Unit-2

Go, change the world®

AI – Solving Problems by Searching:

- Problem-Solving Agents
- Example Problems

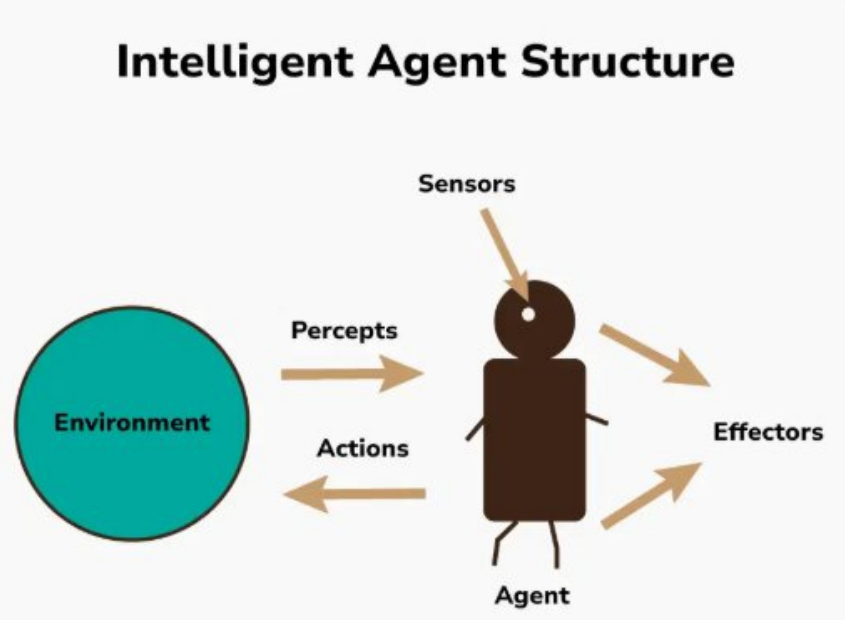
AI – Search Algorithms:

- Uninformed Search Strategies
- Informed (Heuristic) Search Strategies

What is an Intelligent Agent

Go, change the world®

An **intelligent agent** is an autonomous entity that **perceives its environment through sensors** and **acts upon that environment using actuators** to achieve specific goals.



Components of an Intelligent Agent:

- **Sensors** – to perceive the environment

(e.g., cameras, microphones, or data inputs in software)

- **Actuators** – to interact with or change the environment

(e.g., motors, speakers, or output functions in programs)

- **Perception** – the process of interpreting sensor data
- **Decision-Making/Reasoning** – using logic, rules, or learning algorithms
- **Performance Measure** – a metric to evaluate the agent's success

Understanding Problem-Solving Agents and Search

Go, change the world®

- When the next best move is not apparent, an Intelligent Agent has to think about the sequence of steps that lead to the goal.

  - This process of planning – Searching
  - Agent that does this – Problem-Solving Agent

- Best real-world examples and applications:

  - Pathfinding in GPS Navigation Systems (Google Maps)
  - Robot Path Planning (Warehouse Automation)
  - Chess-playing AI (e.g., AlphaZero)
  - Automated Drug Discovery
  - Network Routing in Packet-Switched Networks

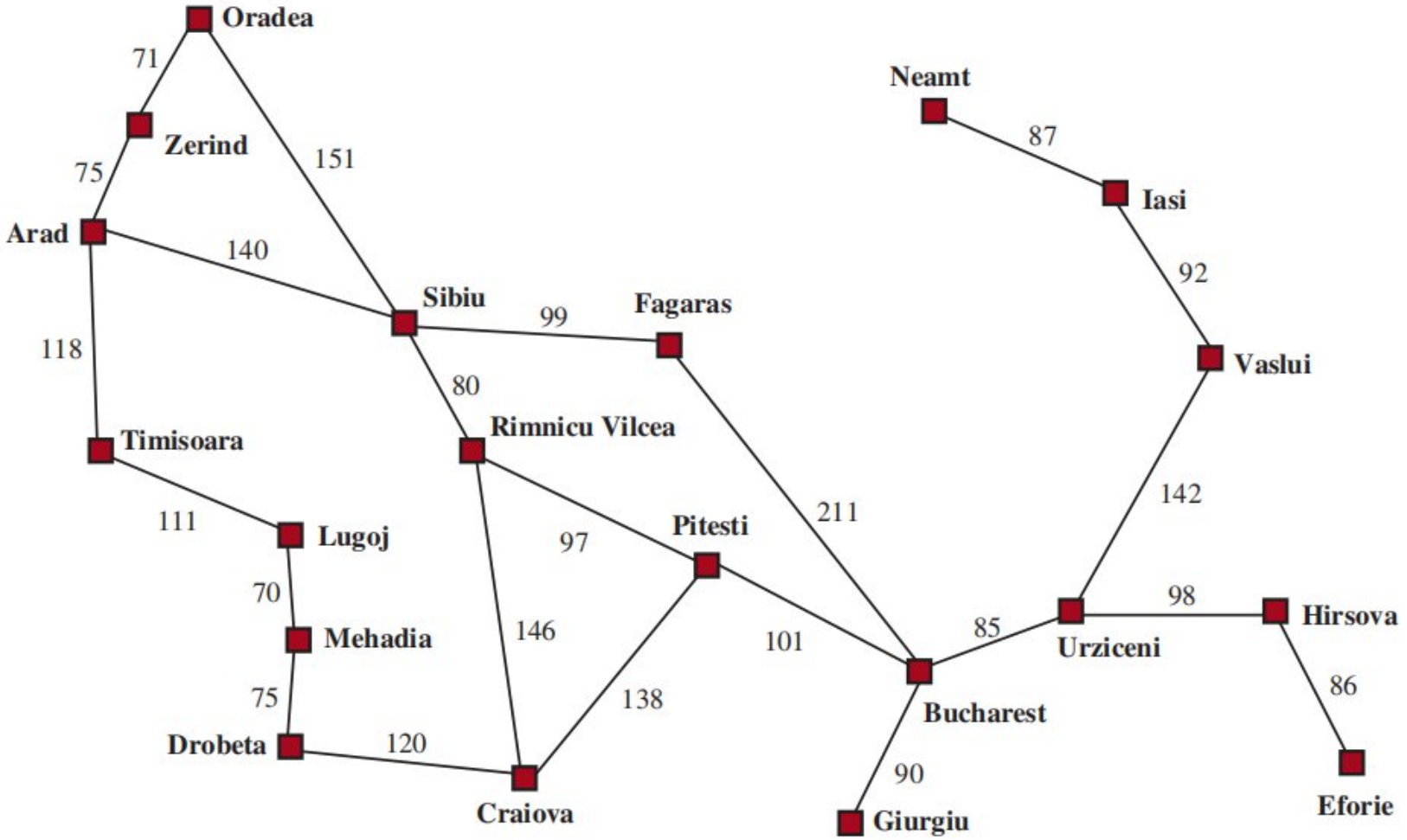
Problem-Solving Agents - **Student Planning Exam Preparation**

Go, change the world®

- A student has 3 days to prepare for **multiple subjects:** DAA, AIML, and CN.
- She/He wants to **maximize marks**, but time is limited.
- **Must plan:** what topics to prioritize, how long to spend on each, and when to revise.
- **Without a study plan**, they might waste time or miss important chapters.
- **Key Message**: The student acts as an agent doing **goal-directed planning**—choosing a smart **sequence of study steps** based on limited time and resources.

Problem-Solving Agents

Imagine an agent enjoying a touring vacation in Romania



Problem-Solving Agents

Imagine an agent enjoying a touring vacation in Romania

- Suppose the agent is currently in the city of Arad and has a nonrefundable ticket to fly out of Bucharest the following day.
- The agent observes street signs and sees that there are three roads leading out of Arad: one toward Sibiu, one to Timisoara, and one to Zerind.
- None of these are the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.
- If the agent has no additional information—that is, if the environment is **unknown**—then the agent can do no better than to execute one of the actions at random.
- With that information, the agent can follow five-phase problem-solving process as mentioned in next slide.

Five Components of Problem-Solving Agents

- Goal Formulation: The agent (or person) decides on a **specific goal or outcome** it wants to reach.
- Problem Formulation: What are the available actions, resources, and constraints? How do I model this problem?
- Search: What is the best sequence of actions to reach my goal?
- Solution: Which path or plan should I follow?
- Execution: Take action and follow the plan!

Open loop and Closed loop

It is an important property that in a **fully observable, deterministic, known environment**, ***the solution to any problem is a fixed sequence of actions****:* drive to Sibiu, then Fagaras, then Bucharest.

If the model is correct, then once the agent has found a solution, it can ignore its percepts while it is executing the actions—closing its eyes, so to speak—because the solution is guaranteed to lead to the goal. Control theorists call this an **open-loop system**: ignoring the percepts breaks the loop between agent and environment.

If there is a chance that the model **is incorrect, or the environment is nondeterministic**, then the agent would be safer using a **closed-loop** approach that monitors the percepts

In **partially observable or nondeterministic environments**, a solution would be a **branching strategy** that recommends different future actions depending on what percepts arrive. For example, the agent might plan to drive from Arad to Sibiu but might need a contingency plan in case it arrives in Zerind by accident or finds a sign saying “Drum ˆInchis” (Road Closed).

What is a Search Problem and its key components

- A **search problem** is defined by a set of components that help an intelligent agent find a path from a **starting point to a goal** in a given environment.

- **Key Components of a Search Problem:**

  - State Space
  - Initial State
  - Goal State
  - Actions
  - Transition Model
  - Cost Function

Search Problems and Solutions

Go, change the world®

Component	Description	Example
Initial state	Where the agent begins	Agent in square A (Vacuum World)
Actions	Possible moves the agent can take	Move Left, Move Right, Suck
Transition model	Defines the outcome of each action	If Suck in dirty square → clean it
Goal test	Determines whether a state meets the goal	All squares are clean
Path cost	Numerical cost of a solution path	Steps taken or energy used

- A **solution** is a sequence of actions that leads from the initial state to a goal state.
- An **optimal solution** has the lowest path cost.

Formulating Problems

Go, change the world®

To formulate a problem:

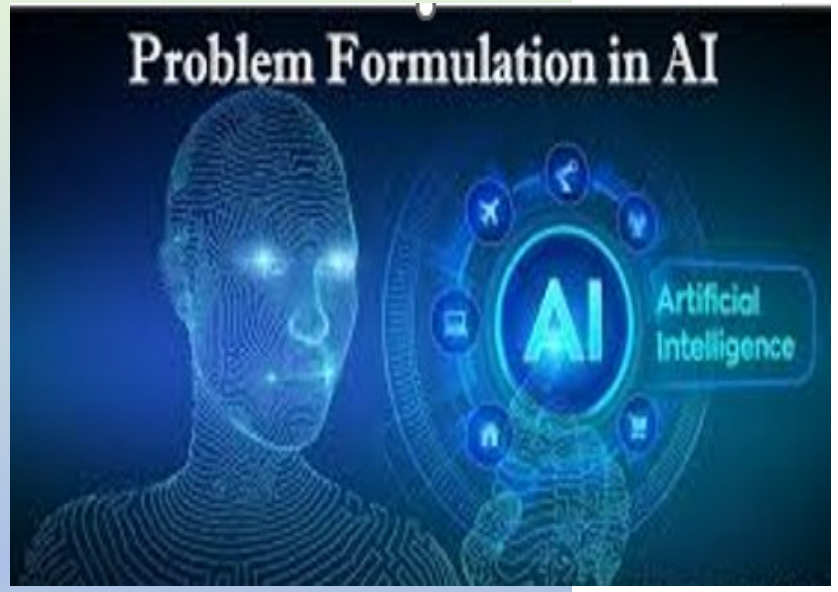
1. **Define states** – How to represent each situation (e.g., position of agent).

2. **Define actions** – What can be done in each state.

3. **Define transition model** – What happens after an action.

4. **Define goal test** – How to recognize success.

5. **Define path cost** – To compare different solutions.



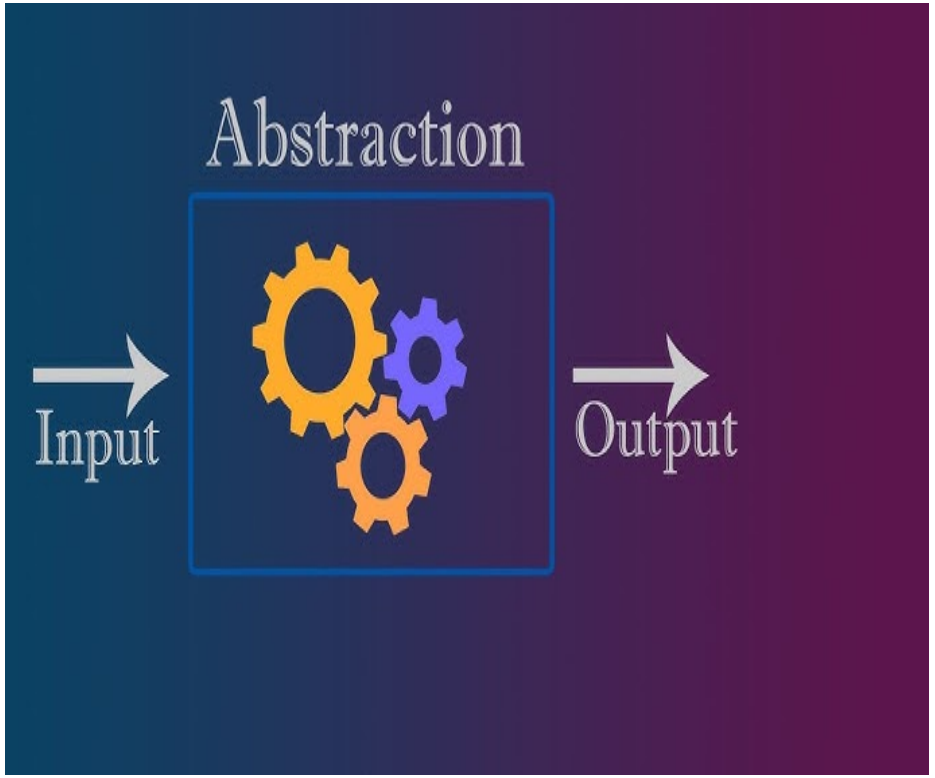
Good formulation simplifies the problem and reduces the search space.

Formulating Problems

Our formulation of the problem of getting to Bucharest is a **model**—an abstract mathematical Description.

The process of removing detail from a representation is called **abstraction**.

When we **formulate a problem**, we must decide **how much detail** of the real world to include in our problem representation. This **degree of detail or simplification** is called the **level of abstraction**.



Formulating Problems

Go, change the world®

High-Level Abstraction (Abstract Representation)

•Focuses only on **major decisions or goals**, ignoring fine details.

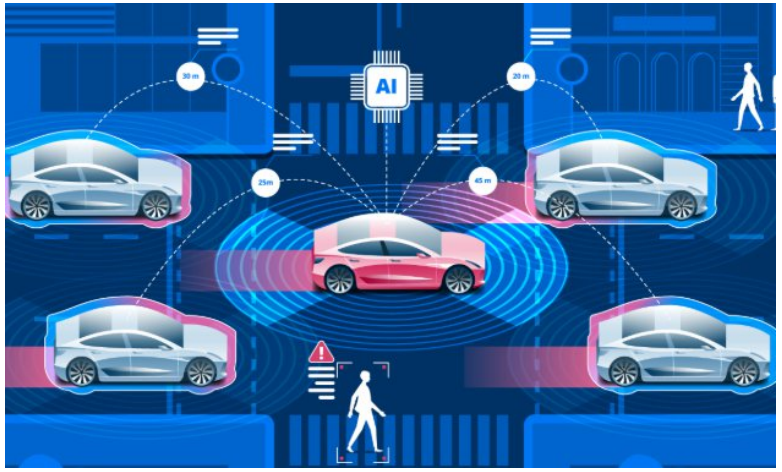
•Used for **planning, reasoning, or conceptual understanding**.

•Makes the problem **simpler and faster** to solve but **less precise**.

Example:

In a **route-finding problem**, we may just represent cities as nodes and distances as edges — not individual lanes, signals, or traffic lights.

→ This is a **high-level abstraction**.



Low-Level Abstraction (Detailed Representation)

•Includes **fine-grained details** of states, actions, and transitions.

•Leads to **more accurate solutions**, but **requires more computation**.

•Used when **precision and realism** are important.

Example:

In a **self-driving car navigation**, the model includes every lane, obstacle, and pedestrian movement.

→ This is a **low-level abstraction**.

Example- Grid World Problem
sokoban puzzle

Go, change the world®

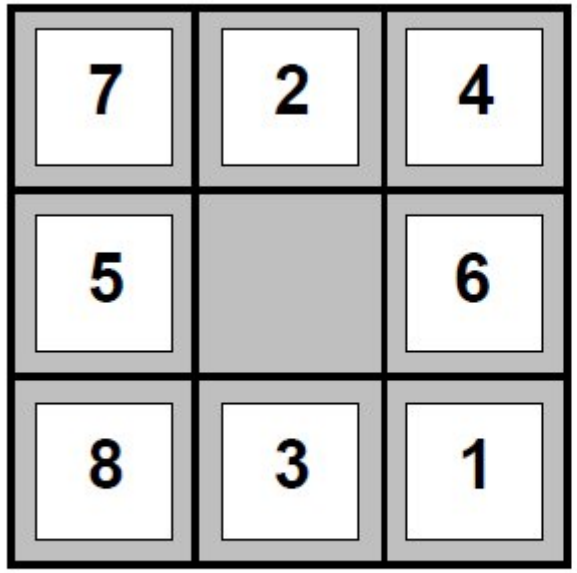
- **8-puzzle problem**

States integer locations of tiles

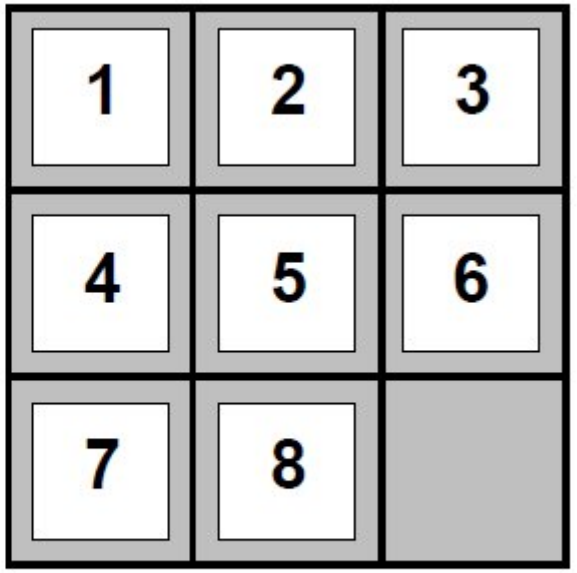
Actions *left*, *right*, *up*, *down*

Goal test = goal state?

Path cost 1 per move



Start State



Goal State

Environment: Warehouse grid with boxes and storage locations.

Example- sokoban puzzle

Component	Description
States	All possible arrangements of 8 tiles and 1 blank
Actions	Move blank: Up, Down, Left, Right
Transition	Swap blank with adjacent tile to reach new state
Goal Test	Check if current arrangement matches target (usually 1–8 with blank last)
Path Cost	Number of moves to reach goal (e.g., 1 per move)

Example- Grid World Problem
Vacuum cleaner

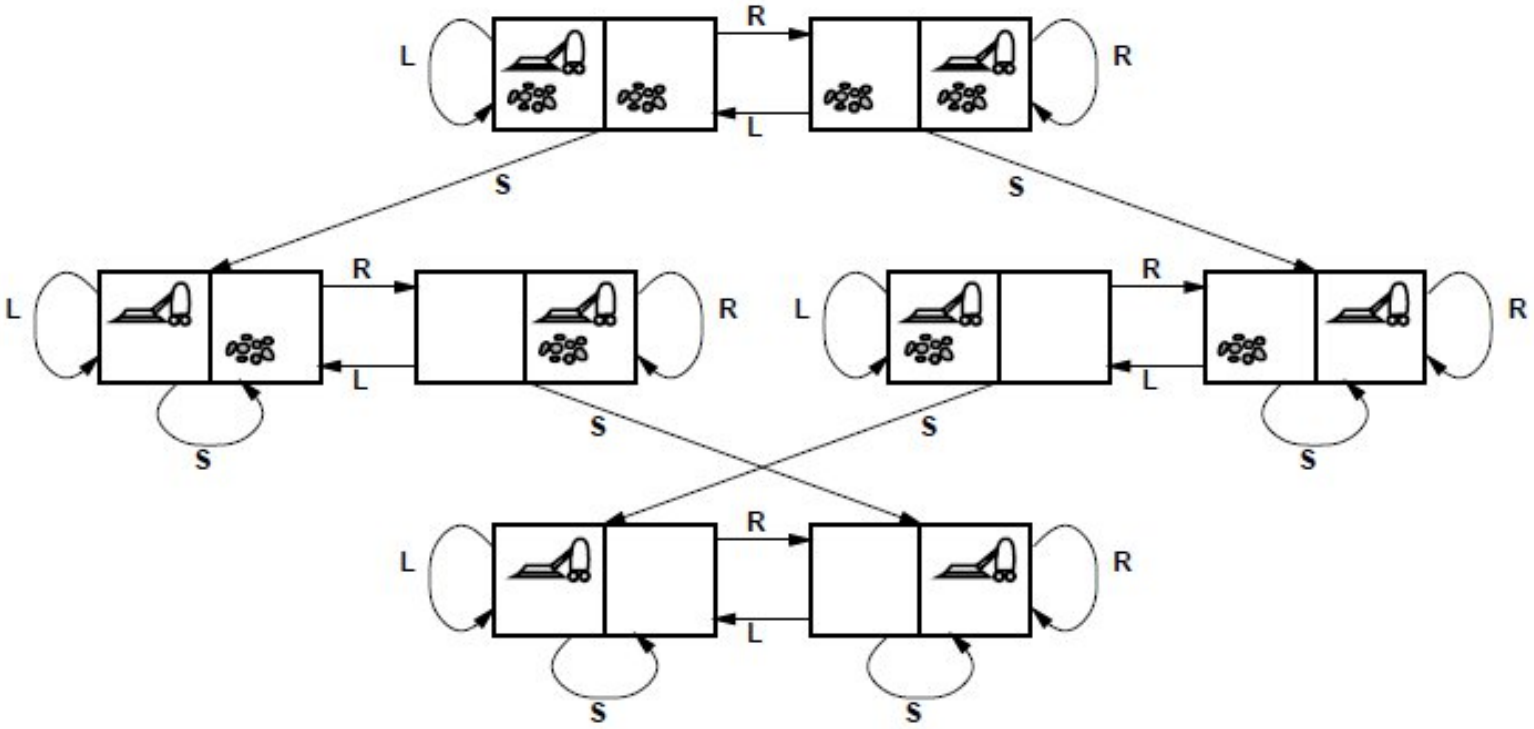
Go, change the world®

States the integer dirt and robot locations

Actions ***left, right, suck, noOp***

Goal test ***not dirty*****?**

Path costs 1 **per operation (**0 **for** ***noOp*****)**



Environment: Two rooms (A and B); each can be dirty or clean.

Example- Grid World Problem
Vacuum cleaner

Go, change the world®

Component	Description
State Space	8 possible states (Vacuum position + room cleanliness)
Initial State	Any of the 8 states
Goal State(s)	Any state where **both rooms are clean**
Actions	L, R, S – Move left/right, Suck dirt
Transition Model	ACTION causes new state (e.g., Suck removes dirt in current room)
Action Cost	Typically cost = 1 per action
Path	A sequence of actions (e.g., Suck → Right → Suck)
Optimal Solution	Minimum-cost path to a goal (e.g., Clean both rooms in 3 steps)

Example- Real-World Problem: Route-Finding

Example: Route-Finding Between Cities

- **States:** Locations (cities).
- **Actions:** Drive between connected cities.
- **Path cost:** Distance or travel time.
- **Goal test:** Reaching the destination city.

Algorithms Used:

- Uniform Cost Search (finds shortest distance).
- A* Search (uses heuristic, e.g., straight-line distance).

Real Applications:

- GPS navigation systems (Google Maps, Waze).
- Airline route optimization.
- Delivery routing for logistics.

Search Algorithms

Go, change the world®

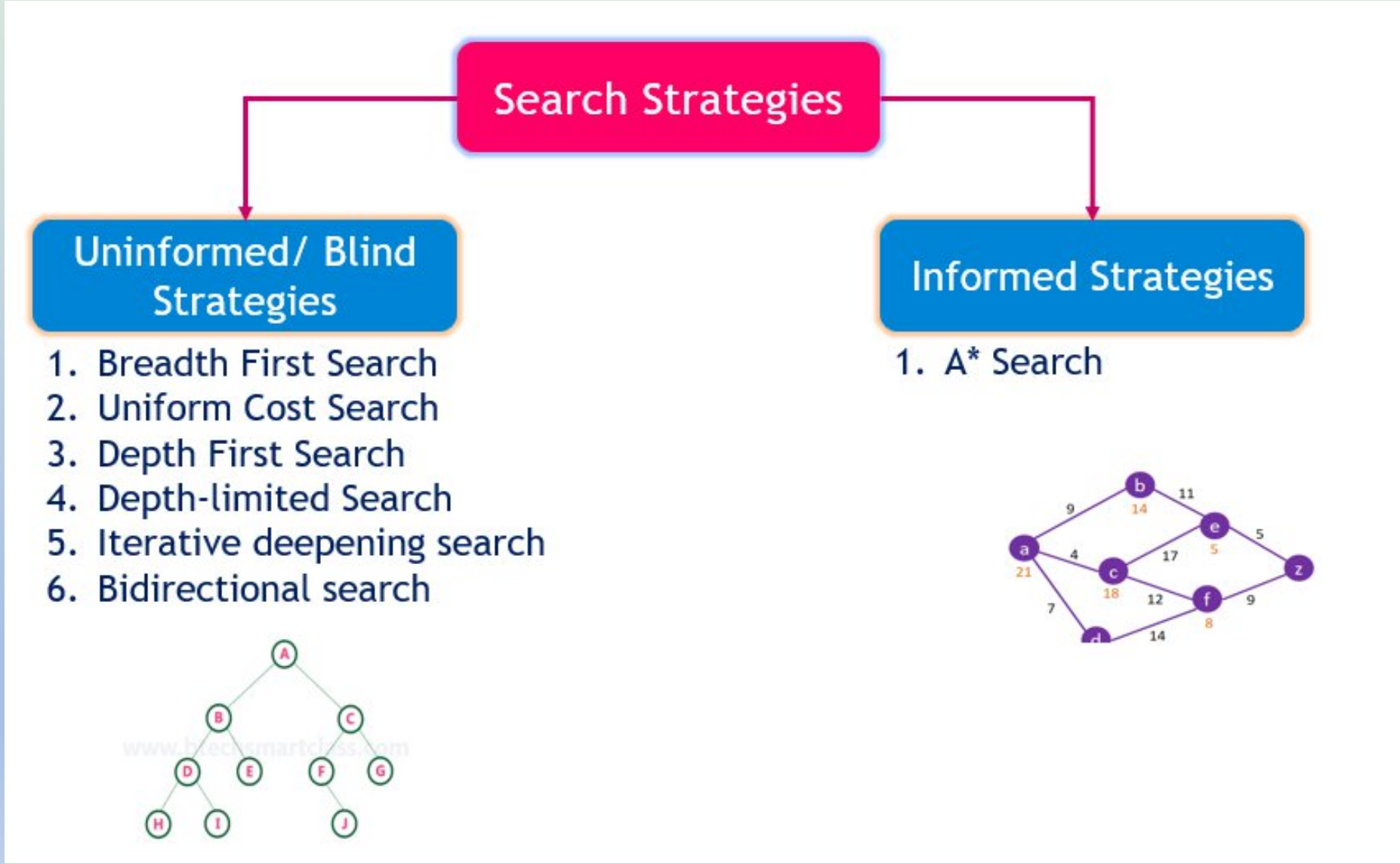
- Search is a fundamental concept in AI for **problem-solving**, **pathfinding**, and **planning**.
- It involves exploring possible states or paths to reach a **goal state** from an **initial state**.
- Search strategies are broadly classified into:
  - Uninformed (Blind) Search Strategies
  - Informed (Heuristic) Search Strategies

Search Algorithms

- Uninformed search algorithms have **no additional information** about the goal other than the definition of the goal itself.
- They explore the search space blindly. It examines each node of the tree until it achieves the goal node.
- Informed strategies use heuristic functions (h(n)) that estimate the cost from a node to the goal.
- They use domain-specific knowledge to guide the search intelligently.
- Informed search strategies can find a solution more efficiently than an uninformed search strategy.

Search Strategies

Go, change the world®



Breadth First Search

Go, change the world®

- **Breadth-First Search (BFS)** is an **uninformed** graph traversal and pathfinding algorithm that explores all **neighbor nodes level-by-level** before moving on to the next level.
- It is guaranteed to find the **shortest path** (in terms of the number of steps) in an unweighted graph.
- When all actions have the same cost, an appropriate strategy is breadth-first search, in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. This is a systematic search strategy that is therefore complete even on infinite state spaces.
- We could implement breadth-first search as a call to **BEST-FIRST-SEARCH** where the evaluation function f(n) is the depth of the node—that is, the number of actions it takes to reach the node.

Breadth First Search

How BFS Works:

- BFS uses a **queue (FIFO)** data structure:
- Start from the **initial node (source)**.
- Add it to the **queue** and mark it as **visited**.
- While the queue is not empty:
  - Remove the front node.
  - Visit all **unvisited neighbors**, add them to the queue and mark as visited.
- Repeat until the **goal node** is found or all nodes are visited.

Breadth First Search

Go, change the world®

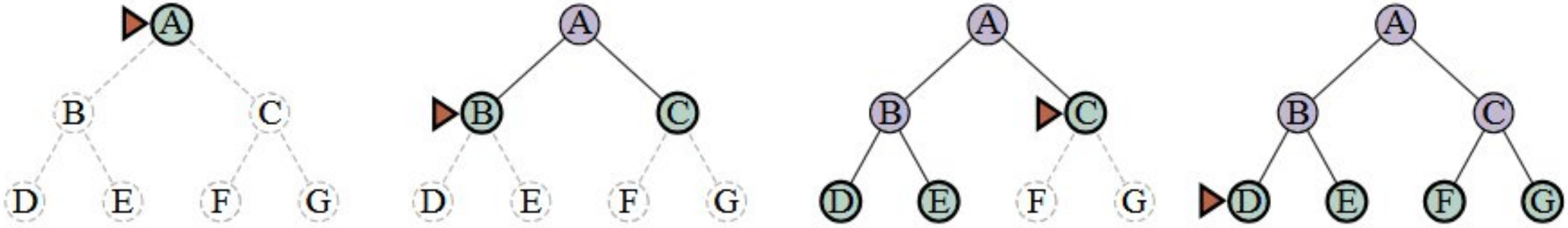


Figure: Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Time & Space Complexity of BFS in a Uniform Tree:

- If the branching factor is ‘b’ and the depth of the solution is ‘d’, then:

Time Complexity: $O(b^d)$

(BFS explores all nodes level-by-level up to depth d)

Space Complexity: $O(b^d)$

(BFS stores all nodes at the current level in memory)

Breadth First Search

Advantages:

- Simple search strategy
- BFS is complete if there exists a solution
- If there are multiple solutions, then a minimal solution will be found (just as in the case – Multiple G's of the previous problem)

Disadvantage:

- Cannot be efficiently used unless the search space is relatively small.

Breadth First Search

Go, change the world®

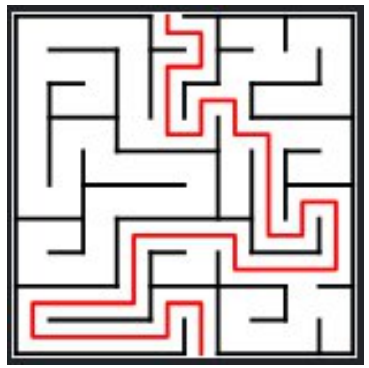
Real-Time Applications of BFS:

Application Area	Use Case & How BFS is Used
Social Networks	Find the shortest connection between two users (e.g., LinkedIn "degrees of separation"). BFS explores all immediate friends before going to friends-of-friends.
Web Crawlers	BFS explores links on a page level-wise to avoid going too deep into one website. It helps ensure all sites are discovered fairly.
GPS/Map Navigation	For unweighted maps, BFS finds the shortest route (e.g., city-to-city path with equal weight roads).
Chatbots / Word Suggestions	In spell checkers or NLP models, BFS explores valid dictionary transformations from a word (like "hit" to "hot", "dot") to reach a target word.
Broadcast in Networks	BFS is used to simulate flooding in network protocols where a packet is sent from the source to all reachable nodes in layers.
Robot Navigation in Mazes	BFS ensures the shortest number of steps to reach the goal in grid mazes or warehouse navigation.
Minimum Number of Moves	BFS is used to solve puzzles like 8-puzzle, Rubik's Cube (to find min moves).

Depth-First Search (DFS)

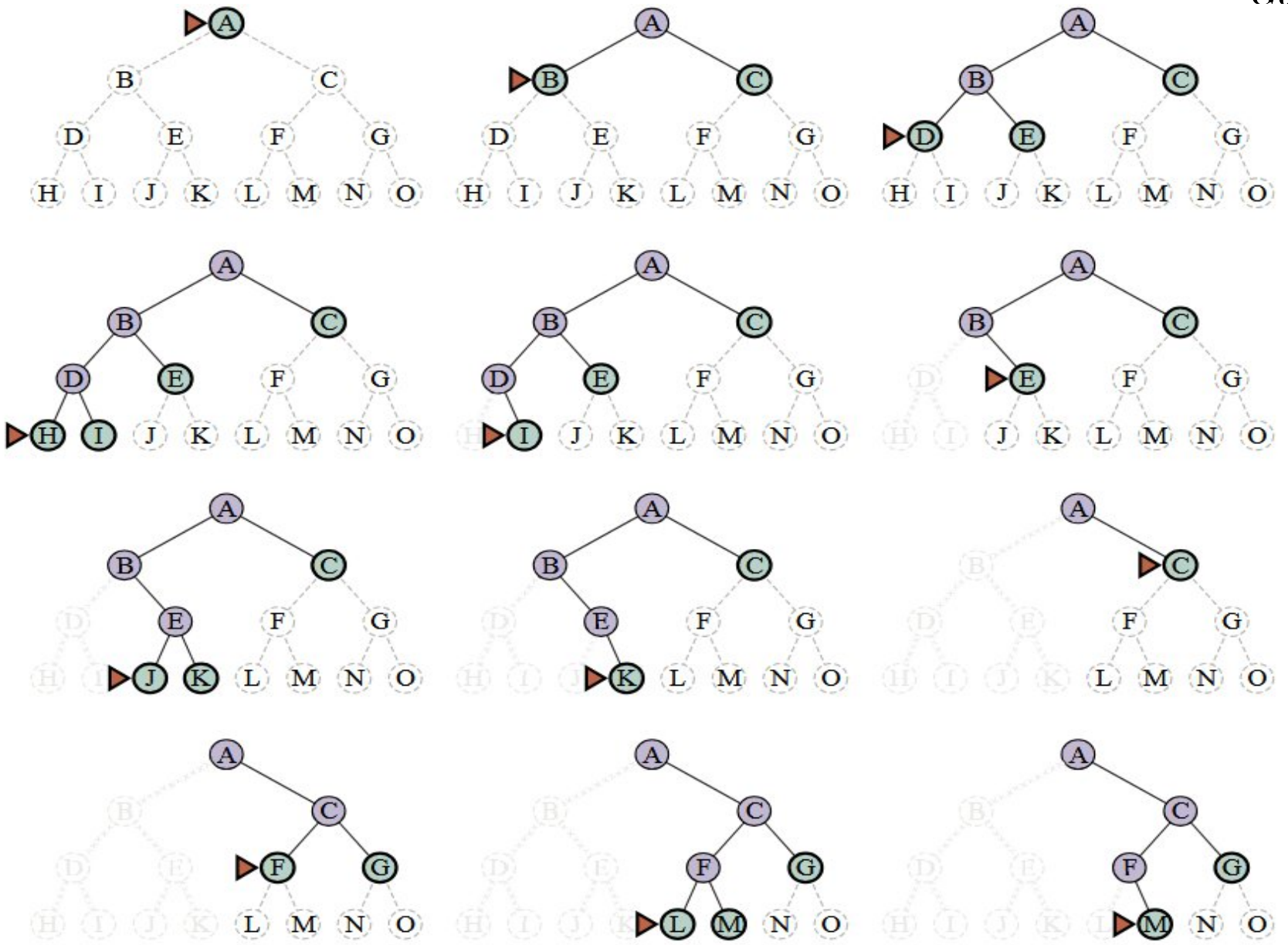
Go, change the world®

- Depth-first search constantly expands the deepest node in the frontier first.
- Search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- The search then “backs up” to the next deepest node that still has unexpanded successors.
- Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not the cheapest.



Depth-First Search (DFS)

Go. change the world®



Depth-First Search (DFS)

Advantages:

- Requires less memory – Only nodes of the current path are stored

- Finds a solution without searching or examining much of the search space

Disadvantages:

- May find a sub-optimal solution (that is deeper ‘G’ than the best solution wrt ACG (BFS) – ABDCG (DFS)

- Incomplete: Without a depth bound, one may not find the solution even if one exists

Depth-First Search (DFS)

Go, change the world®

Strategy	Data Structure	Complete	Optimal	Time Complexity	Space Complexity
Breadth-First Search (BFS)	Queue (FIFO)	Yes	Yes (if cost = 1)	O(b^d)	O(b^d)
Depth-First Search (DFS)	Stack (LIFO)	No	No	O(b^m)	O(bm)

Where:

b = branching factor

d = depth of shallowest solution

m = maximum depth of the tree

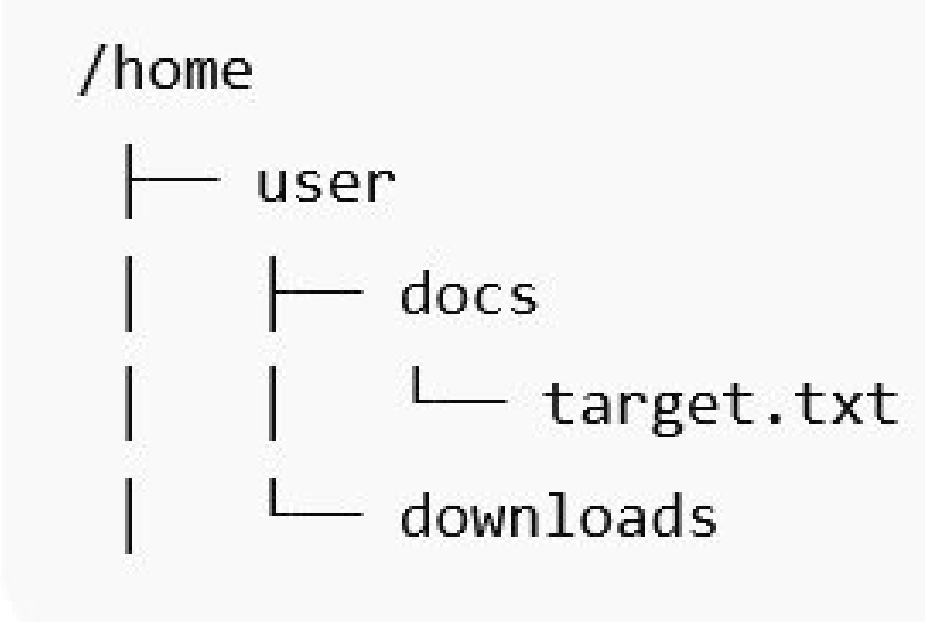
Depth-First Search (DFS)

Real-Time Applications of DFS:

Application Area	Use Case & How DFS is Used
Solving Mazes / Puzzles	DFS dives deep into paths, making it ideal for problems like solving mazes, Sudoku, N-Queens, where backtracking is key.
Topological Sorting	Used in scheduling tasks or dependency resolution (e.g., build systems, compilers).
Cycle Detection	DFS helps detect cycles in graphs (e.g., in deadlock detection in operating systems).
Connected Components	DFS identifies isolated components in networks or images (e.g., blob detection in image processing).
Game Tree Search (AI)	DFS is used in minimax/alpha-beta pruning to evaluate deep sequences of moves (e.g., chess, tic-tac-toe).
File System Traversal	Recursively exploring files/folders is a DFS strategy (e.g., searching in nested directories).

Depth-First Search (DFS)

- **DFS Example – File Search:**
- To search for a file in nested folders:



Informed (Heuristic) Search strategies (A* search)

Key Concept: Heuristic Function (h(n))

- A **heuristic** is an estimate of the cost from node **n** to the goal.
- It helps guide the search in promising directions.
- A* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
- A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
- It is essentially a best first search algorithm.

Informed (Heuristic) Search strategies (A* search)

- A* Algorithm extends the path that minimizes the following function-

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

- Here,
- ‘n’ is the last node on the path
- g(n) is the cost of the path from the start node to node ‘n’ (depth of the node)
- h(n) is a heuristic function that estimates the cost of the cheapest path from node ‘n’ to the goal node (number of misplaced tiles at the nth node)

Informed (Heuristic) Search strategies (A* search)

Given an initial state of an 8-puzzle problem and a final state to be reached-

- Find the most cost-effective path to reach the final state from the initial state using the A* Algorithm.

- Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

2	8	3
1	6	4
7		5

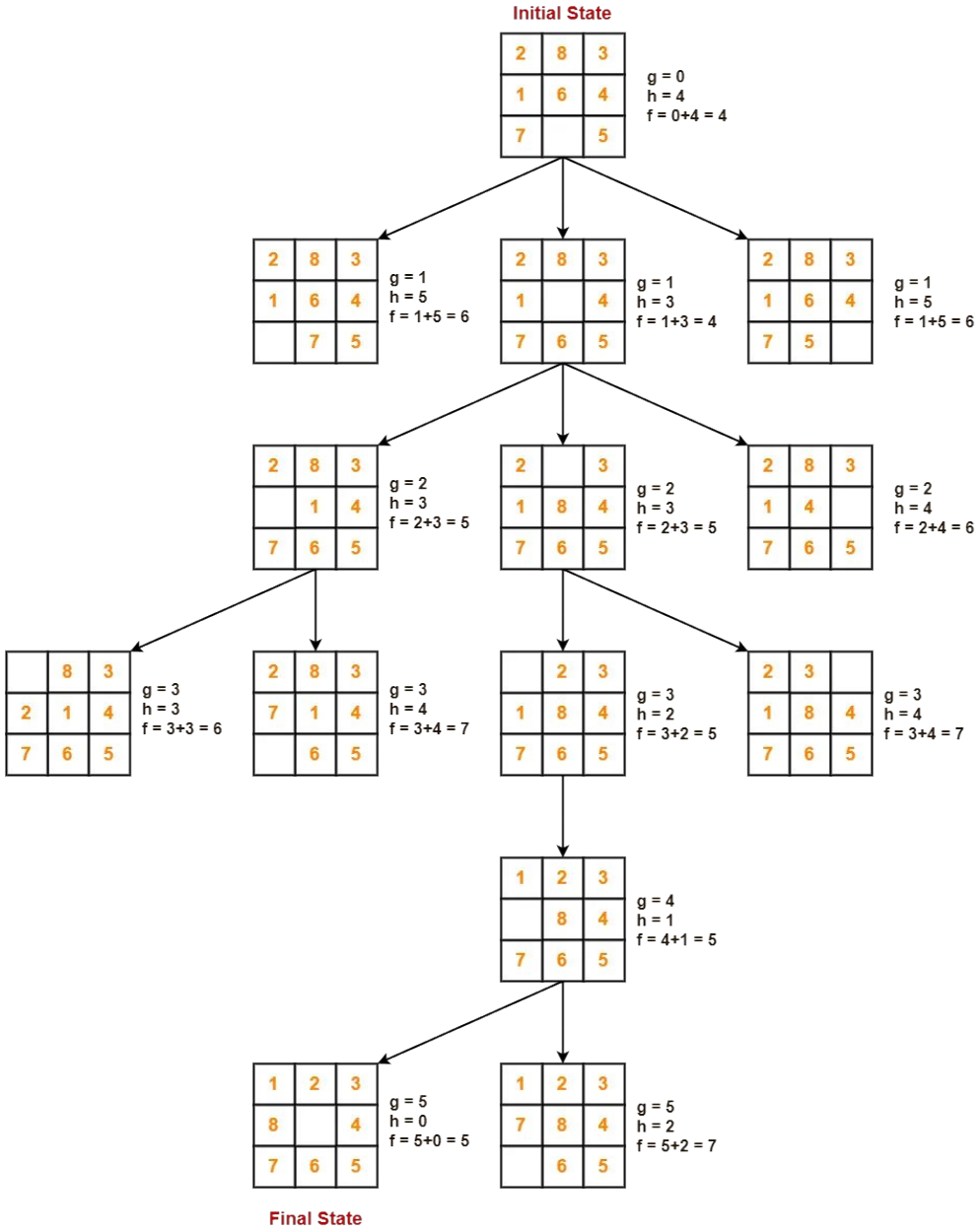
Initial State

1	2	3
8		4
7	6	5

Final State

Informed (Heuristic) Search strategies (A* search)

Go, change the world®



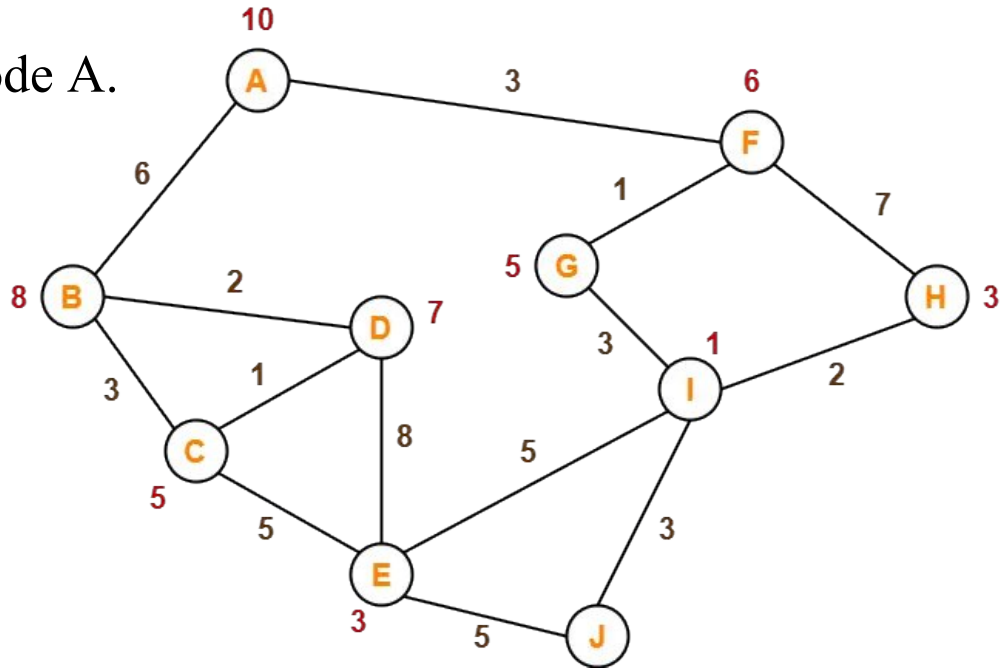
Informed (Heuristic) Search strategies (A* search)

Go, change the world®

- Consider the following graph: The numbers written on the edges represent the distance between the nodes.
- The numbers written on nodes represent the heuristic value.
- Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.

Solution- Step-01:

•We start with node A. Node B and Node F can be reached from node A.



A* Algorithm calculates f(B) and f(F).
•f(B) = 6 + 8 = 14
•f(F) = 3 + 6 = 9

Since f(F) < f(B), so it decides to go to node F.

Path- A → F

Informed (Heuristic) Search strategies (A* search)

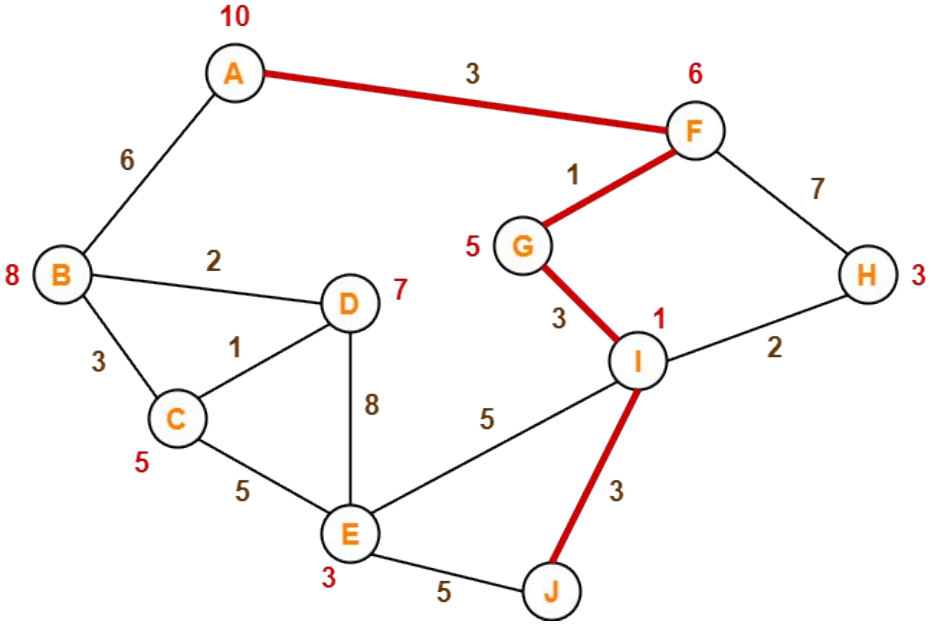
Step 02: Node G and Node H can be reached from Node F.

A* Algorithm calculates f(G) and f(H). f(G) = (3+1) + 5 = 9 f(H) = (3+7) + 3 = 13
Since f(G) < f(H), so it decides to go to node G.
Path- A → F → G

Step-03: Node I can be reached from node G.
A* Algorithm calculates f(I). f(I) = (3+1+3) + 1 = 8 It decides to go to node I.
Path- A → F → G → I

Step-04: Node E, Node H and Node J can be reached from node I.
A* Algorithm calculates f(E), f(H) and f(J).
f(E) = (3+1+3+5) + 3 = 15, f(H) = (3+1+3+2) + 3 = 12
f(J) = (3+1+3+3) + 0 = 10

Since f(J) is least, so it decides to go to node J.
Path- A → F → G → I → J
This is the required shortest path from node A to node J.



Informed (Heuristic) Search strategies (A* search)


```
def print_state(state):
    for row in state:
        print(" ".join(map(str, row)))


def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j


def move_up(state):
    i, j = find_blank(state)
    if i > 0:
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
        return new_state
    else:
        return None
```

Informed (Heuristic) Search strategies (A* search)


```
def move_down(state):
    i, j = find_blank(state)
    if i < 2:
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
        return new_state
    else:
        return None


def move_left(state):
    i, j = find_blank(state)
    if j > 0:
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
        return new_state
    else:
        return None
```

Informed (Heuristic) Search strategies (A* search)


```
def move_right(state):
    i, j = find_blank(state)
    if j < 2:
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
        return new_state
    else:
        return None


def calculate_heuristic(state, goal_state):
    h = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal_state[i][j]:
                h += 1
    return h
```

Informed (Heuristic) Search strategies (A* search)


```
def a_star(initial_state, goal_state):
    OPEN = [(calculate_heuristic(initial_state, goal_state), 0, initial_state, 0)]
    CLOSED = set()

    while OPEN:
        f, g, current_state,iteration = min(OPEN)
        OPEN.remove((f, g, current_state, iteration))
        CLOSED.add(tuple(map(tuple, current_state)))

        print_state(current_state)
        print()

        if current_state == goal_state:
            print(f"{hooray}Solution found!")
            print(f"Iteration:- {iteration}")
            return
```

Informed (Heuristic) Search strategies (A* search)


```
successors = [
        (move_up(current_state), "UP"),
        (move_down(current_state), "DOWN"),
        (move_left(current_state), "LEFT"),
        (move_right(current_state), "RIGHT")
    ]

    successors = [(s, move) for s, move in successors if s is not None and tuple(
        map(tuple, s)) not in CLOSED]

    for successor, move in successors:
        h = calculate_heuristic(successor, goal_state)
        g_successor = g + 1
        f_successor = g_successor + h
        i = iteration + 1

        if (h, g_successor, successor, i) not in OPEN:
            OPEN.append((f_successor, g_successor, successor, i))

print(f"{hooray}No solution found.")
```

Informed (Heuristic) Search strategies (A* search)


```
initial_state = []
print("Enter the initial state (3x3 matrix):")
for _ in range(3):
    row = list(map(int, input().split()))
    initial_state.append(row)



goal_state = []
print("Enter the goal state (3x3 matrix):")
for _ in range(3):
    row = list(map(int, input().split()))
    goal_state.append(row)

a_star(initial_state, goal_state)
```