if it exists
- Time complexity:. Time required by an algorithm to find a solution.
- space complexity:. Space req. 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

Iteration	Bidirectional Tree
K=0	start 1
K=1 K=0	14 15 Goal 16
K=1	Start 1 2 3 4
K=2 K=1	11 12 13 14 15 Goal 16
K=2	start 1 2 3 4 5 6 7 8 9
K=3 K=2 K=1	11 14 Goal 16

Fig: 2.8 Trace of Bidirectional space

→ we will compare searches discussed above on those parameters. Let us assume 'b' to be the branching factor and 'd' to be the depth of the tree in 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.

BFS:- The total number of nodes generated in the worst case is

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

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 relavent problem information and finds solutions faster.

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

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 is 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 ~~impro~~ 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 ~~of~~ 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 = ∅;
  FOUND = false;
* while (open ≠ ∅ 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, compute their 'g' values and store them in OPEN list;
        - Store 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 augument 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 goto start else quit

End

2.5.3 Hill Climbing

Quality Measurement turns Depth-first search into Hill climbing. 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 depth-first order, but the choises are ordered according to some heuristic value. For example, in travelling salesman problem, straight line 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

  FOUND = false

* while (OPEN ≠ empty and FOUND = false) do

  {

  - remove the top element from OPEN list and call it NODE;
  - if NODE is 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, 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 situations, 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 in 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 breadth-first search. The beam width bounds the memory required to perform the search, at the expense of risking termination or completeness and optimality. The reason for such risk is that the goal state potentially might have pruned.

Algorithm (Beam Search)

Input: START and GOAL states

Local Variables: OPEN, NODE, SUCCs, W_OPEN, FOUND;

OUTPUT: Yes or NO

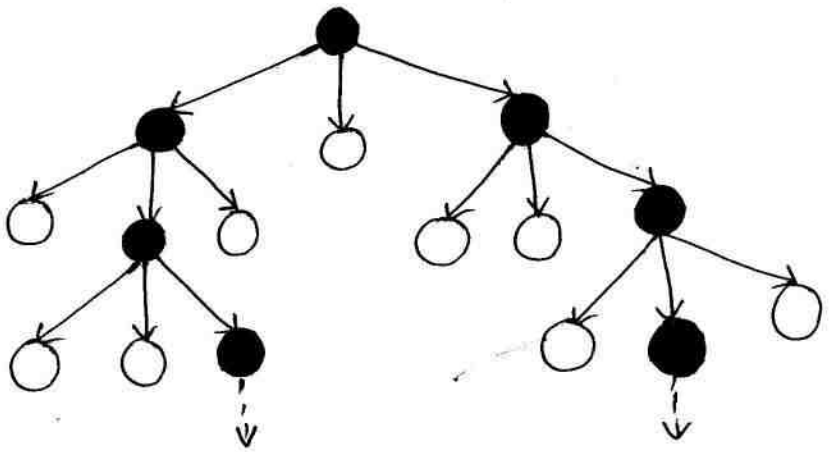
METHOD:

* 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.



continue till goal state is found or
not able to proceed ~~further~~

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:

* initialization 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 = ~~false~~ 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 the 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.

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 the 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 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. Unexpected leaf nodes of expanded nodes are stored in a queue with corresponding f values. this 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 ≠ ϕ 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 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 ∈ 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 the Best-Node and change the values of g and f for OLD else ignore;
}
- if succ ∈ 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 propagate the change to OLD's children

Using depth first search else ignore; (21)

}

- 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 and 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. Let us consider the following puzzle and try to solve 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

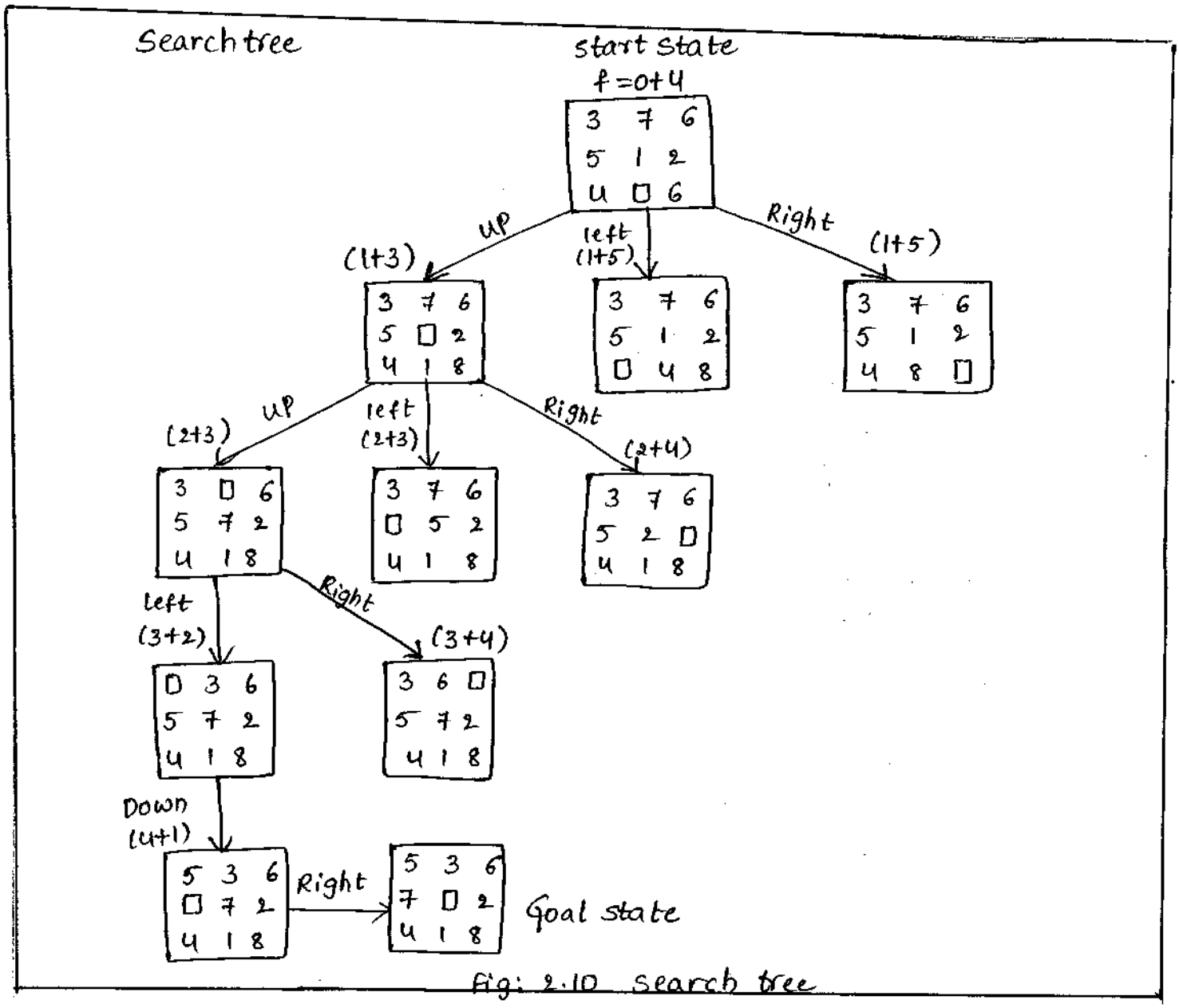


Fig: 2.10 search tree

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 titles (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.

Optimal Solution by A* algorithm

(22)

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 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 a 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 fartheraway than we thought. Now we go back and try another path and will find the optimal path.

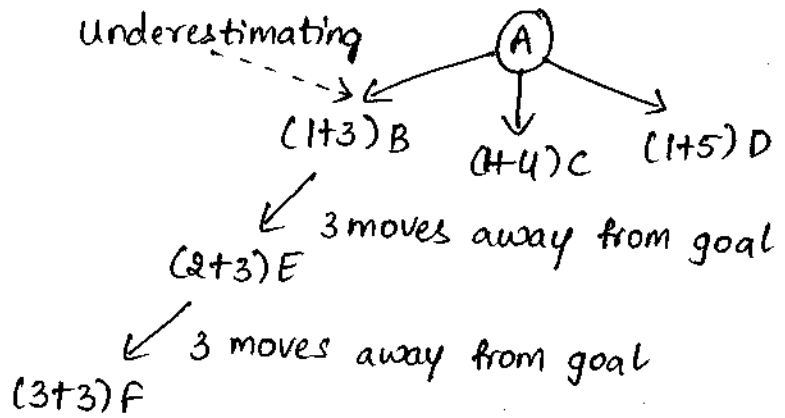


fig 2.11 Example search graph for underestimation

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.

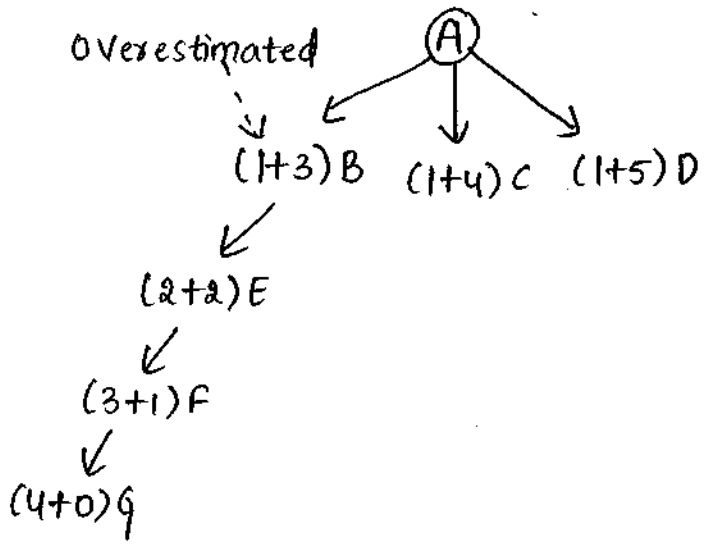


Fig 2.12:- Example search Graph for overestimation

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:

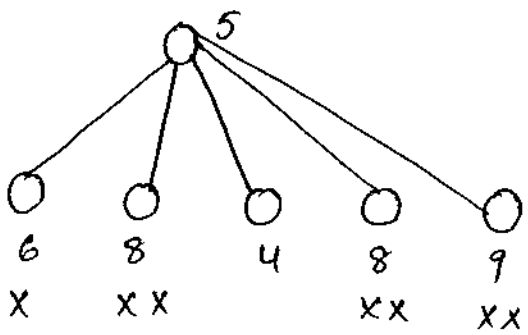
- For each iteration, perform a DFS pruning off a branch when its total cost (g+h) exceeds a given threshold.

(13)

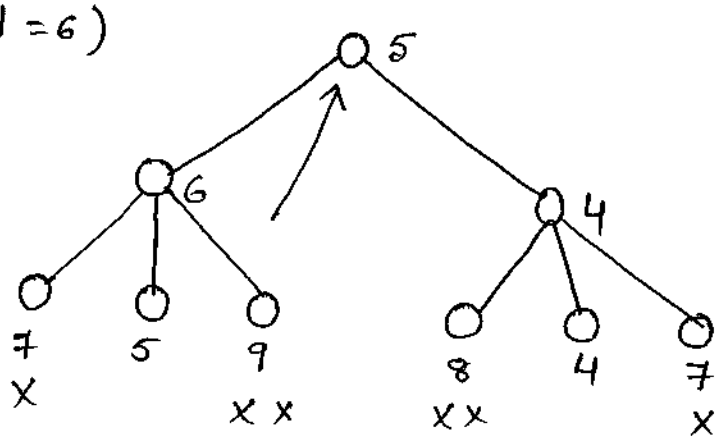
- The initial threshold starts at the estimate cost of the start state and increases for each iteration of the algorithm.
- The threshold used for the next iteration is the minimum cost of all values exceeded the current threshold.
- These steps are repeated till we find a goal state.

→ 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.

1st iteration (Threshold = 5)



2nd Iteration (Threshold = 6)



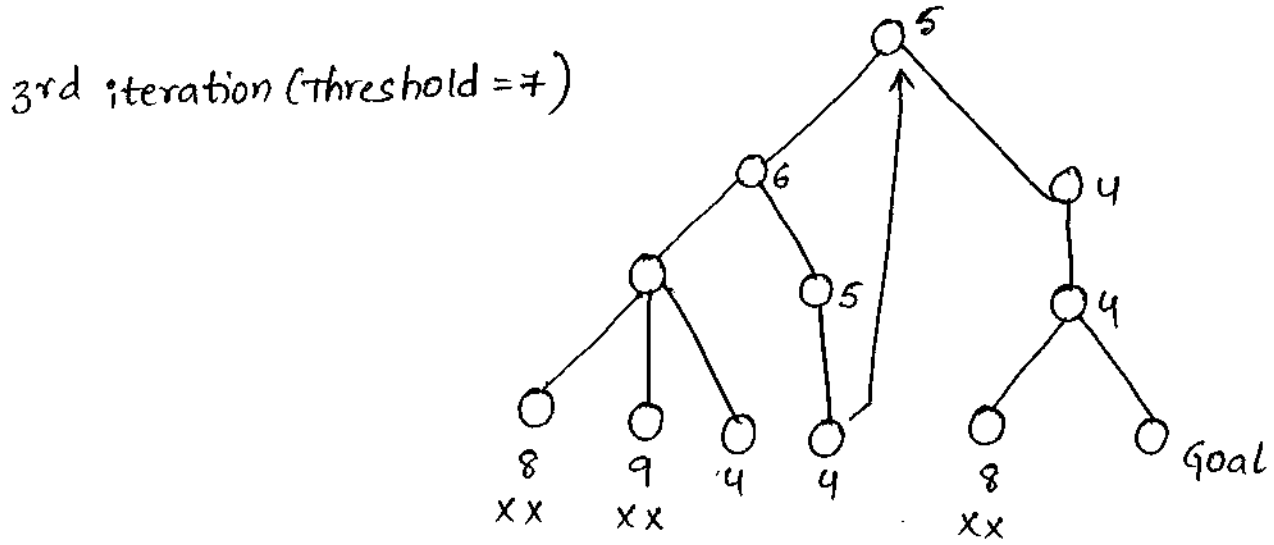


Fig 2.13 Working of IDA*

→ 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 no. 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 function. iterations.

Constraint satisfaction:

Many AI problems can be viewed as problems of constraint satisfaction in which the goal is to solve some problems 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 arthemetical operations has some or all of its digits replaced by letters

and the original digits must be found. In such a puzzle, each (24) 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 $\{x_1, x_2, \ldots x_n\}$, with each $x_i \in D_i$ 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, $1 \le i \le n$, a value of $x_i \in D_i$, 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 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 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 augumented 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 implementedas a DF search (Rich and Knight, 2003)

Algorithm:-

- until a complete solution is found or all paths have lead to dead ends
  {
  - select an unexpected node of the search algorit 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 numerical (0-9) in such a way that each letter is assigned unique digit which satisfy the following addition:

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

- constraints: No two letters have the same value.
- 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 constraints 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:

$E+L=S \longrightarrow C_1$

$S+L+C_1=E \longrightarrow C_2$

$2A+C_2=M \longrightarrow C_3$

$2B+C_3=A \longrightarrow C_4$

$G=C_4$

→ He can easily see that G has to be non-zero digit, so the value of carry C_4 should be 1 and hence G=1

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

Example:- let us solve another crypt-arithmetic puzzle

C_3	C_2	C_1	
	T	W	O
+	T	W	O
F	O	U	R

Constraints equations:

$2O=R \longrightarrow C_1$

$2W+C_1=U \longrightarrow C_2$

$2T+C_2=O \longrightarrow C_3$

$F=C_3$

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

(26)

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

$G = C_4$

$2B + C_3 = A \rightarrow C_4$

$2A + C_2 = M \rightarrow C_4$

$S + L + C_1 = E \rightarrow C_2$

$E + L = S \rightarrow C_1$

Initial state

$G = ?\ ; A = ?\ ; M = ?\ ; E = ?\ ;$

$S = ?\ ; B = ?\ ; L = ?$

1. $G = C_4 \Rightarrow \boxed{G = 1}$
2. $2B + C_3 = A \rightarrow C_4$

   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 $C_2 = 1$ X
   - If $B = 5$ → if $C_3 = 1 \Rightarrow A = 1$ X (AS $G = 1$ ready)
   - For $B = 6$ we get similar contradiction while generating the search tree.
   - If $\boxed{B = 7}$ then for $C_3 = 0$, we get $\boxed{A = 4}$ $\Rightarrow M = 8$ if $C_2 = 0$ that leads to contradiction later, so this path is pruned. If $C_2 = 1$, then $\boxed{M = 9}$
3. Let us solve $S + L + C_1 = E$ and $E + L = S$
   - Using both equations, we get $2L + C_1 = 0 \Rightarrow \boxed{L = 5}$ and $C_1 = 0$.
   - Using $L = 5$, we get $S + 5 = E$ that should generate carry $C_2 = 1$ as shown below. above.
   - so $S + 5 \geq 9 \Rightarrow$ possible values for E are $\{2, 3, 6, 8\}$ (with carry bit $C_2 = 1$)
   - If $E = 2$ then $S + 5 = 12 \Rightarrow S = 7$ (as $B = 7$ already)
   - If $E = 3$ then $S + 5 = 13 \Rightarrow S = 8$
   - therefore $\boxed{E = 3}$ and $\boxed{S = 8}$ are fixed up
4. Hence we get the final solution as given below on backtracking, we may find more solutions. In this case we get only one solution.

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

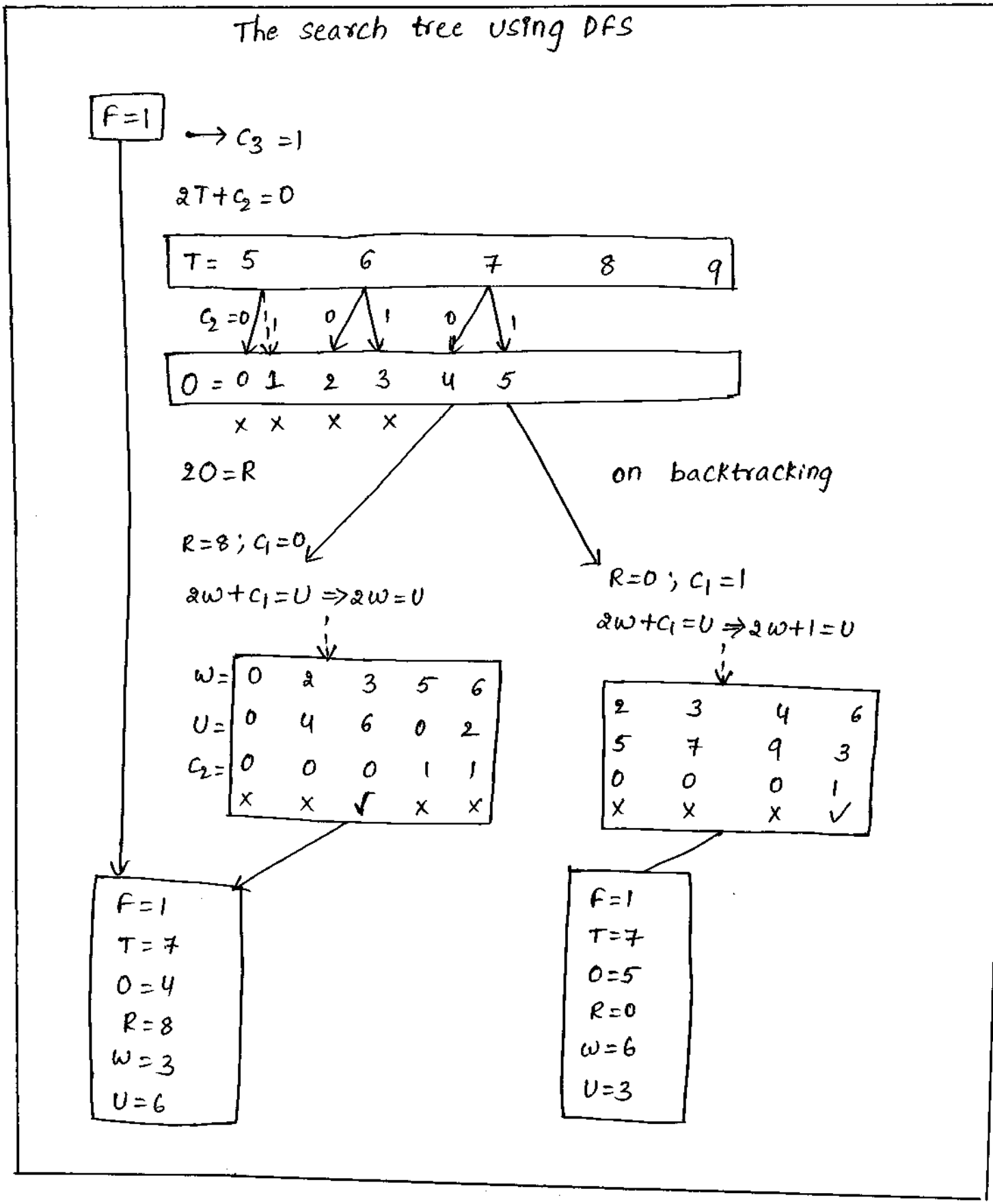
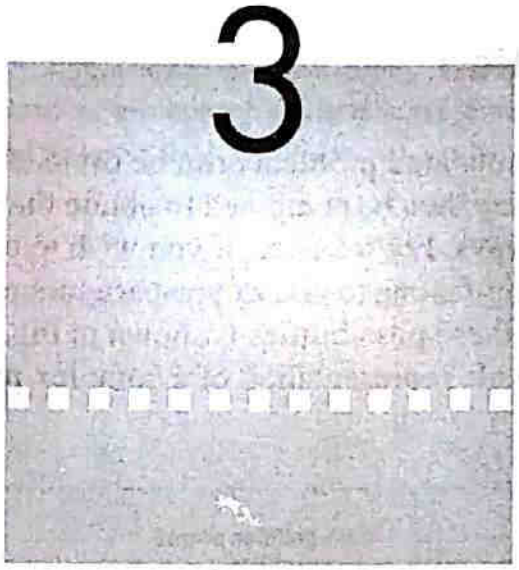


Fig 2.15 Search Tree for Crypt-Arithmetic puzzle



Problem Reduction and Game Playing

3.1 Introduction

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

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

3.2 Problem Reduction

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

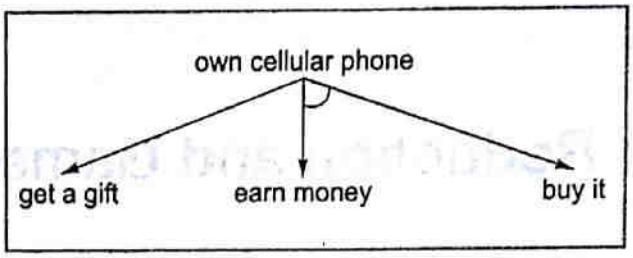


Figure 3.1 A Simple AND–OR Graph

Thus, this structure may prove to be useful to us in a number of problems involving real situations. To find a solution using AND–OR graphs, we need an algorithm similar to the 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-re tion concept. It consists of three rods and a number of disks of different sizes which can slide any rod [Paul Brna 1996]. The puzzle starts with the disks being stacked in descending orde their sizes, with the largest at the bottom of the stack and the smallest at the top, thus maki conical shape. The objective of the puzzle is to move the entire stack to another rod by using 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 anot 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 f rod_1 to rod_2 making use of rod_3. Let us develop an algorithm which shows that this prob can be solved by reducing it to smaller problems. Basically the method of recursion will be use solve this problem. The game tree that is generated will contain AND–OR arcs. The solution 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 smaller $n - 1$ disks in the same order from rod_1 to rod_3, and then move the largest disk fr

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.

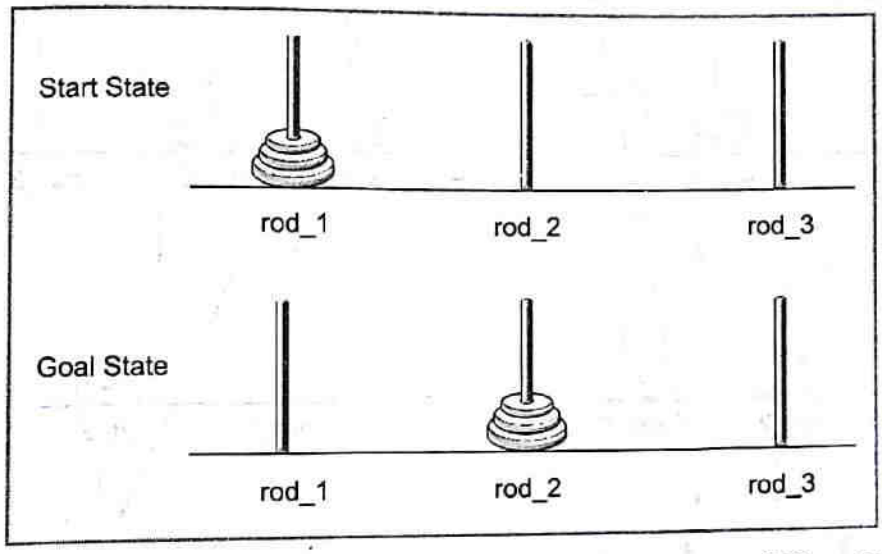


Figure 3.2 The Start and Goal States of a Three-disk *Tower of Hanoi* Problem

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.

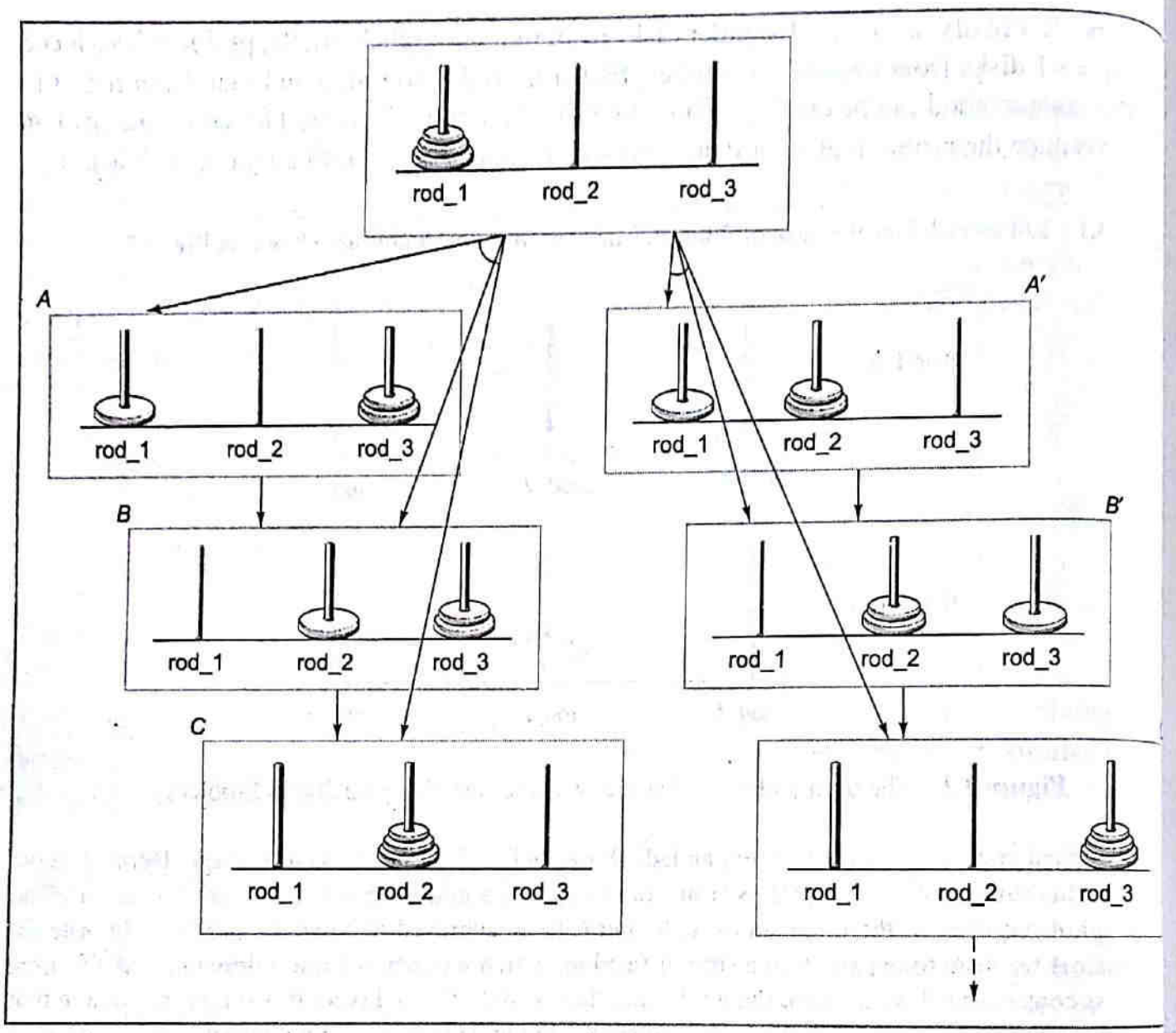


Figure 3.3 An AND–OR Graph for a Three-disk Problem

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

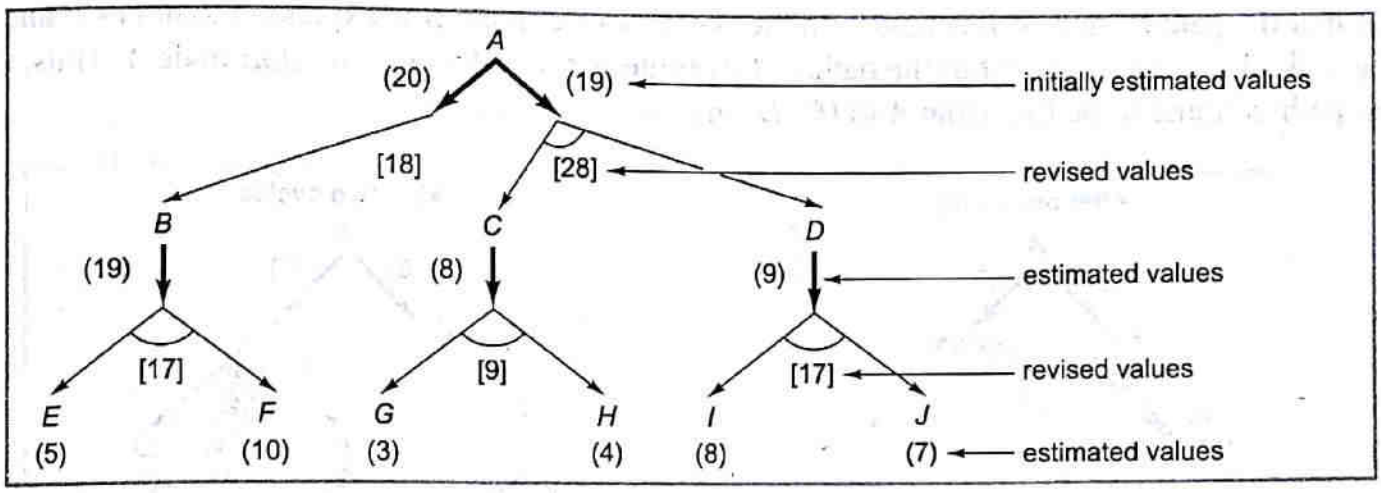


Figure 3.4 An Example of AND–OR Graph

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

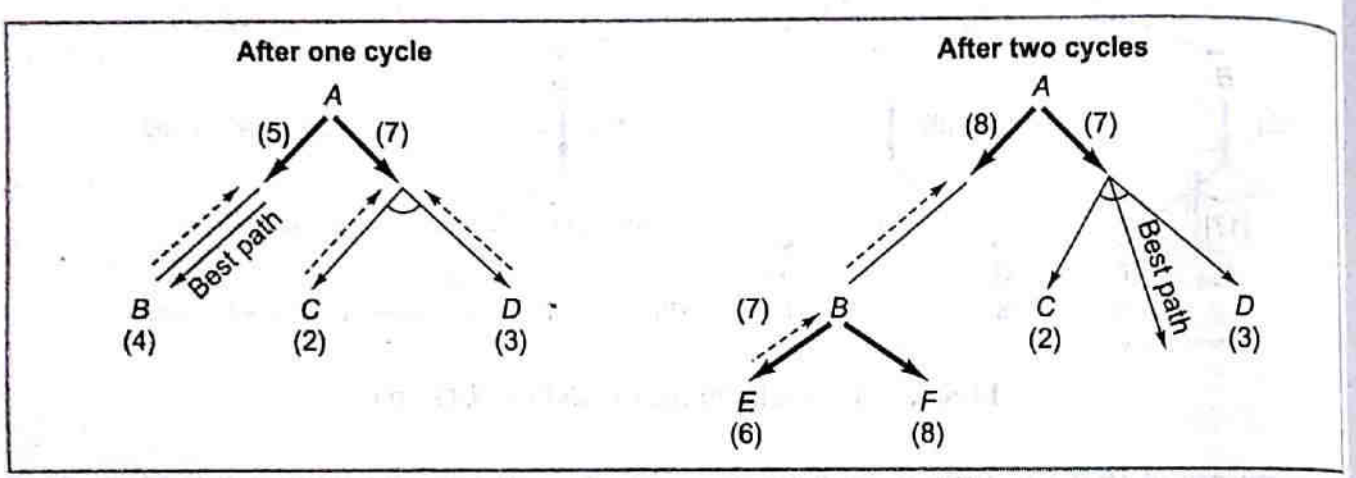


Figure 3.5 Labelling Procedure: First Two Cycles

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

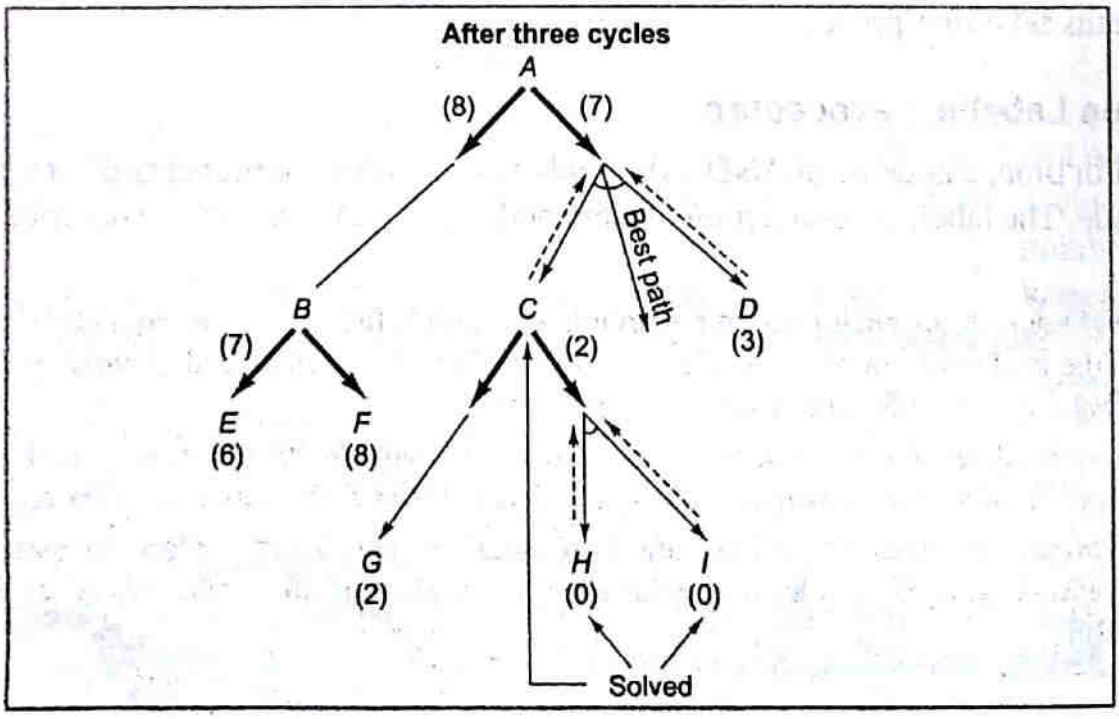


Figure 3.6 Labelling Procedure: Third Cycle

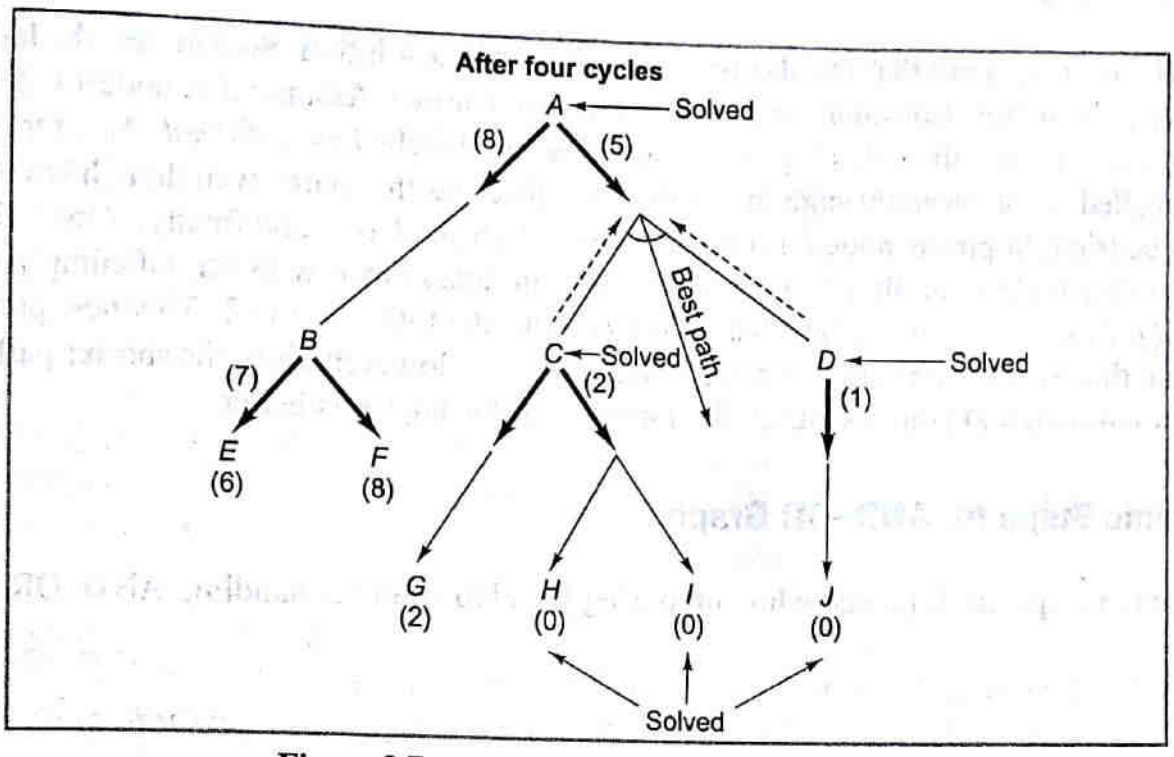


Figure 3.7 Labelling Procedure: Fourth Cycle

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).

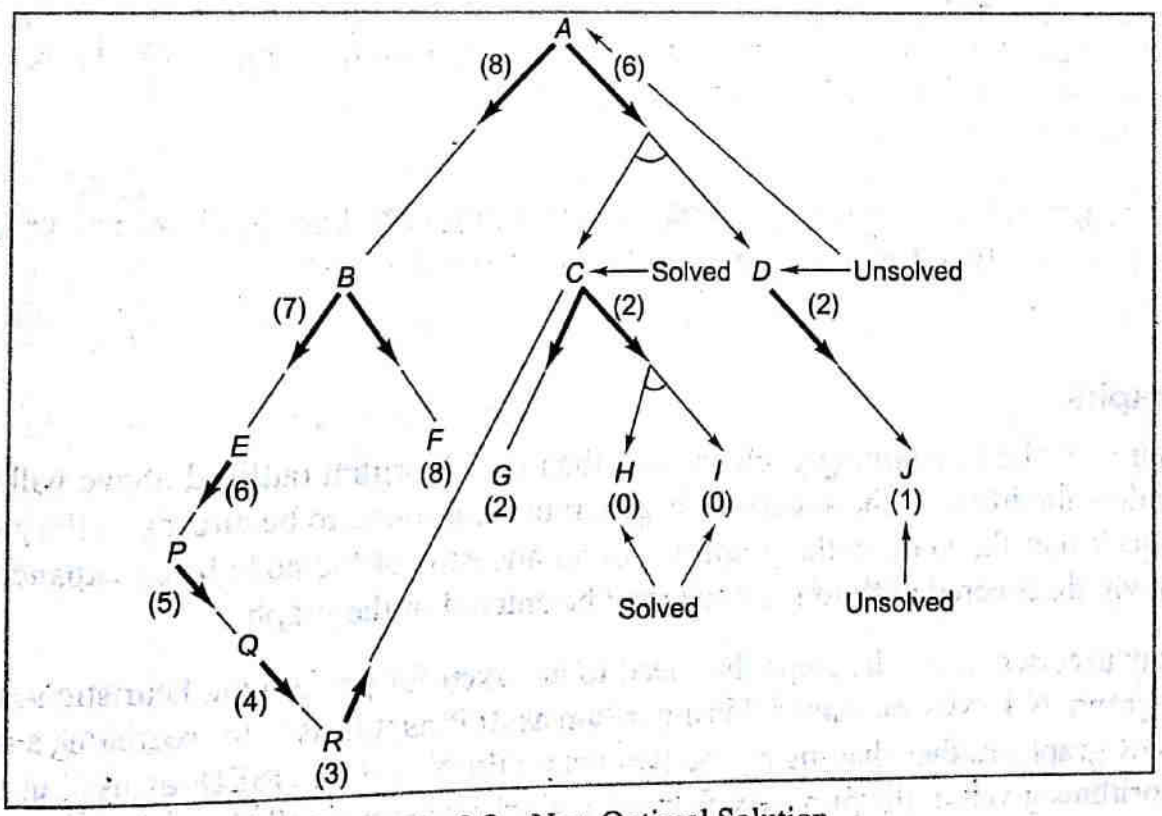


Figure 3.8 Non-Optimal Solution

It is not always necessary that the shorter path will lead to a solution. Sometimes, the longer pa may prove to be better. Consider the example discussed above. Assume that node J is labelled unsolved after the fourth cycle (Fig. 3.7), then D is also labelled as *unsolved*. As a result, A w also be labelled as *unsolved* through this path. Therefore, another path, even though having high cost, will be tried. Suppose, node E is expanded to P, P to Q, Q to R, and finally, R to C (Fig. 3. We notice that node C is already solved and then in accordance with the labelling procedu nodes R, Q, P, E, B, and A get labelled as *solved* with the total cost as 8. This new path to C longer than the previous one coming directly from A to C. However, since the shorter path will n 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 al paths))
  {
  - Traverse the graph along the best path and expand all unexpande 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 chang 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 wo properly unless modified. If the successor is generated and found to be already in the graph, th we must check that the node in the graph is not an ancestor of the node being expanded. If no 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 AND–OR graph. N.J. Nilsson named this algorithm as AO^* as it is used for searching a solution an AND–OR graph. Rather than using the two lists, OPEN and CLOSED, as used in OR gra 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. Ea

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 (ob. tained 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 bel

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 ances. tor 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 en 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-go which may lead to non-optimal solution. Let us explain the need of interaction between sub-goa

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

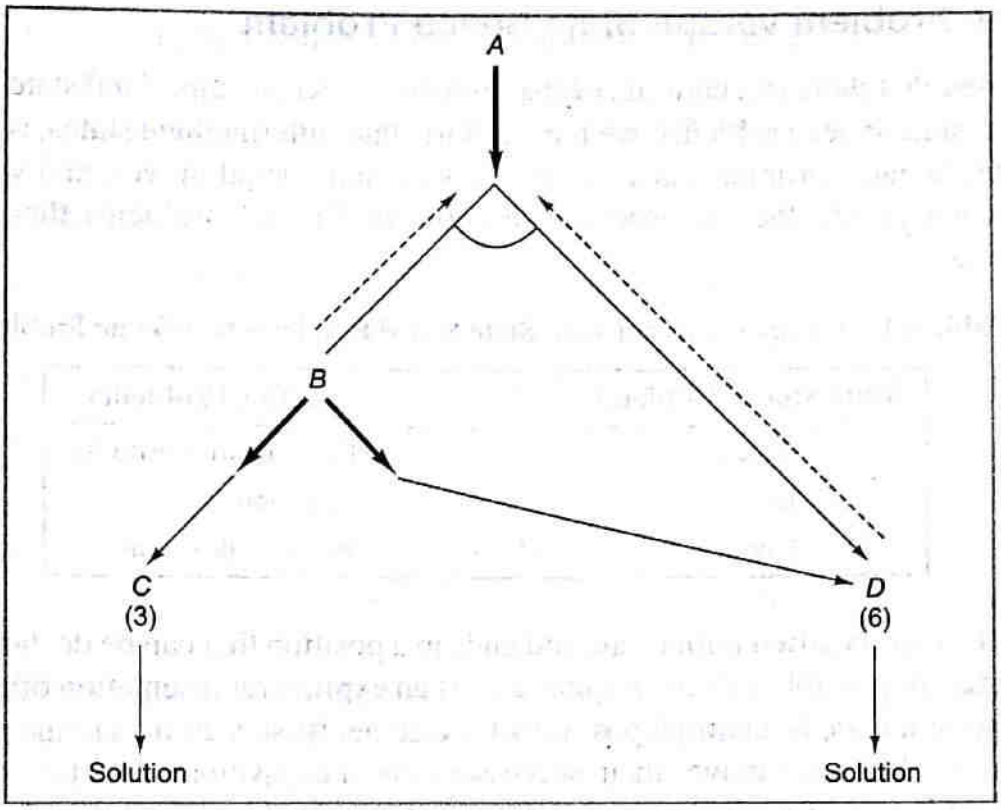


Figure 3.9 Interaction between Sub-Goals

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).

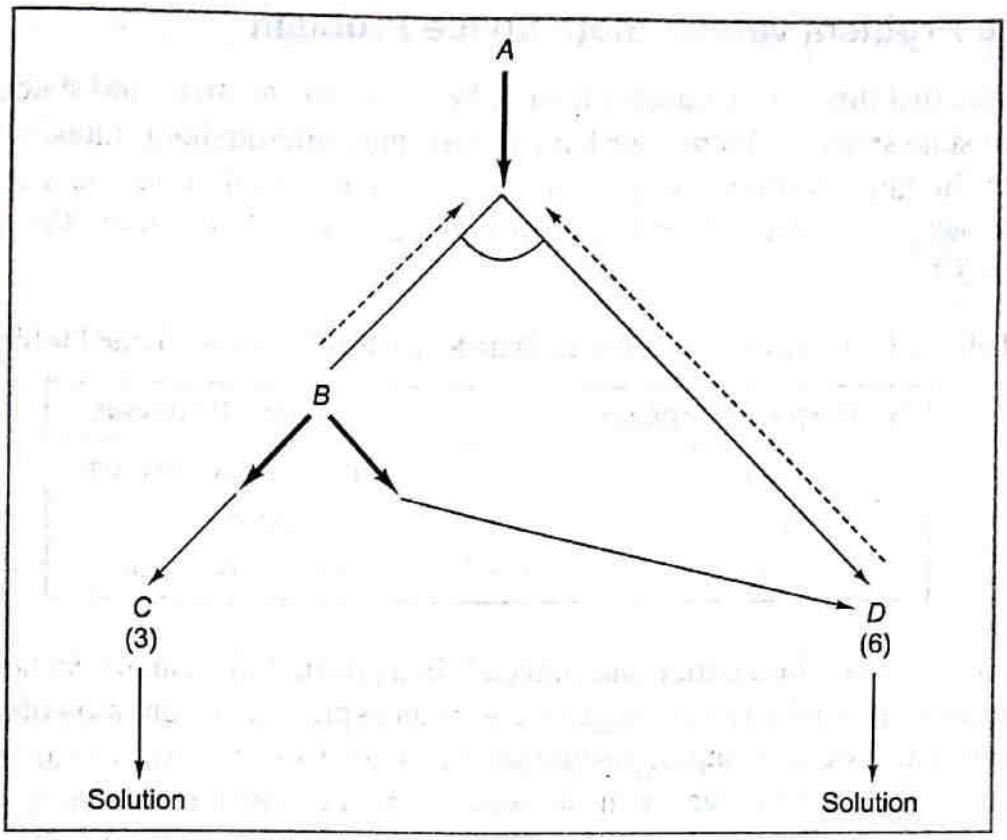


Figure 3.9 Interaction between Sub-Goals

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 proble For example, in state space problems, we have a start state, intermediate states, rules or operat and a goal state. In game problems also we have a start state, legal moves, and winning positi (goals). To further clarify the correspondence between the two problems the comparisons 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 a *loss* for the other, or possibly a *draw*. A *game tree* is an explicit representation of all possible p of the game. The root node is an initial position of the game. Its successors are the positions tha first player can reach in one move; their successors are the positions resulting from the sec player's moves and so on. Terminal or leaf nodes are represented by WIN, LOSS, or DRAW. E 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 av able to one player from a given position can be represented by the OR nodes, whereas the mo available to his opponent are the AND nodes. Therefore, in the game tree, one level is treated an OR node level and other as AND node level from one player's point of view. On the other h 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 maxim ing the minimum gain. In game playing involving computers, one player is assumed to be computer, while the other is a human. During a game, two types of nodes are encountered, name MAX and MIN. The MAX node will try to maximize its own game, while minimizing the opp nent's (MIN) game. Either of the two players, MAX and MIN, can play as the first player. We assign the computer to be the MAX player and the opponent to be the MIN player. Our aim i make the computer win the game by always making the best possible move at its turn. For this have to look ahead at all possible moves in the game by generating the complete game tree then decide which move is the best for MAX. As a part of game playing, game trees labelled 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 poin the game. The leaf nodes are labelled as WIN, LOSS, or DRAW depending on whether t represent a win, loss, or draw position from MAX's point of view. Once the leaf nodes assigned their WIN–LOSS–DRAW status, each non-terminal node in the game tree can

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.

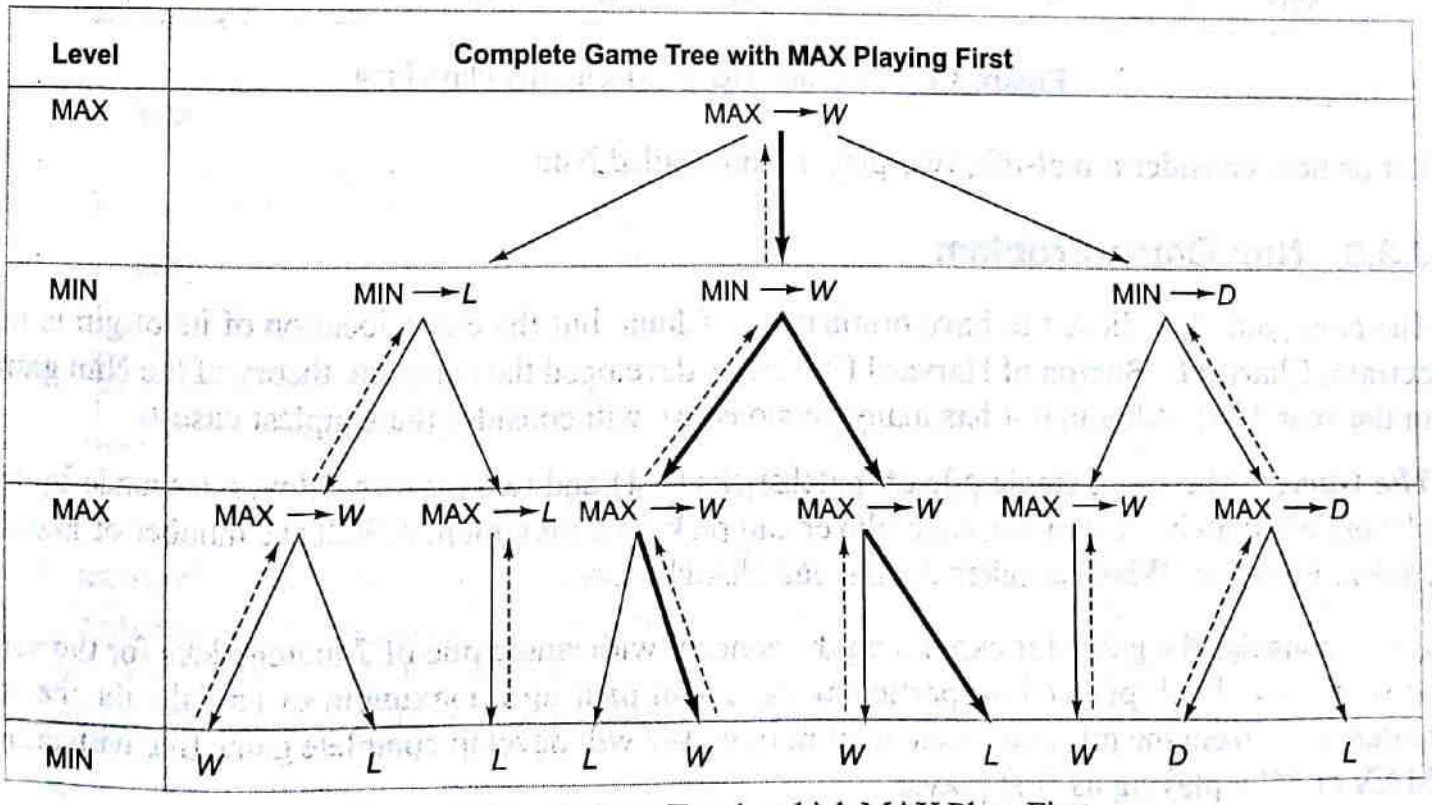


Figure 3.10 A Game Tree in which MAX Plays First

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

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

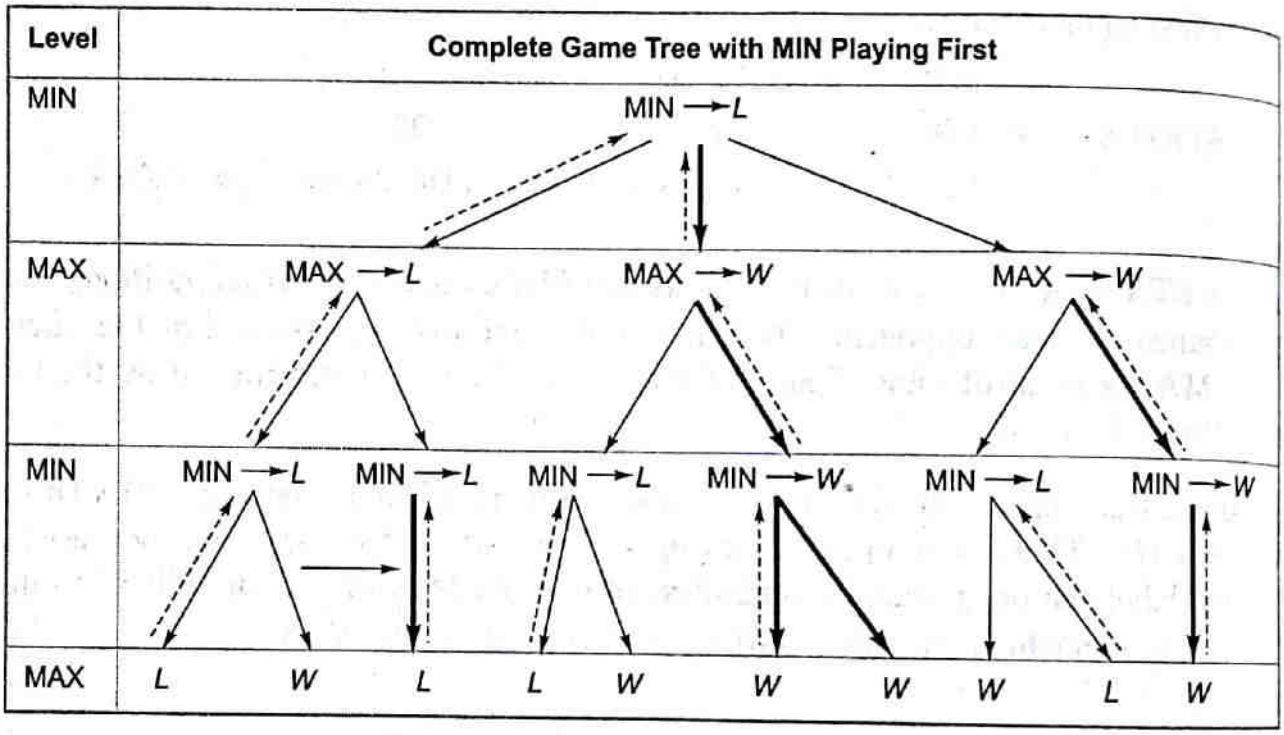


Figure 3.11 A Game Tree in which MIN Plays First

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
certain. Charles L. Bouton of Harvard University developed the complete theory of the Nim
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
players alternately. In a move, each player can pick up a maximum of half the number of
sticks 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
of simplicity. Each player in a particular move can pick up a maximum of half the numb
matchsticks from the pile at a given point in time. We will develop complete game tree with
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.

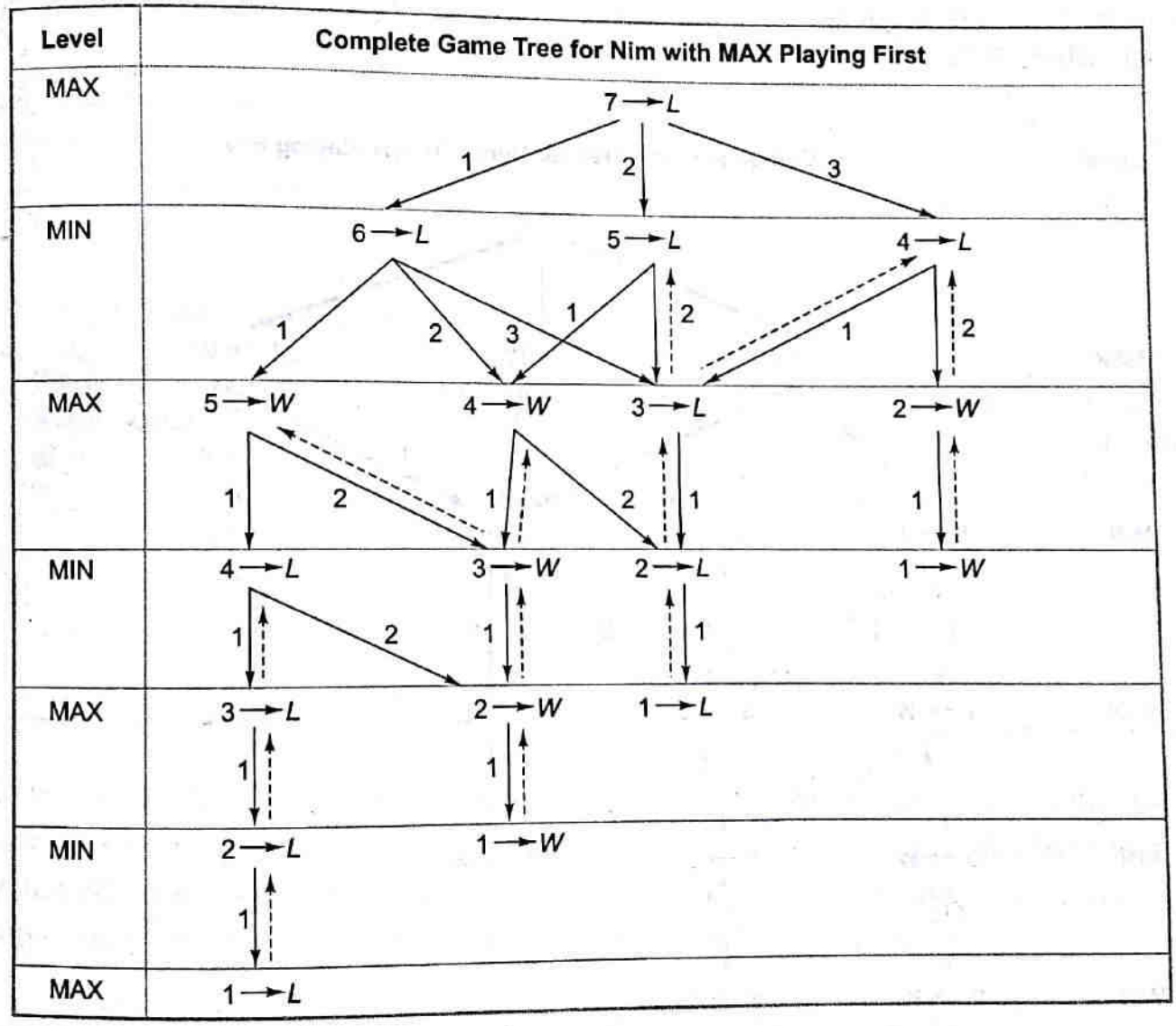


Figure 3.12 Game Tree for Nim in which MAX Plays First

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 one game-playing strategies with respect to a computer. In this case, MAX player is considered to be 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 $M \in \{1, 3, 7, 15, 31, \ldots\}$ 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 $2X_{i-1} + 1$, where $X_0 = 1$ for $i > 0$.

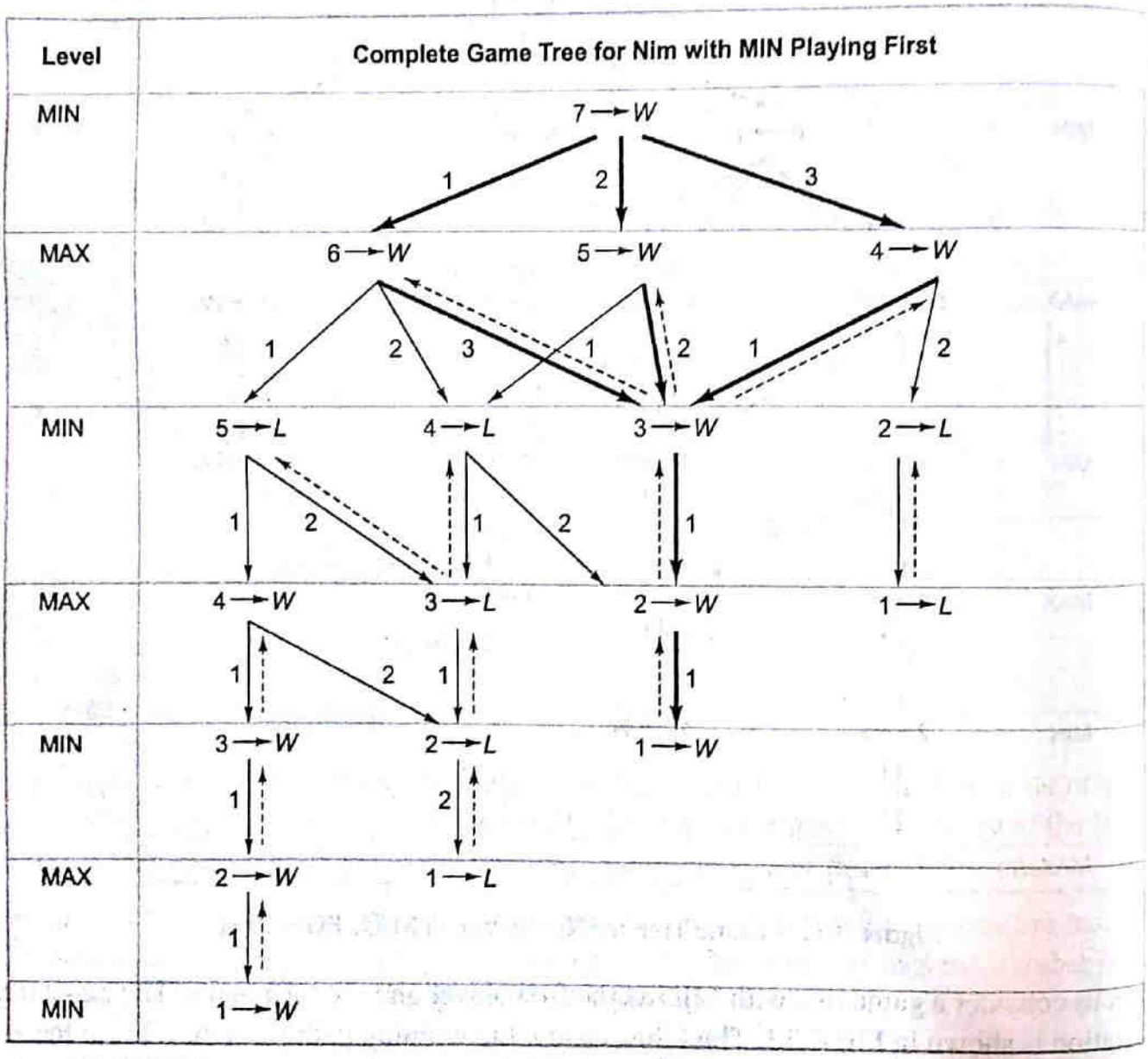


Figure 3.13 Game Tree for Nim in which MIN Plays First

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

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

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

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

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

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

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

Validity of Cases for Winning of MAX Player

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

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

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

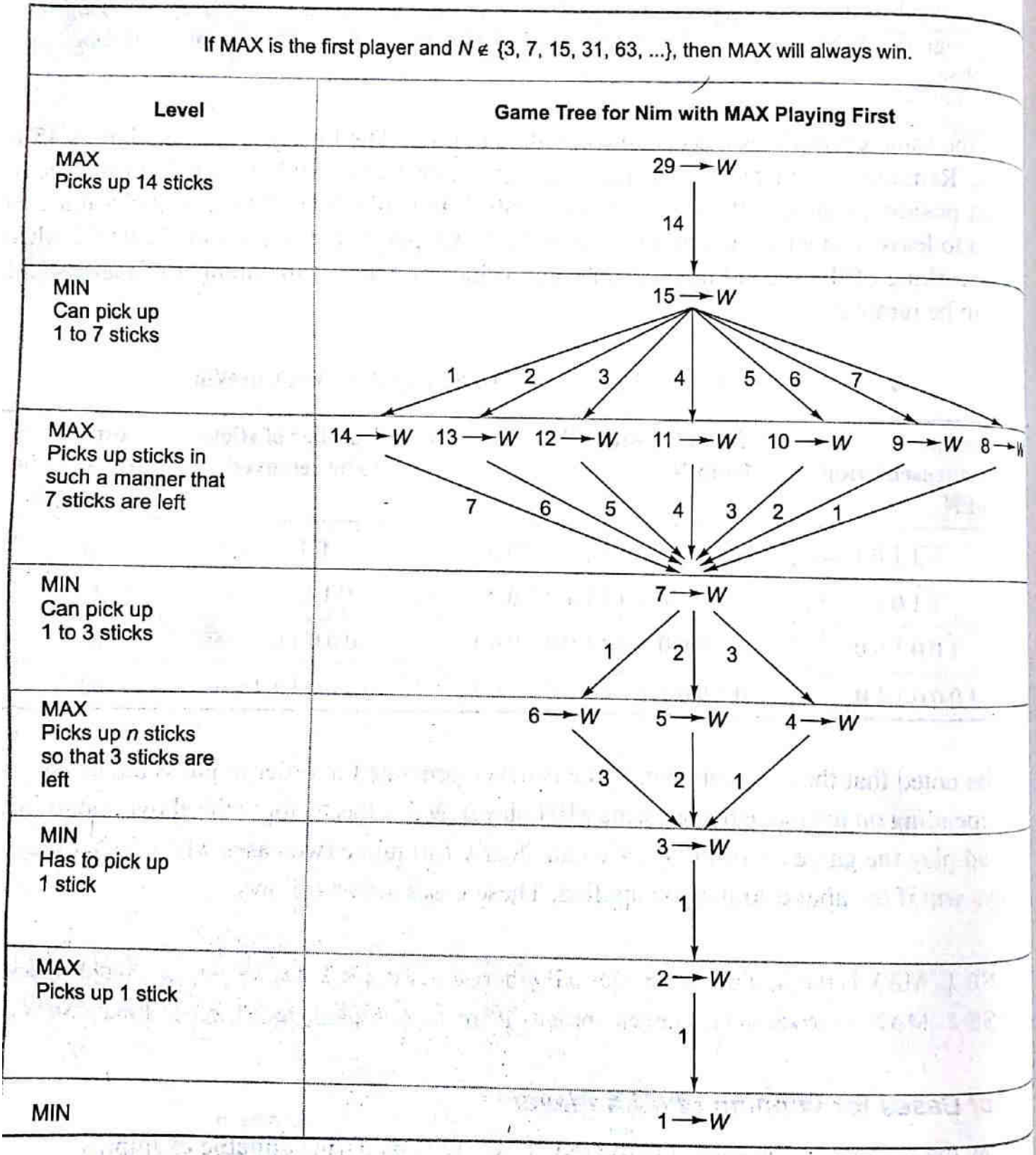


Figure 3.14 Validity of Case 1 (Example for $N = 29$)

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

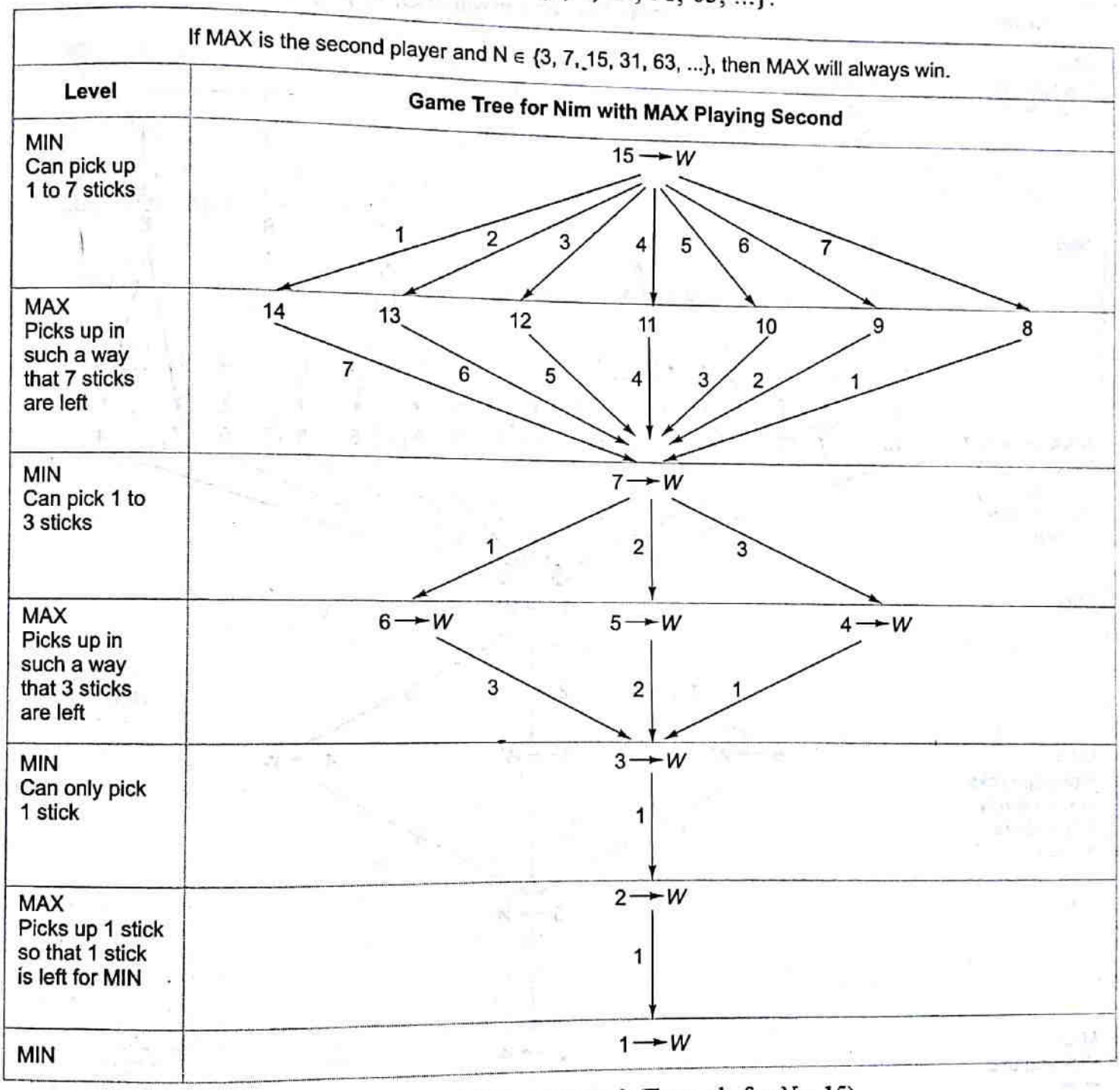


Figure 3.15 Validity of Case 2 (Example for $N = 15$)

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 $M \in \{1, 3, 7, 15, 31, 63, ...\}$.

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 chanc winning from this position relative to other similar positions. The best evaluation function 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 acc calculations. This function provides numerical assessment of how favourable the game state MAX. We can use a convention in which a positive number indicates a good position for M while a negative number indicates a bad position. The general strategy for MAX is to play in a manner that it maximizes its winning chances, while simultaneously minimizing the chanc the opponent. The heuristic values of the nodes are determined at some level and then the sta accordingly propagated up to the root. The node which offers the best path is then chosen to a move. For the sake of convenience, let us assume the root node to be a MAX node. Conside one-ply and two-ply games shown in Fig. 3.19; the score of leaf nodes is assumed to be calcu using evaluation functions. The values at the nodes are backed up to the starting position.

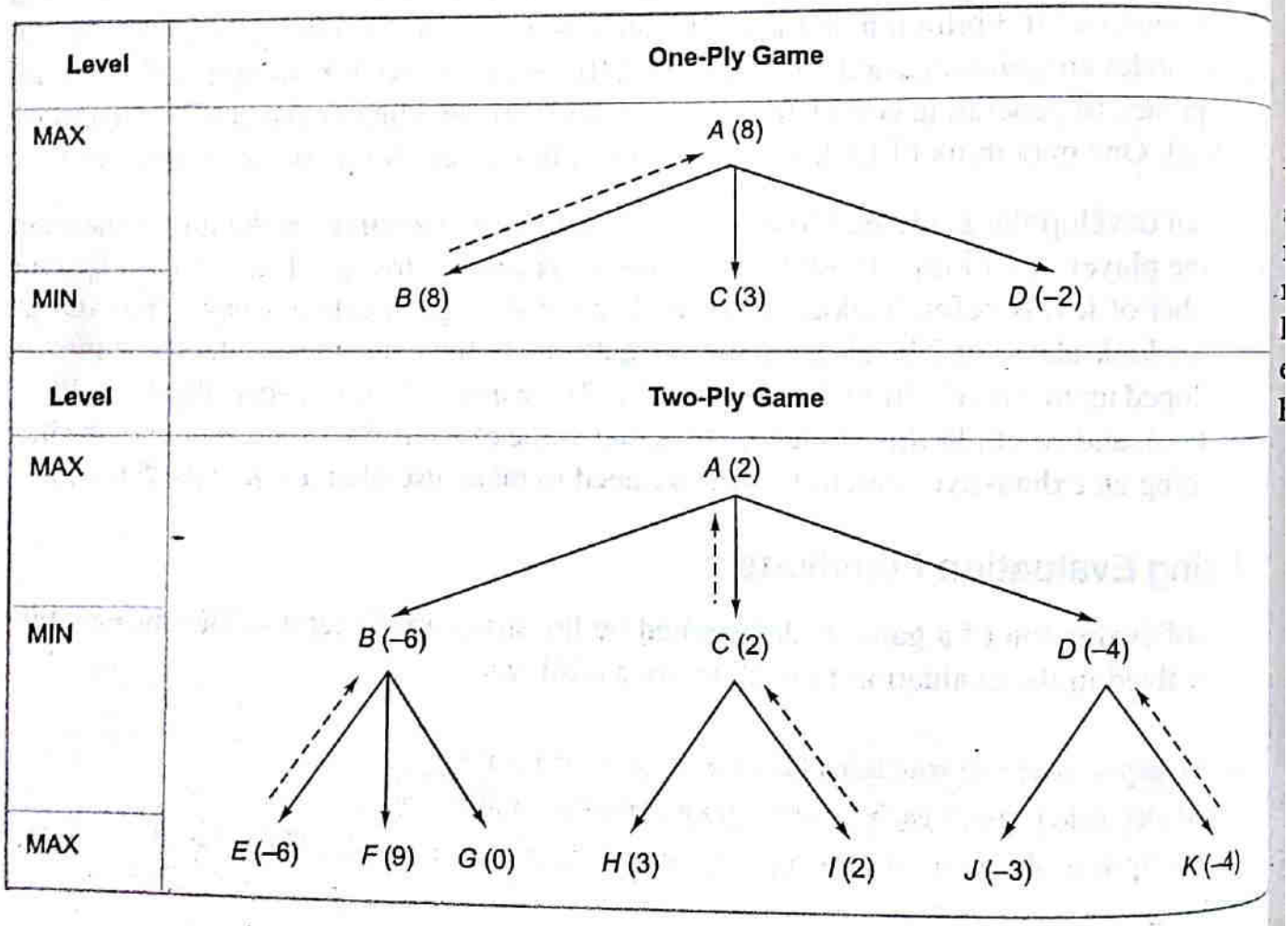


Figure 3.19 Using Evaluation Functions in One-Ply and Two-Ply Games

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

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

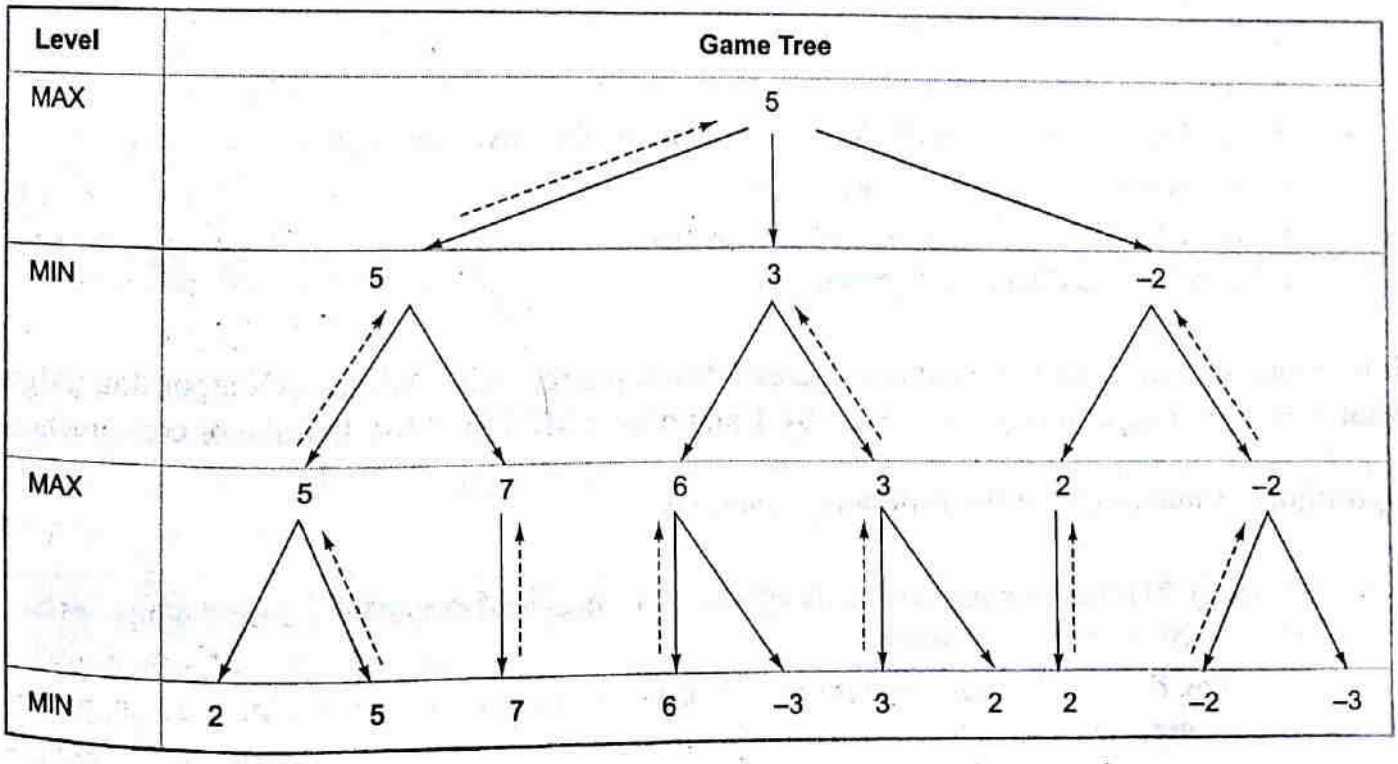


Figure 3.20 A Game Tree Generated using MINIMAX Procedure

using MINIMAX procedure up to a depth of three. At MAX level, maximum value of its succe 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 t MIN will get a chance to play a move. Whenever MAX gets a chance to play, it will genera game tree of depth 3 from the state generated by MIN player in order to decide its next move. process will continue till the game ends with, hopefully, MAX player winning. The algorith steps of a MINIMAX procedure can be written in the following manner [Rich & Knight 200

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 t current position.
- Compute the static value of the leaf nodes at depth *d* from the current position of the game tree usi 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 (Al rithm 3.3). Let us represent player MAX by 1 and player MIN by 0 for the sake of convenien

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 variable corresponding to position.
- EVAL (*Pos, Player*): This function returns a number representing the goodness of *Pos* for the play 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.

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.

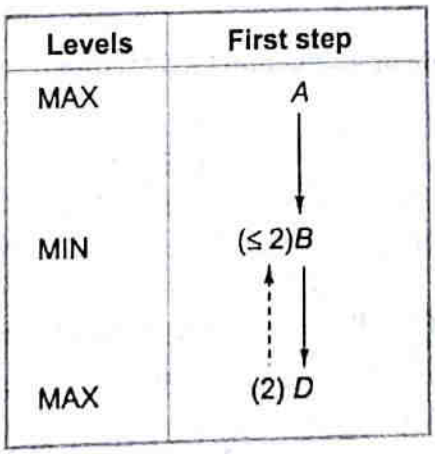


Figure 3.22 α–β Pruning Algorithm: Step I

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 bo 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 sec step as shown in Fig. 3.23. The state E gets $\alpha = 7$ and since there is no further successor (assumed), the β value at state B becomes equal to 2. Once the β value is fixed at MIN level, 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 α From Fig. 3.24 we note that the value at state C is ≤ 1 and the value of a root A cannot be less 2; the path from A through C is not useful and thus further expansion of C is pruned. There there is no need to explore the right side of the tree fully as that result is not going to alter the decision. Since there are no further successors of A (assumed), the value of root is fixed as 2, is, $\alpha = 2$.

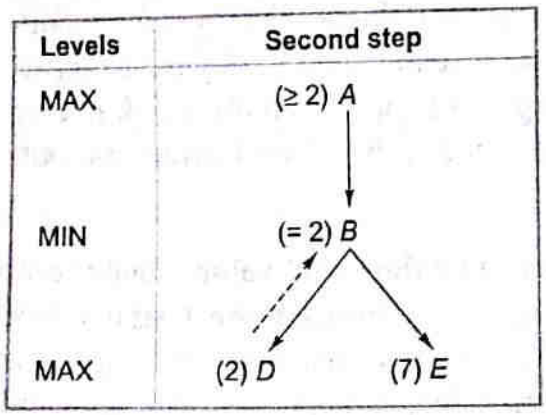


Figure 3.23 α–β Pruning Algorithm: Step 2

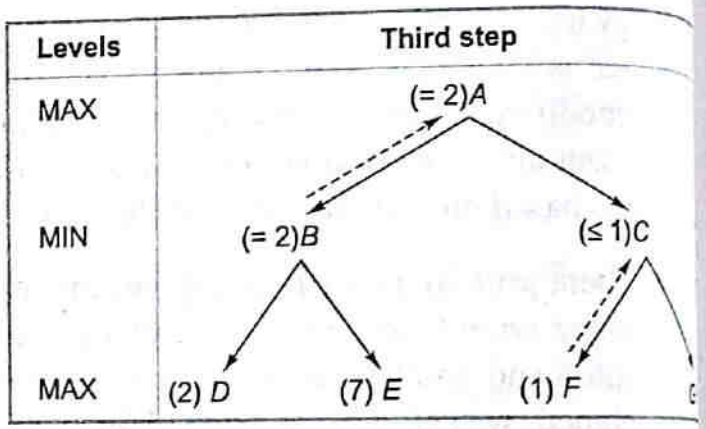


Figure 3.24 α–β Pruning Algorithm: Step 3

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

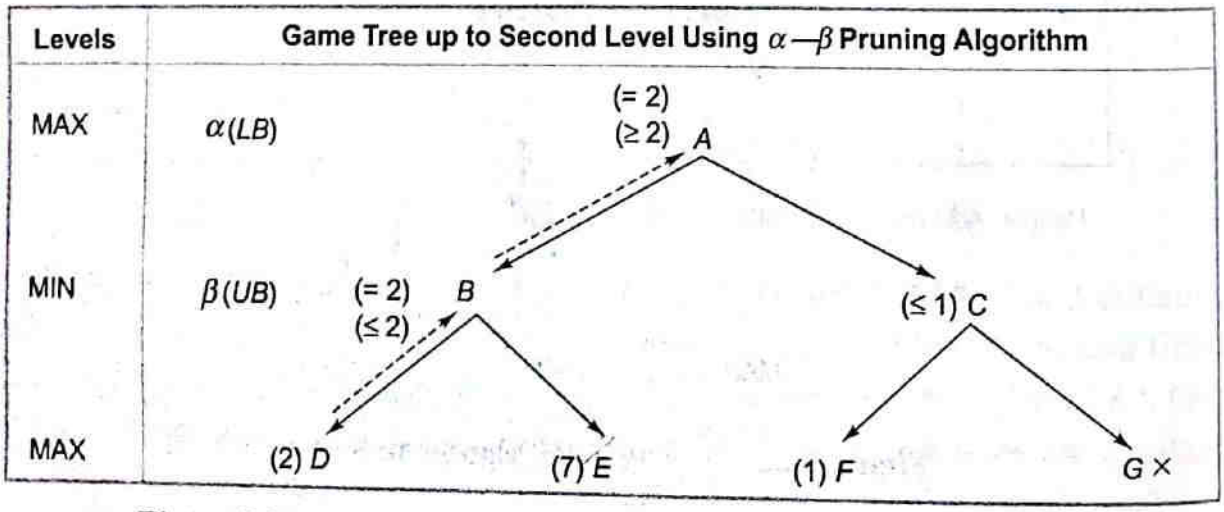


Figure 3.25 Game Tree Generation using α–β Pruning Algorithm

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

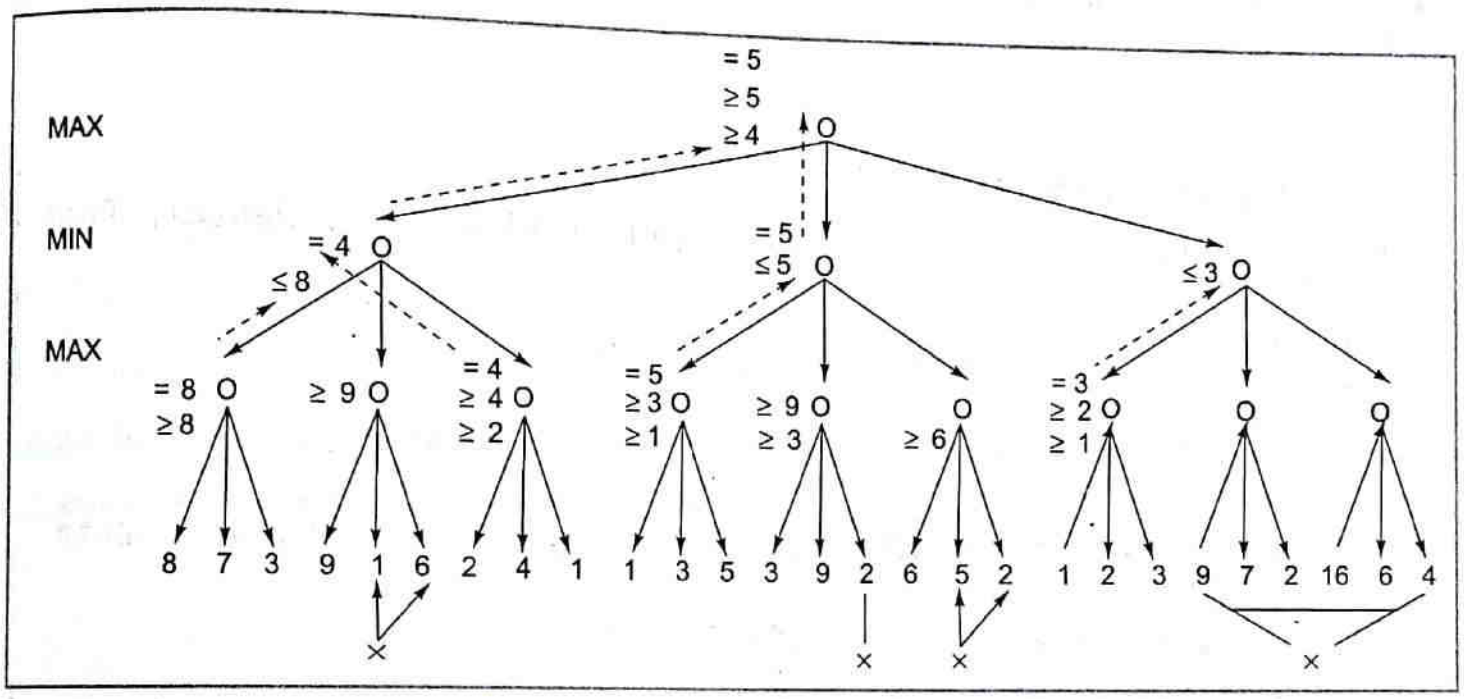


Figure 3.26 A Game Tree of Depth 3 and Branching Factor 3

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

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

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

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

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

3.6 Two-Player Perfect Information Games

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

Chess

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

Checkers

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

Othello

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

Go

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

Backgammon

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