VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

UNIT - II
Artificial Intelligence

III / II IT, R 16 - JNTUK

Ms. M. Rajya Lakshmi
VVIT

Contents

- **Problem solving: state-space search and control strategies:**
  - Introduction 23
  - General problem solving 24
  - Characteristics of problem 32
  - Exhaustive searches 34
  - Heuristic search techniques 44
  - Iterative deepening A* 57
  - Constraint satisfaction 57
- **Problem reduction and game playing:**
  - Problem reduction 66
  - Game playing 75
  - Alphabeta pruning 93
  - Two-player perfect information games 99

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
- This is one of the focused areas AI and can be characterized as a systematic search using a range of possible steps to achieve some solution
- In AI, the problems are frequently modelled as a state space problem where the **state space** is a set of all possible states from start to goal states
- A 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

- **Traditional Search:** Searches are applied to the existing graphs
- **AI based Search:** States of the graph are generated as they are explored for the solution path
- Types of Problem-Solving methods:
  - **General Purpose**: Applicable to wide variety of problems
  - **Special Purpose**: Tailor-made for a particular problem and exploits very specific features of the problem.

- The most general approach for solving problem is to generate solution and test it.
- For generating new state in search space, an action/operation/rule is applied and tested whether it is **goal state** or not.
- The order of application of the rules to the current state is called ***control strategy***.

General problem solving

The following sections describe facilitating the modelling of problems and search processes

- **Production system**
  - **Water jug problem**
  - **Missionaries and cannibals problem**
- **State – Space search**
  - **Eight puzzle problem**
- **Control strategies**

Production System (PS)

- PS helps AI programs to do search process more conveniently in state-space problems
- This systems consists of start and goal states along with DBs that consists related info.
- PS consists of no.of production rules in which each production rule has left and a right side
- Left side determines the applicability of rule
- Right side describes the action to be performed if the rule is applied
- Rule's left side is current state & right side describes the new state that is obtained by applying the rule

- PS also consists of Control Strategies that specify the sequence of rule applied
- In addition to usefulness of PS in describing search, following are other advantages:
  - A good way to model the strong state-driven nature of intelligent action
  - New rules can be easily added to a/c for new situations without disturbing rest of system
  - It is important in RT applications where new i/p to the DB changes the behaviour of the system

Water jug problem

Problem statement:

- We have two jugs, a 5 Lt. and a 3 Lt. without measuring markers on them.
- There is endless water supply through tap
- Our task is to get 4 Lt. of water in the 5 Lt. jug

Solution

- State space for this problem can be described as the set of ordered pairs of integers (X,Y)
  - X – no.of Lt. of water in 5 Lt. jug
  - Y – no.of Lt. of water in 3 Lt. jug
- Operations are defined as production rules as in Table

Production rules for water jug problem

Rule No.	Left of rule	Right of rule	Description
1	(X, Y | X < 5)	(5, Y)	Fill 5 Lt. jug
2	(X, Y | X >0)	(0, Y)	Empty 5 Lt. jug
3	(X, Y | Y < 3)	(X, 3)	Fill 3 Lt. jut
4	(X, Y | Y > 0)	(X, 0)	Empty 3 Lt. jug
5	(X, Y | X +Y <= 5 & Y > 0)	(X+Y, 0)	Empty 3 Lt. into 5 Lt. jug
6	(X, Y | X + Y <= 3 & X > 0)	(0, X+Y)	Empty 5 Lt. into 3 Lt. jug
7	(X, Y | X + Y >= 5 & Y > 0)	(5, Y – (5-X)) Until 5 Lt. jug is full	Pour water from 3 Lt. jug into 5 Lt. jug
8	(X, Y | X + Y >= 3 & X > 0)	(X – (3 - Y), 3)	Pour water from 5 Lt. jug into 3 Lt. jug is full

Solution path 1

Rule applied	5 Lt. jug	3 Lt. 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	-	

Solution path 2

Rule applied	5 Lt. jug	3 Lt. 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
- If the cannibals outnumber the missionaries, they will be killed by cannibals
- How can they all cross over safely

Solution

- State space of the problem can be described as the set of ordered pairs of left and right banks of the river as (L, R)
- Each bank is represented as a list [nM, mC, B]
  - n- no.of missionaries M
  - m- no.of cannibals C
  - B- boat
- The table consists of production rules based on the chosen representation
- One of the possible solution path trace is given in table

1. Start state: ([3M, 3C, 1B], [0M, 0C, 0B])
   - 1B means the boat is present & 0B means absent
2. Any state: ([n1M, m1C, _], [n2M, m2C, _]),

   with constraints/ conditions at any state as

   n1(!=0)>=m1; n2(!=0)>=m2; n1+n2=3,

   m1+m2=3; boat can be either side
3. Goal state: ([0M, 0C, 0B], [3M, 3C, 1B])

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	([n1M, m1C, 1B], [n2M, m2C, 0B])	->	([(n1-2)M, m1C, 0B], [(n2+2)M, m2C, 1B])
L2	([n1M, m1C, 1B], [n2M, m2C, 0B])	->	([(n1-1)M, (m1-1)C, 0B], [(n2+1)M,(m2+1)C, 1B])
L3	([n1M, m1C, 1B], [n2M, m2C, 0B])	->	([n1M, (m1-2)C, 0B], [n2M,(m2+2)C, 1B])
L4	([n1M, m1C, 1B], [n2M, m2C, 0B])	->	([(n1-1)M, m1C, 0B], [(n2+1)M, m2C, 1B])
L5	([n1M, m1C, 1B], [n2M, m2C, 0B])	->	([n1M, (m1-1)C, 0B], [n2M,(m2+1)C, 1B])
Rules for boat coming from right bank to left bank of the river			
R1	([n1M, m1C, 0B], [n2M, m2C, 1B])	->	([(n1+2)M, m1C, 1B], [(n2-2)M, m2C, 0B])
R2	([n1M, m1C, 0B], [n2M, m2C, 1B])	->	([(n1+1)M, (m1+1)C, 1B], [(n2-1)M,(m2-1)C, 0B])
R3	([n1M, m1C, 0B], [n2M, m2C, 1B])	->	([n1M, (m1+2)C, 1B], [n2M,(m2-2)C, 0B])
R4	([n1M, m1C, 0B], [n2M, m2C, 1B])	->	([(n1+1)M, m1C, 1B], [(n2-1)M, m2C, 0B])
R5	([n1M, m1C, 0B], [n2M, m2C, 1B])	->	([n1M, (m1+1)C, 1B], [n2M,(m2-1)C, 0B])

Solution path

Rule number	([3M, 3C, 1B], [0M, 0C, 0B]) <- START STATE
L2	([2M, 2C, 0B], [1M, 1C, 1B])
R4	([3M, 2C, 1B], [0M, 1C, 0B])
L3	([3M, 0C, 0B], [0M, 3C, 1B])
R5	([3M, 1C, 1B], [0M, 2C, 0B])
L1	([1M, 1C, 0B], [2M, 2C, 1B])
R2	([2M, 2C, 1B], [1M, 1C, 0B])
L1	([0M, 2C, 0B], [3M, 1C, 1B])
R5	([0M, 3C, 1B], [3M, 0C, 0B])
L3	([0M, 1C, 0B], [3M, 2C, 1B])
R5	([0M, 2C, 1B], [3M, 1C, 0B])
L3	**([0M, 0C, 0B], [3M, 3C, 1B]) -> GOAL STATE**

State – Space search

- Similar to Production System
- This facilitates easy search
- A path can be found from starting state to goal state while solving the problem
- This consists of four parts basically:
  1. Set ‘S’ containing start state
  2. Set ‘G’ containing goal state
  3. Set of nodes in the graph/tree/path, each node represents the state in problem-solving process
  4. Set of arcs connecting nodes, each arc corresponds to operator that is a step in problem solving process

- Solution path is a path through the graph from a node in S to a node in G
- Objective of search algorithm is to find a solution path in the graph
- There may be more than one solution path
- Exercise a choice between various solution paths based on:
  - Some criteria of goodness or
  - Some heuristic function
- Lets take missionaries & cannibals problem

- Possible operators (applied in this problem) are {2M0C, 1M1C, 0M2C, 1M0C, 0M1C}
  - 2M - means two missionaries
  - 1C – means one cannibal, etc.
- These operators can be used for both the sides of the river
- If boat is on left side, use “->”
- If boat is on right side, use “<-”
- For the sake of simplicity, lets use (L:R), here L=n1Mm1C1B, R=n2Mm2C0B
- Here B means boat, 1 is present & 0 is absent

- Start state: (3M3C1B : 0M0C0B) or (331:000)
- Goal state: (0M0C0B : 3M3C1B) or (000:331)
- Following are filtered out:
  - Invalid states
  - Illegal operators not applicable to some states
  - Some states that are not at all required
- (1M2C1B : 2M1C0B) is an invalid state, which leads to one M & two Cs on the left side
- In case of (2M2C1B : 1M1C0B), the operator 0M1C or 0M2C would be illegal
- Applying same operator both the ways is waste since it leads to previous state, which is said to be looping situation

- To illustrate the progress, develop a tree of nodes, each node represents a state
- Root node represents the start state
- Arcs represents, application of one of the operators
- The nodes for which no operator is applied, are leaf nodes
- Search space generated using valid operators are shown in fig
- The sequence of operators applied to solve the problem is given in table

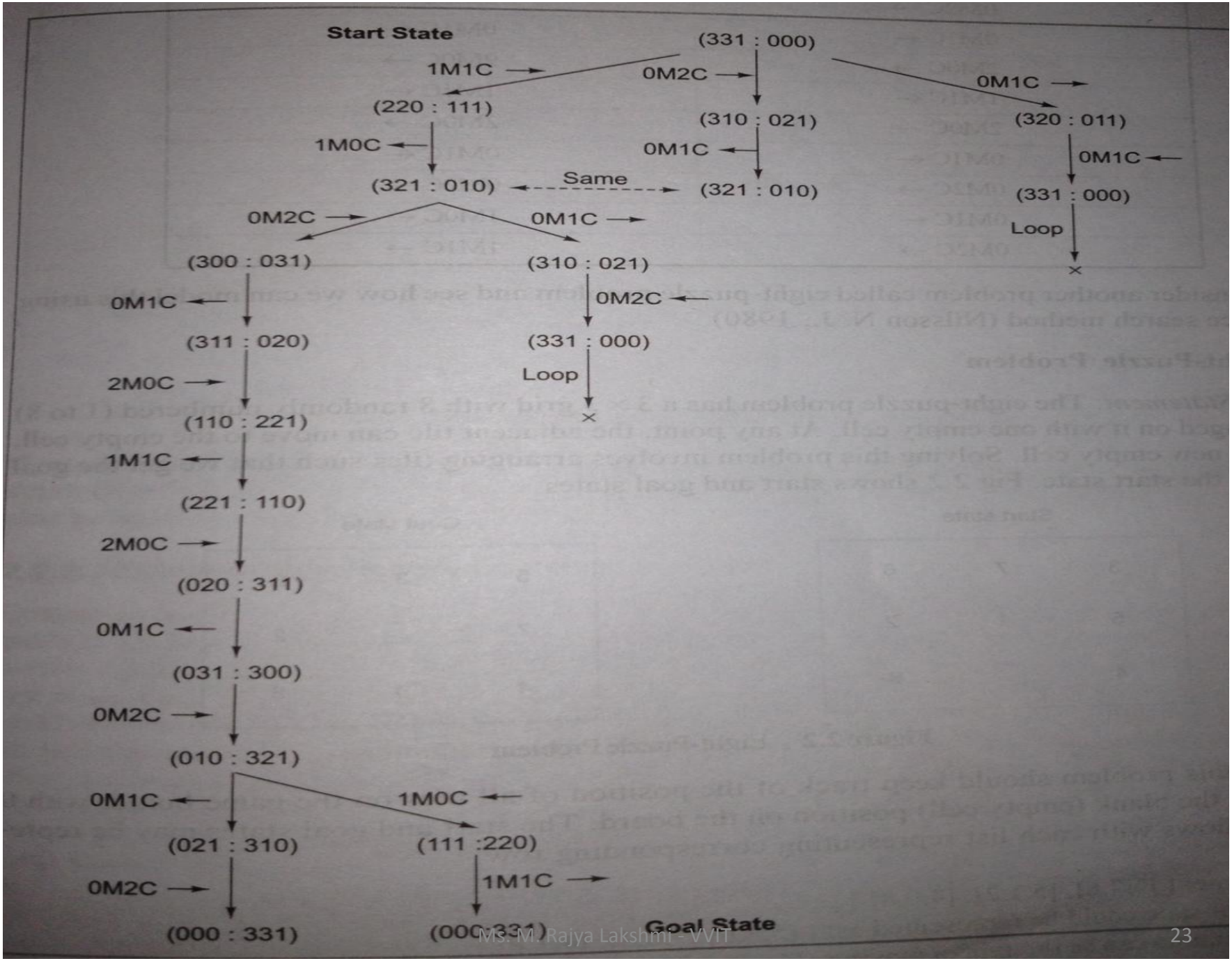
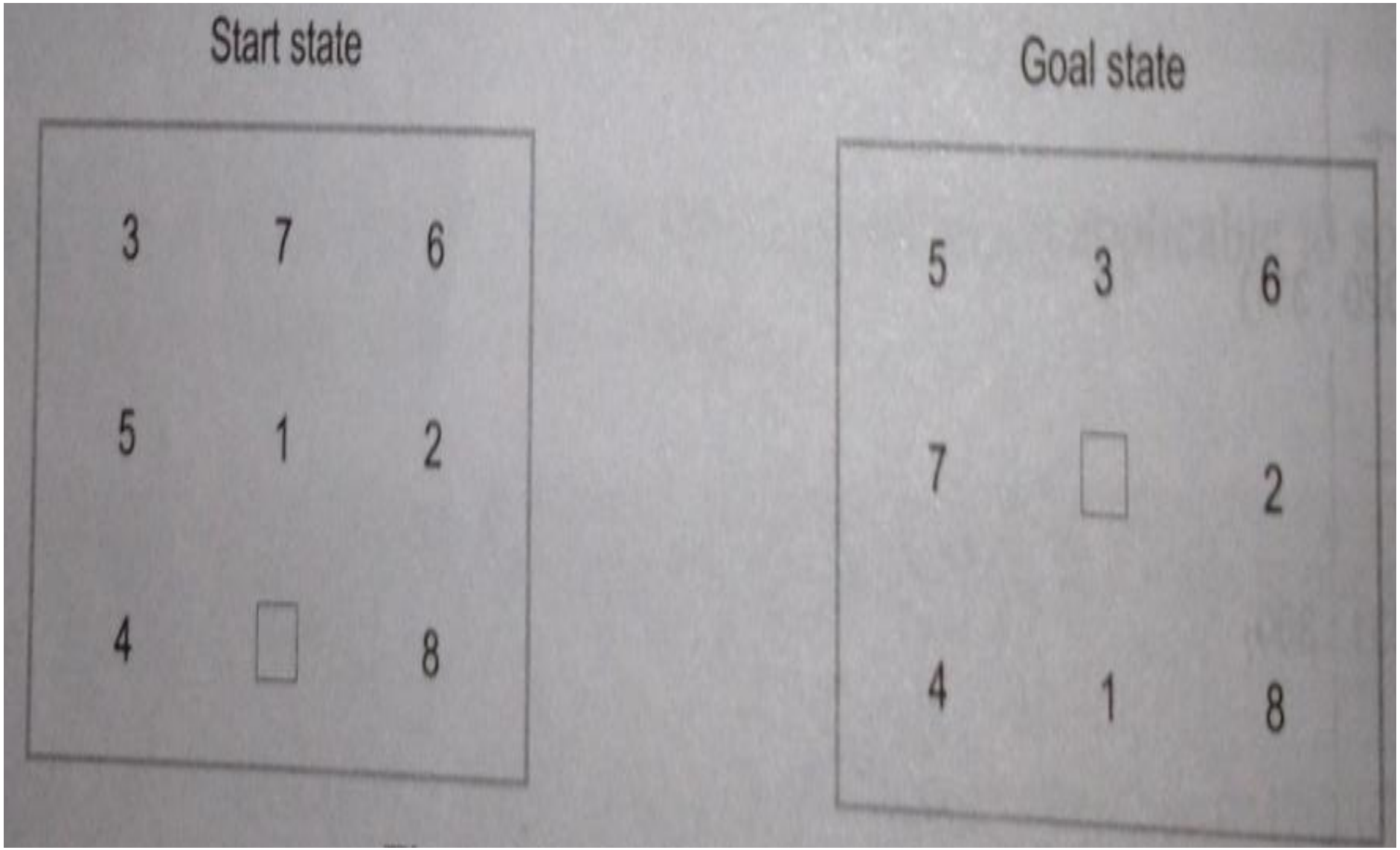

Start State
(331 : 000)
1M1C
0M2C
0M1C
(220 : 111)
(310 : 021)
(320 : 011)
1M0C
0M1C
0M1C
(321 : 010)
Same
(321 : 010)
(331 : 000)
0M2C
0M1C
Loop
(300 : 031)
(310 : 021)
0M1C
0M2C
(311 : 020)
(331 : 000)
2M0C
Loop
(110 : 221)
1M1C
(221 : 110)
2M0C
(020 : 311)
0M1C
(031 : 300)
0M2C
(010 : 321)
0M1C
1M0C
(021 : 310)
(111 :220)
0M2C
1M1C
(000 : 331)
(000 :331)
Goal State

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 →

Eight puzzle problem

- Problem statement:
- A 3x3 grid with 8 randomly numbered (1-8) tiles arranged on it with one empty cell
- At any point, the adjacent tile can move to the empty cell, creating a new empty cell
- Solving this problem involves arranging tiles such that we get the goal state


Start state
3 7 6
5 1 2
4 8
Goal state
5 3 6
7 2
4 1 8

- A state of the problem should keep track of the position of all tiles on the game board
- 0 represents blank position
- The states are
  - Start state: [[3,7,6], [5,1,2], [4,0,8]]
  - Goal state: [[5,3,6], [7,0,2], [4,1,8]]
  - Operations can be {up, down, left, right}
- To simplify, a search tree up to level 2 is shown in the fig. to illustrate the use of operators to generate next state
- Continue search like this to reach goal state

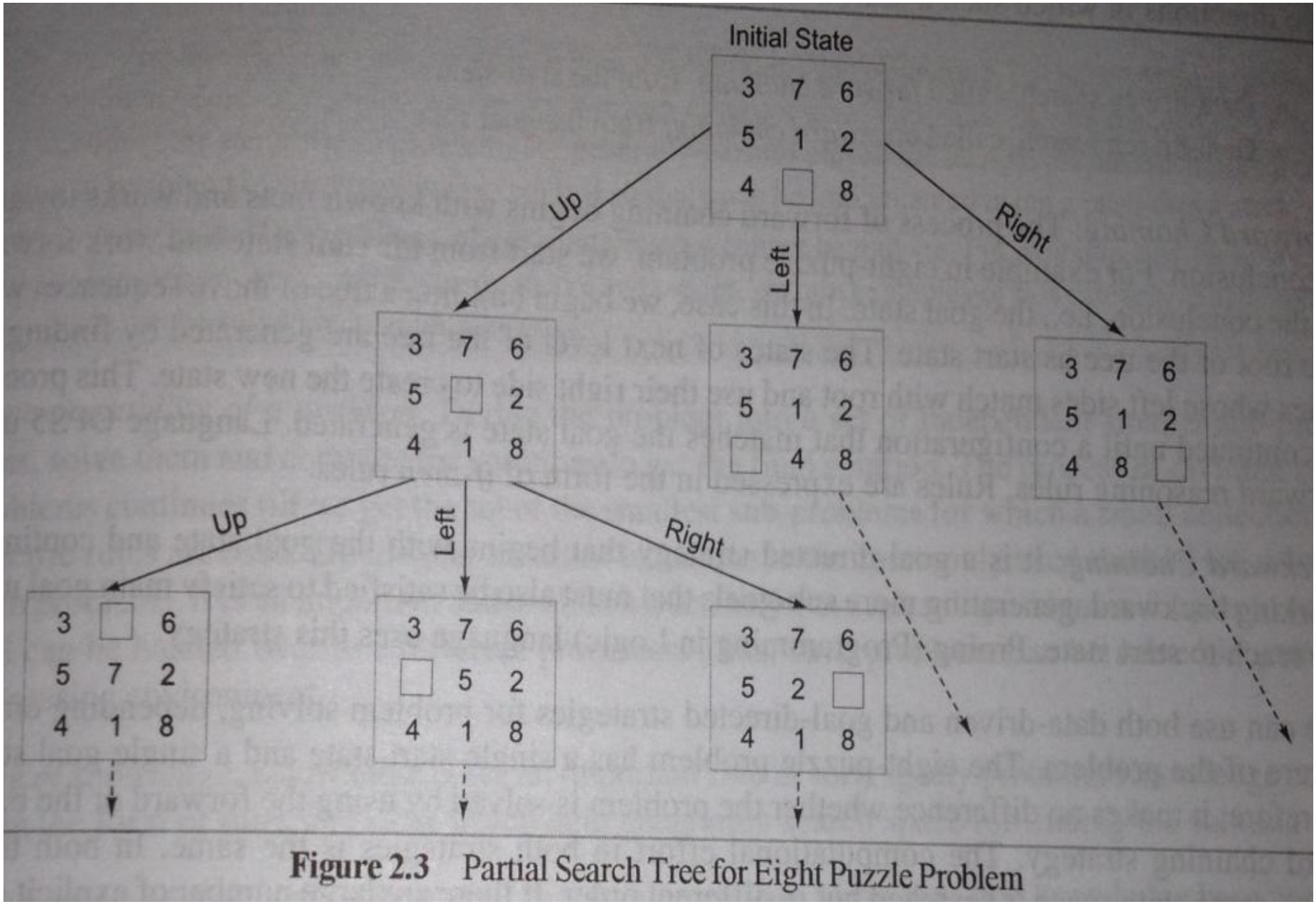


Figure 2.3 Partial Search Tree for Eight Puzzle Problem

Control strategies

- One of the most important components of problem solving
- It describes the order of application of the rules to the current state
- It should move towards solution by exploring the solution space in a systematic way
- DFS & BFS are systematic approaches
- Both are blind searches

- In DFS, single branch of the tree is followed until it yields a solution
- Or some pre-specified depth has reached, and then go back to immediate previous node and explore other branches
- In BFS, a search space tree is generated level wise until a solution is found or some specified depth is reached
- Both are exhaustive, uninformed and blind
- To solve real-world problems, effective control strategy must be used

- To find the correct strategy for a given problem, there are two directions
  - **Data-driven search**, called **forward chaining**, from start state
  - **Goal-driven search**, called **backward chaining**, from the goal state

Forward chaining

- Begins with known facts and works towards a conclusion
- Ex. In eight puzzle problem, we start from start state and work forward to the goal state
- In this a tree is started building with move sequences with the start state as root of tree
- The states of next level are generated by all rules
- This process is continued until a configuration that matches the goal state

Backward chaining

- It is a goal-directed strategy
- Begins with goal state and continued working backward
- In many cases this is useful strategy
- The eight puzzle problem have single start state and single goals state
- It makes no difference when the problem is solved using either chaining strategy since computational effort in both the strategies is same

Example

As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American.

Prove That: Robert is criminal

Characteristics of problem

- Analysing the problem along several key characteristics is a must before solving it
- Some of the types are:
  - **Type of problem**
  - **Decomposability of problem**
  - **Role of knowledge**
  - **Consistency of knowledge base used**
  - **Requirement of solution**

Type of problem

- There are 3 types of problems in real life
  - **Ignorable**
    - Solution steps can be ignored for these problems
    - E.g. in proving a theorem, a lemma can be ignored
  - **Recoverable**
    - Solution steps can be undone for these problems
    - E.g. water jug prob, if a jug is filled, it can be emptied
    - Used in single player puzzles, solved by back tracking, using push-down stack
  - **Irrecoverable**
    - Solution steps cannot be undone for these problems
    - E.g. any two player games, like chess, snakes and ladders, etc.
    - These are solved by planning process

Decomposability of problem

- Divide the problem into a set of independent smaller sub-problems
- Solve them and combine the solutions to get final answer
- Each sub-problem is solved by a different processor in parallel processing environment
- Divide-and-conquer technique is the commonly used method

Role of knowledge

- Knowledge plays an important role in solving any problem
- Knowledge could be in the form of rules and facts
- They help in generating search space for finding the solution

Consistency of knowledge base used

- The knowledge base used to solve the problem should be consistent
- Inconsistent knowledge base will lead to wrong solutions
- E.g. if the knowledge is in the form of rules and facts as follows:
  - It is humid, it will rain, then it is daytime
  - It is sunny day
  - It is night-time
- This is not consistent as there is a contradiction

Requirement of solution

- Problem should be analysed whether the solution is **absolute** or **relative**
- Exact solution is absolute solution, reasonably good or approximate solution is relative solution
- E.g. 1 - for water jug problem, one solution is considered and no need to find better solution
- The solution is absolute, it is any path solution
- E.g. 2 - for travelling salesman problem, unless all routes are known, it is difficult to find the shortest path
- The solution is relative, it is best path solution
- Best-path problems are computationally harder than any-path problems

Exhaustive searches

- Following are the few uninformed exhaustive searches:
  - **Breadth-First-Search (BFS)**
  - **Depth-First-Search (DFS)**
  - **Depth-First-Iterative Deepening**
  - **Bidirectional Search**

Breadth-First-Search (BFS)

- BFS expands all the states one step away from start state, and then expands two steps away, and then three steps away, and so on until a goal state is reached
- BFS always give an optimal solution
- BFS uses two lists, viz. **OPEN** & **CLOSED**
- **OPEN is a Queue**, that contains the states that are to be expanded
- **CLOSED is a Stack**, that contains the states that are already expanded
- E.g. a BFS is implemented to check whether a goal node exists or not for water jug problem

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

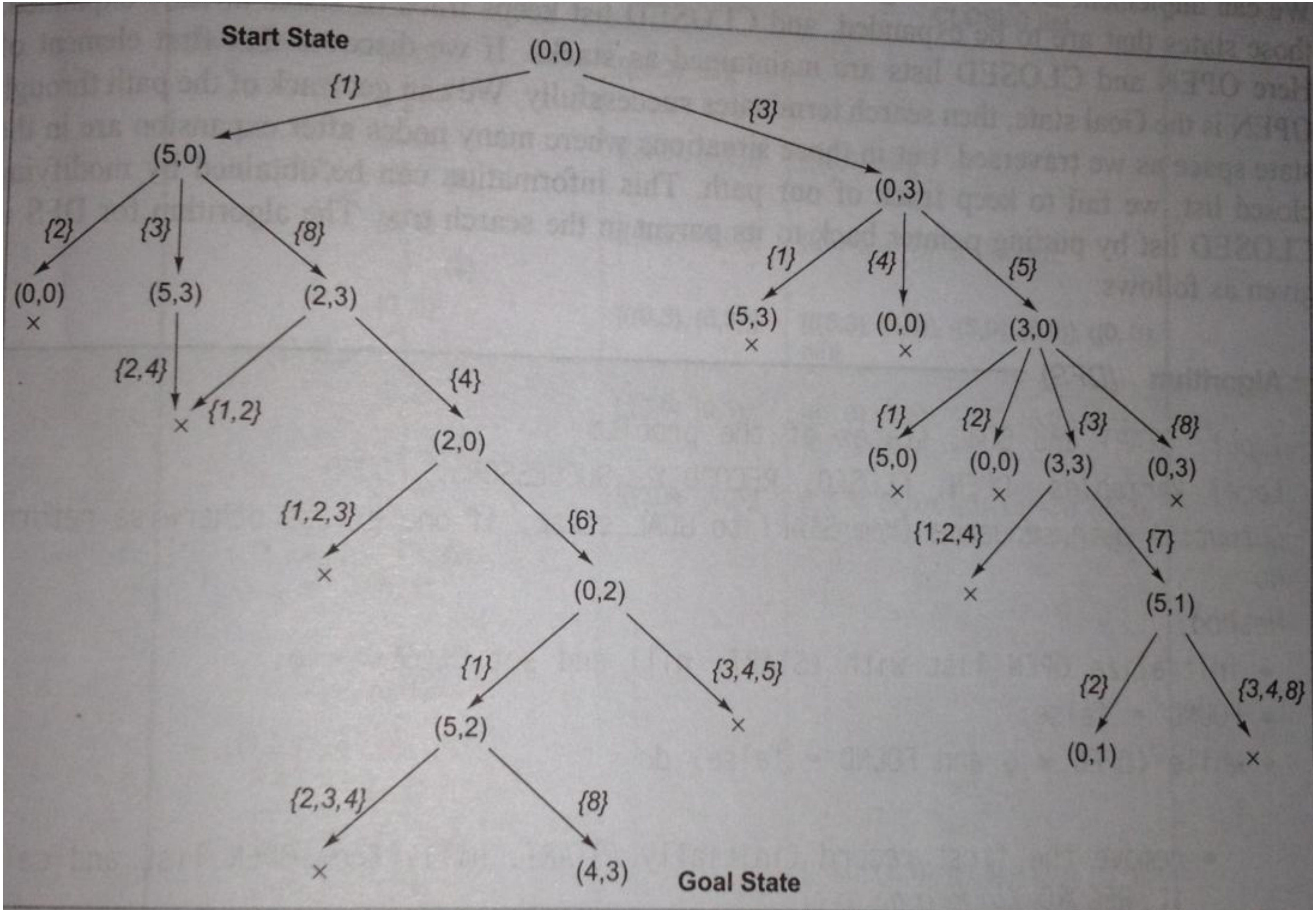


Figure 2.4 Search Tree Generation using BFS

- At each state, applicable rule is applied
- If it generates one of previous states, cross it
- If new state is generated, expand it in BFS style
- A path is visible from start to goal states by tracing the tree in reverse through parent link
- The path is optimal and more shorter path is not available
- Solution path:

  (0,0) -> (5,0) -> (2,3) -> (2,0) -> (0,2) -> (5,2) -> (4,3)

Depth-First-Search (DFS)

- DFS is implemented by using two stacks OPEN & CLOSED
- OPEN list contains the states that are to be expanded
- CLOSED list keeps track of states already expanded
- If we discover the first element of the OPEN as goal state, get track of the path as we traversed

Algorithm *(DFS)*

Input: START and GOAL states of the problem

Local Variables: OPEN, CLOSED, RECORD_X, SUCCESSORS, FOUND

Output: A path sequence from START to GOAL state, if one exists otherwise return No

Method:

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

  }

  }/* end while */

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

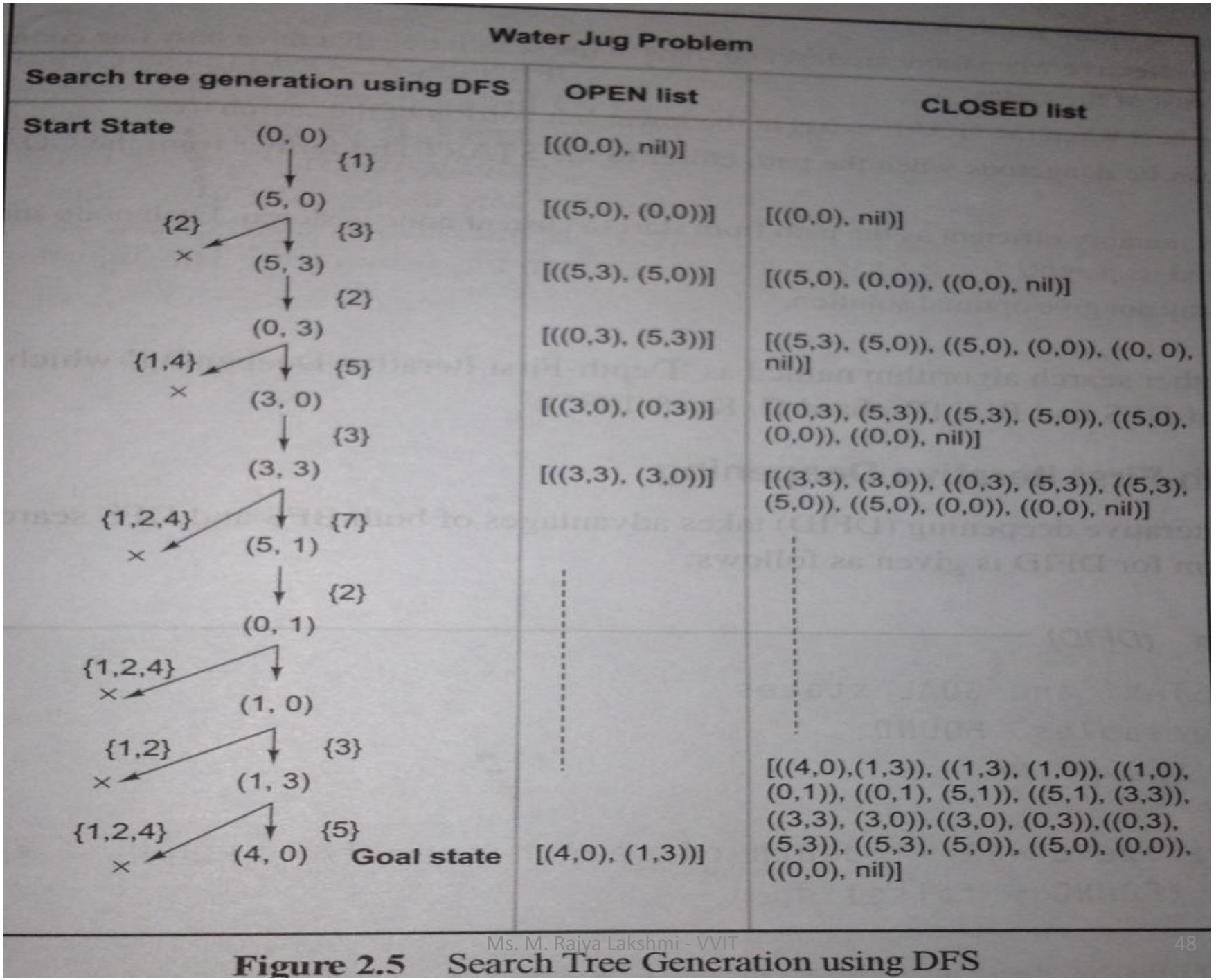


Figure 2.5 Search Tree Generation using DFS

Comparisons

- BFS is effective when the branching factor of the tree is low
- BFS can work even in infinitely deep trees
- BFS requires huge memory since no.of nodes in each level increases exponentially
- BFS gives optimal solution

Comparisons .. contd

- DFS is effective when there are few sub-trees in the search tree
- DFS can be dangerous when the path closer to the START and farther from the GOAL has been chosen
- DFS is memory efficient as the path from start to current node is stored, each node should contain state and its parent
- DFS may not give optimal solution

Depth-First-Iterative Deepening

- DFID takes the advantages of BFS & DFS
- DFID expands all nodes at a given depth before expanding any nodes at greater depth
- DFID gives optimal solution of shortest path from start state to goal state
- At any time it is performing a DFS and never searches deeper than depth ‘d’
- Thus it uses the space *O(d)*
- Disadvantage, is it performs wasted computation
- Working of DFID algorithm is as follows

Algorithm *(DFID)*

Input: START and GOAL states

Local Variables: FOUND;

Output: Yes or No

Method:

- initialize d = 1 /* depth of search tree */ . FOUND = false
- while (FOUND = false) do
  {
  - perform a depth first search from start to depth d.
  - if goal state is obtained then FOUND = true else discard the nodes generated in the search of depth d
  - d = d + 1

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

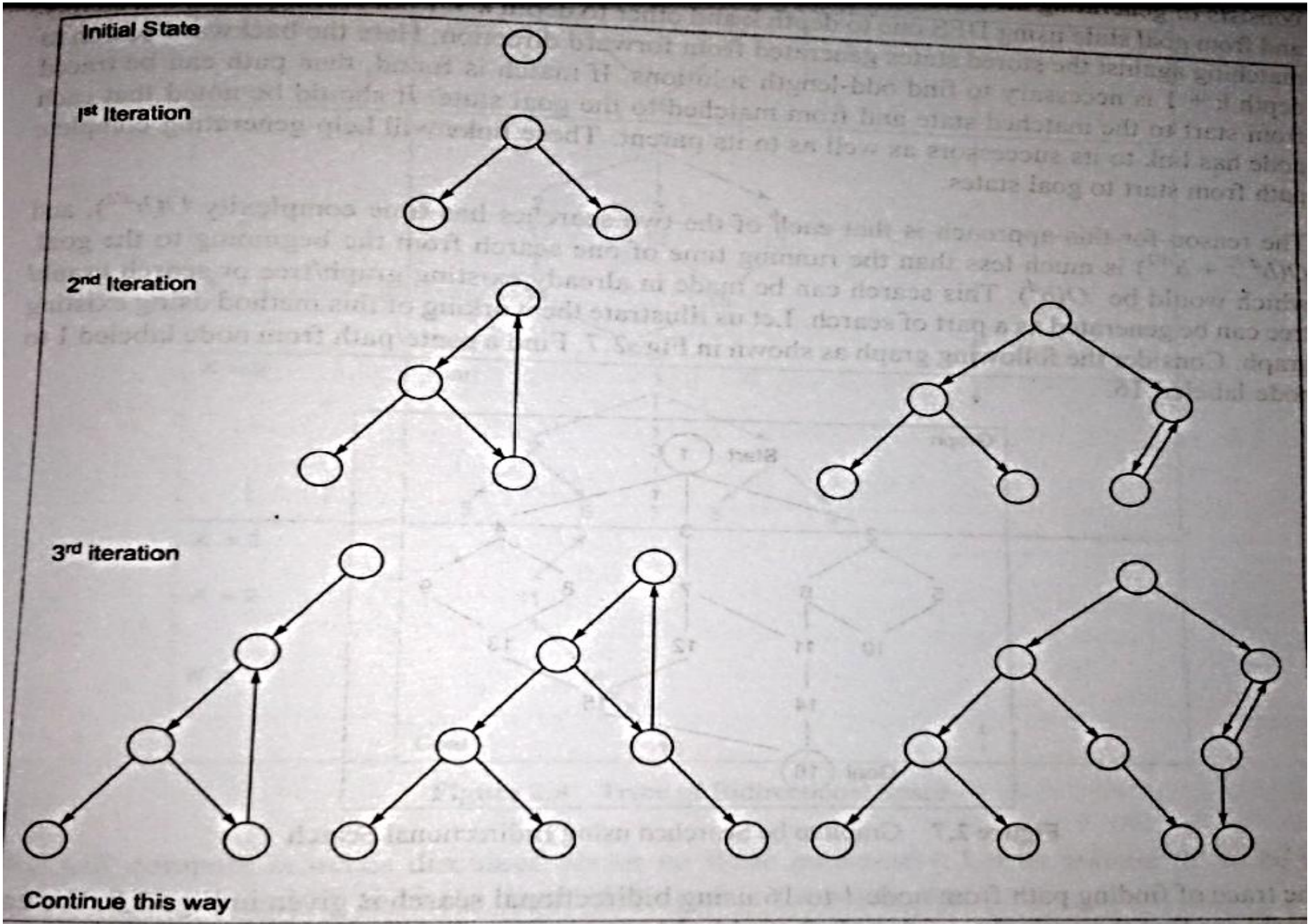
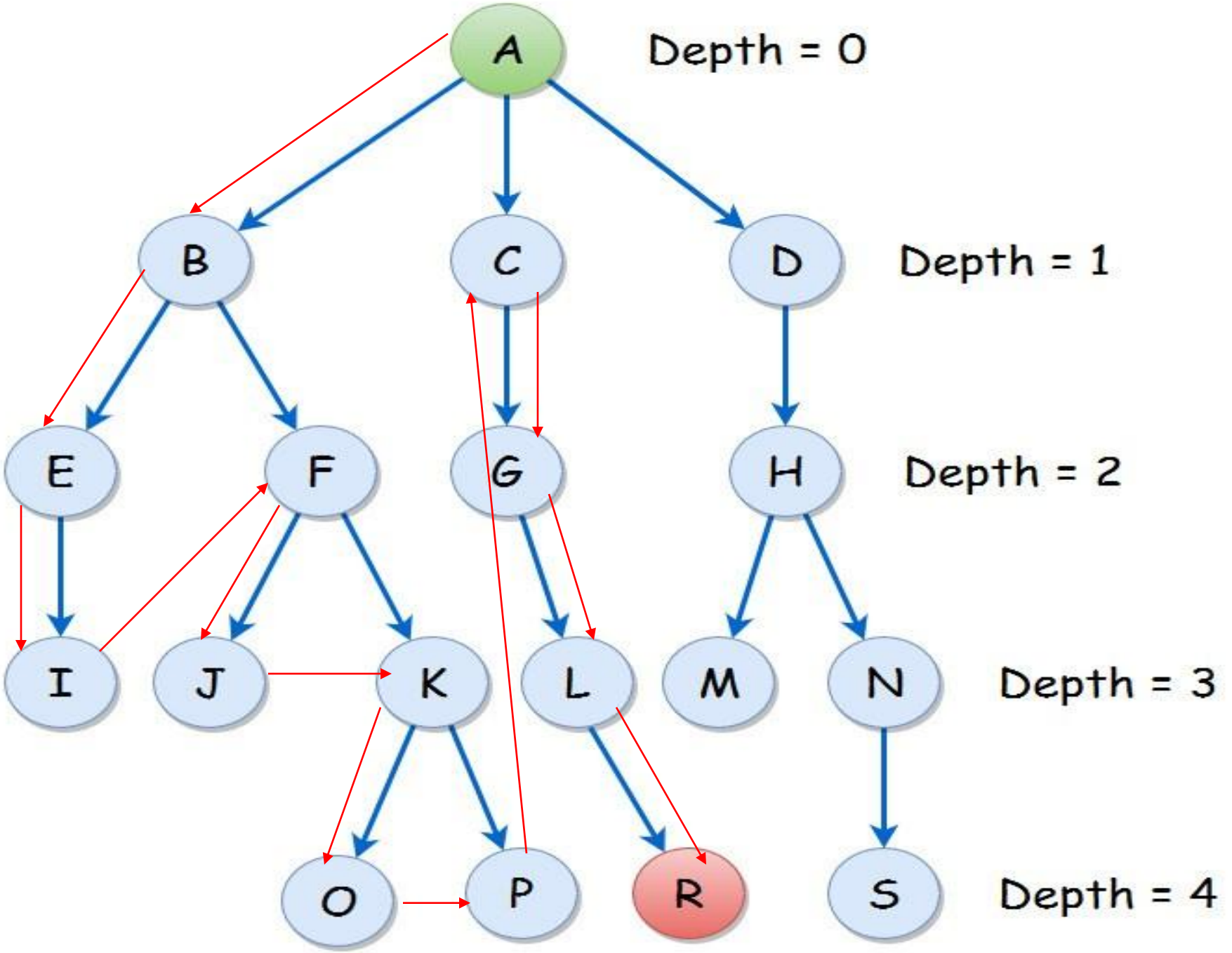
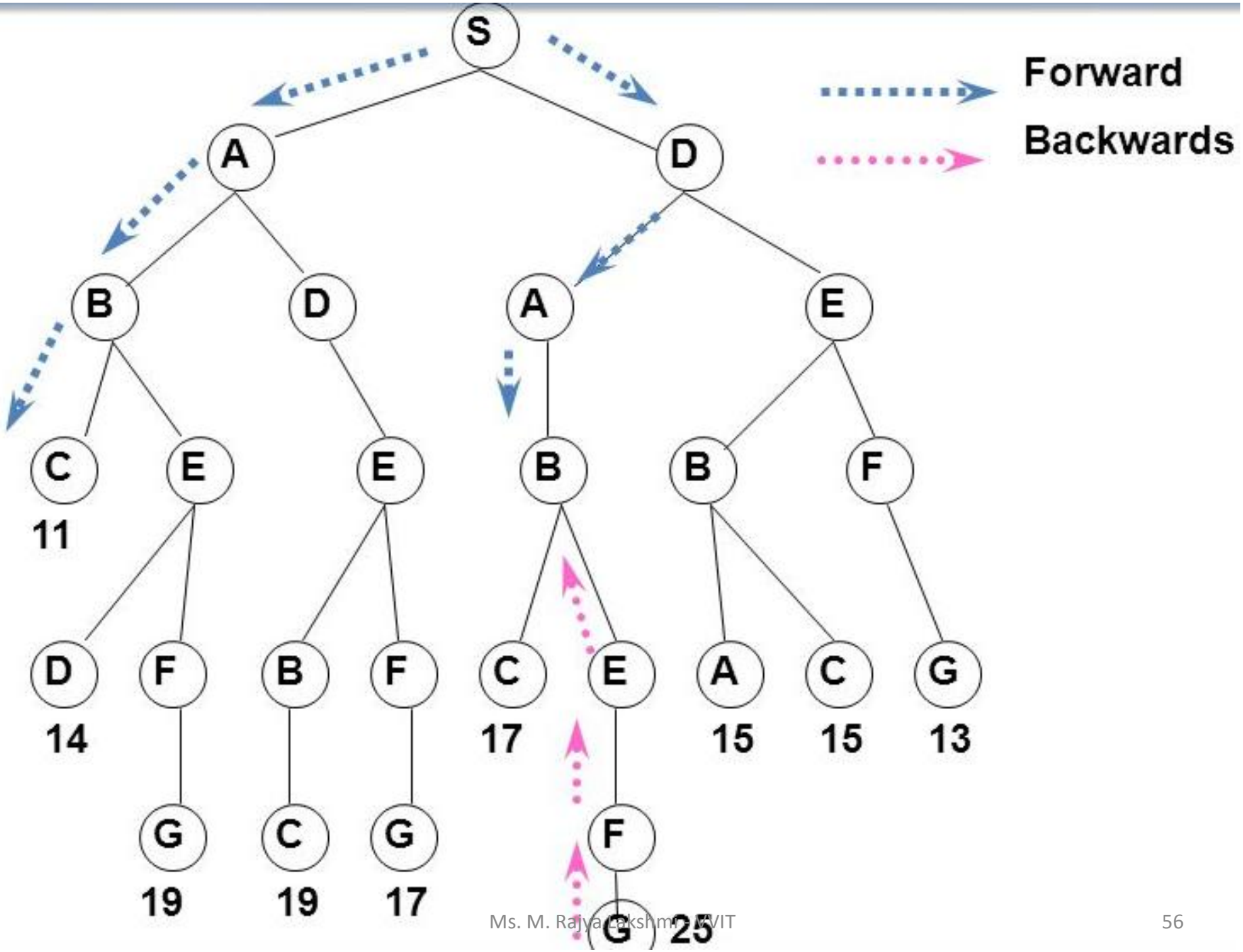


Figure 2.6 Search Tree Generation using DFID


A
Depth = 0
B
C
D
Depth = 1
E
F
G
H
Depth = 2
I
J
K
L
M
N
Depth = 3
O
P
R
S
Depth = 4

Bidirectional Search

- It runs two simultaneous searches
- One search moves forward from start to goal and another moves back from goal to start
- Searching is stopped when both meet in middle
- Useful if there are only one start and goal states
- If match is found, path can be traced from start to match state and match to goal state
- Each node has link to its successors and parents
- Each of two searches has time complexity $O(b^{d/2})$ and $O(b^{d/2}+b^{d/2})$ is much less running time


S
A
D
B
D
A
E
C
E
E
B
B
F
11
D
F
B
F
C
E
A
C
G
14
17
15
15
13
G
C
G
F
19
19
17
G
25
Forward
Backwards

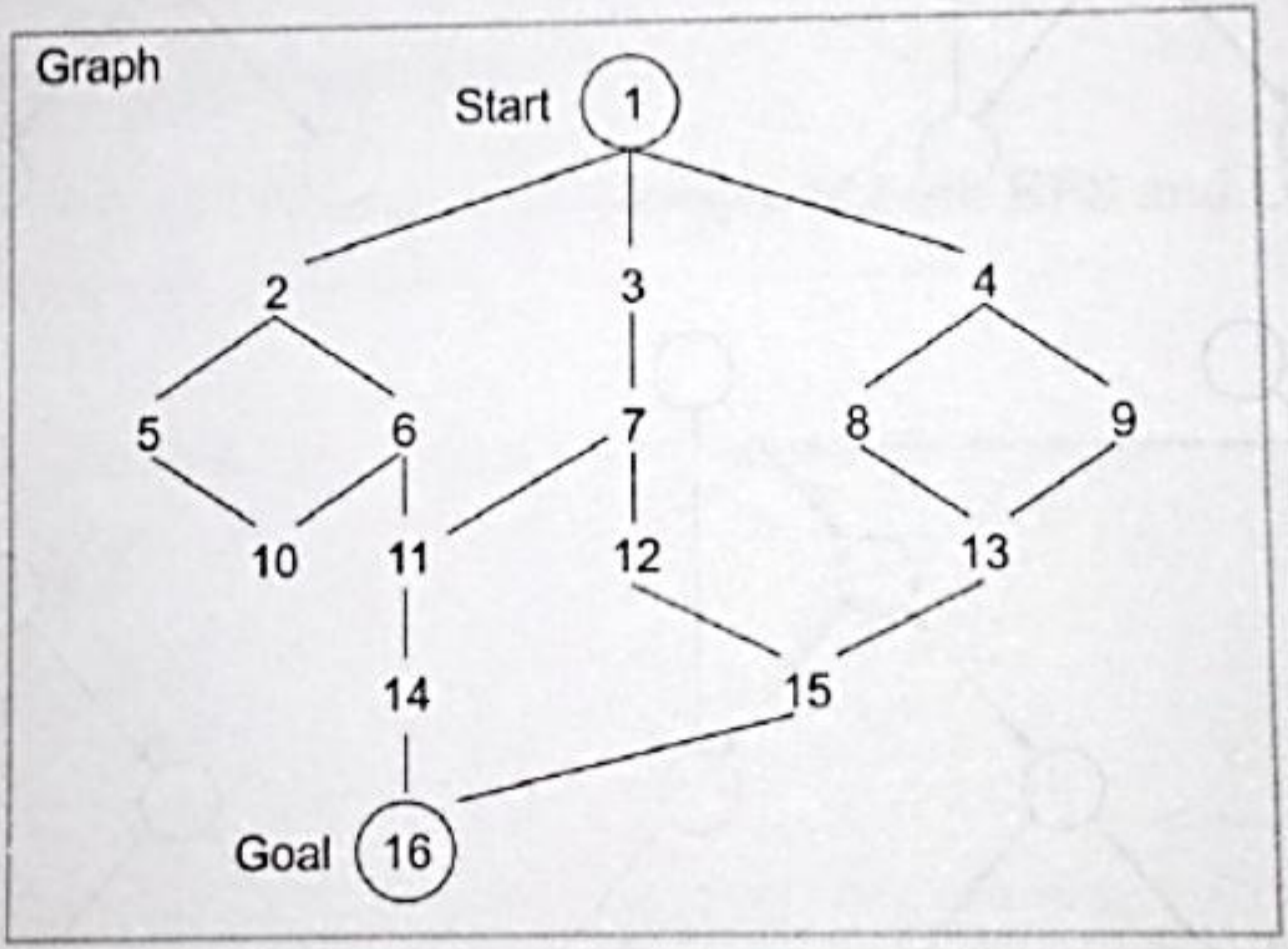


Figure 2.7 Graph to be Searched using Bidirectional Search

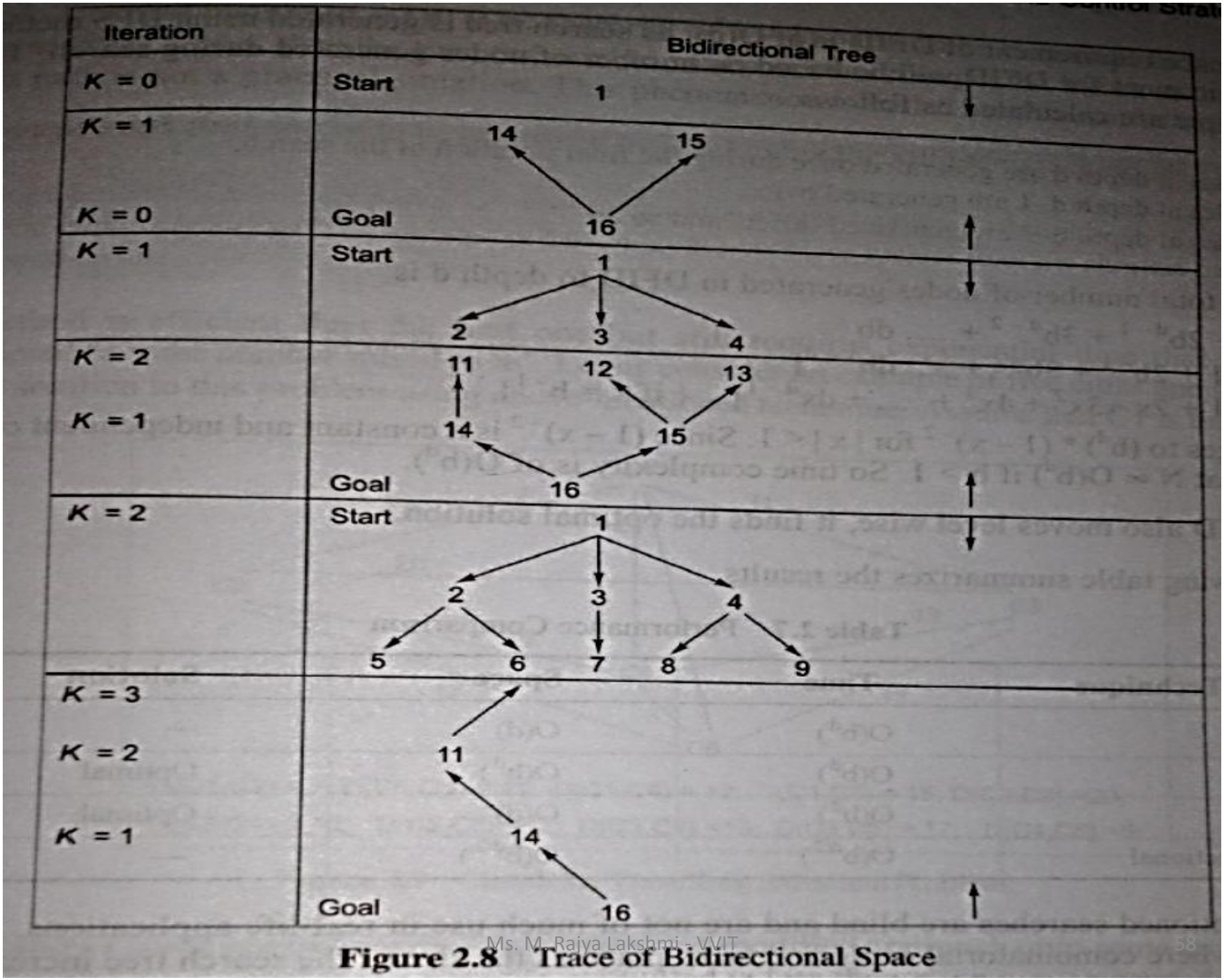


Figure 2.8 Trace of Bidirectional Space

Analysis of search methods

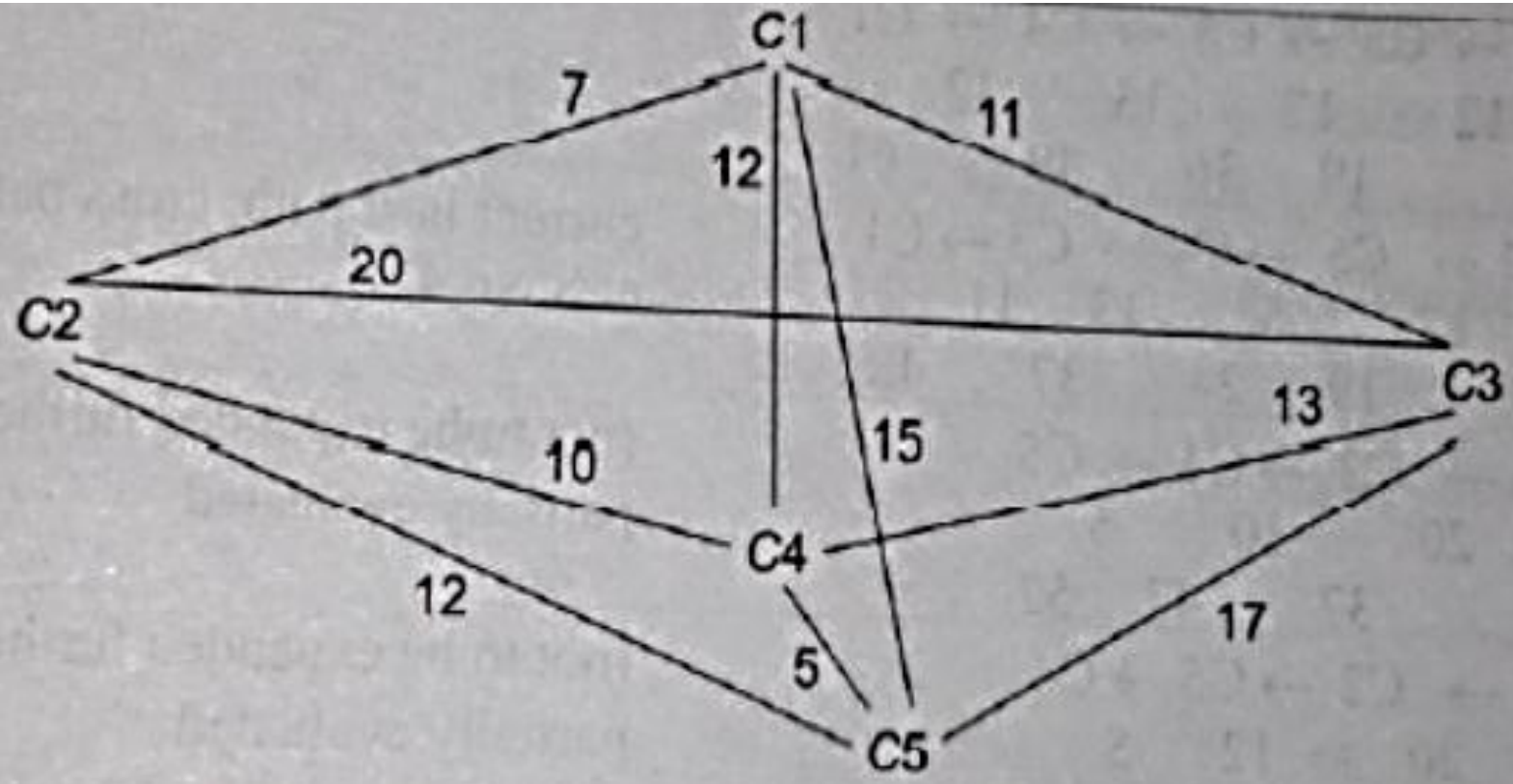
- Effectiveness of any search strategy in problem solving is measured in terms of:
  - Completeness: Algorithm guarantees a solution if it exists
  - Time complexity: Time required to find a solution
  - Space complexity: Space required to find a solution
  - Optimality: The algorithm is optimal if it finds the highest quality solution when there are several different solutions for the problem

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	$O(b^{d/2})$	$O(b^{d/2})$	—

Travelling salesman problem

- There are n cities and the distance between each pair of the cities is given
- Find the shortest route of visiting all cities once and return back to starting point
- This require (n-1)! Paths to be examined
- This phenomenon is called ‘combinatorial explosion’
- Start generating complete paths, keeping track of the shortest path found so far
- Stop exploring any path as soon as its partial length becomes greater than the shortest path length found so far



D(C1,C2) = 7; D(C1,C3) = 11; D(C1,C4) = 12; D(C1,C5) = 15; D(C2,C3) =20;
D(C2,C4) = 10; D(C2,C5) =12; D(C3,C4) =13; D(C3,C5) = 17; D(C4,C5) =5;

Figure 2.9 Graph for Travelling Salesman Problem

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

Heuristic search techniques

- This is a criterion to find out most effective way to achieve the goal among all choices
- It no longer guarantees to find the best solution but finds a very good solution (by using good heuristics)
- There are two types of heuristics
  - **General purposes heuristics** – useful in various problem domains
  - **Special purpose heuristics** – domain specific

General purposes heuristics

- A General purpose heuristics for combinatorial problem is Nearest Neighbour Algorithms that work by selecting the locally superior alternative
- In many AI problems it is often difficult to measure precisely the goodness of a solution
- For real-world problems, it is 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

Branch and bound search

Algorithm *(Branch and Bound)*

Input: START and GOAL states

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

Output: Yes or No

Method:

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

    }

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

IMG_0710.JPG

Hill climbing

Algorithm *(Simple Hill Climbing)*

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

- store initially the st IMG_0711.JPG in a OPEN list (maintained as stack): FOUND = false:
- while (OPEN ≠ empty and Found = false) do
  {
  - remove the top element from OPEN list and call it NODE:
  - if NODE is the goal node, then FOUND = true else
    - find SUCCs of NODE, if any;
    - sort SUCCs by estimated cost from NODE to goal state and add them to the front of OPEN list:

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

Beam search

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

Best first search

Algorithm *(Best-First Search)*

IMG_0714JPG

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

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

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

A* algorithm

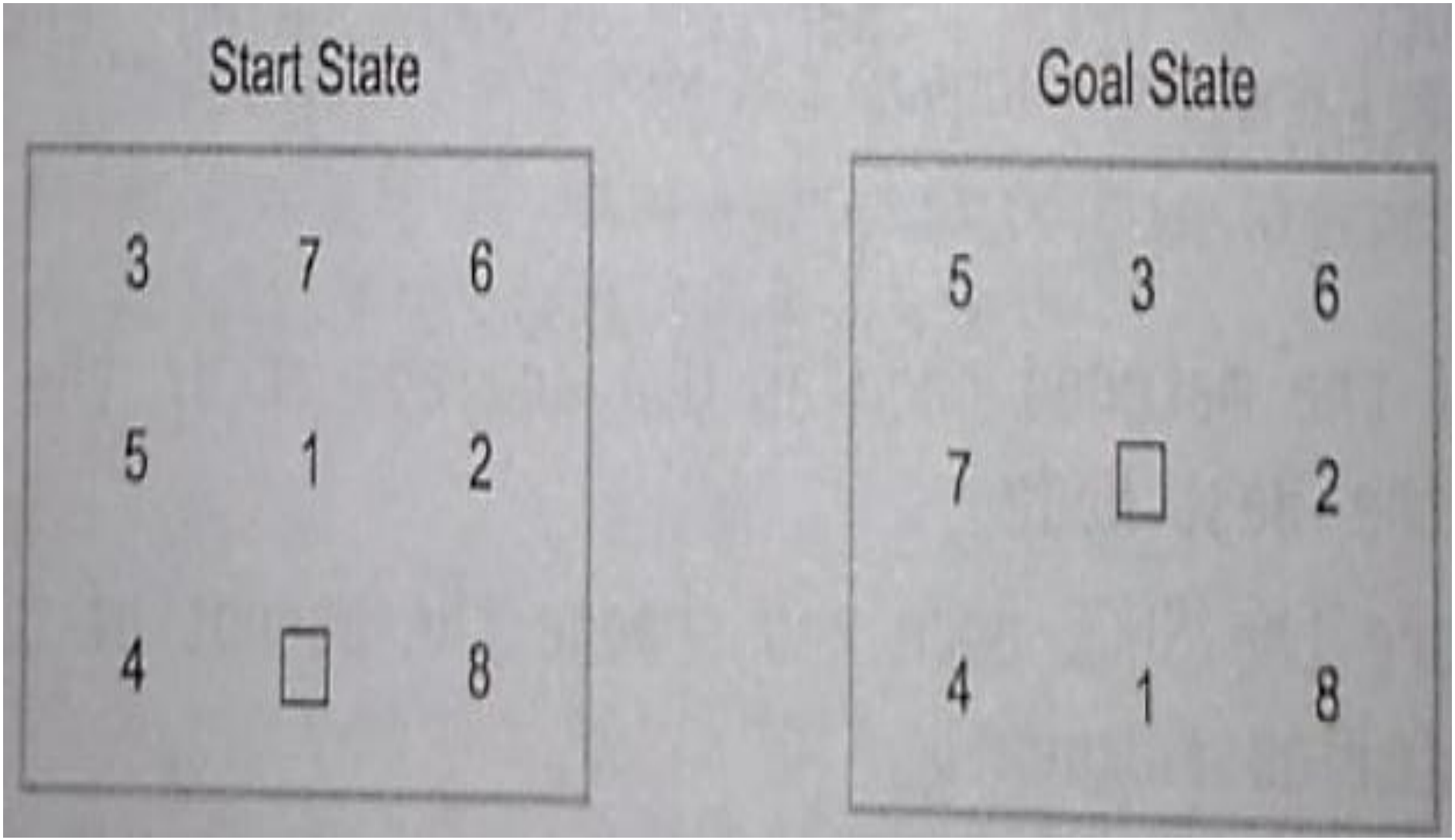
- It uses a heuristic or evaluation function f(X)
- F(n) = g(n) + h(n)
- Function g is the cost of starting node to node n
- Function h is the cost of node n to goal node
- A* algorithm incrementally searches all the routes starting from starting node until it finds the shortest path to goal node

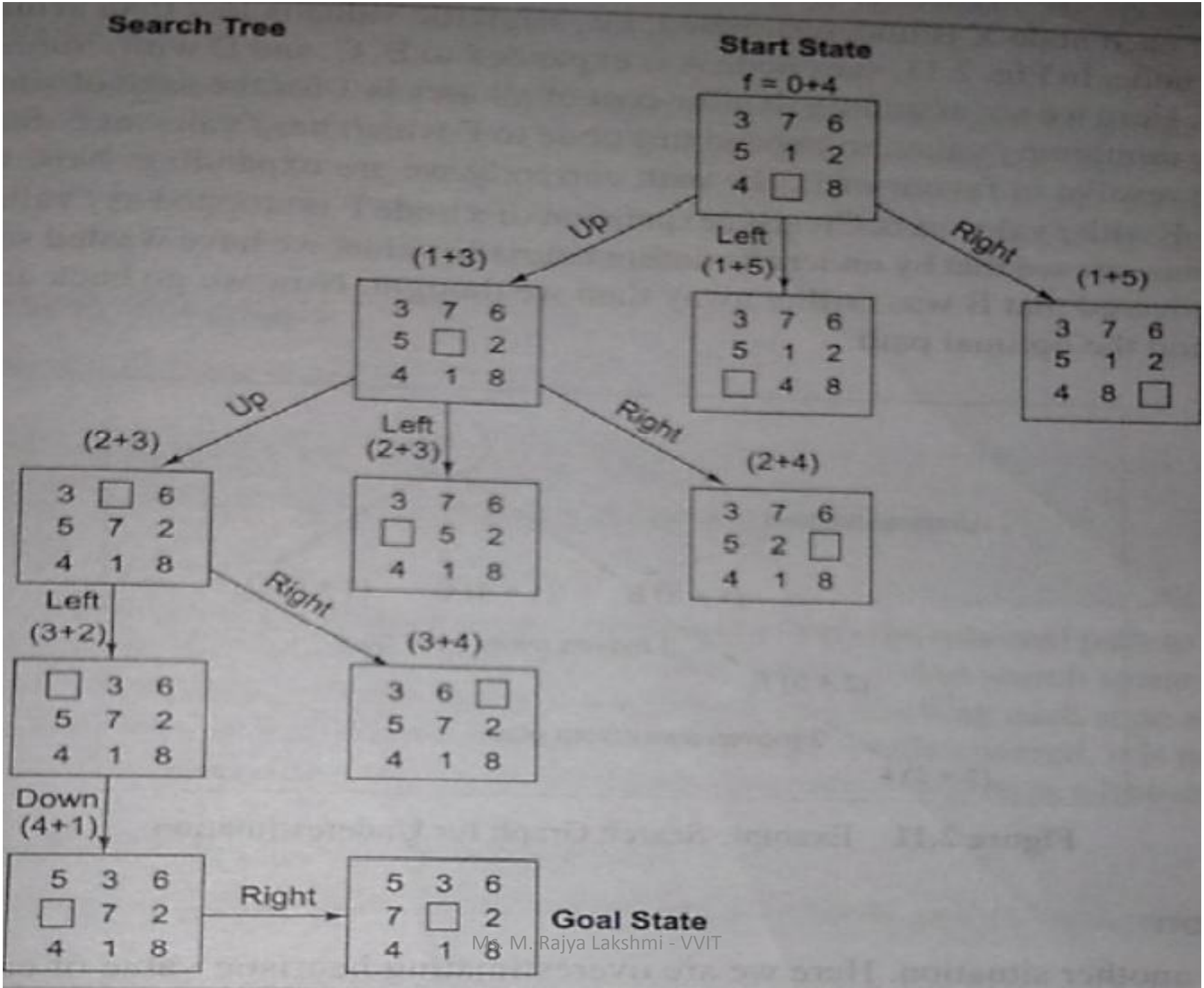
Algorithm (A*)

```
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 the Best_Node;
    • for each SUCC do
      {
      • establish parent link of SUCC:  /* This link will help to
        recover path once the solution is found */
      • compute g(SUCC) = g(Best_Node) + cost of getting from Best_Node
        to SUCC;
      • if SUCC ∈ OPEN then /* already being generated but not
        processed */
        {
        • call the matched node as OLD and add it in the successor list
          of the Best_Node;
        • ignore the SUCC node and change the parent of OLD, if required
          as follows:
          • if g(SUCC) < g(OLD) then make parent of OLD to be Best_Node
            and change the values of g and f for OLD else ignore;
      }
    • If SUCC ∈ CLOSED then /* already processed */
      {
      • call the matched node as OLD and add it in the list of the
        Best_Node successors;
      • ignore the SUCC node and change the parent of OLD, if required
        as follows:
        • if g(SUCC) < g(OLD) then make parent of OLD to be
          Best_Node and change the values of g and f for OLD and
          propogate the change to OLD's children using depth first
          search else ignore;
```

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

- Starting with a given node, the algorithm expands the node with the lowest f(X) value
- It maintains a set of partial solutions
- Unexpanded leaf nodes of expanded leaf nodes are stored in a queue with f values
- E.g. solve eight puzzle problem using A* algo.
- F(X) = g(X)+h(X)
- h(X) -> no.of tiles, not in their goal position in given state X
- g(X) -> depth of node X in the search tree

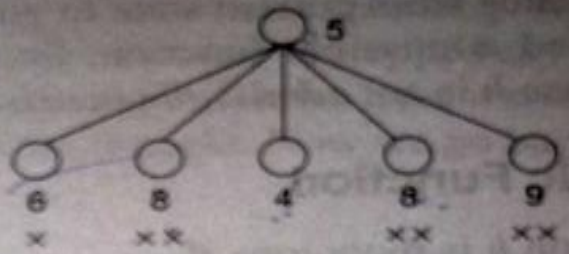

Start State
3 7 6
5 1 2
4 8
Goal State
5 3 6
7 2
4 1 8



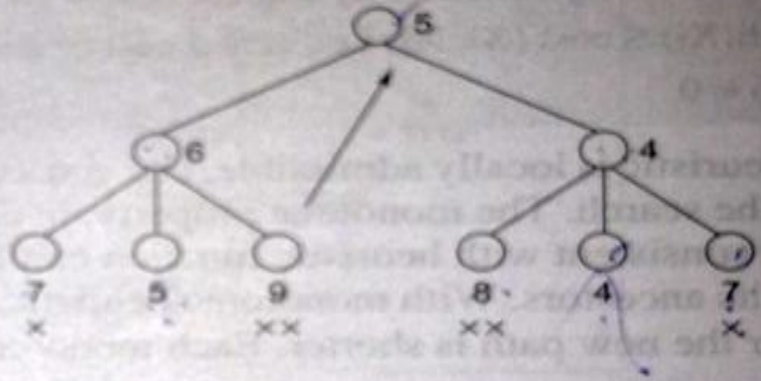
Iterative deepening a*

- IDA* is the combination of the DFID & A* algo
- Here, successive iterations are corresponding to increasing values of the total cost of a path rather than increasing depth of the search
- For each iteration, perform a DFS pruning off a branch when its total cost (g+h) exceeds a given threshold
- Initial threshold starts at the estimate cost of the start state and increase for each iteration of the algorithm
- Threshold used for the next iteration is the minimum cost of all values exceeded the current threshold
- Repeat until find a goal

1st iteration (Threshhold = 5)



2nd iteration (Threshold = 6)



3rd iteration (Threshold = 7)

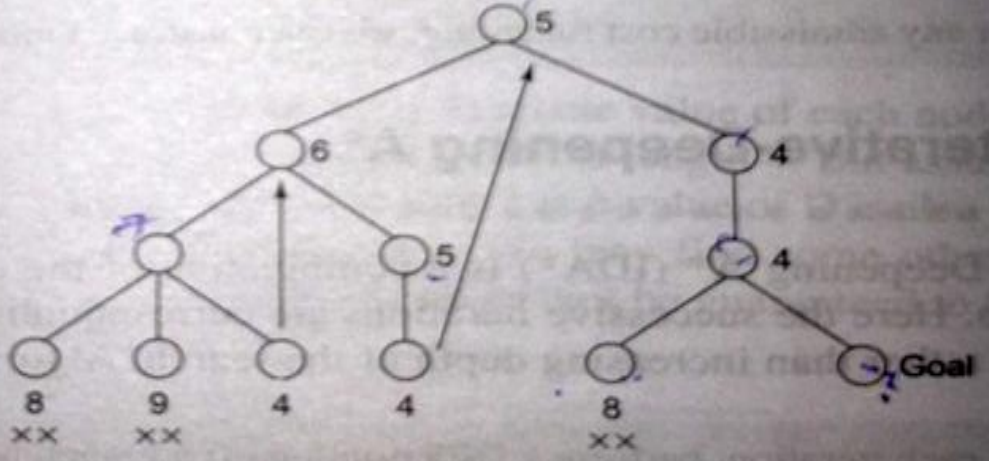


Figure 2.13 Working of IDA*

- The IDA* finds a least cost or optimal solution
- Uses far less space than A*
- It expands approximately the same number of nodes as the A* in a tree search
- Simpler to implement as there are no OPEN and CLOSED lists to be maintained
- A simple recursion performs DFS inside an outer loop to handle iterations

Constraint Satisfaction

- Many AI problems can be viewed as problems of constraint satisfaction
- In this the goal is to solve some problem state instead of optimal path
- These are Constraint Satisfaction (CS) Problems
- Search can be made easier in those cases in which the solution is required to satisfy local consistency conditions
- E.g. Cryptography, n-Queen Problem, map colouring, crossword puzzle, etc.

Algorithm

- until a complete solution is found or all paths have lead to dead ends

  {

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

  }

- Stop

Crypt-Arithmetic puzzle

- Problem statement: solve the puzzle by assigning 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

$C_4C_3C_2C_1$

B A S E

+ B A L L

G A M E S

- Constraints equations are:
- E+L = S
- $S+L+C_1=E$
- $2A+C_2=M$
- $2B+C_3=A$
- $G=C_4$

1. $G = C_4 \Rightarrow \boxed{G = 1}$

2. $2B + C_3 = A \longrightarrow C_4$

   2.1 Since $C_4 = 1$, therefore, $2B + C_3 > 9 \Rightarrow$ B can take values from 5 to 9.

   2.2 Try the following steps for each value of B from 5 to 9 till we get a possible value of B.

   - If B = 5
     - if $C_3 = 0 \Rightarrow A = 0 \Rightarrow M = 0$ for $C_2 = 0$ or $M = 1$ for $C_2 = 1$ ×
     - if $C_3 = 1 \Rightarrow A = 1$ × (as G = 1 already)
   - For B = 6 we get similar contradiction while generating the search tree.
   - If $\boxed{B = 7}$, 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 above
   - So $S+5 > 9 \Rightarrow$ 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 and on backtracking, we may find more solutions. In this case we get only one solution.

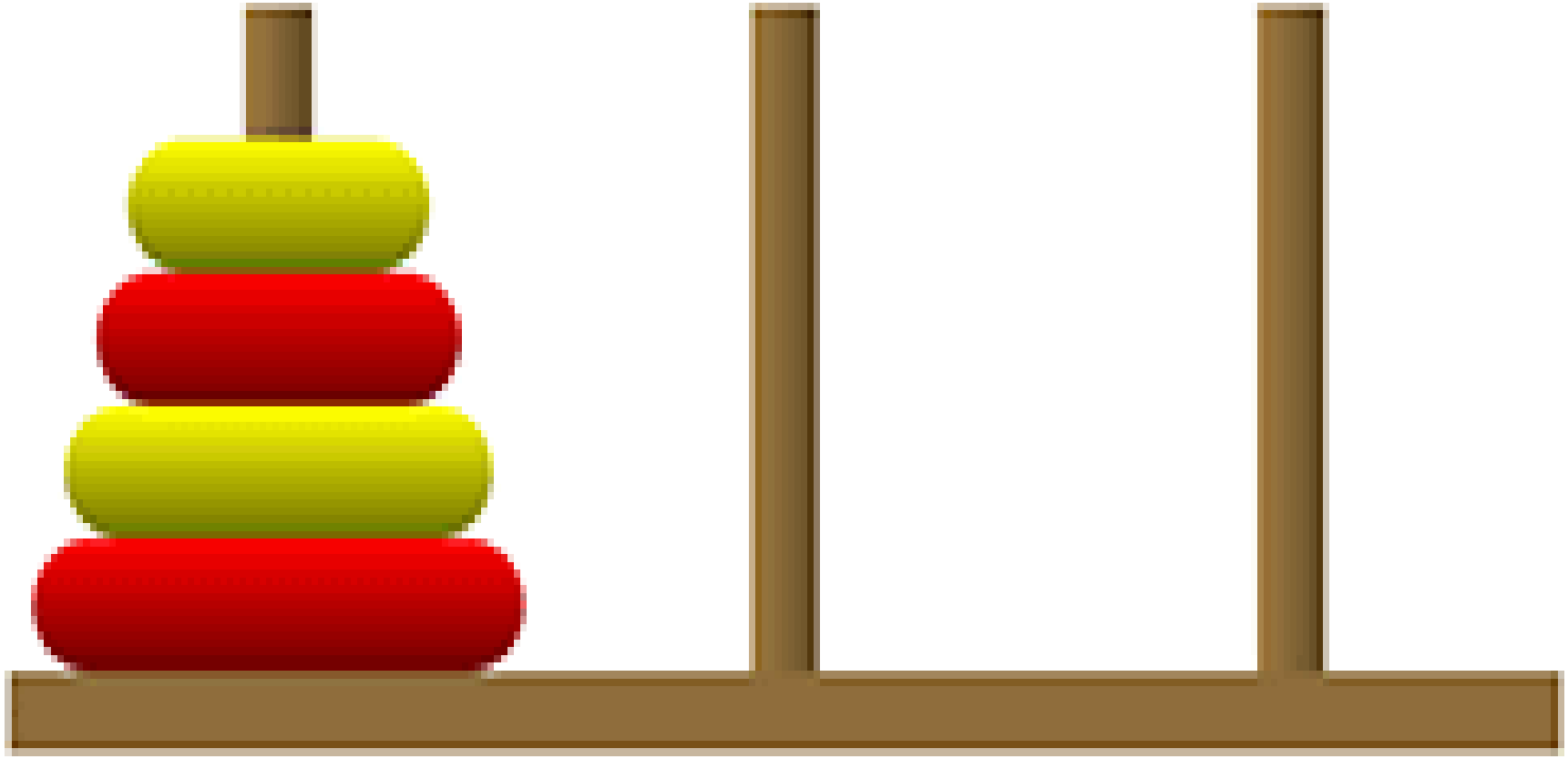
   G = 1 ; A = 4 ; M = 9; E = 3 ; S = 8 ; B = 7 ; L = 5

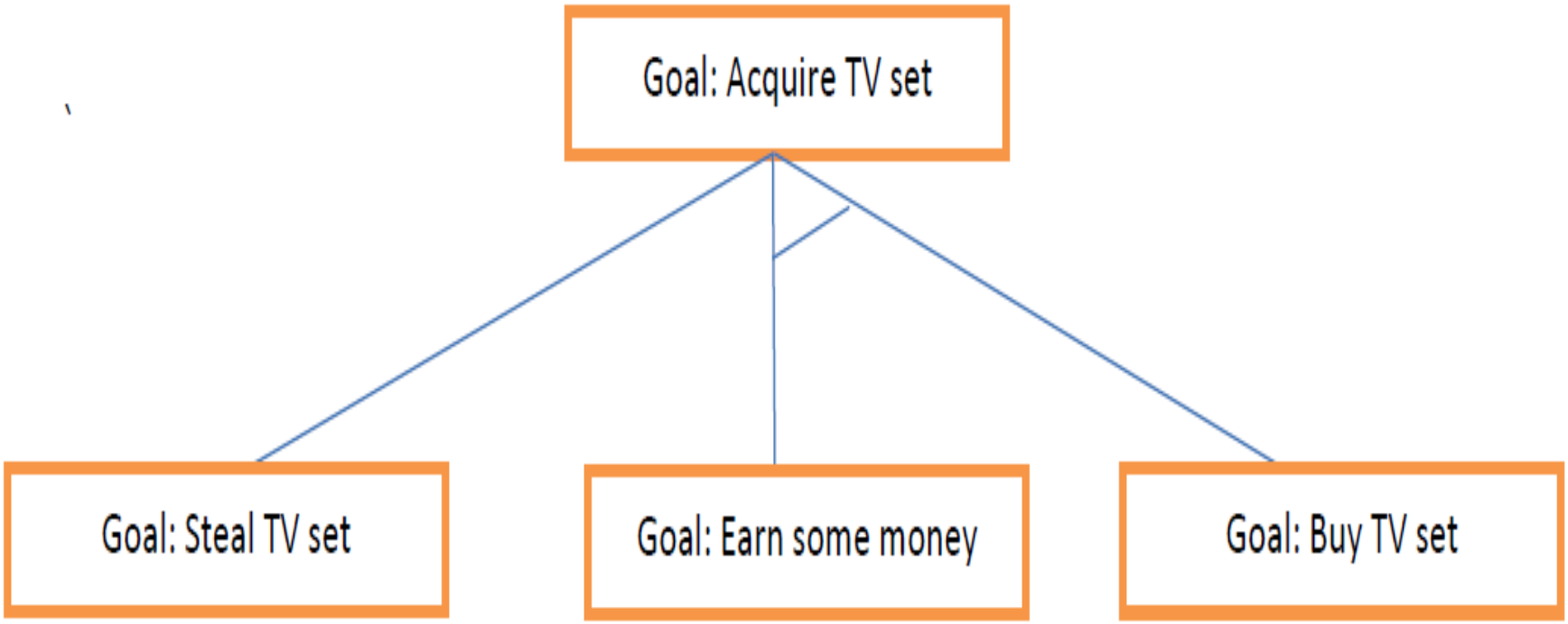
Problem reduction

- Complex problems are divided into simpler sub-problems using AND-OR graphs
- Solution of sub-problem may then be combined to obtain the final solution
- E.g. Tower of Hanoi need problem reduction
- Objective of the problem is to move entire stack of disks to another rod
- Only one disk may be moved at a time
- Each move consists of taking the uppermost disk and sliding it onto another
- No disk will be placed on top of a smaller disk

- Consider there are n disks on Rod1
- Move all n disks onto Rod2 making use of Rod3
- Lets develop an algorithm that can solve by reducing it to smaller problems
- The game tree that is generated will contain AND-OR arcs
- Solution involves the following steps
- If n=1, move the only disk from Rod1 to Rod2
- If n>1, then move all the top n-1 disks in the same order to other rod

- Move the largest disk from Rod1 to Rod2
- Move finally all n-1 disks of Rod3 to Rod2
- So the problem is reduced to moving n-1 disks from one rod to another
- First, from Rod1 to Rod3 and then from Rod3 to Rod2
- 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, etc. until only one disk is left





A SIMPLE AND-OR GRAPH

- Consider an AND-OR Graph where each arc with a single successor has a cost of 1
- Numbers given in () denote estimated cost of path
- Numbers given in [] represent revised cost of path
- Begin search from A & compute the heuristic values for its successors, i.e. B & (C,D) as 19 & (8,9) respectively
- Estimated cost of paths from A to B is 20 & from A to (C,D) is 19, which is a better path, So, expand this path, C to (G,H) & D to (I,J)
- Heuristic values of G,H,I & J are 3,4,8,7 resp., which lead to revised costs of C & D as 9 & 17 resp.
- The revised costs of path from A to (C,D) is calculated as 28, is not the best path now

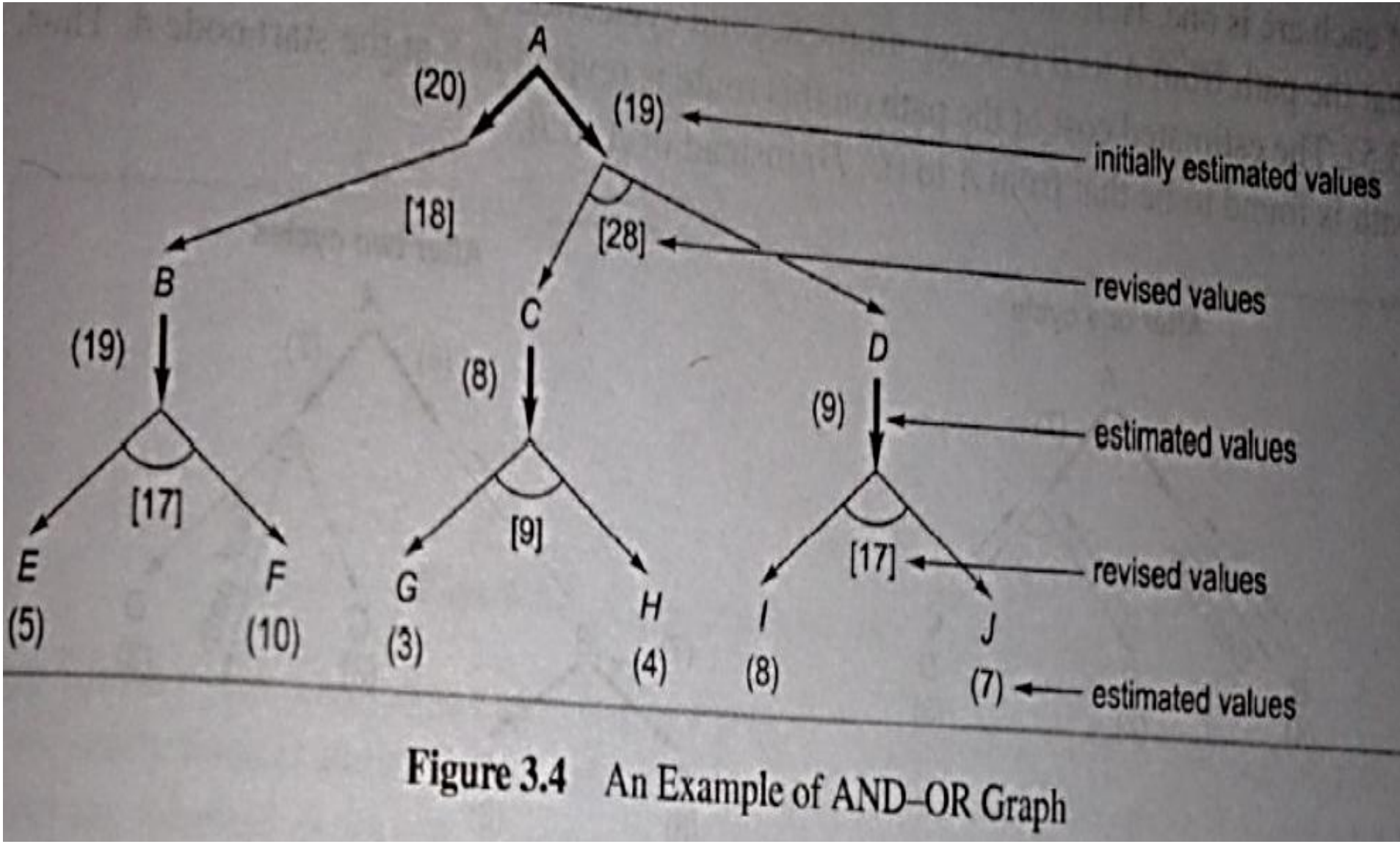


Figure 3.4 An Example of AND–OR Graph

- So chose the path from A to B for expansion, heuristic value of B is 17, best path
- 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

Node status labelling procedure

- 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 are:
  - **Terminal node**: cannot be expanded further, if this is the goal node, then label as ‘solved’ else ‘unsolved’
  - **Non-terminal AND node:** is labelled as unsolved as soon as one of its successors is found to be unsolvable, labelled as ‘solved ’ if all of its successors are solved
  - **Non-terminal OR node:** labelled as ‘solved’ if one of its successors is labelled ‘solved’; labelled as ‘unsolved’ if all of its successors are found to be unsolvable

Algorithmic Steps for AND–OR Graphs

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

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

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

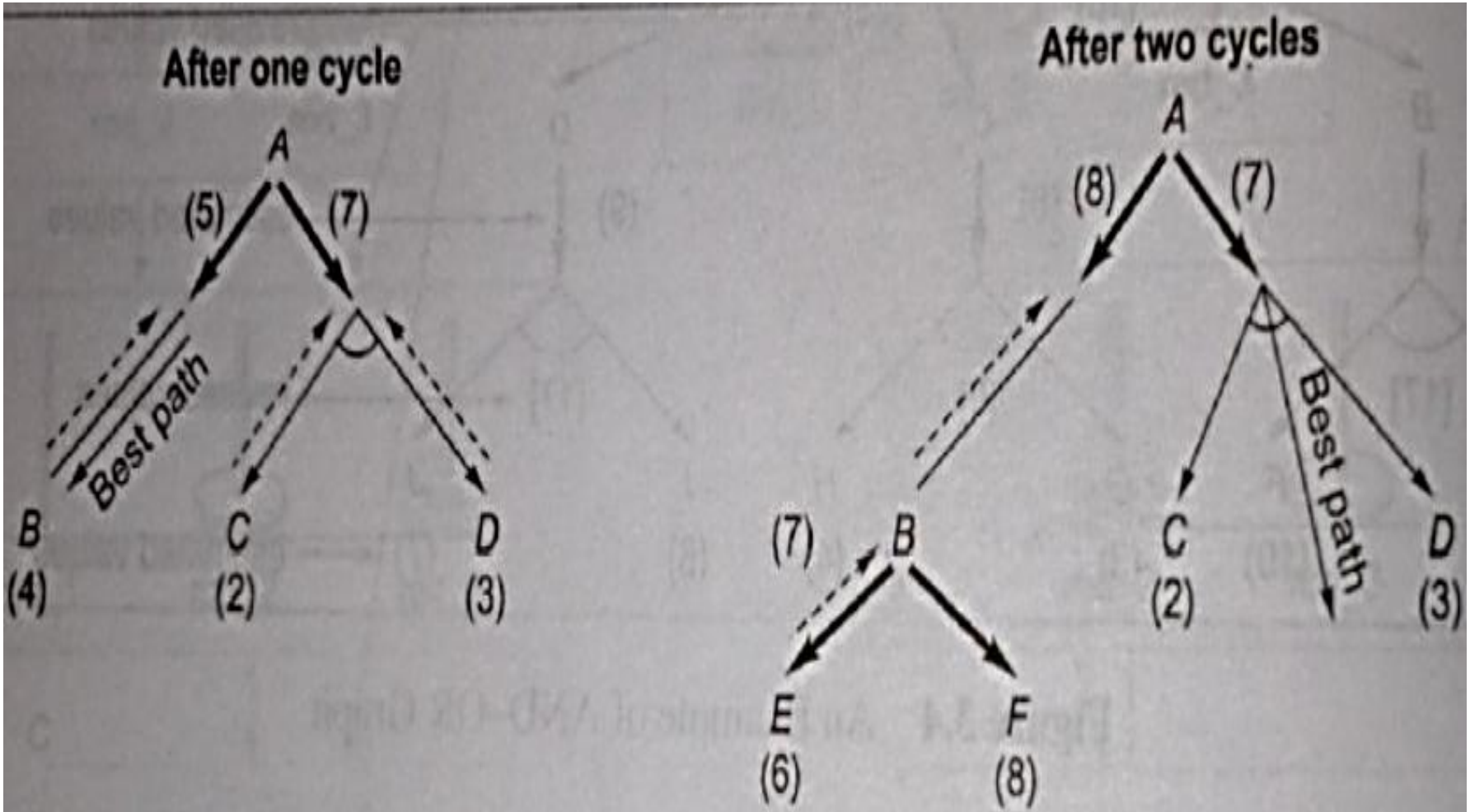


Figure 3.5 Labelling Procedure: First Two Cycles

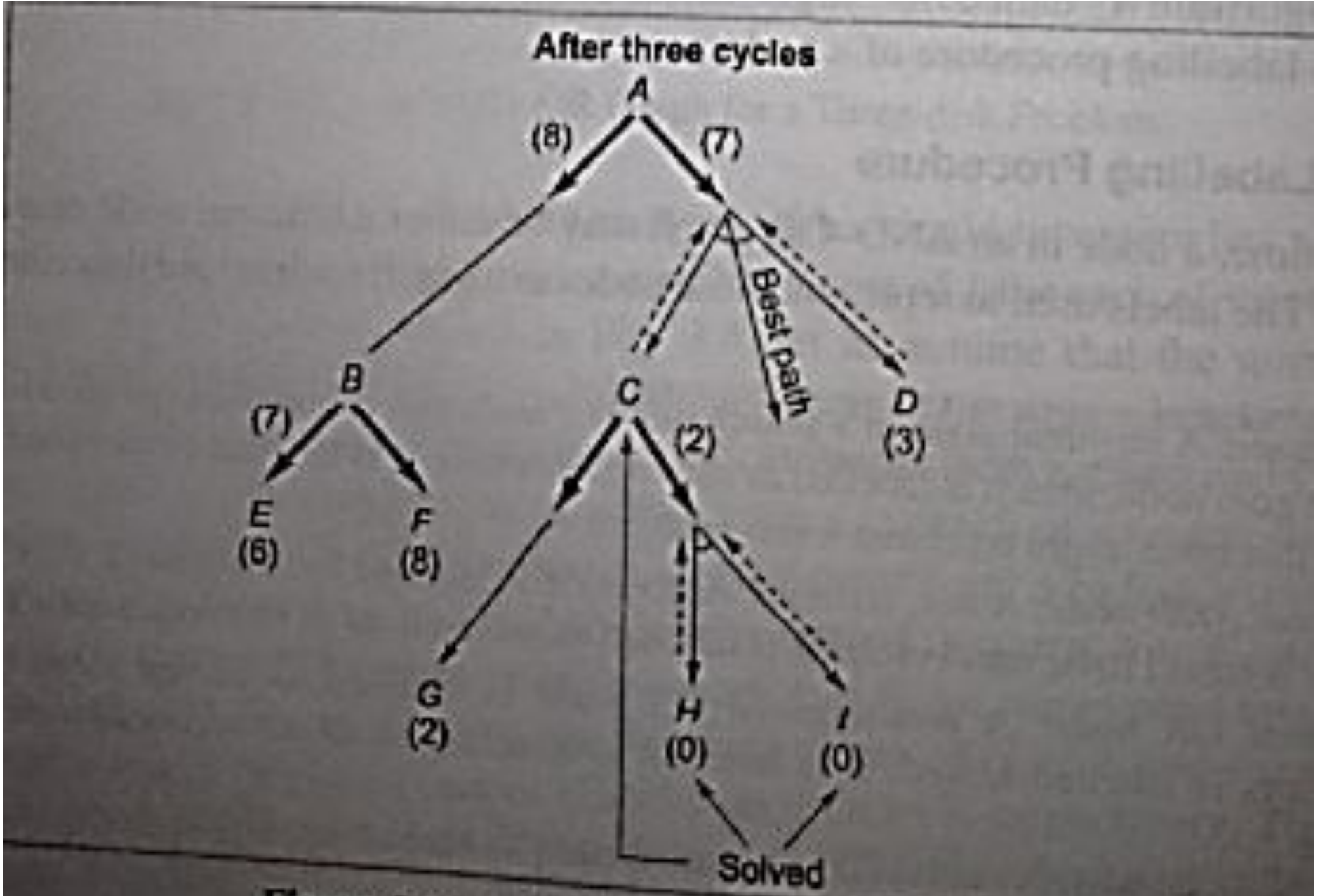


Figure 3.6 Labelling Procedure: Third Cycle

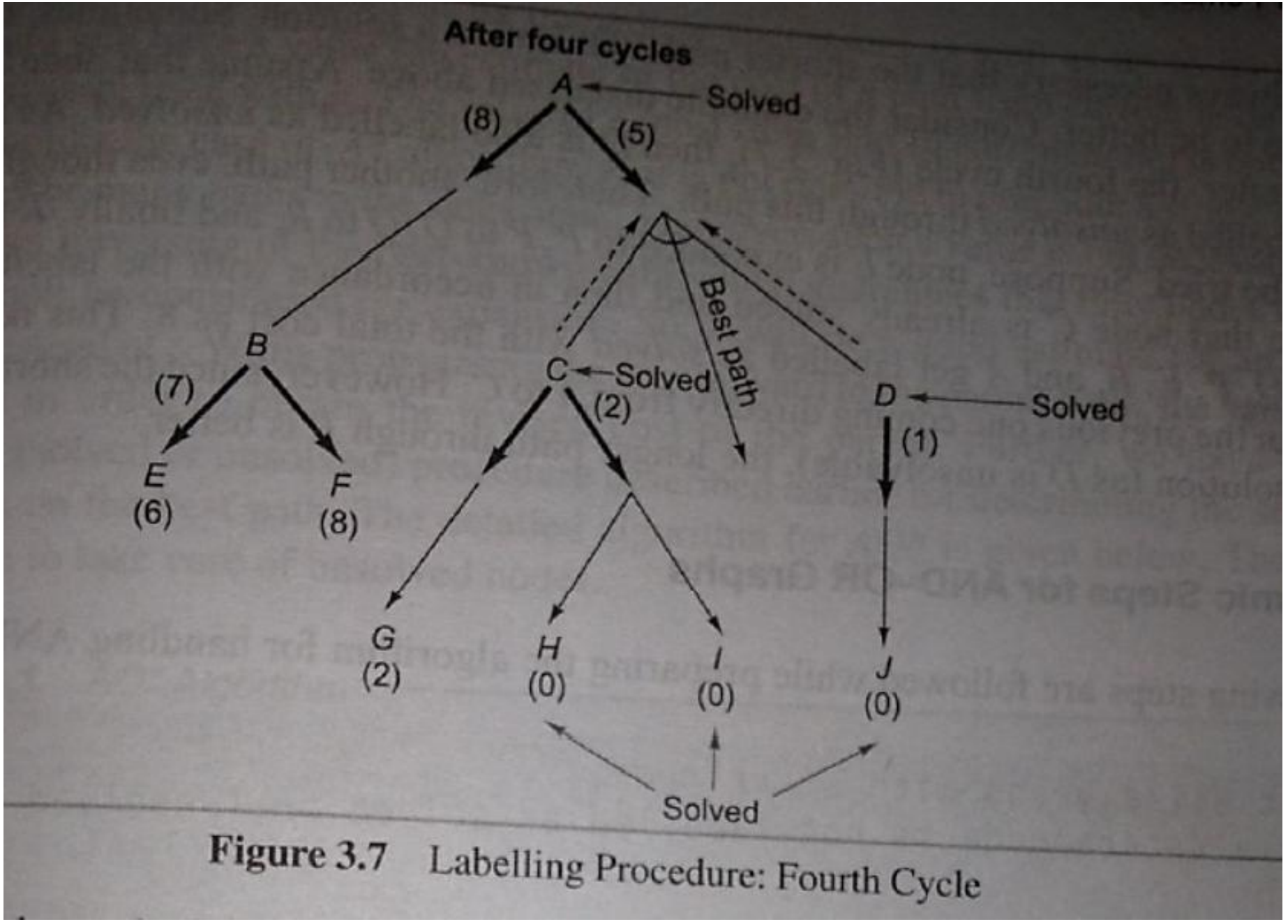


Figure 3.7 Labelling Procedure: Fourth Cycle

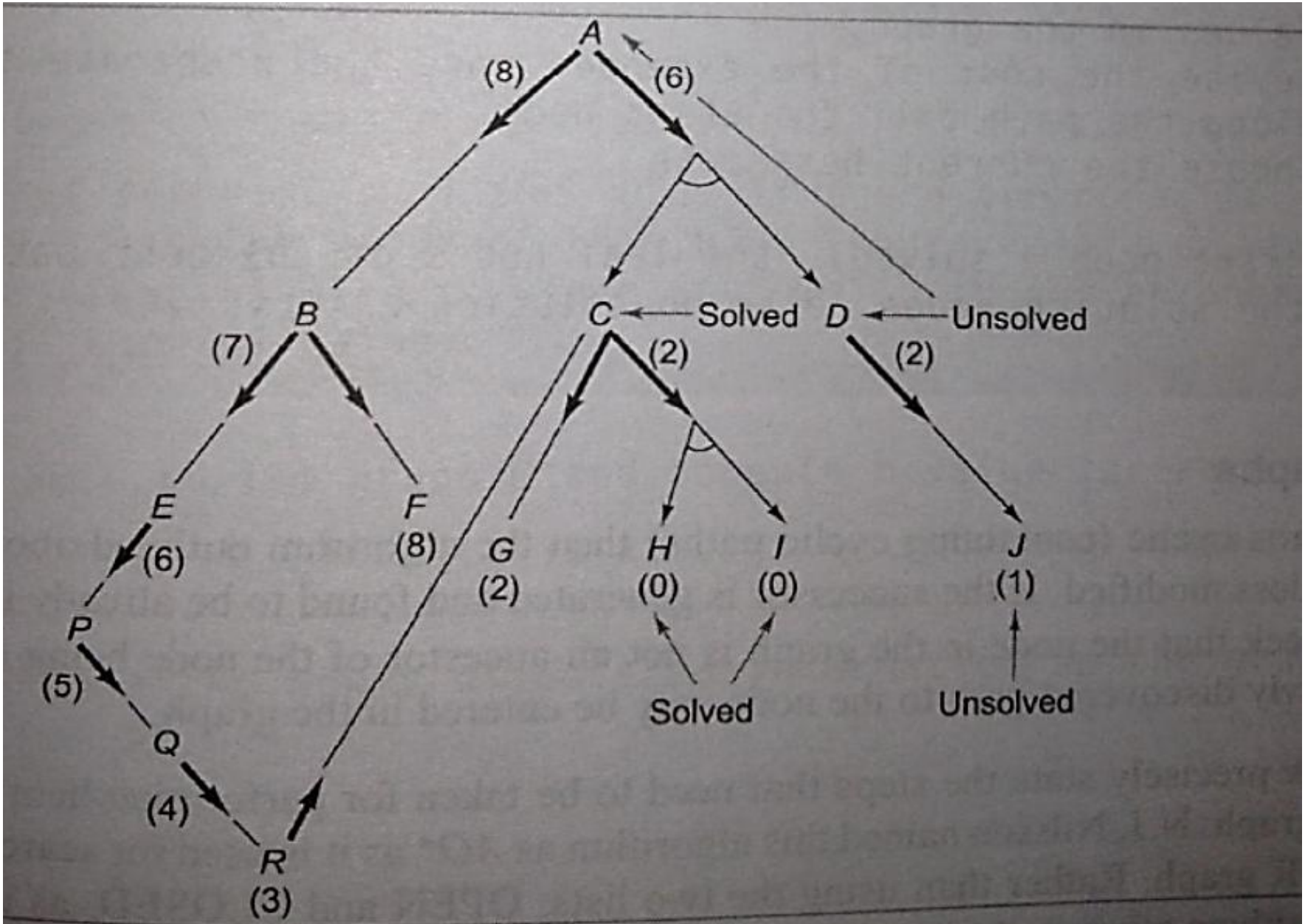


Figure 3.8 Non-Optimal Solution

Game playing

- A game is defined as a sequence of choices where each choice is made from a number of discrete alternatives
- Every sequence ends in a certain outcome and every outcome has a definite value for the opening player
- Only two-player games are considered in which both the players have exactly opposite goals

- Games can be classified into two types
  - **Perfect information games:** both the players have access to the same information about the game in progress, e.g. Tic-Tac-Toe, Chess, etc.
  - **Imperfect information games:** players do not have access to complete information about the game, e.g. cards, dice, etc.
- The study is restricted to discrete and perfect information games
- A game is said to be discrete if it contains a finite number of states

- A typical characteristic of a game is to look ahead at future positions to succeed
- Search procedures for two player games are:
  - **Game problem Vs state space problem**
  - **Status labelling procedure in Game tree**
  - **Nim game problem**

Game problem Vs state space problem

- State space problems have start & intermediate states, rules/operations and a goal state
- Game problems also have a start, legal moves and winning positions

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

- Game begins from a specified initial state and ends in win (for one player)/ loss (other)/ draw
- A game tree is an explicit representation of all possible plays of the game
- Root node is an initial position of game, its successors are the positions resulting from the second player's moves and so on
- Terminal nodes are represented by WIN/ LOSS/ DRAW
- Each path to a terminal node represents a different complete play of the game

- In game theory, maximum possible loss is minimized and minimum gain is maximized
- In game, two nodes are there, MAX & MIN
- MAX node tries to maximize its own game and minimizes (MIN) the opponent's game
- For this, generate the complete game tree with all possible moves and then decide which move is the best for MAX
- As a part of game playing, game trees labelled as MAX level and MIN level are generated alternately

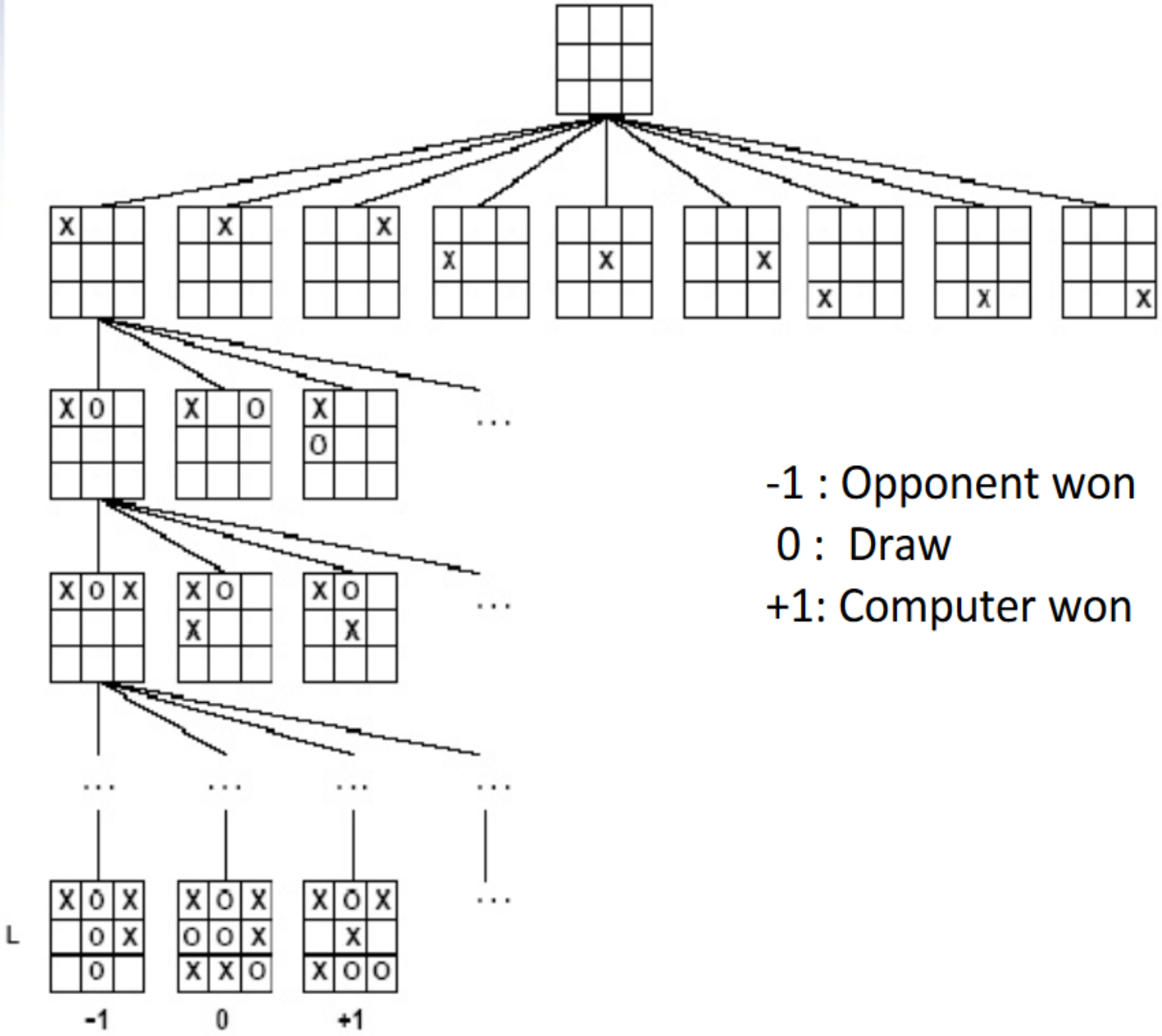
MIN(0)

MAX(X)

MIN(0)

TERMINAL

Utility



-1 : Opponent won

0 : Draw

+1: Computer won

Status labelling procedure in Game tree

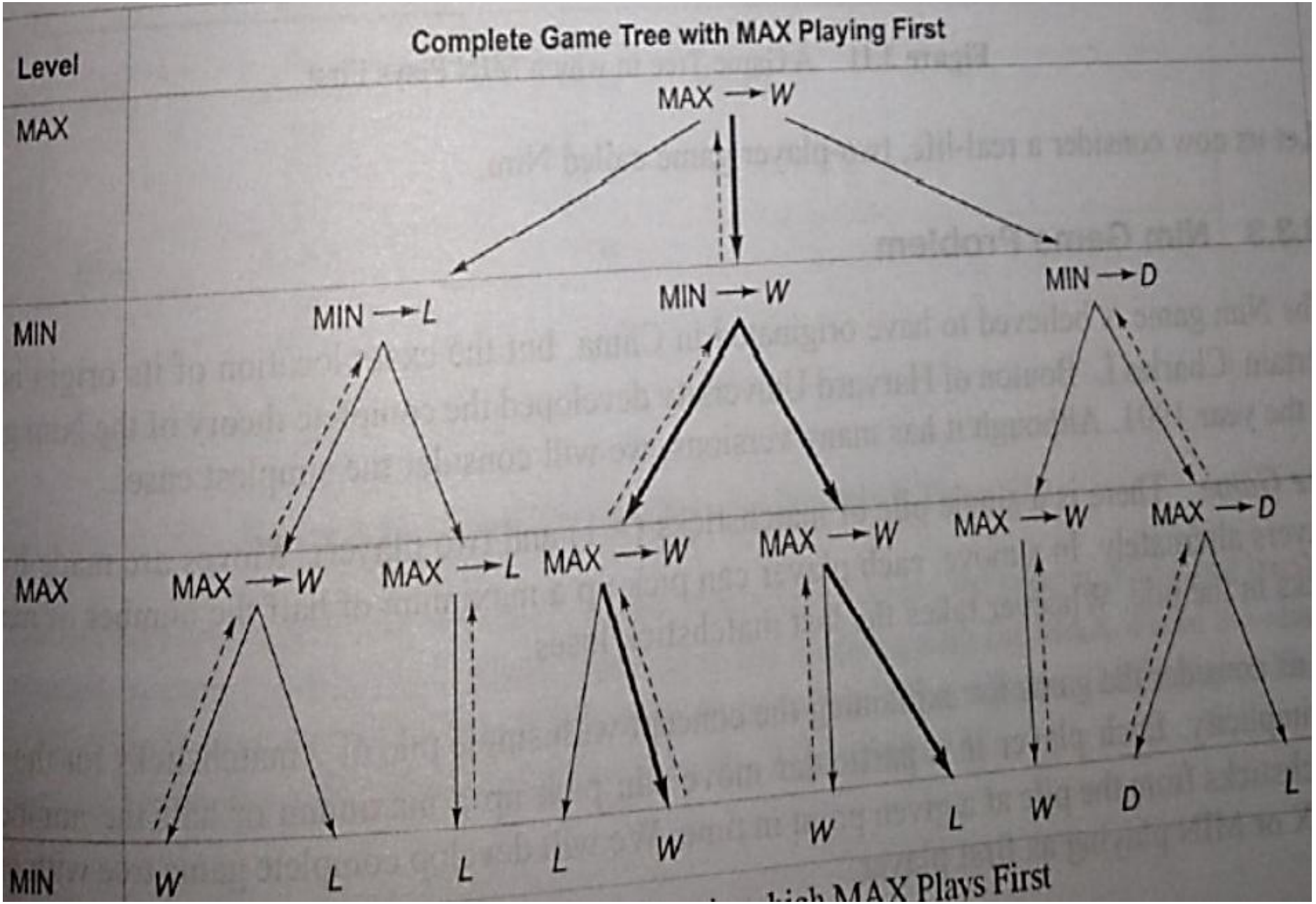
- Each level is labelled in the game tree according to the player who makes the move at that point
- Status labelling process is as follows

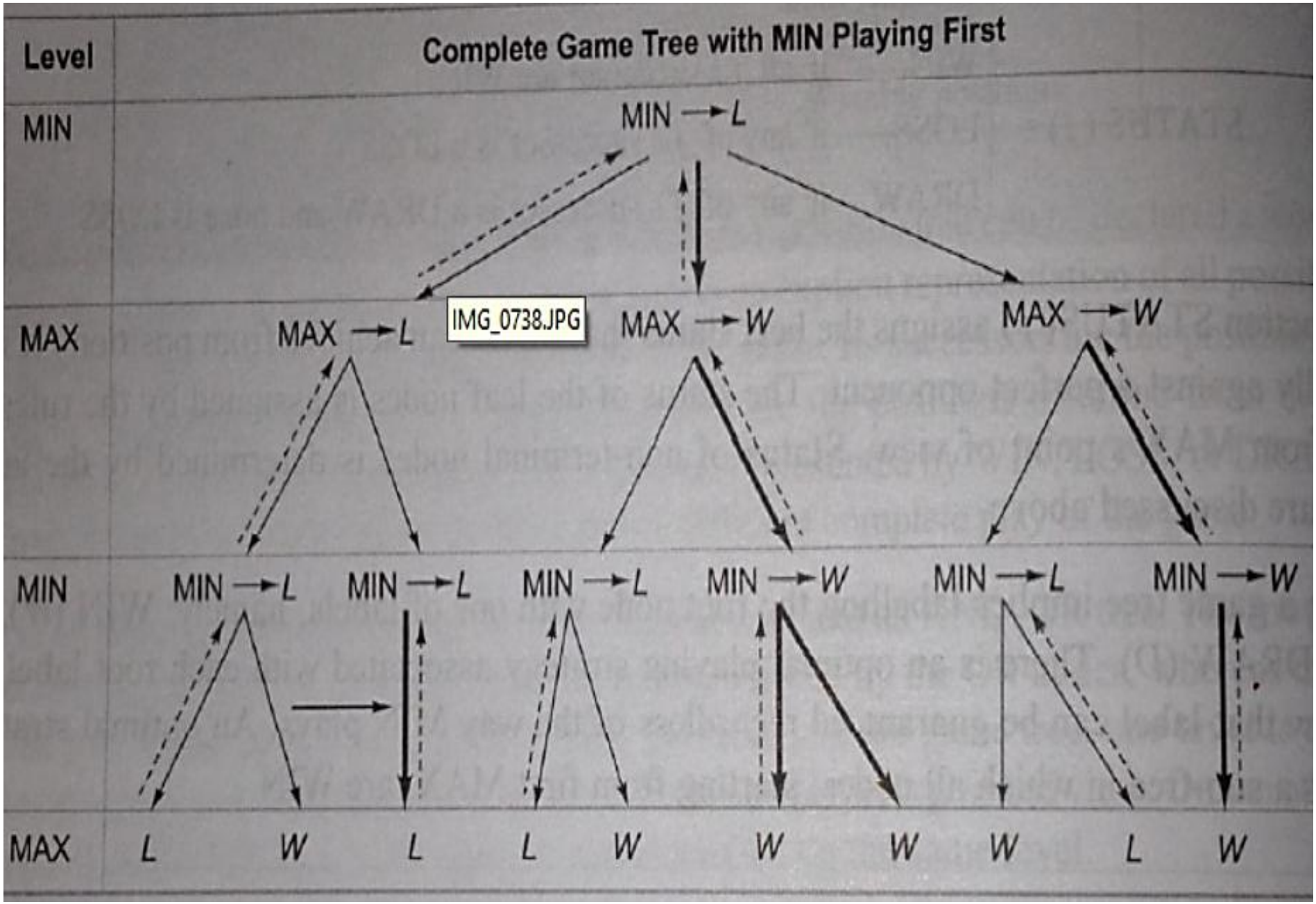
- If j is a non-terminal MAX node, then

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

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

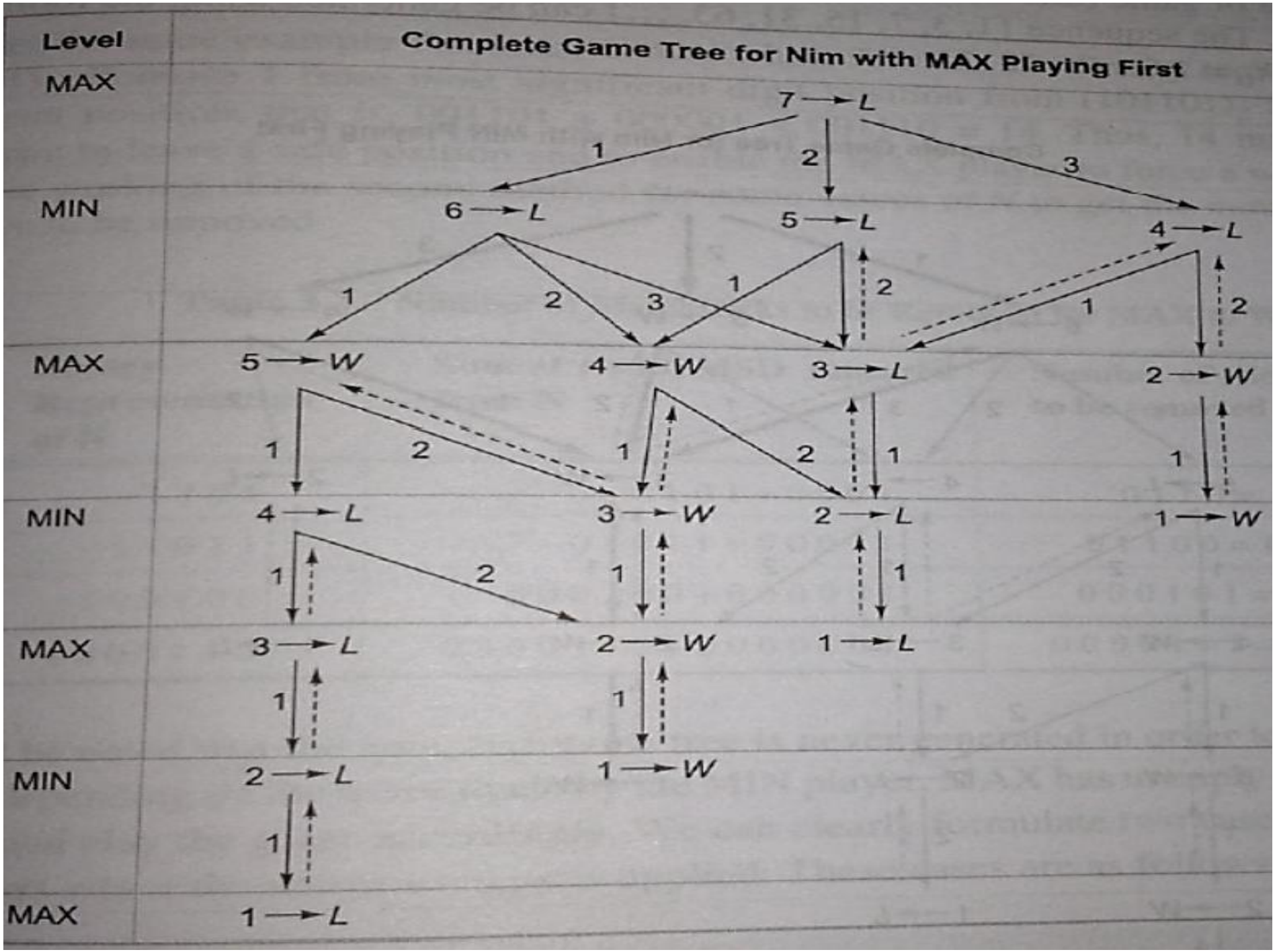
$$\text{STATUS}(j) = \begin{cases} \text{WIN}, & \text{if all } j\text{'s successor are WIN} \\ \text{LOSS}, & \text{if any of } j\text{'s successor is a LOSS} \\ \text{DRAW}, & \text{if any of } j\text{'s successor is a DRAW and none is LOSS} \end{cases}$$


Level
Complete Game Tree with MAX Playing First
MAX
MIN
MAX
MIN
MAX → W
MIN → L
MIN → W
MIN → D
MAX → W
MAX → L
MAX → W
MAX → W
MAX → W
MAX → D
W
L
L
L
W
W
L
W
D
L
which MAX Plays First

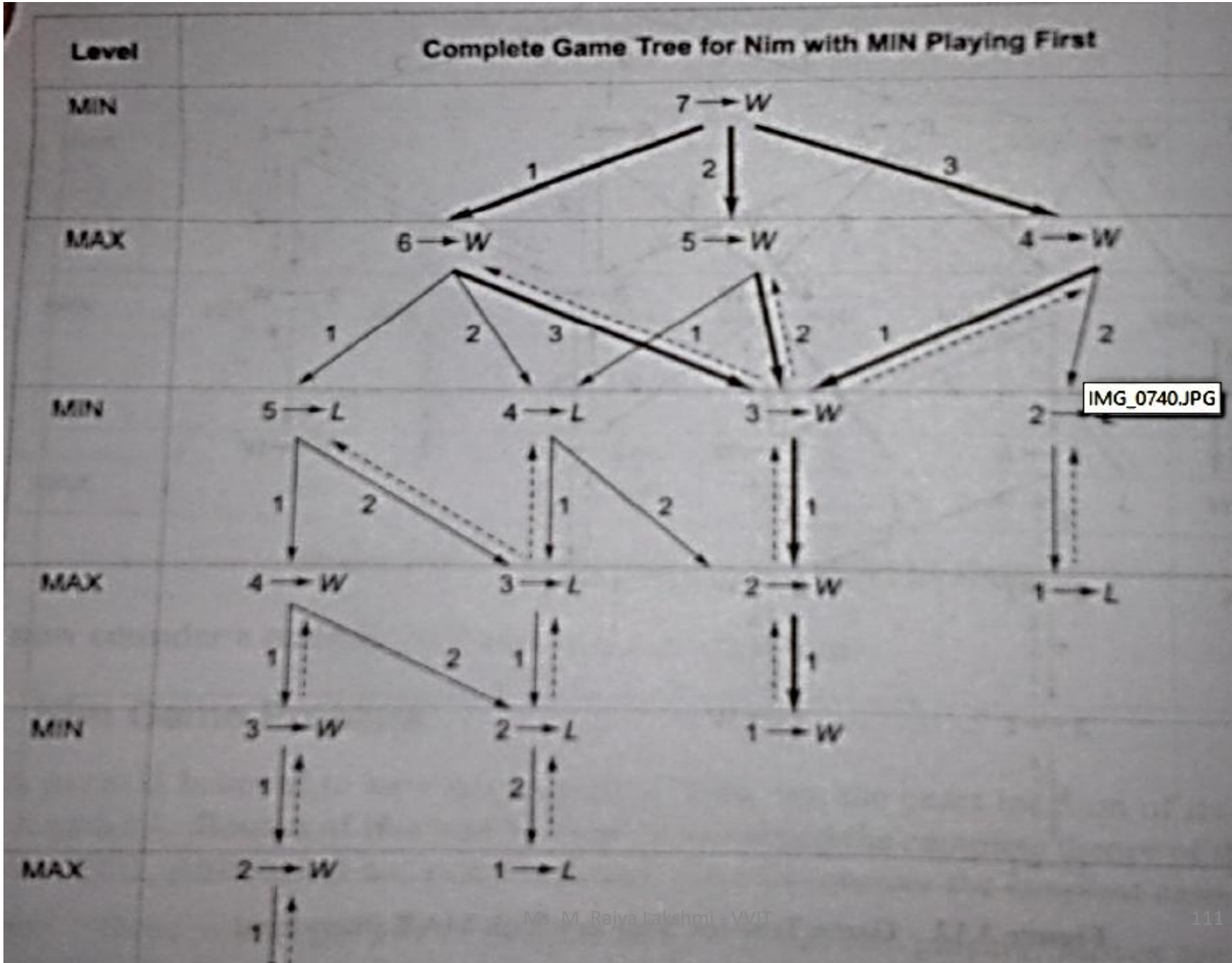

Level
Complete Game Tree with MIN Playing First
MIN
MAX
MIN
MAX
MIN → L
MAX → L
MAX → W
MAX → W
MIN → L
MIN → L
MIN → L
MIN → W
MIN → L
MIN → W
L
W
L
L
W
W
W
W
L
W

Nim game problem

- The Game: There is a single pile of match sticks (>1) and two players
- Moves are made by the players alternately
- Each player can pick up a max of half the no.of match sticks in the pile
- Whoever takes the last match stick, loses
- We will develop complete game tree with either MAX or MIN playing as first player


Level
Complete Game Tree for Nim with MAX Playing First
MAX
MIN
MAX
MIN
MAX
MIN
MAX
7 → L
6 → L
5 → L
4 → L
5 → W
4 → W
3 → L
2 → W
4 → L
3 → W
2 → L
1 → W
3 → L
2 → W
1 → L
2 → L
1 → W
1 → L
1
2
3

- Each node contains the total no.of sticks in the pile and is labelled as W / L with procedure
- If a single stick is left at the MAX level then it is assigned L, if one stick is left at the MIN level then W is assigned
- The label W/L is assigned from MAX's point of view at leaf nodes
- Arcs carry the no.of sticks to be removed
- Dotted lines show the propagation of status
- Thick lines show the winning path in following fig


Level
Complete Game Tree for Nim with MIN Playing First
MIN
MAX
MIN
MAX
MIN
MAX
7 → W
6 → W
5 → W
4 → W
5 → L
4 → L
3 → W
2 → L
4 → W
3 → L
2 → W
1 → L
3 → W
2 → L
1 → W
2 → W
1 → L
IMG_0740.JPG

- Strategy: if at the time of MAX's turn there are N sticks are in a pile, then MAX can force a win by leaving M sticks for the MIN to play
- M ϵ {1,3,7,15,31,63,…} using the rule of game
- The sequence can be generated using the formula $X_i=2X_{i-1}+1$, where $X_0=1$ for i>0
- To formulate a method which determine the number of sticks to be picked by MAX, there are two ways

- **First is**, from sequence figure out the closest number less than the given number N sticks
- The difference b/w N and that number gives the desired no.of sticks that have to be picked
- E.g. if N=45, closes no. is 31, then 45-31=14, hence 14 sticks are to be picked by MAX
- **Second is**, the desired number is obtained by removing the MSB from the binary representation of N and adding it to the LSB

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

Alphabeta pruning

- Also known as backward pruning
- Used to reduce no.of branches explored and no.of static evaluations applied
- Requires two threshold values,
  - one represents lower bound(α) on the value that a maximizing node may be assigned
  - another represents upper bound (β) on the value that a minimizing node may be assigned

- Each MAX node has an α value which never decreases and each MIN node has a β value which never increases

Two-player perfect information games