Unit II: Approaches of Artificial Intelligence (12 Hours)

2.1 Characteristics of AI Problems: Well Defined Problems,Constraint Satisfaction Problem
2.2 Problem Formulation
2.2.1 Problem Specification
2.2.2 State Space Search with examples (8-puzzle, TSP, Water Jug Problem)
2.2.3 Problem Reduction
2.2.4 Production System
2.2 Searching Techniques
2.2.1 Types of Searching: Uninformed and Informed
2.2.2 Breadth First Search (BFS)
2.2.3 Depth First Search (DFS)
2.2.4 Bidirectional Search
2.2.5 Hill Climbing Search
2.2.6 Simulated Annealing Search
2.2.6 Greedy Search/Best First Search
2.2.7 A* Search
2.3 Min-Max Algorithm
2.4 Alpha-Beta Pruning (Cutoff)

2.1 Characteristics of AI Problems: Well Defined Problems,Constraint Satisfaction Problem

AI problems are typically formulated so that an intelligent agent can search for a solution efficiently. Two important categories are Well-Defined Problems and Constraint Satisfaction Problems.

1. Well-Defined Problems

A well-defined problem is one where all aspects of the problem are clearly specified. These problems are suitable for classical AI search techniques.

Characteristics:

1. Initial State

The starting point of the problem.
Example: Initial configuration of a chessboard.

2. State Space

The set of all possible states reachable from the initial state.
Example: All possible legal board positions in chess.

3. Actions (Operators)

The set of actions that move the system from one state to another.
Example: Legal chess moves.

4. Transition Model

Describes how actions change the state.
Example: Moving a piece updates the board configuration.

5. Goal Test

A condition to determine whether a goal has been reached.
Example: Checkmate in chess.

6. Path Cost

A numeric cost associated with a path.
Example: Number of moves taken to reach the goal.

Examples:
8-Puzzle problem
Maze navigation
Chess (in simplified form)

2. Constraint Satisfaction Problems (CSPs)

A Constraint Satisfaction Problem (CSP) is a special type of well-defined problem where the solution must satisfy a set of constraints.

Components of a CSP:

1. Variables
Objects whose values need to be assigned.
Example: X_1, X_2, X_3

2. Domains
Possible values each variable can take.
Example: $\{1, 2, 3\}$

3. Constraints
Rules that restrict the values variables can take together.
Example: $X_1 \neq X_2$

Characteristics:

Solution is a complete and consistent assignment of values to all variables.
Focuses on constraint satisfaction, not path cost.
Can be solved using backtracking, constraint propagation, and heuristic search.

Examples:
Map coloring problem
N-Queens problem
Sudoku
Scheduling problems

2.2 Problem Formulation

Problem formulation is the process of deciding how to represent a real-world problem in a form that an AI agent can understand and solve using search techniques.
A good problem formulation is crucial because different formulations of the same problem can lead to very different solution efficiencies.

Components of Problem Formulation

To formulate a problem, the following elements must be defined:

1. Initial State

The state from which the agent starts.
It describes the current situation of the environment.
Example: Starting position of a robot in a room.

2. State Space

The set of all possible states reachable from the initial state.
May be finite or infinite.
Example: All positions the robot can reach in the room.

3. Actions (Operators)

The set of actions available to the agent.
These actions transform one state into another.
Example: Move left, right, up, or down.

4. Transition Model

Describes the result of performing an action in a given state.
Often written as:
Result(state, action) → new state

5. Goal Test

A condition that determines whether a state is a goal state.
Example: Robot reaches the target location.

6. Path Cost (Cost Function)

Assigns a numeric cost to each path or action.
Used to find optimal solutions.
Example: Distance traveled, time taken, or energy used.

Example of Problem Formulation
Vacuum Cleaner Problem
Initial State: Location of vacuum and dirt distribution
State Space: All possible combinations of agent location and dirt status
Actions: Move left, move right, suck
Goal Test: All rooms are clean
Path Cost: Number of actions performed

Importance of Problem Formulation

Determines efficiency of the solution
Helps in selecting the appropriate search strategy
Poor formulation can make a simple problem computationally expensive

2.2.1 Problem Specification

Problem Specification is the formal description of a problem that an AI agent must solve.
It clearly defines what the agent knows, what it wants to achieve, and what actions are available.
A well-specified problem allows the AI system to create a structured model for planning and searching.

A complete problem specification includes:

1. Initial State
The starting condition of the problem.

2. Goal State
The desired final condition the agent must reach.

3. Operators / Actions
All allowed moves the agent can take to change the state.

4. State Space
The set of all possible states reachable from the initial state.

5. Path Cost
A measure that assigns a cost/value to each action (optional but useful for optimization).

Real-Life Example: Mobile Phone Unlock Pattern
• Initial State: Locked screen
• Goal State: Enter correct unlock pattern
• Operators: Swipe up/down/diagonal to connect dots
• State Space: All possible patterns connecting the dots
• Path Cost: Minimum number of moves (optional)

2.2.2 State Space Search with examples (8-puzzle, TSP, Water Jug Problem)

State Space Search is the method used by an AI agent to explore all possible states (situations) to reach the goal
state from the initial state.
A state space is represented as a graph or tree, where:
• Nodes = States
• Edges = Actions
• Path = Sequence of actions
• Goal Test = Checks if the goal state is reached
AI uses search algorithms like BFS, DFS, A* to move through the state space efficiently.

Key Components of State Space

1. State: A specific configuration of the system.
2. Initial State: Where the search begins.
3. Goal State: Desired condition to be reached.
4. Actions: Moves that transform one state to another.
5. Search Path: Sequence of actions leading to the goal.

Examples of State Space Search

Example 1: 8-Puzzle

The 8-puzzle problem is one of the most common examples used to explain state space search in AI.
It consists of 8 numbered tiles and one blank space arranged on a 3×3 board.
The objective is to transform the initial state into the goal state using legal moves.

1. States (Representation)
A state describes the current arrangement of the tiles and the blank.
Initial State
6 3 1
8 _ 5
2 4 7
Goal State
1 2 3
8 _ 6
7 5 4
Any valid 3×3 arrangement is a possible state.

2. Initial State
Any valid board configuration may serve as the starting point.

3. Goal State
Any configuration may be chosen as the goal.
In many books, the “standard goal state” is:
1 2 3
4 5 6
7 8 _
But the goal can be changed depending on the problem.

4. Legal Moves (Operators)

The blank tile _ may move in four directions (if possible):

1. Blank moves left
2. Blank moves right
3. Blank moves up
4. Blank moves down

Each move generates a new legal state.

5. Path Cost

Each move costs 1 unit.

Thus, path cost = number of moves needed to reach the goal.

The optimal solution = minimum number of moves.

6. State Space Tree (as shown in Fig. 2.2)

The figure shows how the search expands from the root (initial state):

```
Initial State
|
|— First Move → New State
|
|— Second Move → Another State
|
… and so on
```

Each node represents a board configuration.

Each branch represents a move (action).

The goal is found when a node matches the goal state.

Example 2: Water Jug Problem

The Water Jug Problem is a classical AI example used to demonstrate state space representation and search strategies.

We are given two jugs of fixed capacities, and the objective is to measure a specific quantity of water using only these jugs, without any measuring scale.

Problem Setup

• One 4-liter jug
• One 3-liter jug

Goal: Measure exactly 2 liters of water using these jugs.

1. States (Representation)

A state is represented as a pair (x, y):
 x = amount of water in the 4-liter jug
 y = amount of water in the 3-liter jug

Example states:
 (0,0) → both jugs empty
 (4,0) → 4-liter jug full
 (1,3) → 4-liter jug has 1 liter, 3-liter jug full

2. Initial State

(0, 0)
Both jugs are initially empty.

3. Goal State

Any state where the 4-liter jug contains 2 liters, such as:
(2, y)
(y may be 0 or any valid value)

4. Legal Actions (Operators)

The valid operations are:

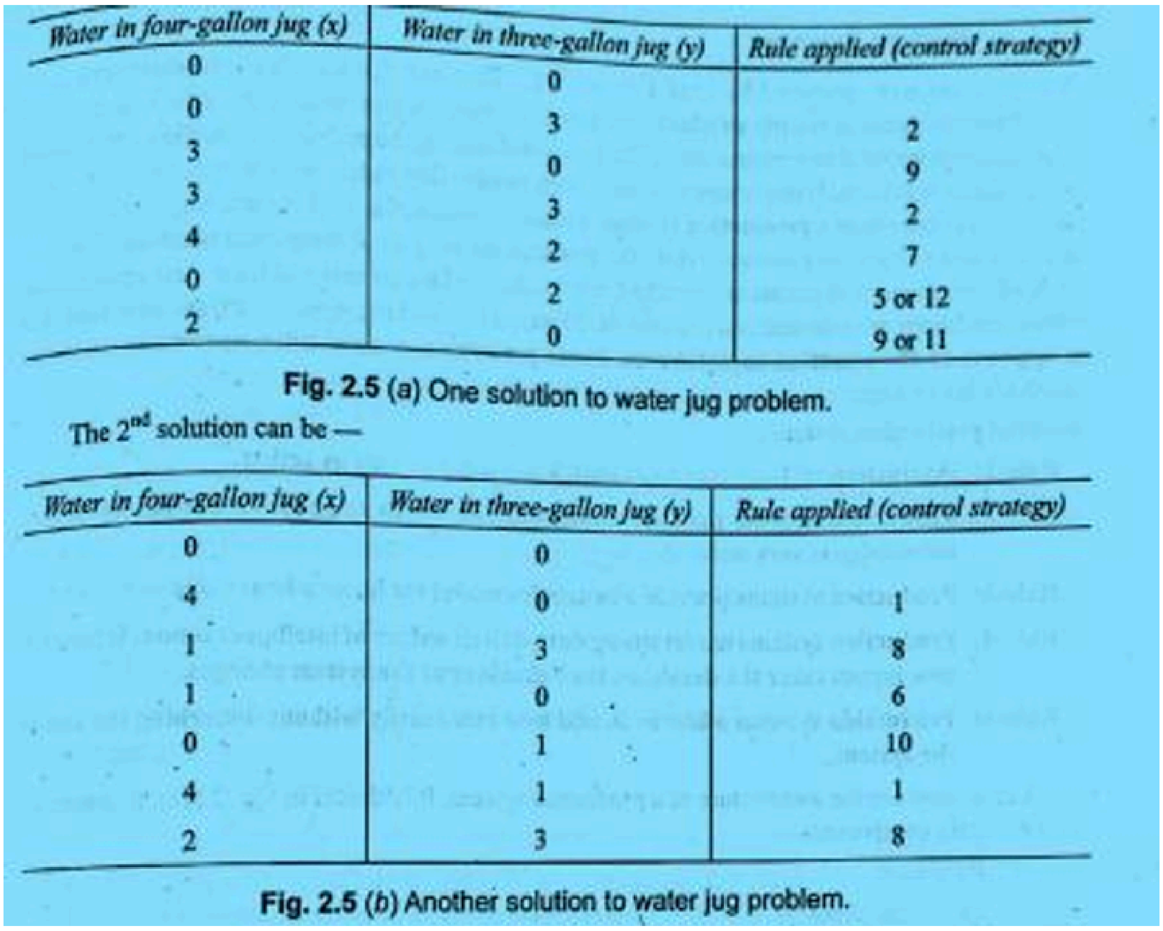
1. Fill a jug
o Fill 4-liter jug → (4, y)
o Fill 3-liter jug → (x, 3)

2. Empty a jug
o Empty 4-liter jug → (0, y)
o Empty 3-liter jug → (x, 0)

3. Pour water from one jug to another
o Pour from 4L to 3L
o Pour from 3L to 4L
o Stop when either receiving jug is full or giving jug is empty
These operators generate new legal states in the state space.

5. State Space Diagram (Conceptual)
(0,0)
|
(4,0)
|
(1,3)
|
(1,0)
|
(0,1)
|
(4,1)
|
(2,3)
|
✔ Goal reached: (2,3)
This sequence is one valid path to reach the goal.

6. Path Cost
Each action costs 1 step.
Total path cost = number of steps taken to reach a goal state.
Optimal solutions minimize the number of steps.



Water in four-gallon jug (x)	*Water in three-gallon jug (y)*	*Rule applied (control strategy)*
0	0	
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5 or 12
2	0	9 or 11

Fig. 2.5 (*a*) One solution to water jug problem.

The 2nd solution can be —

Water in four-gallon jug (x)	*Water in three-gallon jug (y)*	*Rule applied (control strategy)*
0	0	
4	0	1
1	3	8
1	0	6
0	1	10
4	1	1
2	3	8

Fig. 2.5 (*b*) Another solution to water jug problem.

Example 3: Traveling Salesman Problem (TSP)
Initial State: Starting city.
Goal State: Return to the starting city after visiting all cities exactly once.
Operators: Move from one city to another.
State Space: All possible permutations of city visits (n! possibilities for n cities).
Path Cost: Total distance or travel cost.

Solution Method:
Exact: Dynamic programming, branch and bound.
Approximate: Genetic algorithms, simulated annealing, heuristics.

Illustration:
Cities: A, B, C, D
Possible path: A → B → C → D → A
Goal: Find the shortest tour covering all cities.

2.2.3 Problem Reduction

Problem Reduction is an approach in Artificial Intelligence where a complex problem is broken down into a set of smaller, simpler sub-problems.
Each sub-problem is easier to solve, and the final solution is obtained by combining the solutions of these smaller parts.
This method is especially useful when the original problem is too large or too complex to solve directly.

Problem Reduction = breaking a big problem → solving smaller parts → combining results → achieving the final solution.

Instead of solving the whole problem at once, AI reduces it into manageable segments, solves them individually, and then assembles the final solution.

Why Problem Reduction is Useful?

Simplifies complex tasks
Reduces computational effort
Allows reuse of smaller solutions
Helps structure the problem logically
Supports recursion and divide-and-conquer strategies

Classic AI Example: Missionaries and Cannibals Problem
The Missionaries and Cannibals Problem is a classical AI problem used to illustrate problem reduction, state representation, and safe/unsafe state evaluation.

The goal is to safely transport three missionaries (M) and three cannibals (C) across a river using a boat that carries a maximum of two people at a time.
A key constraint is: At no point can cannibals outnumber missionaries on either side, or the missionaries will be eaten.

1. Problem Setup
Initial State
All three missionaries and all three cannibals are on the left side of the river: (3M, 3C, Left)

Goal State
All must safely reach the right side: (0M, 0C, Right)

Boat Condition
• Boat can carry 1 or 2 people
• Boat must have at least one person to move
• Moves from left → right or right → left

2. State Representation
A state is represented as:
(M_left, C_left, Boat_side)
Example: (2, 3, Right)
Means:
• 2 missionaries on left
• 3 cannibals on left
• Boat on right side

3. Legal Moves (Operators)
Possible safe boat actions:
• Take 1 missionary
• Take 2 missionaries
• Take 1 cannibal
• Take 2 cannibals
• Take 1 missionary + 1 cannibal

Each action generates a new state, which must be checked for safety.

4. Safe vs Unsafe States

A safe state must satisfy:

• Either missionaries ≥ cannibals,

OR Missionaries = 0 (none to be harmed)

Unsafe Example:

(1M, 3C) → unsafe (cannibals outnumber missionary)

5. Problem Reduction Approach

The original problem is large, but AI reduces it into safe intermediate sub-states, for example:

(3,3,Left)

(3,1,Right)

(3,2,Left)

(1,1,Right)

(2,2,Left)

(0,0,Right) → Goal

The problem becomes solving a sequence of safe transitions.

6. Example Solution Path (Valid Moves)

One common shortest solution:

1. (3,3,L) → (3,1,R) [Two cannibals cross]
2. (3,1,R) → (3,2,L) [One cannibal returns]
3. (3,2,L) → (1,2,R) [Two missionaries cross]
4. (1,2,R) → (2,2,L) [One missionary returns]
5. (2,2,L) → (0,2,R) [Two missionaries cross]
6. (0,2,R) → (0,3,L) [One cannibal returns]
7. (0,3,L) → (0,1,R) [Two cannibals cross]
8. (0,1,R) → (0,0,L) [One cannibal returns]
9. (0,0,L) → (0,0,R) [Two cannibals cross]

Goal achieved safely.

2.2.4 Production System

A Production System is a rule-based model of problem solving used in Artificial Intelligence.
It consists of a set of conditions and actions that define how an intelligent agent transitions from one state to another within a search space. Production systems are central to solving problems like puzzles, planning, diagnosis, and rule-based reasoning.

Components of a Production System

1. Production Rules (IF–THEN Rules)
These specify how to transform one state into another.
Example:
IF blank is left of tile X THEN swap blank and X

2. Working Memory (Current State)
Stores information about the current situation of the problem.

3. Rule Interpreter / Control System
Determines which rule to apply next.
This involves search strategies like DFS or BFS.

4. Conflict Resolution Strategy
If multiple rules apply, the system decides which rule fires first.

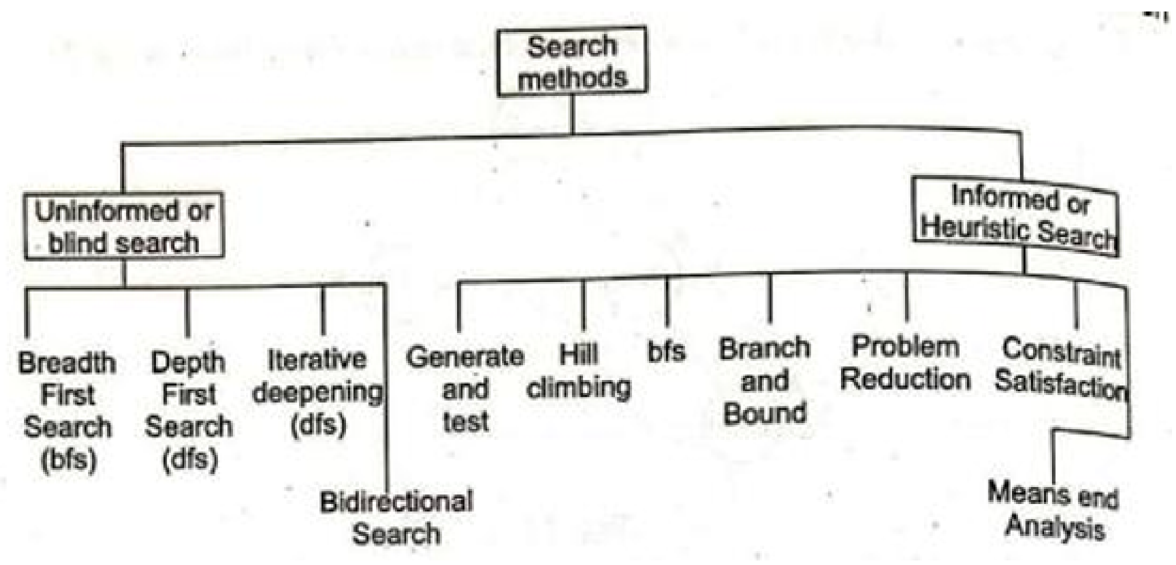
Production System as a Search Process
A production system creates a state space tree/graph.
Each rule application generates a new state.
State Space Representation
• Nodes = States
• Edges = Actions (rules applied)
• Root Node = Initial State
• Goal Test = Check if goal state reached
This makes DFS and BFS the core methods for exploring the state space.

2.2 Searching Techniques

Searching techniques are methods used by AI agents to explore the state space of a problem to find a solution. They are broadly classified into uninformed (blind) and informed (heuristic) search strategies.

Real-Life Example: Route Finding (Google Maps)
When you enter a destination:
• AI checks all possible roads (states)
• Chooses actions (turns)
• Uses heuristics like distance, traffic
• Finds the best path using A* search
This is a practical example of searching in real-world AI.

2.2.1 Types of Searching: Uninformed and Informed



AI search techniques are broadly divided into two major categories based on the type of information they use to explore the state space:
1. Uninformed (Blind) Search
2. Informed (Heuristic) Search
These two categories determine how intelligently the search tree is explored to find the goal state.

1. Uninformed (Blind) Search

Uninformed search also called blind search explores the search space without any domain specific knowledge or heuristics.
It treats all nodes equally and chooses which path to explore next based solely on general rules like node depth or path cost.
Real-Life Analogy:
Searching for a name in a phone contact list without knowing the first letter — checking one by one.

Common Uninformed Search Methods
1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)
3. Depth-Limited Search (DLS)
4. Iterative Deepening Search (IDS)
5. Uniform Cost Search (UCS)
6. Bidirectional Search

2. Informed (Heuristic) Search

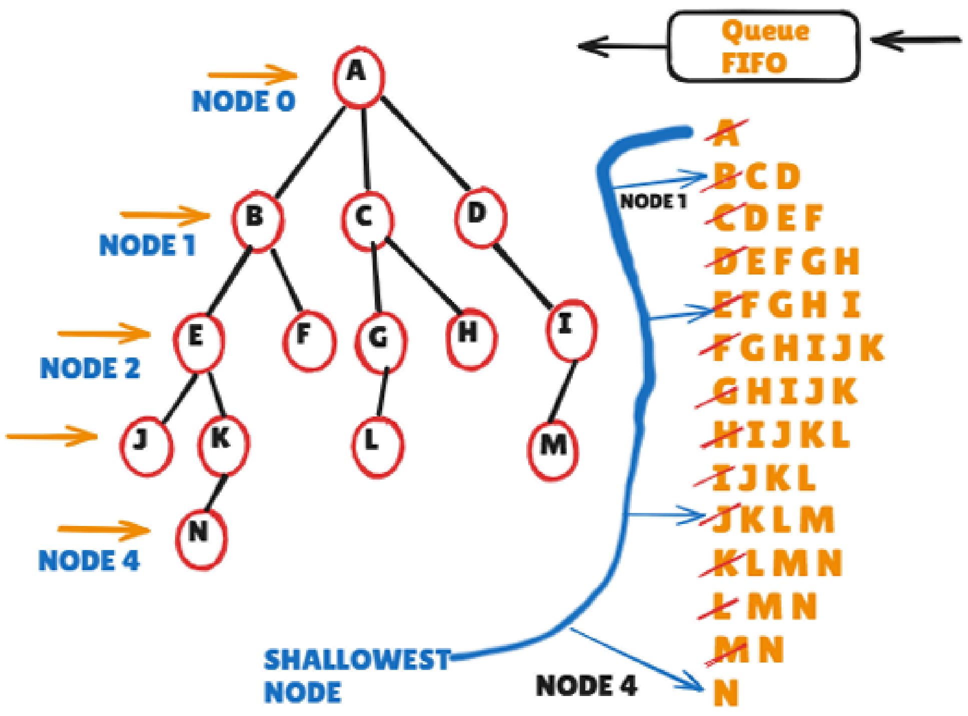
Informed search uses domain knowledge in the form of heuristics to make smarter decisions during the search process.
These heuristics estimate how close a state is to the goal guiding the search more efficiently.
Real-Life Analogy: Using Google Maps, which estimates distance and traffic to guide you to the best route.

Common Informed Search Methods
1. Greedy Best-First Search
2. A* Search (A-star)
3. Hill Climbing
4. Simulated Annealing

2.2.2 Breadth First Search (BFS)

Breadth-First Search (BFS) is an uninformed search technique that explores the search space level by level.
It starts from the initial state and expands all its neighbors before moving to the next level.
BFS guarantees that the first time it encounters the goal state, the path found is the shortest (minimum number of steps), making it complete and optimal for equal-cost problems.

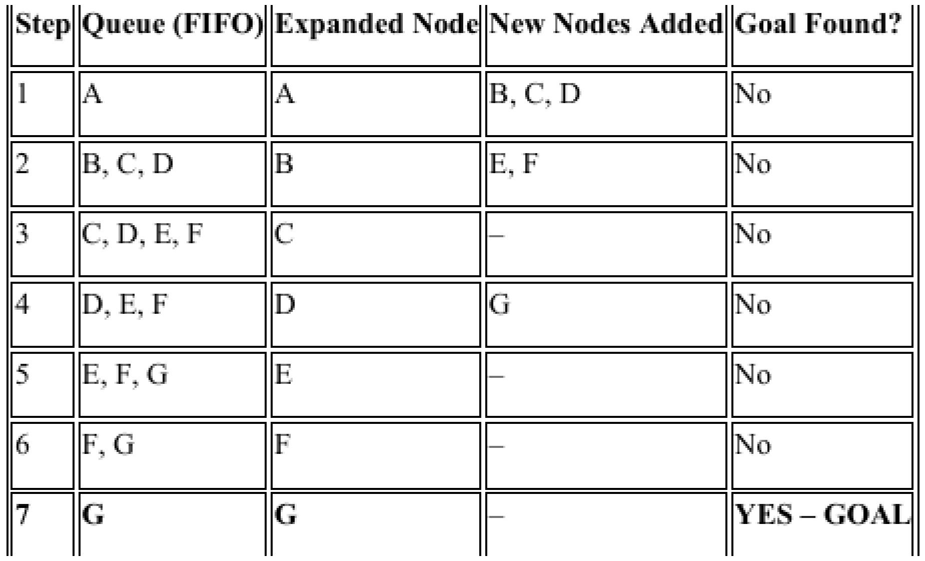
Key Characteristics of BFS
• Explores nodes in increasing depth
• Uses a FIFO Queue
• Guaranteed to find the shortest path
• Time and space complexity can be high



BFS Example: Simple Graph Search

```
   A
/  |  \
B  C  D
/ \       \
E   F     G   ← Goal
```

We search for goal G starting from A.



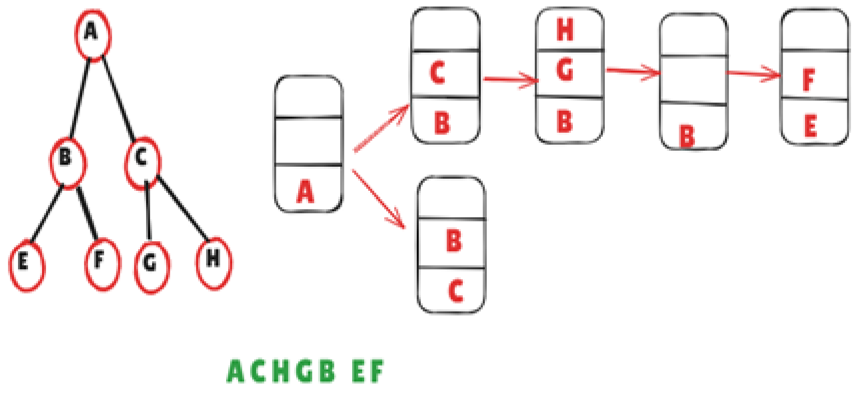
Step	Queue (FIFO)	Expanded Node	New Nodes Added	Goal Found?
1	A	A	B, C, D	No
2	B, C, D	B	E, F	No
3	C, D, E, F	C	–	No
4	D, E, F	D	G	No
5	E, F, G	E	–	No
6	F, G	F	–	No
7	**G**	**G**	–	**YES – GOAL**

2.2.3 Depth First Search (DFS)

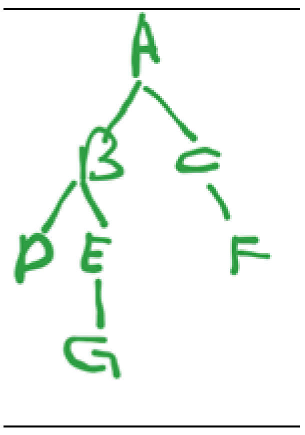
Depth-First Search (DFS) is an uninformed search technique that explores the search space by going as deep as possible along one path before backtracking.
DFS uses a stack structure (LIFO), meaning the most recently generated node is expanded first.
DFS is memory efficient because it stores only the current path, but it does not guarantee the shortest path and can get stuck in deep or infinite branches.

Key Characteristics of DFS
• Explores one branch deeply before exploring
others
• Uses Stack (LIFO)
• Low memory usage

• Not optimal (may find long or wrong path first)
• Can get stuck in cycles or infinite paths unless controlled



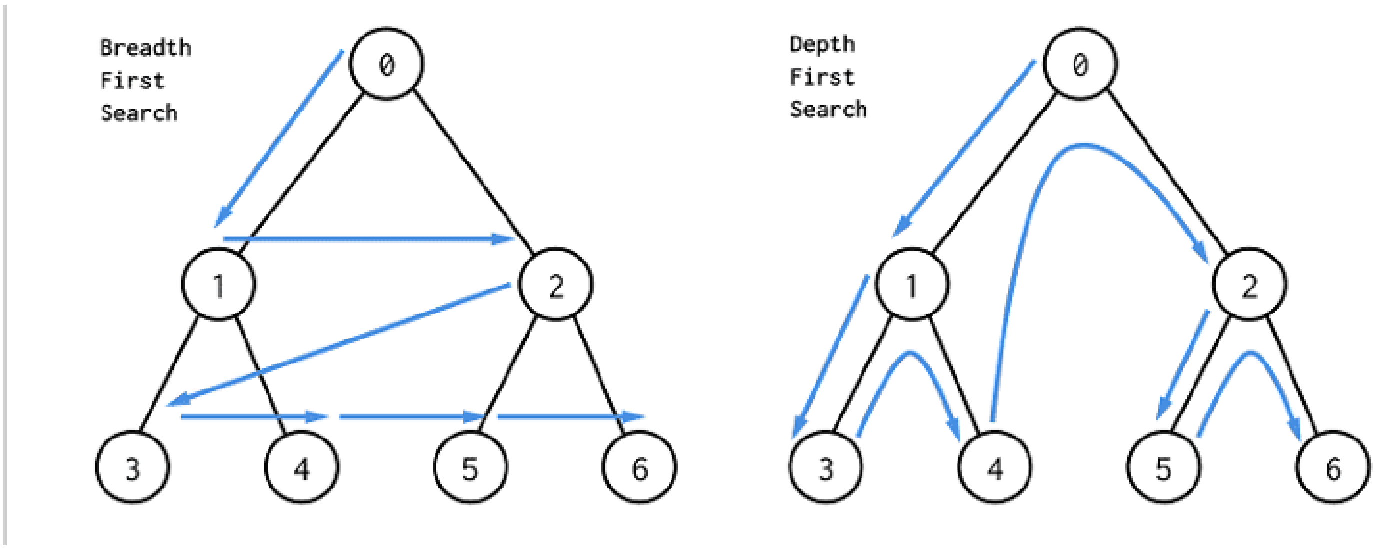
Numerical Question (DFS – Graph Based)
Q. Consider the following undirected graph:
• A is connected to B, C
• B is connected to A, D, E
• C is connected to A, F
• D is connected to B
• E is connected to B, G
• F is connected to C
• G is connected to E
Assume that when exploring neighbors, the nodes are taken in alphabetical order.Perform Depth-First Search (DFS) starting from node A and show the contents of the stack at each step.



Answer: DFS Traversal & Stack Trace
✔ DFS Traversal Order:
A → B → D → E → G → C → F

Step-by-Step DFS Using Stack (LIFO)

Step	Stack (Top → Bottom)	Current Node Expanded	Visited Set After Step
1	A	A	{A}
2	B, C	B	{A, B}
3	D, E, C	D	{A, B, D}
4	E, C	E	{A, B, D, E}
5	G, C	G	{A, B, D, E, G}
6	C	C	{A, B, D, E, G, C}
7	F	F	{A, B, D, E, G, C, F}
8	– (empty)	– (DFS complete)	{A, B, D, E, G, C, F}



2.2.4 Bidirectional Search

Bidirectional Search is an uninformed search technique that conducts two simultaneous searches:
1. Forward search from the initial state, and
2. Backward search from the goal state.
The search continues until both searches meet in the middle, drastically reducing the total number of nodes explored.

Key Idea
Instead of searching the entire space from start to goal, Bidirectional Search splits the problem:
Initial State ←→ Goal State
↖ Meet at Middle ↗
Both frontiers expand toward each other, meeting at a common node.

Algorithm Steps
1. Initialize two queues → Q_1 for forward search, Q_2 for backward search
2. Begin BFS from both ends
3. Expand one level at a time
4. Check after each expansion whether the frontiers meet
5. Once they meet → path found

Example (Simple Graph)
Goal: Search from A to G
A — B — C — D — E — F — G
Forward Search: A → B → C
Backward Search: G → F → E
They meet near D, completing the path.
This is much faster than exploring the whole path from A to G.

Heuristic

A heuristic is an estimate or educated guess that helps an AI search algorithm decide which direction to explore first.
It does NOT have to be perfect or accurate — it only needs to be close enough to guide the search toward the goal.
In AI search (like A*, Greedy), heuristic is written as:
$h(n)$=estimated cost from node n to the goal
Heuristics make search faster and more goal-directed.

Real-Life Explanation
If you are standing at Golpark and want to go to Bhairahawa, there are two distances:

1. Actual Road Distance (Real Distance)
• Road route = 3 km
• This is the true cost, but you don't know it during search.

2. Straight-Line Distance (Bird-Fly Distance)
• You "see" Bhairahawa is roughly 1 km away if you draw a straight line.
• This is an estimate, not the real road distance.

This straight-line (1 km) is the heuristic.

2.2.5 Hill Climbing Search

Hill Climbing is an informed local search algorithm that tries to reach the best (highest value) state by repeatedly moving to a neighboring state that is better than the current state.
It resembles the idea of climbing a hill step by step until no higher point is available.
It is a variant of Greedy Search with no backtracking because it always chooses the immediate best move.

Key Idea

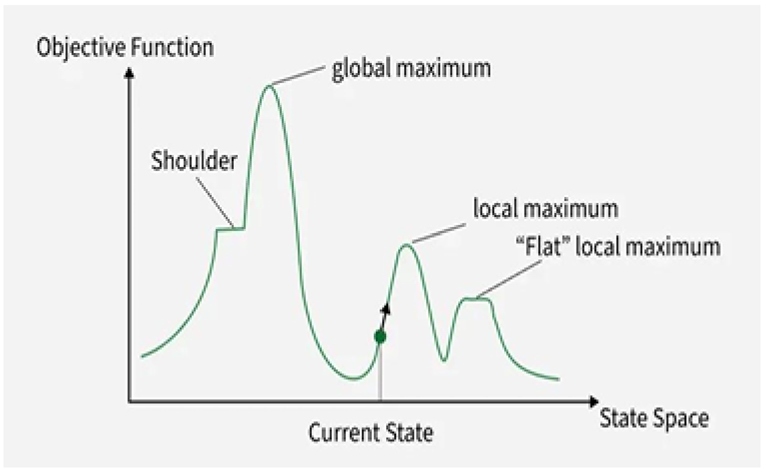
• Start with an initial solution
• Evaluate all neighbors
• Move to the neighbor with the highest value
• Repeat until no better neighbor exists

Hill climbing is simple but suffers from traps such as local maxima, ridges, and plateaus.

Types of Hill Climbing

1. Simple (Steepest-Ascent) Hill Climbing
o Looks at all neighbors and picks the best one.
2. Greedy (Best-First) Hill Climbing
o Evaluates neighbors one by one → takes the first improvement.
3. Random Restart Hill Climbing
o Restarts several times with different initial states → best final result.
4. Stochastic Hill Climbing
o Chooses a random better move instead of the best move.

Problems in Hill Climbing



Hill Climbing can get stuck in:

1. Local Maxima
A peak that is lower than the global maximum.
Algorithm stops thinking it reached the best point, but a better one exists.

2. Plateaus
A flat area with many equal-valued states → no direction to move → algorithm stops.

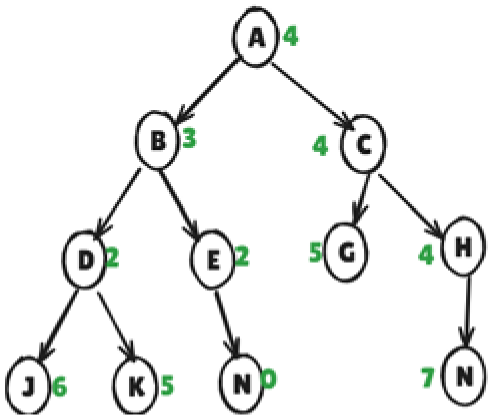
3. Ridges
A narrow path where the best direction is not aligned with the steepest direction → algorithm cannot climb effectively.

4. SHOULDER
A shoulder is a small rise before another descent.

Algorithm Steps
1. Start with a current state S
2. Loop:
o Evaluate neighbors of S
o If a neighbor is better, move to it
o If no better neighbor exists → STOP
3. Return current state (best found)

Questions:

Start = A and Goal = N
A(4) connects to B(3) and C(4)
B(3) connects to D(2) and E(2)
D(2) connects to J(6) and K(5)
E(2) connects to N(0)
C(4) connects to G(5) and H(4)
H(4) connects to N(7)



Use Simple Steepest-Ascent Hill Climbing (always move to the child with the lowest h, and stop when no child has lower h than the current node).

1. Starting from A, trace the steps of hill climbing and show the path followed.
2. Where does the algorithm stop? Does it reach the goal N?
3. Identify:
o the global optimum
o the local optimum where hill climbing gets stuck

Step 1 – Start at A

Current node: A (h = 4)
Children of A:
• B (h = 3)
• C (h = 4)

Best child (lowest h): B (3) (since 3 < 4)

Move: A → B

Step 2 – At node B

Current node: B (h = 3)

Children of B:
• D (h = 2)
• E (h = 2)
(Both are better than B)

Among children, best h = 2 (either D or E).

Assume we pick D (2) (tie broken alphabetically or arbitrarily).

Move: B → D
Current path so far:
$A \to B \to D$

Step 3 – At node D

Current node: D (h = 2)
Children of D:
• J (h = 6)
• K (h = 5)

Compare:
• h(J) = 6 > 2
• h(K) = 5 > 2

All neighbors are worse (higher h).
According to simple hill climbing:
If no child has lower h than current node, STOP.
So the algorithm stops at D.

Final Hill-Climbing Path
$A \to B \to D$

The algorithm does not continue to explore from E, and therefore it never reaches N.

Does It Reach the Goal N?

No.
• Hill climbing stops at D (h = 2) because:
o Both children J (6) and K (5) are worse than D.
• But there exists a better node N (h = 0) on another branch:

From B:
$B \to E \to N$
but hill climbing never goes back to try E after committing to D.

So this is exactly a case where hill climbing gets stuck.

2.2.6 Simulated Annealing Search

Simulated Annealing (SA) is a probabilistic local search algorithm used to find a good (near optimal) solution in large and complex search spaces.
It is an improvement over Hill Climbing because it can escape local maxima/minim a by sometimes accepting worse moves.
The name comes from annealing in metallurgy, where metal is heated and slowly cooled so atoms settle into a low-energy state.

Basic Idea

• Start with an initial state (a possible solution).
• At each step:
o Move to a neighboring state.
o If the new state is better, accept it.
4. the local optimum where hill climbing stops, and
5. the global optimum in this graph.
6. Briefly explain why hill climbing fails on this
graph using the idea of local vs global optimum.
o If the new state is worse, accept it with some probability that depends on:
▪ how much worse it is (ΔE)
▪ a temperature value T
When T is high → more randomness → more exploration.
When T gets low → behavior becomes like hill climbing (only improves).

Acceptance Rule
Let:
• *Ecurrent*= cost/value of current state
• *Enew*= cost/value of new state
• $\Delta E = Enew - Ecurrent$
Then:
• If $\Delta E < 0$→ new state is better → always accept
• If $\Delta E > 0$→ new state is worse → accept with probability:
P $= e{-}\Delta E/T$

As T → 0, this probability becomes very small → almost no worse moves accepted.

Cooling Schedule

The temperature T is gradually decreased:

$T_{new} = \alpha \cdot T_{old}, 0 < \alpha < 1$

- If α is close to 1 → very slow cooling, better solution but more time.
- If α is small → fast cooling, quicker but may miss good solutions.

Question

Hill Climbing that gets stuck in a Local Optimum

Start state = A, Goal state = M.

A(10) connects to B(7), C(9), D(8)

B(7) connects to E(4), F(5), G(6)

C(9) connects to O(9), P(7)

D(8) connects to Q(8)

E(4) connects to H(2), I(3)

F(5) connects to L(1), M(0)

G(6) connects to R(4)

H(2) connects to J(5), K(6)

I(3) connects to S(3)

J(5) has no children

K(6) has no children

L(1) connects to T(2)

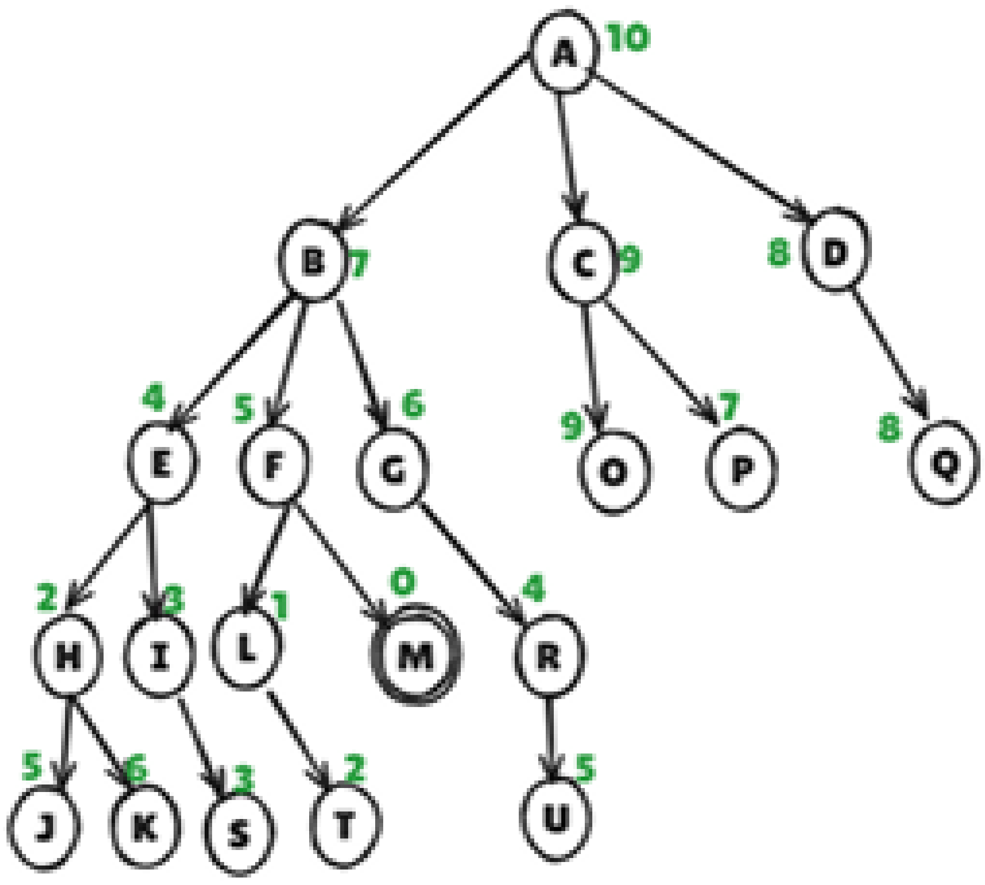
M(0) has no children (this is the goal)

O(9), P(7), Q(8) have no children

R(4) connects to U(5)

S(3), T(2), U(5) have no children

Solution



Graph (given in your question)

We are minimizing h(n) (lower h = better).

- Start state = A(10)
- Goal state = M(0)

Connections (undirected / neighbors):

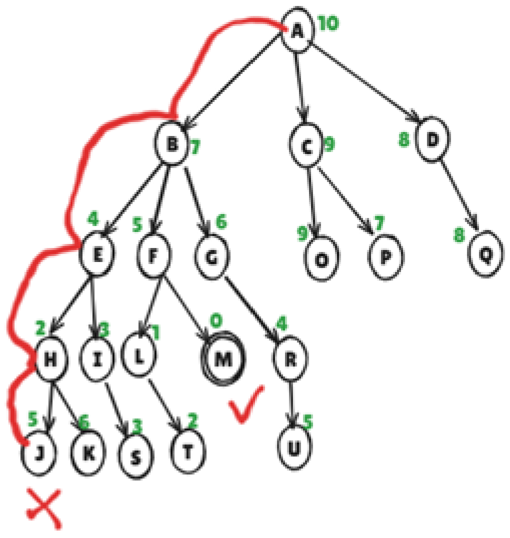
- A(10) → B(7), C(9), D(8)
- B(7) → A(10), E(4), F(5), G(6)
- C(9) → A(10), O(9), P(7)
- D(8) → A(10), Q(8)
- E(4) → B(7), H(2), I(3)
- F(5) → B(7), L(1), M(0)

• G(6) → B(7), R(4)
• H(2) → E(4), J(5), K(6)
• I(3) → E(4), S(3)
• J(5) → H(2)
• K(6) → H(2)
• L(1) → F(5), T(2)
• M(0) → F(5)
• O(9) → C(9)
• P(7) → C(9)
• Q(8) → D(8)
• R(4) → G(6), U(5)
• S(3) → I(3)
• T(2) → L(1)
• U(5) → R(4)

Hill Climbing on this graph (why it gets stuck)

Using steepest-ascent hill climbing (minimize h):

• Start at A(10)
o Best child = B(7) → move to B
• At B(7)
o Children: E(4), F(5), G(6) → best = E(4) → move to E
• At E(4)
o Children: H(2), I(3), B(7) → best = H(2) → move to H
• At H(2)
o Children: E(4), J(5), K(6) → all worse than 2



Hill climbing stops at H(2) (local optimum)
It never reaches M(0).

Now solve the SAME problem using Simulated Annealing

We’ll show one possible run of Simulated Annealing that does reach M(0).

Setup

• Objective: minimize h(n)
• Start at A(10)
• Temperature schedule (example):
o $T0=10$, then $T \leftarrow 0.8 \times T$after each move

• Neighbor = any adjacent node

• Acceptance rule:
o If $\Delta E = h\text{new} - h\text{current} \leq 0 \rightarrow$ always accept
o If $\Delta E > 0 \rightarrow$ accept with probability
$P = e - \Delta E/T$

We’ll show a lucky run where SA escapes from H(2) and eventually reaches M(0)

Step-by-step Simulated Annealing Run

• Curr = current node (h)
• Cand = candidate neighbor (h)
• ΔE = h(cand) – h(curr)
• T = temperature
• P = e^{-ΔE/T} if ΔE > 0

Step 0 – Start
• Curr = A(10)
• T = 10n

Choose neighbor randomly (say B(7)).
• ΔE = 7 − 10 = −3 (better) → accept
Move A → B

Step 1 – At B(7)
• Curr = B(7)
• T = 10 × 0.8 = 8

Neighbors: A(10), E(4), F(5), G(6)

Suppose candidate = E(4)

• $\Delta E = 4 - 7 = -3$ → better → accept
Move B → E

Step 2 – At E(4)
• Curr = E(4)
• T = 8 × 0.8 = 6.4

Neighbors: B(7), H(2), I(3)

Pick candidate H(2)
• $\Delta E = 2 - 4 = -2$ → better → accept
Move E → H

Now we're at the same local optimum H(2) where hill climbing got stuck.

Step 3 – Escape from local optimum (key SA behavior)
• Curr = H(2)
• T = 6.4 × 0.8 = 5.12

Neighbors: E(4), J(5), K(6)

Any move is worse than 2.

Suppose we randomly pick candidate E(4):
• $\Delta E = 4 - 2 = +2$ (worse)
• Acceptance probability:

$P = e^{-2/5.12} \approx e^{-0.39} \approx 0.68$

Pick a random number r = 0.5 → since r < 0.68 → ACCEPT.

SA accepts a worse move back to E(4) thanks to non-zero temperature.
Move H → E

This is exactly where hill climbing would never move, but SA does.

Step 4 – At E(4) again
• Curr = E(4)
• T = 5.12 × 0.8 ≈ 4.1

Neighbors: B(7), H(2), I(3)

Let's suppose this time candidate is F(5) via B (you can implement neighbor selection with a bit of randomness; for explanation we'll just jump to useful ones).

To keep it simple, imagine we pick F(5) as candidate from the neighborhood exploration (e.g., via E → B → F in implementation; for this conceptual run, treat F as a reachable neighbor we evaluate next).

• ΔE = 5 − 4 = +1 (worse)
• Acceptance probability:
$P = e^{-1/4.1} \approx e^{-0.24} \approx 0.79$

Random r = 0.3 < 0.79 → accept
Move → F(5)

Now at F(5), which hill climbing would never choose (worse than E), SA can still be here.

Step 5 – From F to a much better node

- Curr = F(5)
- T = 4.1 × 0.8 ≈ 3.28

Neighbors: B(7), L(1), M(0)

Pick candidate L(1):

- ΔE = 1 − 5 = −4 (better) → always accept

Move F → L

Step 6 – From L to the goal M

- Curr = L(1)
- T = 3.28 × 0.8 ≈ 2.62

Neighbors: F(5), T(2)

But we know M(0) is directly connected to F, so another move like:

L(1) → F(5) (maybe back), then F(5) → M(0)

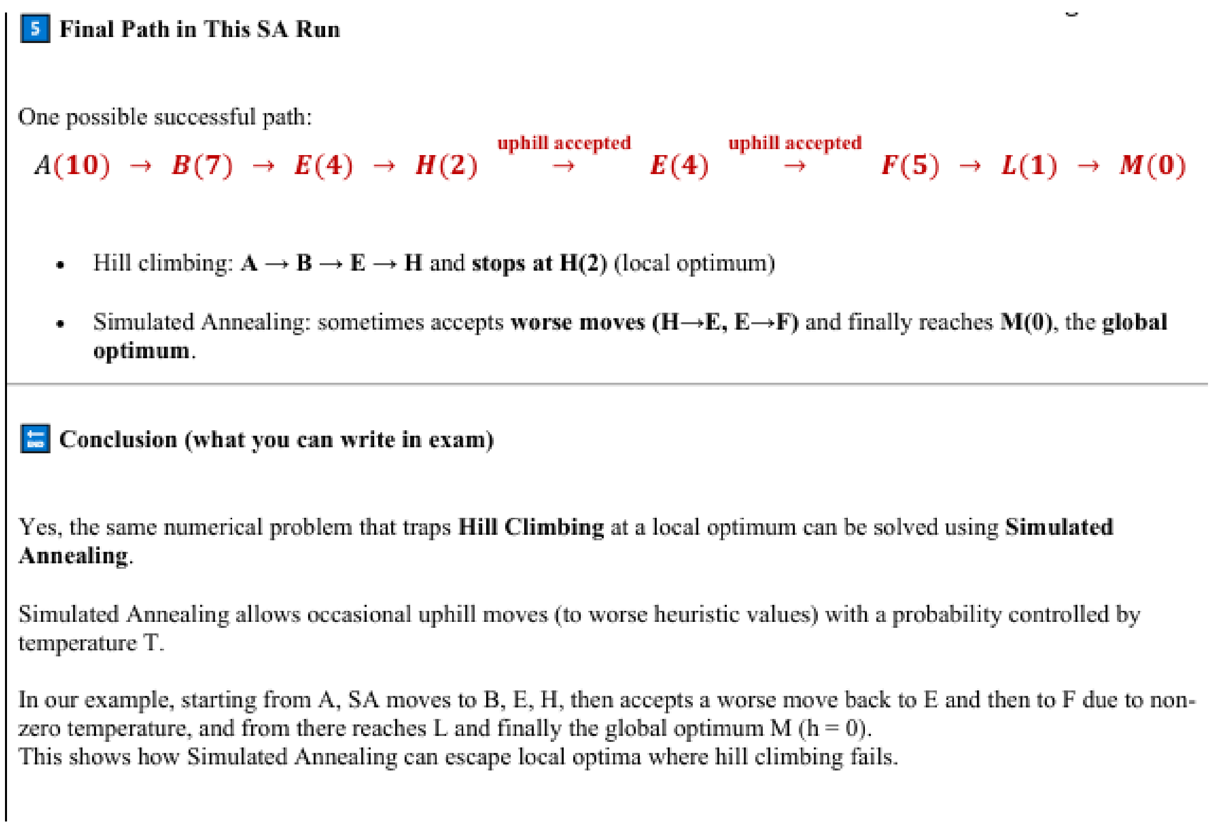
or directly choose F's neighbor M next.

Let's jump to F → M for simplicity:

From F(5) → pick M(0):

- ΔE = 0 − 5 = −5 → better → accept

Goal reached: M(0)



5️⃣ **Final Path in This SA Run**

One possible successful path:

$$A(10) \rightarrow B(7) \rightarrow E(4) \rightarrow H(2) \xrightarrow{\text{uphill accepted}} E(4) \xrightarrow{\text{uphill accepted}} F(5) \rightarrow L(1) \rightarrow M(0)$$

- Hill climbing: **A → B → E → H** and **stops at H(2)** (local optimum)
- Simulated Annealing: sometimes accepts **worse moves (H→E, E→F)** and finally reaches **M(0)**, the **global optimum**.

Conclusion (what you can write in exam)

Yes, the same numerical problem that traps **Hill Climbing** at a local optimum can be solved using **Simulated Annealing**.

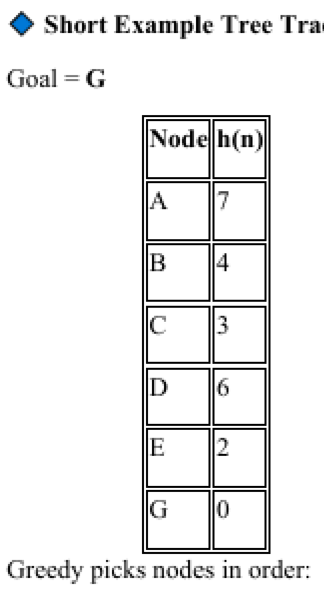
Simulated Annealing allows occasional uphill moves (to worse heuristic values) with a probability controlled by temperature T.

In our example, starting from A, SA moves to B, E, H, then accepts a worse move back to E and then to F due to non-zero temperature, and from there reaches L and finally the global optimum M (h = 0).
This shows how Simulated Annealing can escape local optima where hill climbing fails.

2.2.6 Greedy Search/Best First Search

Best-First Search (BFS) is an informed search strategy that selects and expands the most promising node first, based on a heuristic function h(n).
The goal is to reach the target state as quickly as possible by following the "bestlooking" path.
When Best-First Search uses only h(n) to choose the next node, it is specifically called: Greedy Best-First Search
Greedy Best-First Search: It is called greedy because it always chooses the node that appears nearest to the goal, without considering the total path cost.

Evaluation Function
Greedy Best-First Search uses:
Where: $f(n) = h(n)$
h(n) = heuristic estimate of the distance or cost from node n to the goal


◆ **Short Example Tree Trace**

Goal = **G**

Node	**h(n)**
A	7
B	4
C	3
D	6
E	2
G	0

Greedy picks nodes in order:

$A \rightarrow C \rightarrow E \rightarrow G$

Because it always picks the smallest h-value next.

2.2.7 A* Search

A* (A-star) is an informed search algorithm that finds the optimal (least-cost) path by combining:
 g(n): cost from the start node to node n
 h(n): heuristic estimate of cost from node n to the goal

A* selects the node with the minimum value of:
$f(n) = g(n)+h(n)$
• g(n) = cost so far
• h(n) = estimated cost to goal
• f(n) = estimated total cost of a path through n

A* is widely used in:
• GPS navigation
• Robotics
• Route planning
• Games (pathfinding)
• AI search problems

Key Concepts

1. $f(n) = g(n) + h(n)$

A* chooses the node with the lowest f(n) first.

2. Heuristic h(n)

A good heuristic should be:

- Admissible: $h(n) \leq$ true cost from n to goal
(never overestimates)
- Consistent (Monotonic):
$h(n) \leq c(n,m)+h(m)$

These conditions ensure optimality.

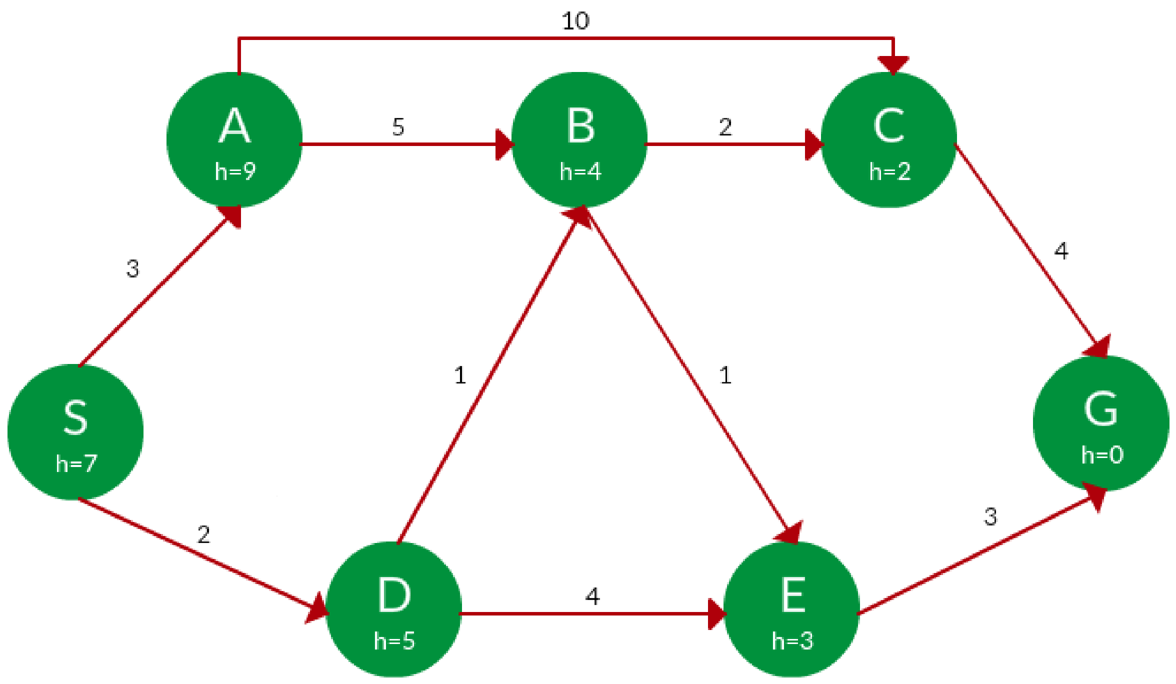
3. OPEN and CLOSED Lists

- OPEN: frontier nodes waiting to be expanded
- CLOSED: nodes already expanded

Algorithm Steps of A*

1. Insert start node into OPEN with f(start) = g(start) + h(start).
2. Repeat until goal reached or OPEN empty:
o Pick node with smallest f(n) from OPEN
o Move it to CLOSED
o If it is the goal → success
o Else expand node, compute f(n) for children
o If child is new or cheaper → update and push to OPEN
3. Return optimal path

For Example: Find the path to reach from S to G using A* search.



Starting from S the algorithm computes g(x) + h(x) for all nodes in the fringe at each step choosing the node with the lowest sum. The entire work is shown in the table below. It search like:

Path	h(x)	g(x)	f(x)
S	7	0	7
S -> A	9	3	12
S -> D ✓	5	2	7
S -> D -> B ✓	4	2 + 1 = 3	7
S -> D -> E	3	2 + 4 = 6	9
S -> D -> B -> C ✓	2	3 + 2 = 5	7
S -> D -> B -> E ✓	3	3 + 1 = 4	7
S -> D -> B -> C -> G	0	5 + 4 = 9	9
S -> D -> B -> E -> G ✓	0	4 + 3 = 7	7

- **Path:** S->D->B-> G-> E and Cost: 7

Question in Table Form Nodes: S (Start), B, C, D, E, F, G (Goal) Heuristic h(n) = estimated cost from node n to G.

From	To	Cost (step)
S	B	4
S	C	3
B	F	12
B	E	10
C	D	7
C	E	10
D	E	2
E	G	4
F	G	16

Node	h(n)
S	14
B	12
C	11
D	6
E	4
F	16
G	0

Task: Using A* with *f*(*n*)=*g*(*n*)+*h*(*n*), find the optimal path from S to G and its total cost. (Assume ties in f(n) are broken in favour of the node with smaller h(n).)

Step 0 – Initialization

Current	g	h	f	OPEN after step (node: f)	CLOSED
S	0	14	14	S:14	—

Step 1 – Expand S

Neighbors: B, C

- B: g=4, h=12 → f=16
- C: g=3, h=11 → f=14

Current expanded	New/updated nodes	g	h	f	OPEN (node: f)	CLOSED
S	B	4	12	16	B:16, C:14	{S}
	C	3	11	14		

Next chosen (min f): C (f=14)

Step 2 – Expand C

Neighbors: D, E, (S already closed)

- D: g = 3 + 7 = 10, h=6 → f=16
- E via C: g = 3 + 10 = 13, h=4 → f=17

Current expanded	New/updated nodes	g	h	f	OPEN (node: f)	CLOSED
C	D	10	6	16	B:16, D:16, E:17	{S, C}
	E (via C)	13	4	17		

Next chosen (min f, tie by smaller h): D (f=16, h=6) over B (f=16, h=12)

Step 3 – Expand D

Neighbors: E, (C closed)

- E via D: g = 10 + 2 = 12, h=4 → f=16
  (better than previous g(E)=13, so **update**)

Current expanded	New/updated nodes	g	h	f	OPEN (node: f)	CLOSED
D	E (updated)	12	4	16	B:16, E:16	{S, C, D}

Now OPEN has B(16,h=12) and E(16,h=4); **pick E** (smaller h).

Step 4 – Expand E

Neighbors: G, B, C, D (B open, others closed)

- G: g = 12 + 4 = 16, h=0 → f=16

Current expanded	New/updated nodes	g	h	f	OPEN (node: f)	CLOSED
E	G	16	0	16	B:16, G:16	{S, C, D, E}

Next, **G** is chosen (goal).

Step 5 – Goal

We stop when G is selected.
Backtracking parents (S→C→D→E→G) gives:
Optimal path: $S \rightarrow C \rightarrow D \rightarrow E \rightarrow G$

Total cost:

$g(G) = 3+7+2+4= 16$

This matches the whiteboard result $S > C > D > E > G$ with total cost 16.

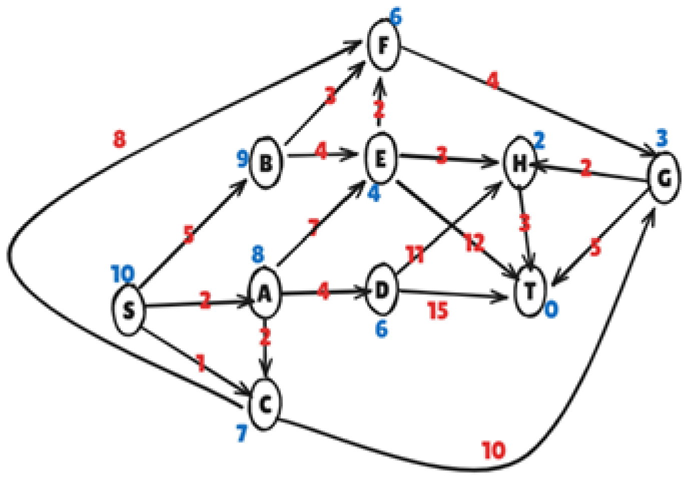
Question

A delivery van starts at S and wants the minimum-cost path to city T. Edges are road costs. The graph is: Edges (undirected, cost in brackets):

- S–A(2), S–B(5), S–C(1)
- A–C(2), A–D(4), A–E(7)
- B–E(4), B–F(3)
- C–F(8), C–G(10)
- D–H(11), D–T(15)
- E–H(3), E–T(12), E–F(2)
- F–G(4)
- G–H(2), G–T(5)
- H–T(3)

Heuristic h(n) = estimated distance to T:

Node	h(n)
S	10
A	8
B	9
C	7
D	6
E	4
F	6
G	3
H	2
T	0



Use A* with

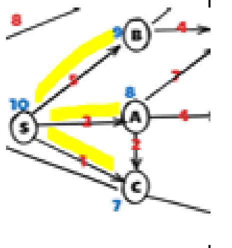
$f(n) = g(n)+h(n)$

where g(n) is path cost from S.

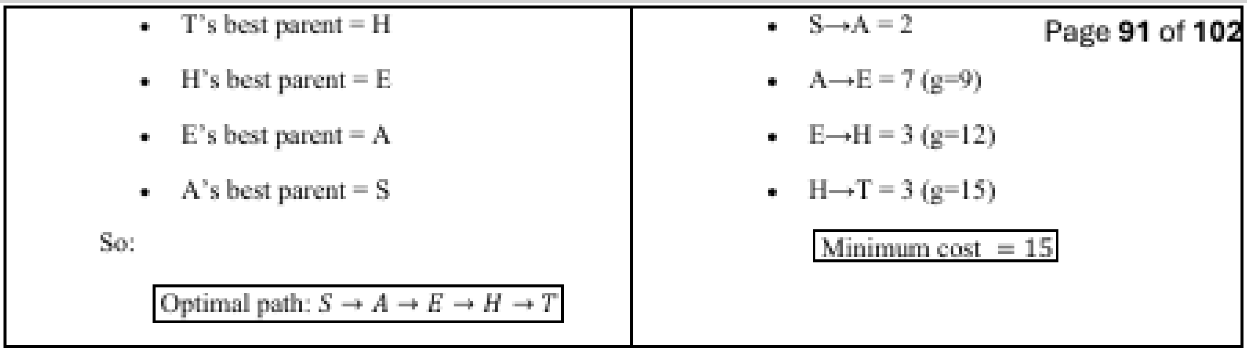
Find the optimal path S → T and its cost. Show the main A* steps (OPEN/CLOSED).

✅ Solution 1 – A* on Route Map

We'll track: **node, g, h, f**, and OPEN/CLOSED lists.
Start: **S**

Step 0 – Initialize • g(S) = 0, h(S) = 10 → f(S) = 0+10=10 • OPEN = { S }, CLOSED = { }	**Step 1 – Expand S**  8 9 B 4 5 7 10 S 8 A 2 4 2 1 C 7 Neighbors: • A: g = 0+2 = 2, f = 2+8 = 10 • B: g = 0+5= 5, f = 5+9 = 14 • C: g = 0+1= 1, f = 1+7 = 8 OPEN = { A(2,8,10), B(5,9,14), C(1,7,8) } CLOSED = { S } Pick lowest f → **C** (f=8).
Step 2 – Expand C Neighbors: S (closed), A, F, G • A via C: g = 1+2 = 3 → worse than existing g(A)=2 → ignore • F: g = 1+8 = 9, h=6 → f=15 • G: g = 1+10 = 11, h=3 → f=14 OPEN = { A(2,8,10), B(5,9,14), F(9,6,15), G(11,3,14) } CLOSED = { S, C } Pick lowest f → **A** (f=10).	**Step 3 – Expand A** Neighbors: S, C, D, E • D: g = 2+4 = 6, h=6 → f=12 • E: g = 2+7 = 9, h=4 → f=13 OPEN = { B(5,9,14), F(9,6,15), G(11,3,14), D(6,6,12), E(9,4,13) } CLOSED = { S, C, A } Pick lowest f → **D** (f=12).
Step 4 – Expand D Neighbors: A (closed), H, T • H: g = 6+11 = 17, h=2 → f=19 • T via D: g = 6+15 = 21, h=0 → f=21 OPEN = { B(5,9,14), F(9,6,15), G(11,3,14), E(9,4,13), H(17,2,19), T(21,0,21) } CLOSED = { S, C, A, D }	**Step 5 – Expand E** Neighbors: A (closed), B, H, T, F • B via E: g = 9+4 = 13 (worse than existing g(B)=5) → ignore • H via E: g = 9+3 = 12, h=2 → f=14 (**better than old H(17,2,19)** → update)

Lowest f → **E** (f=13).	• T via E: g = 9+12 = 21 (same as existing 21) • F via E: g = 9+2 = 11 (worse than g(F)=9) → ignore OPEN = { B(5,9,14), G(11,3,14), F(9,6,15), H(12,2,14), T(21,0,21) } CLOSED = { S, C, A, D, E } Three nodes have f=14: B, G, H. We can pick any; choose **B** first.
Step 6 – Expand B Neighbors: S (closed), E (closed), F • F via B: g = 5+3 = 8, h=6 → f = 14 **(better than old F(9,6,15)** → update) OPEN = { G(11,3,14), H(12,2,14), F(8,6,14), T(21,0,21) } CLOSED = { S, C, A, D, E, B } Pick any f=14; choose **F**.	**Step 7 – Expand F** Neighbors: B (closed), E (closed), G • G via F: g = 8+4 = 12 (worse than existing g(G)=11) → ignore OPEN = { G(11,3,14), H(12,2,14), T(21,0,21) } CLOSED = { S, C, A, D, E, B, F } Pick **G** (f=14).
Step 8 – Expand G Neighbors: C (closed), F (closed), H, T • H via G: g = 11+2 = 13 (> current g(H)=12) → ignore • T via G: g = 11+5 = 16, h=0 → f = 16 (**better** than old T(21)) → update OPEN = { H(12,2,14), T(16,0,16) } CLOSED = { S, C, A, D, E, B, F, G } Pick **H** (f=14).	**Step 9 – Expand H** Neighbors: D (closed), E (closed), G (closed), T • T via H: g = 12+3 = **15**, h=0 → f = 15 (**better** than 16) → update OPEN = { T(15,0,15) } CLOSED = { S, C, A, D, E, B, F, G, H } Next node is **T** (goal).
Step 10 – Goal • Expand **T**, goal reached. • g(T) = **15** is the optimal cost (because A* with admissible heuristic is optimal).	
We now reconstruct the path using parents:	Total cost:



Page 91 of 102

• T's best parent = H • H's best parent = E • E's best parent = A • A's best parent = S So: $\boxed{\text{Optimal path: } S \to A \to E \to H \to T}$	• S→A = 2 • A→E = 7 (g=9) • E→H = 3 (g=12) • H→T = 3 (g=15) $\boxed{\text{Minimum cost } = 15}$

2.3 Min-Max Algorithm

The Min-Max algorithm is a decision-making algorithm used in two-player games (like chess, tic-tac-toe).
It assumes both players play optimally:
Maximizer tries to maximize the score.
Minimizer tries to minimize the score.

Steps
Generate the game tree up to a certain depth.
Evaluate terminal states using a heuristic function (e.g., win = +1, loss = -1, draw = 0).
Propagate values upward:
At Max nodes, choose the maximum child value.
At Min nodes, choose the minimum child value.
The root node's value determines the best move.

Example (Tic-Tac-Toe)
If it's your turn (Max), you look at all possible moves.
For each move, simulate the opponent's response (Min).
Choose the move that maximizes your minimum guaranteed outcome.

```
        MAX
      /     \
    MIN      MIN
   /  \    /   \
  3     5  2     9
```

- Left MIN → min(3,5) = 3
- Right MIN → min(2,9) = 2
- MAX → max(3,2) = **3**

✅ Best move gives value **3**

2.4 Alpha-Beta Pruning (Cutoff)

Alpha-Beta pruning is an optimization of Min-Max that reduces the number of nodes evaluated.
It cuts off branches that cannot possibly affect the final decision.
This reduces the number of nodes evaluated, making the algorithm faster.

Key Terms
Alpha (α): Best value the Maximizer can guarantee so far.
Beta (β): Best value the Minimizer can guarantee so far.
If at any point α ≥ β, further exploration of that branch is unnecessary (pruned).

Steps
Start with α = -∞, β = +∞.
Traverse the game tree like Min-Max.
Update α and β as you go:
At Max nodes → update α.
At Min nodes → update β.
Prune when α ≥ β.

Benefits
Same result as Min-Max, but with fewer computations.
Makes deep searches feasible in complex games like chess.

Gpt:
Pruning rule

If $\alpha \geq \beta$ → prune (cutoff)

No need to explore further because the opponent won't allow it

Why it works

Produces the same result as Min–Max

Skips unnecessary calculations

Example intuition

If MIN already has a move that gives value ≤ 2, and MAX has an alternative that guarantees ≥ 5, MIN will never allow the worse option → prune that branch