Unit II

Approaches of AI

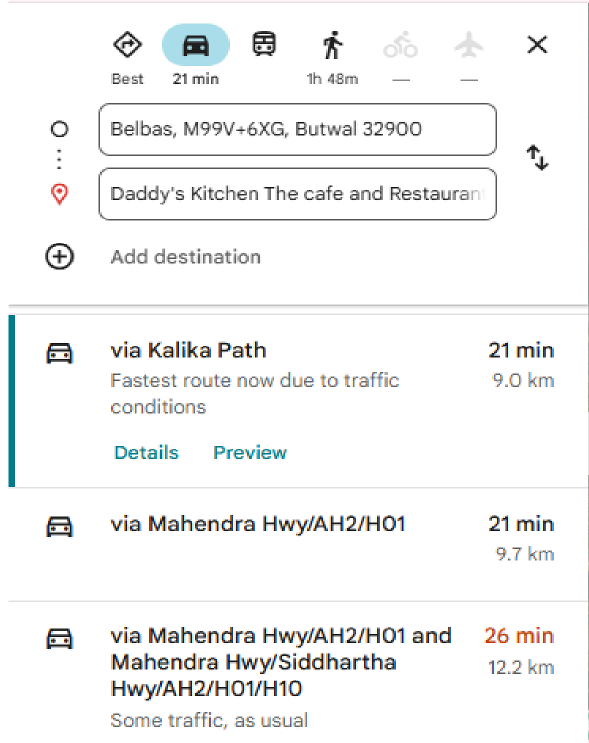
Sanjeev Thapa, Er. DevOps, SRE, CKA, RHCSA, RHCE, RHCSA-Openstack, MTCNA, MTCTCE, UBSRS, HEv6, Research Evangelist

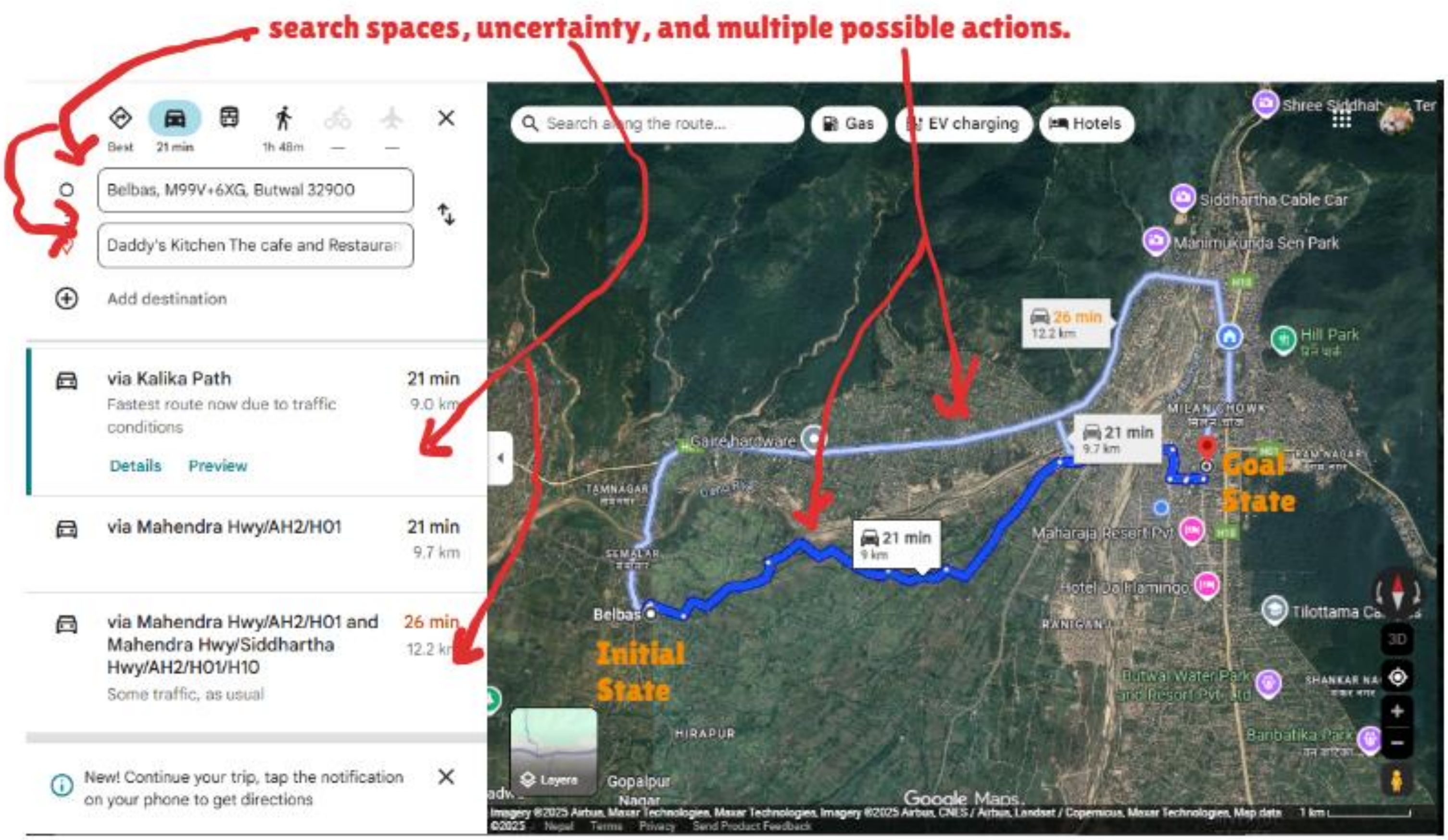
2.0 Introduction

- Artificial Intelligence (AI) deals with the study and design of **intelligent agents** that can **perceive**, **reason**, and **act** to achieve goals.
- AI approaches provide systematic methods for solving problems, searching for solutions, and making rational decisions in complex environments.
- AI problems often involve large search spaces, uncertainty, and multiple possible actions.
- Therefore, AI uses structured approaches such as **problem formulation**, **state space search**, **heuristic methods**, and **game-playing techniques** to find optimal or near-optimal solutions.

Real-Life Example:

When you use **Google Maps**, the system observes your location, evaluates many possible routes, predicts traffic conditions, and selects the **best and shortest path**. This is AI problem-solving in action.



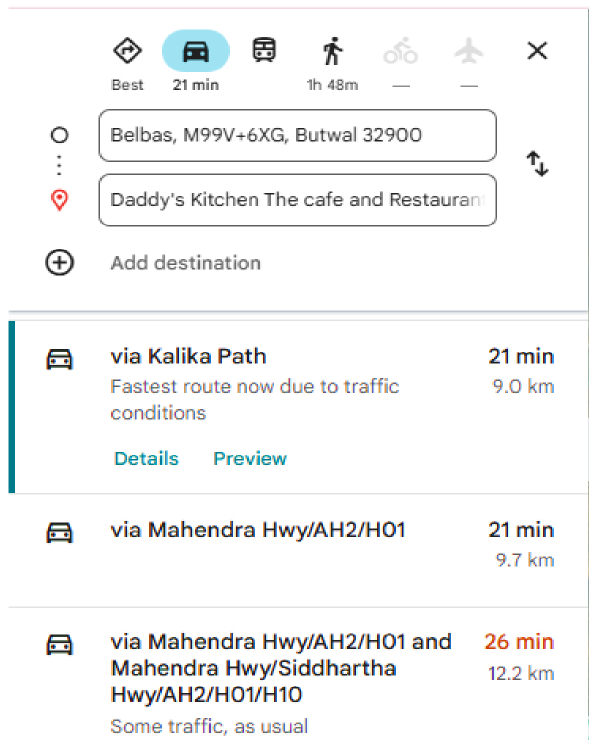


2.1 Problem Solving

- Problem solving in Artificial Intelligence refers to the process where an intelligent agent **moves from an initial state to a goal state** by applying a sequence of valid actions.
- An AI system must identify the problem, understand the steps required, explore possible solutions, and select the **most efficient path**.
- A problem is solved when the agent finds a sequence of actions that **transforms** the current state into the desired goal state.
- To achieve this, AI uses structured tools such as **state space representation**, **search strategies**, **heuristics**, and **production rules**.

Key Ideas of AI Problem Solving

- Clear understanding of **start and goal conditions**
- Logical representation of actions
- Exploring alternative solutions
- Selecting the best or optimal action sequence
- Handling complexity, uncertainty, or constraints



Real-Life Example (Simple & Clear):

A **delivery app (Foodmandu / Pathao Food)** solving “shortest delivery route” is performing problem solving.

- **Initial State**: Restaurant location
- **Goal State:** Customer location
- **Actions:** Possible roads/turns
- **Search:** Explore shortest path
- **Solution:** Fastest route provided to rider

2.1.1 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**

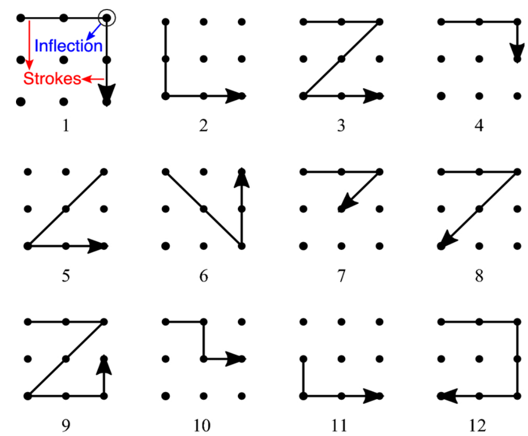
   The set of all possible states reachable from the initial state.

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

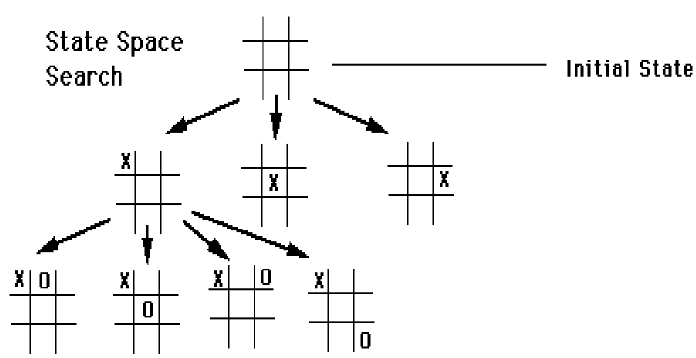
- **Initial State:** Locked screen
- **Goal State:** Enter correct unlock pattern
- **Operators:** Swipe up/down/diagonal to connect dots
- **State Space:** All possible patterns connecting the dots
- **Path Cost:** Minimum number of moves (optional)



2.1.2 State Space Search with Examples

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:

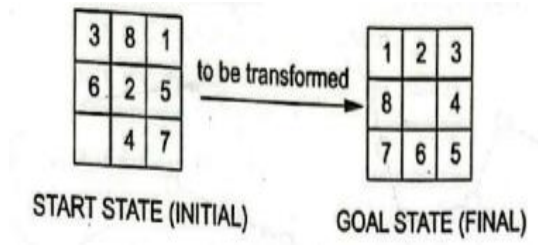
- **Nodes** = States
- **Edges** = Actions
- **Path** = Sequence of actions
- **Goal Test** = Checks if the goal state is reached



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.

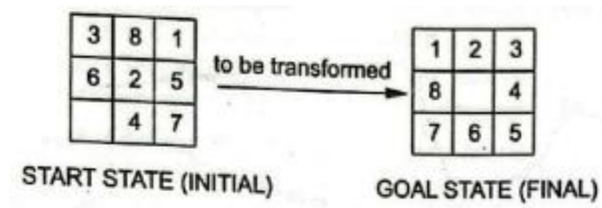


Examples of State Space Search

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 _

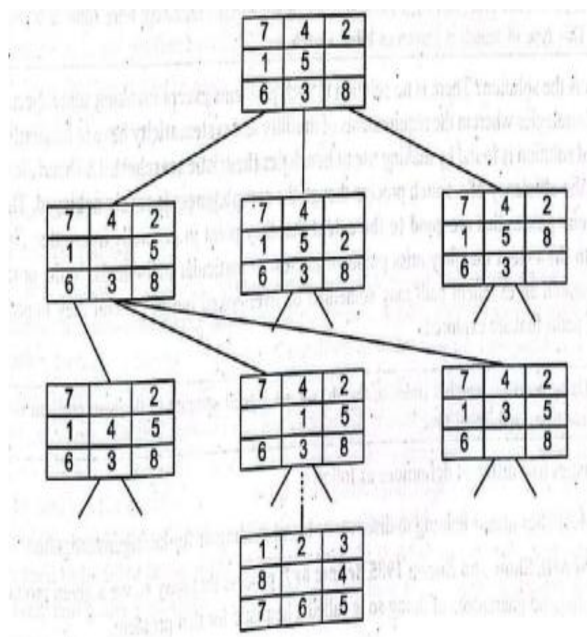
But the goal can be changed depending on the problem.

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)**:

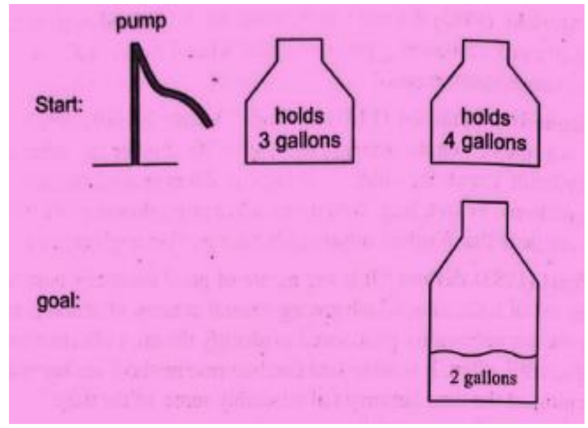
```
Initial State
   |
   |— First Move → New State
   |
   |— Second Move → Another State
   |
   … and so on
```

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**
   - Fill 4-liter jug → (4, y)
   - Fill 3-liter jug → (x, 3)
2. **Empty a jug**
   - Empty 4-liter jug → (0, y)
   - Empty 3-liter jug → (x, 0)
3. **Pour water from one jug to another**
   - Pour from 4L to 3L
   - Pour from 3L to 4L
   - 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.

The **operators** to be used to solve the problem is shown in table-1 below—

Table 1: Production rules (or operators) for the water jug problem

1. $(x, y) \rightarrow (4, y)$ if $x < 4$	Fill the 4-gallon jug
2. $(x, y) \rightarrow (x, 3)$ if $y < 3$	Fill the 3-gallon jug
3. $(x, y) \rightarrow (x - d, y)$ if $x > 0$	Pour some water out of the 4-gallon jug
4. $(x, y) \rightarrow (x, y - d)$ if $y > 0$	Pour some water out of 3-gallon jug
5. $(x, y) \rightarrow (0, y)$ if $x > 0$	Empty the 4-gallon jug on the ground
6. $(x, y) \rightarrow (x, 0)$ if $y > 0$	Empty the 3-gallon jug on the ground
7. $(x, y) \rightarrow (4, y - (4 - x))$ if $x + y \geq 4$ and $y > 0$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full

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: Route Finding (Real-Life Example)

Route finding is a practical and commonly used example of **state space search** in Artificial Intelligence. Modern navigation systems such as **Google Maps, Apple Maps, or Pathao Navigation** use AI algorithms to determine the **best possible route** from a starting location to a destination.

🔷 1. States (Representation)

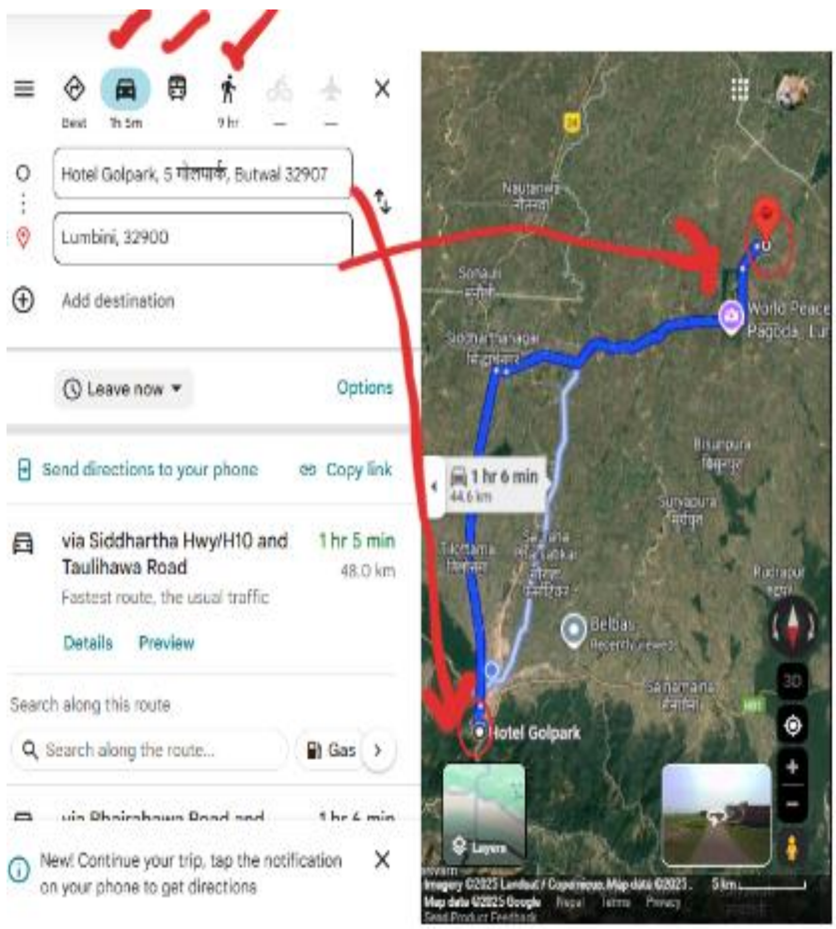
A **state** represents a geographical location or road intersection.

Examples:

- Butwal
- Bhairahawa
- Lumbini
- Kalikanagar
- Golpark

In AI representation:

State = Current city/intersection



🔷 2. Initial State

The user's **current location**.

Example:

Initial State = Butwal (Golpark)

🔷 3. Goal State

The user's **destination**.

Example:

Goal State = Lumbini

🔷 4. Actions

Actions represent possible movements between roads:

- Turn left
- Turn right

- Go straight
- Take highway
- Follow alternate road

Each action moves the agent to a **new road segment**, forming a new state.

◆ 5. State Space

The state space consists of all the **roads, intersections, and possible paths** between the source and destination.

This forms a **graph**, where:

- **Nodes** = Cities or intersections
- **Edges** = Roads connecting them

◆ 6. Path Cost

To choose the best route, AI uses different cost measures:

- **Distance** (km)
- **Time** (minutes)
- **Traffic level**
- **Fuel consumption**
- **Road quality**

Most navigation apps use **time** as the main path cost.

◆ 7. Search Algorithm Used

Navigation systems typically use:

A Search Algorithm*

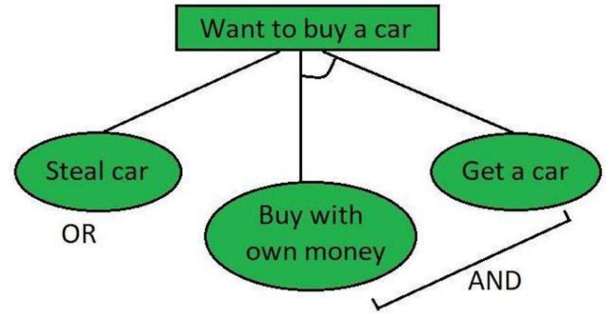
$$f(n) = g(n) + h(n)$$

- **g(n)** = path cost so far
- **h(n)** = estimated remaining distance (heuristic)

A* finds the **shortest and fastest** route efficiently.

2.1.3 Problem Reduction

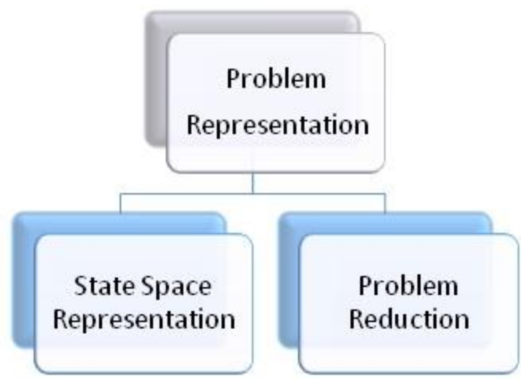
- Problem Reduction is an approach in Artificial Intelligence where a **complex problem** is broken down into a set of **smaller, simpler sub-problems**.



- 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.

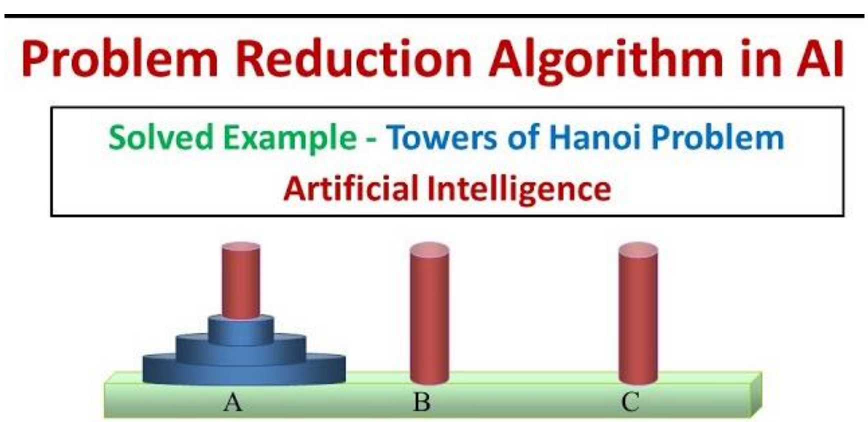
Problem Reduction = breaking a big problem → solving smaller parts → combining results → achieving the final solution.

- 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.
- https://www.slideserve.com/niles/missionary-cannibal

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)

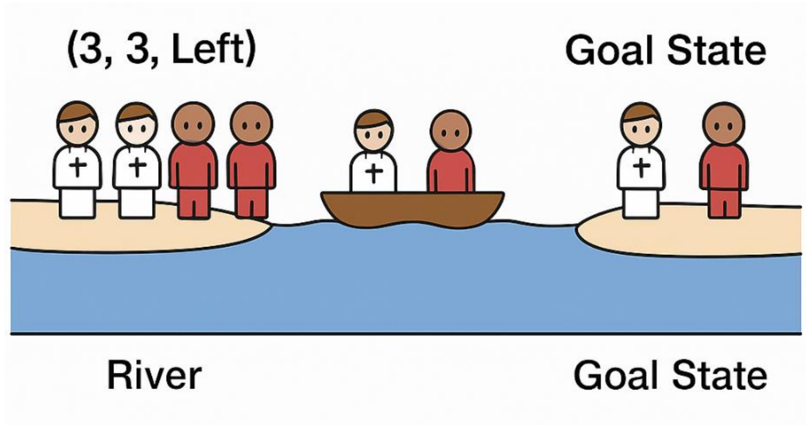
Goal State

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.

🔷 7. Why This Problem Is Important in AI?

- Demonstrates **problem reduction**
- Shows **state-space representation**

- Teaches **safe vs unsafe states**
- Useful for **search algorithms** **(BFS, DFS, A*)**

🔷 Real-Life Example (Simple & Clear)

Planning a Trip

A long journey (e.g., Butwal → Pokhara → Kathmandu) is reduced into smaller steps:

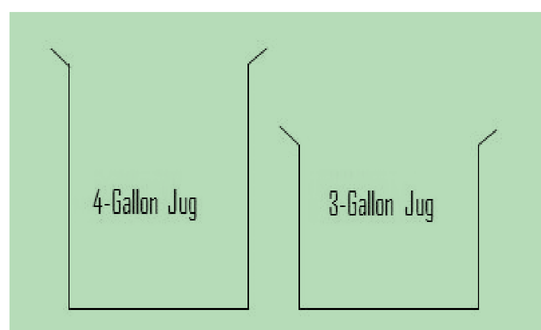
1. Plan bus to Pokhara
2. Plan hotel stay
3. Plan travel to Kathmandu
4. Plan sightseeing

Each sub-task is solved separately but contributes to the final travel plan.

🔷 Problem Reduction Tree (Conceptual)

```
Main Problem
  /      |            \
Sub1    Sub2     Sub3
 |          \       |
Solved       Solved |
        Combined Solution
```

2.1.4 Production Systems (with BFS & DFS Focus)



- 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.

Example: Water Jug Production System	
This classic problem demonstrates how **production rules** create a **state space** that can be explored using **BFS or DFS**.	
Problem Setup	A “state” is written as:

You have: **4-liter jug** **3-liter jug** **Goal:** Measure **exactly 2 liters** of water.	(x, y) x = water in 4-liter jug y = water in 3-liter jug
1. Initial State (0, 0) Both jugs are empty.	**2. Goal State** Any state where the 4-liter jug has **2 liters**, such as: (2, y)
3. Production Rules (IF → THEN Actions) These rules generate new states	
Rule 1: Fill a jug IF 4L jug not full → THEN fill 4L IF 3L jug not full → THEN fill 3L Example: (0,0) → (4,0) (0,0) → (0,3)	**Rule 2: Empty a jug** IF 4L jug has water → THEN empty it IF 3L jug has water → THEN empty it Example: (4,0) → (0,0)
Rule 3: Pour water from one jug to another Pour 4L → 3L Pour 3L → 4L Example: (4,0) → (1,3) (3L jug becomes full, 4L keeps 1 liter) Each rule transforms the current state into a **new legal state**.	
4. Water Jug State Space (Conceptual Sequence) A possible solution path: (0,0) → (4,0) [Fill 4L]	

→ (1,3) [Pour 4L into 3L] → (1,0) [Empty 3L] → (0,1) [Pour 1L into 3L] → (4,1) [Fill 4L] → (2,3) [Pour 4L into 3L] ✓ Goal achieved: (2,3) Each arrow represents a **production rule** being applied.	
5. How BFS & DFS Apply	
◆ **BFS (Breadth-First Search)** Explores all possible states **level-by-level** Finds **shortest sequence of steps** Best when you want minimum moves Example: It would find the **shortest path** to reach (2,y).	◆ **DFS (Depth-First Search)** Follows one sequence deeply May find long or non-optimal solutions Uses less memory Useful for exploring possible rule sequences
6. Why This Example Is Important in AI • Shows how **rules generate new states** • Demonstrates **state space search** • BFS & DFS act as **control strategies** • Represents a real problem solvable via logic + search • Teaches **problem specification** and **state transitions**	

◆ 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.

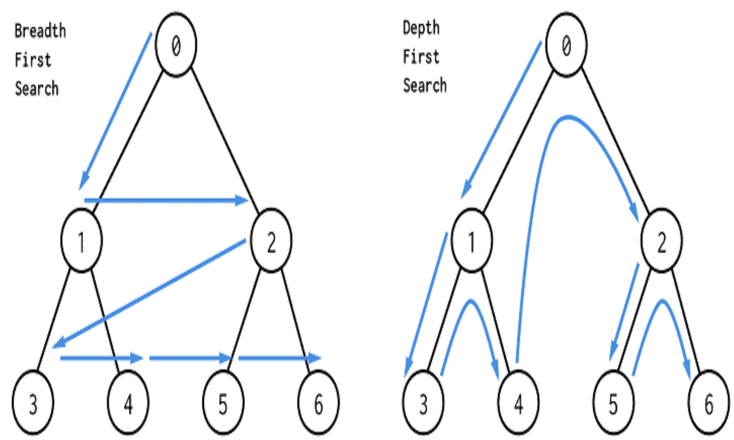
State Space Representation

- **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.

◆ Central Role of BFS and DFS

Production systems NEED a method to explore the state space.
The two fundamental methods are:



1. Breadth-First Search (BFS)

- Expands all nodes at the current level before moving deeper
- Guarantees **shortest path**
- Uses **Queue (FIFO)**
- Suitable when path cost or optimality matters

Example in 8-Puzzle:
BFS finds the minimum number of moves to reach the goal configuration.

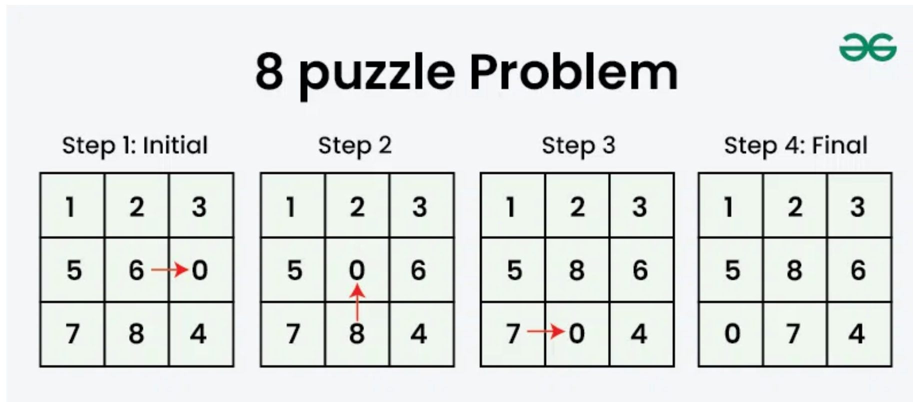
2. Depth-First Search (DFS)

- Explores one path as deep as possible
- Uses **Stack (LIFO)**
- Memory efficient
- Not guaranteed to find shortest path
- Suitable for deep or complex rule chains

Example:
Trying multiple rule sequences in a logic puzzle.

🔷 How Production Rules Generate Search Space

Consider the 8-Puzzle:



Production Rules

- Move blank left
- Move blank right
- Move blank up
- Move blank down

Applying these rules expands new states:

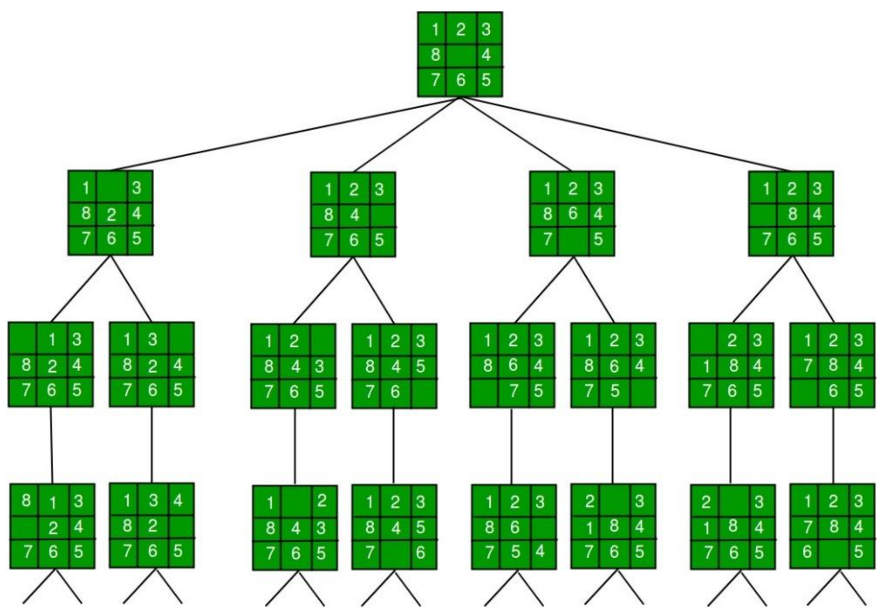
Initial State

/ \

Apply R1 Apply R2

| |

State A State B



The agent then uses BFS or DFS to explore the state space toward the goal.

IF <antecedent–1>
<antecedent–2>
⋮
<antecedent–n>
THEN
<consequence 1>|(with certainty C_1)
<consequence 2>|(with certainty C_2)
⋮
<consequence n>|(with certainty C_n)

🔷 Limitations

- Rule explosion (too many rules)
- Can be slow without heuristics
- Needs good conflict resolution

2.2 Searching Techniques

🔷 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.

🔷 Why Searching Is Important in AI?

- Many AI problems can be represented as a **search tree**
- AI needs systematic ways to explore all possibilities
- Search helps AI find not just *any* solution but often the **optimal** one
- Used in planning, robot navigation, route finding, puzzle solving, game-playing, etc.

➢ Searching is one of the most important approaches in Artificial Intelligence.

➢ AI uses **search techniques** to explore possible states of a problem until a **goal state** is found.

➢ A search algorithm systematically examines **states**, **actions**, and **paths** to find the **best or optimal solution**.

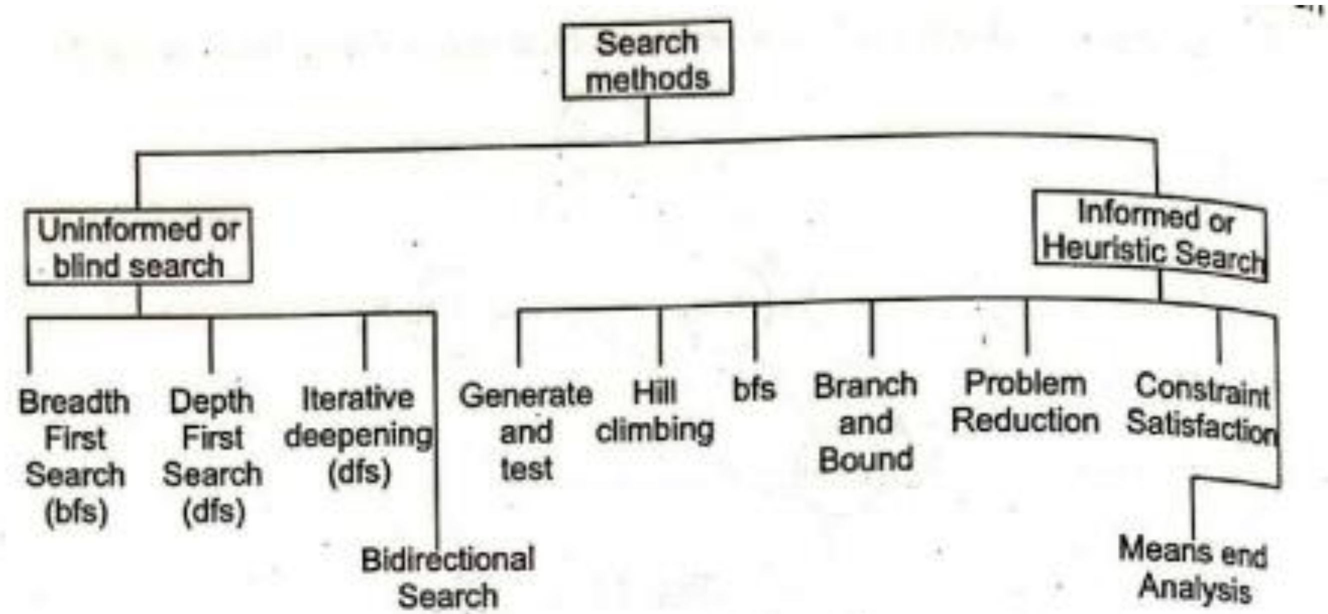
➢ An AI agent performs search when:

- It does not know the solution directly
- Multiple possible choices (branches) exist
- It must choose the best sequence of actions

Search techniques are broadly classified into:

1. **Uninformed (Blind) Search**
2. **Informed (Heuristic) Search**

2.2.1 Types of Searching



Type	Uses Heuristic?	Speed	Optimality	Examples
Uninformed Search	❌ No	Slow	Sometimes	BFS, DFS, UCS
Informed Search	✓ Yes	Fast	If h(n) good	A*, Greedy, Hill Climbing

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.

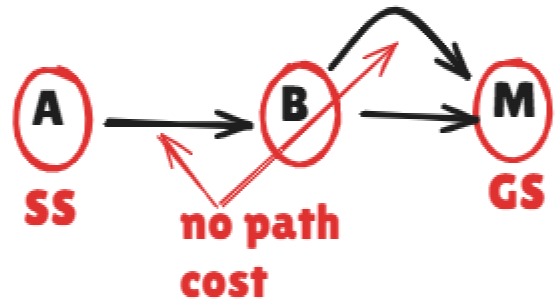
🔷 **1. Uninformed (Blind) Search**

Uninformed search methods **do not use any additional knowledge** about the problem beyond what is provided in the problem definition.

They explore the search space **blindly**, without estimating how close they are to the goal.

Characteristics

- No heuristic guidance
- Exhaustive exploration
- Guaranteed to find solution if one exists (in some methods)
- Often slow and memory-intensive



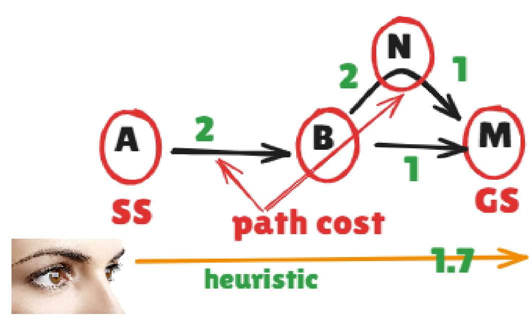
Common Uninformed Search Methods

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**

Real-Life Analogy:
Searching for a name in a phone contact list **without knowing the first letter** — checking one by one.

🔷 2. Informed (Heuristic) Search

Informed search methods use **heuristics** to estimate how close the current state is to the goal. This makes the search **faster**, **smarter**, and more **goal-directed**.



Characteristics

- Guided by heuristic function **h(n)**
- Reduces search time
- More efficient for large state spaces
- Can be optimal if heuristic is admissible

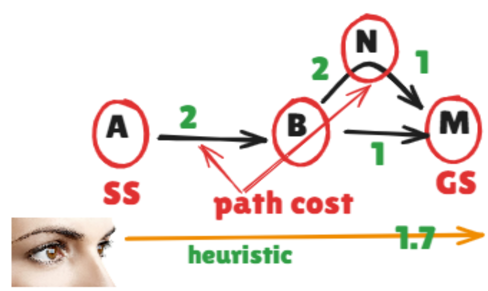
Common Informed Search Methods

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

Real-Life Analogy:
Using **Google Maps**, which estimates distance and traffic to guide you to the best route.

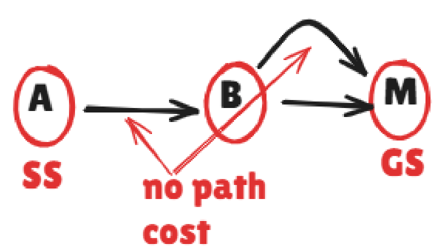
2.2.2 Uninformed/Blind/Brute

- Uninformed search methods are those that **do not use any heuristic or additional knowledge** about the problem domain.
- They explore the search space **blindly**, considering only the information provided in the **problem definition** (initial state, actions, goal test).
- Because they lack guidance, these techniques may explore many unnecessary paths — but they are **complete** and often **guarantee a solution** if one exists.



🔷 **Characteristics of Uninformed Search**

- No heuristic function (**h(n)**)
- Search without knowledge of "how close" a state is to the goal
- Systematic and simple
- May be slow or memory-heavy
- Some methods guarantee optimality



🔷 **Common Uninformed Search Methods**

1. Breadth-First Search (BFS)

- Explores all nodes at the same depth first
- Uses a **Queue (FIFO)**
- **Complete & Optimal** for equal step costs
- Expensive in memory

2. Depth-First Search (DFS)

- Explores a path deeply before backtracking
- Uses a **Stack (LIFO)**
- Low memory requirement
- Not always optimal

3. Depth-Limited Search (DLS)

- DFS with a depth cutoff
- Prevents infinite loops

4. Iterative Deepening Search (IDS)

- Repeatedly applies DLS with increasing depth
- Combines low memory (DFS) + optimality (BFS)

5. Uniform Cost Search (UCS)

- Expands the **least-cost** node
- Uses priority queue
- **Optimal** when step costs vary

6. Bidirectional Search

- Two simultaneous searches:
    - from initial state
    - from goal state
- Meets in the middle → reduces complexity

🔷 **Why Called Blind / Brute-Force?**

Because:

- They "blindly" explore the search tree
- They do not estimate the quality of states
- They may explore **every possible node** before finding the goal

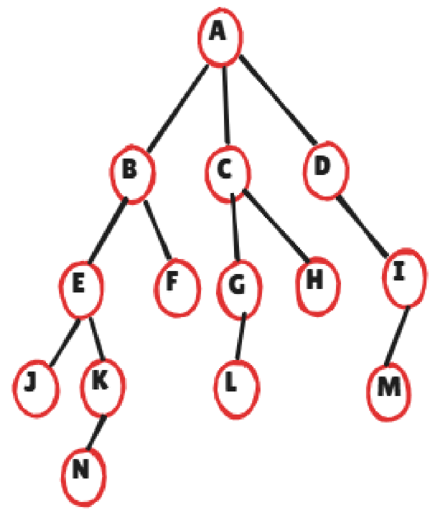
Example:
Trying all possible keys on a keychain one by one without knowing which key fits.

🔷 **Real-Life Example (Simple)**

Searching for a name in a phonebook without knowing the alphabetical section:

- You check each name one by one → similar to BFS/DFS.

2.2.2.1 Breadth-First Search (BFS)



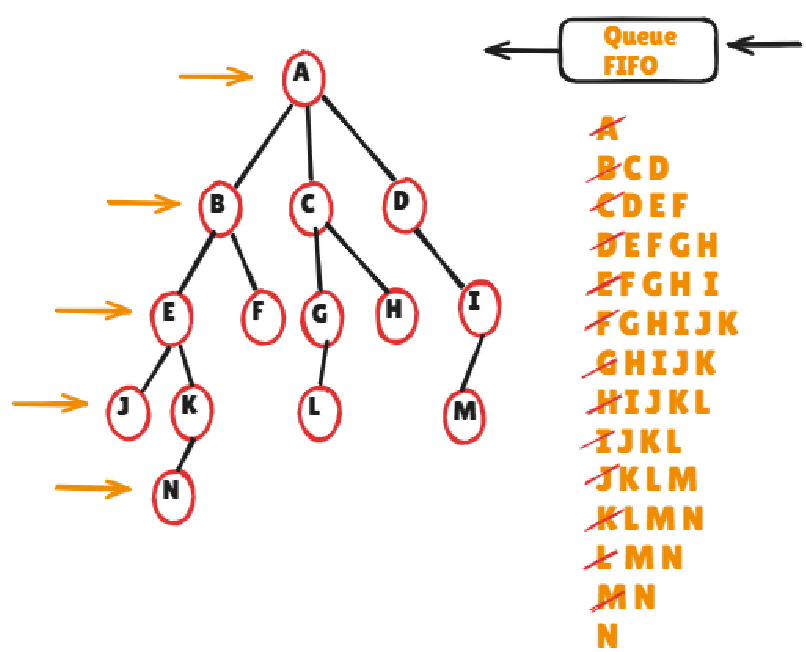
➢ Breadth-First Search (BFS) is an **uninformed search technique** that explores the search space **level by level**.

➢ 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 Working Principle**

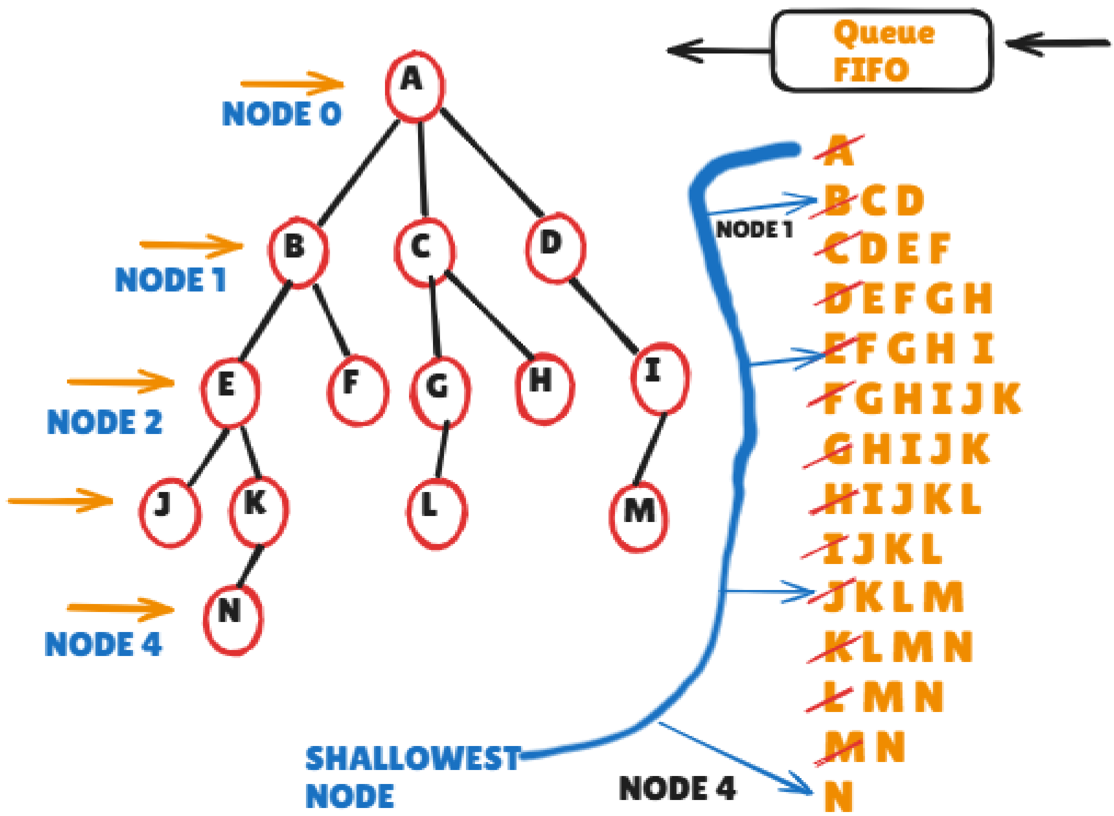
1. Start by inserting the **initial state** into a queue
2. Remove the first element (front)
3. If it is the **goal**, stop
4. Otherwise, expand all its children and add them to the **back of the queue**

5. Repeat until the goal is found or queue is empty

🔷 Data Structure Used

Queue (FIFO – First In, First Out)

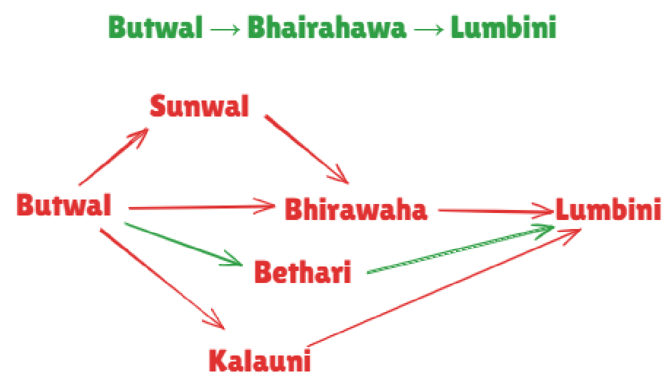
- First generated node → first processed



🔷 Properties of BFS

Property	Description
Complete	✓ Yes — if a solution exists, BFS will find it
Optimal	✓ Yes — for equal step costs
Time Complexity	O(b^d)
Space Complexity	O(b^d) — very high
Strategy	Explores nodes level-wise

🔷 Example: Route Finding



Finding the shortest path between two cities (e.g., **Butwal → Bhairahawa → Lumbini**).
BFS guarantees the least number of turns/roads if all roads have equal cost.

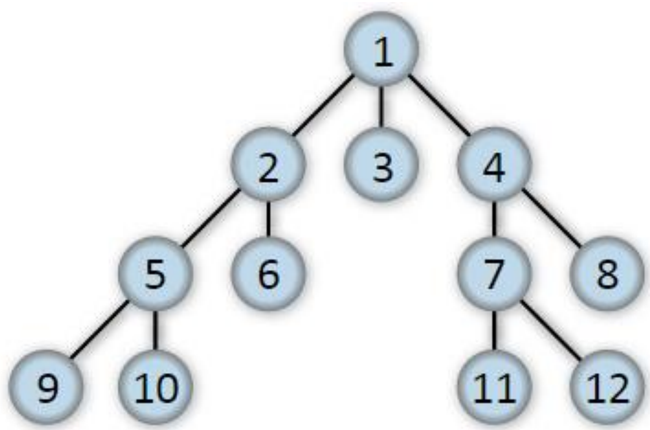
🔷 Real-Life Analogy

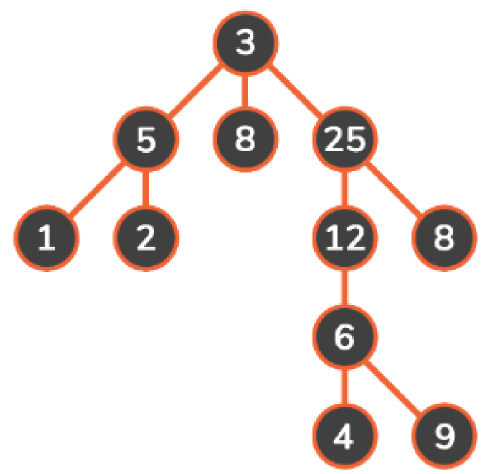
Searching for someone in a building floor-by-floor:

- Check ground floor rooms first
- Then 1st floor rooms
- Then 2nd floor rooms

This is exactly BFS.

Classwork (Just Now)





📌 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**
Explores closest nodes first *Uses a queue* *Complete (always finds a solution)* *Optimal (shortest path if costs equal)* *Memory-heavy for large trees*				

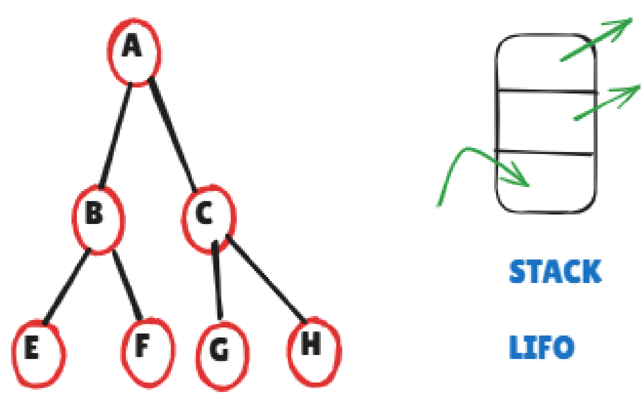
2.2.2.2 Depth-First Search (DFS)

- 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.

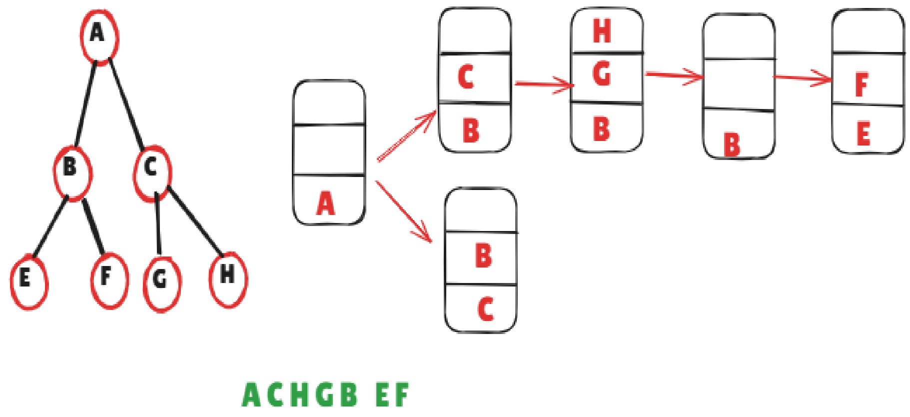
Key Characteristics of DFS

- 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

◆ **DFS Working Principle**

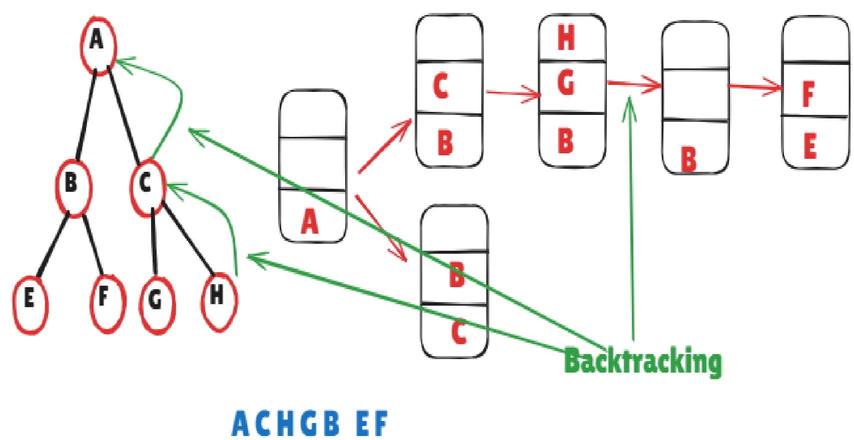


1. Push initial node into stack
2. Pop top node
3. If it is the goal, stop
4. Otherwise, expand it and push all children
5. Continue until stack is empty

◆ **Data Structure Used**

Stack (LIFO – Last In, First Out)

Recently added nodes are explored first.



◆ **DFS Search Tree (Conceptual)**

A

/ | \

B C D

/

E

DFS explores:
A → B → E → (backtrack) → C → D

◆ **Properties of DFS**

Property	**Description**
Complete	❌ No (fails on infinite depth)
Optimal	❌ No (does not guarantee shortest path)
Time Complexity	$O(b^m)$
Space Complexity	O(bm) — very low

Property	Description
Strategy	Deep-first exploration

Where:

- **b** = branching factor
- **m** = maximum depth

🔷 DFS Stack Table (Step-by-Step)

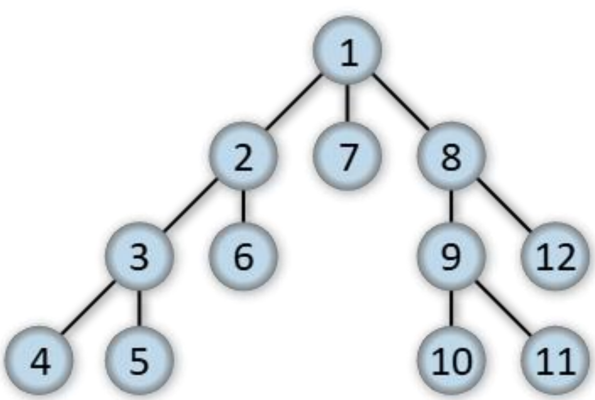
Step	Stack (Top → Bottom)	Expanded Node
1	A	A
2	B, C, D	B
3	E, F?, C, D	E
4	–	**Goal found**

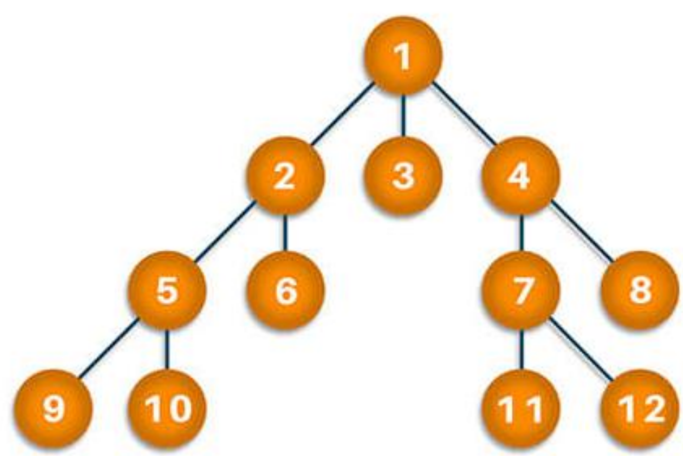
(E is first discovered in the deep branch → DFS stops immediately)

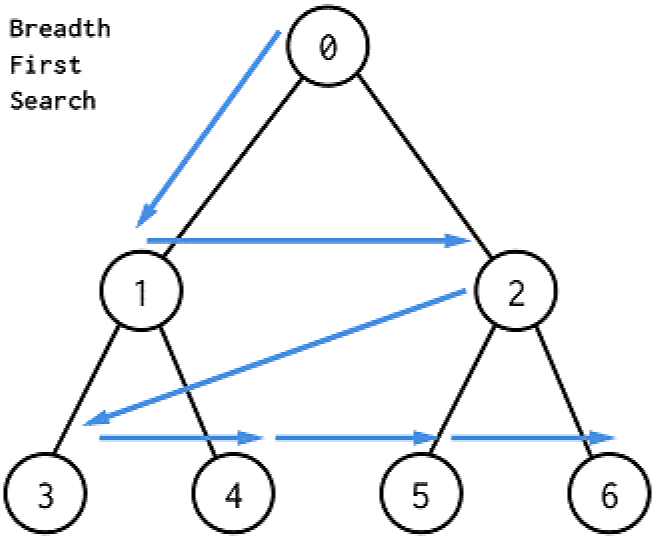
🔷 Real-Life Analogy

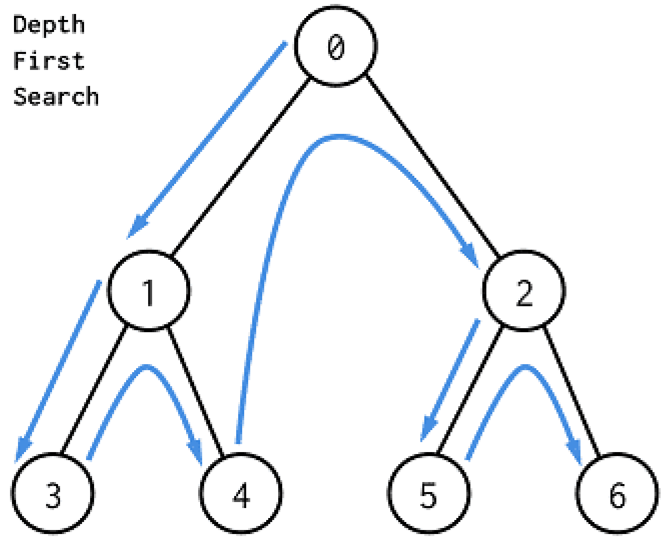
Searching for a lost item by checking a **long corridor first**, going to the end, and only then checking other rooms.
DFS always prefers deep paths first.

Classwork (Just Now)









📘 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**.

👉 **Prform 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)

Sanjeev Thapa, Er. DevOps, SRE, CKA, RHCSA, RHCE, RHCSA-Openstack, MTCNA, MTCTCE, UBSRS, HEv6, Research Evangelist

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}

✅ **MCQ Questions: BFS & DFS**

1. Which data structure is used by BFS?
A. Stack
B. Queue
C. Priority Queue
D. Linked List

2. Which data structure is used by DFS?
A. Stack
B. Queue
C. Heap
D. Hash Table

3. BFS always finds the ______ path if all edges have equal weight.
A. Longest
B. Random
C. Shortest
D. Deepest

4. DFS is best suited for problems where the solution is located at:
A. Shallow depth
B. Deep depth
C. Middle level
D. No constraints

5. The worst-case time complexity of BFS is:
A. O(b)
B. $O(b^2)$

C. $O(b^d)$
D. $O(n^2)$
(b = branching factor, d = depth)

6. DFS is not complete when:
A. Branching factor is small
B. Path cost is uniform
C. State space is finite
D. State space is infinite

7. Which search can get stuck in an infinite loop?
A. BFS
B. DFS
C. Both
D. Neither

8. BFS is optimal when:
A. All step costs are equal
B. Step costs vary
C. Graph is cyclic
D. Graph is disconnected

9. DFS uses _______ memory compared to BFS.
A. More
B. Equal
C. Less
D. Variable

10. Depth-Limited Search is used to prevent:
A. Cycles
B. Large branching
C. Infinite depth traversal
D. Optimal solutions

11. Which search expands all neighbors of a node before moving deeper?
A. DFS
B. BFS
C. Hill-Climbing
D. A*

12. DFS works naturally with:
A. Recursion
B. Iteration only
C. Priority rules
D. Cost-based heuristics

13. In which scenario is BFS preferred?
A. Very deep trees
B. Memory is limited
C. Finding shortest path
D. Depth is unknown

14. DFS is commonly used in:

A. Route finding
B. Cycle detection
C. Uniform-cost search
D. Greedy algorithms

15. The space complexity of BFS is:
A. O(bm)
B. O(b^d)
C. O(m)
D. O(1)

✅ **MCQ Answers with Reasons**

1. Which data structure is used by BFS?
Correct Answer: B. Queue
Reason:
BFS explores nodes **level by level**. It processes nodes in the order they are discovered → **First In, First Out (FIFO)** behavior → implemented using a **queue**.

2. Which data structure is used by DFS?
Correct Answer: A. Stack
Reason:
DFS goes **as deep as possible** before backtracking. This uses **Last In, First Out (LIFO)** behavior → implemented using a **stack** (or recursion stack).

3. BFS always finds the ______ path if all edges have equal weight.
Correct Answer: C. Shortest
Reason:
BFS explores all nodes at **distance 1**, then distance 2, etc. So, the **first time** it reaches the goal, it has found the path with **minimum number of steps** → shortest path (for equal-cost edges).

4. DFS is best suited for problems where the solution is located at:
Correct Answer: B. Deep depth
Reason:
DFS explores **deep paths first**, so if the solution is far down in the search tree, DFS may find it faster than BFS (which explores shallow levels first).

5. The worst-case time complexity of BFS is:
Correct Answer: C. O(b^d)
Reason:
In the worst case, BFS explores **all nodes up to depth d**, with each node having up to **b children (branching factor)** → total nodes ≈ b^d.

6. DFS is not complete when:
Correct Answer: D. State space is infinite
Reason:
DFS can keep going down an **infinite path** and may **never backtrack** to explore other branches where a solution exists → not complete in infinite state spaces.

7. Which search can get stuck in an infinite loop?
Correct Answer: B. DFS
Reason:
DFS may follow a path that **loops** or continues indefinitely, especially without **cycle checking or depth limit**. BFS, on the other hand, explores level-wise and eventually terminates if the graph is finite.

8. BFS is optimal when:
Correct Answer: A. All step costs are equal
Reason:
With equal step costs, the path with **fewest steps** is also the **lowest cost path**. BFS expands in increasing depth → first goal found = optimal.

9. DFS uses ______ memory compared to BFS.
Correct Answer: C. Less
Reason:
DFS only needs to store the **current path and a few siblings**, giving space **O(bm)**, while BFS stores **all nodes at current level** → space **$O(b^d)$**, which is much larger.

10. Depth-Limited Search is used to prevent:
Correct Answer: C. Infinite depth traversal
Reason:
DLS is just DFS with a **maximum depth limit**. This prevents DFS from going endlessly down an infinite branch.

11. Which search expands all neighbors of a node before moving deeper?
Correct Answer: B. BFS
Reason:
BFS explores **all nodes at current depth** (all neighbors), then moves to the **next level**. DFS goes deep first, not neighbor-by-neighbor.

12. DFS works naturally with:
Correct Answer: A. Recursion
Reason:
Recursive function calls use the **call stack**, which works exactly like DFS: go deep → return → backtrack. So DFS is very naturally implemented using **recursion**.

13. In which scenario is BFS preferred?
Correct Answer: C. Finding shortest path
Reason:
Because BFS guarantees the **shortest path in terms of number of steps** when all edges have equal weight, it is ideal for shortest path problems in unweighted graphs.

14. DFS is commonly used in:
Correct Answer: B. Cycle detection
Reason:
DFS can keep track of visited nodes and recursion stack → easily detect **back edges** → widely used for **cycle detection** in graphs and in algorithms like topological sort.

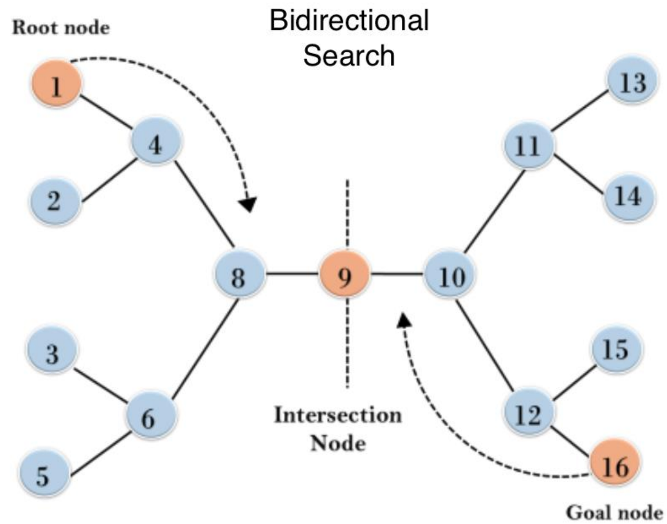
15. The space complexity of BFS is:
Correct Answer: B. $O(b^d)$
Reason:
BFS stores **all nodes at the current frontier (level)**. In the worst case, the number of nodes at the deepest level is ≈ **b^d**, so both time and space complexity are **$O(b^d)$**.

2.2.2.4 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.

🔷 Why It Is Fast

If a regular BFS takes **bᴅ** time,
Bidirectional Search takes approximately:

$$O(b^{d/2} + b^{d/2}) = O(b^{d/2})$$

This is **much faster** because exponential growth is cut into half depth.

🔷 Requirements

Bidirectional Search works only when:

- The **initial** and **goal** states are clearly defined
- It is possible to **reverse** actions (goal → initial)
- Branching factor is manageable
- There is a **unique goal state** (or small goal set)

🔷 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.

🔷 Advantages

✓ Very fast compared to BFS
✓ Greatly reduces number of nodes visited
✓ Ideal when state space is large but depth is small

🔷 Limitations

✗ Hard when actions cannot be reversed
✗ Difficult if there are multiple goal states
✗ Memory requirement is still high (two BFS frontiers)

Example: Route as a Simple Graph

1 Route as a Simple Graph

Ram wants to go from **Golpark** to **Lumbini** via:

Golpark → Chaura → Manigram → Bhairahawa → Bethari → Pars → Lumbini

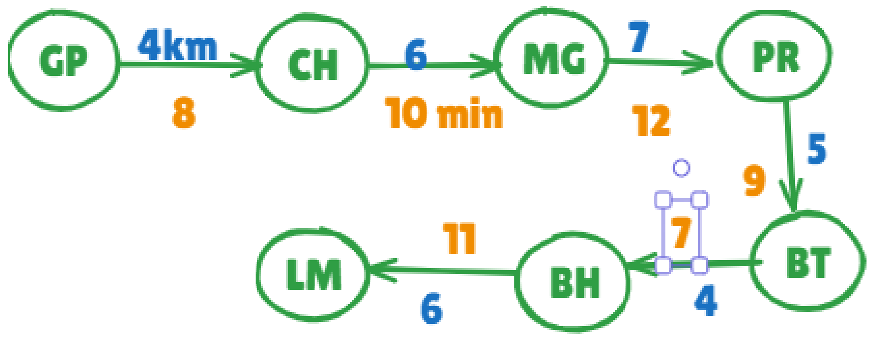
Treat each place as a node in a line graph:

Golpark – Chaura – Manigram – Bhairahawa – Bethari – Pars – Lumbini

Let's assume approximate **distance and time**:

Edge	**Distance (km)**	**Time (min)**
Golpark → Chaura	4 km	8 min
Chaura → Manigram	6 km	10 min
Manigram → Bhairahawa	7 km	12 min
Bhairahawa → Bethari	5 km	9 min
Bethari → Pars	4 km	7 min
Pars → Lumbini	6 km	11 min

Total (straight route):
Distance = 4+6+7+5+4+6 = **32 km**
Time = 8+10+12+9+7+11 = **57 min**



Start state: **Golpark**

Goal state: **Lumbini**

2️⃣ Normal BFS (Single Direction, Start → Goal)

BFS from **Golpark** just walks forward level by level (here levels are just next city).

We assume adjacency:

- **Golpark ↔ Chaura**
- **Chaura ↔ Golpark, Manigram**
- **Manigram ↔ Chaura, Bhairahawa**
- **Bhairahawa ↔ Manigram, Bethari**
- **Bethari ↔ Bhairahawa, Pars**
- **Pars ↔ Bethari, Lumbini**
- **Lumbini ↔ Pars**

🔷 **BFS Queue Simulation**

Step	Queue (front → back)	Expanded Node	Visited Set
1	Golpark	Golpark	{Golpark}
2	Chaura	Chaura	{Golpark, Chaura}
3	Manigram	Manigram	{Golpark, Chaura, Manigram}
4	Bhairahawa	Bhairahawa	{Golpark, Chaura, Manigram, Bhairahawa}
5	Bethari	Bethari	{… , Bethari}
6	Pars	Pars	{… , Pars}
7	Lumbini	Lumbini	{… , Lumbini}

👉 BFS finds the path:
Golpark → Chaura → Manigram → Bhairahawa → Bethari → Pars → Lumbini

---

3️⃣ Bidirectional Search (From Golpark and Lumbini)

Now we do **Bidirectional BFS**:

- Forward BFS from **Golpark**
- Backward BFS from **Lumbini**
- Both expand level by level until their **visited sets meet**

🔷 **Initial**

- Forward queue Q_f= [Golpark]
- Backward queue Q_b= [Lumbini]
- Visited_f = {Golpark}
- Visited_b = {Lumbini}

🔷 **Step-by-step Bidirectional BFS**

🔶 **Step 1 – Forward from Golpark**

Forward Step	Q_f (front → back)	Expanded	New Added	Visited_f
F1	Golpark	Golpark	Chaura	{Golpark, Chaura}

Now backward side:

Backward Step	Q_b (front → back)	Expanded	New Added	Visited_b
B1	Lumbini	Lumbini	Pars	{Lumbini, Pars}

No meeting yet (no common node in Visited_f and Visited_b).

🔶 Step 2 – Forward from Chaura, Backward from Pars

Forward:

Forward Step	Q_f	Expanded	New Added	Visited_f
F2	Chaura	Chaura	Manigram	{Golpark, Chaura, Manigram}

Backward:

Backward Step	Q_b	Expanded	New Added	Visited_b
B2	Pars	Pars	Bethari	{Lumbini, Pars, Bethari}

Still no common node.

🔶 Step 3 – Forward from Manigram, Backward from Bethari

Forward:

Neighbors of **Manigram**: Chaura, Bhairahawa

Chaura already visited, so add only **Bhairahawa**.

Forward Step	Q_f	Expanded	New Added	Visited_f
F3	Manigram	Manigram	Bhairahawa	{Golpark, Chaura, Manigram, Bhairahawa}

Backward:

Neighbors of **Bethari**: Bhairahawa, Pars

Pars already visited, so add only **Bhairahawa**.

Backward Step	Q_b	Expanded	New Added	Visited_b
B3	Bethari	Bethari	Bhairahawa	{Lumbini, Pars, Bethari, Bhairahawa}

👉 Now **Bhairahawa** is in **both Visited_f and Visited_b**.
So **meeting node** = **Bhairahawa**.

🔷 **Reconstructing the Path**

From **forward side** (start → meet):

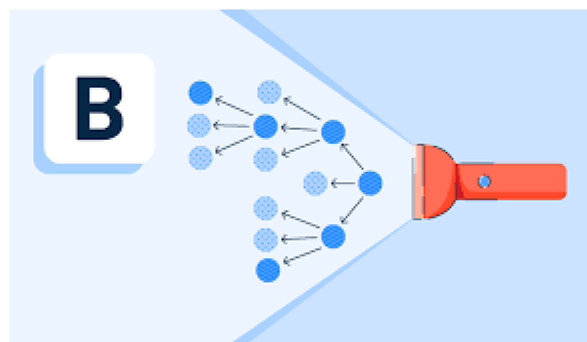
- Golpark → Chaura → Manigram → Bhairahawa

From **backward side** (goal → meet):

- Lumbini ← Pars ← Bethari ← Bhairahawa
  → So forward direction: Bhairahawa → Bethari → Pars → Lumbini

✅ Final path from Bidirectional Search:
Golpark → Chaura → Manigram → Bhairahawa → Bethari → Pars → Lumbini
Same path, but **search effort was reduced** because each search explored only half the depth.

2.2.2.5 Beam Search (NIC) (Not in Course)



Beam Search is a **heuristic search technique** that keeps only a **fixed number of best nodes** (called the *beam width*) at each level of the search tree.
It is more memory-efficient than Best-First Search and faster than exploring all nodes.

It is considered a **hybrid** between:

- **Breadth-First Search (BFS)** → because it explores level-wise
- **Greedy Search** → because it selects "best" nodes based on heuristic value

---

🔷 **Key Idea**

Instead of expanding *all* nodes at each level (like BFS), Beam Search expands **only the top K most promising nodes** according to a heuristic value.

Where:

- **K = beam width** (user-defined)
- Lower K = faster, less accurate
- Higher K = slower, more accurate

🔷 How Beam Search Works

1. Start from the initial state.
2. Expand all successors.
3. **Select the best K nodes** based on the heuristic $h(n)$.
4. Discard all other nodes.
5. Repeat level-by-level until the goal is found.

Beam Search trades **optimality** for **speed & memory efficiency**.

🔷 Example (Simple & Clear)

Assume we want to search for the best path from **Start (S)** to **Goal (G)**.

At one level, we generate:

Node	Heuristic Cost h(n)
A	3
B	7
C	2
D	9

If beam width **K = 2**, keep only **two best nodes**:
→ **C (2)** and **A (3)**

Discard B and D.

At next level, expand only C and A, again picking the top nodes.

This continues until **G** is reached.

🔷 Where Beam Search is Used

Beam Search is widely used in:

- **Speech recognition**
- **Machine translation (NLP)**
- **Autocorrect / text prediction**
- **Large search spaces** where BFS is too costly
- **Game playing (with large branching)**

Example:
Google's autocomplete uses beam search to keep only a few top probable word sequences.

🔷 Advantages

✓ Uses **much less memory** than BFS
✓ Faster than exploring all nodes
✓ Works well in large state spaces
✓ Good for **approximate** solutions

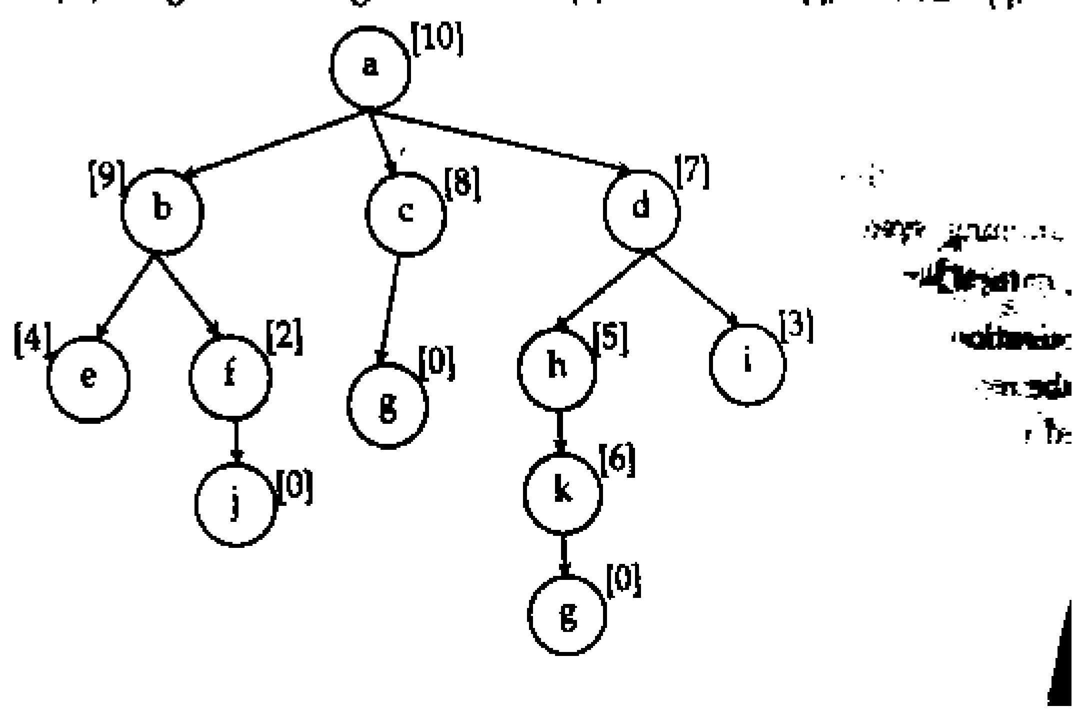
🔷 Disadvantages

✗ Not optimal — may miss the best solution
✗ May discard the correct path early
✗ Depends heavily on heuristic quality
✗ Beam width selection is tricky

Example: Beam Search

Example: Tracing of beam search algorithm

Consider, start node (N) = a, goal node = g, beam width (n) = 2, OPEN = { }, CLOSE = { }.



Start node = a	**Goal node = g**	**Beam width n = 2**
OPEN = { } , CLOSE = { }		

Heuristic values (h) are shown inside brackets in the diagram.

📌 **LEVEL 0 (Start)**

OPEN = { a(10) }
CLOSE = { }
Expand **a**.
Children of a:

- b(9)
- c(8)
- d(7)

Now pick the **best 2** (lowest heuristic value = most promising).
Sorted by heuristic:

1. d(7) ← best
2. c(8)
3. b(9)

Beam Width = 2 → Keep only:

👉 **OPEN = { d(7), c(8) }**
👉 **CLOSE = { a }**

📌 **LEVEL 1**

Expand nodes in OPEN one-by-one.

➤ **Expand d(7)**
Children:

- h(5)
- i(3)

➤ **Expand c(8)**
Child:

- g(0) ← **Goal found here BUT beam selection must still apply!**

All children generated this level:

- h(5)
- i(3)
- g(0)

Sort by heuristic:

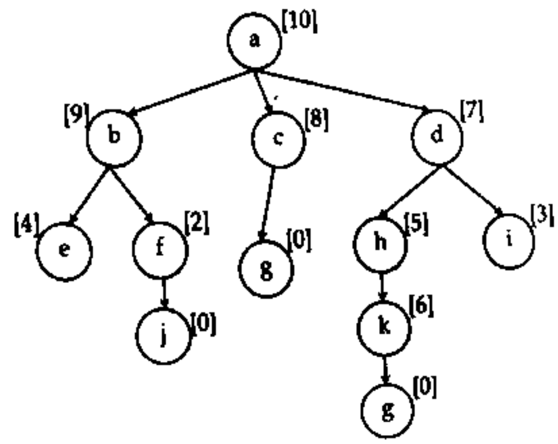
1. g(0)
2. i(3)
3. h(5)

Beam width = 2 → Keep only:
👉 **OPEN = { g(0), i(3) }**
👉 **CLOSE = { a, d, c }**

⚠ Since **g is already in OPEN**, search terminates.

🎉 **Goal Found: g**

Beam search stops as soon as the goal appears in OPEN.



✔ **Final Trace Table (Clean)**

LEVEL	EXPANDED	GENERATED NODES	BEST 2 SELECTED (BEAM)
0	a(10)	b(9), c(8), d(7)	**d(7), c(8)**
1	d(7), c(8)	h(5), i(3), g(0)	**g(0), i(3)**
	Goal **g** is now reached → STOP.		
	Beam Search Path = a → d → g (Because node d generated g earlier through beam selection process)		

2.2.2.5 Heuristic

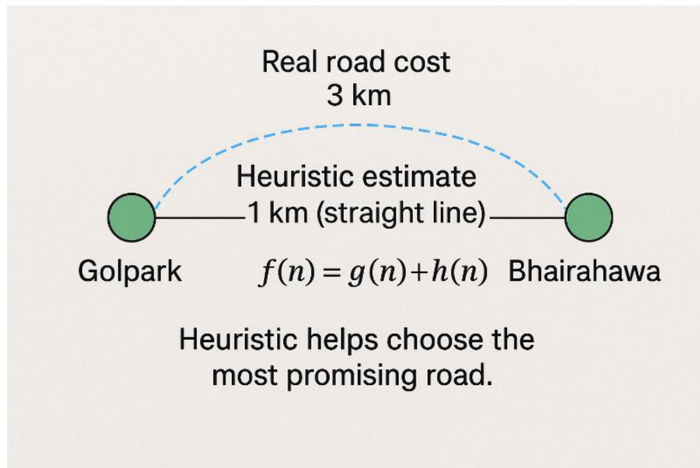
A **heuristic** is an **estimate** or **educated guess** that helps an AI search algorithm decide which direction to explore first.
It does **NOT** have to be perfect or accurate —
it only needs to be **close enough** to guide the search toward the goal.

In AI search (like A*, Greedy), heuristic is written as:

$$h(n) = \text{estimated cost from node } n \text{ to the goal}$$

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**.

🎯 **So, why heuristic?**
Because:
- It **underestimates** the real distance
- It gives a **quick estimate** to guide search
- It does **not require exact roads**
- It helps AI choose the best direction to move first

This is exactly the **Euclidean heuristic (h(n))** used in A* Search.

📌 **Example in AI Terms**
State = "Golpark"
Goal = "Bhairahawa"
- Real road cost (g) = **3 km**
- Heuristic estimate (h) = **1 km** (straight line)

In A*:

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

Heuristic helps choose the **most promising road**.

🚗 **Why AI Uses Heuristics Like This**

Because AI wants to make search **faster**:

- Without heuristic (Uninformed search):
  → AI checks all possible routes blindly (BFS, DFS)
- With heuristic (Informed Search):
  → AI quickly moves toward Bhairahawa because "straight-line distance is small"

Thus, heuristic **saves time**, **reduces nodes explored**, and **guides search intelligently**.

✔ **Short, Simple Definition (Exam sentence)**

A **heuristic** is a problem-specific, approximate estimate that guides search toward the goal by predicting which state is likely to be better or closer.

2.2.2.5 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**

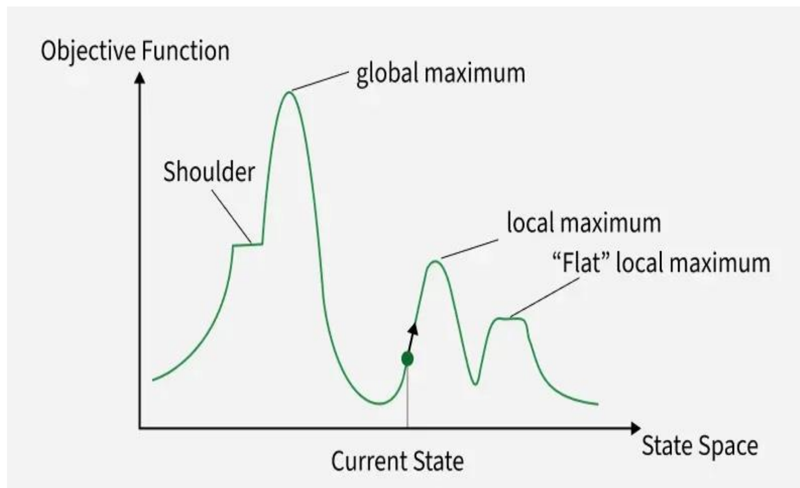
Hill climbing is simple but suffers *from traps such as* ***local maxima****,* ***ridges****, and* ***plateaus***.

🔷 Types of Hill Climbing

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

🔷 Example

Imagine Ram wants to climb a hill in fog.
He can only see a few steps around him and always chooses the direction that goes upward.

- If the hill has multiple peaks → he may climb a **small hill** (local maximum) instead of the tallest one (global maximum).
- If he reaches a flat top → he stops (plateau).

This is exactly how hill climbing behaves.

◆ **Algorithm Steps**

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

◆ **Real AI Application**

- Feature selection in ML
- Scheduling & timetabling
- Path optimization
- Robotics movement planning

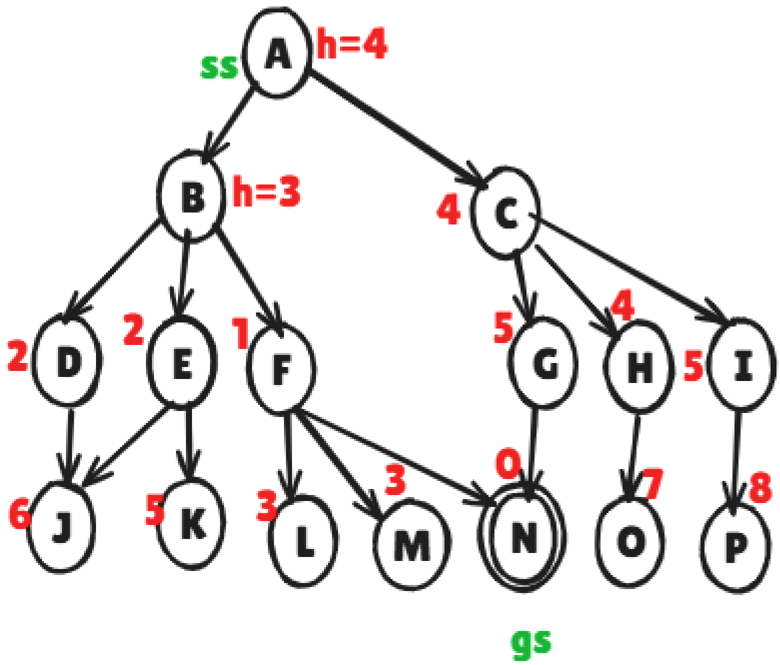
◆ **Advantages**

✓ Simple & easy to implement
✓ Uses very little memory
✓ Works well for continuous optimization

◆ **Disadvantages**

✗ Gets trapped in local maxima/minima
✗ Prone to plateaus
✗ No guarantee of reaching global optimum
✗ Depends heavily on start point

Example: Start at A and Goal at N



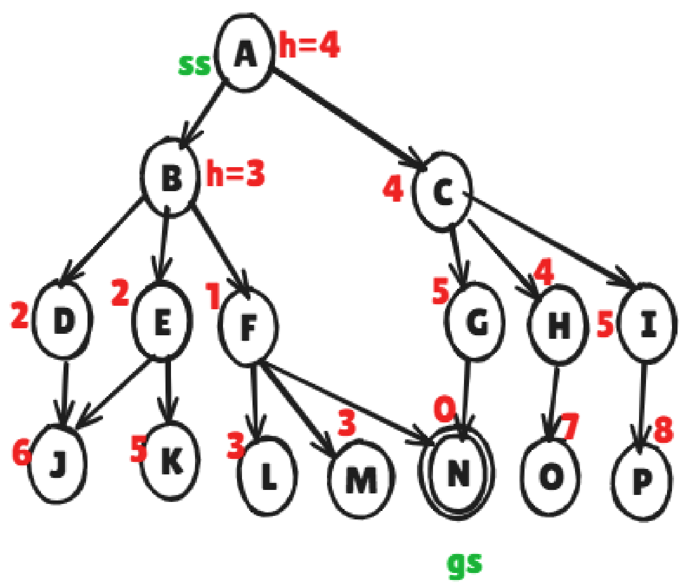
Example: Goal at N

We **assume**:

- We always choose the **child with the smallest h-value** (steepest improvement).
- We **move only if h(new) < h(current)**.
- If no neighbor has lower h, we **stop** (local minimum).
- Start state $S = A$, goal node $G = N$.

✅ 1. **Understanding Your Tree (h-values as height / cost)**

A = 4
B = 3
D = 2
J = 6
E = 2
K = 5
F = 1
L = 3
M = 3
N = 0 ← GOAL (global minimum)
C = 4
G = 5
H = 4
O = 7
I = 5
P = 8



➞ Lower h = better
➞ Hill-climbing always chooses **lower h**
➞ So "global optimum" = **h = 0** (node N)

✅ **2. Find the Hill Climbing Path**
Start at **A (4)** → choose lowest child (best improvement)

Step	Current Node	Children	Best Child	Move
1	A(4)	B(3), C(4)	**B(3)**	A → B
2	B(3)	D(2), E(2), F(1)	**F(1)**	B → F
3	F(1)	L(3), M(3), N(0)	**N(0)**	F → N (Goal!) ✓

⭐ **Final Hill-Climbing Path:**

$$A \rightarrow B \rightarrow F \rightarrow N$$

So the algorithm **stops at node F (h = 1)** because **none of its neighbors have a lower heuristic value**.

✅ **3. Now map your numerical values to the *landscape figure***

("global maximum", "local maximum", "plateau", "shoulder")
Using **your h-values**, we classify:

🔷 **GLOBAL OPTIMUM (best possible state)**
👉 The state with the **lowest h-value**

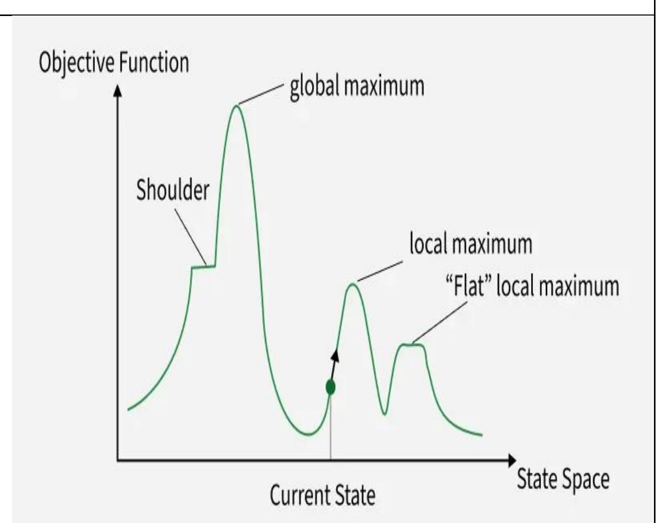
N (h=0)= Global Minimum / Global Optimum

This is the GOAL (gs).

🔷 **LOCAL MAXIMUM / LOCAL MINIMUM**

- A **local minimum** is a node whose children all have **higher h-values**.
- A **local maximum** (in climbing upward) would have the opposite.

In your tree, local minimum nodes (valleys):



Node	h	Children h-values	Reason
D	2	J(6)	J > 2 → D is a local minimum
E	2	K(5)	K > 2 → E is a local minimum
F	1	L(3), M(3), N(0)	Not a local min because N < 1
G	5	N/A	Leaf → local minimum but not relevant
H	4	O(7)	O > 4 → local minimum
I	5	P(8)	P > 5 → local minimum

Most important "hill-climbing trap nodes":

- D (h=2)
- E (h=2)
- H (h=4)
- I (h=5)

If hill climbing ever goes to D or E, it gets *stuck*.

🔷 PLATEAU

A **plateau** is when several nodes have **the same h-value** and no clear improvement direction.
Examples in your tree:

1. From A → children B(3) and C(4) — not plateau
2. **C(4), H(4)** → both = 4
3. **L(3), M(3)** → plateau region
4. **D(2), E(2)** → plateau (equal heuristic)

Plateau groups:

Plateau Group	Nodes
h = 2	D, E
h = 3	L, M
h = 4	A, C, H

🔷 SHOULDER

A shoulder is a small rise before another descent.

Example in your tree:

- A(4) → B(3) → **D(2)**
  If we went from B to E or D, hill climbing may stall at 2 even though better state exists at 0.

Graphically:

4 → 3 → 2(shoulder)

After that the algorithm might mistakenly think 2 is the best nearby.

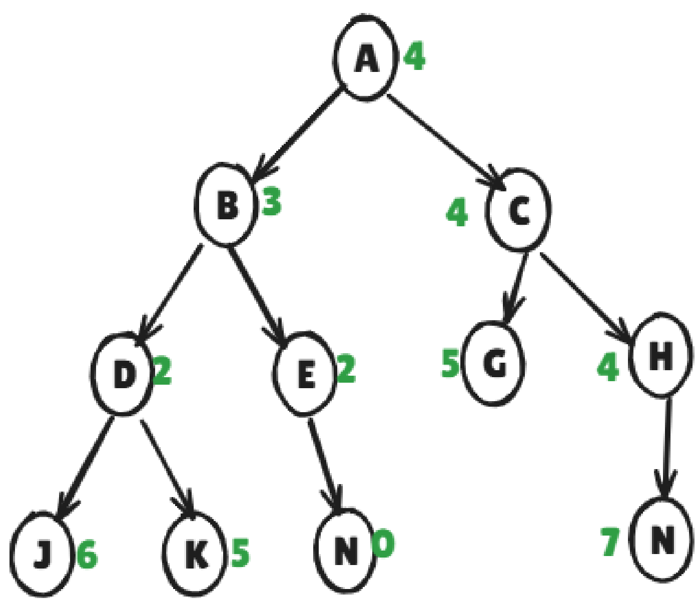
✅ 4. Mapping to the Landscape Diagram (Your Values)

Landscape Feature	Node(s)	Meaning in your tree
Global Maximum / Minimum	**N (h = 0)**	Best possible solution
Local Maximum / Minimum	D(2), E(2), H(4), I(5)	Algorithm can get stuck
Plateau	{D,E}, {L,M}, {A,C,H}	Many equal-valued neighbors
Shoulder	A → B → D	A small descent before a trap

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:
   - the **global optimum**
   - the **local optimum** where hill climbing gets stuck
   -

🔷 Step 1 – Start at A

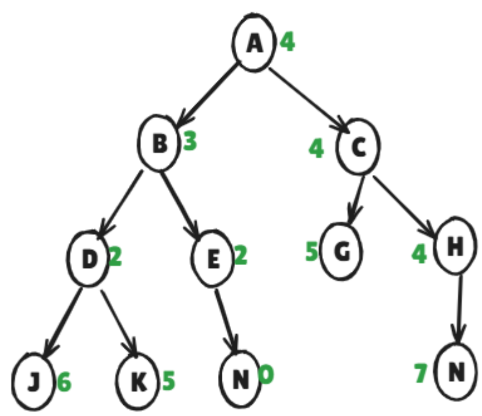
Current node: **A (h = 4)**
Children of A:

- 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:

$$A \rightarrow B \rightarrow D$$

🔷 Step 3 – At node D

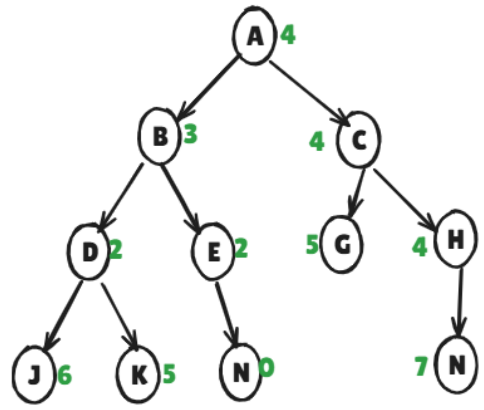
Current node: **D (h = 2)**
Children of D:

- 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

$$\boxed{A \to B \to D}$$

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:
  - 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**.

⛰️ Global vs Local Optimum

🔷 Global Optimum

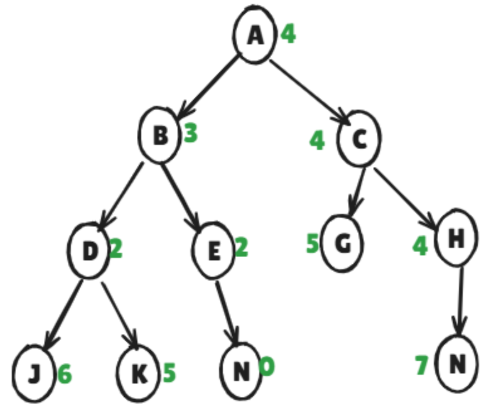
The **best possible** state (smallest h):

$$\boxed{\text{Global optimum} = N,\ h(N) = 0}$$

This is the **true goal**.

🔷 Local Optimum

A node where **all children are worse**, but it is **not** the best overall.



At **D (h = 2)**:

- Children: J(6), K(5) → both higher than 2
- So D is a **local minimum** (local optimum for hill climbing).

$$\boxed{\text{Local optimum where the search gets stuck} = D,\ h(D) = 2}$$

Hill climbing **thinks** D is best in its neighborhood and stops, even though there is a **better global solution N (0)**.

Example:

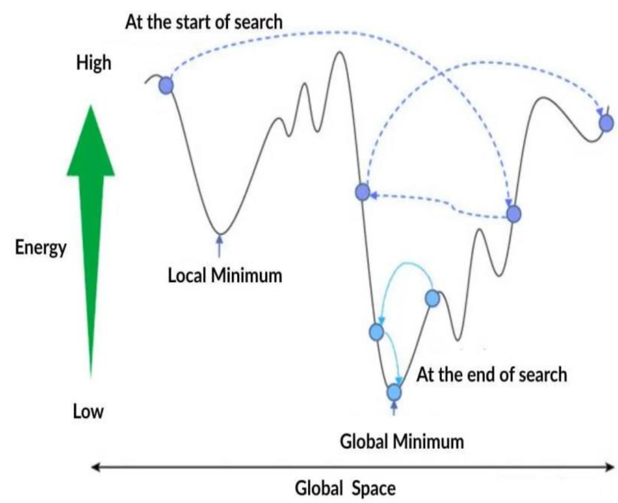
Question 1	Question 2
Hill Climbing that **reaches** the Goal	Hill Climbing that **gets stuck** in a Local Optimum
Start state = **A**, Goal state = **N**.	Start state = **A**, Goal state = **M**.
A(10) connects to B(9), C(7), D(8) B(9) connects to E(6), F(8), O(11), P(12) C(7) connects to F(5), G(9) D(8) connects to Q(9), R(10) E(6) connects to H(4), I(6) F(5) connects to H(4), G(9) G(9) connects to S(8) H(4) connects to J(3), K(2) I(6) connects to T(7) J(3) has no children K(2) connects to L(1), M(3) L(1) connects to N(0) M(3) has no children N(0) has no children O(11), P(12), Q(9), R(10), S(8), T(7) have no children	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
Question: 1. Using steepest-ascent hill climbing starting from **A**, list the sequence of nodes visited until the algorithm stops. 2. State whether the algorithm reaches the goal node **N**.	Question: 1. Using steepest-ascent hill climbing starting from **A**, trace the exact path followed until the algorithm stops. 2. Does the algorithm ever reach the goal node **M**? 3. Identify:

3. Is the final node a global optimum or just a local optimum? Explain using the heuristic values.	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.

2.2.2.5 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:
  - Move to a **neighboring state**.
  - If the new state is **better**, accept it.
  - 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:

- $E_{current}$= cost/value of current state
- E_{new}= cost/value of new state
- $\Delta E = E_{new} - E_{current}$

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.

🔷 Simple Real Scenario

Suppose you are trying to find the **best route between several cities** (like a small Travelling Salesman Problem).

- **State** = one complete tour order of cities
- **Cost** = total distance of the tour
- **Neighbor** = swap two cities in the order

Simulated Annealing:

1. Starts with a random tour
2. Sometimes accepts longer routes early (to explore widely)
3. Later (as T decreases), it becomes strict and keeps only shorter routes
4. Finally returns a **near-optimal route** without checking all possibilities

---

🔷 Advantages

- Can **escape local maxima/minima**
- Good for **very large, complex search spaces**
- Simple and general (works on many optimization problems)

🔷 Disadvantages

- No guarantee of true **global optimum**
- Performance depends on **cooling schedule** and parameter choice
- May be slow if cooling is too gradual

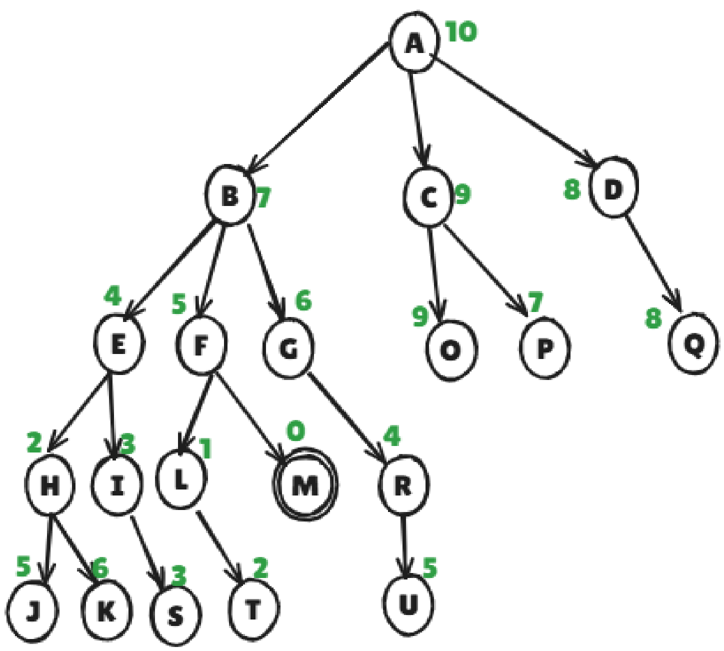
🔷 **Where It Is Used**

- **Combinatorial optimization:** TSP, scheduling, layout design
- **VLSI design**, network routing
- **Parameter tuning** in machine learning
- Many problems where exhaustive search is impossible

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



The **same graph(Questions above) ……………….** that traps Hill Climbing can be solved using **Simulated Annealing**, because SA is allowed to **accept worse moves** (higher h) with some probability and can escape the local optimum.

Now

- Recall the graph
- Show **how Hill Climbing gets stuck**
- Then show a **Simulated Annealing run step-by-step** that reaches the goal **M**

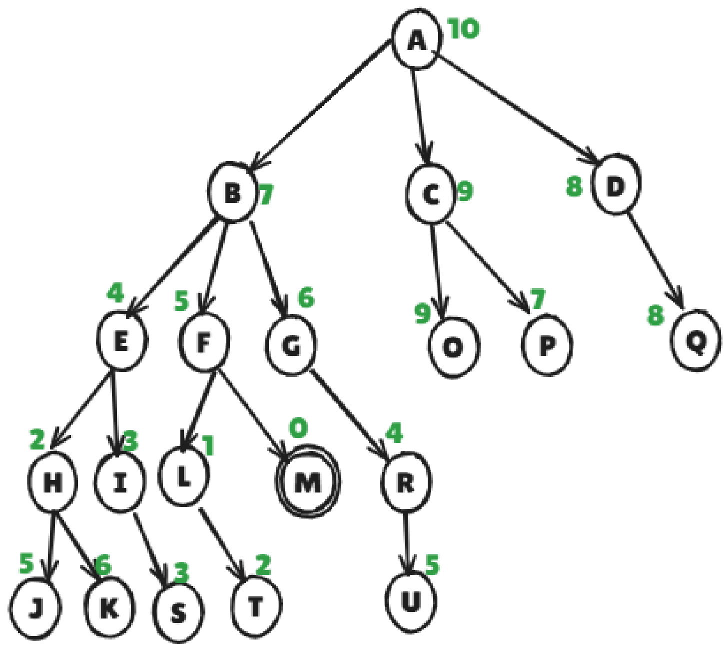
1 Graph (given in your question)

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

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

Connections (undirected / neighbors):

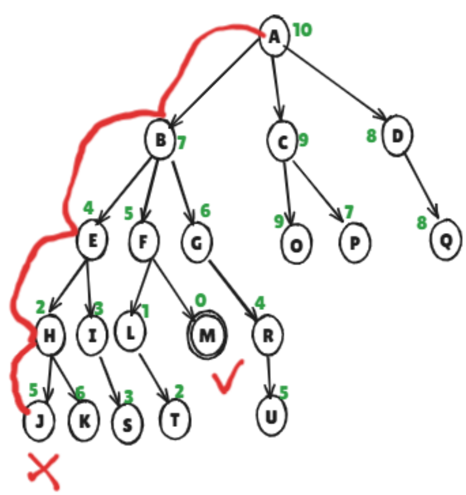
- *A(10) → B(7), C(9), D(8)*
- *B(7) → A(10), E(4), F(5), G(6)*
- *C(9) → A(10), O(9), P(7)*
- *D(8) → A(10), Q(8)*
- *E(4) → B(7), H(2), I(3)*
- *F(5) → B(7), L(1), M(0)*
- *G(6) → B(7), R(4)*
- *H(2) → E(4), J(5), K(6)*
- *I(3) → E(4), S(3)*
- *J(5) → H(2)*
- *K(6) → H(2)*
- *L(1) → F(5), T(2)*
- *M(0)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



2 Hill Climbing on this graph (why it gets stuck)

Using **steepest-ascent hill climbing** (minimize h):

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



👉 **Hill climbing stops at H(2)** (local optimum)

It **never reaches M(0)**.

3️⃣ 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):
  - $T_0 = 10$, then $T \leftarrow 0.8 \times T$after each move
- Neighbor = any adjacent node
- Acceptance rule:
  - If $\Delta E = h_{\text{new}} - h_{\text{current}} \leq 0 \rightarrow$ **always accept**
  - If $\Delta E > 0 \rightarrow$ accept with probability

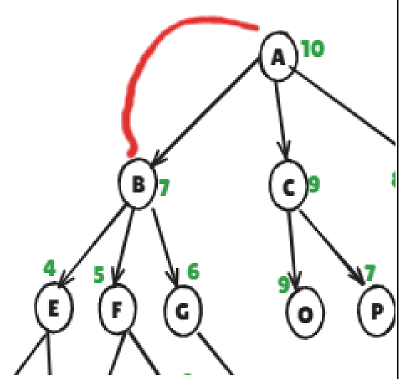
$$P = e^{-\Delta E/T}$$

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

4️⃣ Step-by-step Simulated Annealing Run

🧮 Legend

- 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

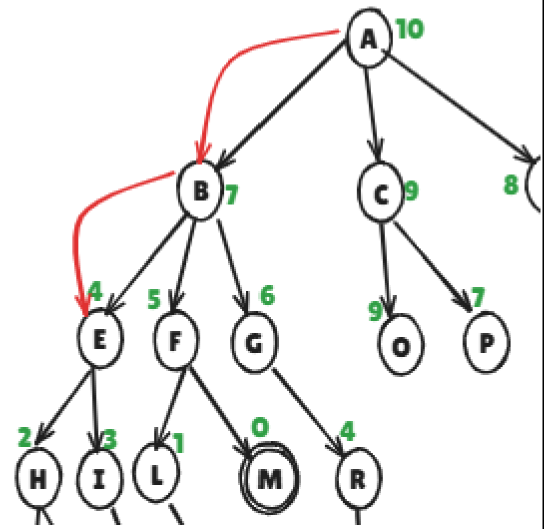
Choose neighbor randomly (say B(7)).

- 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)

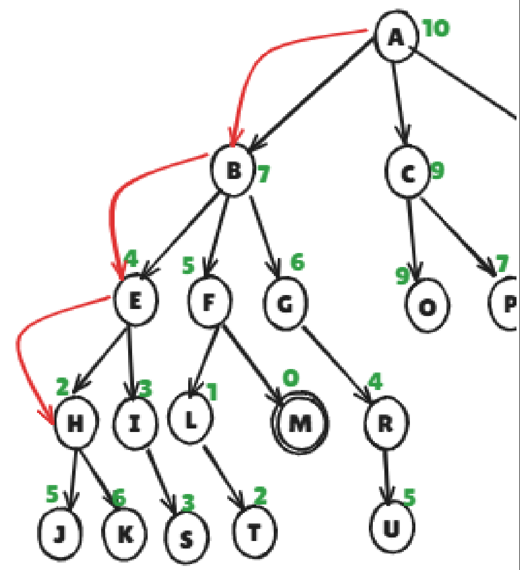
Suppose candidate = **E(4)**

- ΔE = 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)**

- ΔE = 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)

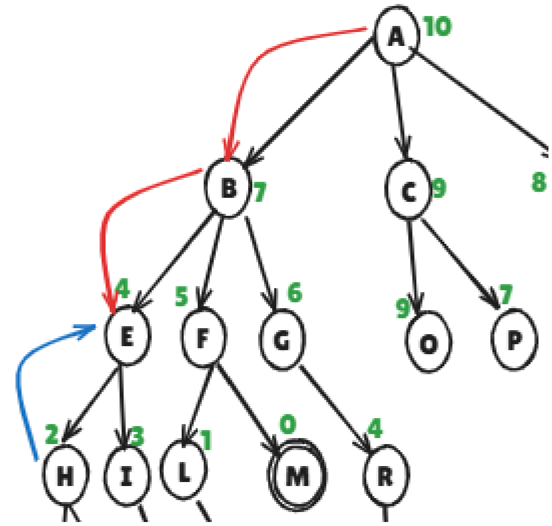
Any move is **worse** than 2.

Suppose we randomly pick candidate **E(4)**:

- ΔE = 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

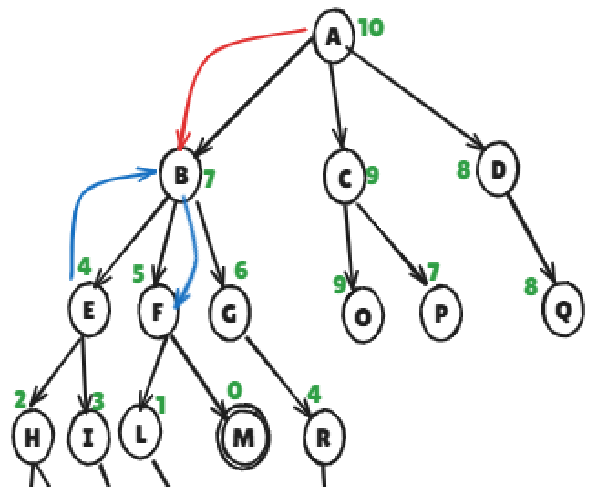
This is exactly where hill climbing would **never move**, but SA does.

◆ **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**

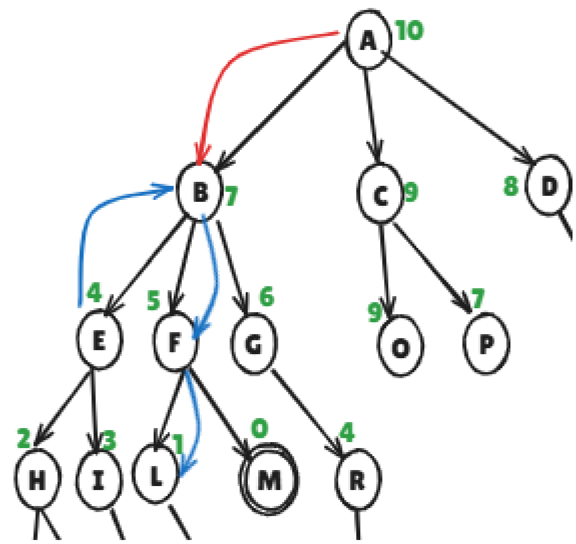
Neighbors: B(7), L(1), M(0)

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)

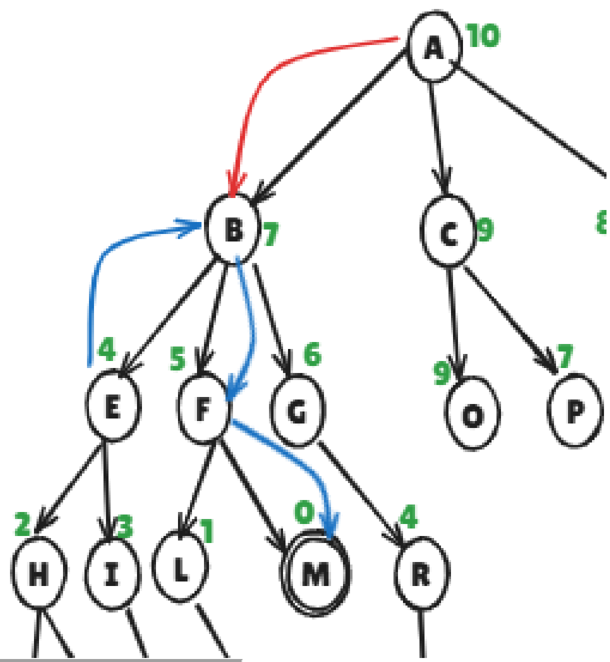
or directly choose F's neighbor M next.

Let's jump to F → M for simplicity:

From **F(5)** → pick **M(0)**:

- ΔE = 0 − 5 = **−5** → better → **accept**

✅ **Goal reached: M(0)**



5 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.

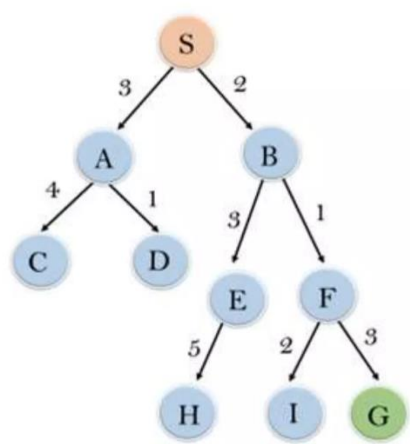
In our example, starting from A, SA moves to B, E, H, then accepts a worse move back to E and then to F due to non-zero temperature, and from there reaches L and finally the global optimum M (h = 0).
This shows how Simulated Annealing can escape local optima where hill climbing fails.

2.2.3.3 Best-First Search/Greedy 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 "best-looking" path.

When Best-First Search uses **only h(n)** to choose the next node, it is specifically called:

Greedy Best-First Search



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

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.

🔷 Key Idea

- Maintain an **OPEN list** of frontier nodes
- At each step, pick the node with the **lowest heuristic value h(n)**
- Expand it and update OPEN
- Stop when the goal is selected for expansion

It works like "follow the direction that *looks* best".

🔷 Evaluation Function

Greedy Best-First Search uses:

$$f(n) = h(n)$$

Where:

- **h(n)** = heuristic estimate of the distance or cost from node **n** to the **goal**

🔷 Example (Simple Real-Life Idea)

Imagine you want to go to **Lumbini** from your home and you have no map, but you can **see the direction**.

You simply follow the direction that seems **closest** to Lumbini (straight-line estimate).
This may lead through:

- a shortcut
- or a dead end
- or a longer route
- but you always pick the direction that **looks nearest**

This is exactly **Greedy Best-First Search**.

🔷 Advantages

✓ Very fast
✓ Low memory usage
✓ Works well when heuristic is accurate
✓ Good for simple, goal-directed searches

🔷 Disadvantages

✗ **Not optimal** – may choose a path that "looks good" but is wrong
✗ **Can get stuck** in loops or dead ends
✗ Highly dependent on heuristic accuracy
✗ Not complete for infinite spaces

🔷 Where It Is Used

- **GPS navigation** (early systems)
- **Robot path planning**
- **Puzzle solving** (8-puzzle heuristics)
- **Web crawling** (most relevant link first)
- **Greedy routing in networks**

🔷 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**.

MCQ
1. Which of the following best defines Artificial Intelligence? A. Study of human biology B. Making machines simulate human intelligence C. Programming languages for data structures D. Creating mechanical devices only
2. A problem-solving agent must first: A. Know the solution B. Choose the goal state C. Formulate a problem D. Reduce the search tree
3. Which component is NOT part of a problem specification? A. Initial state B. Goal test C. Cost function D. CPU speed
4. State Space refers to: A. All possible configurations of states B. Only terminal nodes C. Memory of search algorithms D. Randomly selected states
5. The 8-Puzzle Problem is an example of: A. Production System B. Constraint Satisfaction C. State Space Search D. Forward Chaining
6. In Water Jug Problem, legal operations include: A. Adding new jug B. Filling, emptying, pouring

C. Mixing water with chemicals
D. Boiling water

7. Problem Reduction aims to:
A. Combine multiple problems
B. Convert a problem into a simpler form
C. Increase difficulty of search
D. Reduce memory of algorithm

8. A Production System consists of:
A. States, rules, control strategy
B. Only program counters
C. Sensors and actuators
D. A* and heuristic functions

9. BFS expands nodes:
A. Based on depth limit
B. Level by level
C. From goal to root
D. Randomly

10. BFS uses which data structure?
A. Stack
B. Queue
C. Priority Queue
D. Linked List

11. DFS explores:
A. Widest branch first
B. Deepest node first
C. Only leaf nodes
D. Only root nodes

12. DFS uses which data structure?
A. Stack
B. Queue
C. Heap
D. Tree

13. A limitation of DFS is:
A. Always optimal
B. May get stuck in infinite paths
C. Requires high memory
D. Uses heuristic knowledge

14. BIDIRECTIONAL search works by:

A. Searching forward only
B. Searching backward only
C. Searching from both start and goal
D. Eliminating the goal state

15. Beam Search keeps:
A. Only one best node
B. Fixed number of best nodes
C. All explored nodes
D. Only root node

16. Beam width determines:
A. Storage of states
B. Number of nodes kept at each level
C. Search depth
D. Heuristic accuracy

17. Heuristic value h(n) estimates:
A. Cost from start to goal
B. Cost from node n to goal
C. Exact path cost
D. CPU processing time

18. A heuristic must be:
A. Negative
B. Arbitrary
C. Admissible for A*
D. Exactly equal to real cost

19. Hill Climbing selects:
A. Neighbor with highest f(n)
B. Neighbor with random value
C. Neighbor with best heuristic value
D. Parent node always

20. Hill Climbing often gets stuck in:
A. Global optimum
B. Memory overflow
C. Local maxima or plateaus
D. Infinite BFS queue

21. Simulated Annealing sometimes accepts worse moves to:
A. Reduce memory
B. Escape local maxima
C. Avoid goal state
D. Increase search depth

22. Probability of accepting uphill moves in Simulated Annealing depends on:
A. BFS limit
B. Temperature T
C. Node depth
D. CPU frequency

23. Greedy Best-First Search chooses node with:
A. Lowest g(n)
B. Lowest f(n)=g(n)+h(n)
C. Lowest heuristic h(n)
D. Highest heuristic h(n)

24. Disadvantage of Greedy Search is:
A. Uses memory
B. Always optimal
C. May follow deceptive heuristic
D. Only works for trees

25. In A Search, evaluation function is:*
A. f(n)=h(n)
B. f(n)=g(n)
C. f(n)=g(n)+h(n)
D. f(n)=g(n)-h(n)

26. A Search is optimal only if heuristic is:*
A. Arbitrary
B. Non-admissible
C. Admissible
D. Negative

27. OPEN list in A contains:*
A. Already visited nodes
B. Frontier nodes
C. Only leaf nodes
D. Root node only

28. CLOSED list in search algorithms stores:
A. Nodes not yet seen
B. Expanded nodes
C. All goal nodes
D. Only heuristic values

29. Numerical BFS: If OPEN = [A, B, C], next expanded is:
A. C
B. B

C. A
D. Last inserted node

30. Numerical DFS: If children generated = [B, C, D], next expanded is:
A. B
B. C
C. D
D. Parent node

31. In a numerical Greedy search, OPEN contains: A(5), B(2), C(7). Next expanded:
A. A
B. B
C. C
D. Depends on DFS depth

32. The 8-puzzle legal moves depend on:
A. Tile shapes
B. Blank position
C. Node depth
D. Temperature value

33. In Water Jug Problem, goal test checks:
A. Depth of tree
B. Jug structure
C. Amount of water
D. Heuristic value

34. Problem Reduction solves problems by:
A. Removing goal state
B. Breaking into subproblems
C. Adding more states
D. Using BFS only

35. In production systems, rules are applied based on:
A. Heuristic only
B. Control strategy
C. Numerical order
D. DFS limit

36. Beam search often fails because:
A. It keeps too many nodes
B. It discards useful nodes
C. It uses recursion
D. It never uses heuristics

37. Bidirectional search is most effective when:

A. Goal state is unknown
B. Branching factor is small
C. Search space is enormous
D. Both start and goal states are large

38. Greedy Search may fail to find:
A. Local optimum
B. Global optimum
C. Successor nodes
D. Parent nodes

39. A Search avoids expanding nodes with:*
A. Highest h(n)
B. Highest f(n)
C. Lowest g(n)
D. Equal f(n) values

40. In heuristic search, admissibility ensures:
A. h(n) never underestimates
B. h(n) never overestimates
C. g(n) always decreases
D. BFS is always optimal

✅ Answer Key (40 MCQs)

1. C	16.B	31.B
2. D	17.B	32.B
3. A	18.C	33.C
4. C	19.C	34.B
5. B	20.C	35.B
6. B	21.B	36.B
7. A	22.B	37.C
8. B	23.C	38.B
9. B	24.C	39.B
10.B	25.C	40.B
11.A	26.C	
12.C	27.B	
13.C	28.B	
14.C	29.C	
15.C	30.C	

Reasoning:

1. Which of the following best defines Artificial Intelligence?

Answer: B – Making machines simulate human intelligence
Because AI is about building systems that can **act or think intelligently**, like humans.

2. A problem-solving agent must first:
Answer: C – Formulate a problem
In Russell & Norvig's model, the agent **first formulates the problem**, then searches, then executes.

3. Which component is NOT part of a problem specification?
Answer: D – CPU speed
Problem specification needs **initial state, actions, goal test, path cost** – CPU speed is implementation detail, not part of the problem itself.

4. State Space refers to:
Answer: A – All possible configurations of states
State space = **set of all states** reachable from the initial state using actions.

5. The 8-Puzzle Problem is an example of:
Answer: C – State Space Search
Every tile arrangement is a **state**, and solving is moving through the **state space**.

6. In Water Jug Problem, legal operations include:
Answer: B – Filling, emptying, pouring
These are exactly the **operators** that generate new states.

7. Problem Reduction aims to:
Answer: B – Convert a problem into a simpler form
We break a complex problem into **simpler subproblems** and solve them.

8. A Production System consists of:
Answer: A – States, rules, control strategy
Production system = **set of rules (IF–THEN)** + **working memory (states)** + **control strategy**.

9. BFS expands nodes:
Answer: B – Level by level
Breadth-First Search visits all nodes at **depth d** before going to **depth d+1**.

10. BFS uses which data structure?
Answer: B – Queue
BFS frontier is managed in **FIFO** order, so a **queue** is used.

11. DFS explores:
Answer: B – Deepest node first
It follows one branch **down to depth** before backtracking.

12. DFS uses which data structure?
Answer: A – Stack
DFS is **LIFO**, implemented via an explicit stack or recursion stack.

13. A limitation of DFS is:
Answer: B – May get stuck in infinite paths
On **infinite or cyclic** graphs, DFS can keep going forever on one branch.

14. BIDIRECTIONAL search works by:

Answer: C – Searching from both start and goal
It runs two searches that **meet in the middle**, reducing complexity.

15. Beam Search keeps:
Answer: B – Fixed number of best nodes
Beam width **k** = maximum number of nodes kept at each level.

16. Beam width determines:
Answer: B – Number of nodes kept at each level
Larger width = more nodes retained; smaller width = more aggressive pruning.

17. Heuristic value h(n) estimates:
Answer: B – Cost from node n to goal
By definition, **h(n)** is an estimate of **distance/effort remaining** to the goal.

18. A heuristic must be:
Answer: C – Admissible for A*
For A* to be optimal, h(n) must **never overestimate** true remaining cost (admissible).

19. Hill Climbing selects:
Answer: C – Neighbor with best heuristic value
It greedily moves to the **neighbor that looks best** (lowest h if minimizing).

20. Hill Climbing often gets stuck in:
Answer: C – Local maxima or plateaus
Because it never moves to worse states, it can't escape **local optima** or flat regions.

21. Simulated Annealing sometimes accepts worse moves to:
Answer: B – Escape local maxima
The probability of accepting a **worse** move lets it jump out of local traps.

22. Probability of accepting uphill moves in Simulated Annealing depends on:
Answer: B – Temperature T
Formula $P = e^{-\Delta E/T}$: higher **T** → higher chance of accepting bad moves.

23. Greedy Best-First Search chooses node with:
Answer: C – Lowest heuristic h(n)
Greedy Best-First uses $f(n) = h(n)$and chooses **minimum h**.

24. Disadvantage of Greedy Search is:
Answer: C – May follow deceptive heuristic
Because it only looks at h(n), it can be **misled** and miss the optimal path.

25. In A* Search, evaluation function is:
Answer: C – f(n) = g(n) + h(n)
A* adds **cost so far** (g) + **estimated remaining cost** (h).

26. A* Search is optimal only if heuristic is:
Answer: C – Admissible
Admissible h(n) (never overestimating) is the key condition for **optimality**.

27. OPEN list in A* contains:
Answer: B – Frontier nodes
OPEN = nodes **generated but not yet expanded** (the frontier).

28. CLOSED list in search algorithms stores:
Answer: B – Expanded nodes
Once a node is fully expanded, it's put in **CLOSED** so we don't re-expand it.

29. Numerical BFS: If OPEN = [A, B, C], next expanded is:
Correct Answer: C – A
Reason: BFS uses a **queue (FIFO)**; the **first element** in OPEN, A, is expanded next.
(I previously gave the wrong option for this one; this is the corrected answer.)

30. Numerical DFS: If children generated = [B, C, D], next expanded is:
Answer: C – D
DFS uses **stack (LIFO)**, so the **last generated child** D is expanded first.

31. In a numerical Greedy search, OPEN contains: A(5), B(2), C(7). Next expanded:
Answer: B – B
Greedy chooses the node with **smallest h**, here **2**.

32. The 8-puzzle legal moves depend on:
Answer: B – Blank position
Only the **blank tile's location** determines which moves (up/down/left/right) are legal.

33. In Water Jug Problem, goal test checks:
Answer: C – Amount of water
We ask: "Does some jug contain the **desired quantity** of water?"

34. Problem Reduction solves problems by:
Answer: B – Breaking into subproblems
We reduce a big problem into **smaller subproblems** whose solutions combine.

35. In production systems, rules are applied based on:
Answer: B – Control strategy
When several rules match, the **control strategy** decides which rule to fire.

36. Beam search often fails because:
Answer: B – It discards useful nodes
Keeping only k best nodes can **throw away** nodes on the true optimal path.

37. Bidirectional search is most effective when:
Answer: C – Search space is enormous
Because it reduces time from $O(b^d)$to roughly $O(b^{d/2})$, which is a big gain in **huge** spaces with known start & goal.

38. Greedy Search may fail to find:
Answer: B – Global optimum
It can get a **suboptimal solution** because it only looks locally at h(n).

39. A* Search avoids expanding nodes with:
Answer: B – Highest f(n)
At each step, A* expands the node with **minimum f(n)**; nodes with high f are postponed.

40. In heuristic search, admissibility ensures:
Answer: B – h(n) never overestimates
An admissible heuristic is **optimistic**: $h(n) \leq h^*(n)$(true remaining cost).

2.2.3.4 Branch and Bound Search (NIC)

Branch and Bound is a **systematic search strategy** used mainly for **optimization problems** (e.g., shortest path, TSP, knapsack).
The idea is to **explore branches of a search tree**, but **cut off (bound)** any branch that **cannot lead to a better solution** than the best one found so far.

1 Real-life example: courier finding the cheapest route

A courier in **Butwal (Root)** must deliver a parcel to **Goal G** and has several possible routes:

- Left side roads → usually longer
- Middle roads → medium
- Right side roads → might be very good or very bad

Each *partial route* has a **lower bound** (minimum possible fuel cost if we continue along that route).
The courier wants the **cheapest route**.

Branch = try extending one partial route
Bound = if even the *best possible* cost on this route is already **worse** than a route we've found, we **cut (prune)** that branch.

So instead of trying *every* path, we:

- Keep a **current best found route** (incumbent cost)
- For each new partial route, compute a **bound**
- If bound ≥ best_cost → **don't waste time** on that subtree

Branch and Bound is a systematic search technique for optimization problems that explores a solution tree by branching on partial solutions and uses bounds to prune branches that cannot lead to a better solution than the best found so far.

◆ Key Idea

1. **Branch**
   - Generate/expand partial solutions (nodes) → these form a **tree**.
2. **Bound**
   - For each node, compute a **bound**: a lower/upper estimate of the best solution reachable from that node.

- If this bound is **worse** than the **current best solution**, **prune** (discard) that branch.

So search is focused only on **promising branches**, skipping large portions of the tree.

🔷 Terms

- **Current Best (incumbent):**
  Best complete solution found so far, with cost BestCost.
- **Bound:**
  Estimated best possible cost from a partial solution.
  - For **minimization** problems: if Bound ≥ BestCost → prune.
- **Node:**
  Represents a **partial solution** (e.g., partial tour in TSP, partial assignment in knapsack).

🔷 Simple Example (Minimization Idea)

Imagine TSP with 4 cities (A, B, C, D).
We build tours step by step:

- Start at A
- Branch to partial tours: A→B, A→C, A→D
- For each partial tour, compute a **lower bound** on the final tour cost (e.g., using minimum outgoing edges).
- If for some partial tour A→C the bound is already **greater than** the cost of a complete tour found via A→B→D→C→A,
  then any extension of A→C **cannot** be better → **prune** A→C branch.

🔷 Basic Algorithm Sketch

1. Initialize BestCost = ∞, BestSol = null.
2. Put root node (empty or start solution) into a list/queue.
3. While there are nodes left:
   - Select a node N to expand (using some strategy: DFS-like, BFS-like, priority queue, etc.).
   - If N is a **complete solution** and its cost < BestCost, update BestCost and BestSol.
   - Else:
     - Compute **bound(N)**.
     - If bound(N) < BestCost, **branch**: generate children (extended partial solutions) and add them to the list.

- If bound(N) ≥ BestCost, **prune** N (don't expand).

◆ Applications

- **Travelling Salesman Problem (TSP)**
- **Knapsack problem**
- **Job scheduling / assignment** problems
- **Integer programming**
- Any **combinatorial optimization** problem.

◆ Advantages

- Can dramatically **reduce the search space** using good bounds.
- Guarantees **optimal solution** (if bounds are correct).
- Systematic and complete.

◆ Disadvantages

- Performance depends heavily on:
  - Quality of **bounds**
  - Order of exploring branches (how quickly a good incumbent is found)
- In worst case, still **exponential time**.
- More complex to implement than simple DFS/BFS.

Example

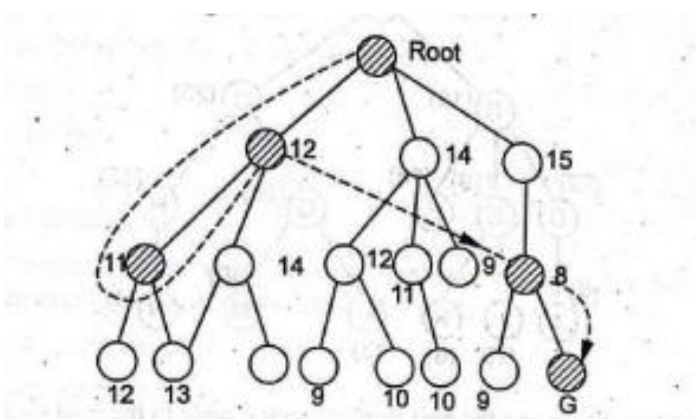
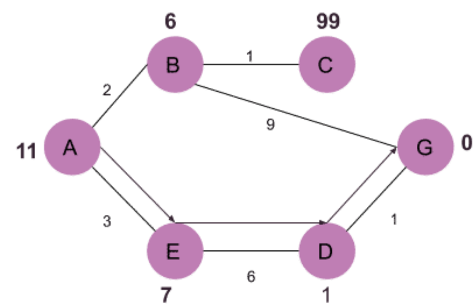


Fig. 2.18 Branch-and-bound search.

2.2.3.5 A* Algorithm

⭐ A* (A-star) is an **informed search algorithm** that finds the **optimal (least-cost) path** by combining:

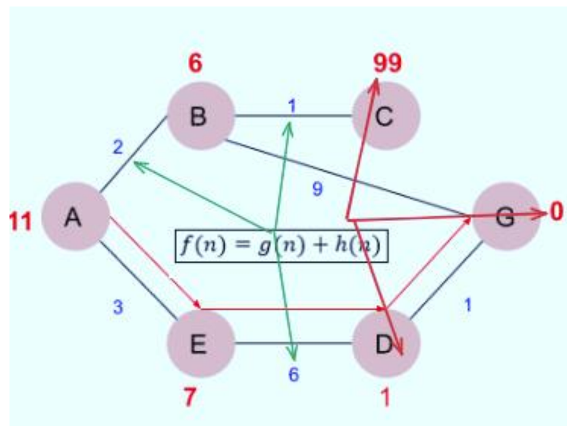


- **g(n):** cost from the start node to node *n*
- **h(n):** heuristic estimate of cost from node *n* to the goal

A* selects the node with the **minimum value of:**

$$\boxed{f(n) = g(n) + h(n)}$$

- **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:**

$$h(n) \leq \text{true cost from n to goal}$$

(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:
   - Pick node with smallest **f(n)** from OPEN
   - Move it to CLOSED
   - If it is the goal → success
   - Else expand node, compute f(n) for children
   - If child is new or cheaper → update and push to OPEN
3. Return optimal path

⭐ Advantages of A*

- Guaranteed **optimal solution** (if heuristic is admissible)
- Efficient: explores fewer nodes than BFS/Uniform-Cost
- Works very well for real-world navigation
- Combines both cost and heuristic knowledge

⭐ Disadvantages of A*

- Can be **slow** if heuristic is poor
- Requires **high memory** (large OPEN and CLOSED lists)
- Performance depends heavily on **quality of h(n)**
- Not suitable for very large state spaces without heuristic pruning

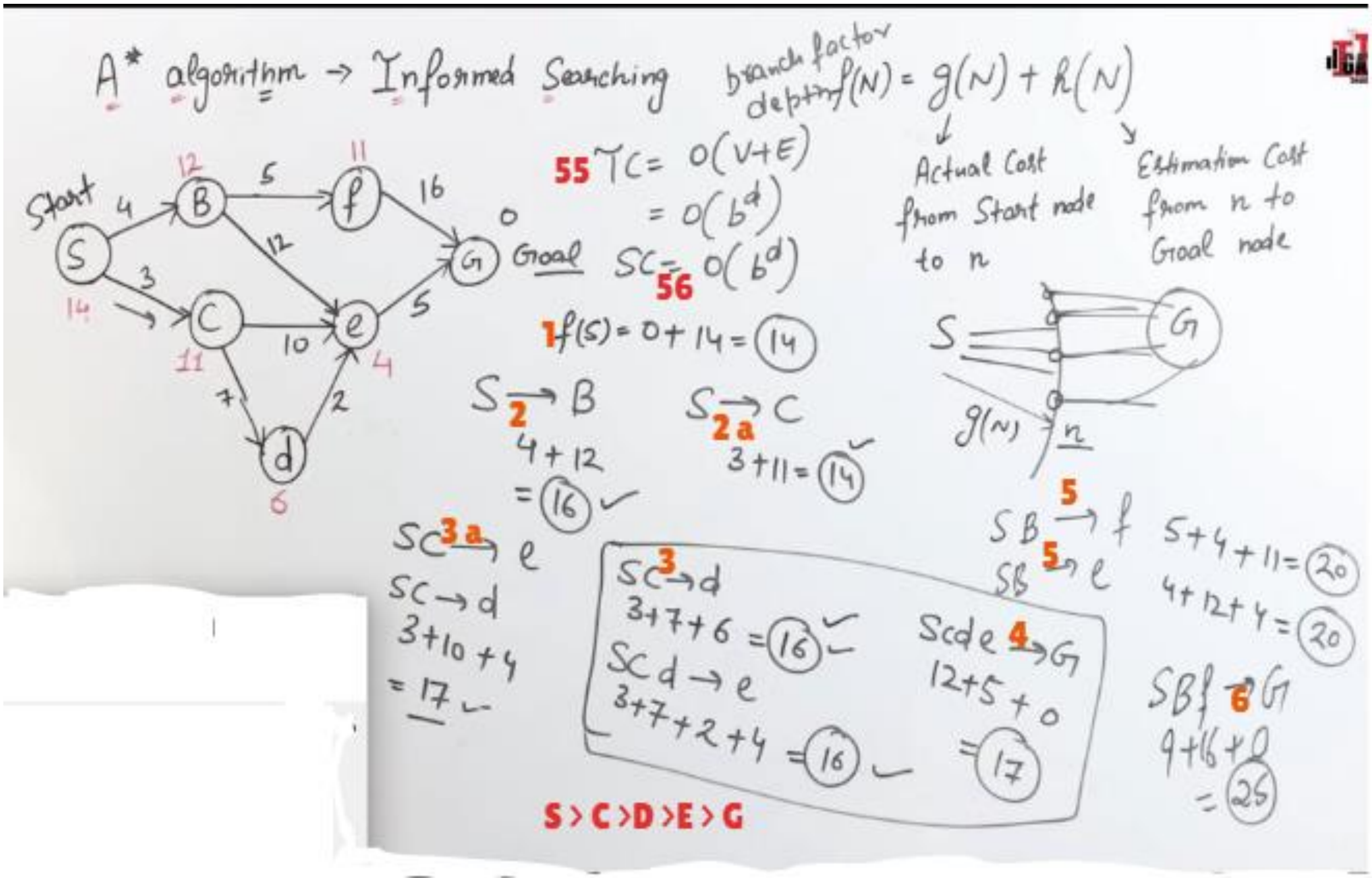
---

⭐ Where A* Is Used

- Google Maps routing
- Robot movement planning

- NPC pathfinding in games
- Network routing
- Puzzle solving (8-puzzle, 15-puzzle)

🚩 **Question**



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:

$$\boxed{\text{Optimal path: } S \to C \to D \to E \to G}$$

Total cost:

$$g(G) = 3 + 7 + 2 + 4 = \boxed{16}$$

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

A robot must move from **Start S** to **Goal G**. The following table shows the **edges**, their **path costs**, and **heuristic values h(n)**. Use A* with:

Question 1

Edge	Path Cost g	Heuristic h
S → A	3	1
S → B	6	2
S → C	4	3
A → D	+2	4
A → E	+5	3
B → E	+4	3
C → F	+3	2
C → G	+6	0
F → G	+2	0
E → G	+4	0

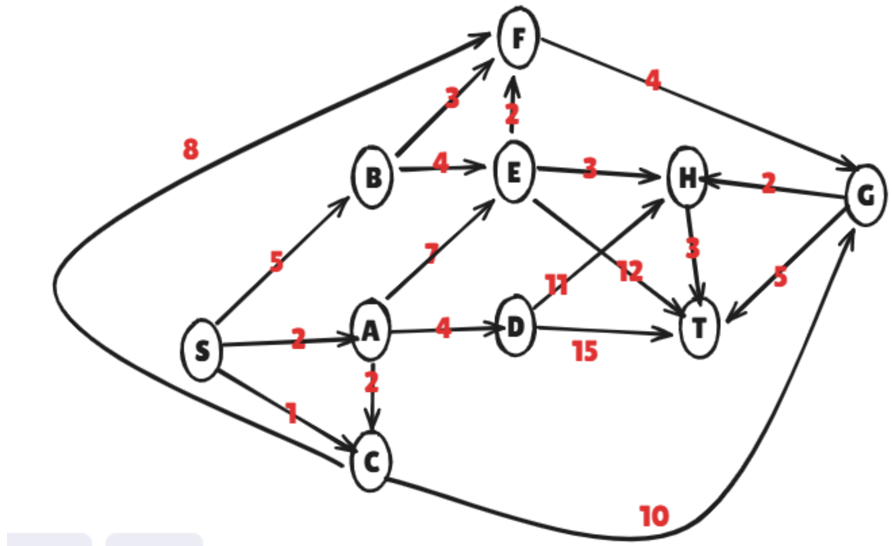
Question 2

Edge	Path Cost g	Heuristic h
S → P	2	7
S → Q	5	4
S → R	3	6
P → T	+4	5
P → U	+6	3
Q → U	+2	3
Q → V	+5	2
R → V	+4	2
U → G	+3	0
V → G	+4	0

A delivery van starts at **S** and wants the **minimum-cost path** to **city T**. Edges are **road costs**. The graph is:

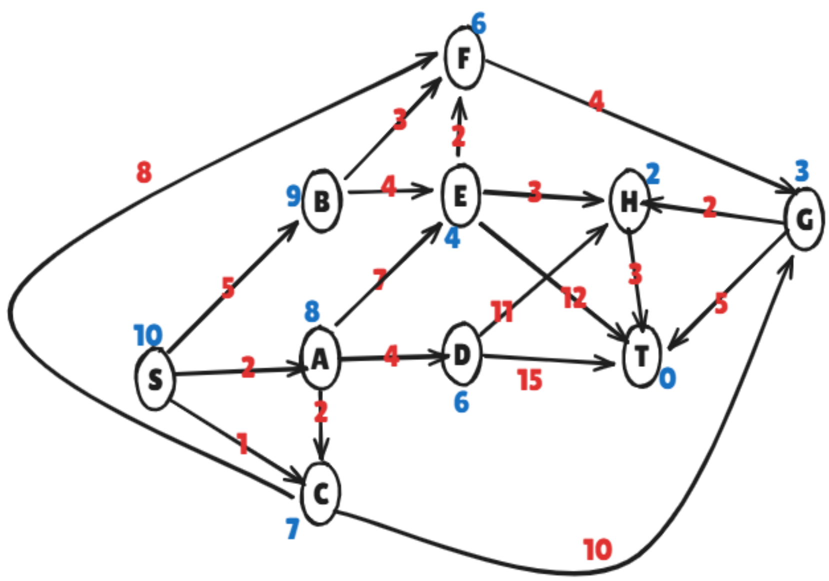
Edges (undirected, cost in brackets):

- 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

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

Sanjeev Thapa, Er. DevOps, SRE, CKA, RHCSA, RHCE, RHCSA-Openstack, MTCNA, MTCTCE, UBSRS, HEv6, Research Evangelist

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**

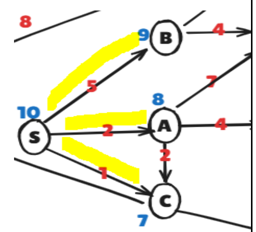
<table>
<tr>
<td>

Step 0 – Initialize

- g(S) = 0, h(S) = 10 → f(S) = 0+10=10
- OPEN = { S }, CLOSED = { }

</td>
<td>

Step 1 – Expand S



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).

</td>
</tr>
<tr>
<td>

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).

</td>
<td>

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).

</td>
</tr>
<tr>
<td>

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 }

</td>
<td>

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)

</td>
</tr>
</table>

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:

• 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) $\boxed{\text{Minimum cost } = 15}$

2.2.3.3 Best-First Search/Greedy Search

✅ **Definition : Best-First Search** is an **informed search** strategy that expands the node which appears to be **closest to the goal** according to a **heuristic function h(n)**.

The most common version is **Greedy Best-First Search**, where:

$$\boxed{f(n) = h(n)}$$

- It **does NOT** consider the path cost from start (g(n)).
- It expands the node with the **lowest heuristic value**.

This makes it fast but **not optimal**.

⭐ **Key Idea**

Greedy Search uses **heuristic only**:

- If a node "looks" closest to the goal, it is expanded first.
- Focuses on speed rather than accuracy.

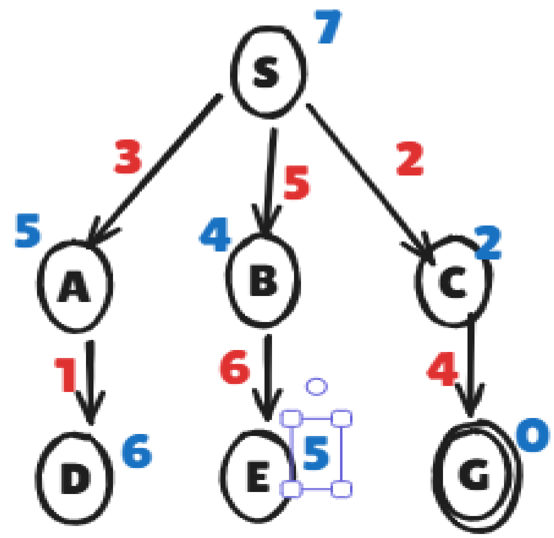
Think of it as:

➡️ "Choose the next node that seems best RIGHT NOW."

⭐ **Algorithm Steps**

1. Put the start node in OPEN.

2. While OPEN is not empty:
   - Pick the node with **minimum h(n)**
   - If it is the goal → return success
   - Else expand it and add children to OPEN
3. Add expanded nodes to CLOSED to avoid loops.



⭐ Example (Simple)

Consider the graph:

Greedy Search Execution

At **S**, children have:

- A (h=5)
- B (h=4)
- C (h=2) ⬅ **SELECTED (best)**

C expanded → G found (h=0).

Path Found:

$$\boxed{S \rightarrow C \rightarrow G}$$

⚠ Even if C→G is more costly, greedy search **ignores cost**.

⭐ Example Table (Greedy Search)

Step	**OPEN (sorted by h)**	**Selected Node**	**Reason**
1	S(7)	S	start
2	A(5), B(4), C(2)	C	lowest h
3	G(0)	G	goal reached

⭐ When is Greedy Search Good?

- When heuristics are **accurate**
- When performance (speed) is more important than correctness
- When the search space is huge

- For web crawling, route finding with good GPS heuristics, etc.

⭐ Advantages

Advantage
Very fast (low memory use)
Simple to implement
Good when heuristic is highly accurate
Explores fewer nodes compared to BFS / DFS

⭐ Disadvantages

Disadvantage
Not optimal – may give wrong shortest path
Can get stuck in local minima
Ignores actual path cost g(n)
Can fall into infinite loops without closed set

⭐ Applications

- Robot navigation
- Route/Map searching
- Network routing
- Game AI (moving toward a target)
- Simple web crawling

⭐ Comparison (Greedy vs A*)

Algorithm	**Formula**	**Optimal?**	**Notes**
Greedy	$f(n)=h(n)$	❌ No	Fast but risky
A*	$f(n)=g(n)+h(n)$	✓ Yes (if h is admissible)	Slower but optimal

---

⭐ **Numerical Question (Practice)**

Edge	Cost	h(n)
S→A	4	6
S→B	2	3
S→C	5	4
A→G	7	0
B→D	4	2
C→G	3	0
D→G	3	0

Question:

Using **Greedy Best-First Search**, find the path from **S to G**.

MCQs

1. Problem solving in AI mainly involves:
a) Finding a sequence of operators to reach a goal
b) Storing data in memory
c) Minimizing hardware usage
d) Neural network activation

2. A well-defined AI problem must include:
a) Only goal state
b) Only initial state
c) Initial state, actions, goal test, path cost
d) None

3. Which of the following is NOT a characteristic of AI problems?
a) Multiple possible solutions
b) Well-structured state space
c) Deterministic transitions always
d) Operators

4. State-space search represents:
a) The space where AI models are stored
b) The set of all states reachable from the start
c) The program files
d) Random memory

5. The 8-puzzle problem belongs to:
a) Decision tree problems
b) State-space search
c) Stochastic optimization
d) Game theory

6. In the water-jug problem, the state is:
a) Amount of water in each jug
b) Jug color
c) Jug cost
d) None

7. Missionaries and Cannibals problem is an example of:
a) Constraint Satisfaction
b) Linear regression
c) Classification
d) Clustering

8. Production rules are represented as:
a) IF <condition> THEN <action>
b) x = m + c
c) P = VI
d) None

9. Breadth-First Search uses which data structure?
a) Stack
b) Queue
c) Priority queue
d) Graph

10. BFS guarantees optimality when:
a) All edges have equal cost
b) Heuristic is admissible
c) Tree is balanced
d) Graph is small

11. Depth-First Search uses which data structure?
a) Queue
b) Stack
c) Heap
d) Linked list

12. DFS is incomplete in:
a) Graphs
b) Finite trees
c) Graphs with cycles & infinite depth
d) Heap

13. Bidirectional search stops when:
a) Both searches meet
b) Heuristic becomes zero
c) Stack is empty
d) Opposite direction fails

14. The branching factor is:
a) Number of nodes per level
b) Number of successors per node
c) Path cost
d) Memory size

15. Hill-Climbing Search uses:
a) g(n)
b) h(n)
c) f(n)=g+h
d) Random selection

16. Hill climbing may get stuck in:
a) Local maxima
b) Global minima only
c) DFS
d) None

17. Simulated annealing helps hill climbing by:
a) Never accepting bad moves
b) Sometimes accepting worse moves
c) Removing heuristic
d) BFS strategy

18. Greedy Best-First Search uses which formula?
a) f(n) = g(n)
b) f(n) = h(n)
c) f(n) = g(n) + h(n)
d) f(n) = 0

19. A uses which evaluation?*
a) h(n)
b) g(n)
c) f(n)=g+h
d) None

20. A heuristic is admissible when:
a) Overestimates cost
b) Underestimates or never overestimates cost
c) Random
d) Always zero

21. In the 8-puzzle, Manhattan distance is:
a) Number of misplaced tiles
b) Sum of row and column distances
c) Square root distance
d) Zero for all states

22. Which search is complete?
a) DFS
b) BFS
c) Hill climbing
d) Simulated annealing

23. A is optimal when:*
a) h(n) is admissible
b) g=0
c) All edges equal
d) BFS used

24. Greedy search may fail because:
a) It ignores g(n)
b) It uses g+h
c) It's BFS
d) It's complete

25. Branch and Bound search selects path with:
a) Max g(n)
b) Min g(n)
c) Min f(n)=g(n)
d) Random

26. PEAS stands for:
a) Performance, Environment, Actuators, Sensors
b) Program, Energy, Action, Storage
c) Performance, Efficiency, Algorithm, Storage
d) None

27. Rational agent selects action that:
a) Minimizes reward
b) Maximizes expected performance
c) Ignores percepts
d) Random

28. Environment that is partially observable means:
a) Agent senses everything
b) Agent senses limited data
c) Hidden states not allowed
d) None

29. BFS time complexity:
a) O(b^d)
b) O(d^b)
c) O(b+d)
d) O(1)

30. DFS space complexity:
a) O(b^d)
b) O(bd)
c) O(1)
d) O(d)

Numerical MCQs

31. For Greedy Search, choose node with minimum h(n):
Nodes: A(h=7), B(h=3), C(h=5)
a) A
b) B
c) C
d) None

32. In A, if g(A)=4 and h(A)=6, f(A)=?*
a) 4
b) 6
c) 10
d) 24

33. For Manhattan distance, tile 8 moves from (2,2) to (0,1). Distance =
a) 1
b) 2
c) 3
d) 4

34. In a graph, g(B)=5, h(B)=3. Greedy will select based on:

a) 5
b) 3
c) 8
d) 2

---

35. In hill climbing: A(h=6), B(h=3), C(h=8). Best move?
a) A
b) B
c) C
d) Stay

---

36. A chooses the node with minimum:*
a) g
b) h
c) g+h
d) random

---

37. BFS expands nodes:
a) depth-wise
b) level-wise
c) random
d) heuristic-wise

---

38. DFS expands nodes:
a) deepest first
b) shallow first
c) heuristic
d) none

---

39. Admissible heuristic property:
a) h(n) > actual cost
b) h(n) ≤ actual cost
c) h(n)=g(n)
d) h(n)=0 always

---

40. Best-first search uses:
a) Open list sorted by h
b) Stack
c) g
d) Queue

---

41. A reduces to Greedy when:*
a) g=0
b) h=0
c) f=(g+h)/2
d) infinite branching

---

42. A reduces to UCS when:*
a) g(n)=0
b) h(n)=0
c) h(n)=1
d) all edges are 0

---

43. Simulated annealing guarantees optimality when:

a) Temperature decreases slowly
b) It always accepts worse moves
c) T=0 early
d) never cools

44. In 8-puzzle, misplaced tiles heuristic is:
a) admissible
b) inconsistent
c) optimal
d) useless

45. Branch-and-bound avoids paths that:
a) Have maximum cost
b) Are below bound
c) Exceed current best bound
d) None

46. BFS memory requirement depends mainly on:
a) Depth
b) Branching factor
c) Both depth and branching factor
d) None

47. DFS may fail due to:
a) Infinite loops
b) Small memory
c) Lack of heuristics
d) All

48. Bidirectional search needs:
a) Two heuristics
b) Two simultaneous searches
c) No graph
d) Neural network

49. If heuristic overestimates actual cost, A becomes:*
a) Incomplete
b) Non-optimal
c) Faster but wrong
d) Optimal

50. The costliest search strategy in worst-case memory use is:
a) DFS
b) Greedy
c) BFS
d) Hill climbing

Answer Key (1–50)

1.a
2.c
3.c
4.b
5.b
6.a
7.a
8.a
9.b
10.a
11.b
12.c
13.a
14.b
15.b
16.a
17.b
18.b
19.c
20.b
21.b
22.b
23.a
24.a
25.c
26.a
27.b
28.b
29.a
30.b
31.b
32.c
33.c
34.b
35.b
36.c
37.b
38.a
39.b
40.a
41.a
42.b
43.a
44.a
45.c
46.c
47.a
48.b
49.b
50.c

✅ Detailed Reasoning for Each Answer

1. Problem solving = finding operator sequence → a
2. Well-defined problem requires all components → c
3. AI problems may be non-deterministic → c
4. State space = all reachable states → b
5. 8-puzzle is classic state-space search → b
6. State is water quantity → a
7. M&C uses constraints → a
8. Production rules use IF–THEN → a
9. BFS uses queue → b
10. BFS optimal with equal-cost edges → a
11. DFS uses stack → b
12. DFS fails in infinite-depth spaces → c
13. Bidirectional search stops when frontiers meet → a
14. Branching factor = successors per node → b
15. Hill climbing uses ONLY h → b
16. Hill climbing gets trapped in local maxima → a
17. SA sometimes accepts worse moves → b
18. Greedy uses f=h → b
19. A uses f=g+h → c*
20. Admissible = never overestimates → b
21. Manhattan = sum of row/column distance → b
22. BFS is complete → b
23. A optimal when h ≤ true cost → a*
24. Greedy ignores g → a
25. Branch & Bound uses minimum g → c
26. Performance, Environment, Actuators, Sensors → a
27. Rational agents maximize expected performance → b
28. Partial means limited percepts → b
29. BFS = O(b^d) → a
30. DFS = O(bd) → b
31. Minimum h=3 → B → b
32. f=4+6=10 → c
33. |2–0| + |2–1| = 3 → c
34. Greedy only uses h=3 → b
35. Hill climbing picks lowest h=3 → B
36. A picks lowest f=g+h → c*
37. BFS expands level-wise → b
38. DFS expands deepest node first → a
39. Admissible: h ≤ actual → b
40. Best-first sorted by h → a
41. If g=0, f=h → greedy → a
42. If h=0, f=g → UCS → b
43. Slow cooling guarantees SA optimality → a
44. Misplaced tiles never overestimate → admissible → a
45. B&B prunes paths exceeding current bound → c
46. BFS memory = breadth × depth → c
47. DFS infinite loops → a
48. Needs two simultaneous searches → b
49. Overestimate → non-optimal → b
50. BFS needs largest memory → c

Sanjeev Thapa, Er. DevOps, SRE, CKA, RHCSA, RHCE, RHCSA-Openstack, MTCNA, MTCTCE, UBSRS, HEv6, Research Evangelist