UNIT - 3

Problem-Solving Agents

Introduction

In Artificial Intelligence, a *problem-solving agent* is an intelligent system that makes decisions by **searching through possible actions** in order to achieve a **predefined goal**.
Unlike simple reflex agents (which respond only to current percepts), problem-solving agents are **goal-based** and capable of **reasoning and planning**.

Key idea:
"Don't just react – think about the goal, search for a sequence of actions, and then execute."

Types of Problems:

◆ **1. Single-State Problems *(Fully Observable + Deterministic)***

In a **single-state problem**, the agent has **complete and accurate information** about the current state of the environment.
There is **no uncertainty** — the agent always knows *exactly* where it is and what will happen after performing each action.

Characteristics

- Environment is **fully observable**
- Every action leads to a **known** outcome (deterministic)
- Agent does **not** need to keep track of multiple possibilities
- Solvable with normal search

Example

Solving a Sudoku puzzle or 8-puzzle
– the agent can see the whole grid, knows every value on the board, and knows that adding a digit in one cell affects only that cell.

Why it matters

- Suitable for environments where **information is perfect**
- Search algorithms only need to consider **one current state**

🔹 2. Multiple-State Problems *(Partially Observable + Deterministic)*

In a **multiple-state problem**, the agent **does not know its exact current state**. Instead, it maintains a **set of possible states** that it *might* be in and constantly updates this set as it receives more percepts.

Characteristics

- Environment is **partially observable**
- Agent may not know its current position
- Agent uses observations to **reduce uncertainty**
- Also called **state-estimation problems**

Example

A robot moving in a dark warehouse without GPS.
At the beginning it might be in one of 4 possible locations.
After every move and sensor reading, it eliminates some of the possibilities until it eventually knows where it is.

Why it matters

- The search must consider **all currently possible states**, not just one
- Often solved using **belief states** (a set of possible actual states)

🔹 3. Contingency Problems *(Observable but Non-Deterministic)*

In a **contingency problem**, the agent knows **where it is** but is **not sure how the environment will behave**.
A single action might have **different outcomes**, so the agent needs to plan for **different contingencies**.

Characteristics

- Current state is known (fully observable)
- Environment behaves **unpredictably**
- The agent must prepare **conditional plans** (if… then…)

Example

A delivery robot that must move from A to B:

- If the road is **free**, go straight.

- IF the road is **blocked**, turn right and take a detour.

Even though it knows its state, it cannot guarantee which event will occur.

Why it matters

- The agent must build a **plan with branches**
- Must handle *different possible futures*

◆ 4. Exploration Problems *(Unknown Environment)*

In **exploration problems**, the agent starts with **no knowledge** of the environment at all.
It has no information about states, actions, or outcomes and must **learn and explore** while solving the problem.

Characteristics

- Environment is **completely unknown**
- Agent must **explore and map** the state space as it goes
- Exploration and goal achievement are done **simultaneously**
- Commonly solved with **learning + search**

Example

A rover on Mars.
It lands in an unknown terrain.
It must explore the environment, learn where obstacles are, and also try to reach particular scientific targets.

Why it matters

- Requires a combination of **search + learning**
- Very common in real-world agents where no map is provided

Problem Type	Knowledge of State	Predictability of Actions	Need to Explore
Single-State	Completely known	Deterministic	No
Multiple-State	Uncertain	Deterministic	Yes (to reduce uncertainty)
Contingency	Known	Non-deterministic	Yes (prepare backup plans)
Exploration	Unknown	Unknown	Yes (to build the map)

Problem Formulation

In Artificial Intelligence, once a goal has been identified, the next step for a problem-solving agent is to **formulate** the problem so that it can be **solved using search**.
Problem formulation is the process of precisely describing the **problem in terms that an agent can understand and solve**.

In simple words, problem formulation converts a *real-life scenario* into a *search problem* by defining states, actions, and goals.

◆ **Why is problem formulation important?**

- A poorly formulated problem leads to **slow or impossible searches**.
- When formulated properly, it becomes **easier and faster** for the agent to find a valid solution.
- It specifies *exactly* what the agent is trying to achieve and *how* it can move through the environment.

◆ **Elements of Problem Formulation**

To formulate a problem, **five components** must be clearly defined:

Component	Description
1. Initial State	The state in which the agent begins
2. Actions	All possible operations/moves the agent can take
3. Transition Model	A specification that defines the result of each action
4. Goal Test	A test that checks whether a given state is a goal state
5. Path Cost	A numeric cost associated with a path (used to compare solutions)

Explanation of Each Component

1. Initial State

This is the **starting point** of the agent.
It represents the very first state from which the agent begins searching.

Example: In a route-finding problem, the initial state might be **“Arad”**.

---

2. Actions

These are the **legal moves** or **operations** available to an agent from a given state.

Example: From the city "Arad", the possible actions could be: go to Sibiu, go to Timisoara, or go to Zerind.

---

3. Transition Model

Identifies **what state will result** after the agent performs a given action.
In other words, it defines the **rules of the environment**.

Example: If the agent performs the action **go-to Sibiu**, the transition model says the new state will be **“Sibiu”**.

---

4. Goal Test

A yes/no test used to determine **whether the agent has reached its goal**.

Example: *“Is the current city = Bucharest?”*
If yes → Goal reached.

---

5. Path Cost

A numeric value assigned to every path.
It helps to compare different solutions and select the **cheapest/shortest** one.

Example: In route-finding, path cost might be **total distance** or **total travel time**.

🔹 Important Points to Remember

- Problem formulation is always done **before** the search begins.
- A **good formulation** simplifies the search.
- **Different formulations** of the same problem can produce **different search complexities**.
- In many real-world systems (robotics, navigation, games), designing the **proper formulation is more important than the search itself**.

1. Vacuum Cleaner World

Description
A simple environment with two locations (A and B). Each location may be **clean** or **dirty**. The vacuum agent can move left, move right, or suck up dirt.

Problem Formulation

Component	Description
Initial State	Agent location + status (clean/dirty) of each room
Actions	MoveLeft, MoveRight, Suck
Transition Model	Describes new state after an action (e.g. sucking removes dirt)

Goal Test	Both locations are clean
Path Cost	1 unit per action

Example:
Initial → (A, dirty, dirty)
Action → Suck → (A, clean, dirty)
Action → MoveRight → (B, clean, dirty)
Action → Suck → (B, clean, clean) ✅ (Goal reached)

Why It's Used

- Helps illustrate how an agent explores a **small state space**
- Useful for understanding deterministic / fully observable problems

---

2. 8-Puzzle Problem

Description
A sliding puzzle consisting of 8 numbered tiles and a blank space on a 3x3 board. The goal is to arrange the tiles in order.

Problem Formulation

Component	Description
Initial State	Any starting arrangement of tiles
Actions	Up / Down / Left / Right move of the blank
Transition Model	New arrangement after swapping blank with a neighbor
Goal Test	Tiles arranged in ascending order
Path Cost	Each move has cost = 1

Why It's Used

- Useful for practicing BFS, DFS and A*
- Demonstrates **state representation** and **state-space explosion**

---

3. Route Finding / Map Problem (Romania Map)

Description
Find the shortest path between two cities on a map.
Popular example: **Arad** → **Bucharest** in the Romania road map.

Problem Formulation

Component	Description
Initial State	Starting city (e.g., Arad)
Actions	Drive to a connected neighbouring city
Transition Model	Resulting city after driving from A to B
Goal Test	Current city == Destination (Bucharest)
Path Cost	Distance (or travel time)

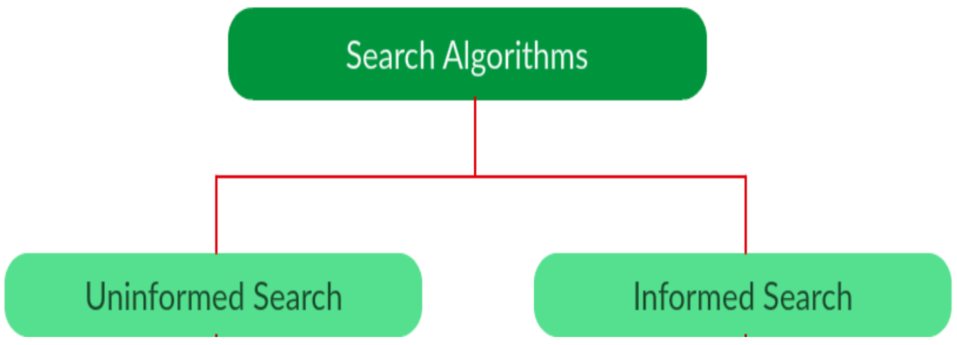
Example (shortest route):
Arad → Sibiu → Fagaras → Bucharest

Why It's Used

- Realistic example of **path planning**
- Used to compare **BFS, UCS, and A*** search strategies

Basic Search Algorithms (Overview)

- In Artificial Intelligence, **search algorithms** are used by problem-solving agents to explore the state space and find a path from the **initial state** to the **goal state**.
- They can be broadly classified into:



Uninformed Search

The search algorithms in this section have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and/or length of actions. Uninformed search is also called **Blind search**. These algorithms can only generate the successors and differentiate between the goal state and non goal state.

The following uninformed search algorithms are discussed in this section.

1. Depth First Search
2. Breadth First Search
3. Uniform Cost Search

1. Breadth-First Search (BFS)

➤ **Definition:**
BFS explores the search space **level by level**, expanding all nodes at the current depth before going deeper.

➤ **Working Principle:**

- Uses a **FIFO Queue**
- Start from the initial node
- Add all its neighbors to the queue
- Remove from the front of the queue and expand
- Continue until the goal is reached
- ➤ **Properties**

Property	Value
Complete	Yes
Optimal	Yes (if all step costs = 1)
Time Complexity	O(b**d)
Space Complexity	O(b**d)

(*b = branching factor, d = depth of solution*)

➤ **Example Use Case:**
Finding the **shortest route** in an **unweighted** graph.

2. Depth-First Search (DFS)

➤ **Definition:**
DFS explores a path **as deep as possible** before backtracking to try another path.

➤ **Working Principle:**

- Uses a **stack** (either explicitly or via recursion)
- Go to one child, then grandchild, and so on…
- If dead-end reached → backtrack and explore next branch

➤ **Properties**

Property	Value
Complete	No (fails in infinite trees)
Optimal	No
Time Complexity	O(b**m) (m = max depth)
Space Complexity	O(b*m)

➤ **Example Use Case:**
Searching a deeply nested directory/folder structure.

3. Uniform Cost Search (UCS)

➤ **Definition:**
UCS expands the **least-cost node first**, using the actual path cost (g(n)).
It is like BFS, but for **weighted graphs**.

➤ **Working Principle:**

- Uses a **priority queue**
- Always picks the node with **lowest total cost so far**

➤ **Use Case:**
Finding the **cheapest route** when roads have different distances.

Informed Search:

Here, the algorithms have information on the goal state, which helps in more efficient searching. This information is obtained by something called a *heuristic.*
In this section, we will discuss the following search algorithms.

1. Greedy Best-First Search
2. *A Search*

- **Search Heuristics:** In an informed search, a heuristic is a *function* that estimates how close a state is to the goal state. For example – Manhattan distance, Euclidean distance, etc. (Lesser the distance, closer the goal.) Different heuristics are used in different informed algorithms discussed below.

1. Greedy Best-First Search (Heuristic Search)

➤ **Definition:**
Greedy search uses **only a heuristic (h(n))** to estimate how close a node is to the goal, and **expands the closest** node.

➤ **Working Principle:**

- Uses priority queue
- Picks the node with the **smallest h(n)**
- Ignores path-cost accumulated so far

➤ **Advantage:** Very fast
➤ **Drawback:** Not optimal; can get stuck in local minima

➤ **Use Case:**
Fast route planning with approximate results (navigation apps in low precision mode)

◆ ***2. A Search (Heuristic + Cost)****

➤ **Definition:**
A* combines **path cost (g(n))** and **heuristic (h(n))** as

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

It expands the node with the **lowest value of f(n)**.
If **h(n)** is admissible (never overestimates), A* is **guaranteed to be optimal**.

➤ **Use Case:**
Shortest path in road maps and game AI (path-finding in maps).