Unit II

Approaches of AI

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

2.2 Problem Formulation

- *2.2.1 Problem Specification*
- *2.2.2 State Space Search with examples (8-puzzle, TSP, WaterJug Problem)*
- *2.2.3 Problem Reduction*
- *2.2.4 Production System*

2.2 Searching Techniques

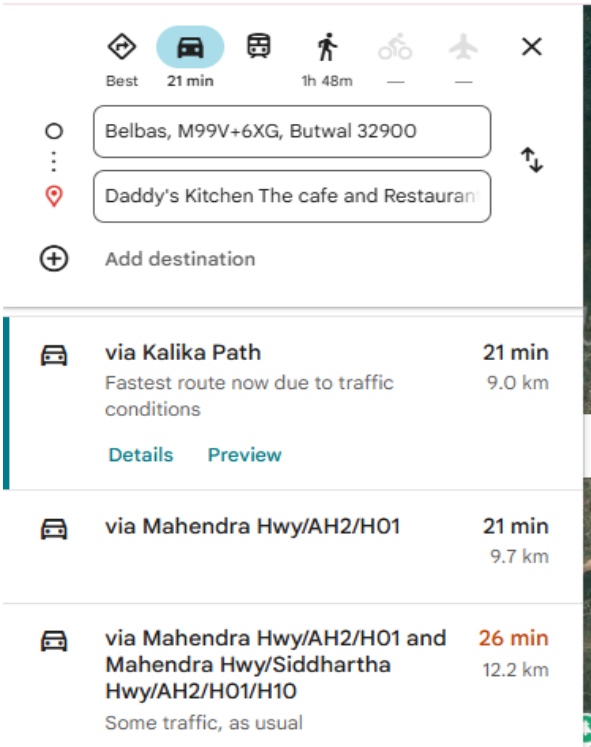
- *2.2.1 Types of Searching: Uninformed and Informed*
- *2.2.2 Breadth First Search (BFS)*
- *2.2.3 Depth First Search (DFS)*
- *2.2.4 Bidirectional Search*
- *2.2.5 Hill Climbing Search*
- *2.2.6 Simulated Annealing Search*
    - *2.2.6 Greedy Search/Best First Search*
- *2.2.7 A* Search*
- *2.3 Min-Max Algorithm*

2.4 Alpha-Beta Pruning (Cutoff)

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

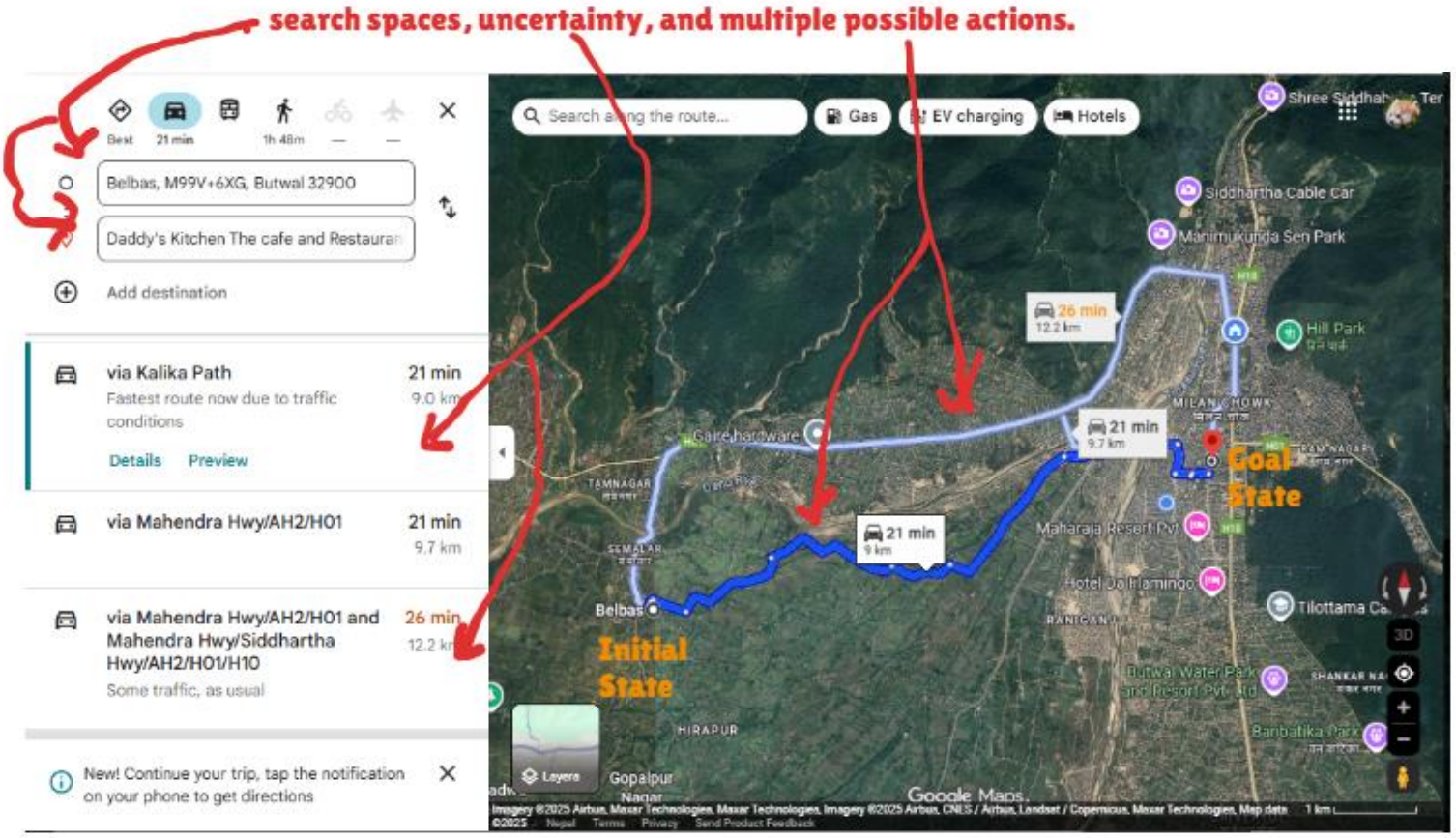
2.0 Introduction

- 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.



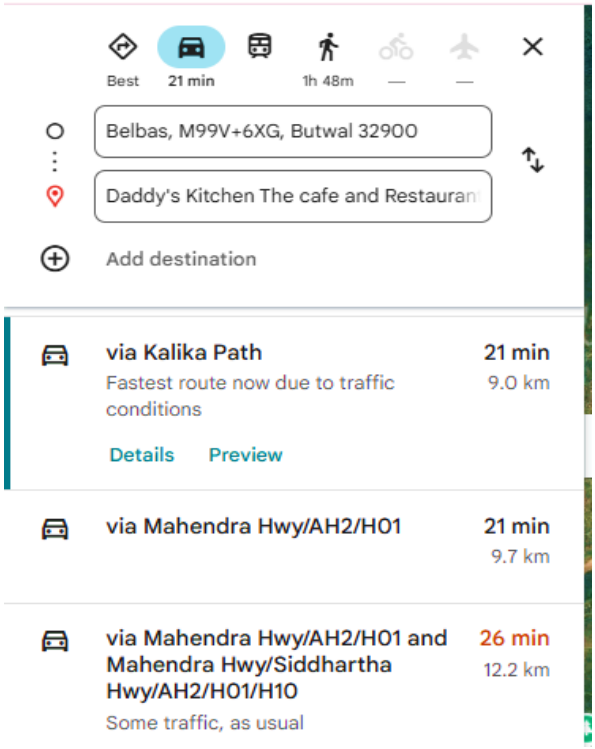
Sanjeev Thapa, Er. DevOps, SRE, CKA, RHCSA, RHCE, RHCSA-Openstack, MTCNA, MTCTCE, UBSRS, HEv6, Research Evangelist

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

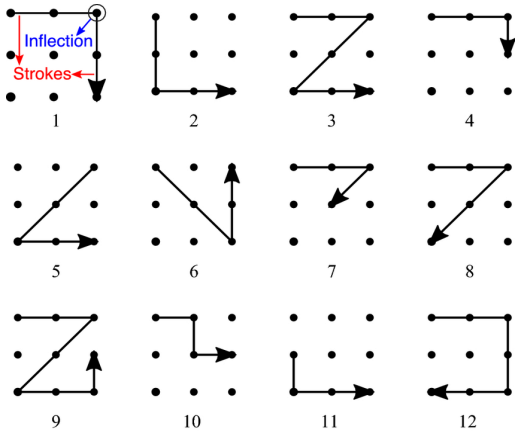
The set of all possible states reachable from the initial state.

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)

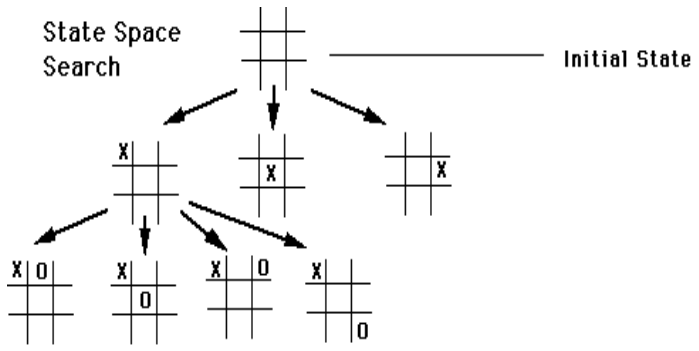


2.1.2 State Space Search with Examples

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:

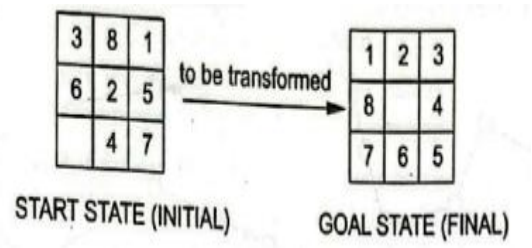
- **Nodes** = States
- **Edges** = Actions
- **Path** = Sequence of actions
- **Goal Test** = Checks if the goal state is reached



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.

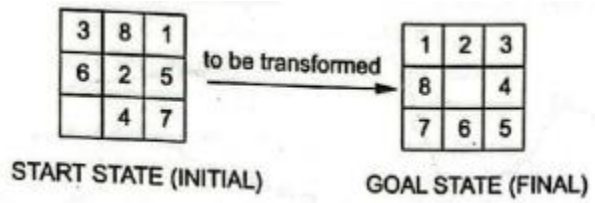


Examples of State Space Search

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 _

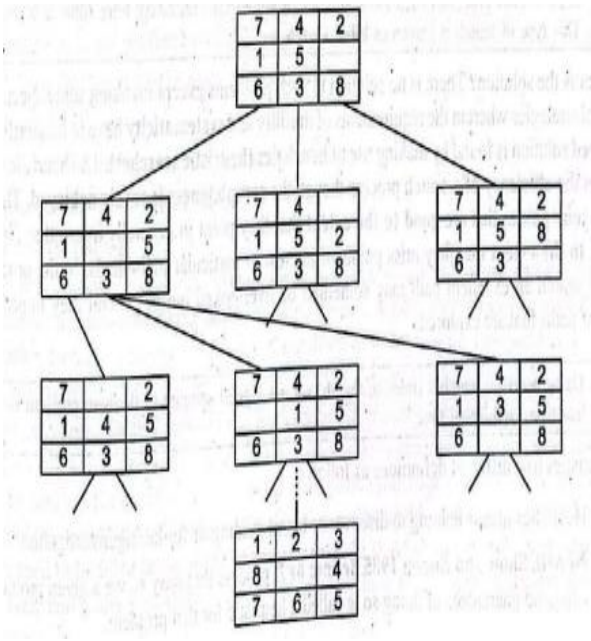
But the goal can be changed depending on the problem.

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)**:

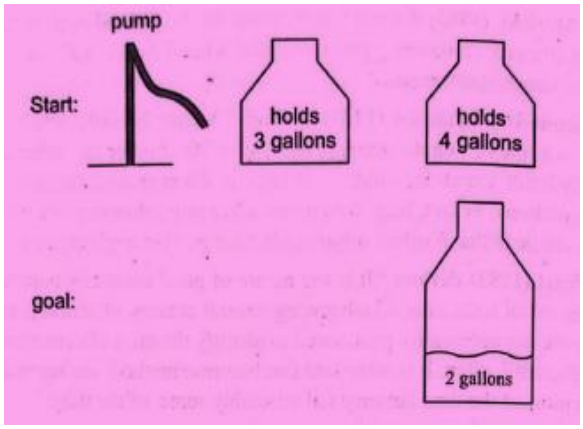
```
Initial State
    |
    |— First Move → New State
    |
    |— Second Move → Another State
    |
    … and so on
```

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)

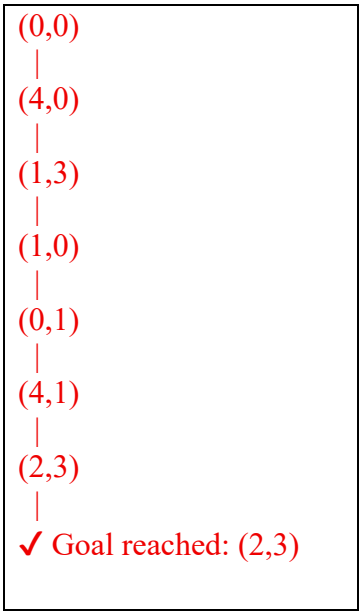
🔷 4. Legal Actions (Operators)

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)



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.

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.

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:

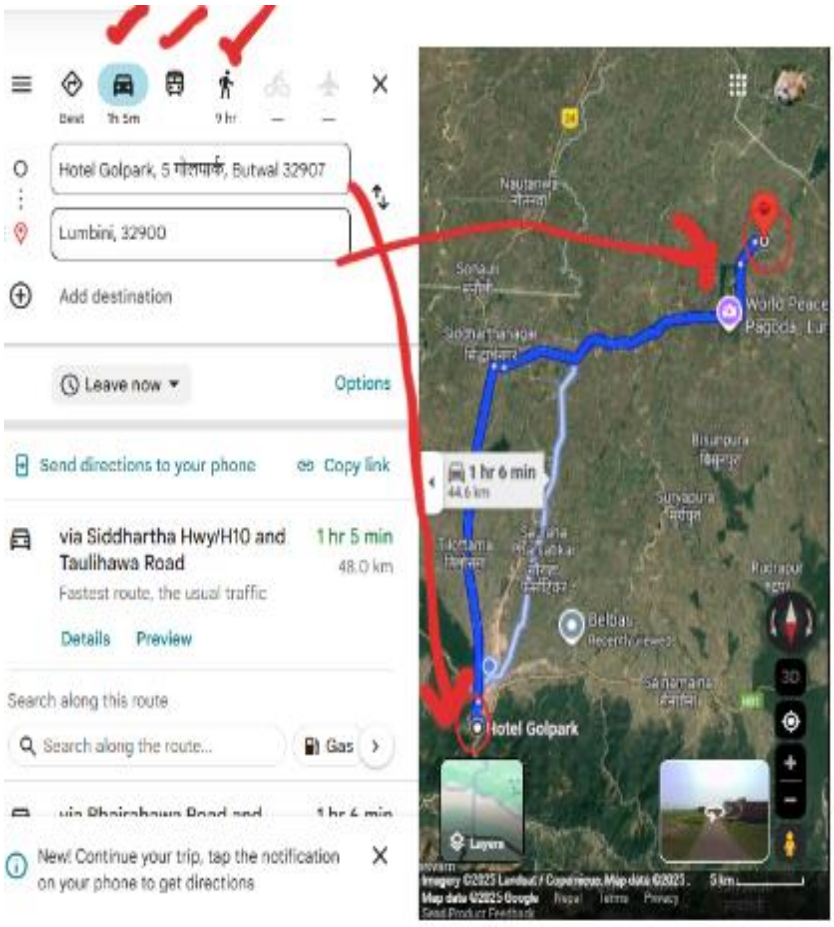
Initial State = Butwal (Golpark)

🔷 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*

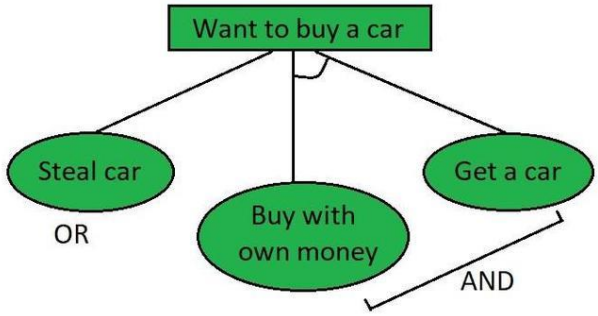
$$f(n) = g(n) + h(n)$$

- **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

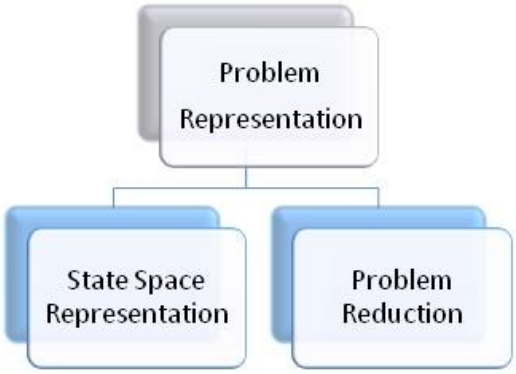
- Problem Reduction is an approach in Artificial Intelligence where a **complex problem** is broken down into a set of **smaller, simpler sub-problems**.



- 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.

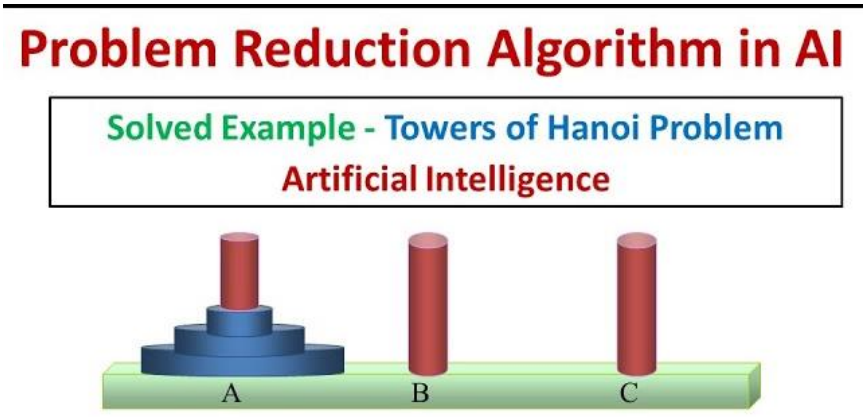
Problem Reduction = breaking a big problem → solving smaller parts → combining results → achieving the final solution.

- 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)

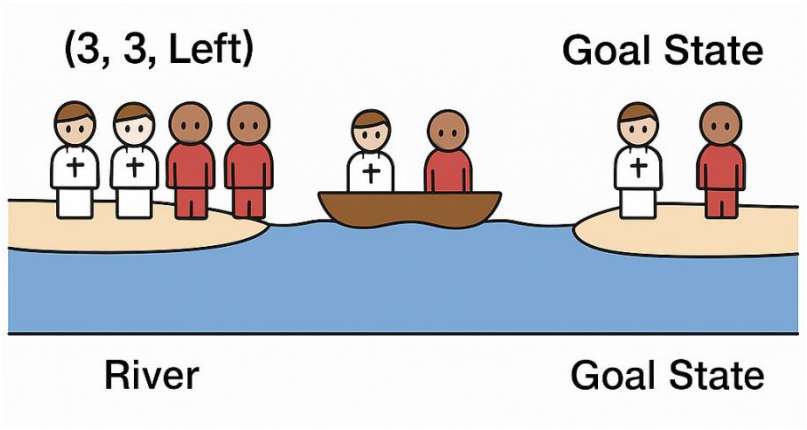
Goal State

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:

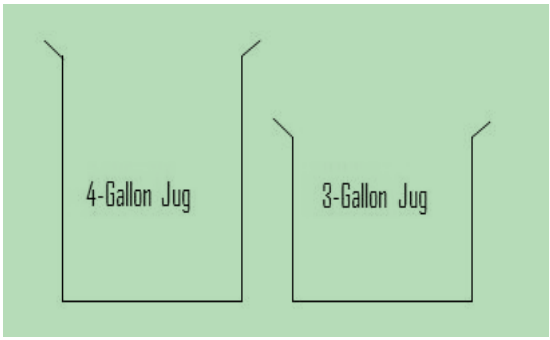
1. Plan bus to Pokhara
2. Plan hotel stay
3. Plan travel to Kathmandu
4. Plan sightseeing

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)**	◆ **DFS (Depth-First Search)**
Explores all possible states **level-by-level**	Follows one sequence deeply
Finds **shortest sequence of steps**	May find long or non-optimal solutions
Best when you want minimum moves	Uses less memory
Example: It would find the **shortest path** to reach (2,y).	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.

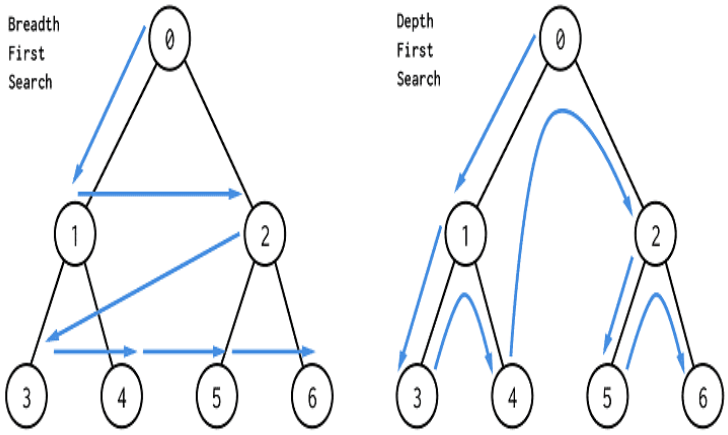
State Space Representation

- **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

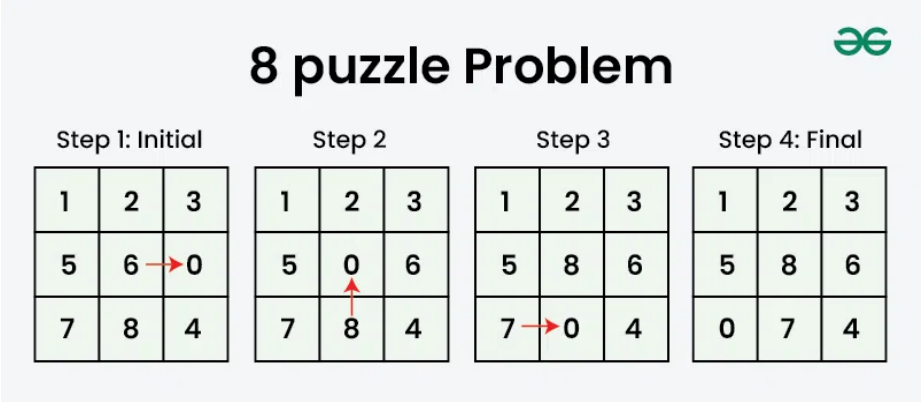
Example in 8-Puzzle:
BFS finds the minimum number of moves to reach the goal configuration.

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:

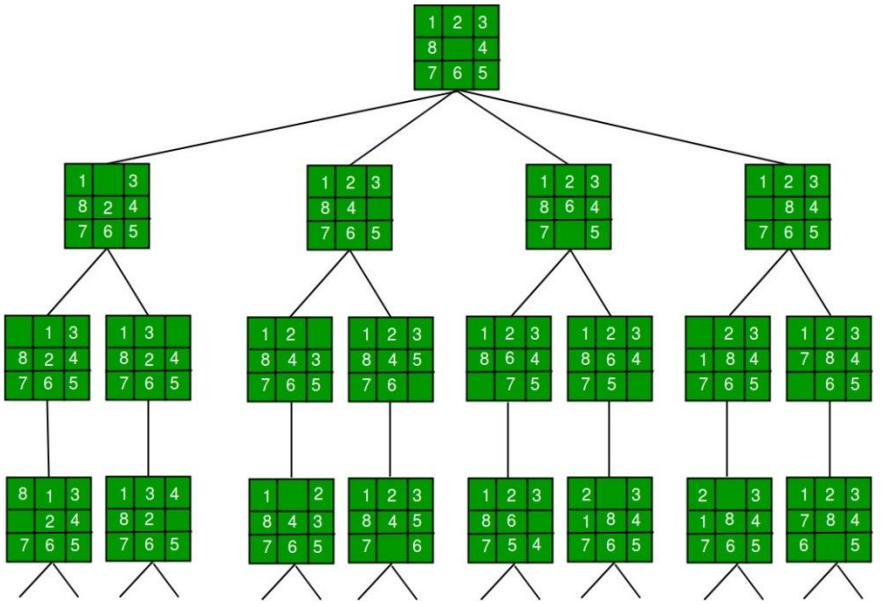
Initial State

/ \

Apply R1 Apply R2

| |

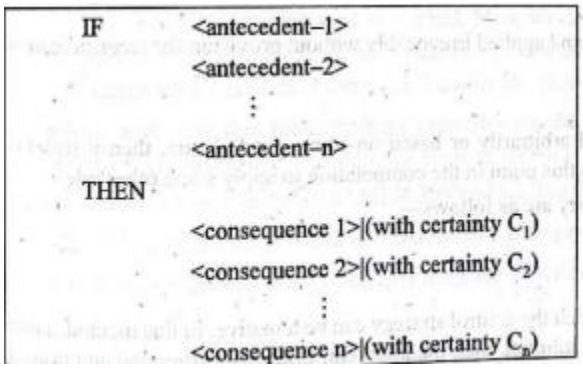
State A State B



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

🔷 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**.

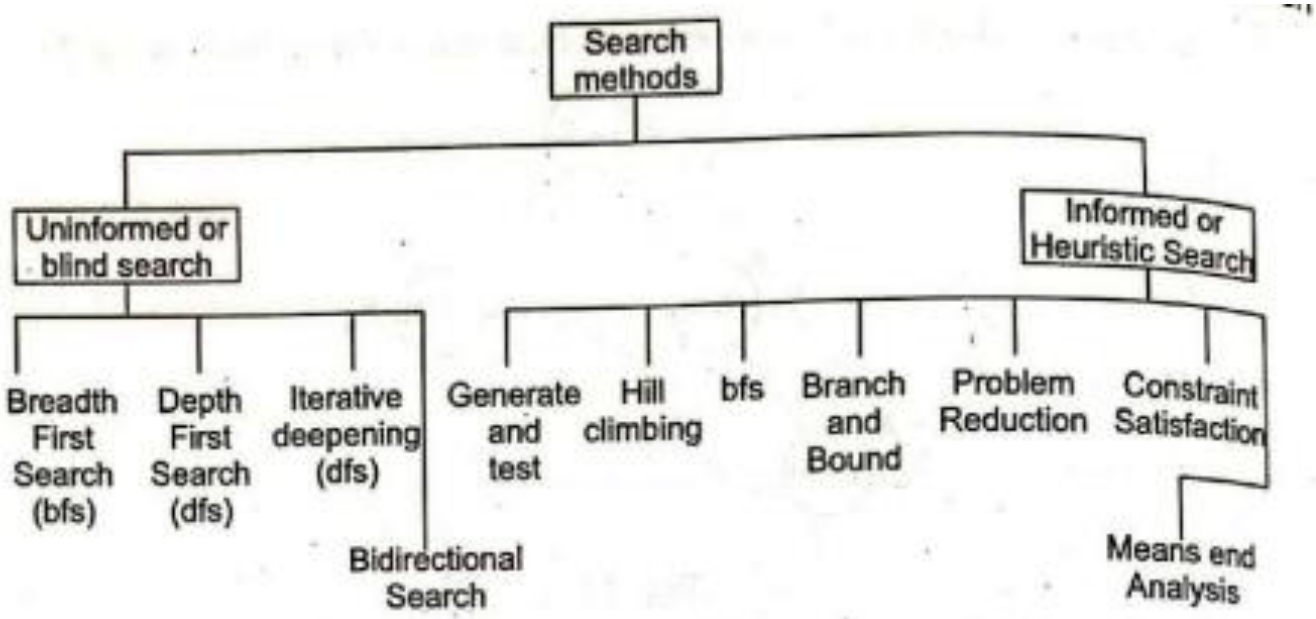
➢ An AI agent performs search when:

- 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**

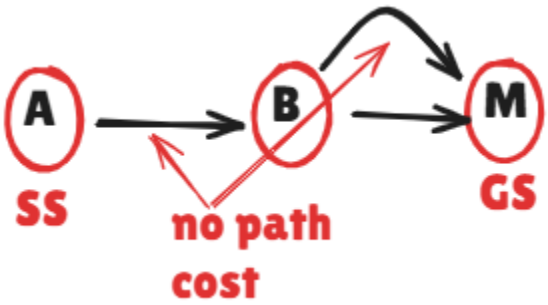
These two categories determine *how intelligently* the search tree is explored to find the goal state.

🔷 **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

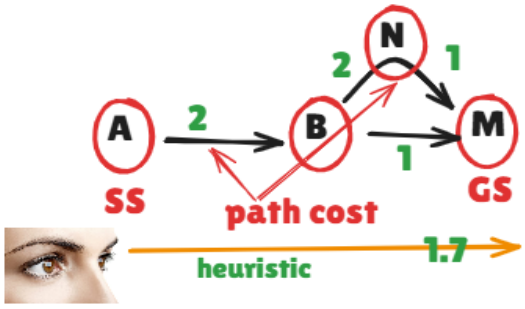
1. **Breadth-First Search (BFS)**
2. **Depth-First Search (DFS)**
3. **Depth-Limited Search (DLS)**
4. **Iterative Deepening Search (IDS)**
5. **Uniform Cost Search (UCS)**
6. **Bidirectional Search**

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

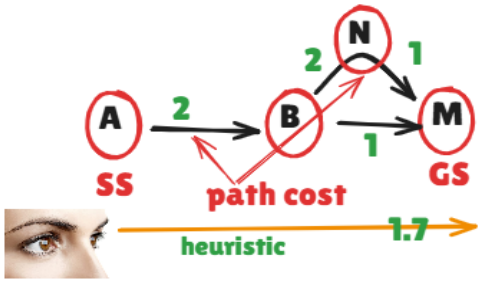
Common Informed Search Methods

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.

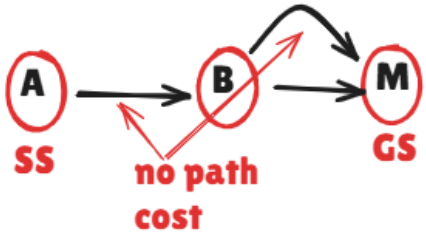
2.2.2 Uninformed/Blind/Brute

- 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

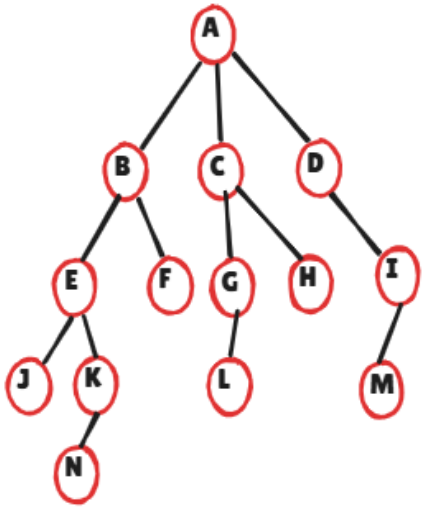
Example:
Trying all possible keys on a keychain one by one without knowing which key fits.

🔷 **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)

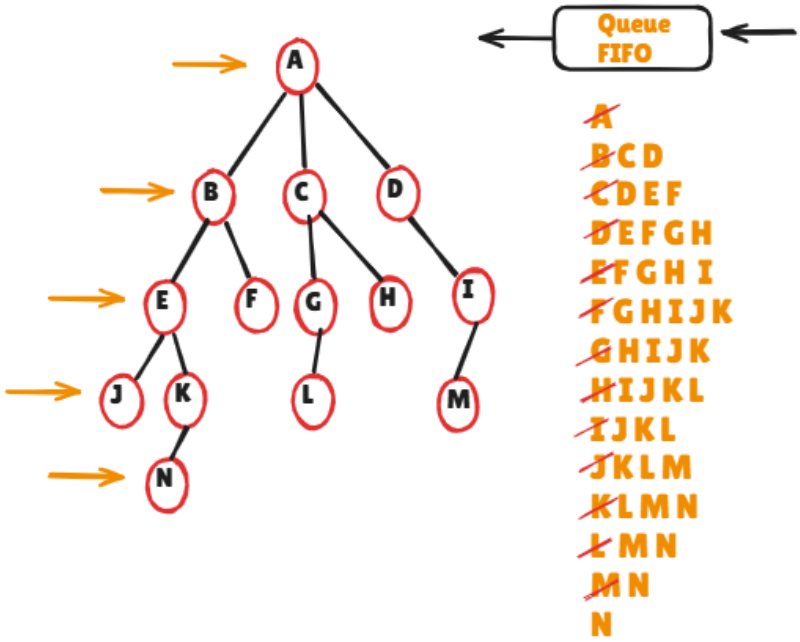


- Breadth-First Search (BFS) is an **uninformed search technique** that explores the search space **level by level**.
- It starts from the **initial state** and expands all its neighbors before moving to the next level.

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**

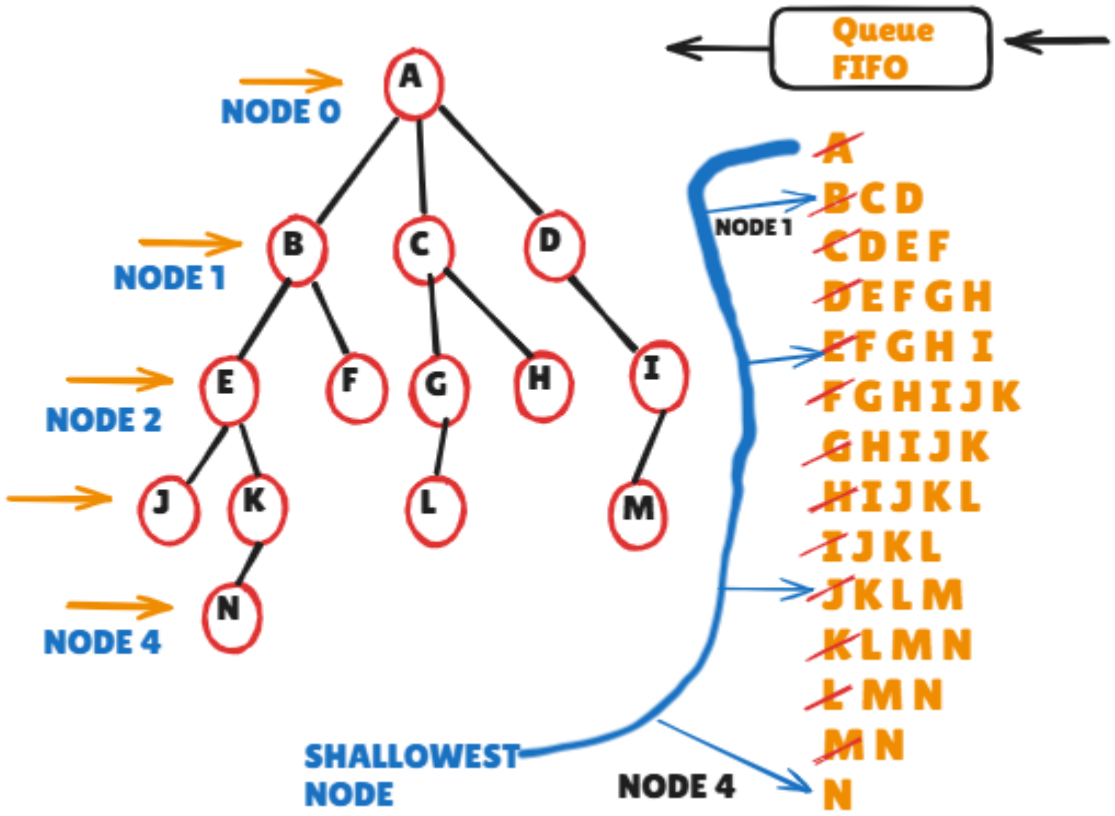
1. Start by inserting the **initial state** into a queue
2. Remove the first element (front)
3. If it is the **goal**, stop
4. Otherwise, expand all its children and add them to the **back of the queue**

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

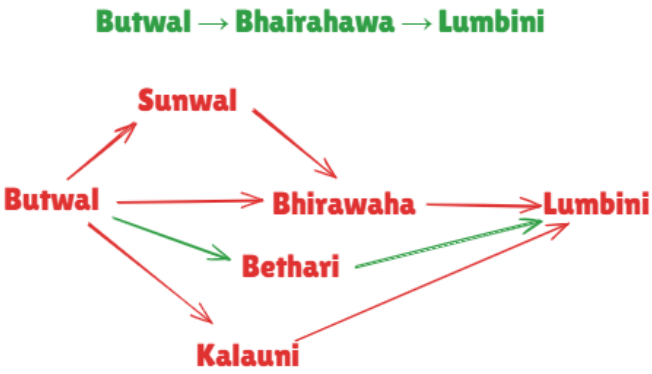


🔷 Properties of BFS

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.



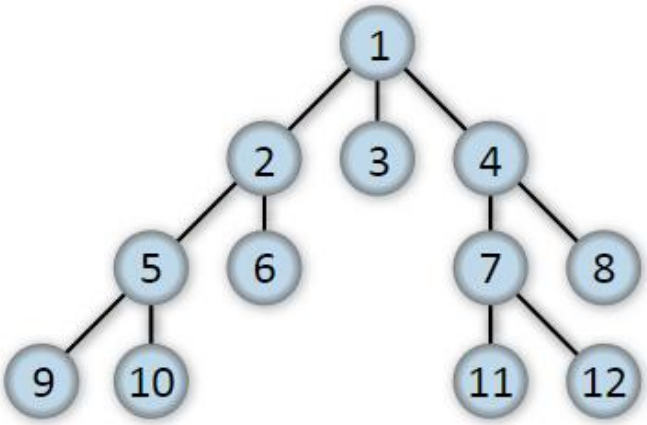
🔷 Real-Life Analogy

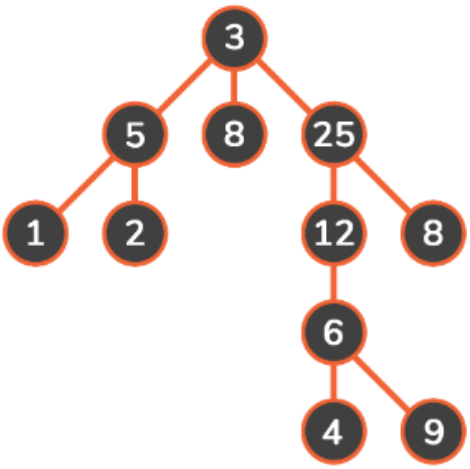
Searching for someone in a building floor-by-floor:

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

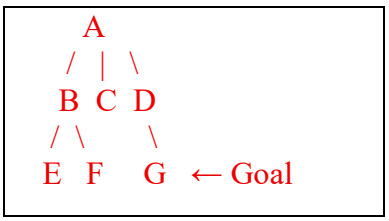
This is exactly BFS.

Classwork (Just Now)





📌 BFS Example: Simple Graph Search



We search for goal **G** starting from **A**.

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

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

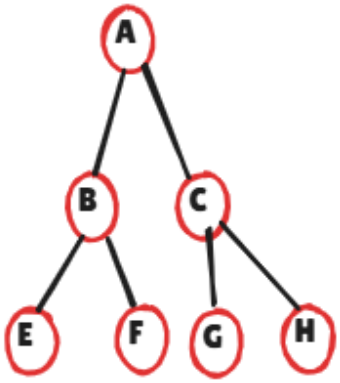
2.2.2.2 Depth-First Search (DFS)

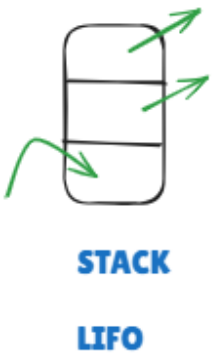
- Depth-First Search (DFS) is an **uninformed search technique** that explores the search space by going **as deep as possible** along one path before backtracking.
- DFS uses a **stack structure (LIFO)**, meaning the most recently generated node is expanded first.

- 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

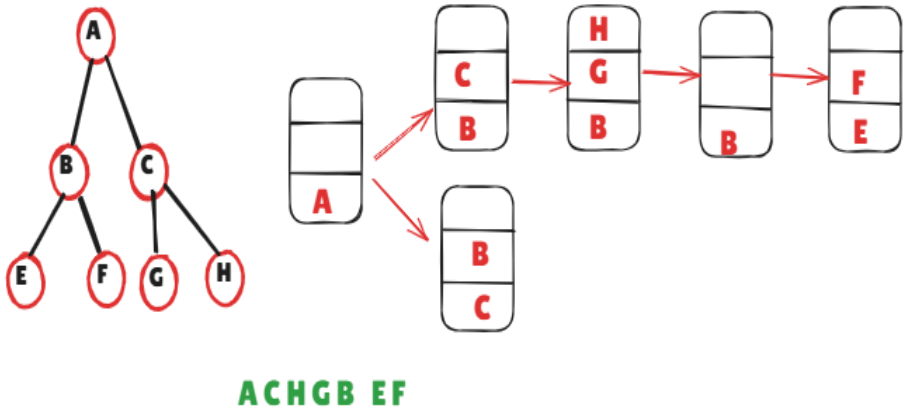
- Explores one branch deeply before exploring others
- Uses **Stack (LIFO)**
- Low memory usage
- Not optimal (may find long or wrong path first)
- Can get stuck in cycles or infinite paths unless controlled





DFS Working Principle

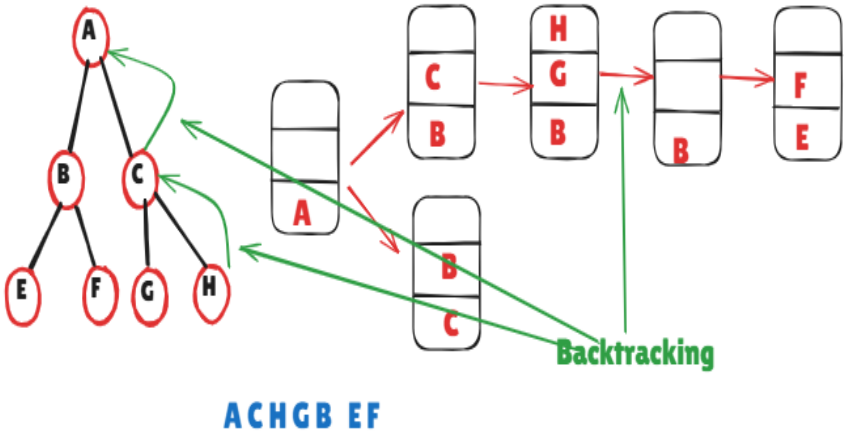
1. Push initial node into stack
2. Pop top node
3. If it is the goal, stop
4. Otherwise, expand it and push all children
5. Continue until stack is empty



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)

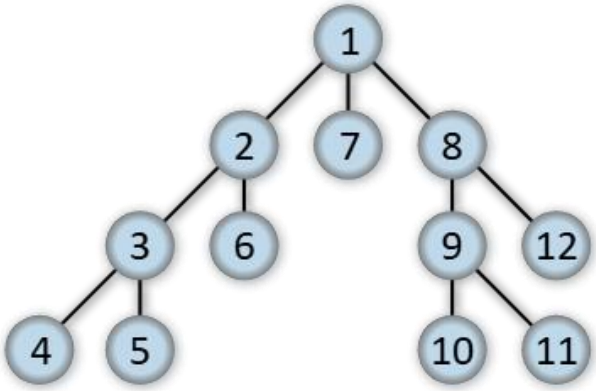
Step	Stack (Top → Bottom)	Expanded Node
1	A	A
2	B, C, D	B
3	E, F?, C, D	E
4	–	**Goal found**

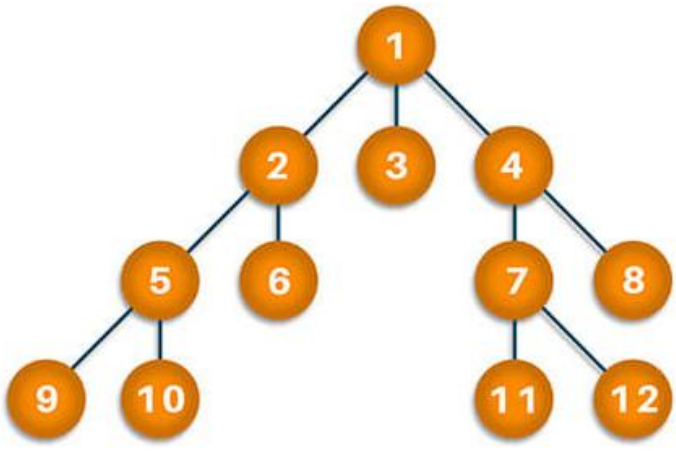
(E is first discovered in the deep branch → DFS stops immediately)

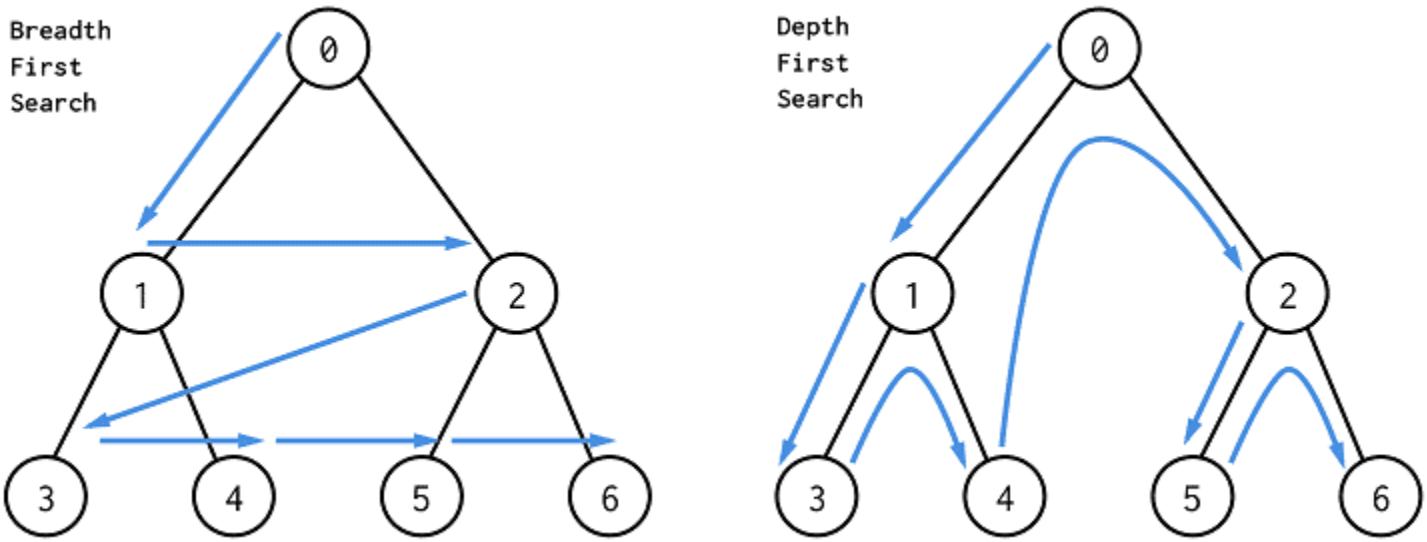
🔷 Real-Life Analogy

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)





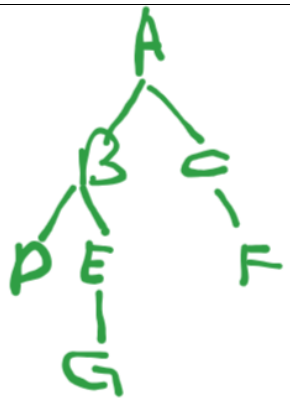


📘 Numerical Question (DFS – Graph Based)

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)

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.

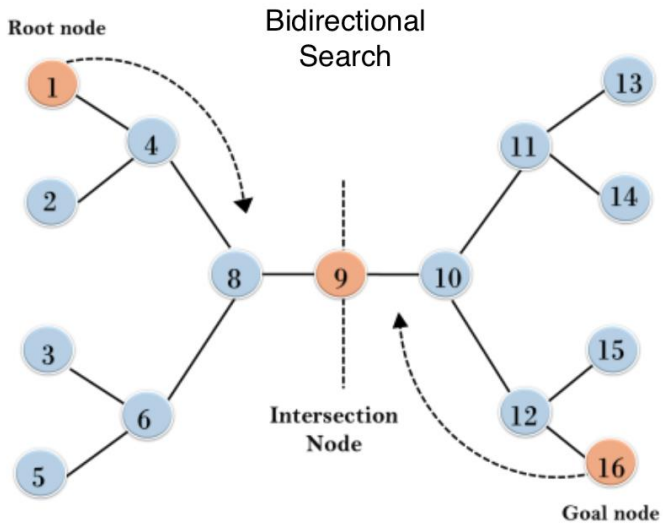
15. The space complexity of BFS is:
Correct Answer: B. $O(b^d)$
Reason:
BFS stores **all nodes at the current frontier (level)**. In the worst case, the number of nodes at the deepest level is ≈ **b^d**, so both time and space complexity are **$O(b^d)$**.

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 b^d 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:

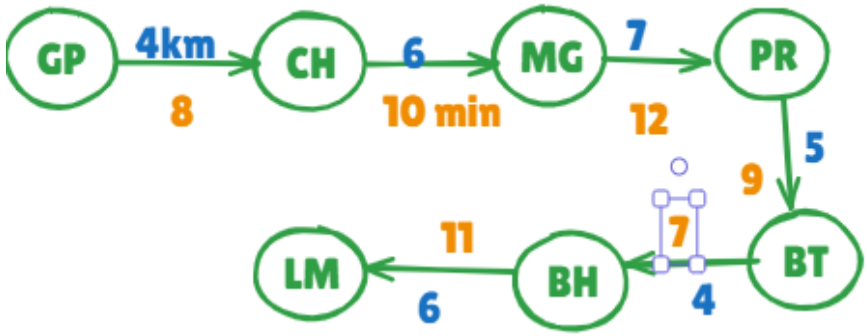
Golpark → Chaura → Manigram → Bhairahawa → Bethari → Pars → Lumbini

Treat each place as a node in a line graph:

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**

---

2 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

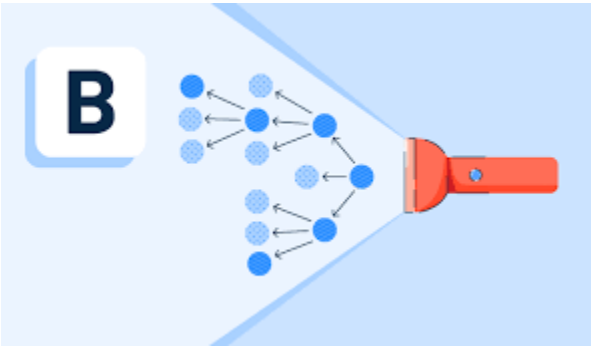
From **backward side** (goal → meet):

- 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

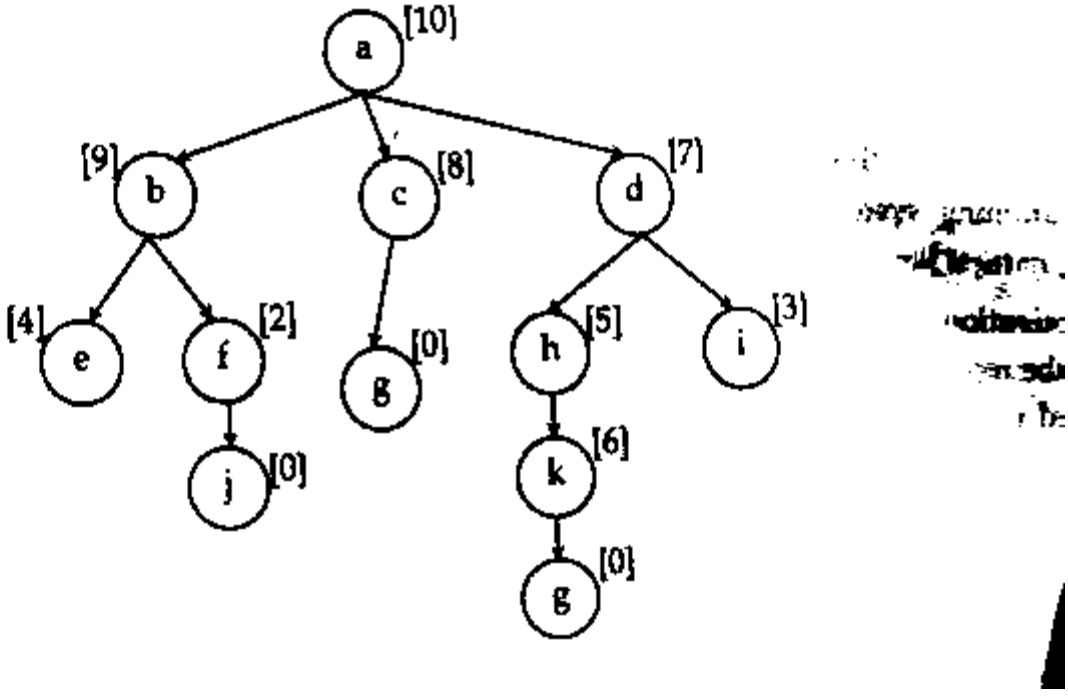
🔷 Disadvantages

✗ 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)

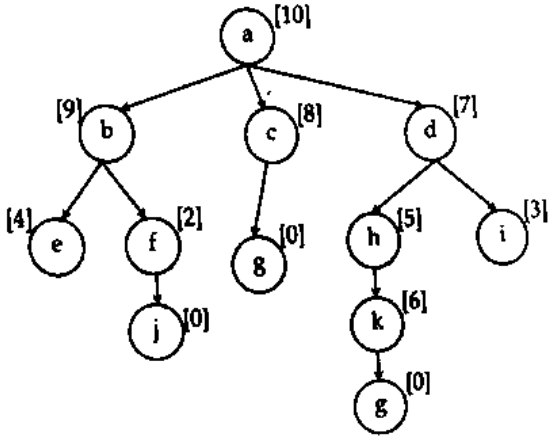
Beam width = 2 → Keep only:

👉 **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:

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

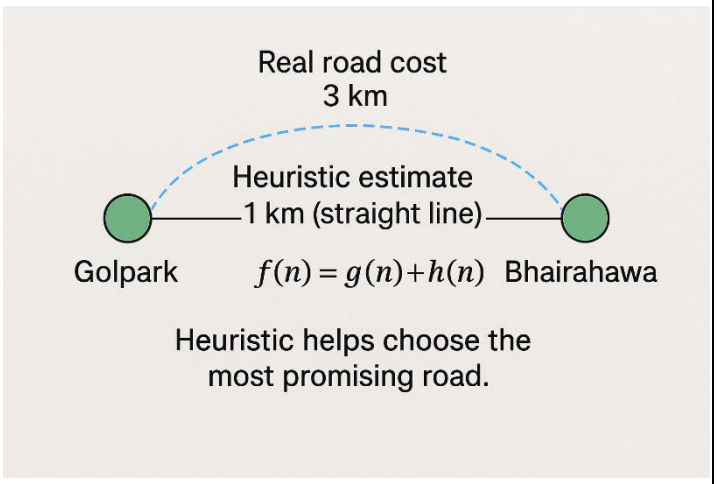
Heuristics make search **faster** and **more goal-directed**.

⚡ **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** 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**

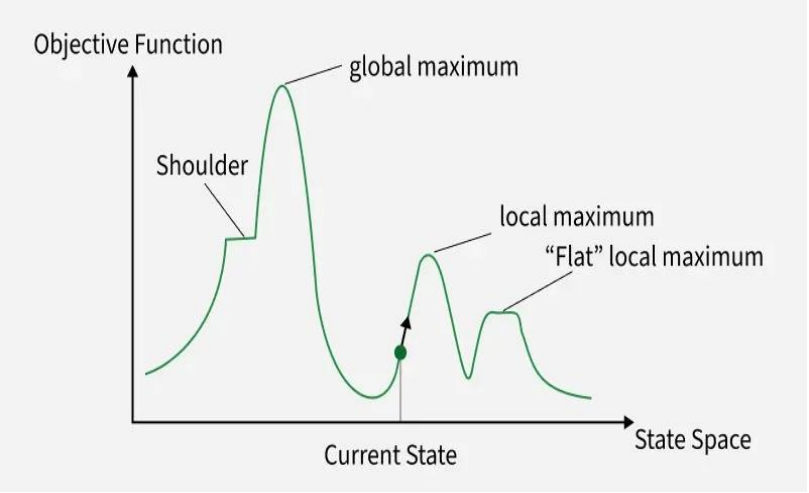
Hill climbing is simple but suffers *from traps such as **local maxima**, **ridges**, and **plateaus***.

🔷 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 (Simple)

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

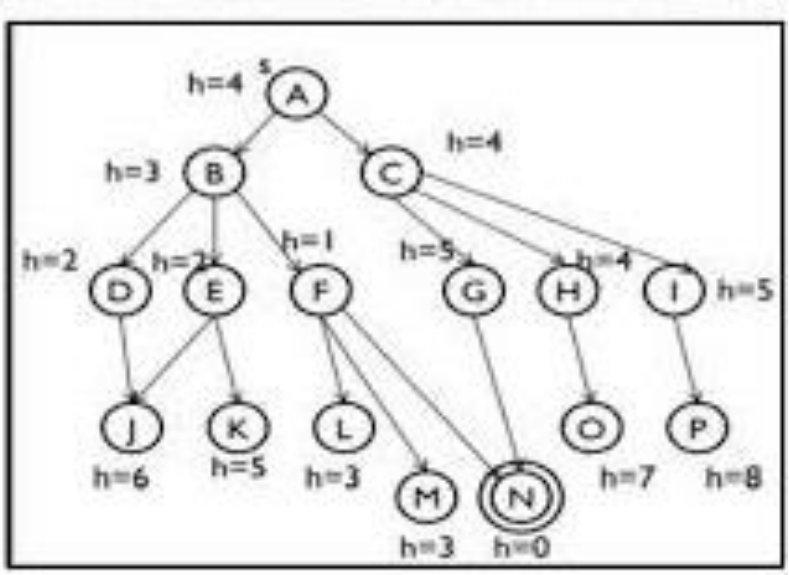
- Feature selection in ML
- Scheduling & timetabling
- Path optimization
- Robotics movement planning

🔷 Advantages

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

🔷 Disadvantages

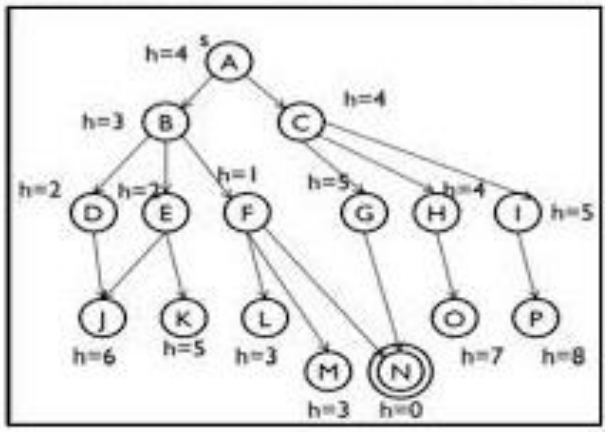
✗ Gets trapped in local maxima/minima
✗ Prone to plateaus
✗ No guarantee of reaching global optimum
✗ Depends heavily on start point

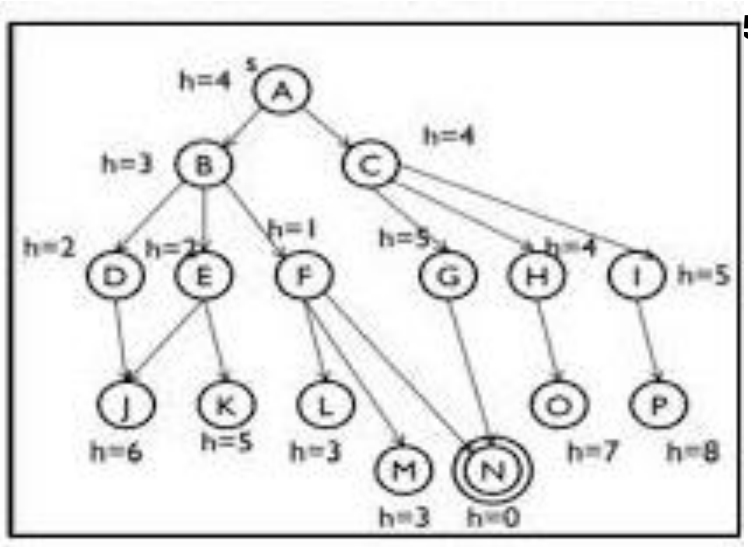


We treat **smaller h = better (closer to goal N)** and use **simple hill-climbing (no backtracking)**.

From the figure:

- A(h=4)
- Children of A: B(h=3), C(h=4), G(h=4), H(h=5)
- Children of B: D(h=2), E(h=4)
- Children of D: J(h=6), K(h=5)
- Goal: N(h=0) (deeper on the C–F–M–N side)





2.2.2.5 Simulated Annealing Search

Simulated Annealing (SA) is an **informed local search algorithm** used to find a **good (near-optimal) solution** in large and complex search spaces.
It is an improvement over **Hill Climbing**, designed to **avoid getting stuck in local maxima/minima**.

The name comes from **annealing in metallurgy** – slowly cooling hot metal so that atoms settle into a low-energy (stable) state.