Artificial Intelligence

Unit II

B.Tech. V Semester CSE A, B, C, D

By
Dr. Loshma Gunisetti,
Associate Professor
CSE Department

Textbook:

1. Artificial Intelligence, Saroj Kaushik, 1st Edition, Cengage Learning

Reference Books:

1. Artificial Intelligence, Elaine Rich, Kevin Knight, Shivashankar B Nair, 3rd Edition, Tata McGraw Hill Education Private Limited., 2009
2. Artificial Intelligence- A modern Approach, 3rd Edition, Stuart Russel, Peter Norvig, Pearson Education

Course Outcomes

After successful completion of this course, the student will be able to:

CO	Course Outcomes	Knowledge Level
1	Illustrate the concept of Intelligent systems and current trends in AI	K2
2	**Apply Problem solving, Problem reduction and Game Playing techniques in AI**	**K3**
3	Illustrate the Logic concepts in AI	K2
4	Explain the Knowledge Representation techniques in AI	K2
5	Describe Expert Systems and their applications	K2
6	Illustrate Uncertainty Measures	K2

Introduction to Problem Solving

- **Problem solving** is a method of deriving solution steps beginning from initial description of the problem to the desired solution
- It is one the focus areas of AI and can be characterized as a systematic search using a range of possible steps to achieve some predefined solution
- 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

Introduction to Problem Solving

- There are **two** types of problem solving methods:
  - **General-purpose method**: Applicable to a wide variety of problems
  - **Special -purpose method**: Tailor-made for a particular problem
- The most general approach for solving a problem is to generate the solution and test it
- For generating a new state in the search space, an action/ operator/ rule is applied and tested whether the state is the goal state or not. In case the state is not the goal state, the procedure is repeated
- The order of application of the rules to the current state is called Control Strategy

General Problem Solving

Production System (PS)

- PS is one of the formalisms that helps AI programs to do search process more conveniently in state space problems
- This system comprises of start (initial) state(s) and goal(final) state(s) of the problem along with one or more databases consisting of suitable and necessary information for the particular task
- PS consists of number of production rules in which each production rule has left side that determines the applicability of the rule and a right side that describes the action to be performed

General Problem Solving

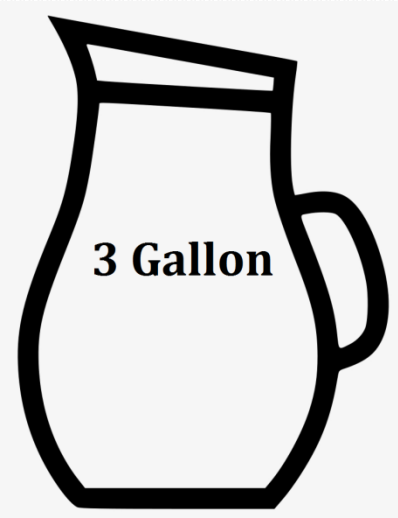
Production System (PS)

- PS consists of control strategies that specify the sequence in which the rules are applied when several rules match at once
- One of the examples of PS is an Expert System which is used for expert opinion in a specific domain
- Advantages of PS:
  - It is a good way to model the strong state-driven nature of intelligent action
  - New rules can be easily added to account for new situations without disturbing the rest of the system
  - It is quite important in real-time environment and applications where new input to the database changes the behaviour of the system

General Problem Solving

Water Jug Problem: We have two jugs, a 5-gallon and 3-gallon capacity respectively with no measuring marker on them. There is endless supply of water through tap

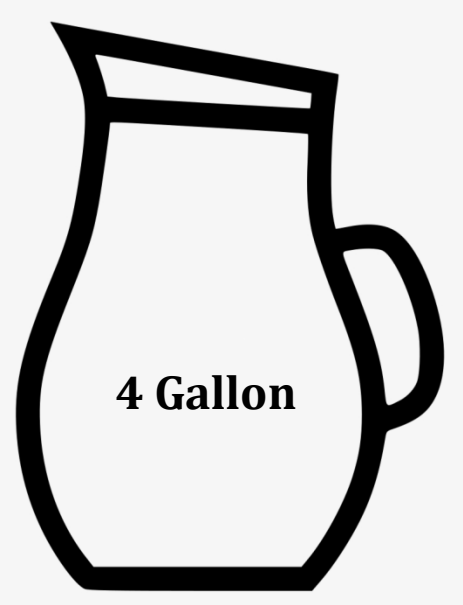
5 -g Jug



3 -g Jug

General Problem Solving

Water Jug Problem : Our task is to get 4 gallon of water in the 5-g jug



5 -g Jug

3 -g Jug

General Problem Solving : Water Jug Problem

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

- **Start state**: (0,0)

- **Goal state**: (4,N) *where N ≤ 3*

General Problem Solving : Water Jug Problem

Possible operations that can be used in this problem are as follows:

- Fill the 5-g jug from the tap and empty the 5-g jug by throwing water down the drain
- Fill the 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 water into the 5-g jug

These rules can be formally be defined as **production rules**

Table 1: Production rules for Water Jug problem

Rule No.	Left of Rule	Right of Rule	Description
1	(X, Y | X < 5)	(5, Y)	Fill 5-g jug
2	(X, Y | X > 0)	(0, Y)	Empty 5-g jug
3	(X, Y | Y < 3)	(X, 3)	Fill 3-g jug
4	(X, Y | Y > 0)	(X, 0)	Empty 3-g jug
5	(X, Y | X +Y ≤ 5 ∧ Y > 0)	(X+Y, 0)	Empty 3-g into 5-g jug
6	(X, Y | X +Y ≤ 3 ∧ X > 0)	(0, X+Y)	Empty 5-g into 3-g jug
7	(X, Y | X +Y ≥ 5 ∧ Y > 0)	(5, Y – (5-X)) **Until the 5-g jug is full**	Pour water from 3-g jug into 5-g jug
8	(X, Y | X +Y ≥ 3 ∧ X > 0)	(X-(3-Y), 3) **Until the 3-g jug is full**	Pour water from 5-g jug into 3-g jug

Table 2: Solution Path 1

Rule applied	5-g jug	3-g jug	Step No.
Start State	0	0	
1	5	0	1
8	2	3	2
4	2	0	3
6	0	2	4
1	5	2	5
8	4	**3**	6
Goal State	4	-	

•There could be more than one solutions for a given problem

Table 3: Solution Path 2

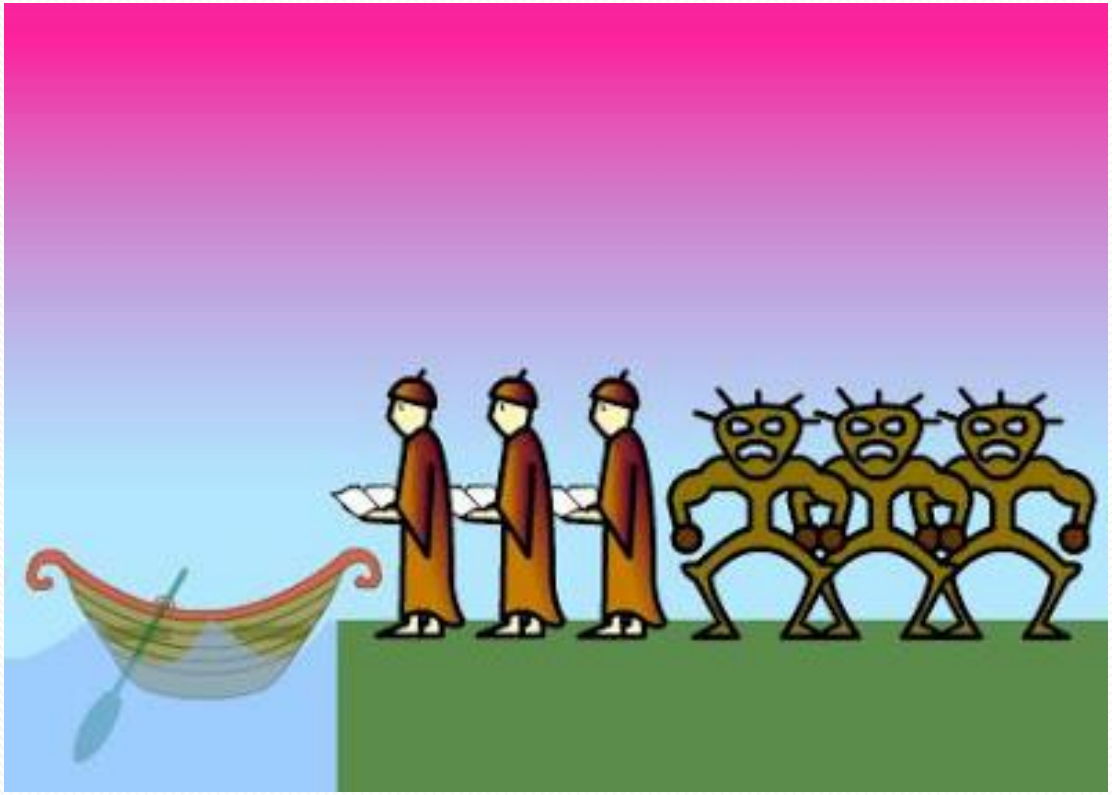
Rule applied	5-g jug	3-g jug	Step No.
Start State	0	0	
3	0	3	1
5	3	0	2
3	3	3	3
7	5	1	4
2	0	1	5
5	1	0	6
3	1	3	7
5	4	0	8
Goal State	4	-	

General Problem Solving : Missionaries and Cannibals Problem

Problem Statement :

Three Missionaries(M) and three Cannibals(C) 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 a river in such a way that Cannibals never outnumber the 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?

General Problem Solving : Missionaries and Cannibals Problem



General Problem Solving : Missionaries and Cannibals Problem

State Space :

- Set of ordered pairs of left(L) and right(R) banks of the river (L,R)
- Each bank is represented as a list [nM, mC, B]

where n – number of Missionaries

m – number of Cannibals

B – represents the boat

General Problem Solving : Missionaries and Cannibals Problem

State Space :

1. Start State : ([3M, 3C, 1B] , [0M, 0C, 0B])
2. Any state : ([n_1M, m_1C, _] , [n_2M, m_2C, _]) , with constraints 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 on either side
3. Goal State: ([0M,0C,0B],[3M, 3C, 1B])

Example Solution : https://www.youtube.com/watch?v=CgW67TBN8zQ

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_2 M, (m_2+1)C, 1B])
Rules for boat going 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_2 M, (m_2-1)C, 0B])

Solution Path for Missionaries and Cannibals Problem

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

- State space is another method of problem representation that facilitates easy search
- Using this method, one can also find a path from start state to goal state while solving a problem
- A state space basically consists of four components:
    1. A set S containing start states of the problem
    2. A set G containing goal states of the problem
    3. Set of nodes (states) in the graph/tree. Each node represents the state in problem-solving process
    4. Set of arcs connecting nodes. Each arc corresponds to operator, that is a step in a problem-solving process

State Space Search

- A solution path is a path through the graph from a node S to a node G
- The main objective of the algorithm is to determine a solution path in the graph
- There may be more than one ways of solving a problem
- One solution path is selected 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

State Space Search

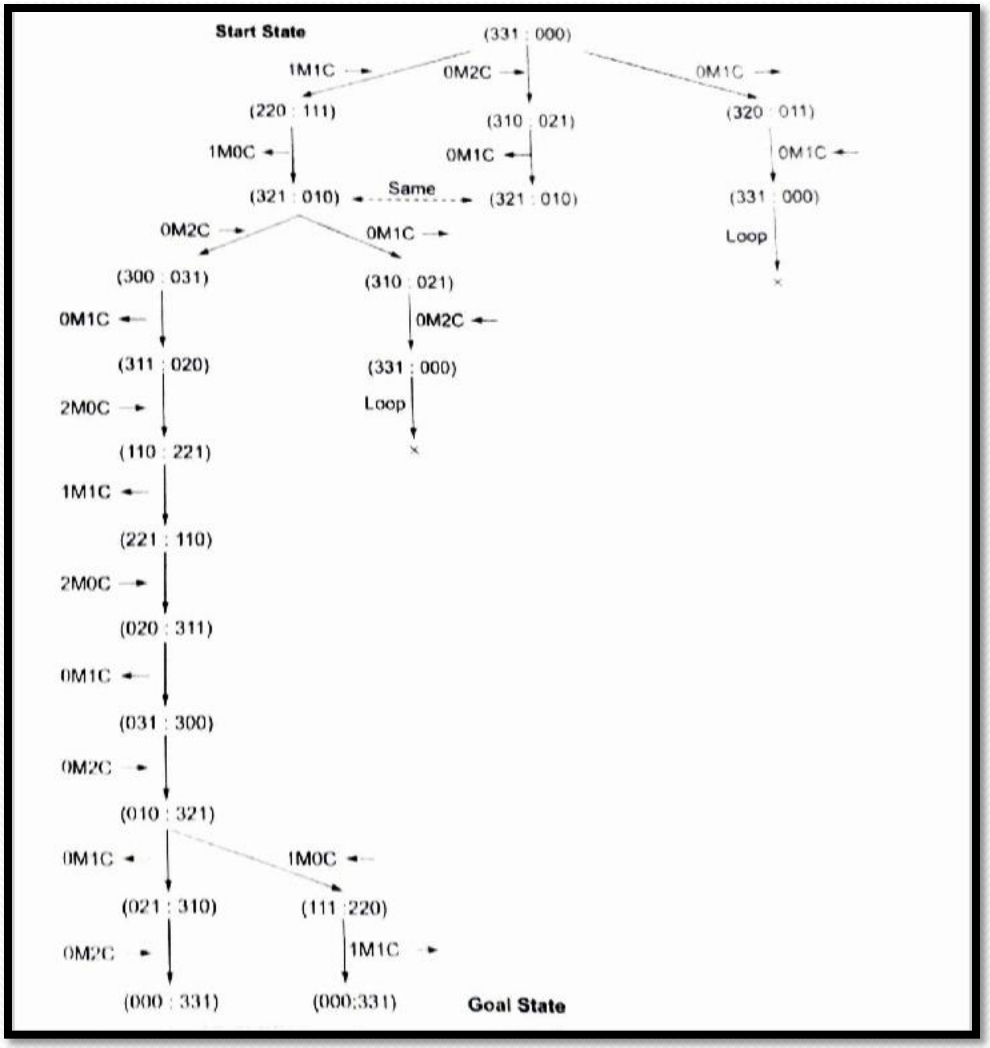
- Considering again the problem of Missionaries and Cannibals
- Possible operators are :

  {2M0C, 1M1C, 0M2C, 1M0C, 0M1C}
- **M** –Missionary, **C-** Cannibal
- If boat is on the left bank, then we write the 'Operator→' and if boat is on the right bank, then we write 'Operator ←'
- State is represented as (L : R) where L=n_1M m_1C 1B and R=n_2M m_2C 0B
- **Start State**: (3M3C1B : 0M0C0B) or simply (331:000)
- **Goal State**: (0M0C0B : 3M3C1B) or simply (000:331)

State Space Search

- We need to filter Invalid States, illegal operators not applicable to some states and some states that are not required at all
- For eg. An invalid state like (1M2C1B : 2M1C0B) is not a possible state
- In case of a valid state like (2M2C1B : 1M1C0B) , the operator 0M1C or 0M2C would be illegal
- Applying the same operator both ways would be a waste of time, since we return to a previous state. It is called a Looping situation. Looping may occur after few steps. Such operations should be avoided

Search Space

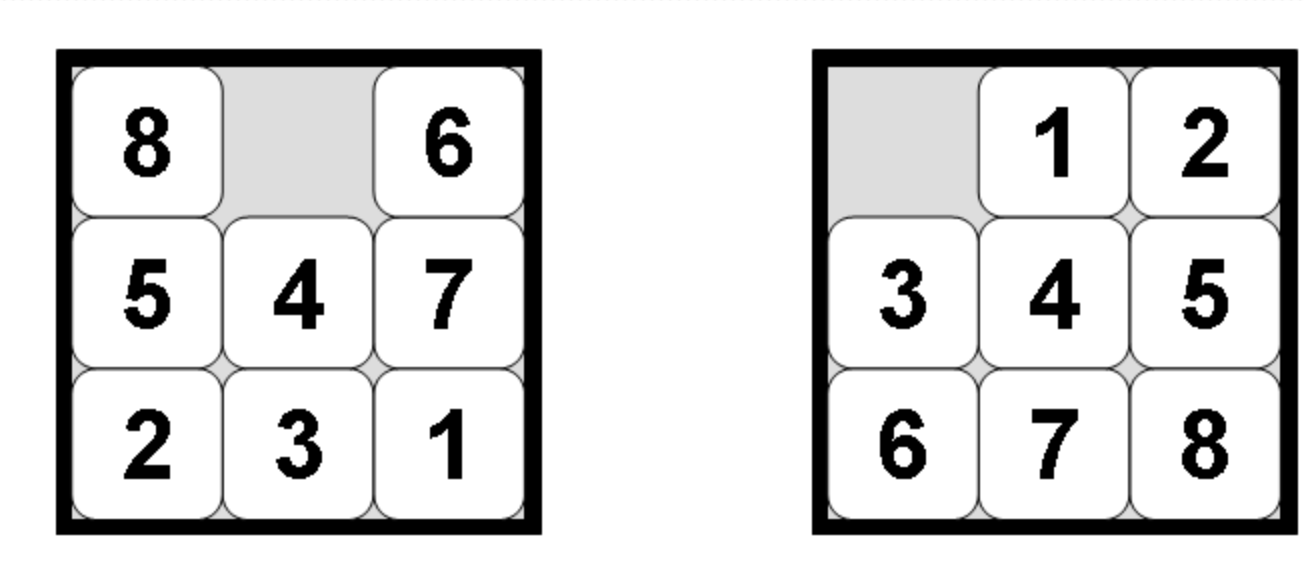


Two solution paths

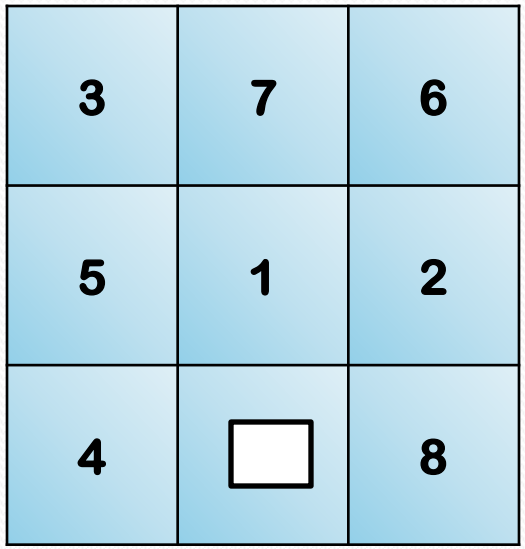
Solution Path 1	Solution Path 2
1M1C →	1M1C →
1M0C ←	1M0C ←
0M2C →	0M2C →
0M1C ←	0M1C ←
2M0C →	2M0C →
1M1C ←	1M1C ←
2M0C →	2M0C →
0M1C ←	0M1C ←
0M2C →	0M2C →
0M1C ←	1M0C ←
0M2C →	1M1C →

The Eight Puzzle Problem

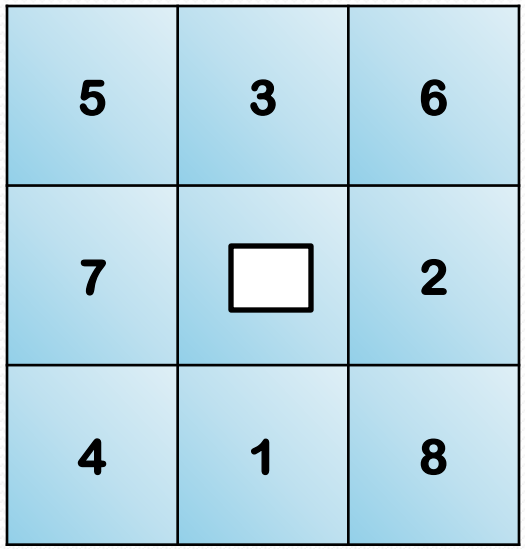
Problem Statement : It has a 3 x 3 Grid with 8 randomly numbered (1 to 8) tiles arranged on it with one empty cell. At any point, the adjacent tile can move to the empty cell, creating a new empty cell. Tiles are arranged such that we get the goal state from start state.



The Eight Puzzle Problem



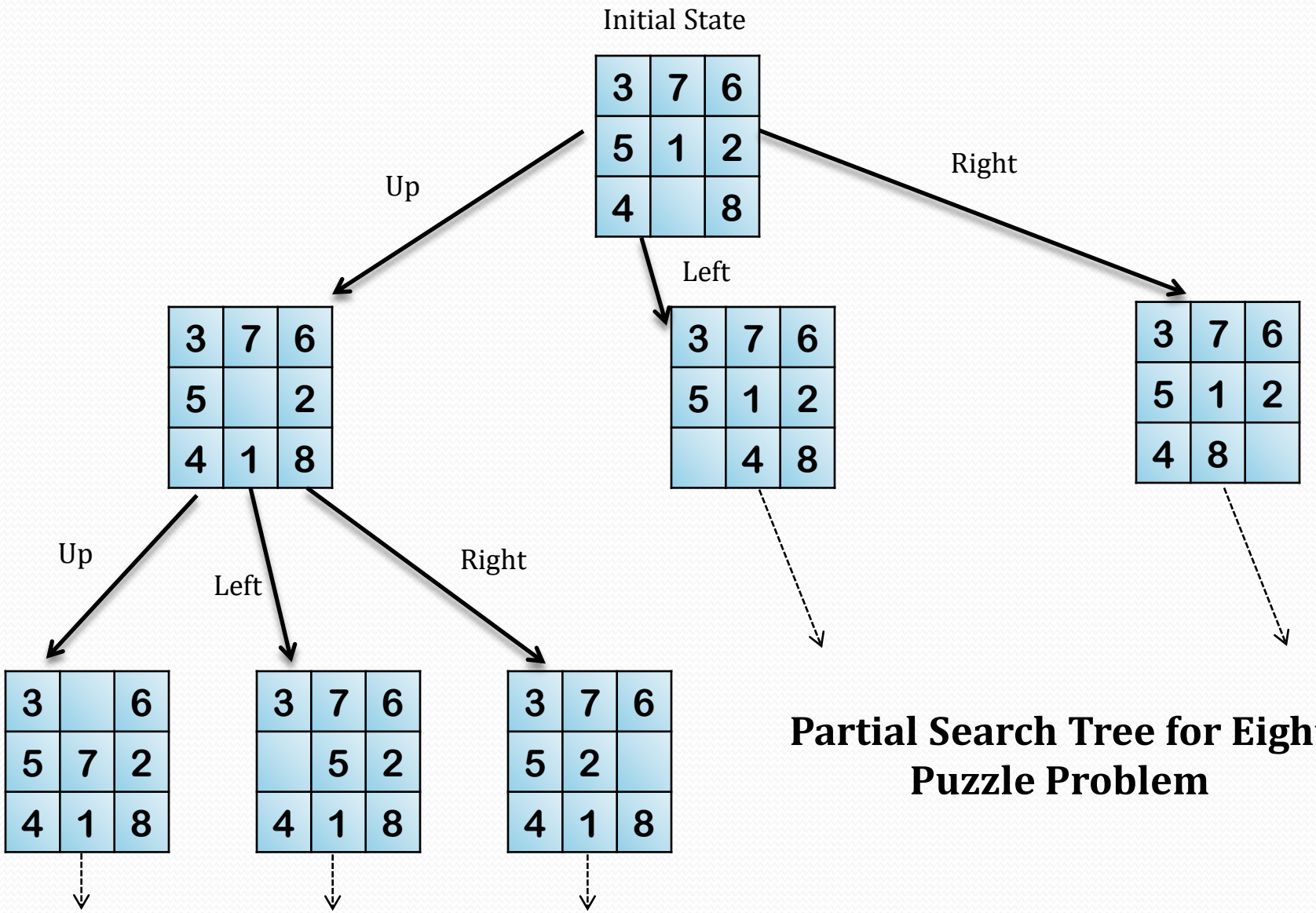
Start State



Goal State

The Eight Puzzle Problem

- A state for this should keep track of the position of all tiles on the game board
- Empty cell on the board is represented by 0
- Start State : [[3, 7, 6] , [5, 1, 2] , [4, 0, 8]]
- Goal State: [[5, 3, 6] , [7, 0, 2] , [4, 1, 8]]
- Valid operators – { Up, Down, Left, Right } , the direction in which blank space effectively moves



Partial Search Tree for Eight Puzzle Problem

Control Strategies

- Control Strategy is one of the most important components of problem solving
- It describes the order of application of the rules to the current state
- It should cause motion towards a solution
- One approach is to select a rule randomly from the applicable rules. This approach is not systematic.
- Breadth First and Depth First search are Systematic control strategies. They are exhaustive, uninformed, blind searches in nature
- If the problem is simple, then any control strategy that causes motion and is systematic will lead to a solution.
- To solve some real world problems, effective control strategy must be used.

Control Strategies

There are two directions in which a search could proceed

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

- **Forward Chaining**- The process of forward chaining begins with known facts and works towards a conclusion.

  For eg. Language OPS5 uses forward reasoning rules. Rules are expressed in the form of if-then rules.

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

Control Strategies

- The computational effort in both the strategies is the same.
- Same state space is searched but in different order.
- If there are a large set of explicit goal states and one start state, then it would not be efficient to solve using backward chaining strategies because we do not know which goal state is closest to the start state. Forward Chaining is better in such situations.
- Move from the smaller set of states to the larger set of states and proceed in the direction with the lower branching factor (the average number of nodes that can be reached directly from single node)
- Theorem proving from a small set of axioms uses backward strategy as the branching factor is significantly greater going forward from axioms to theorem rather than going from theorems to axioms.

Characteristics of Problem

- **Type of problem**: There are three types of problems in real life:
  - **Ignorable**-These are the problems where we can ignore the solution steps

    For eg. 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
  - **Recoverable**-These are the problems where the solutions steps can be undone

    For eg. In Water Jug Problem, if we have filled the jug, we can empty it also
  - **Irrecoverable**- The problems where the solution steps cannot be undone

    For eg. Any two player game such as Chess, Playing Cards, Snake and Ladder

Characteristics of Problem

- **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 solutions
- **Role of Knowledge:** Knowledge plays an important role in solving any problem. Knowledge could be in the form of rules and facts which help generating search space for finding the solution
- **Consistency of Knowledge Base used in solving problem:** Make sure that knowledge base used to solve problem is consistent. Inconsistent knowledge base will lead to wrong solutions

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

Characteristics of Problem

- **Requirement of Solution –** We should analyze the problem whether solution required is **absolute** or **relative**.
  - We call a solution to be **absolute** if we have to find exact solution.
    For eg. Water Jug problem –Any path problem-Solved in reasonable amount of time
  - A solution is **relative** if we have reasonably good and approximate solution.
    For eg. Travelling salesman problem-Best path problem- Computationally harder compared to any-path problem.

Exhaustive Searches

Systematic uninformed exhaustive searches:

1. **Breadth-First Search**
2. **Depth-First Search**
3. **Depth-First Iterative Deepening Search**
4. **Bidirectional Search**

1. Breadth-First Search(BFS)

- The Breadth-First search expands all the states one step away from the start state and then expands all states two steps from start state, then three steps, etc., until a goal state is reached.
- All successor states are examined at the same depth before going deeper.
- The BFS always gives an optimal path or solution.
- This search is implemented using two lists called OPEN and CLOSED
- The OPEN list contains those states that are to be expanded and CLOSED list keeps track of states already expanded
- OPEN list is maintained as a queue and CLOSED list as a stack.

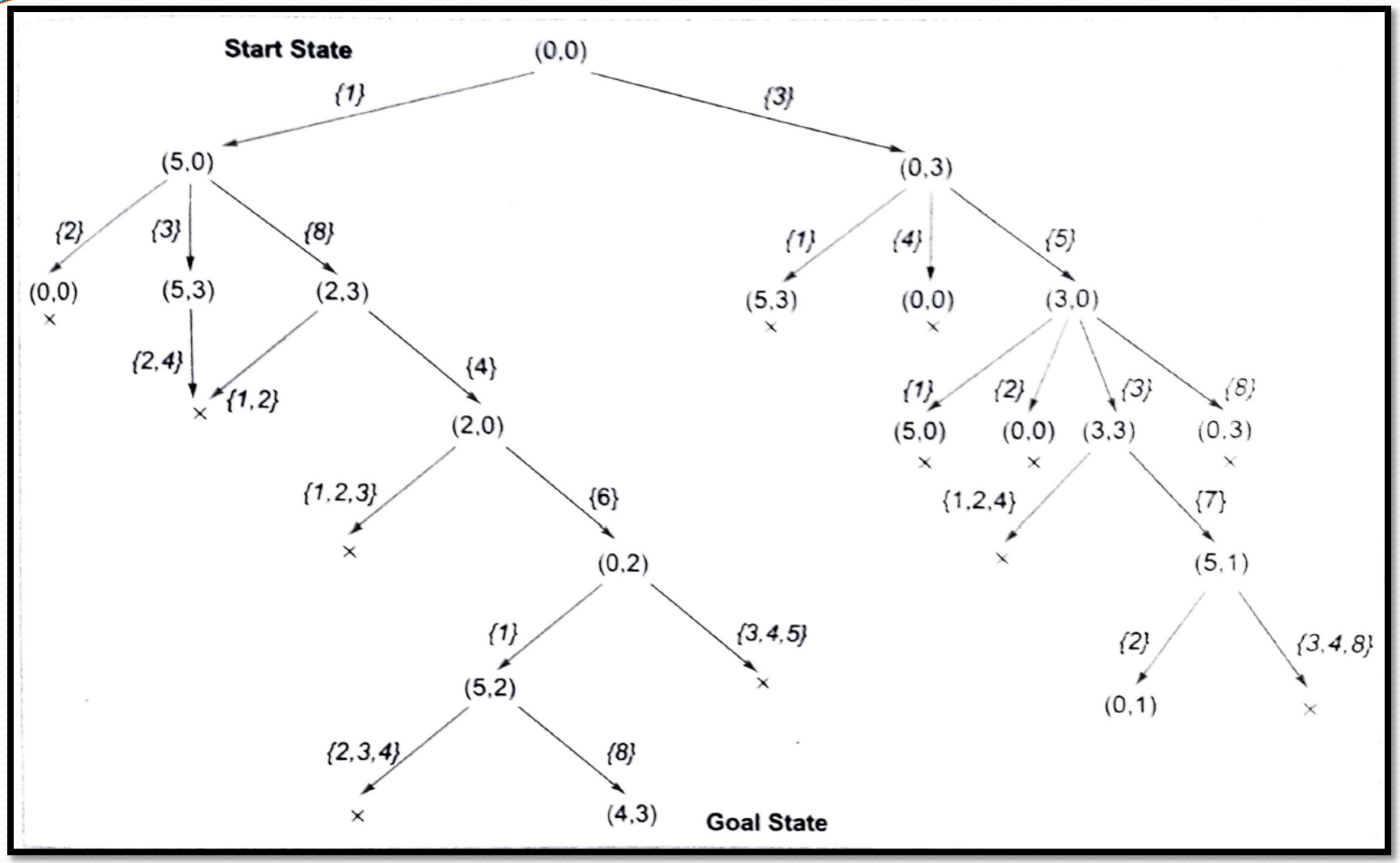
Breadth-First Search(BFS) Algorithm

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

- Initialize OPEN list with START and CLOSED =Φ;
- FOUND = false;
- while (OPEN ≠ Φ and FOUND =false) do
  {
    - remove the first state from OPEN and call it STATE-X;
    - put STATE-X in the front of CLOSED list; /*maintained as Stack*/
    - if STATE-X = GOAL then FOUND = true else
      {
        - perform EXPAND operation on STATE-X, producing a list of SUCCs;
        - remove from successors those states, if any, that are in the CLOSED list;
        - append SUCCs at the end of the OPEN list; /* queue*/
      }
  }
- If FOUND = true then return **Yes** else return **No**
- Stop

Table 1: Production rules for Water Jug problem

Rule No.	Left of Rule	Right of Rule	Description
1	(X, Y | X < 5)	(5, Y)	Fill 5-g jug
2	(X, Y | X > 0)	(0, Y)	Empty 5-g jug
3	(X, Y | Y < 3)	(X, 3)	Fill 3-g jug
4	(X, Y | Y > 0)	(X, 0)	Empty 3-g jug
5	(X, Y | X +Y ≤ 5 ∧ Y > 0)	(X+Y, 0)	Empty 3-g into 5-g jug
6	(X, Y | X +Y ≤ 3 ∧ X > 0)	(0, X+Y)	Empty 5-g into 3-g jug
7	(X, Y | X +Y ≥ 5 ∧ Y > 0)	(5, Y – (5-X)) **Until the 5-g jug is full**	Pour water from 3-g jug into 5-g jug
8	(X, Y | X +Y ≥ 3 ∧ X > 0)	(X-(3-Y), 3)	Pour water from 5-g jug into 3-g jug until 3-g jug is full



Search Tree Generation using BFS for Water Jug Problem

1. Breadth-First Search(BFS)

- Solution Path : (0,0) → (5,0) → (2,3) → (2,0) → (0,2) → (5,2) → (4,3)
- The path information can be obtained by modifying CLOSED list in the algorithm by putting pointer back to its parent.
- It is not memory efficient as partially developed tree is to be kept in the memory but it finds optimal solution or path

2. Depth-First Search (DFS)

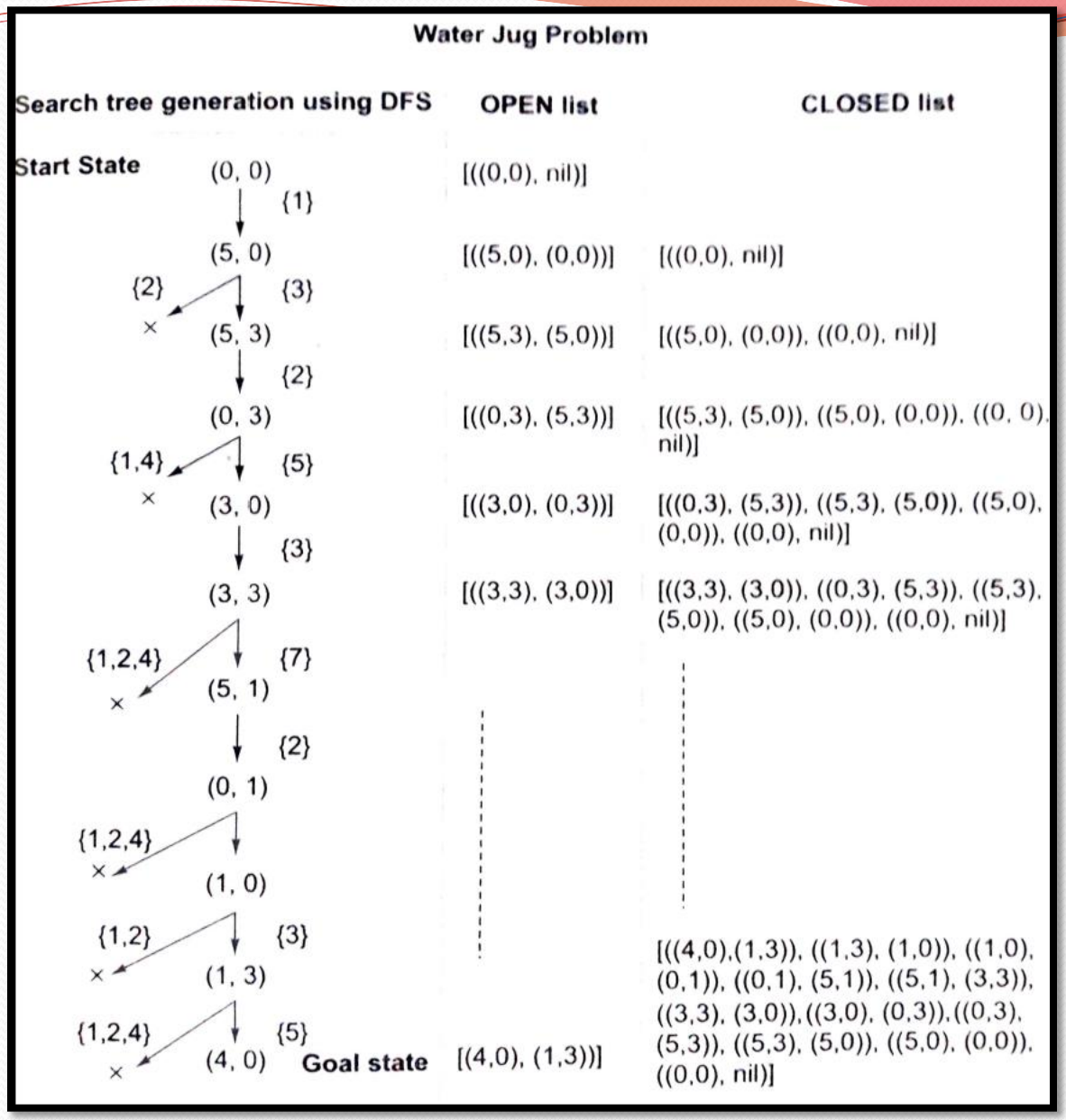
- In the Depth-First Search, we go as far down as possible into the search tree/graph before backing up and trying alternatives
- It works by always generating a descendant of the mostly recently expanded node until some depth cut off is reached and then backtracks to next most recently expanded node and generates one of its descendants
- DFS is memory efficient, as it only stores a single path from the root to leaf node along with the remaining unexpanded siblings for each node on the path
- DFS can be implemented by using two lists called OPEN and CLOSED
- OPEN list contains those states that are to be expanded and CLOSED list keeps track of states already expanded. Both are maintained as stacks.

Depth-First Search (DFS) Algorithm

Input: START and GOAL states
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*/
      }
  }
- If FOUND = true then return the path by tracing through the pointers to the parents on the CLOSED list else return **No**
- Stop

Search Tree Generation using DFS



2. Depth-First Search (DFS)

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

(0,0) → (5,0) → (5,3) → (0,3) →(3,0) → (3,3) → (5,1) → (0,1) → (1,0) → (1,3) → (4,0)

Comparison between BFS and DFS:

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

3. Depth-First Iterative Deepening Search(DFID)

- It takes advantages of both BFS and DFS searches on trees.
- It expands all nodes at a given depth before expanding any nodes at greater depth
- It is guaranteed to find the shortest path or optimal solution from start to goal state

Depth-First Iterative Deepening Search(DFID) Algorithm

Input: START and GOAL states
Local Variables: FOUND
Output: Yes or No
Method:

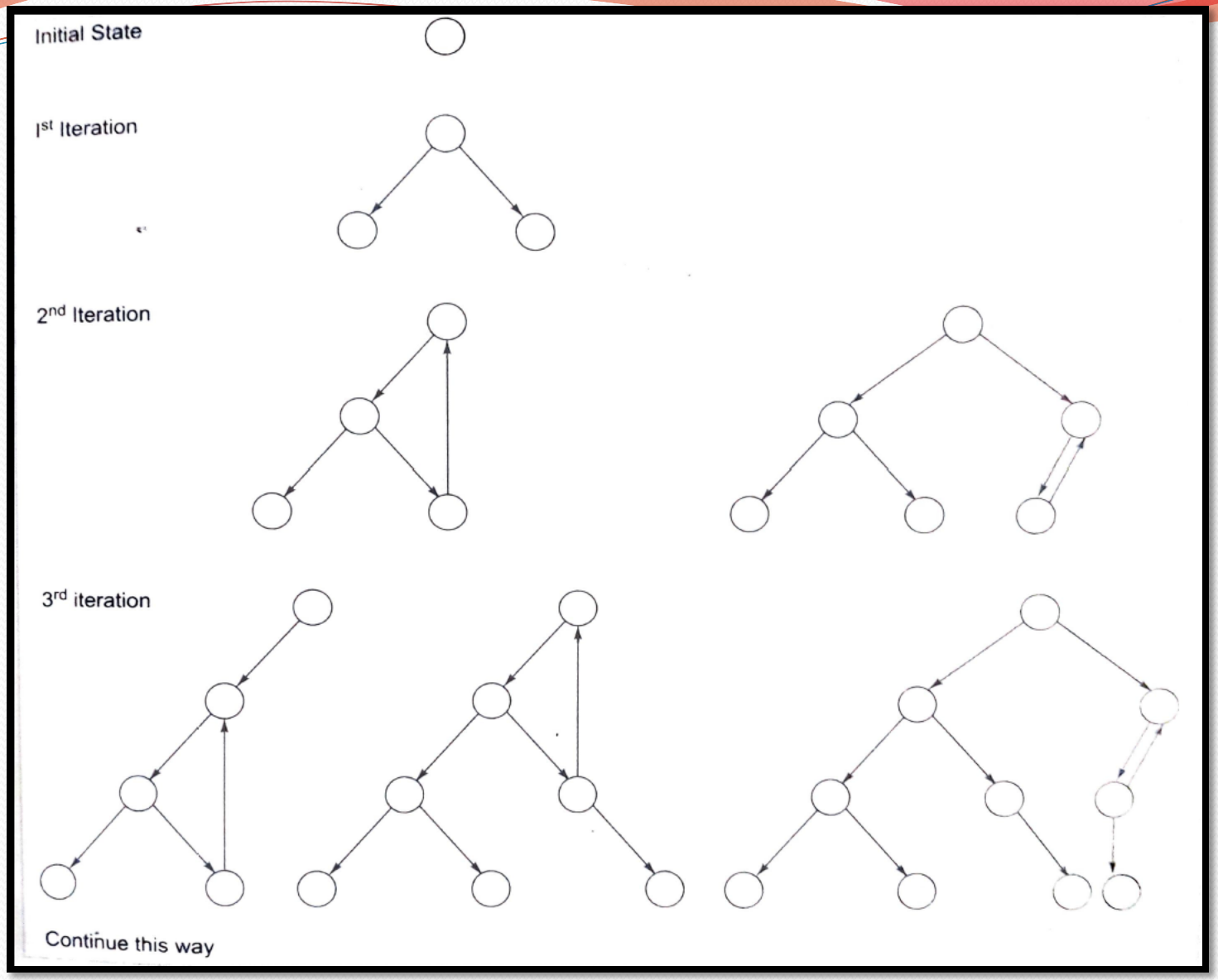
- Initialize d=1 , FOUND = false
- while(FOUND = false) do

  {

  - perform a depth first search from start to depth d
  - if goal state is obtained then FOUND = true else discard the nodes generated in the search of depth d
  - d = d+1

  }

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



Search Tree Generation using DFID

3. Depth-First Iterative Deepening Search(DFID)

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

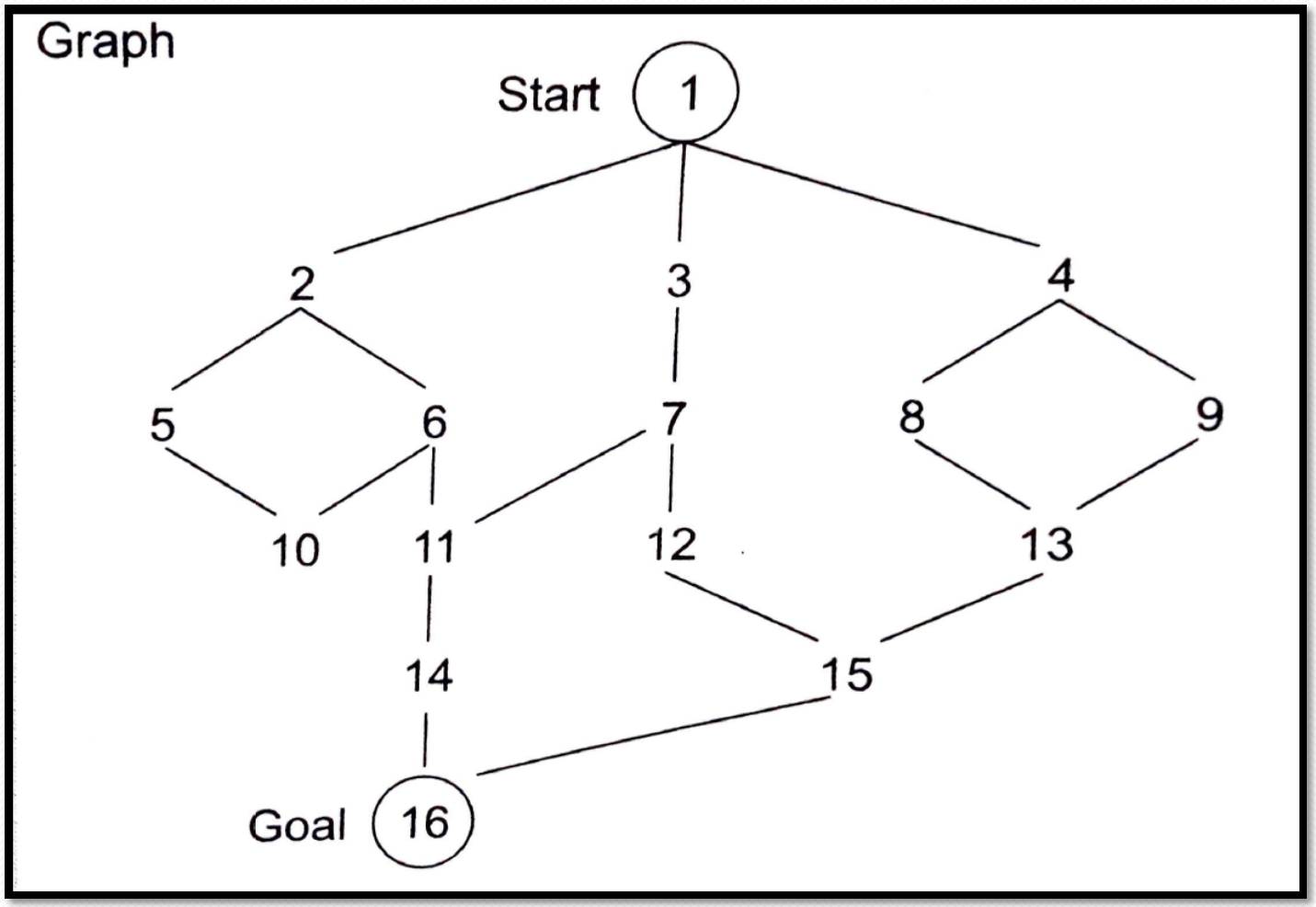
4. Bidirectional Search

- 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 k^{th} 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

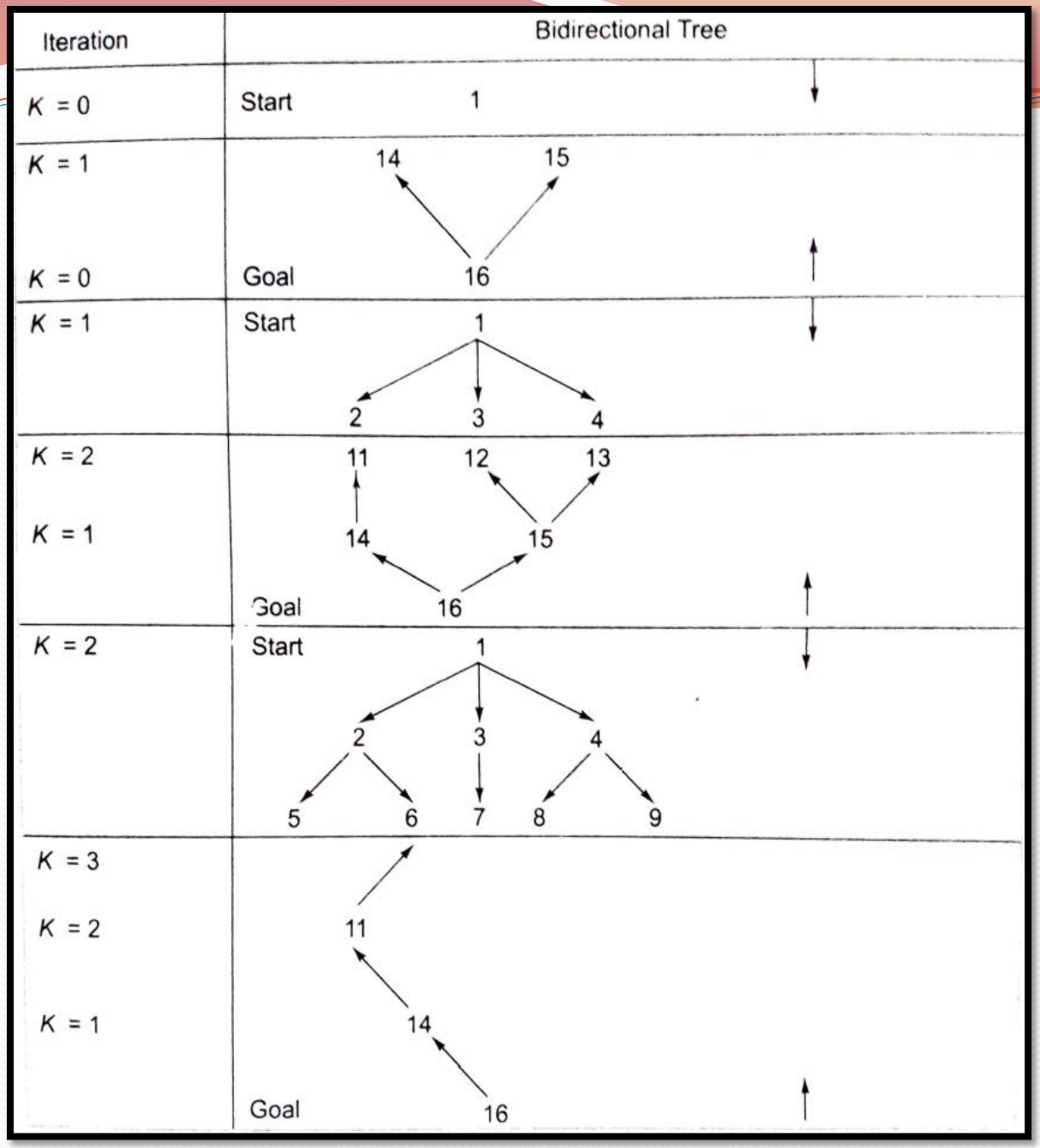
4. Bidirectional Search

- 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 state to goal state
- 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 search is that each of the two searches has time complexity $O(b^{d/2})$ and $O(b^{d/2} + b^{d/2})$ is much less than the running time of one search from the beginning to the goal, which would be $O(b^d)$
- This search can be made in already existing graph/tree or search graph/tree can be generated as a part of search

Find a route /path from node labeled 1 to node labeled 16



Graph to be searched using Bidirectional Search



Trace of Bidirectional Space

Analysis of Search Methods

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

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

Analysis of Search Methods

Let **b** be the branching factor and **d** be the depth of tree in worst case

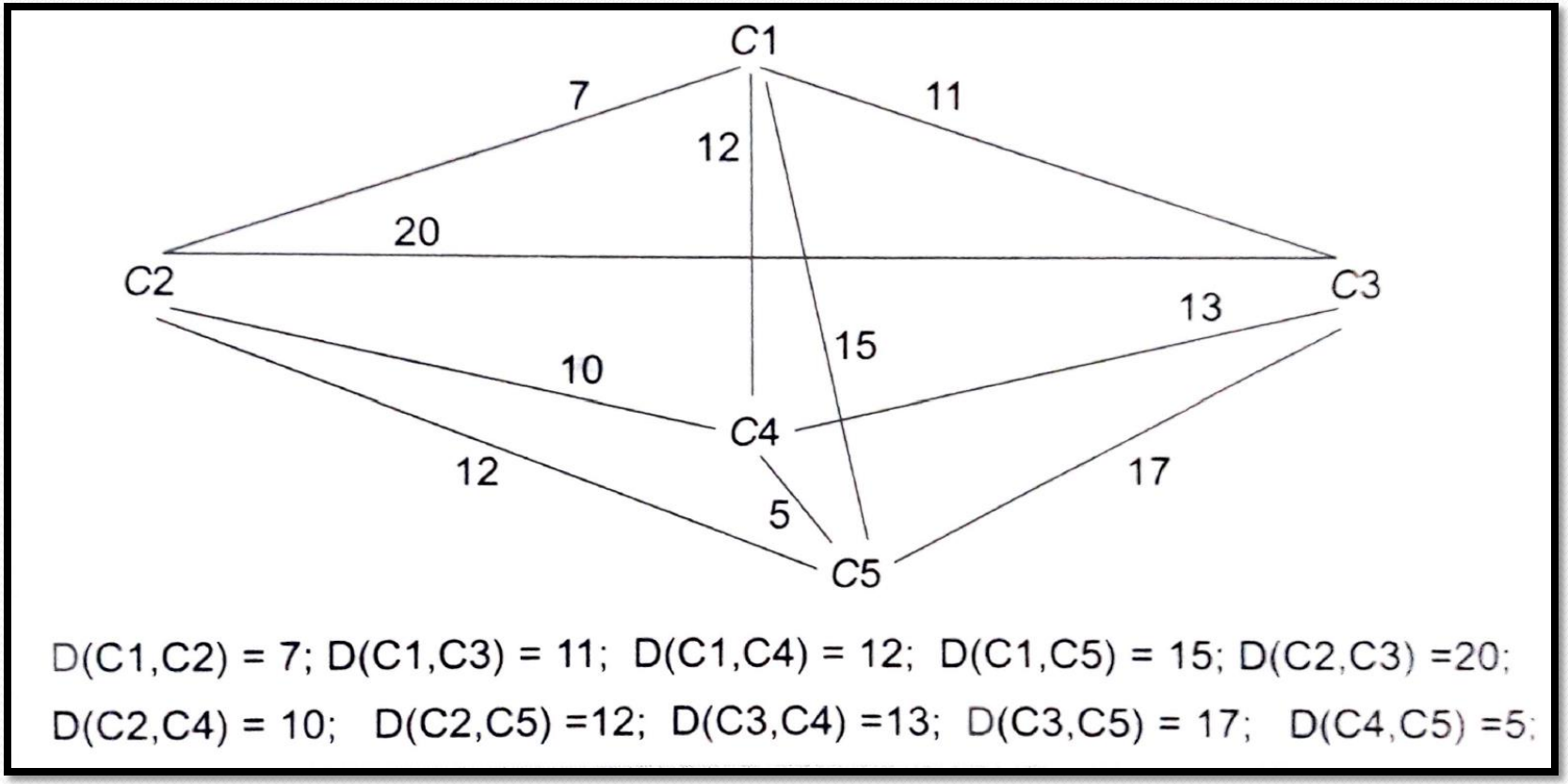
Search Technique	Time	Space	Solution
DFS	$O(b^d)$	O(d)	--
BFS	$O(b^d)$	$O(b^d)$	Optimal
DFID	$O(b^d)$	O(d)	Optimal
Bidirectional	$O(b^{d/2})$	$O(b^{d/2})$	--

Analysis of Search Methods

- Above mentioned searches are blind and are not of much use in real-life applications
- There are problems where combinatorial explosion takes place as the size of the search tree increases, such as Travelling Salesman Problem
- We need to have some intelligent searches which take into account some relevant problem information and finds solutions faster
- Let us consider a problem of travelling salesman
- In Travelling Salesman Problem(TSP), one is required to find the shortest route of visiting all the cities once and returning back to starting point

Analysis of Search Methods

- All possible paths of the search tree are explored and the shortest path is returned
- If there are n cities, then (n-1)! paths need to be examined . If number of cities grows, then the time required to wait a salesman to get the information about the shortest path is not a practical solution. This phenomenon is called a combinatorial explosion
- Above strategy could be improved using the following techniques:
  - Stop generating complete paths, keep track of the shortest path found so far
  - Stop exploring any path as soon as its partial length becomes greater than the shortest path length found so far
- This method is efficient than first one but still requires exponential time that is directly proportional to some number raised to 'n'



Graph for Travelling Salesman Problem

Performance Comparison

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

Continue like this

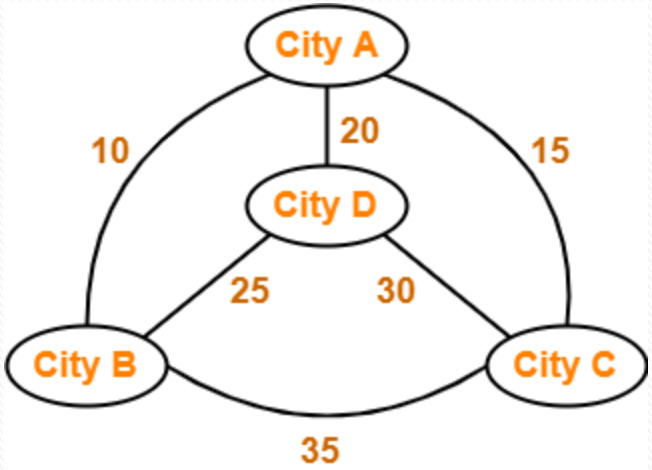
Heuristic Search Techniques

- Heuristic Technique is a criterion for determining which among several alternatives will be the most effective to achieve some goal
- It improves the efficiency of a search process by sacrificing claims of systematic and completeness
- It no longer guarantees to find the best solution but almost always find a very good solution
- We can hope to get good solution to hard problems like Travelling Salesman Problem in less than exponential time.
- There are two types of heuristics :
  - General-purpose heuristics that are useful in various problem domains
  - Special purpose heuristics that are domain specific
- Searches which use some domain knowledge are called Informed Search Strategies

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
- As it expands the least-cost partial path, it is sometimes also called a uniform cost search
- For eg. in travelling salesman problem, g(X) may be the actual distance travelled from start to current node X

Branch and Bound Search Algorithm



Travelling Salesman Problem

Branch and Bound Search Algorithm

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 ≠ Φ 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;

      }

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

Branch and Bound Search

- 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
- In branch and bound method, if g(X)=1 for all operators , then it degenerates to simple breadth first search
- It can be improved if we augment it by dynamic programming , that is, delete those paths which are redundant
- **Generate and test algorithm:**

  Start

  - Generate a possible solution
  - Test if it is a goal
  - If not go to Start else Quit

  End

Hill Climbing

- Hill Climbing is variant of Generate and Test Strategy
- It is an optimization technique that belongs to the family of local searches
- It 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
- Hill Climbing proceeds in a depth first order, but the choices are ordered according to some heuristic value (i.e. measure of remaining cost from current to goal state)

Simple Hill Climbing Algorithm

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

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

      }

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

Problems with Hill Climbing

- The search process may reach to a position that is not a solution but from there no move improves the situation. This will happen if we have reached a local maximum, a plateau or a ridge
- **Local maximum :** It is a state that is better than all its neighbours but not better than some other states which are far away. From this state all moves look to be worse. In such case 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 have 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 a new section of the search space

Problems with Hill Climbing

- **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.

Beam Search

- Beam Search is a heuristic search algorithm in which W (width) 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
- It uses BFS 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
- W is width of Beam Search and B is branching factor, there will be only W * B nodes under consideration at any depth but only W nodes will be selected

Beam Search

- If W=1, then it becomes hill climbing search where always best node is chosen from successor nodes
- If beam width is infinite, then it is identical to BFS
- 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 been pruned
- Assuming W=2 and B=3

Beam Search Algorithm

Input: START and GOAL states

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

Output: Yes or No

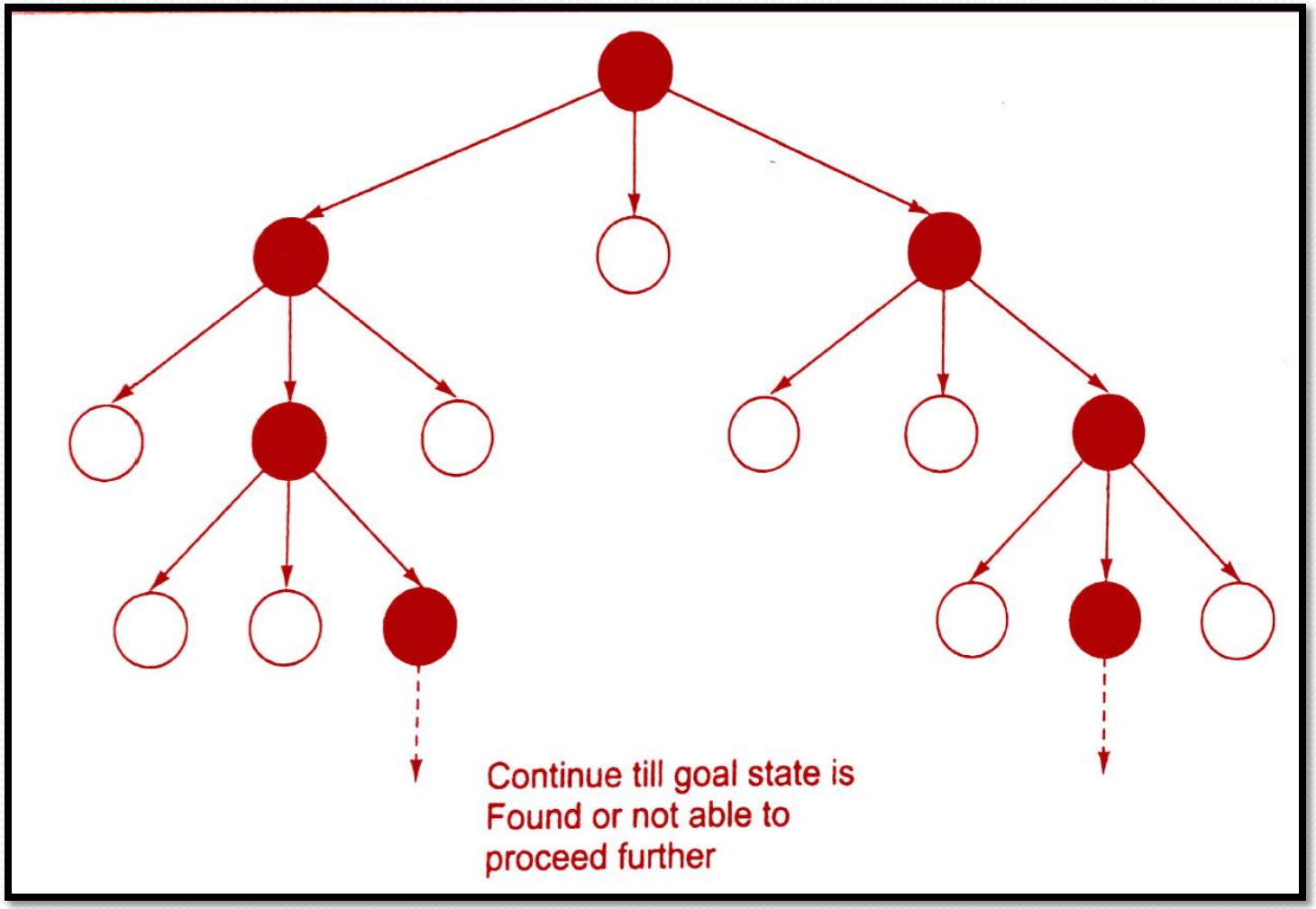
Method:

- 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

    }

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

Search tree generated using Beam Search Algorithm



Best First Search

- It 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
- In hill climbing, sorting is done on successor 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
- Performance varies directly with the accuracy of the heuristic evaluation function

Best First Search Algorithm

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 ≠ Φ 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

Best First Search

Condition for termination

- Instead of terminating when a path is found, terminate when the shortest incomplete path is longer than the shortest complete path

A* Algorithm

- It 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)

  where –

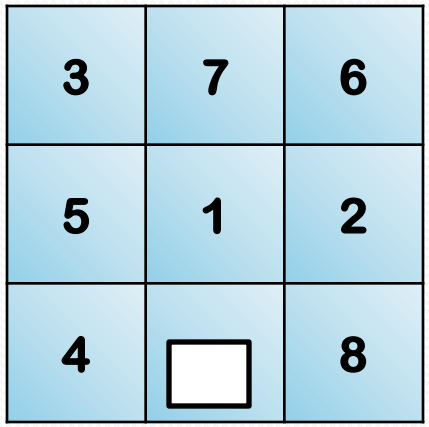
  g is a measure of the cost of getting from the start node to the current node N i.e. sum of costs of the rules that were applied along the best path to the current node

  h is an estimate of additional cost of getting from current node N to the goal node

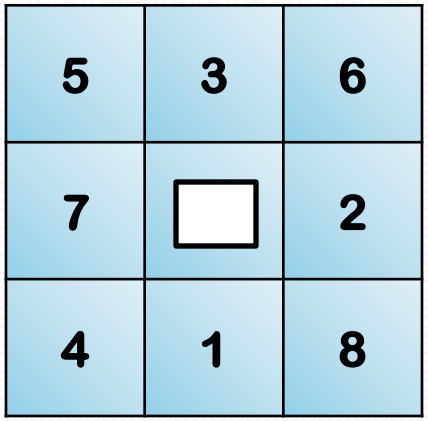
A* Algorithm

- It is called the OR graph / tree search algorithm
- It 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

The Eight Puzzle Problem



Start State

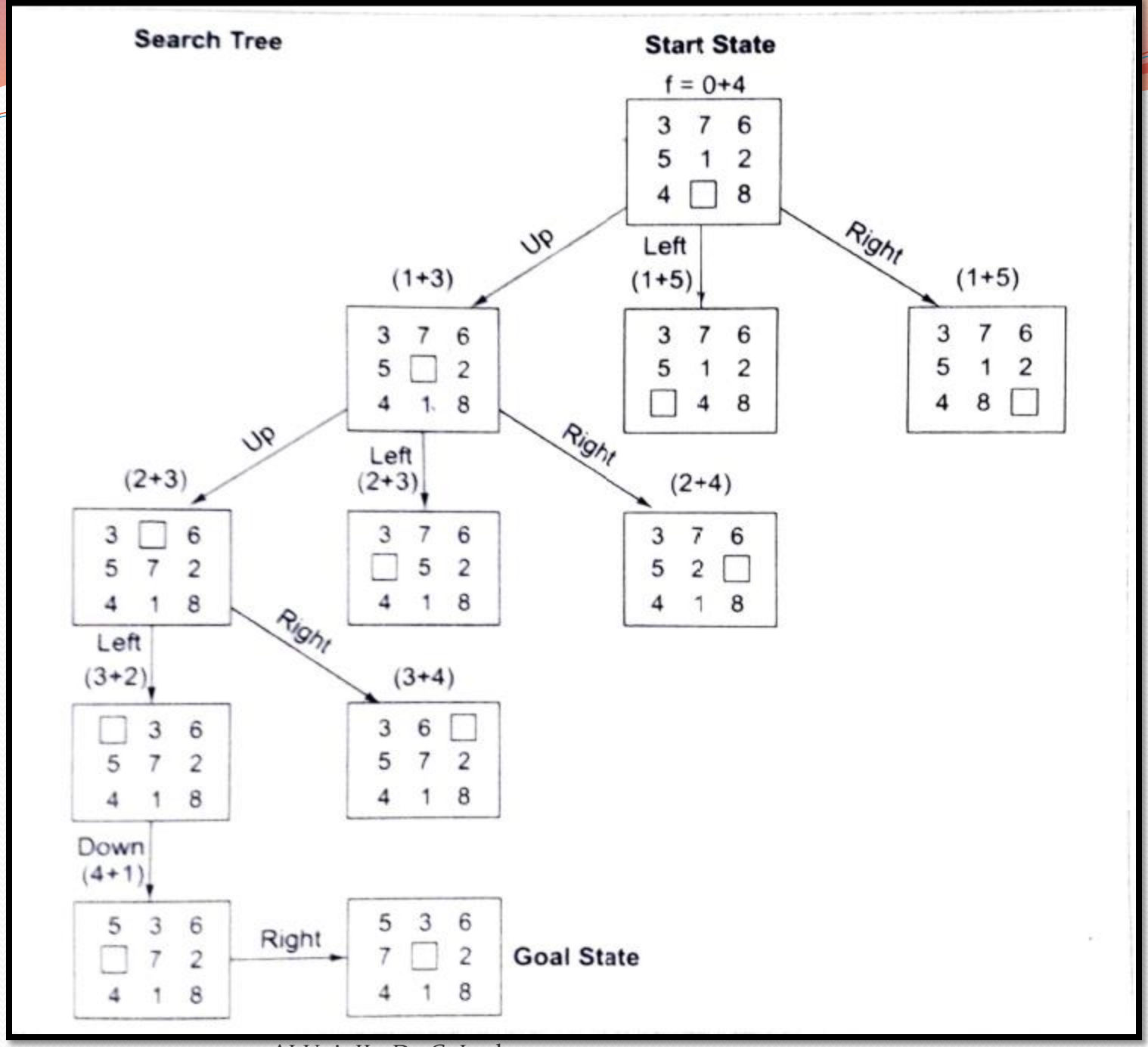


Goal State

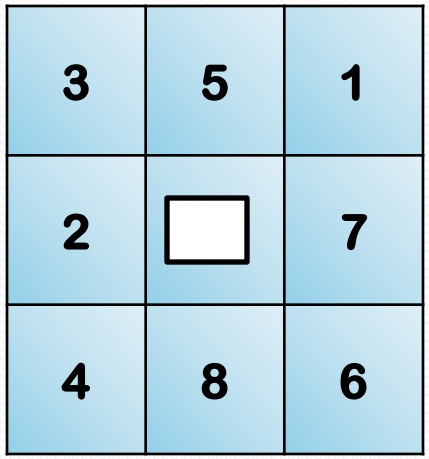
- Consider the eight puzzle problem
- f(X) = g(X) + h(X)

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

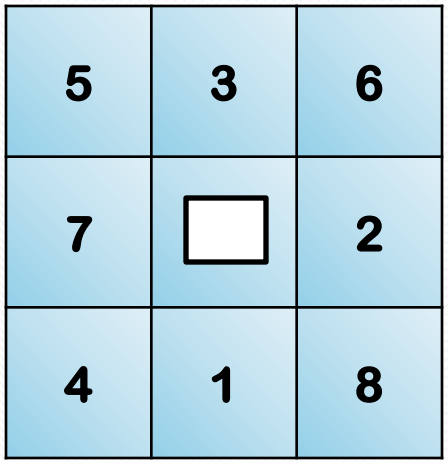
  g(X) = depth of node X in the search tree


Search Tree
Start State
f = 0+4
3 7 6
5 1 2
4 8
Up
Left
(1+5)
Right
(1+5)
(1+3)
3 7 6
5 2
4 1 8
3 7 6
5 1 2
4 8
3 7 6
5 1 2
4 8
Up
Left
(2+3)
Right
(2+4)
(2+3)
3 6
5 7 2
4 1 8
3 7 6
5 2
4 1 8
3 7 6
5 2
4 1 8
Left
(3+2)
Right
(3+4)
3 6
5 7 2
4 1 8
3 6
5 7 2
4 1 8
Down
(4+1)
5 3 6
7 2
4 1 8
Right
5 3 6
7 2
4 1 8
Goal State

- The following Eight Puzzle Problem cannot be solved using the heuristic function mentioned before



Start State



Goal State

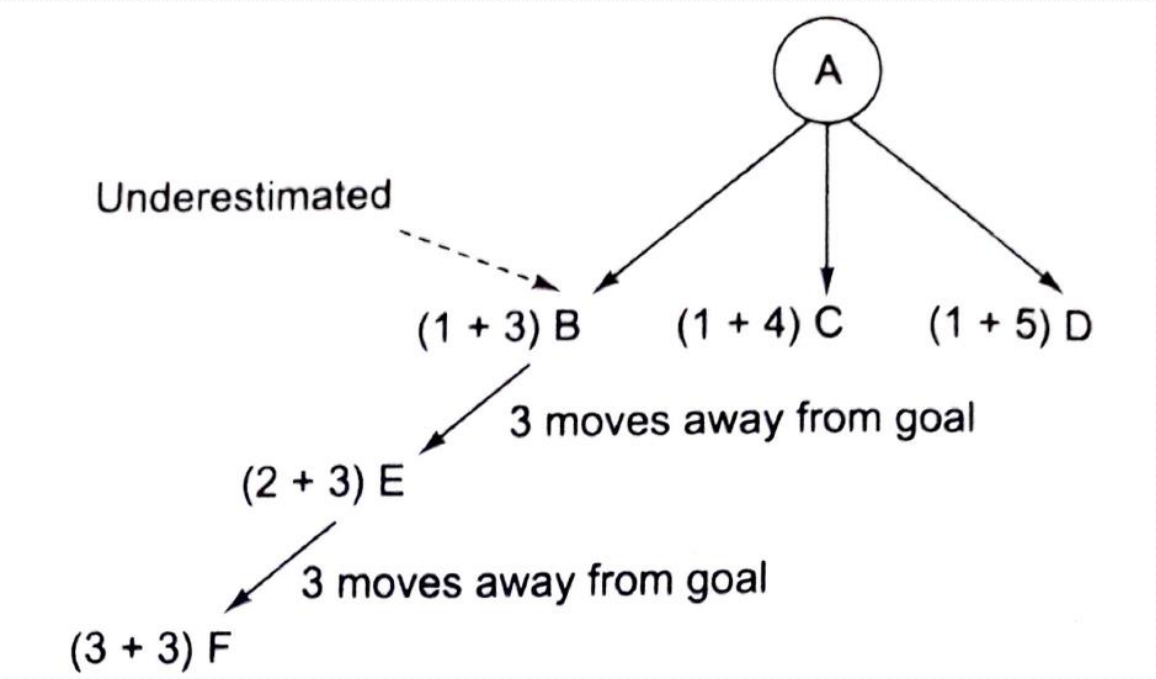
- A better estimate of h function might be as follows.
- The function g may remain same
- h(X) = the sum of the distances of the tiles (1 to 8) from their goal position in a given state X
- Here start state has h(start_state)= 3+2+1+0+1+2+2+1=12

A* Algorithm

1. Initialize: set OPEN=[s], CLOSED=[], g(s)=0, f(s)=h(s)
2. Fail: If OPEN=[], then terminate and fail
3. Select: Select a state with minimum cost ,n, from OPEN and save in CLOSED
4. Terminate: If n∈G then terminate with success and return f(s)
5. Expand: For each successor, m of n
For each successor, m, insert m in OPEN only if
if m∉ [OPEN∪CLOSED]
set g(m)= g[n]+C[n,m]
set f(m)=g(m)+h(m)
if m∈[OPEN∪CLOSED]
set g(m)= min{g[m], g(n)+C[n,m]}
set f(m)=g(m)+h(m)
if f[m] has decreased and m∈CLOSED move m to OPEN
6. Loop: Goto step 2

Optimal Solution by A* Algorithm

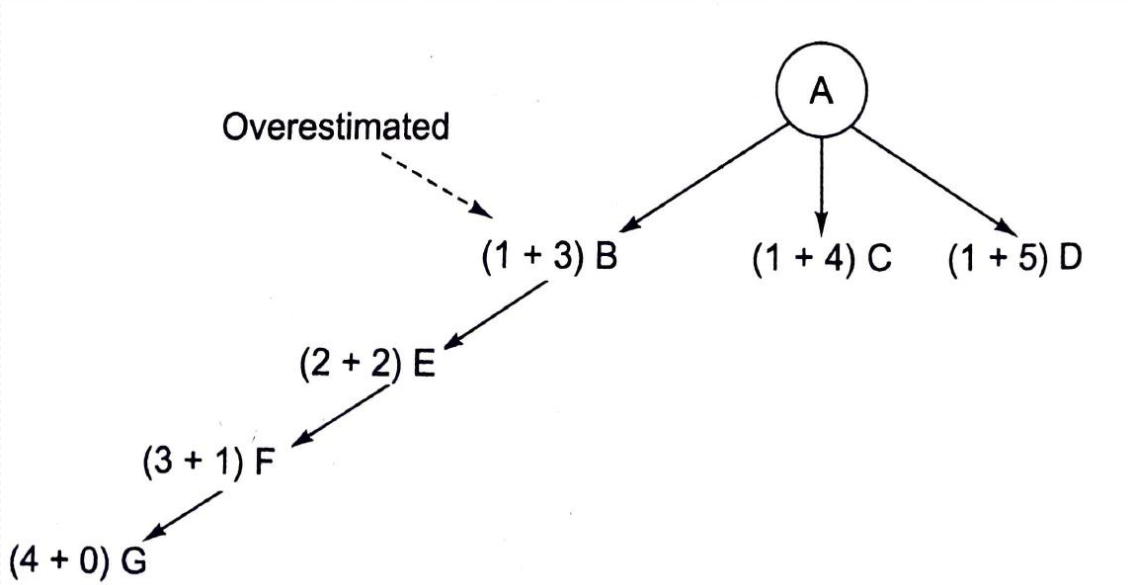
- **Underestimation –** Here we are underestimating heuristic value of each node in the graph/tree.



Example search graph for Underestimation

Optimal Solution by A* Algorithm

- **Overestimation –** Here we are overestimating heuristic value of each node in the graph/tree. By overestimating h we cannot guarantee to find shortest path



Example search graph for Overestimation

Optimal Solution by A* Algorithm

- **Admissibility of A***
  - A search algorithm is admissible if for any graph, it always terminates in an optimal path from start state to goal state, if path exists
  - If heuristic function 'h' underestimates the actual value from current state to goal state, then it bounds to give an optimal solution and hence is called admissible function.
  - A* always terminates with the optimal path in case h is an admissible heuristic function

Monotonic Function

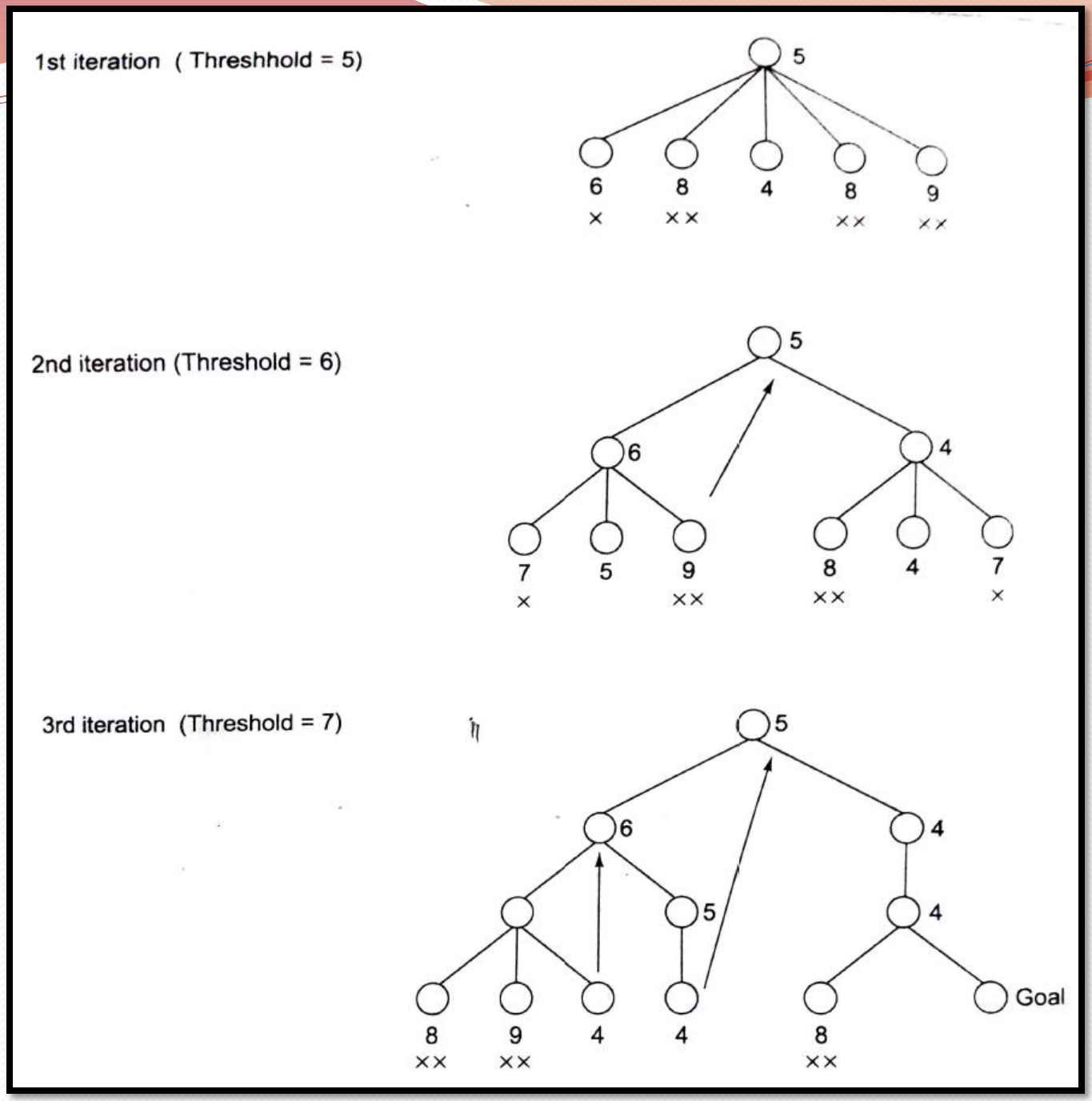
- A heuristic function h is monotone if
  1. ∀ states X_i and X_j such that X_j is successor of X_i

     $h(X_i) - h(X_j) \leq$ cost (X_i, X_j) i.e the actual cost of going from X_i to X_j
  2. h(Goal) =0
- In this case, heuristic is locally admissible i.e., consistently finds the minimal path to each state they encounter in the search
- With monotonic heuristic, if a state is rediscovered, it is not necessary to check whether the new path is shorter
- Each monotonic heuristic function is admissible
- A cost function f is monotone if f(N) ≤ f(succ(N))
- For any admissible cost function f, we can construct a monotonic admissible function

Iterative-Deepening A* (IDA*) Algorithm

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

Working of IDA*



Iterative-Deepening A* (IDA*) Algorithm

- 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 a cheapest path to a solution but uses far less space than A* and it expands approximately the same number of nodes as that of A* in a tree search
- It is simpler to implement than A* as there is no need of 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 which the goal is to solve some problem state that satisfies a given set of constraints instead of finding optimal path to the solution. Such problems are called Constraint Satisfaction (CS) Problems
- Search can be made easier in cases in which the solution is required to satisfy local consistency conditions
- For eg. Cryptography problem, n-Queen Problem, Map colouring, crossword puzzle, etc.
- We can define a Constraint Satisfaction Problem as follows:
  - a set of variables {x1,x2,….,xn}, with each xi ∈ Di with possible values and
  - a set of constraints i.e. relations , that are assumed to hold between the values of the variables

Constraint Satisfaction

- The problem is to find, for each i, $1 \leq i \leq n$, 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 the binary constraints
- A CS problem 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 values to any unassigned variable, provided that it does not conflict with previously assigned variables
- Every solution must be a complete assignment and therefore appears at depth n if there are n variables

Constraint Satisfaction 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 constraints describe 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 following puzzle by assigning numeral(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 (the constraints of arithmetic)

Crypt-Arithmetic Puzzle

- Initial problem state: G = ? ; A =? ; M =?; E =?; S =? ; B =?; L =?

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

Constraint equations are:

E+L =S → C1
S+ L+C1 =E → C2
2A+ C2 =M → C3
2B+ C3=A → C4
G = C4

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

Crypt-Arithmetic Puzzle : Working Steps

Constraint equations are:

G = C4

2B+ C3=A → C4

2A+ C2 =M → C3

S+ L+C1 =E → C2

E+L =S → C1

1. G= C4 ⇒ **G=1**
2. 2B+ C3=A → C4

   2.1 Since C4 =1, therefore 2B +C3 >9 ⇒ 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

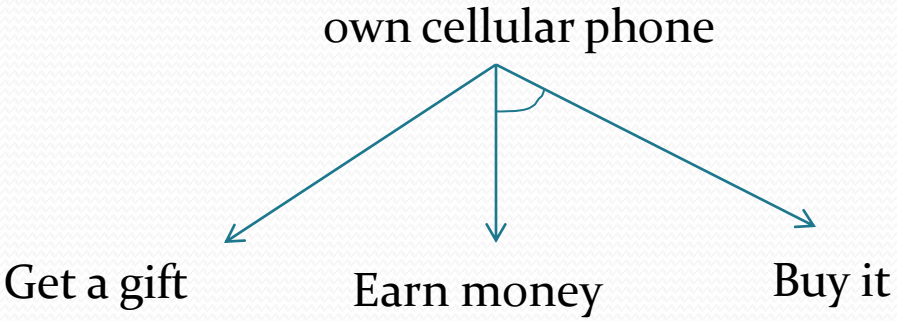
- If B = 5 → if C3 = 0 ⇒ A =0 ⇒ M=0 for C2=0 or M=1 for C2=1 X
  ↘ if C3 =1 ⇒ A=1 X (as G=1 already) ↗
- For B=6 we get similar contradiction while generating the search tree
- If **B = 7** , then for C3 =0 , we get **A=4** ⇒ M=8 if C2 = 0 that leads to contradiction later, so this path is pruned. If C2=1, then **M=9**

Crypt-Arithmetic Puzzle : Working Steps

3. Let us solve S+L + C1 = E and E+L =S
   - Using both equations , we get 2L +C1 =0 ⇒ **L=5** and C1=0
   - Using L=5, we get S+5 =E that should generate carry C2=1 as shown above
   - So S+5>9 ⇒ Possible values for E are {2,3,6,8} (with carry bit C2=1)
   - If E=2 the S+5 =12 ⇒ S=7 X (as B=7 already)
   - If E=3 the S+5 =13 ⇒ S=8
   - Therefore **E = 3** and **S = 8** are fixed up
4. Hence we get the final solution as given below and on backtracking , we may find more solutions. In this case, we got only one solution

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

Problem Reduction

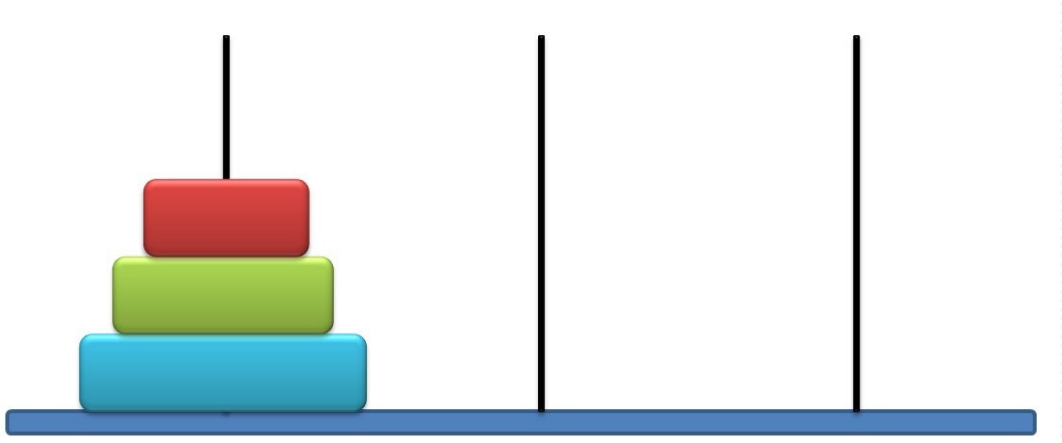


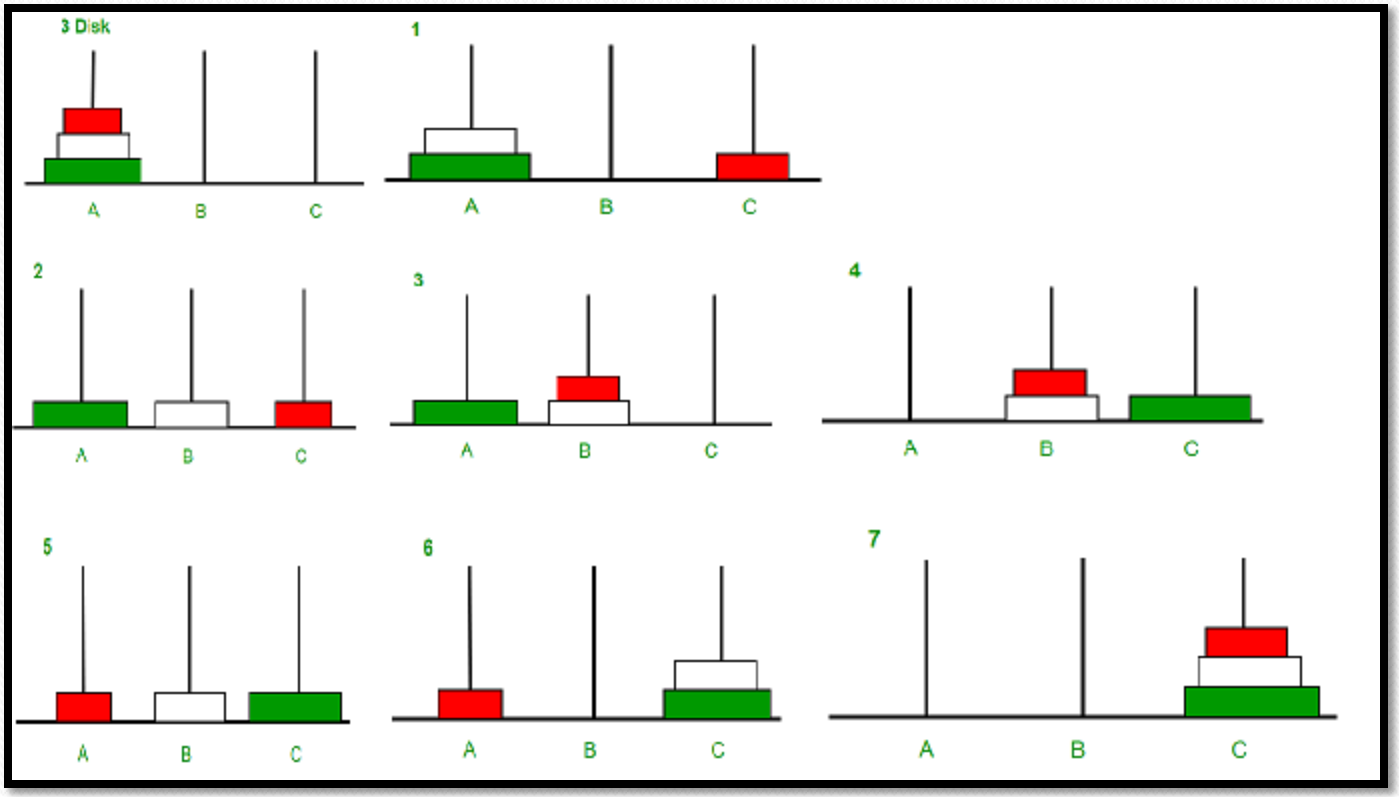
A Simple AND-OR Grpah

- In real world applications, complicated problems can be divided into simpler sub-problems, the solution of each sub-problem may then be combined to obtain the final solution

Problem Reduction : Towers of Hanoi Problem

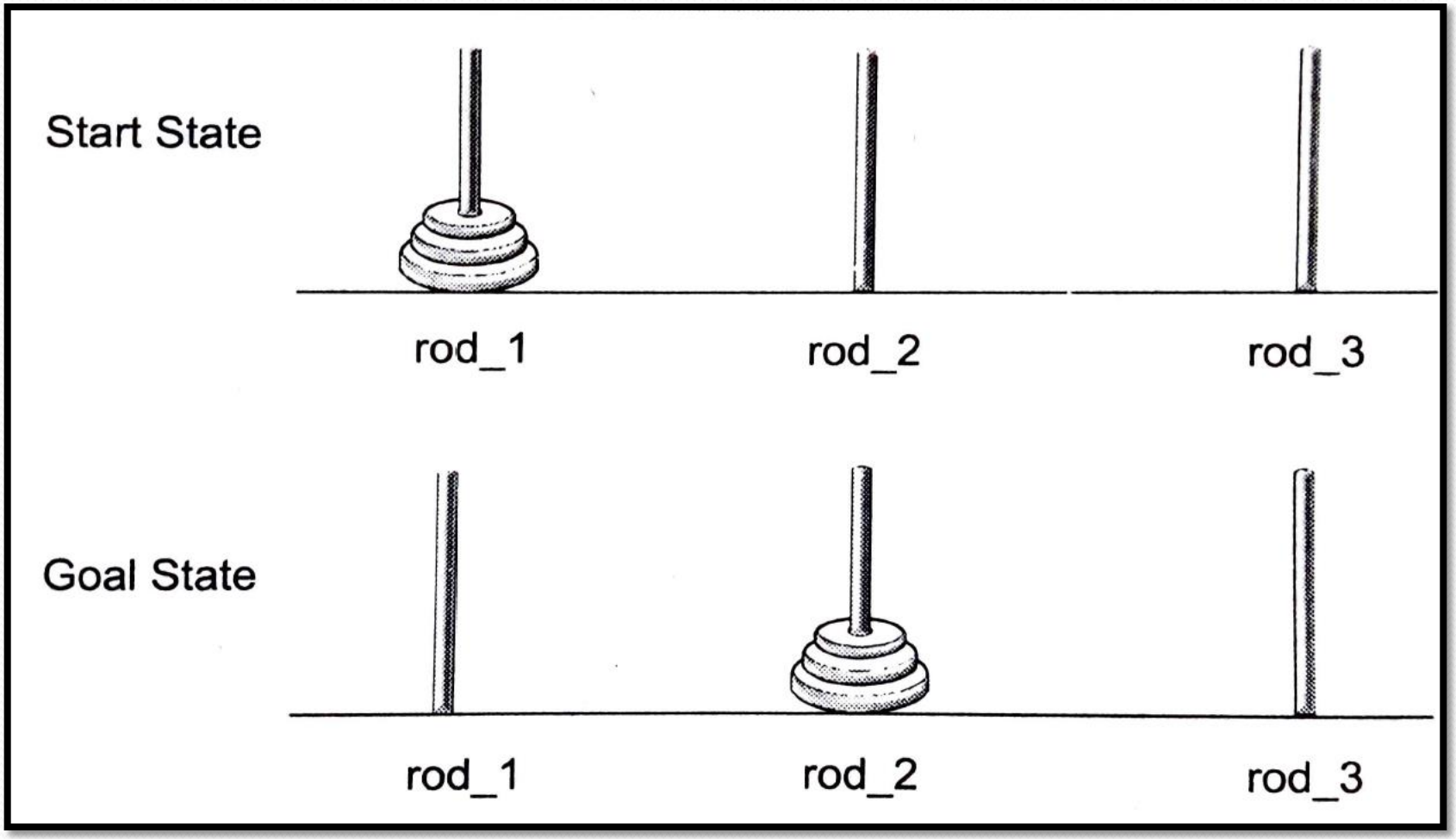
- The objective of the puzzle is to move the entire stack to another rod by using the following rules:
  - Only one disk may be moved at a time
  - Each move consists of taking the uppermost disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod
  - No disk may be placed on top of a smaller disk



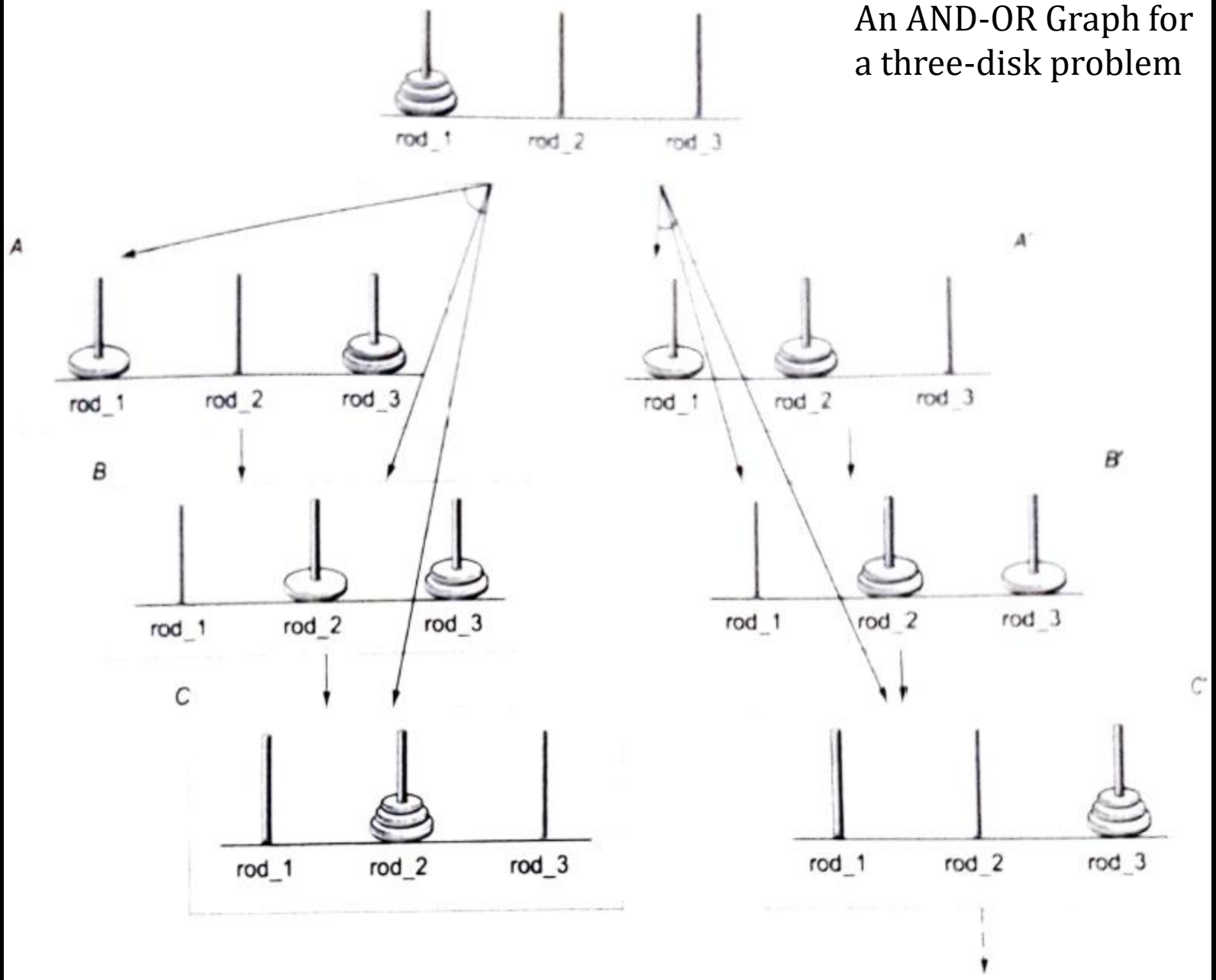


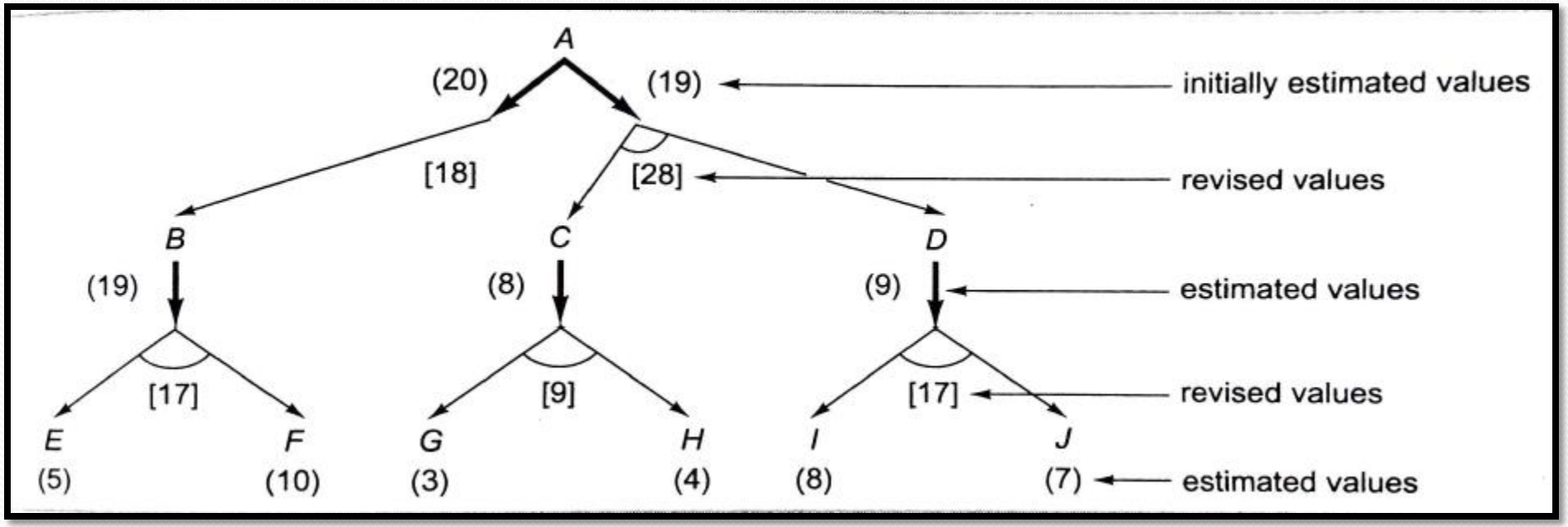
Sample solution for Towers of Hanoi

Towers of Hanoi Problem



An AND-OR Graph for a three-disk problem



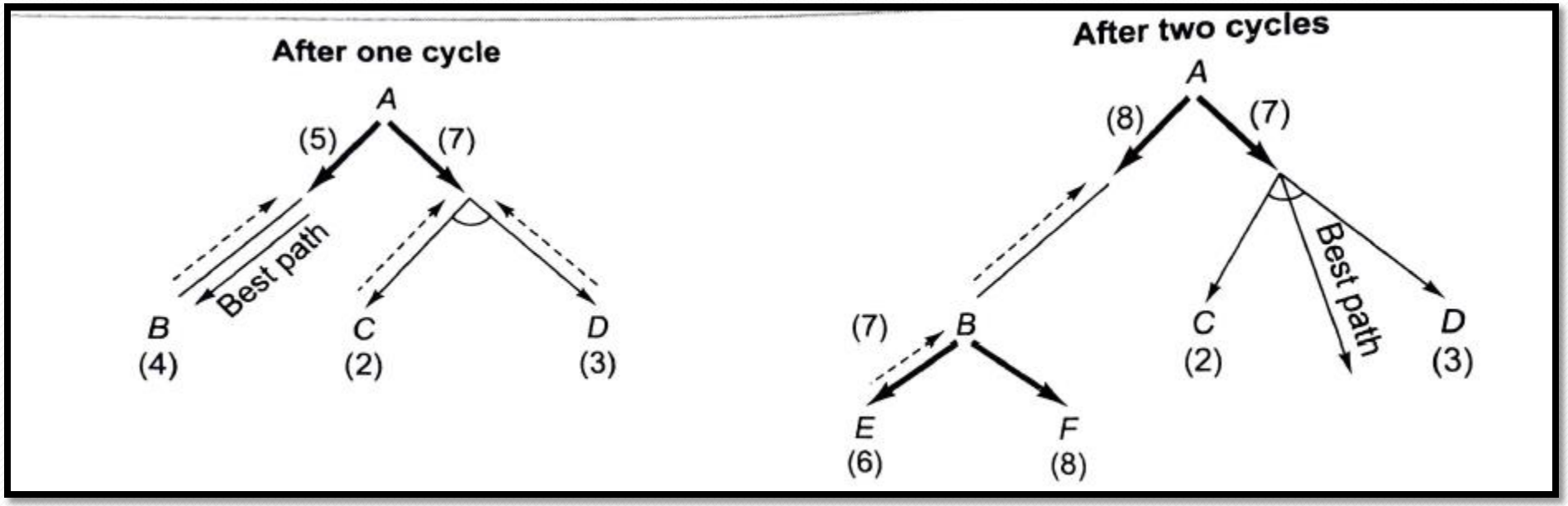


An Example of AND-OR Graph

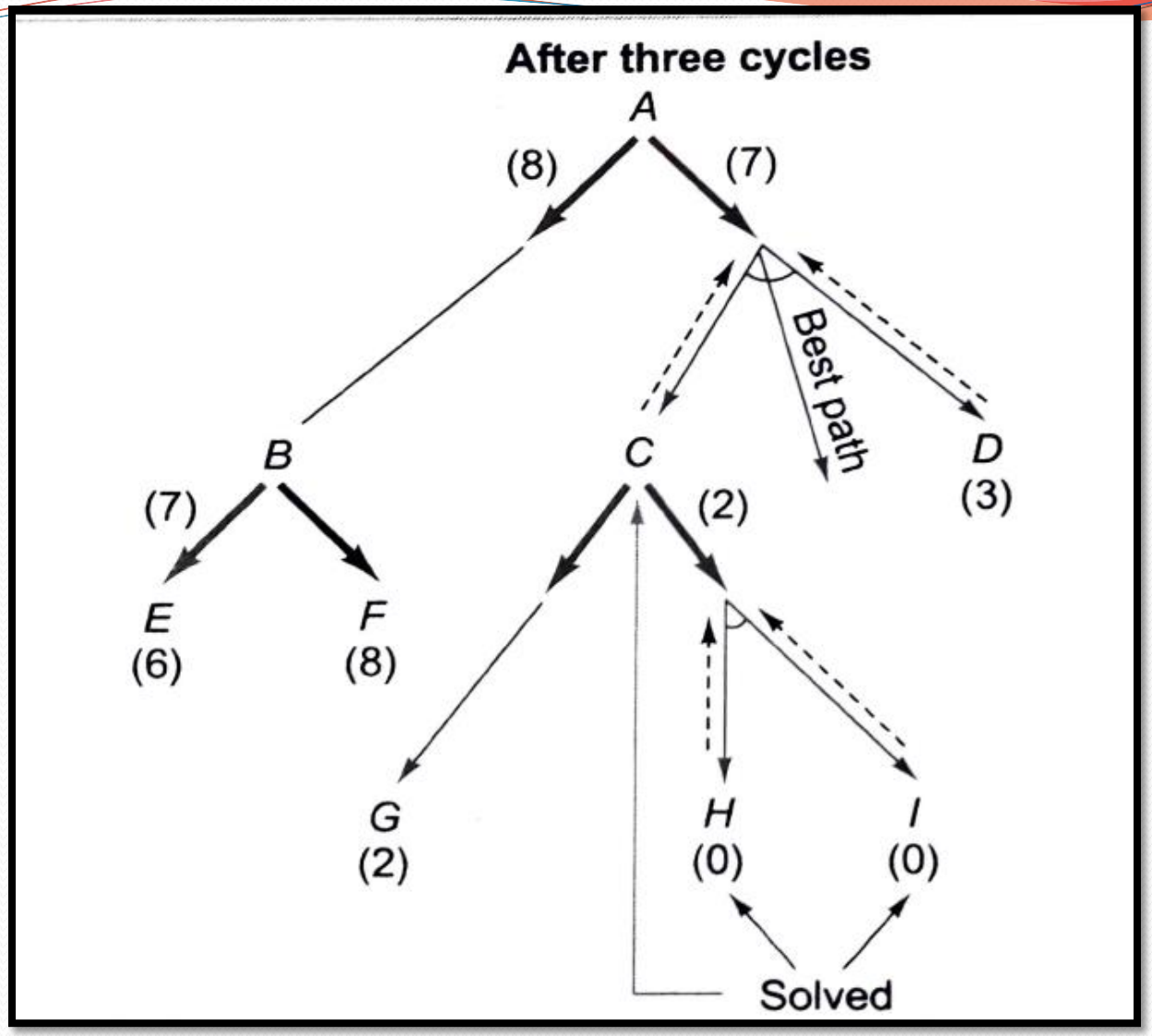
Node Status Labelling Procedure

- 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. 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

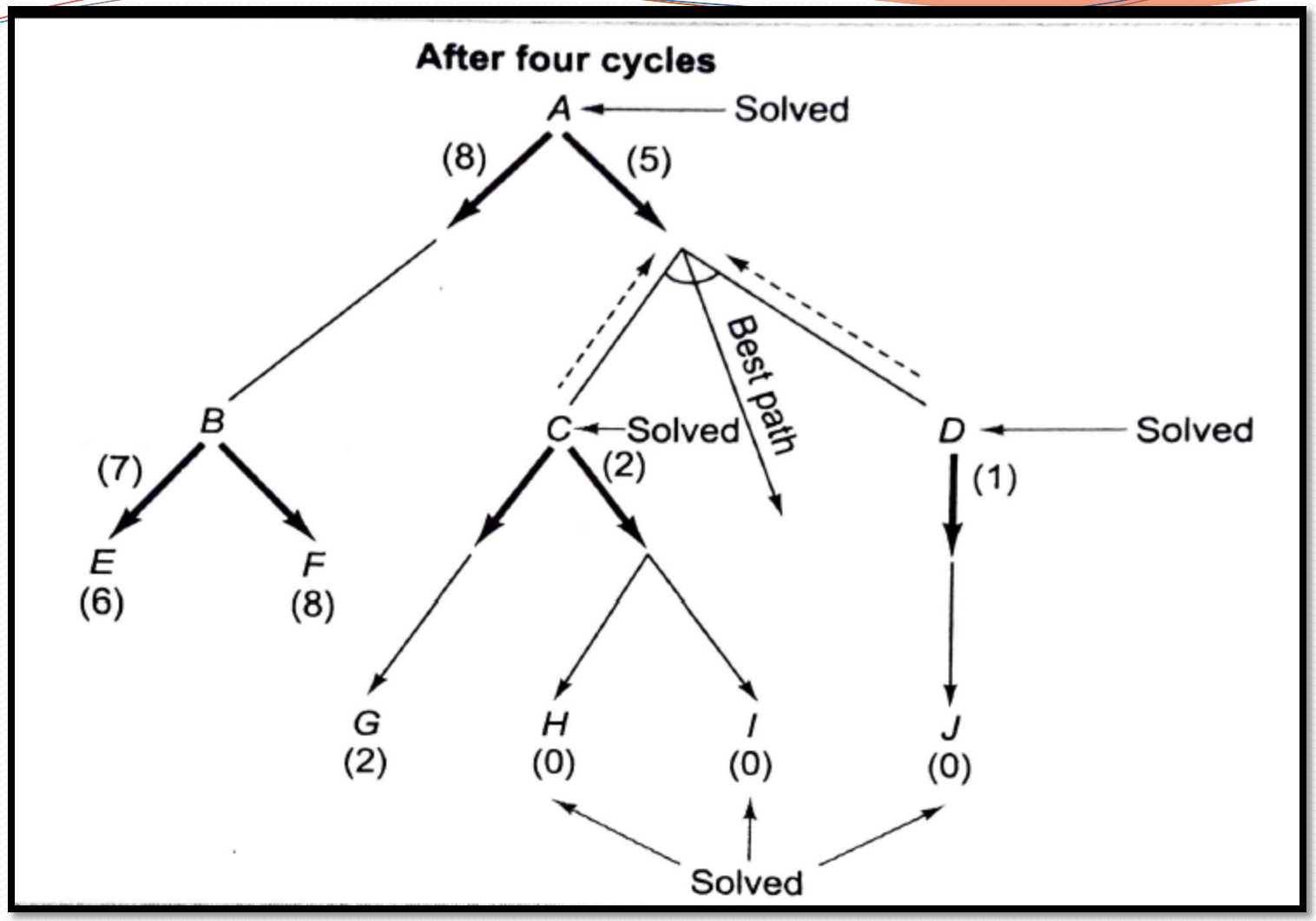
Node Status Labelling Procedure



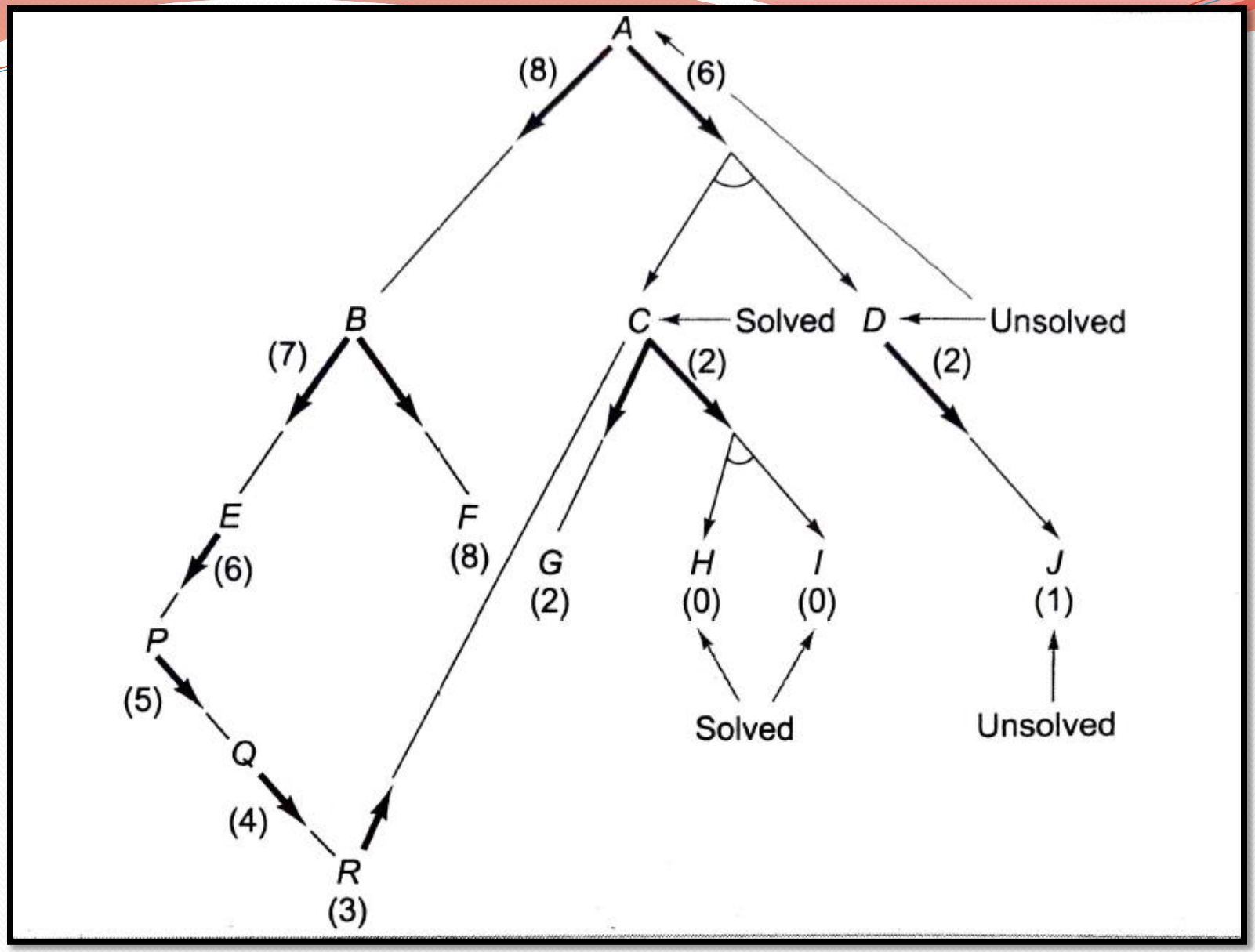
Labelling Procedure: First Two cycles



Labelling Procedure: Third cycle



Labelling Procedure: Fourth cycle



Non-optimal solution

Algorithmic Steps for AND-OR Graphs

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

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

Cyclic Graphs

- If the graph is cyclic then the algorithm discussed before will not work properly unless modified
- If the successor is generated and found to be already in the graph, then we must check that the node in the graph is not an ancestor of the node being expanded
- If not, then the newly discovered path to the node may be entered in the graph'
- This Algorithm is called AO* Algorithm as it used for searching a solution in an AND-OR graph
- Rather than using two lists, OPEN and CLOSED, as used in OR graph search algorithms , a single structure called graph G us used in AO* algorithm

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 the 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 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

    }

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

Algorithm: Propagation of Information Up the Graph

- Initialize L with NODE
- While (L ≠ Φ) do

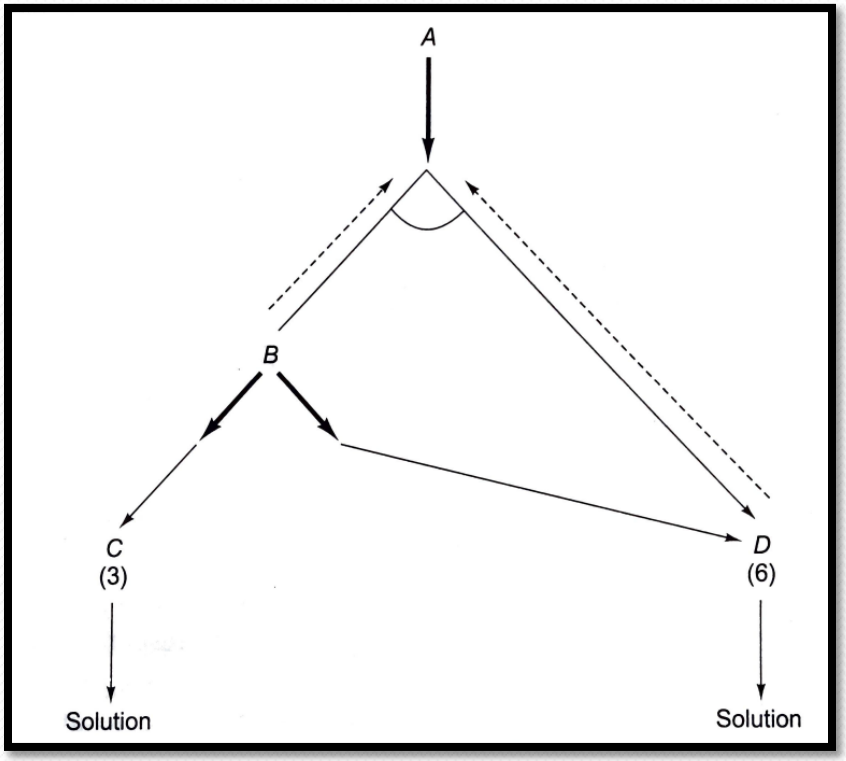
{

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

}

Interaction between sub-goals

- The AO* algorithm discussed above fails to take into account an interaction between sub-goals which may lead to non-optimal solution



Interaction between Sub-Goals

Game Playing

- 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
- Two player games are the most common games considered in AI
- Games can be classified into two types:
  - **Perfect Information Games :** Both the players have access to same information about the game in progress. For eg. Tic-Tac-Toe, Chess, Go, Checker etc.
  - **Imperfect Information Games :** Players do not have access to complete information about the game. For eg. Games involving the use of cards (such as Bridge) and dice
- A game is said to be discrete if it contains a finite number of states or configurations

Game Problem versus State Space Problem

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

- There is natural correspondence between game and state space problems
- A game begins from a specified initial state and ends in a position that can be declared a win for one , a loss for the other, or possibly a draw
- A game tree is an explicit representation of all possible plays of the game
- The root node is an initial position of the game. Its successors are the positions that the first player can reach in one move; their successors are the positions resulting from the second player's moves and so on.

Game Problem versus State Space Problem

- Terminal or leaf nodes are represented by WIN, LOSS or DRAW
- Each path from the root to a terminal node represents a different complete play of the game
- There is a correspondence between a game tree and an AND-OR tree
- The moves available to one player from a given position can be represented by the OR nodes, whereas the moves available to his opponent are the AND nodes
- In the game tree, one level is treated as OR node level and the other as AND node level from one player's point of view.
- In the AND-OR tree, both types of nodes may be on the same level.
- Game theory is based on the philosophy of minimizing the maximum possible loss and maximizing the minimum gain

Game Problem versus State Space Problem

- During a game, two types of node are encountered, namely MAX and MIN
- The MAX node will try to maximize its own game, while minimizing the opponent's (MIN) game
- Either of the two players, MAX and MIN, can play as the first player
- The computer will be assigned as MAX player and the opponent to be the MIN player
- Our aim is to make the computer win the game by always making the best possible move at its turn
- As a part of Game playing, game trees labelled as MAX level and MIN level are generated alternately

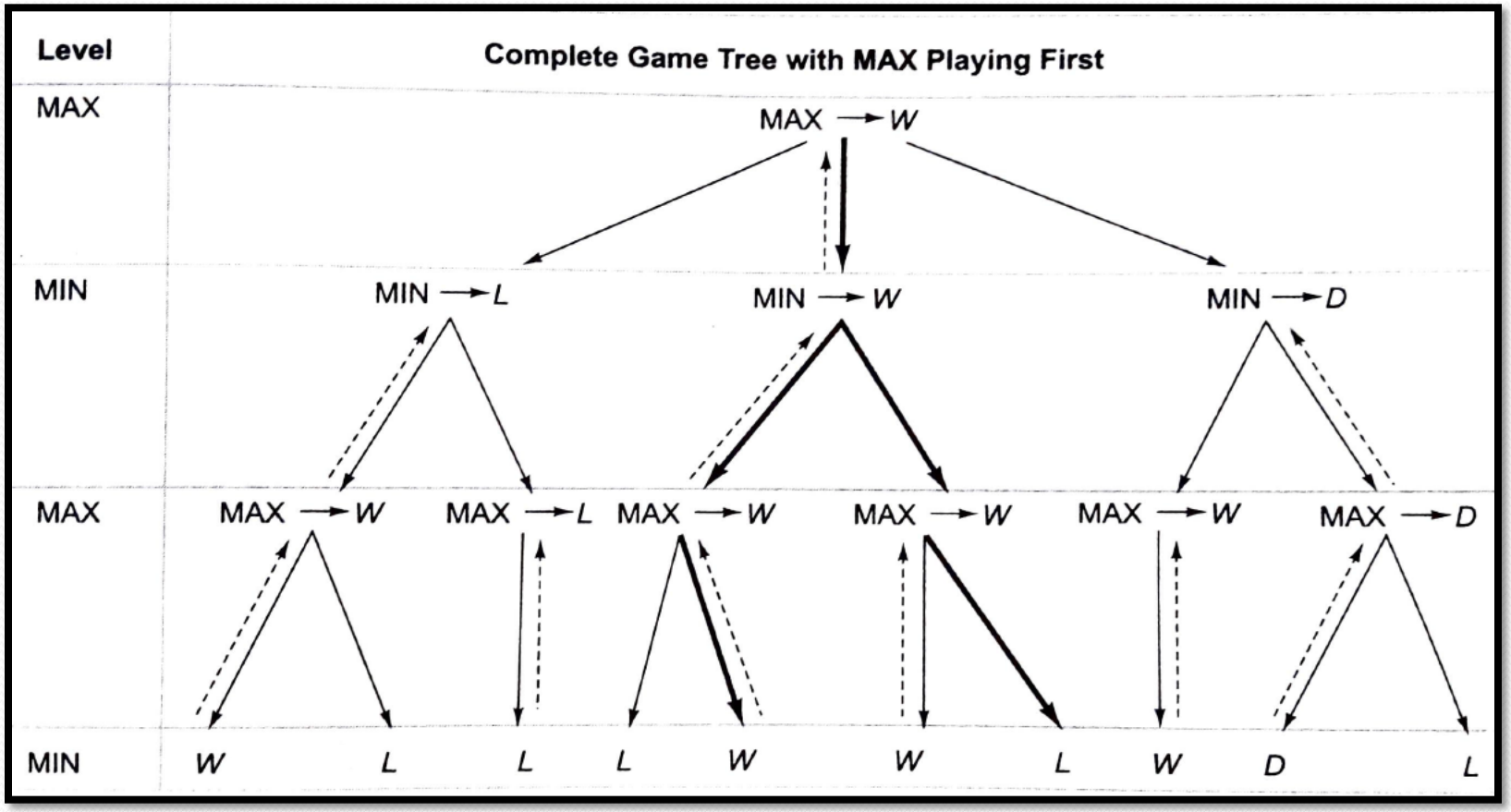
Status Labelling Procedure in Game Tree

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

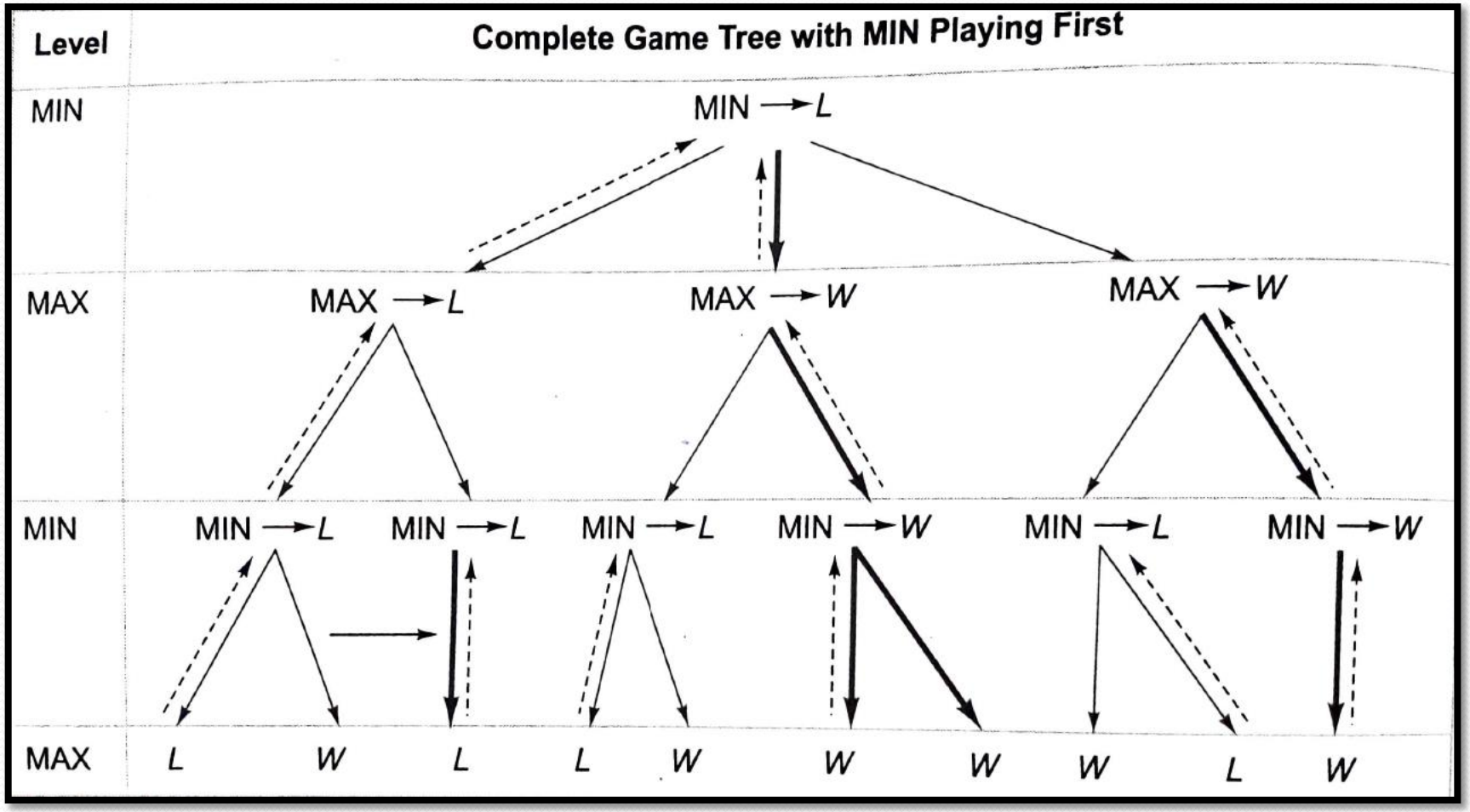
$$\text{STATUS}(j) = \begin{cases} \text{WIN}, & \text{if any of } j\text{'s successor is a WIN} \\ \text{LOSS}, & \text{if all of } j\text{'s successor are LOSS} \\ \text{DRAW}, & \text{if any of } j\text{'s successor is a DRAW and none is WIN} \end{cases}$$

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

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



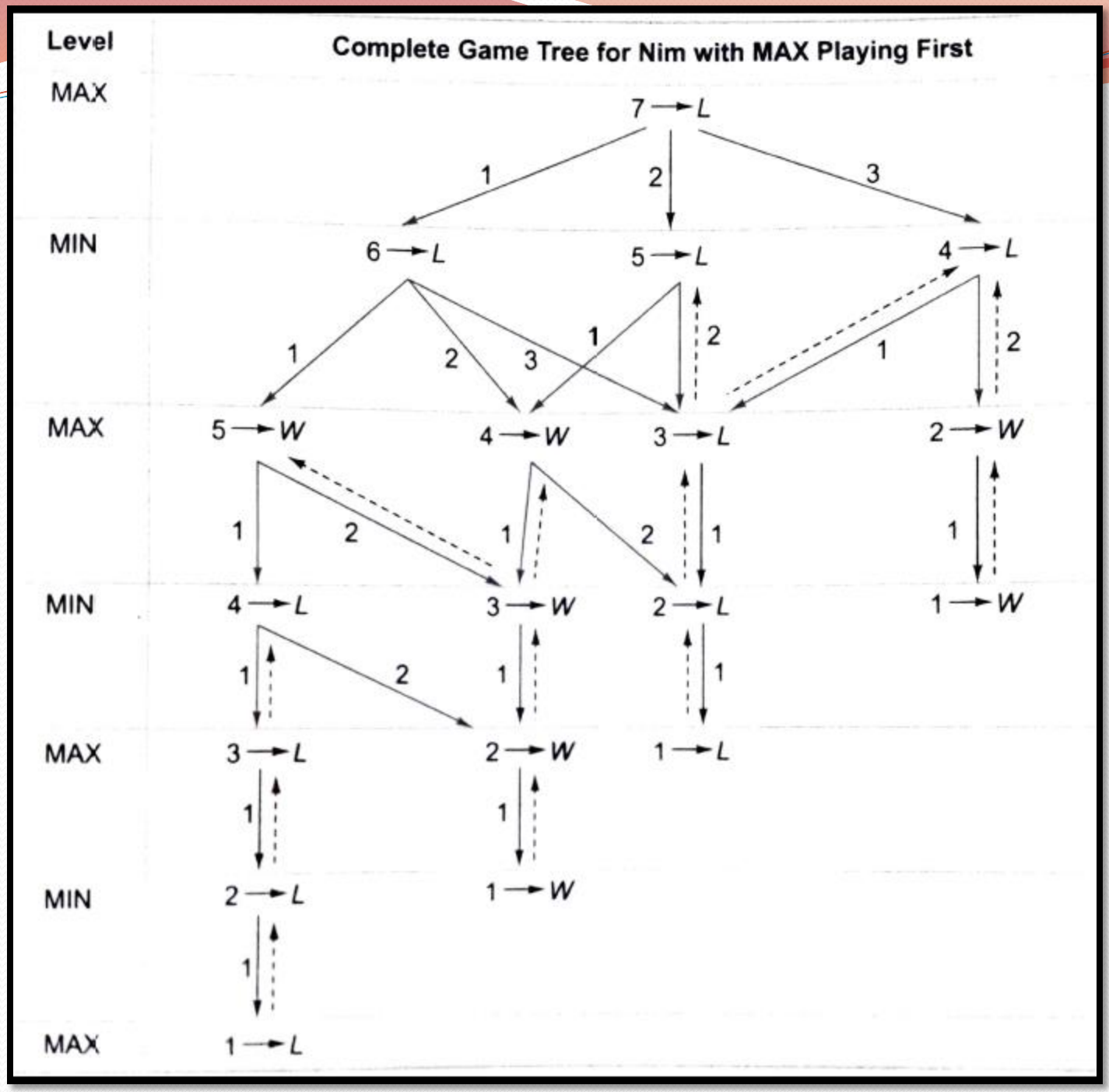
A Game tree in which MAX plays first

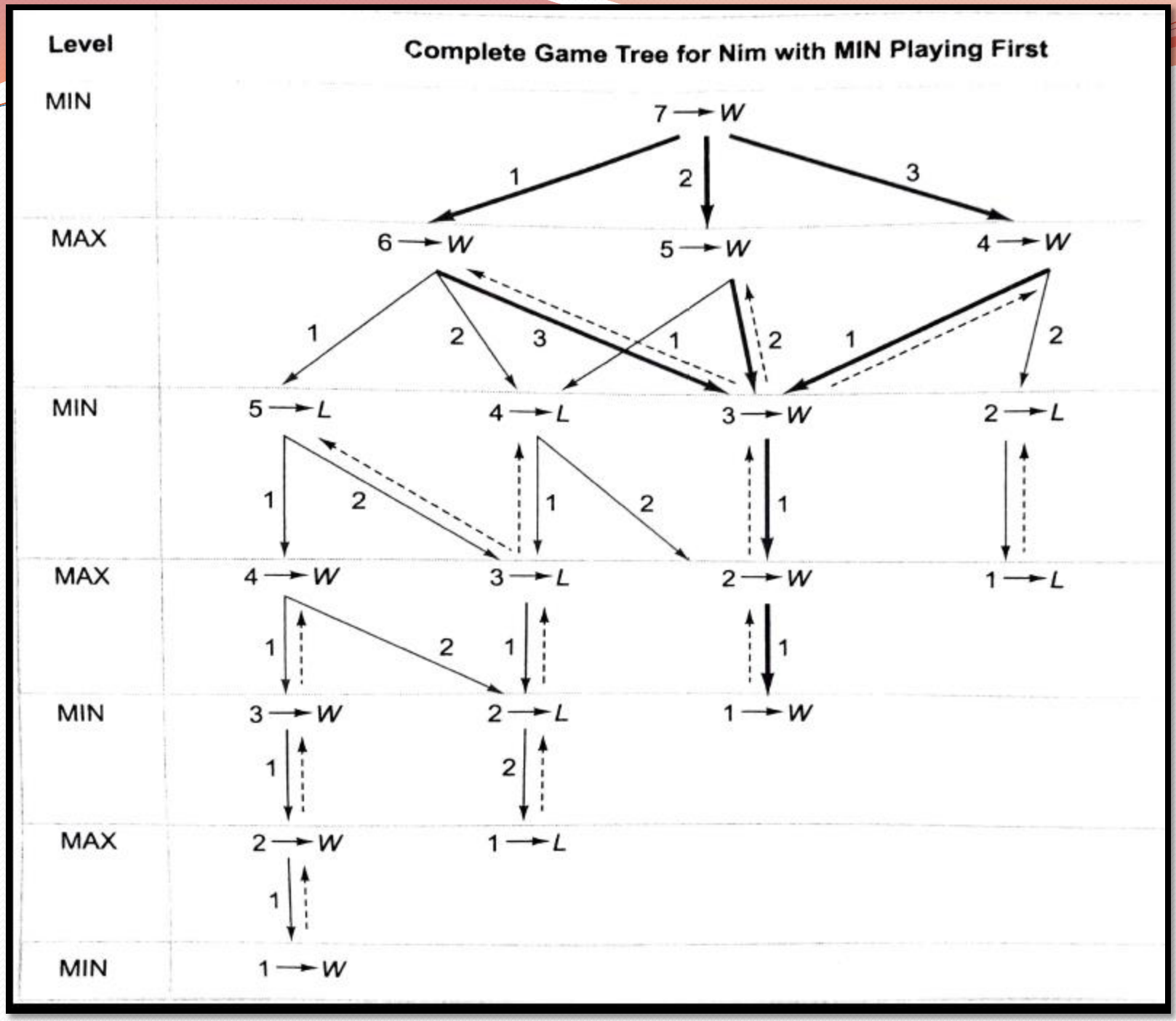


A Game tree in which MIN plays first

Nim Game problem

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


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


Level
Complete Game Tree for Nim with MIN Playing First
MIN
MAX
MIN
MAX
MIN
MAX
MIN
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
1 → W

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 ∈ { 1, 3, 7 ,15, 31, 63, . . . } using the rule of the 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, . . . } can be generated using the formula

  $X_i = 2\ X_{i-1} + 1$, where $X_0 = 1$ for $i > 0$
- A method is to be formulated which will determine the number of matchsticks that have to be picked up by MAX player.
- There are two ways of finding this number

Method

1. 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.

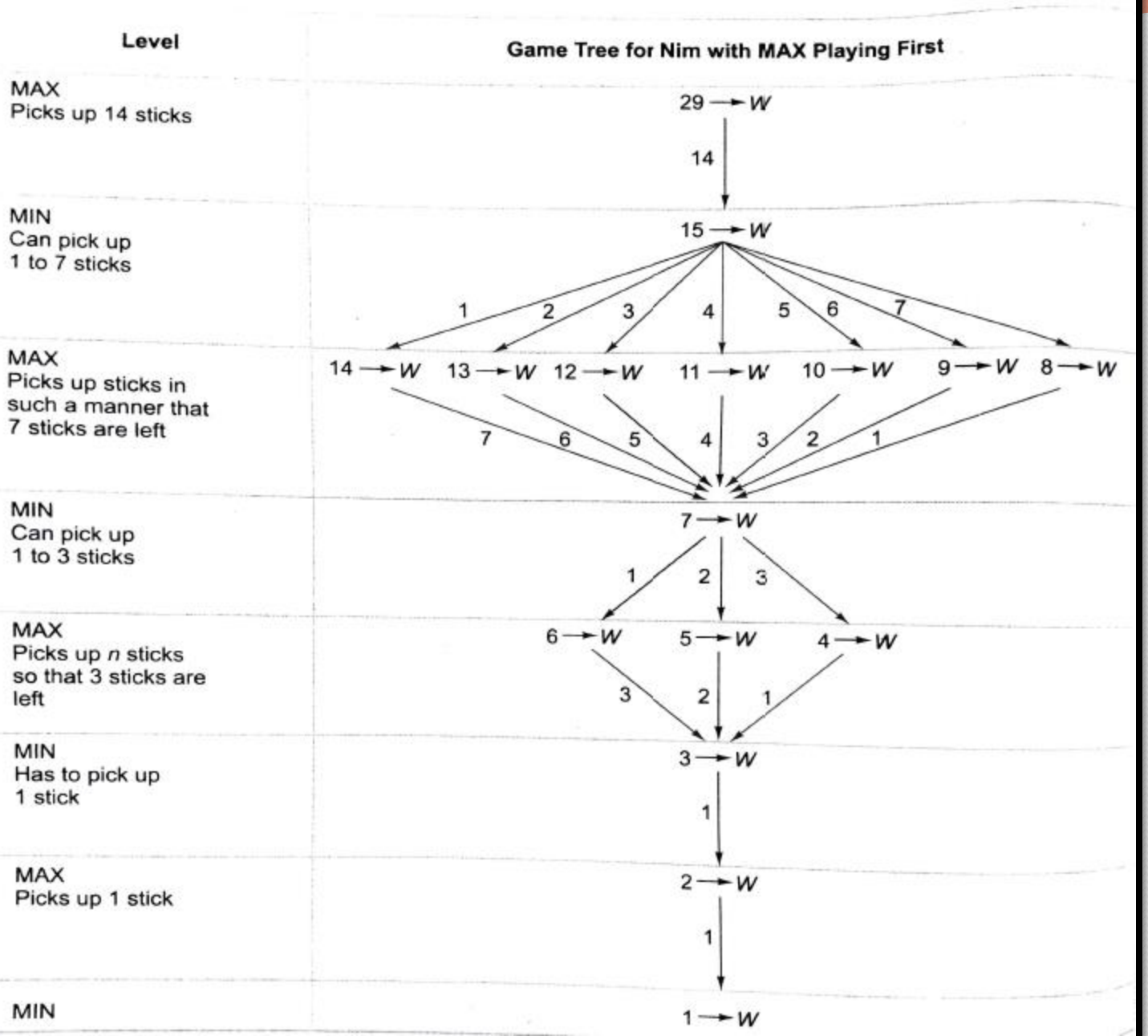
   For eg. If N = 45, the closest number to 45 in the sequence is 31, so we obtain the desired number of matchsticks to be picked up as 14 on subtracting 31 from 45

2. 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

N	Binary Representation of N	Sum of 1 with MSD removed from N	No. 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

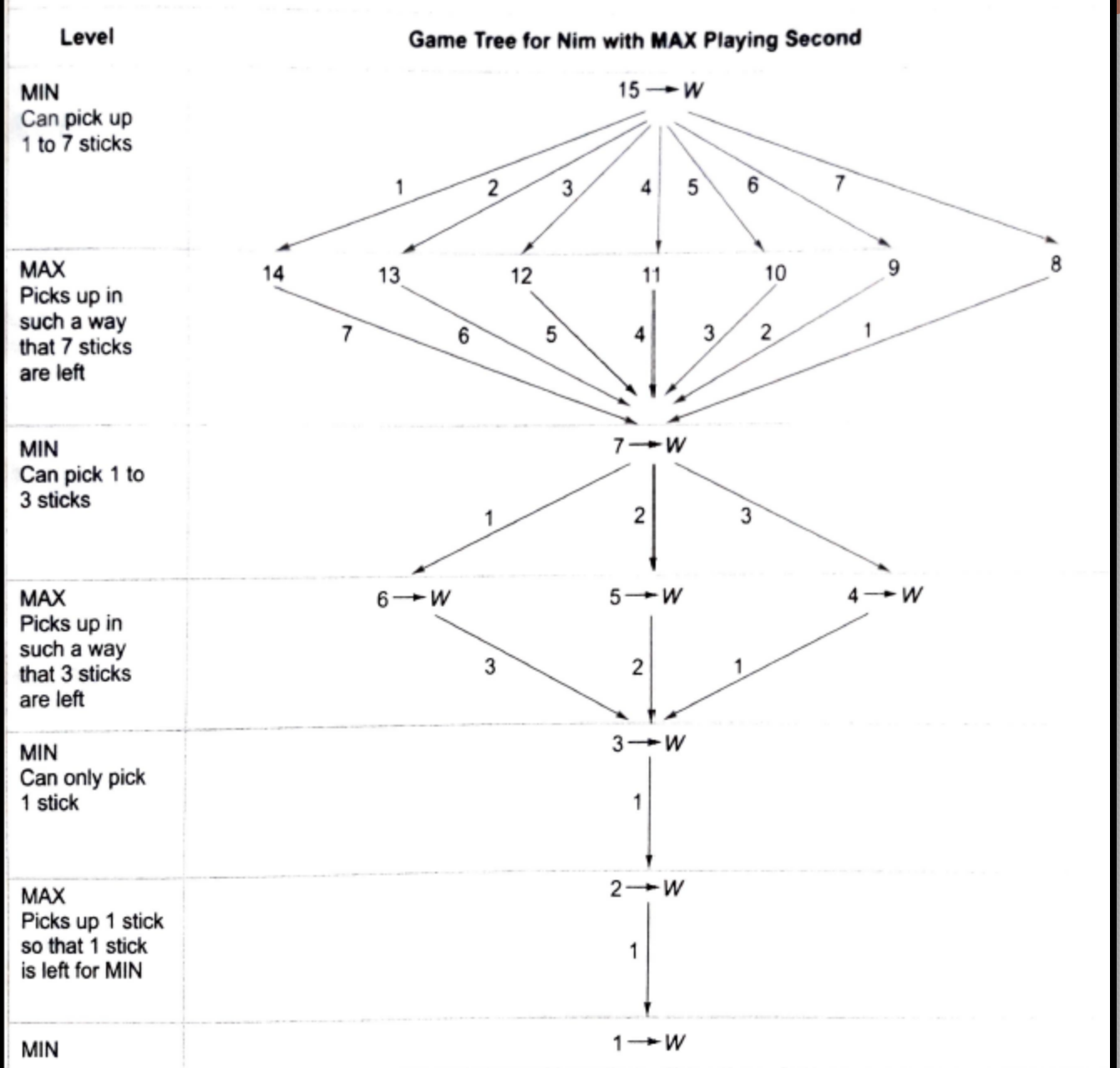
Case 1:
N=29

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



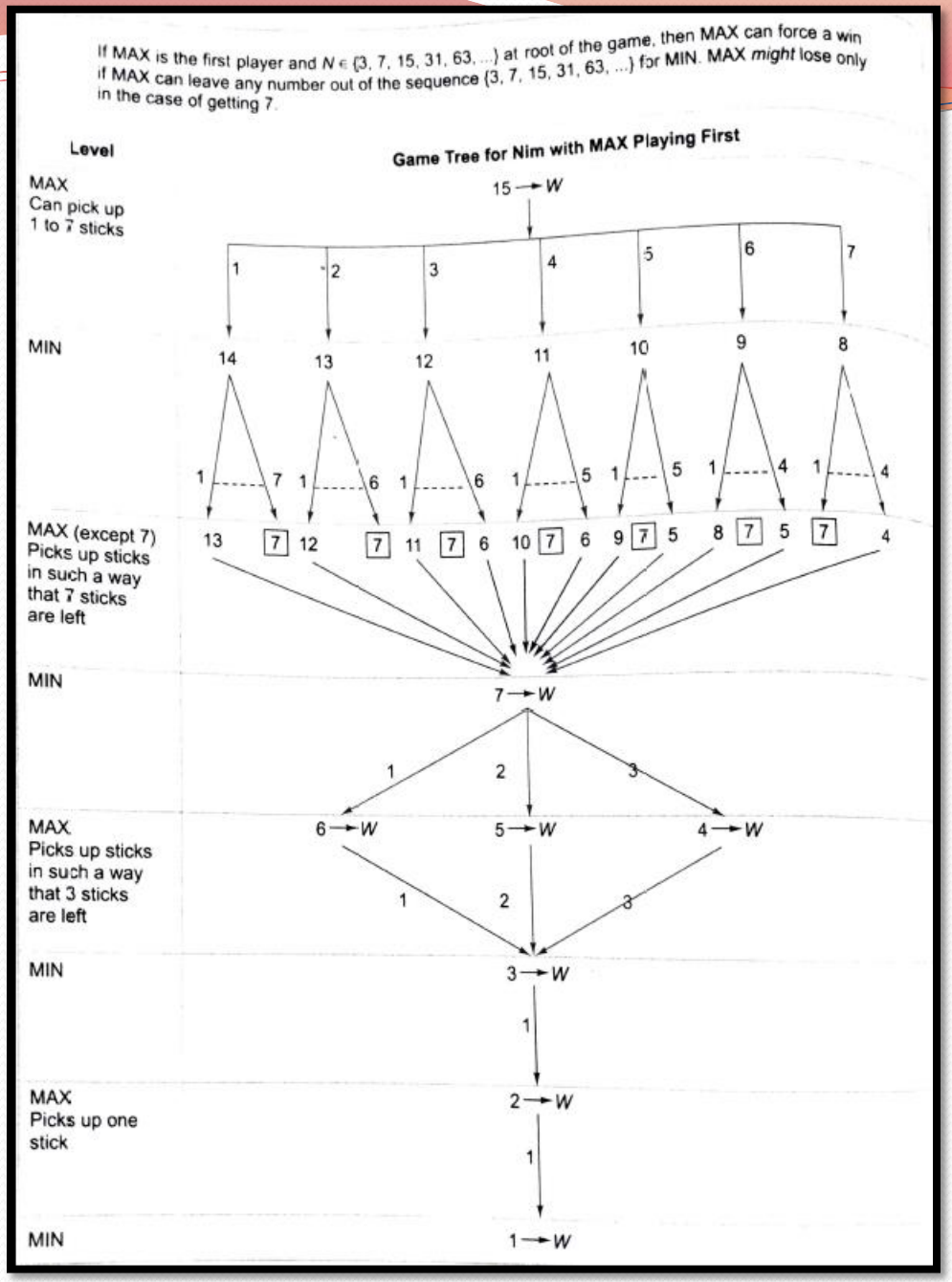
Case 2:
N=15

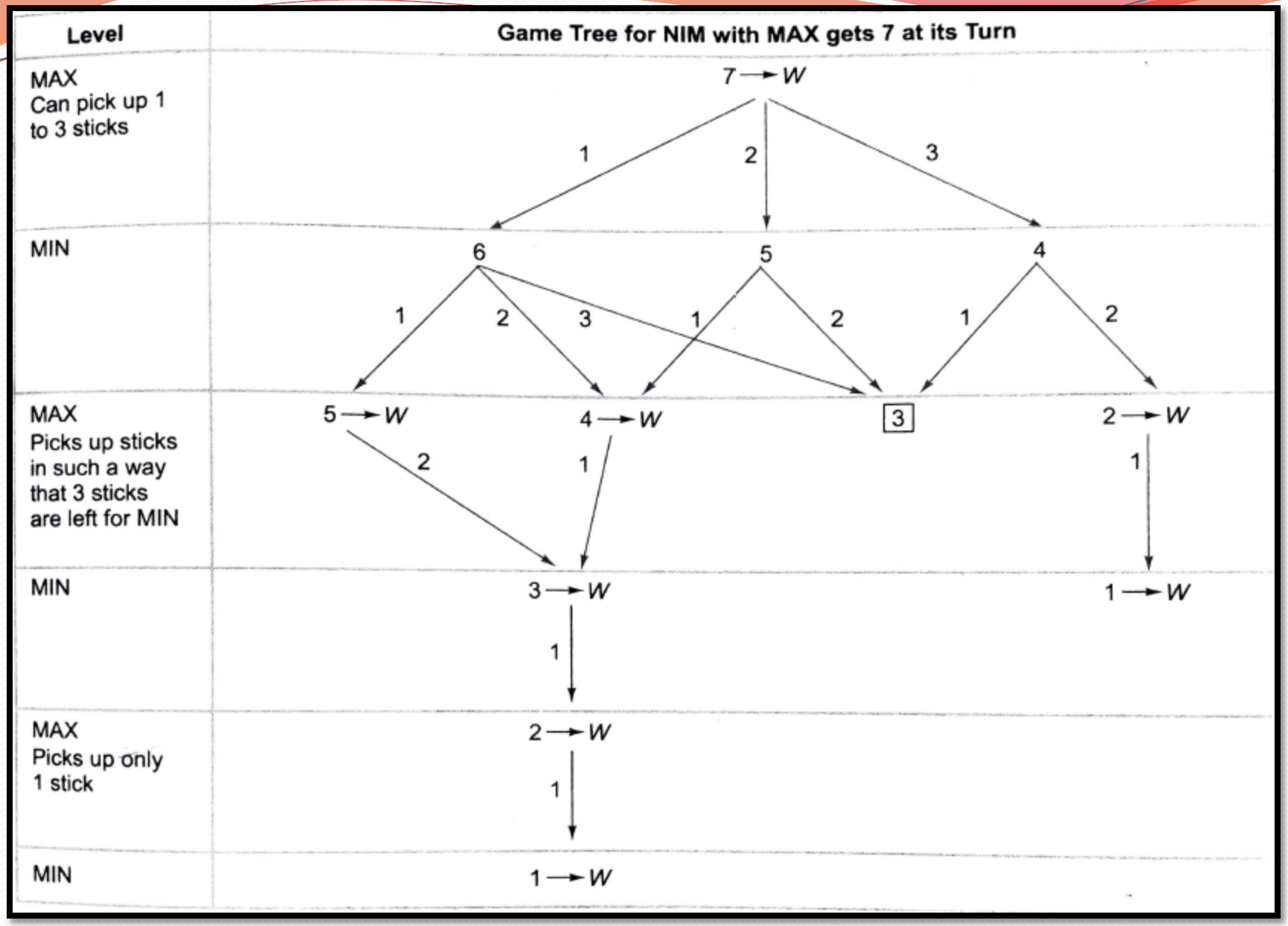
If MAX is the second player and N ∈ {3, 7, 15, 31, 63, ...}, then MAX will always win.



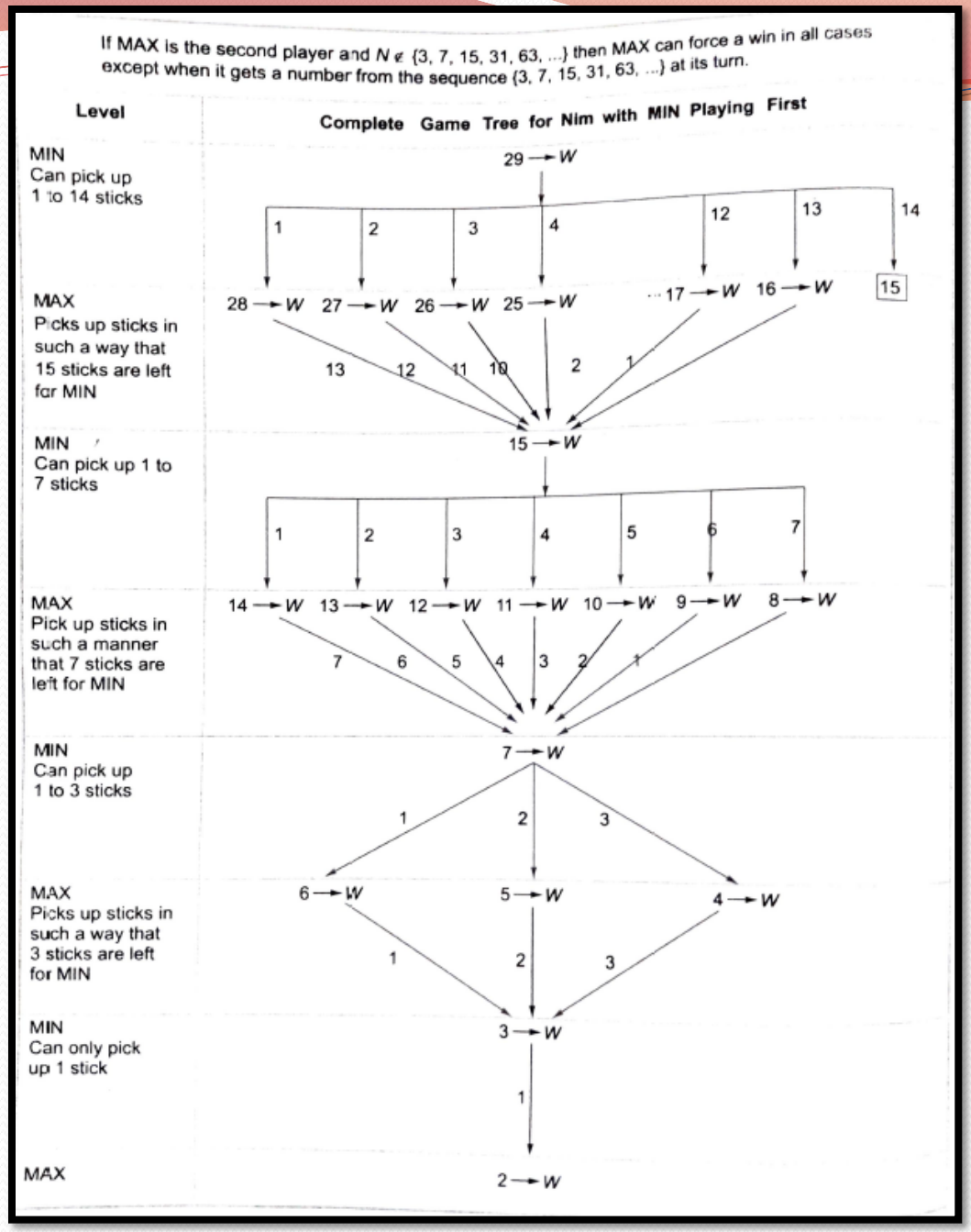
Case 3:
N=15

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

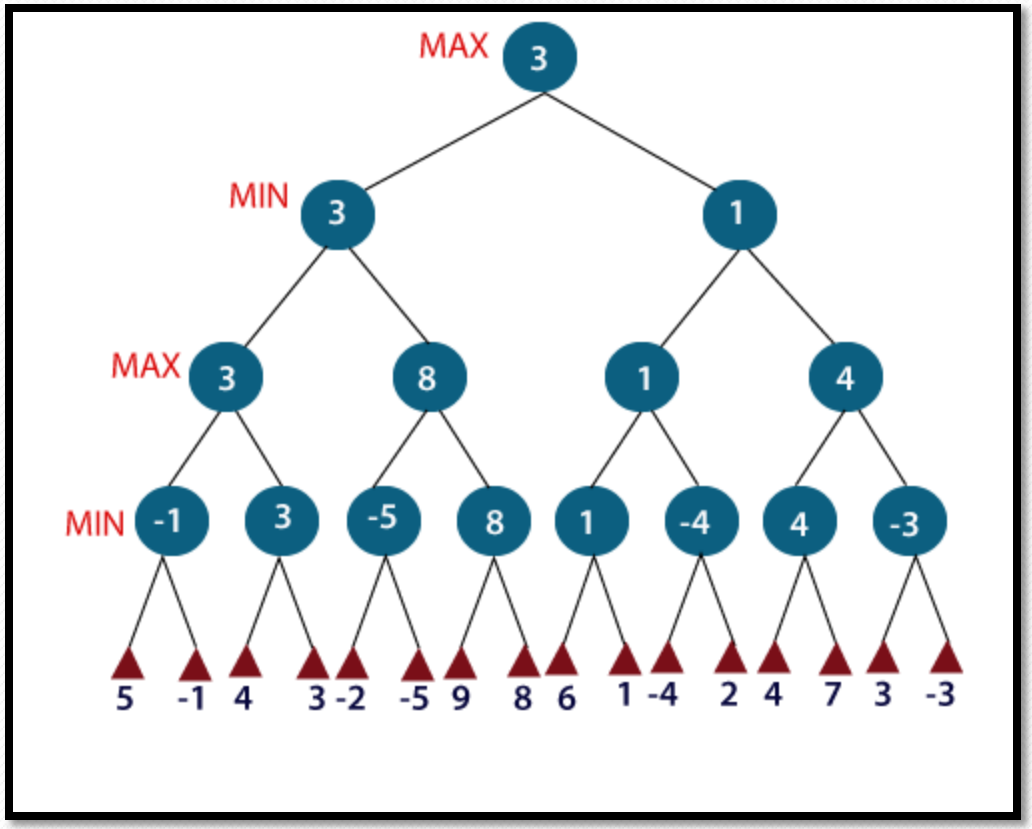



Level
Game Tree for NIM with MAX gets 7 at its Turn
MAX
Can pick up 1
to 3 sticks
7 → W
1
2
3
MIN
6
5
4
1
2
3
1
2
1
2
MAX
Picks up sticks
in such a way
that 3 sticks
are left for MIN
5 → W
4 → W
3
2 → W
2
1
1
MIN
3 → W
1 → W
1
MAX
Picks up only
1 stick
2 → W
1
MIN
1 → W

Case 4:
N=29


If MAX is the second player and N ∉ {3, 7, 15, 31, 63, ...} then MAX can force a win in all cases except when it gets a number from the sequence {3, 7, 15, 31, 63, ...} at its turn.
Level
Complete Game Tree for Nim with MIN Playing First
MIN
Can pick up
1 to 14 sticks
MAX
Picks up sticks in such a way that 15 sticks are left for MIN
MIN
Can pick up 1 to 7 sticks
MAX
Pick up sticks in such a manner that 7 sticks are left for MIN
MIN
Can pick up
1 to 3 sticks
MAX
Picks up sticks in such a way that 3 sticks are left for MIN
MIN
Can only pick up 1 stick
MAX
29 → W
28 → W
27 → W
26 → W
25 → W
17 → W
16 → W
15
15 → W
14 → W
13 → W
12 → W
11 → W
10 → W
9 → W
8 → W
7 → W
6 → W
5 → W
4 → W
3 → W
2 → W

MINIMAX Procedure



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
- It 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
- This strategy requires the maintenance of two threshold values: one representing a lower bound (α) on the value that a maximizing node may ultimately be assigned (called alpha) and another representing upper bound (β) on the value that a minimizing node may be assigned (called beta)

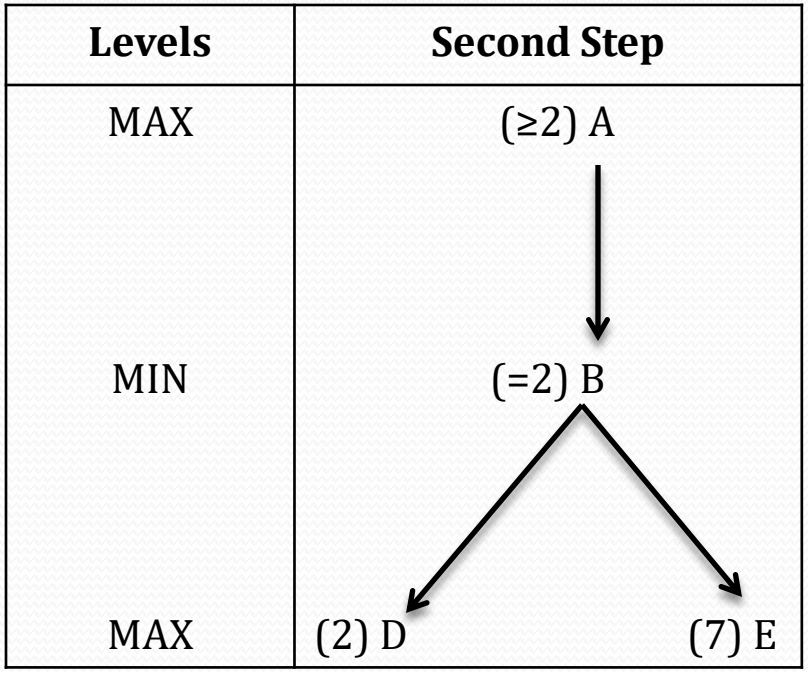
Alpha –Beta Pruning

- 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

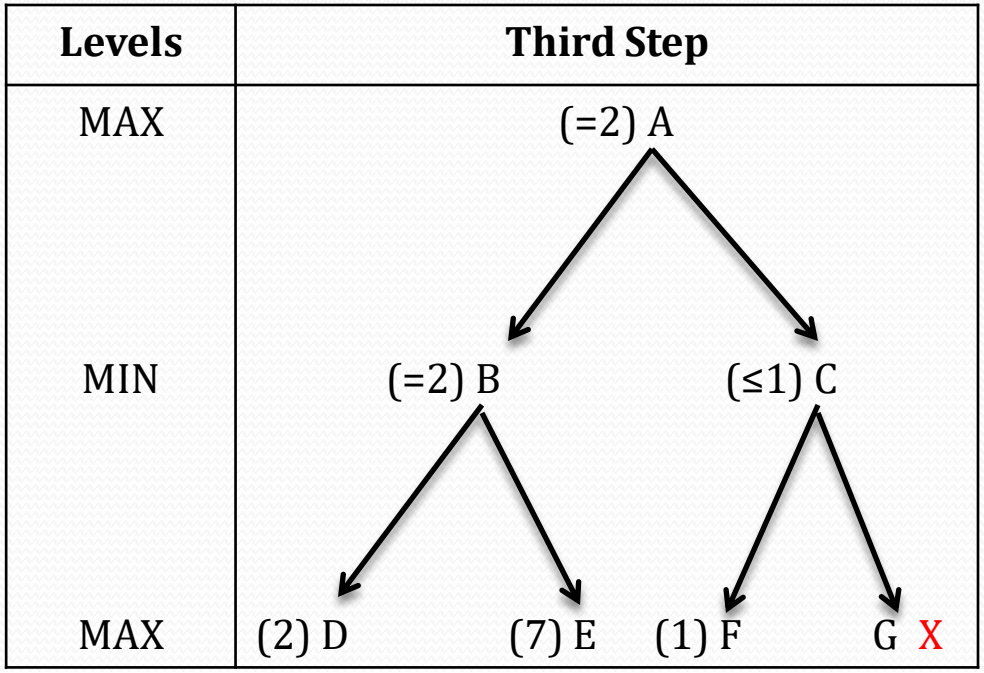
α-β pruning algorithm: Step 1

Levels	First Step
MAX	A
MIN	(≤2)B
MAX	(2) D

Alpha –Beta Pruning

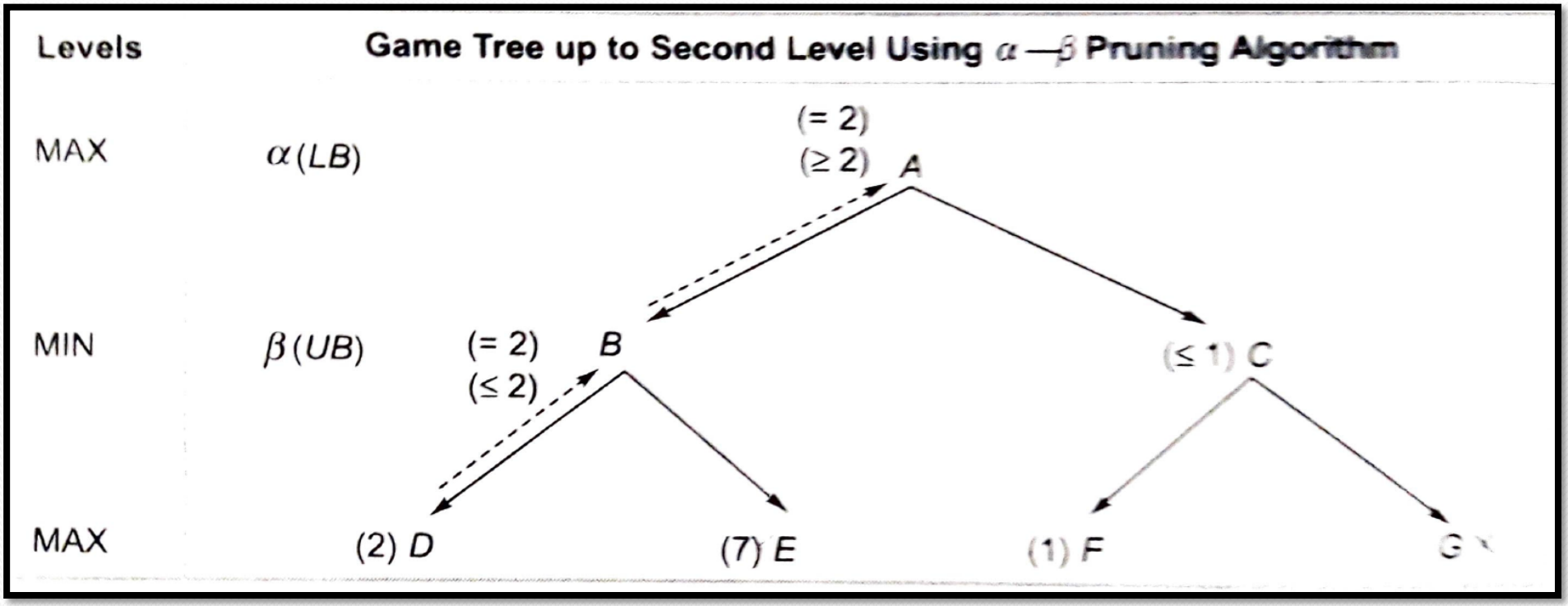


α-β pruning algorithm: Step 2



α-β pruning algorithm: Step 3

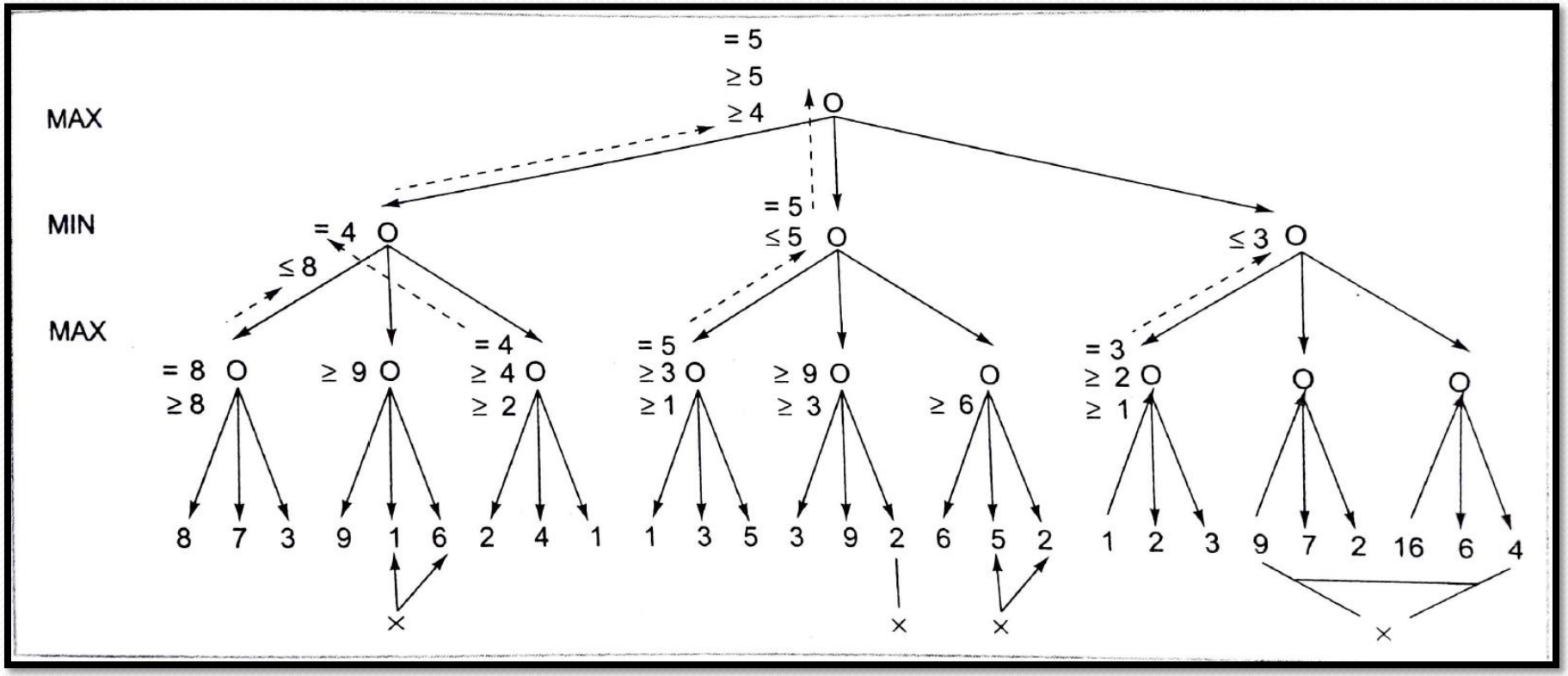
Alpha –Beta Pruning



Game tree generation using α-β Pruning algorithm

Alpha –Beta Pruning

- Consider a Game tree of depth 3 and branching factor 3
- If the full game tree of depth 3 is generated then there are 27 leaf nodes for which static evaluation needs to be done
- If we apply the α-β pruning, then only 16 static evaluations need to be made



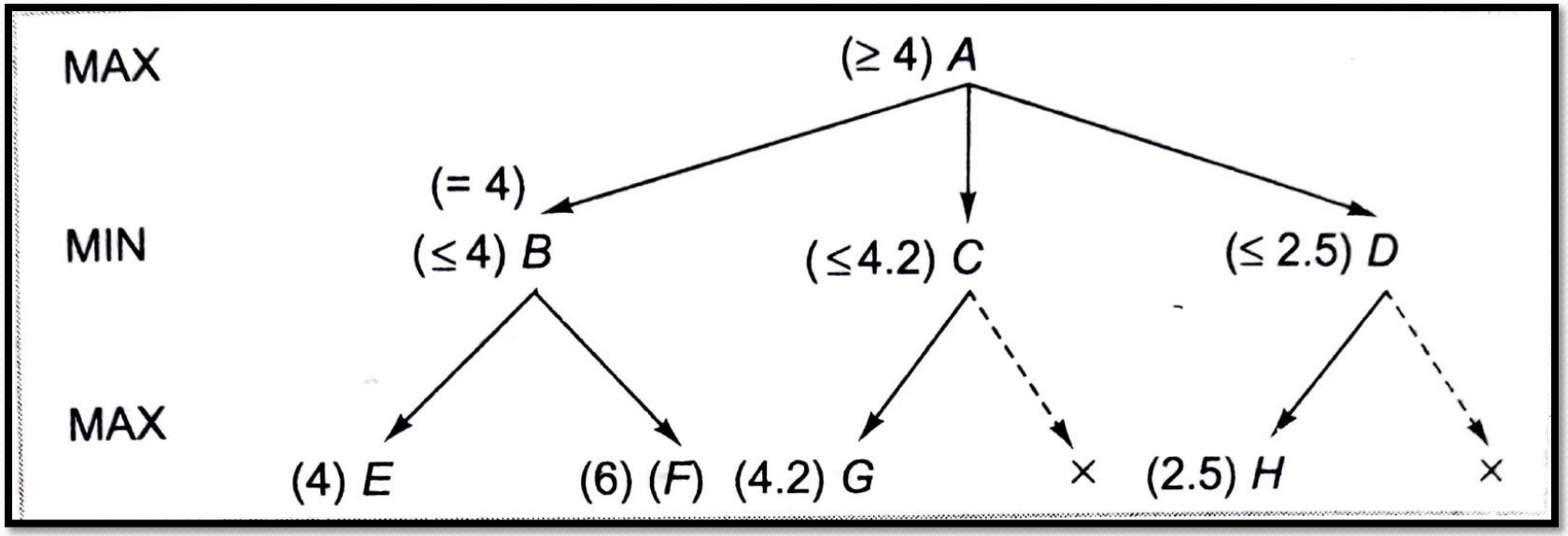
Alpha –Beta Pruning Algorithm

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if β ≤ α then
                break (* α cutoff *)
        return value

(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

Refinements to α-β Pruning

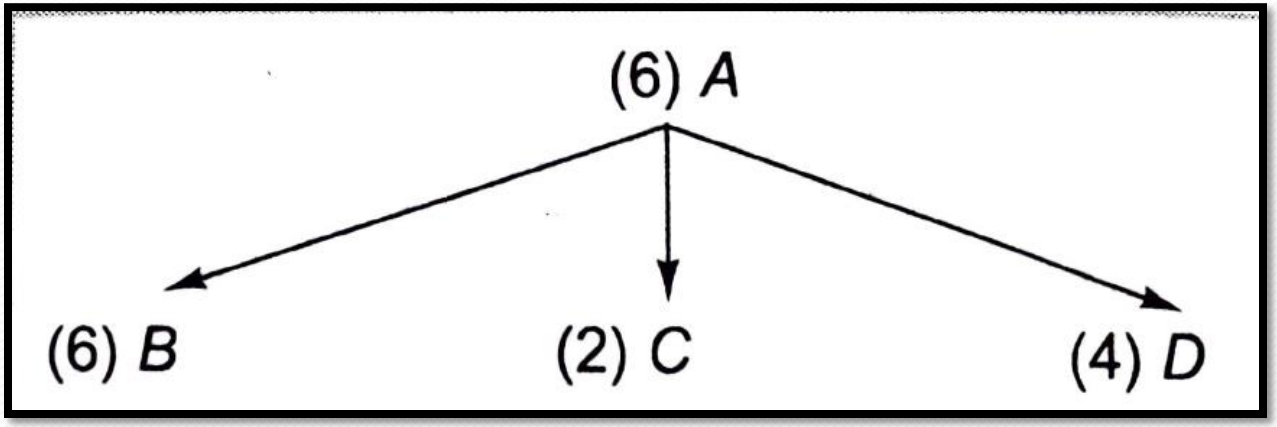
1. Pruning of Slightly Better Paths



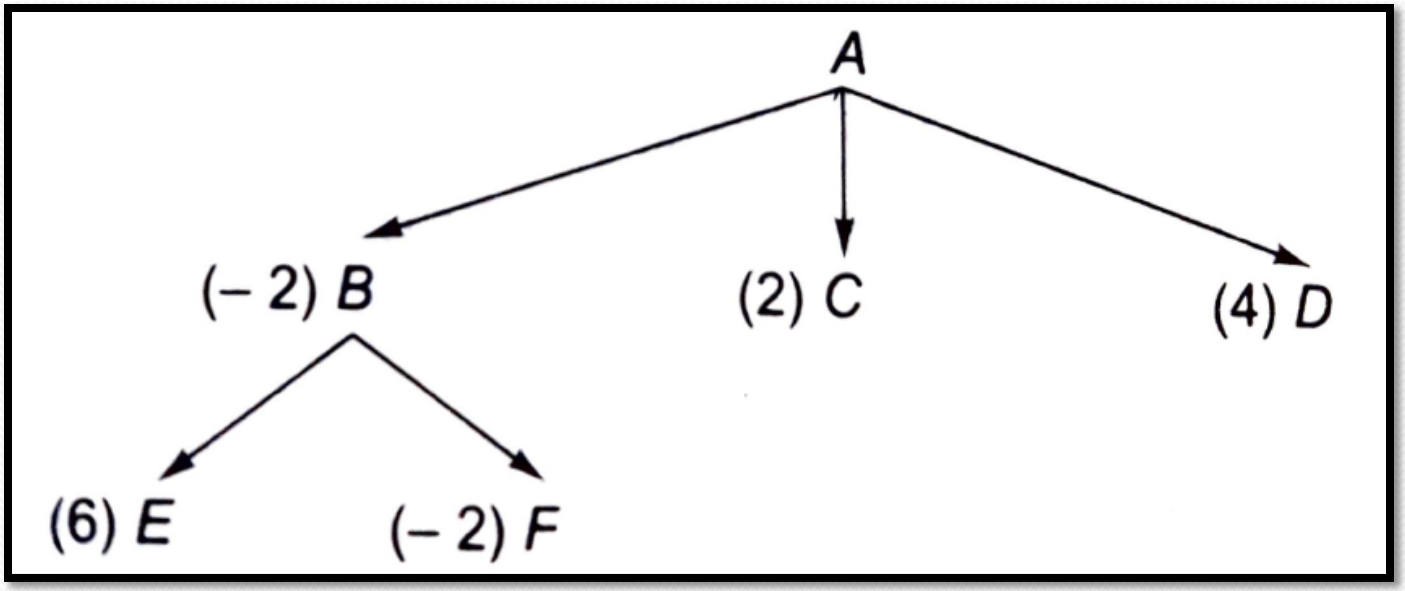
Pruning of Slightly Better Paths

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

2. Waiting for Quiescence



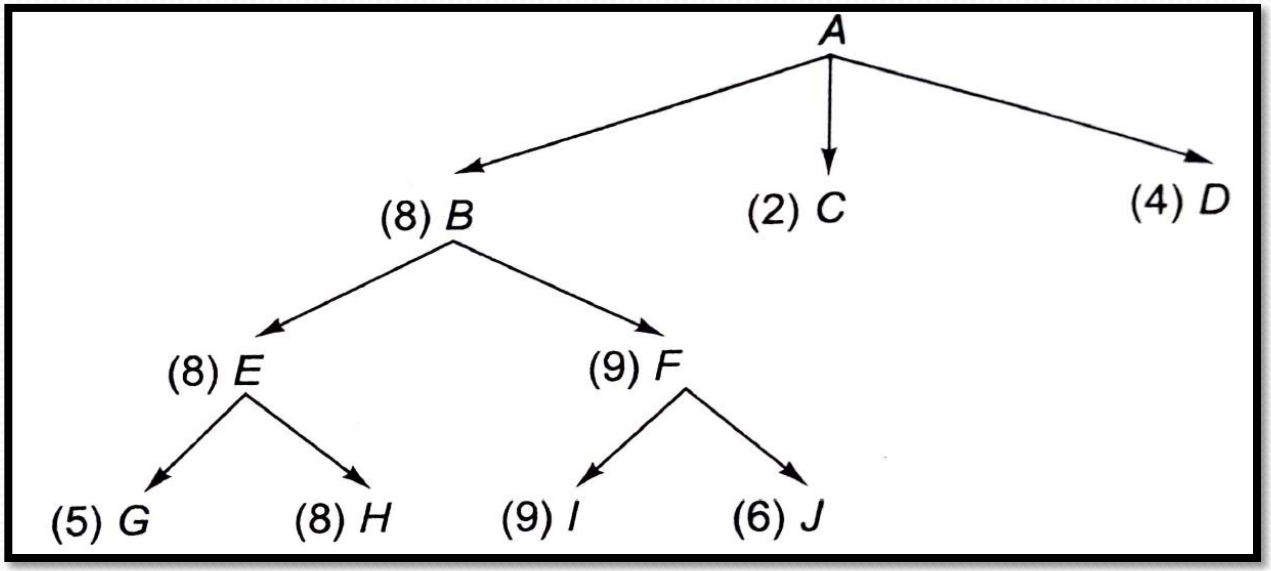
A One-Level Deep Game Tree



Expansion of Node B

2.Waiting for Quiescence

- We can see that our estimate of the worth of Node B has changed. We can stop exploring the tree at this level and assign a value of -2 to B and therefore decide that B is not a good move. Such short-term measures do not unduly influence our choice of moves, we should continue the search further until no such drastic change occurs from one level to the next or till the condition is stable. Such situation is called Waiting for Quiescence.



Extending Unstable Node to Obtain Stable Condition

3. Secondary Search

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

Alternative to α-β Pruning MINIMAX Procedure

- The α-β Pruning MINIMAX Procedure has some problems even with all the refinements.
- It is based on the assumption that the opponent will always choose an optimal move. In a winning situation, this assumption is acceptable, but in a losing situation, one may try other options and gain some benefit in case the opponent makes a mistake.
- Suppose we have to choose one move out of two possible moves, both of which may lead to bad situations for us if the opponent plays perfectly. MINIMAX procedure will always choose the bad move out of the two, however, here we can choose an option which is slightly less bad than the other.

Alternative to α-β Pruning MINIMAX Procedure

- There might be a situation when one move appears to be only slightly more advantageous than the other. Then, it might be better to choose the less advantageous move.
- To implement such systems, we should have a model of individual playing styles of opponents.

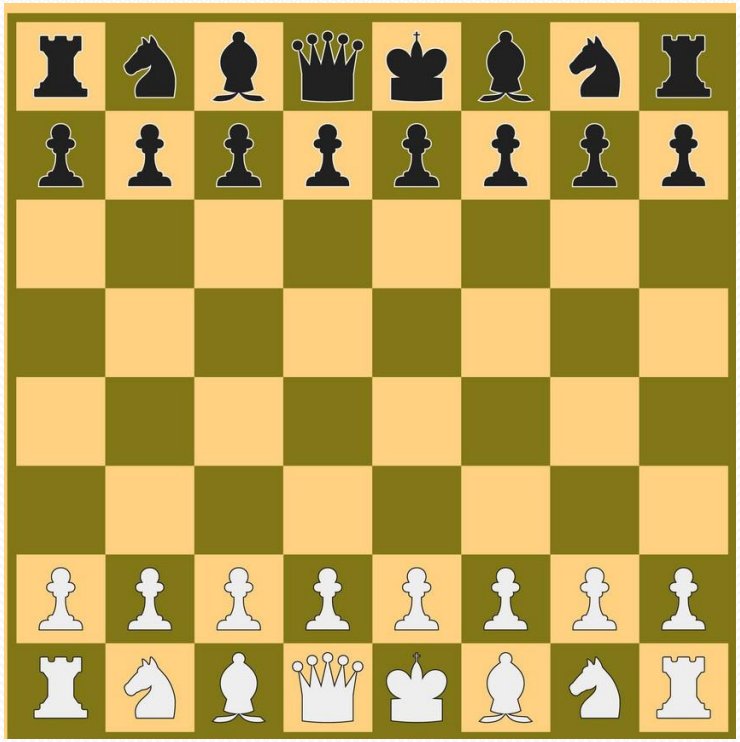
Iterative Deepening

- Rather than searching till a fixed depth in a given game tree, it is advisable to first search only one-ply, then apply MINIMAX to two-ply, then three-ply till the final goal state is searched
- For eg. CHESS 5 uses this procedure
- In competitions, there is an average amount of time allowed per move. The idea that enables us to conquer this constraint is to do as much look-ahead as can be done in the available time
- While using iterative deepening we can keep on increasing the look-ahead depth until we run out of time
- We can arrange to have a record of the best move for a given look-ahead even if we have to interrupt our attempt to go one level deeper
- With effective ordering, α-β pruning MINIMAX algorithm can prune many branches and the total search time can be decreased.

Two-Player Perfect Information Games

1. Chess

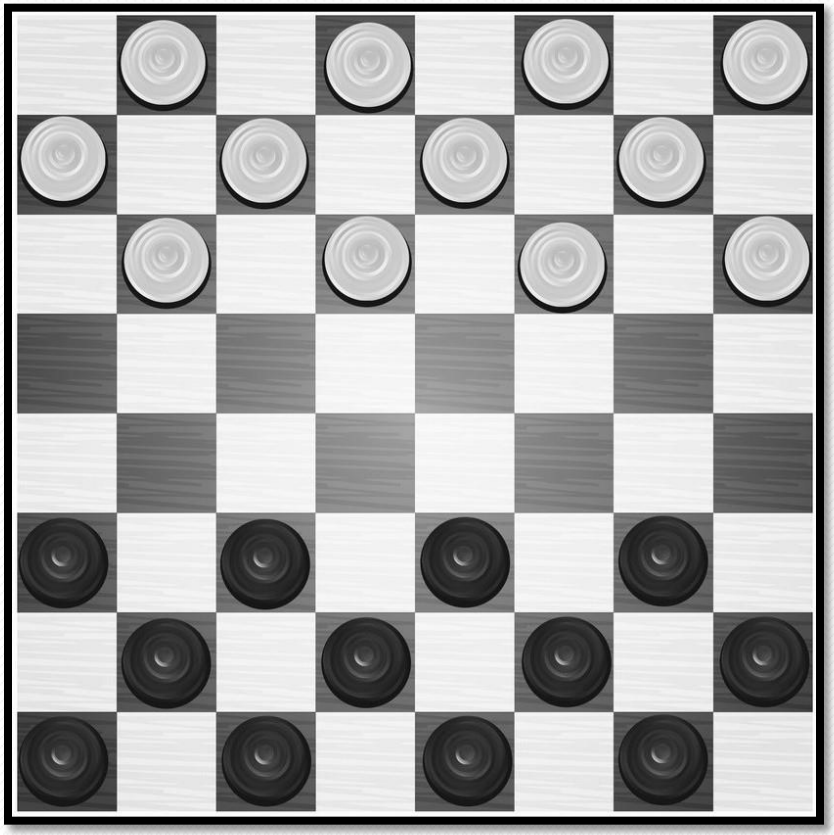
- Chess is a competitive two-player game played on a chequered board with 64 squares arranged in a 8 X 8 square



Two-Player Perfect Information Games

2. Checkers (Draughts)

- Checkers is a two-player game played on a chequered 8 X 8 square board



Two-Player Perfect Information Games

2. Checkers (Draughts)

- 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
- 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
- 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 kings 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

Two-Player Perfect Information Games

3. Othello (Reversi)

- It is a two-player board game which is played on a 8 X 8 square grid with pieces that have two distinct bi-coloured sides

Two-Player Perfect Information Games

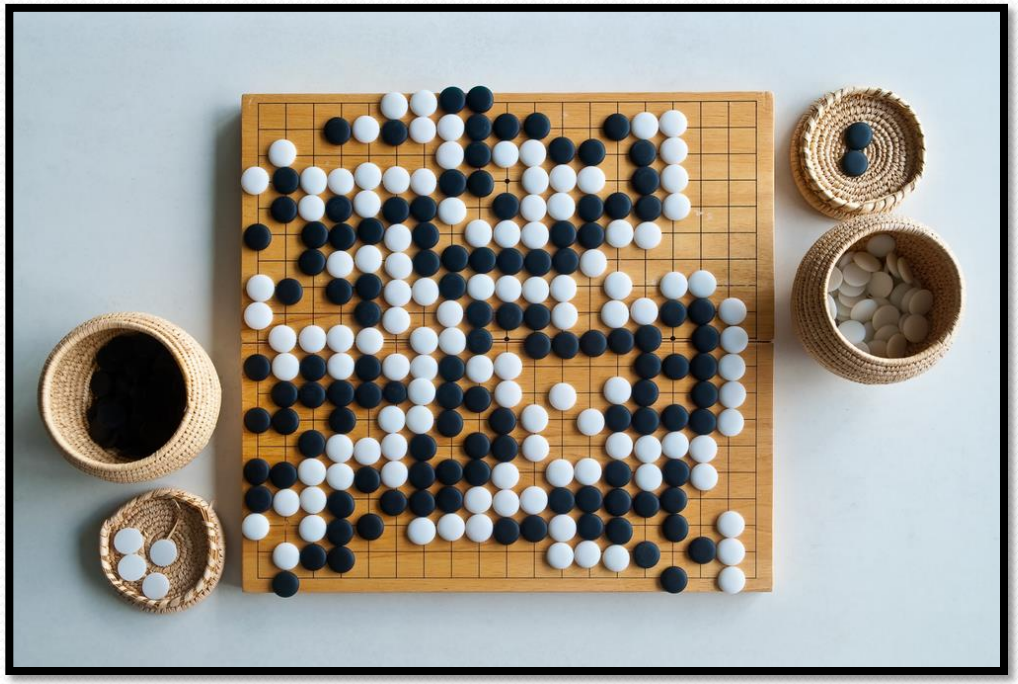
3. Othello (Reversi)

- There are sixty-four identical game pieces called *disks* (often spelled "discs"), which are light on one side and dark on the other.
- Players take turns placing disks on the board with their assigned color facing up.
- During a play, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color.
- The object of the game is to have the majority of disks turned to display your color when the last playable empty square is filled.

Two-Player Perfect Information Games

4. Go

- It is a strategic two-player board game in which the players play alternately by placing black and white stones on the vacant intersection of a 19 X 19 board



Two-Player Perfect Information Games

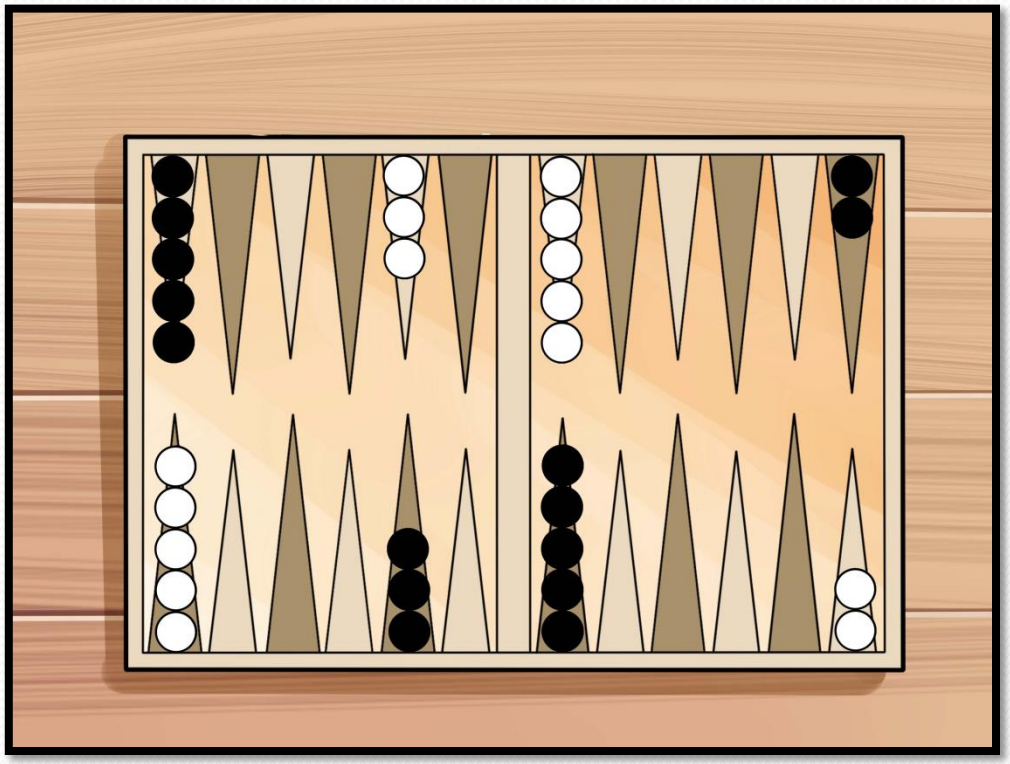
4. Go

- A game of Go starts with an empty board. Each player has an effectively unlimited supply of pieces (called stones), one taking the black stones, the other taking white. The main object of the game is to use your stones to form territories by surrounding vacant areas of the board. It is also possible to capture your opponent's stones by completely surrounding them.
- Players take turns, placing one of their stones on a vacant point at each turn, with Black playing first. Note that stones are placed on the intersections of the lines rather than in the squares and once played stones are not moved. However they may be captured, in which case they are removed from the board, and kept by the capturing player as prisoners.

Two-Player Perfect Information Games

5. Backgammon

- It is a two-player board game in which the playing pieces are moved using dice



Two-Player Perfect Information Games

5. Backgammon

- It is a two-player board game in which the playing pieces are moved using dice
- A player wins by removing all of his pieces from the board
- Luck and strategy play an important role
- Two players have 15 checkers of one colour each
- With each roll of the dice a player must choose from numerous options for moving his/her checkers and anticipate the possible counter moves by the opponent