Artificial Intelligence

Problem Solving Using Search

Christian sy

Specific Learning Outcomes

By the end of this topic, students will be able to:

1. **Explain** how intelligent agents use search algorithms to perceive, decide, and act in solving problems;
2. **Differentiate** between uninformed and informed search strategies in terms of performance and applicability to AI agents;
3. **Construct** and interpret search trees (BFS, DFS, UCS, IDS, Greedy Best-First, A*) to model AI agent behavior;
4. **Apply** search strategies and heuristics to guide intelligent agents in problem-solving environments such as the Vacuum World or mazes;
5. **Evaluate** the effectiveness of search algorithms in real-world applications of intelligent systems (e.g., robotics, navigation, game AI).

Part I
AI Agents

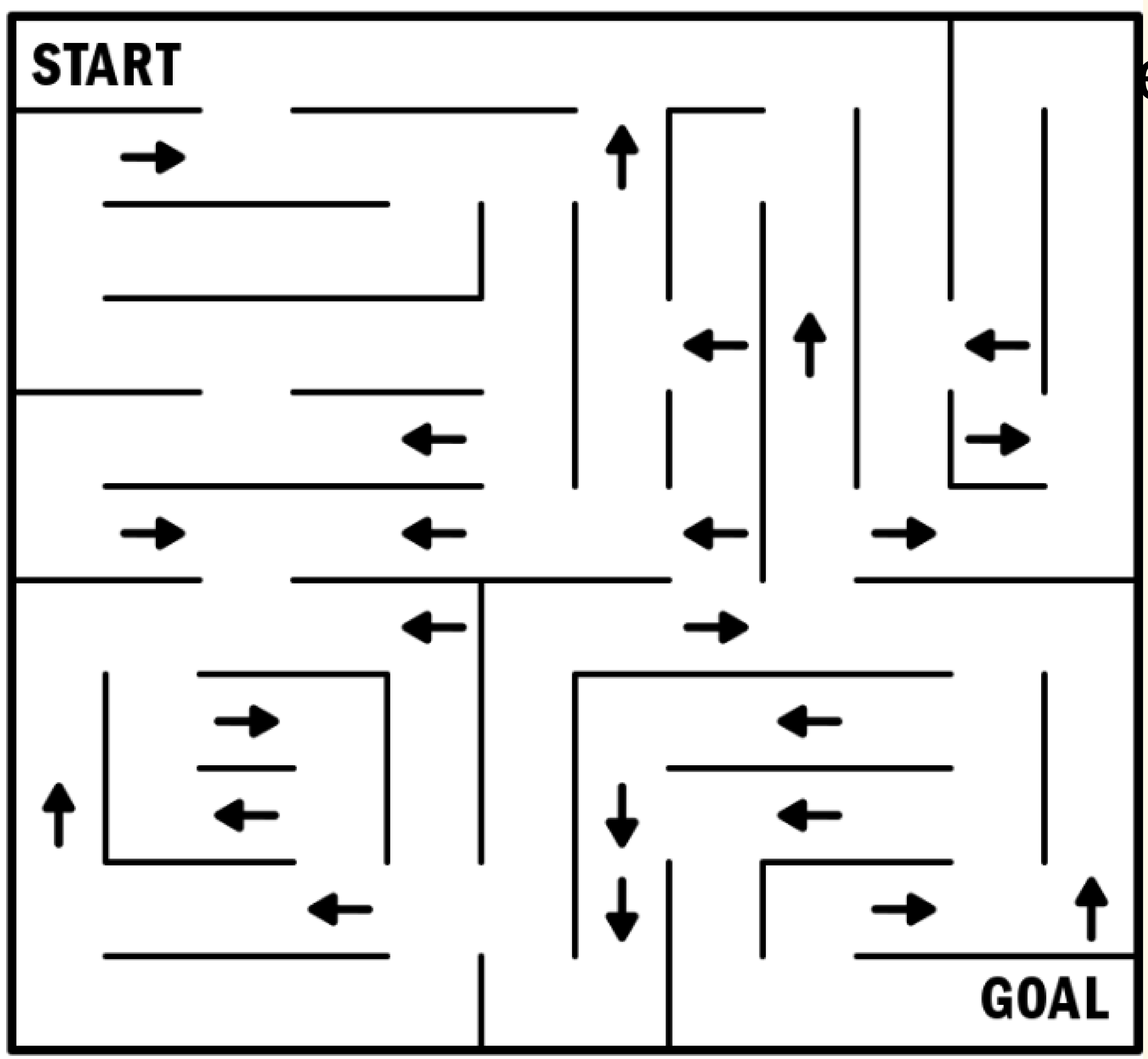
Problem Solving Using Search

Imagine you're playing **hide-and-seek with a robot friend in your house**:

- The **robot is the intelligent agent**.
- The **goal** is to find you.
- The **house** represents the environment (with rooms, furniture, and possible hiding spots).

Now, how does the robot decide *where to look first*?

- If it **checks every corner step by step** (Breadth-First Search), it won't miss anything—but it might take a lot of time and memory.
- If it **runs straight into one room and searches deeply** (Depth-First Search), it might find you quickly… or waste time if you're in a different room.
- If it **balances both approaches, searching deeper little by little** (Iterative Deepening Search), it can guarantee finding you without using too much memory.
- Later, if it **uses clues like footprints or sounds** (heuristics), it can search more intelligently (Informed Search: Greedy, A*).



em Solving Using Search

In AI, problem solving means finding a sequence of steps (a *solution*) that transforms the current state into the goal state.

Key Idea: Break down complex tasks into well-defined states, operators, and goals.

Example:

•Solving a maze → states = positions in maze, operators = movements, goal = exit.

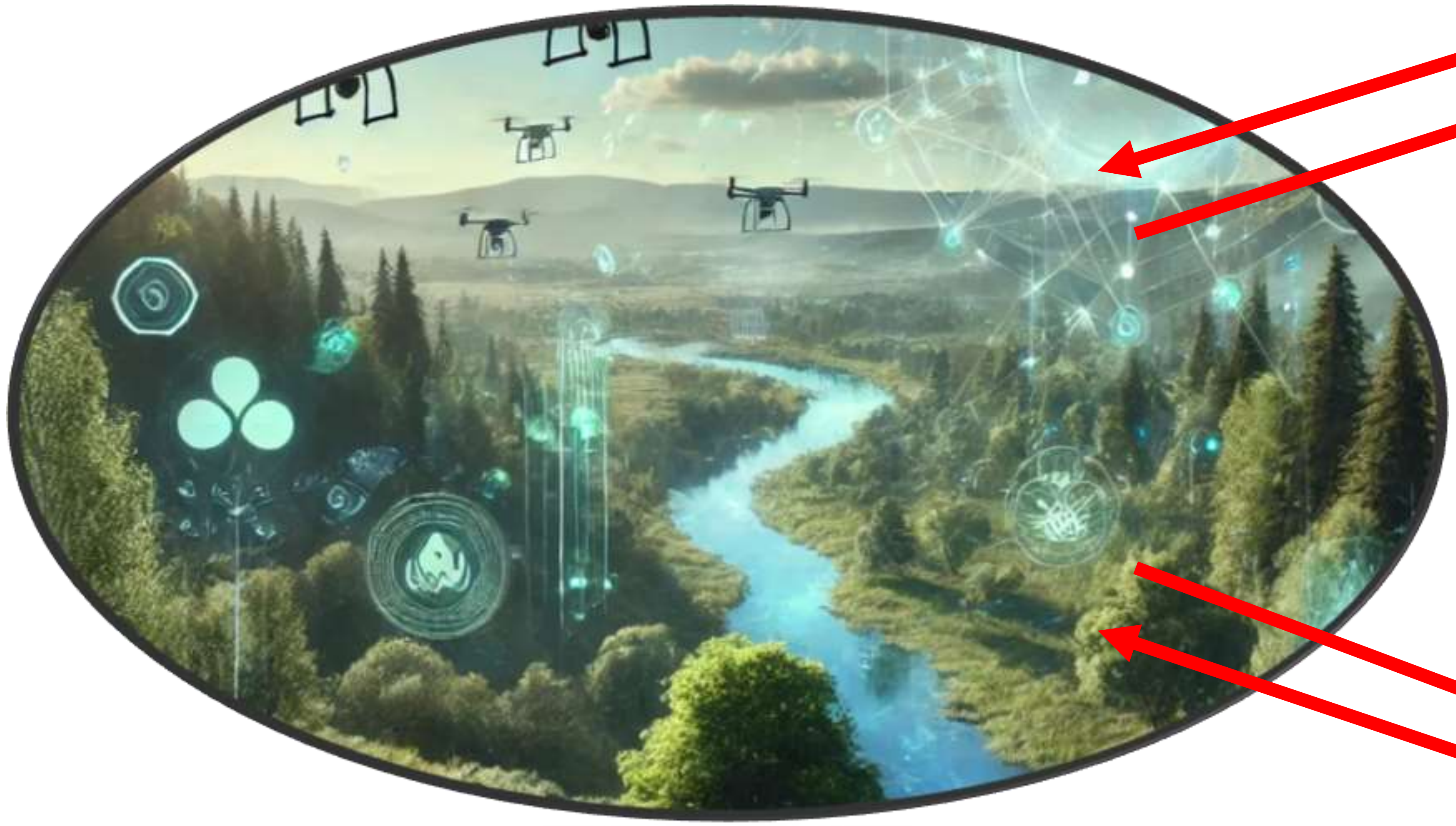
•Chess → state = current board, operators = moves, goal = checkmate.

Problem Solving Using Search

A goal and a set of means for achieving the goal is called a **problem**, and the process of exploring what the means to solve these problems is called **search**.

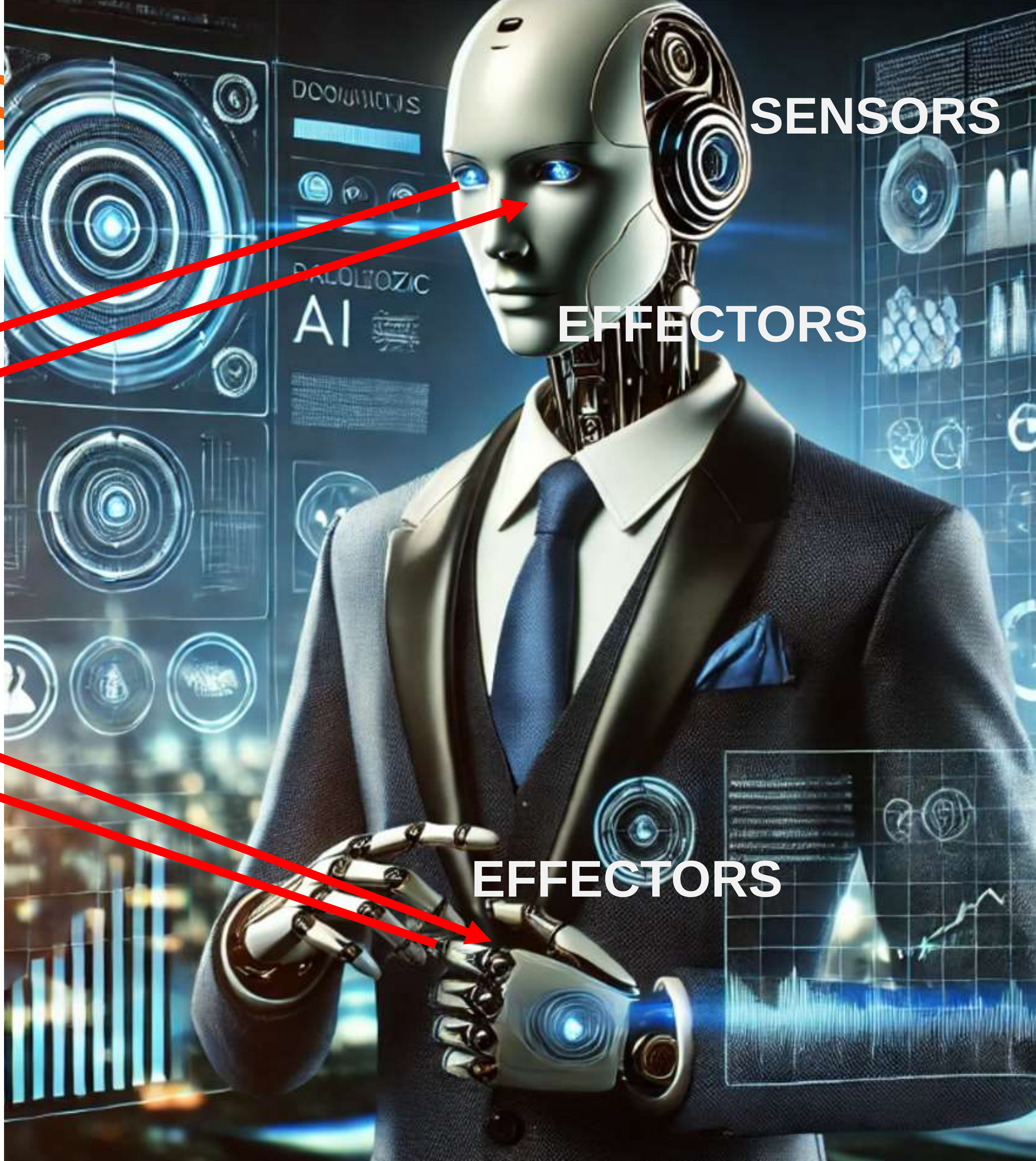
Intelligent Agents

An agent is a system that...



ENVIRONMENT

- perceives its environment through its sensors
- acts on its environment through actuators or effectors



Intelligent Agents

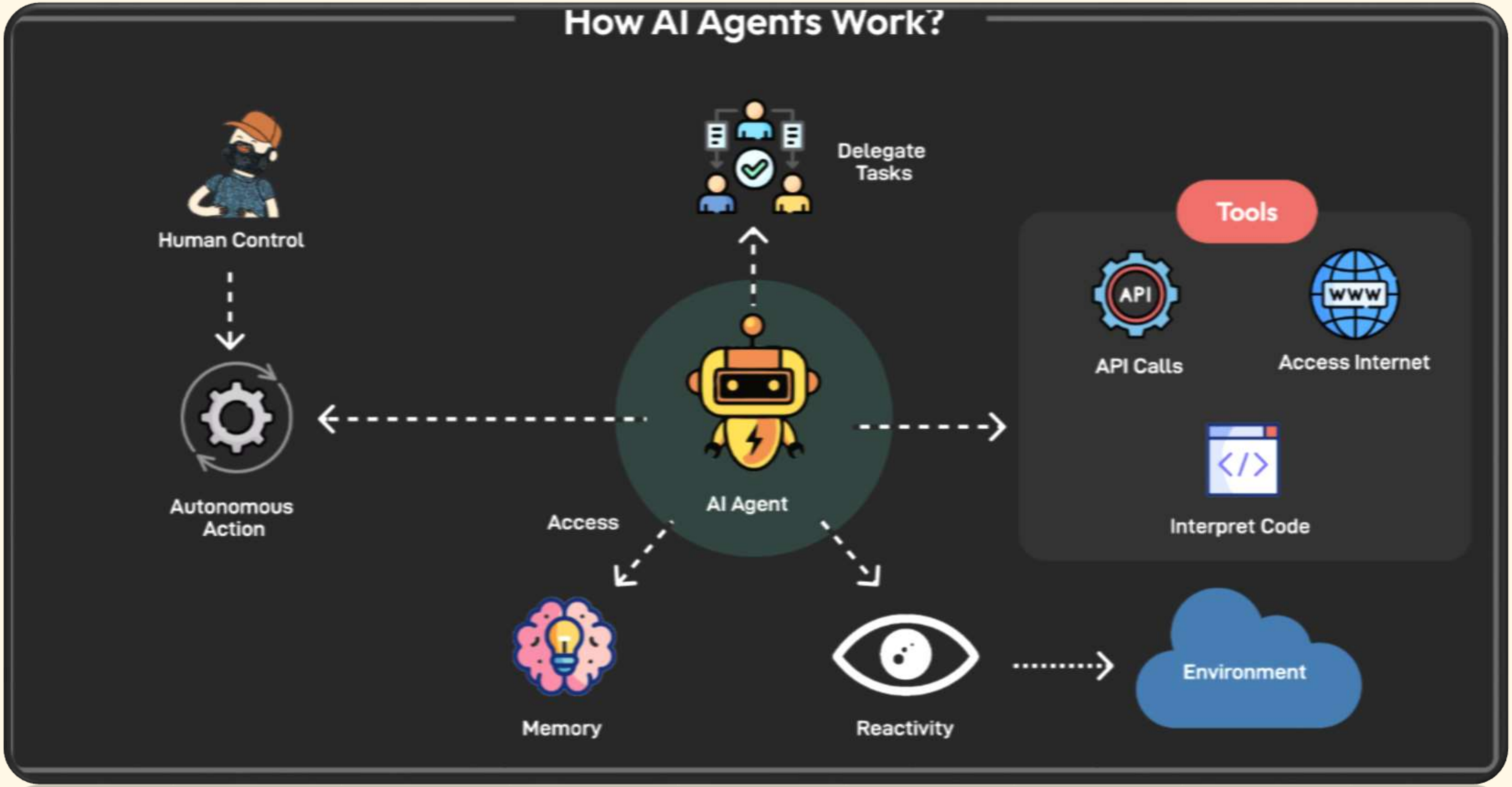
Anything that can be **viewed** *as* **perceiving** its **environment** through **sensors** and **acting** upon that environment through its **effectors** to maximize progress towards its **goals**.

Intelligent Agents vs AI Agents

Intelligent Agents and **AI Agents** generally refer to the *same concept*, but the wording is used in slightly different contexts:

- **Intelligent Agent** – is the broader term used in **AI theory, philosophy, and computer science** to describe any entity that *perceives its environment, reasons, and takes action to achieve goals*. Not all intelligent agents have to be strictly "AI" (e.g., a thermostat can be seen as a simple intelligent agent, even if it's not AI).
- **AI Agent** – emphasizes that the agent is built on **artificial intelligence techniques**, meaning it uses algorithms, reasoning, learning, and decision-making. This term is common in AI textbooks and applied computer science.

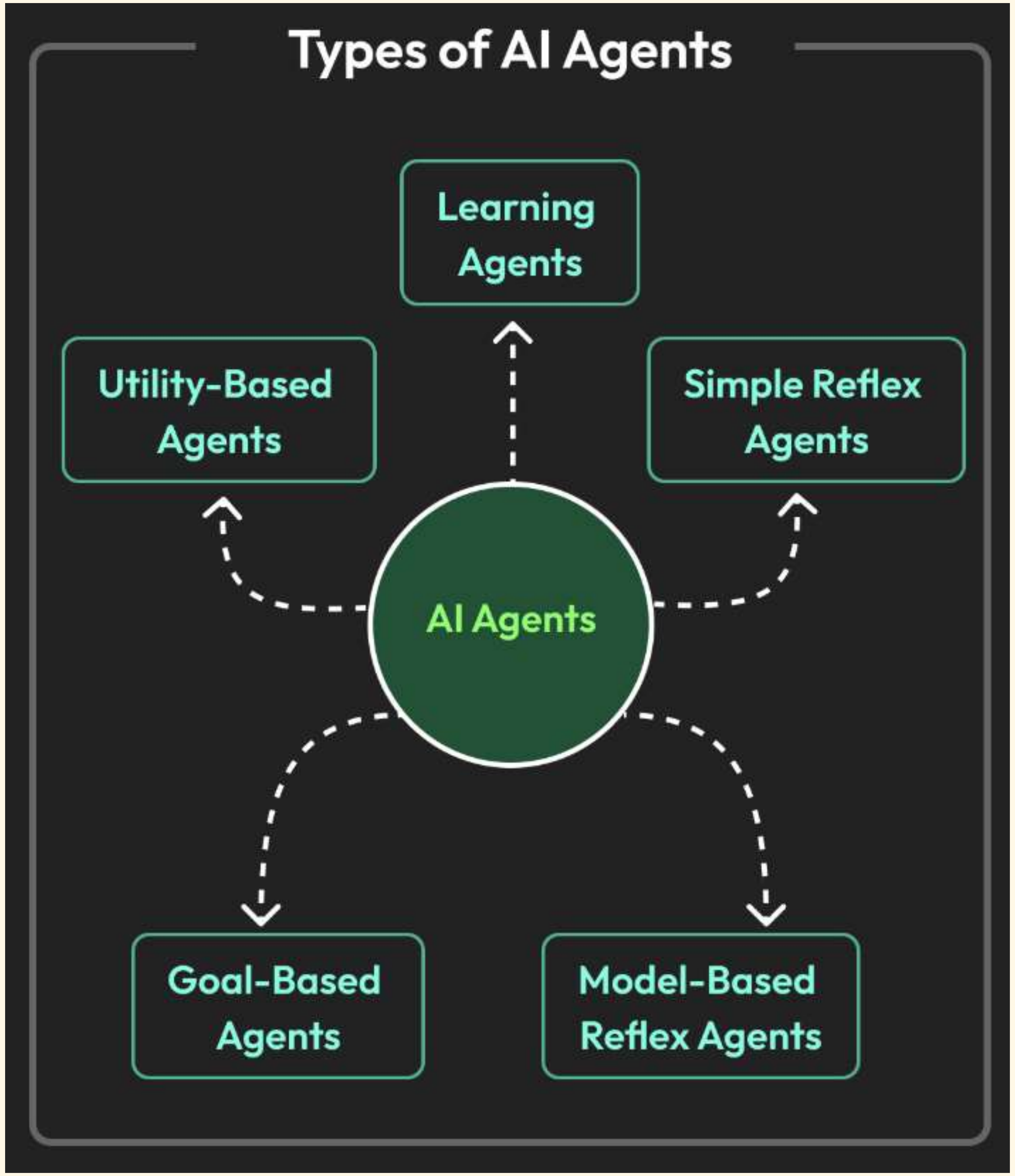
Ai Agents


How AI Agents Work?
Human Control
Delegate Tasks
Tools
API
API Calls
WWW
Access Internet
Autonomous Action
AI Agent
Interpret Code
Access
Memory
Reactivity
Environment

Key characteristics of AI agents are as follows

- An agent can perform autonomous actions without constant human intervention. Also, they can have a human in the loop to maintain control.
- Agents have a memory to store individual preferences and allow for personalization. It can also store knowledge. A (Large Language Model) LLM can undertake information processing and decision-making functions.
- Agents must be able to perceive and process the information available from their environment.
- Agents can also use tools such as accessing the internet, using code interpreters and making API calls.
- Agents can also collaborate with other agents or humans.

Types of Intelligent and AI Agents



Types of Intelligent and AI Agents

1. Simple Reflex Agents

Description: Work on condition–action rules: "If condition → Do action."

No memory, only respond to the current percept.

Example:

A thermostat: *If temperature < 25°C → turn heater ON*.

A basic vacuum cleaner bot: *If dirt detected → siphon dirt*.

Types of Intelligent and AI Agents

2. Model-Based Reflex Agents

Description: Maintain an **internal model** of the world to handle partially observable environments.

Example:

- Self-driving car: remembers traffic lights it passed even if they are no longer in view.
- Virtual assistant (like Siri/Alexa): keeps track of your conversation context.

Types of Intelligent and AI Agents

3. Goal-Based Agents

Description: Take actions to achieve a **goal state**, not just react.

Sub-Type: Problem-Solving Agents (like route-finding in GPS, puzzle solvers, chess engines).

Example:

- Delivery drone: gets from origin to destination while avoiding obstacles.
- Navigation apps (Waze, Google Maps).

Types of Intelligent and AI Agents

4. Utility-Based Agents

Description: Beyond goals, they maximize a **utility function** (measure of happiness, efficiency, or satisfaction).

Example:

- Netflix recommendation system: recommends shows that maximize your viewing satisfaction.

Types of Intelligent and AI Agents

5. Learning Agents

Description: Improve performance by **learning from past experiences**.

Components:

- *Learning element* (improves the agent).
- *Critic* (gives feedback).
- *Performance element* (does the actual work).

Example:

- Spam filters: learn to detect new spam emails.
- ChatGPT itself: trained from user interactions to improve responses.

Types of Intelligent and AI Agents

6. Collaborative / Multi-Agent Systems

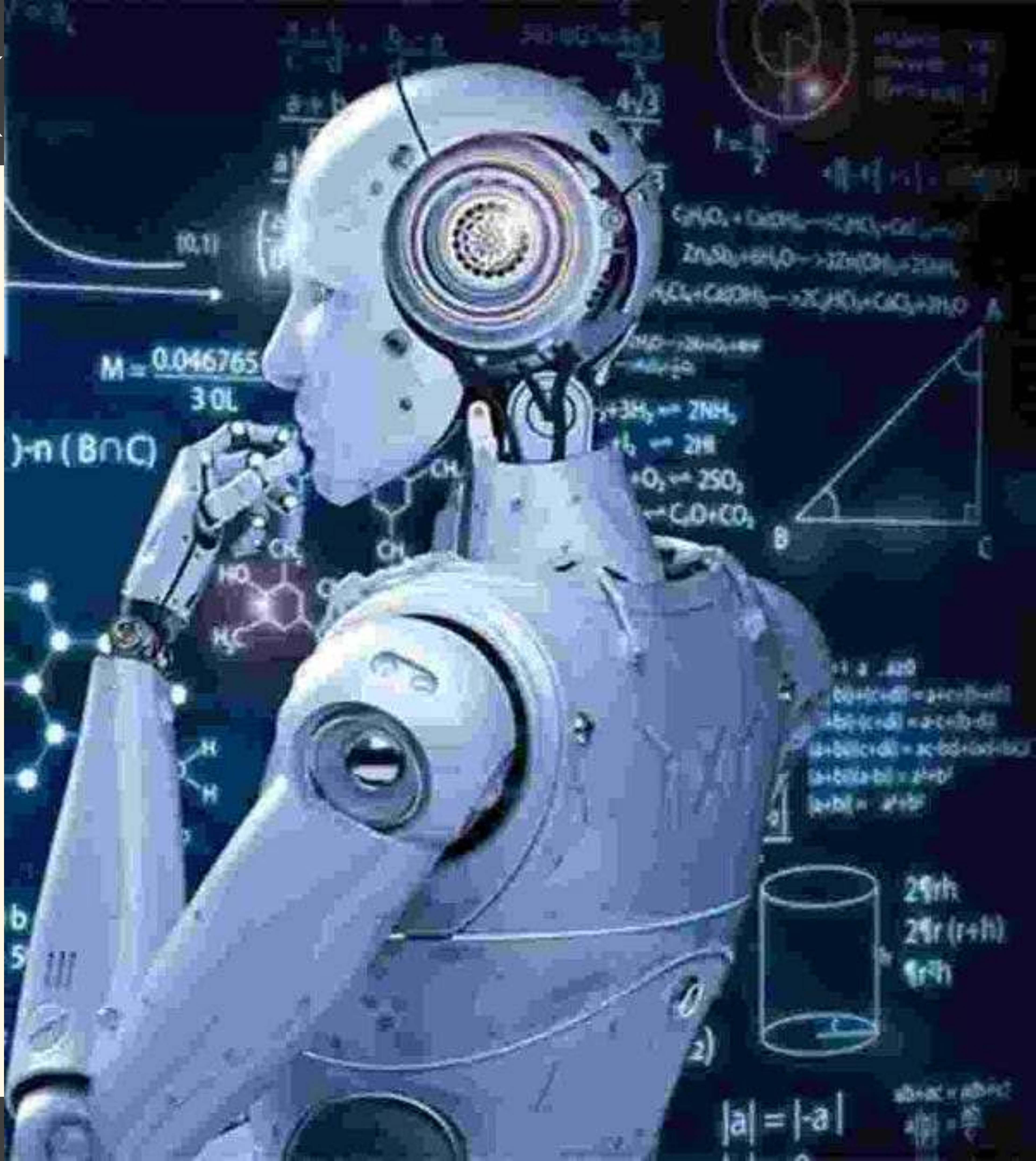
Description: Multiple agents working together (or competing) to solve problems.

Example:

- Swarm robotics (drones coordinating in search & rescue).
- Multiplayer online game AIs.
- Smart grid energy systems with many agents balancing supply & demand.

Problem Solving Agents (

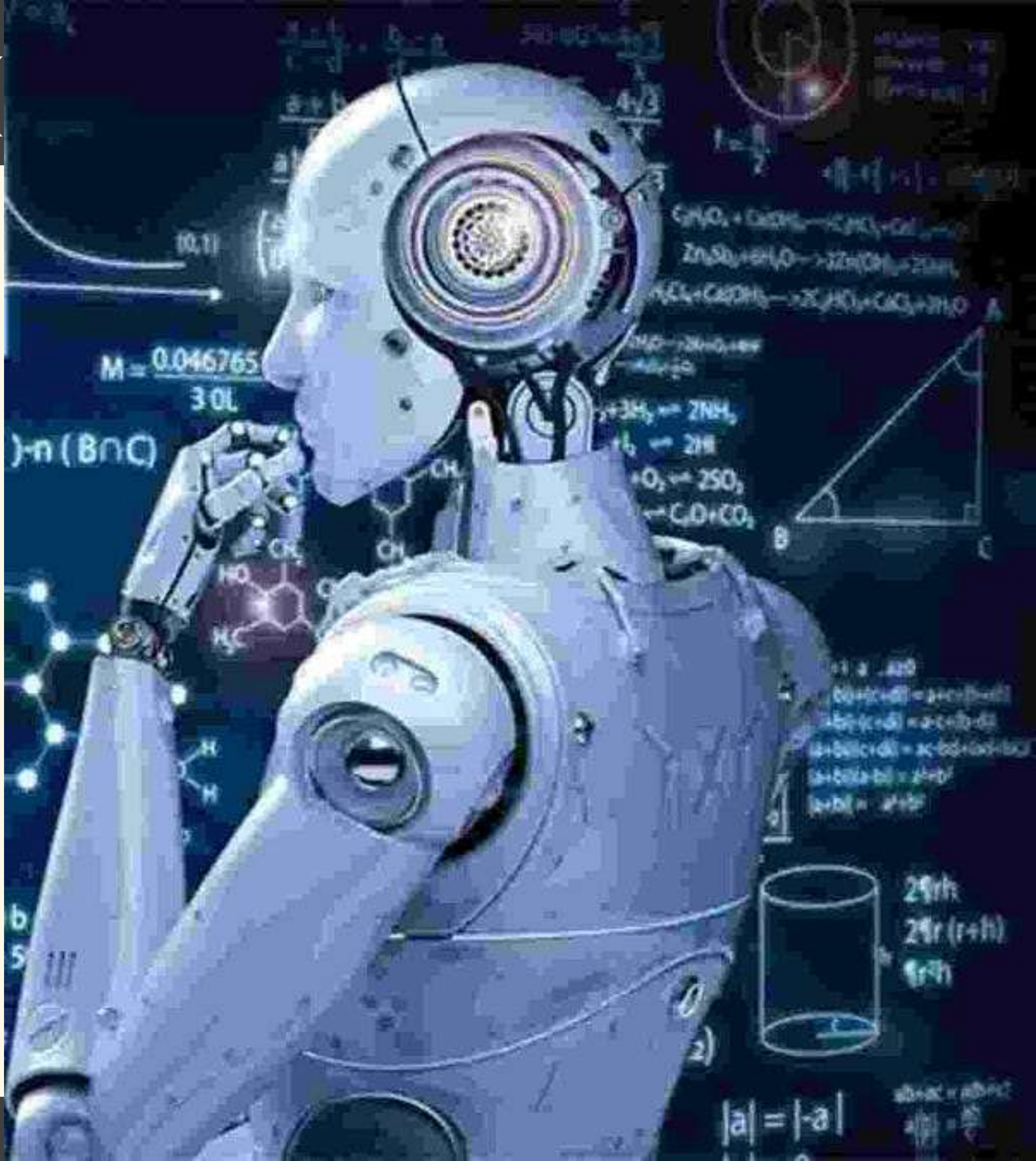
PSA is a goal-based agent that decides what to do by finding sequences of actions that lead to desirable states.



Problem Solving Agents (

A PSA must:

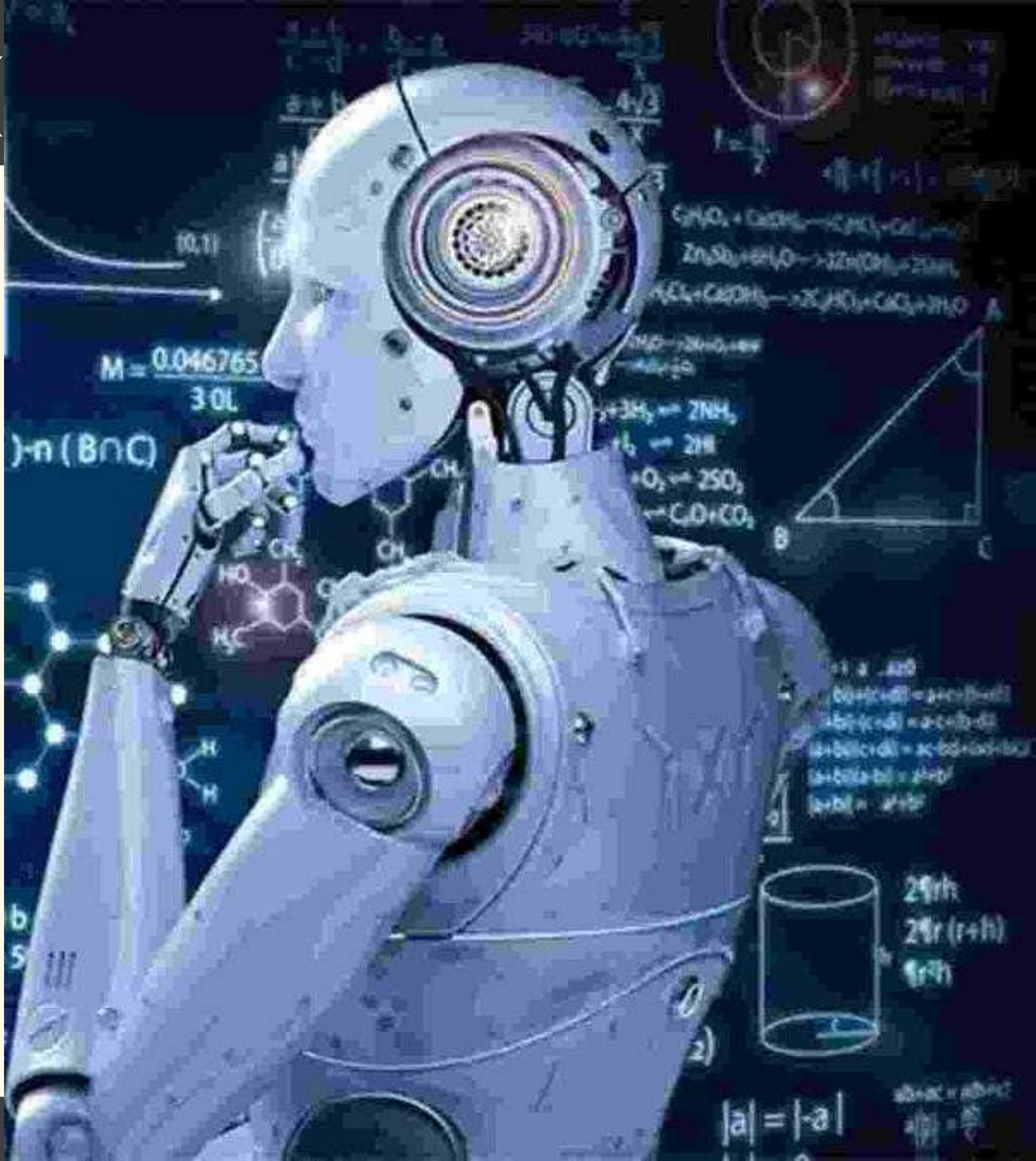
- formulate its problem and goal
- search for the action sequence that makes it achieve its goal
- execute the best action sequence it has found



Problem Solving Agents (

Terminologies:

- **State:** a representation of problem elements at a given moment or a (representation of) a physical configuration.
- **Goal:** the set of world states that we want to reach
- **Action:** transitions between world states
- **Search Algorithm:** takes a problem as input and returns a solution in the form of an action sequence
- **Execution:** agent performs the action sequence to reach the goal



Problem Solving Agents (PSA) - Analogy

Imagine you're planning a **trip to another city**:

- **You (the traveler)** = the **Problem-Solving Agent**
- **Your current city** = the **initial state**
- **Your destination city** = the **goal state**
- **Possible routes (bus, car, train, plane)** = the **set of actions/operators**
- **Google Maps or Waze** = the **search strategy**
- **The map of all roads** = the **state space**

How this works?

- **Formulate the problem** → You decide *where you want to go*.
- **Define states and actions** → You identify routes, roads, and transportation modes.
- **Search for a solution** → You check different routes (shortest path, fastest time, cheapest).
- **Execute the plan** → You travel using the chosen route.

Just like a traveler, a PSA must:

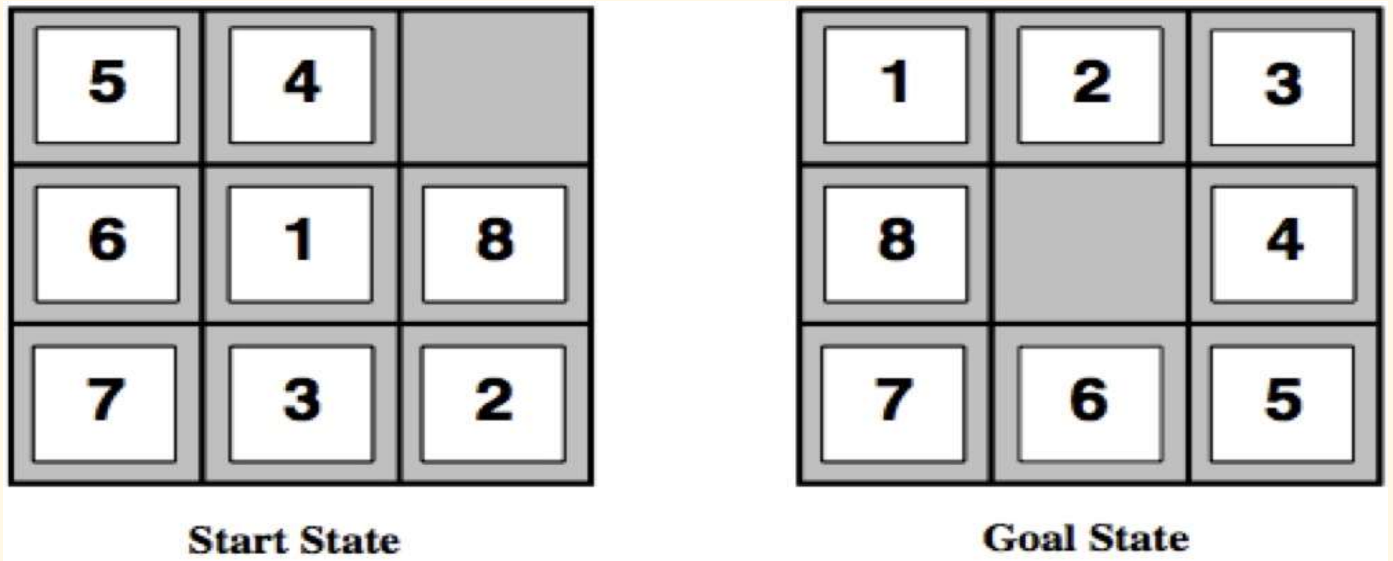
- Perceive the environment (map, traffic conditions).
- Decide the best course of action (shortest path, cheapest ride, or least traffic).
- Act to reach the goal (follow the route).

Problem Solving Agents (PSA) Example

The 8-puzzle consists of a 3 x 3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The objective is to reach the goal state.

Action

- Moving a tile **up, down, left, or right** into the blank space (when possible).
- Example: If the blank is in the middle, four tiles can move into it.



Search Algorithm

- The agent uses a search strategy to find a sequence of actions leading from the start state to the goal state.

Examples:

- **Breadth-First Search (BFS):** explores all states level by level.
- **A* Search (with Manhattan distance heuristic):** finds the shortest path efficiently by estimating the number of moves left.

Execution

- Once the solution (action sequence) is found, the agent executes the moves step by step until the puzzle reaches the goal state.

Example action sequence:

- Move tile 4 right → Move tile 1 up → Move tile 8 left → … until goal achieved.

Problem Solving Agents (PSA) Example

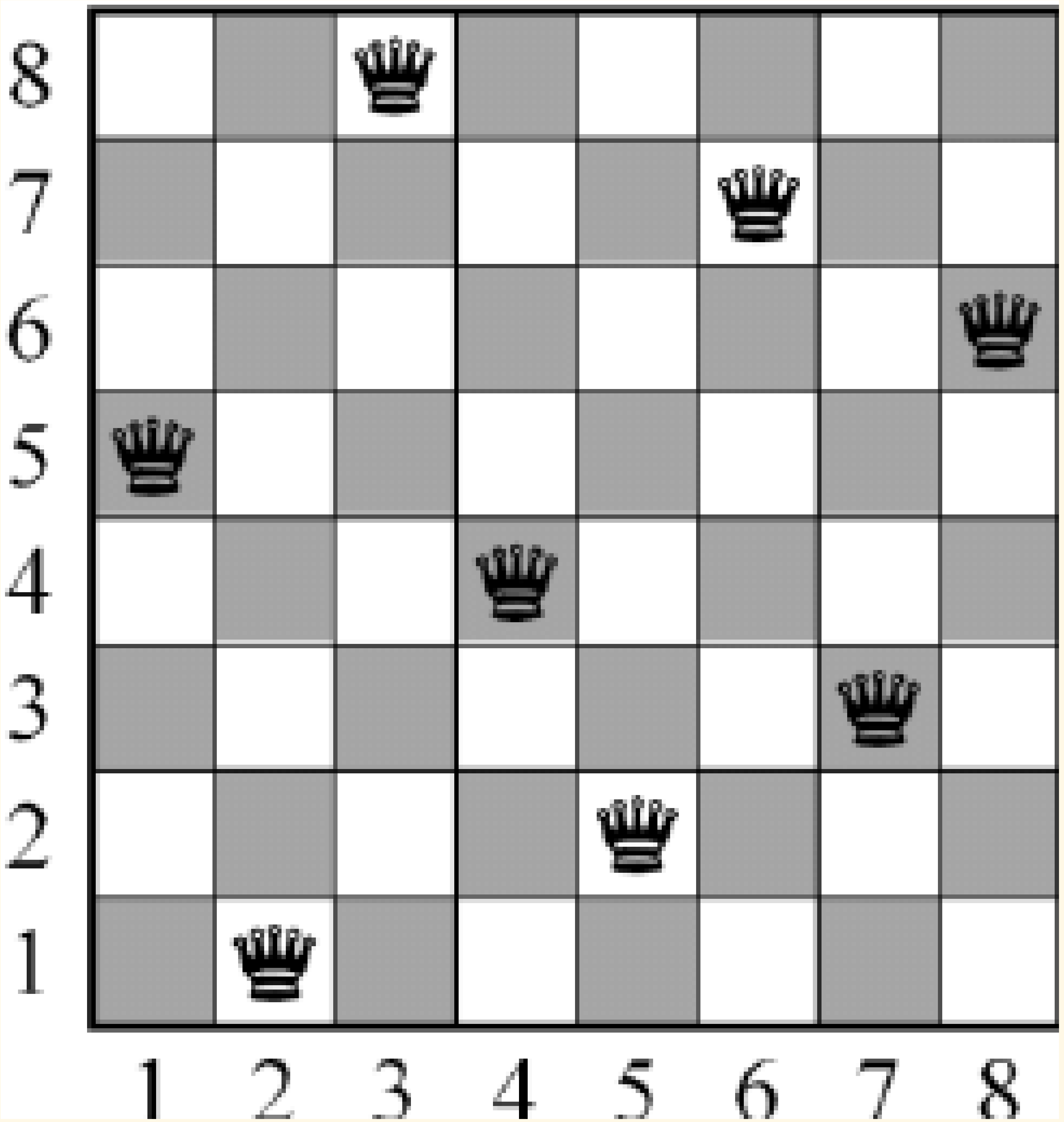
The 8-puzzle is not just a toy problem. It represents **pathfinding and reconfiguration problems**, which appear in:

- **Robotics**: rearranging parts into correct positions.
- **Logistics**: moving packages in a warehouse with limited space.
- **AI in Games**: solving puzzles and NPC decision-making.

Problem Solving Agents (PSA) Example

8-Queens' Problem

- The objective in the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.
- A queen attacks any piece in the same row, column or diagonal.



Problem Solving Agents (PSA) Example - **8-Queens' Problem**

State

- A partial chessboard configuration with some queens placed.
- Example: Placing 4 queens on the board without conflict.

Goal

- A configuration where all 8 queens are placed such that none of them attack each other (i.e., no two queens share the same row, column, or diagonal).

Action

- Place a queen in a valid position on the next row (or move an existing queen if needed).
- Each move changes the current board configuration (state transition).

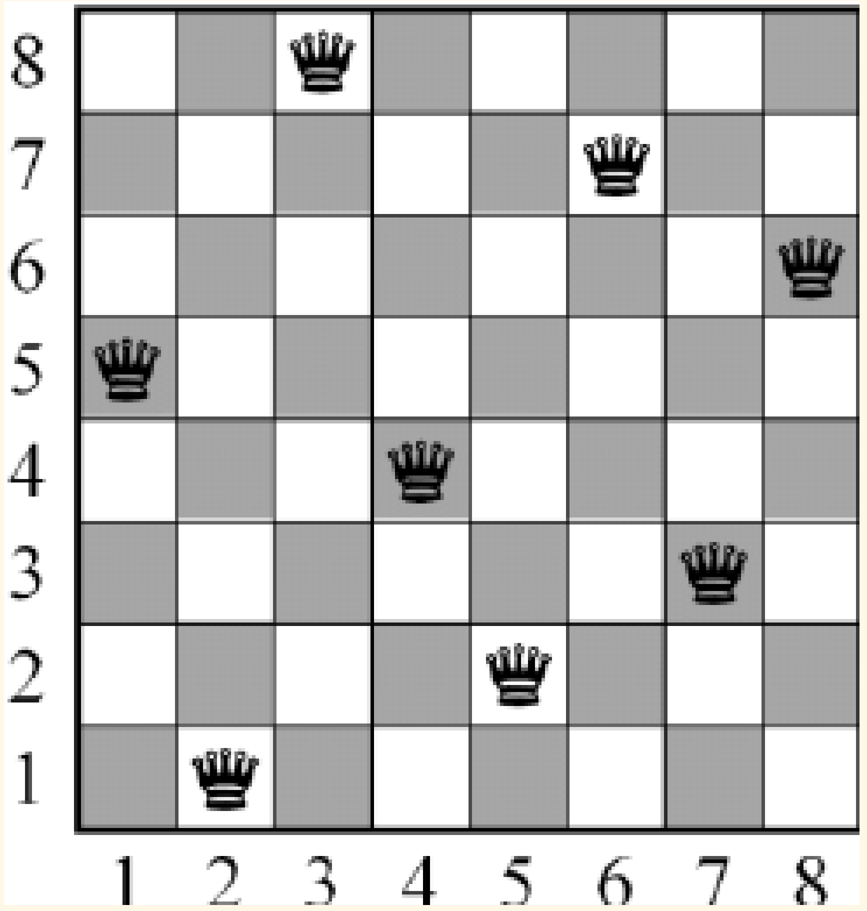
Search Algorithm

Different algorithms can be applied:

- Backtracking Search: Place queens one by one, and if a conflict arises, backtrack and try a new position.

Execution

- Once the algorithm finds a valid configuration (solution state), the agent places queens according to the solution sequence → Goal achieved.



Problem Solving Agents (PSA) Real World Applications

Imagine scheduling exams for 8 university courses:

- Each course (queen) must be placed in a time slot (row/column) without conflicting with other courses that share students (diagonal conflicts).
- The AI scheduler acts like the problem-solving agent, testing different schedules until it finds one with no conflicts.

Real-World Applications of Problem-Solving Agents

1. Navigation Systems (e.g., Google Maps, Waze)

State: Current location on the map.

Goal: Destination address (e.g., from BUCS to SM Legazpi).

Action: Move from one road/turn to another.

Search Algorithm: A* (A-star) or Dijkstra’s Algorithm to find the shortest/fastest path.

Execution: The GPS provides driving instructions, and the driver (or autonomous vehicle) follows them.

Real-World Applications of Problem-Solving Agents

2. Autonomous Delivery Drones

State: Drone's current position + battery level.

Goal: Deliver package to the customer's location.

Action: Fly in a particular direction, recharge, avoid obstacle.

Search Algorithm: A* search or heuristic-based path planning (sometimes with reinforcement learning).

Execution: Drone autonomously follows the computed flight path.

Real-World Applications of Problem-Solving Agents

3. Robotics (Warehouse Robots like Amazon Kiva)

State: Robot's current location in the warehouse.

Goal: Pick up an item and deliver it to a packing station.

Action: Move forward/backward, turn left/right, pick/drop item.

Search Algorithm: A* search with optimization for avoiding collisions.

Execution: Robot navigates aisles, fetches the product, and delivers it.

AI Agent System Architecture

AI Agent

A system with AI agents can be built with different architectural approaches.

1. **Single Agent**: Agents can serve as personal assistants.
2. **Multi-Agent**: Agents can interact with each other in collaborative or competitive ways.
3. **Human Machine**: Agents can interact with humans to execute tasks more efficiently.

AI Agent

Single Agent: Agents can serve as personal assistants.

- A single AI agent operates in an environment, perceives it, and acts to achieve specific goals.
- Acts as a personal assistant or autonomous decision-maker.
- Example Roles:
  - Personal assistant AI (e.g., Siri, Alexa, Google Assistant)
  - Game AI opponent (a chess engine playing against you)
  - Robotics (a cleaning robot navigating your living room)

Architecture Flow:
Environment ↔ Perception (input) → Decision-making (AI logic) → Action (output) ↔ Environment
Key Point: Focuses on **autonomy** and efficiency in solving problems **without needing other agents**.

AI Agent

Multi-Agent: Agents can interact with each other in collaborative or competitive ways.

A system where multiple AI agents coexist, either **collaborating** to solve a task or **competing** in shared environments.

Types of Interactions:

- **Collaborative** → Agents share knowledge/resources to solve complex problems.
- **Competitive** → Agents work against each other (e.g., in auctions, games, negotiations).

Example Roles:

- **Collaborative**: Autonomous delivery drones coordinating routes to avoid collisions.
- **Competitive**: AI traders in stock markets competing for best prices.
- **Simulations**: Traffic management systems where cars are AI agents coordinating flow.

Architecture Flow:

Agent 1 ↔ Environment ↔ Agent 2 ↔ … Agent n (Agents can also directly communicate with each other.)

Key Point: Useful when problems are **too complex for a single agent**, requiring **distributed intelligence**.

AI Agent

Human Machine: Agents can interact with humans to execute tasks more efficiently.

Definition: An AI agent interacts directly with humans to **enhance human abilities** or execute tasks more efficiently.
Role of Humans: Humans provide **guidance, oversight, or collaboration** while the AI provides speed, memory, and automation.
Example Roles:

- **Human-in-the-loop systems** → Doctors assisted by AI in diagnosis.
- **Conversational AI** → Chatbots answering customer service queries.
- **Decision support systems** → AI helping managers analyze big data before deciding.
- **Robotics** → Collaborative robots (cobots) assisting workers in manufacturing.

Architecture Flow: Human ↔ AI Agent ↔ Environment
Key Point: Focuses on **augmenting human capabilities**, combining **human judgment** with **AI efficiency**.

types of problems in AI problem-solving

- ✓ Single-state Problems
- ✓ Multiple-state Problems
- ✓ Contingency Problems
- ✓ Exploratory Problems

types of problems in AI problem-solving

Single-state Problems

- Problems where the agent has a complete and perfect information about the environment and the current state. The outcome of each action is predictable.

Key Feature: Deterministic and fully observable.

Real-world Example:

- **Solving a Sudoku puzzle**: The board is fully known, rules are fixed, and each move (placing a number) has a predictable outcome.
- **Chess against a computer (with visible board)**: All pieces and moves are known; no uncertainty.

types of problems in AI problem-solving

Multiple-state Problems

- Problems where the agent does not know its exact current state. Instead, it must reason across a **set of possible states**.

Key Feature: Partially observable environments.

Real-world Example:

- **Robot in a warehouse with limited sensors**: If GPS is inaccurate or sensors are noisy, the robot may not know its exact location but has to reason over possible positions.
- **Self-driving cars in fog**: The car may not perfectly detect road lines or other vehicles, so it assumes possible positions of objects.

Contingency Problems

- Problems where the outcome depends not only on the agent’s actions but also on **external, unpredictable events**. The agent must prepare contingency plans.

Key Feature: Involves **uncertainty** and requires **if-then strategies**.

Real-world Example:

- **Autonomous drone delivery**: Weather (wind, rain) may affect flight. The agent needs backup routes.
- **Online shopping system**: Payment may fail, item may be out of stock, or user may cancel. AI must adapt to these possible scenarios.

Exploratory Problems

- Problems where the agent has **no prior knowledge** of the environment and must explore to learn about it.

Key Feature: Involves **trial-and-error** and **learning from interaction**.

Real-world Example:

- **Mars rover exploration**: It doesn’t know the terrain beforehand; it learns through sensors as it moves.
- **AI in video game “Minecraft”**: The agent explores the world, gathers resources, and learns strategies without prior knowledge.

Problem

1. Single-state Problems

Definition: The agent has complete knowledge of the environment and can act directly to achieve the goal.

Vacuum World Example:

The vacuum agent knows exactly where the dirt is and where it is located.

If it starts in Room A, and it knows Room A is clean while Room B is dirty, it simply moves to Room B and siphons up the dirt.

Type: Single-state because the agent has **full knowledge** and doesn't need to guess.

2. Multiple-state Problems

Definition: The agent doesn't know exactly which state it is in; it must reason over multiple possible states.

Vacuum World Example:

- The vacuum agent doesn't know its starting position (A or B).
- It knows one room is dirty, but it doesn't know which one.
- The agent must consider **both possibilities** (being in A or B) and plan actions that work in either case, like siphoning dirt first, then moving.

Type: Multiple-state because the agent is **uncertain of its initial state** and must consider multiple possible situations.

Problem

3. Contingency Problems

Definition: The outcome depends on external conditions, so the agent needs a **conditional plan** (if–then strategy).

Vacuum World Example:

The agent doesn't know whether Room B is dirty or clean until it goes there.

It forms a plan like:

- If Room A is dirty → clean it, then go to Room B.
- If Room A is clean → go directly to Room B.
- When in Room B → if dirty, clean; else stop.

Type: Contingency because the agent makes **conditional decisions** based on what it perceives during execution.

4. Exploratory Problems

Definition: The agent has **no prior knowledge** of the environment and must **explore** to gather information.

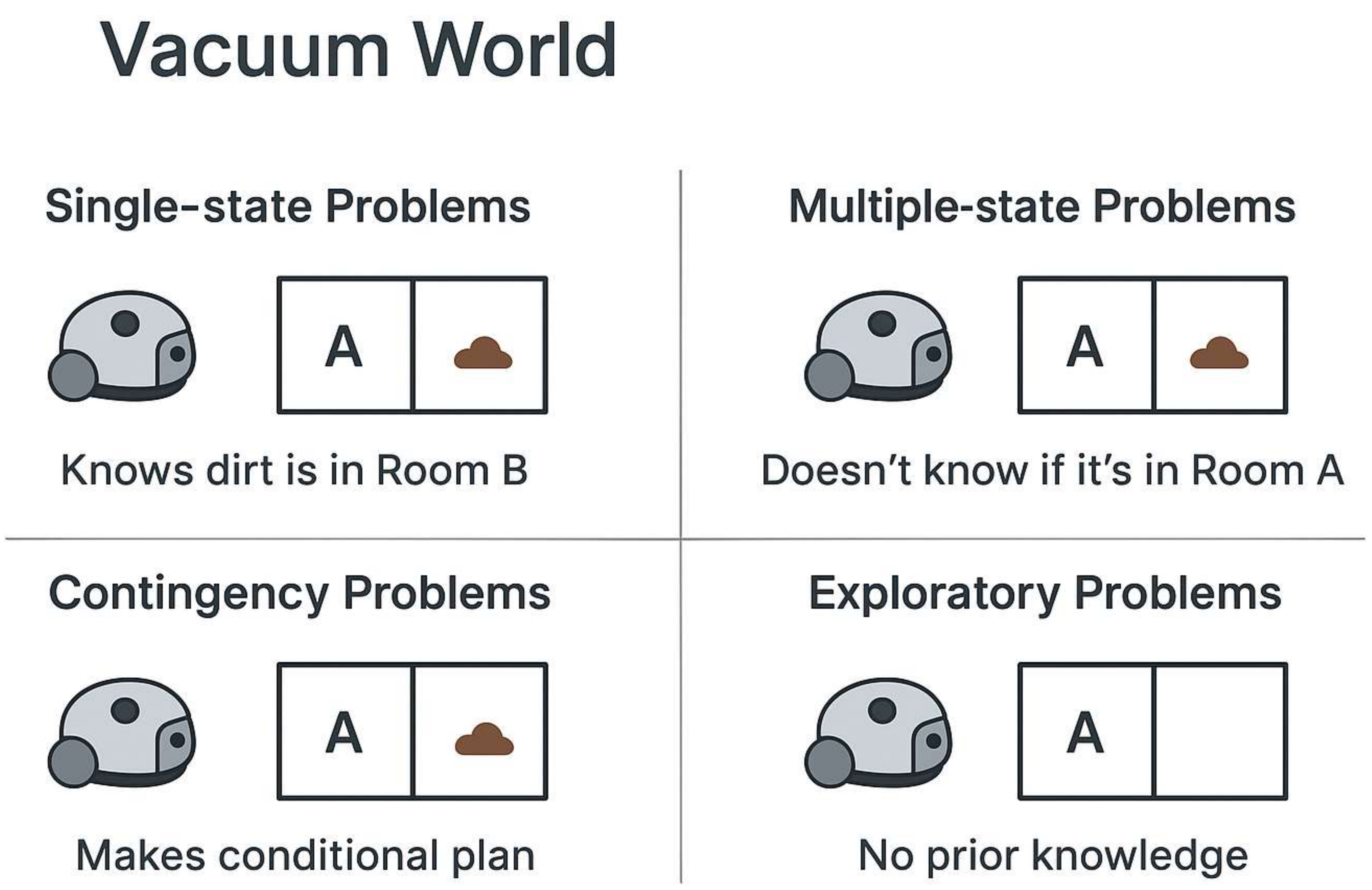
Vacuum World Example:

The vacuum agent doesn't know the layout (number of rooms, their locations, or where dirt is).

It must explore randomly: move left, right, siphon, etc., until it gradually learns the map and dirt locations.

Type: Exploratory because the agent must **explore and learn** the environment from scratch.

Vacuum World AI Problem



Components of a Problem in AI

States (Initial State)

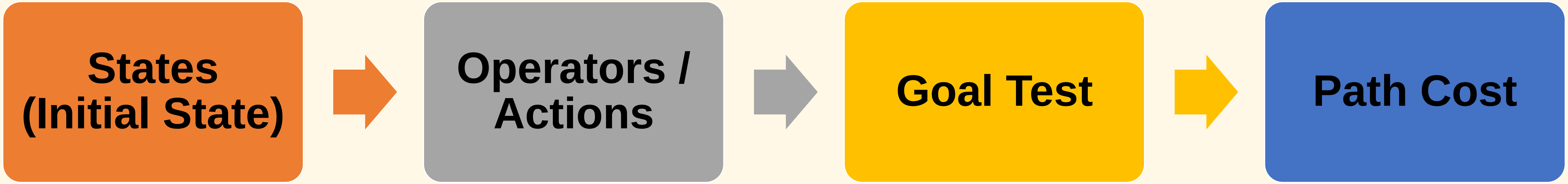
A **state** is a representation of the problem's elements at a given moment or a snapshot of the world.

The **initial state** is where the agent begins before performing any action.

Each state should be well-defined so the agent can reason about possible transitions.

Example:

- In the **8-puzzle problem**, the initial state is the starting arrangement of tiles.
- In **Google Maps navigation**, the initial state is the starting location.



Components of a Problem in AI

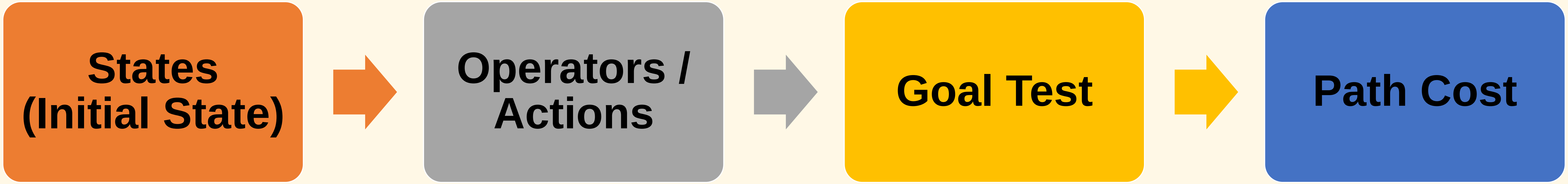
Operators / Actions (Solution)

These are the **possible actions** an agent can take to move from one state to another.

Each operator defines a **transition rule**.

Example:

- In the **vacuum world**, the actions are: *move left, move right, siphon dirt*.
- In **chess**, the actions are the legal moves of a piece.



Components of a Problem in AI

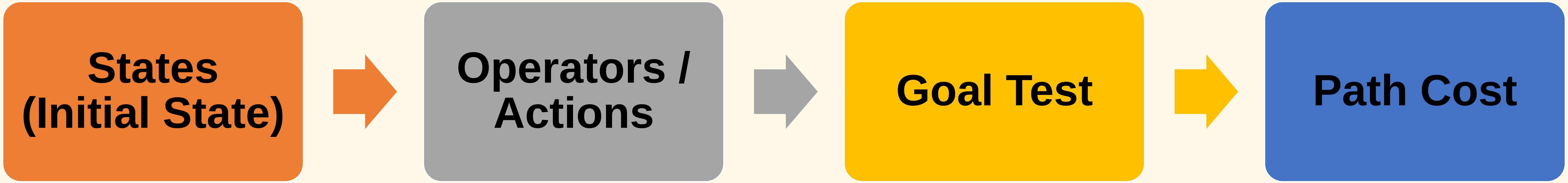
Goal Test

This is a procedure to check if the current state is the desired **goal state**.

Determines if the problem has been solved.

Example:

- In the **8-queens problem**, the goal test checks if eight queens are placed without threatening each other.
- In **package delivery by a robot**, the goal is to verify if the package reached the destination.



Components of a Problem in AI

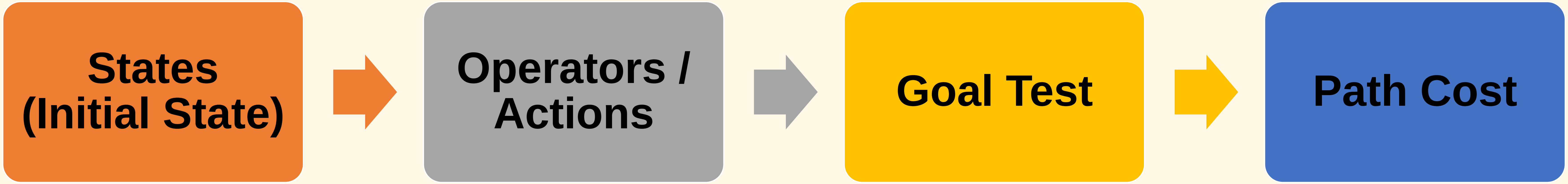
Path Cost

A numerical function that evaluates the **quality of a path** taken from the initial state to the goal.

Helps find not just *any solution*, but the **best solution (optimal)**.

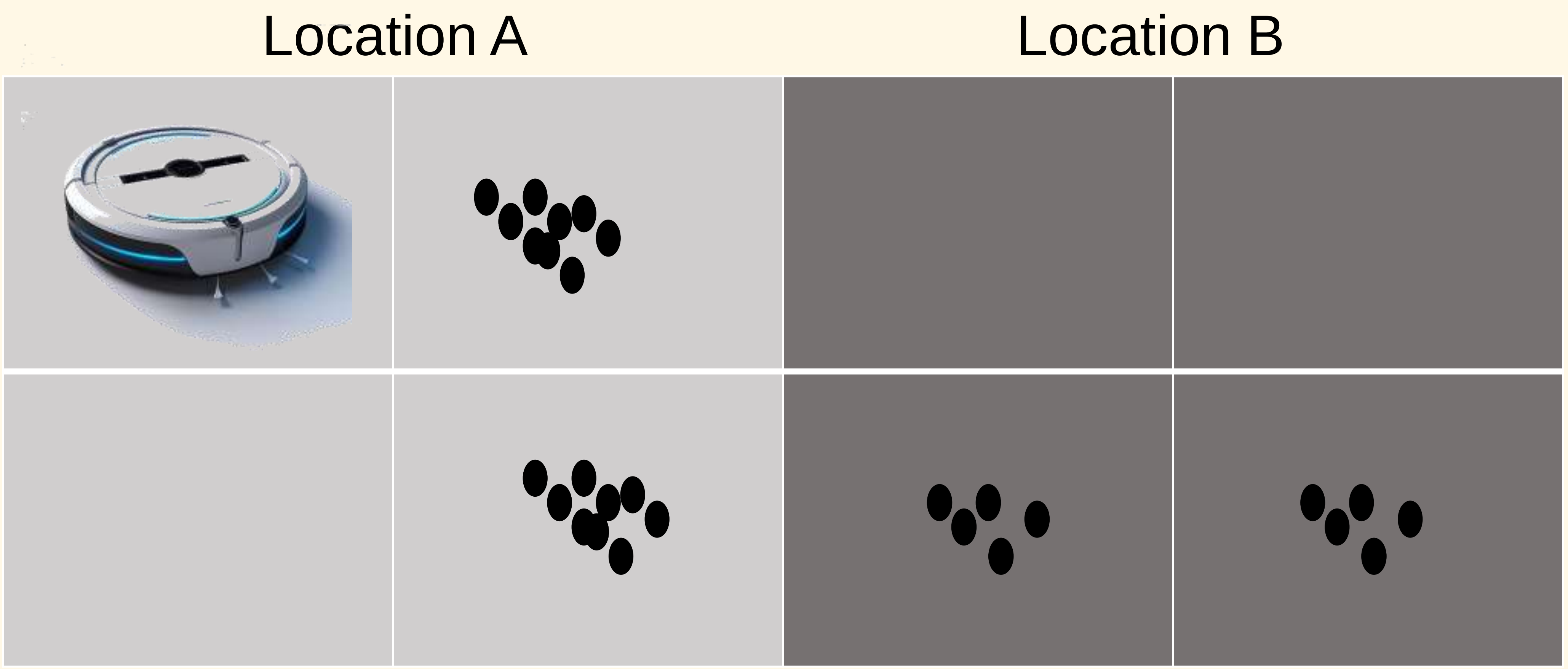
Example:

- In **Google Maps**, path cost can be: distance, travel time, or fuel usage.



Example: vacuum world

- States: ?
- Actions: ?
- Goal Test: ?
- Path Cost: ?



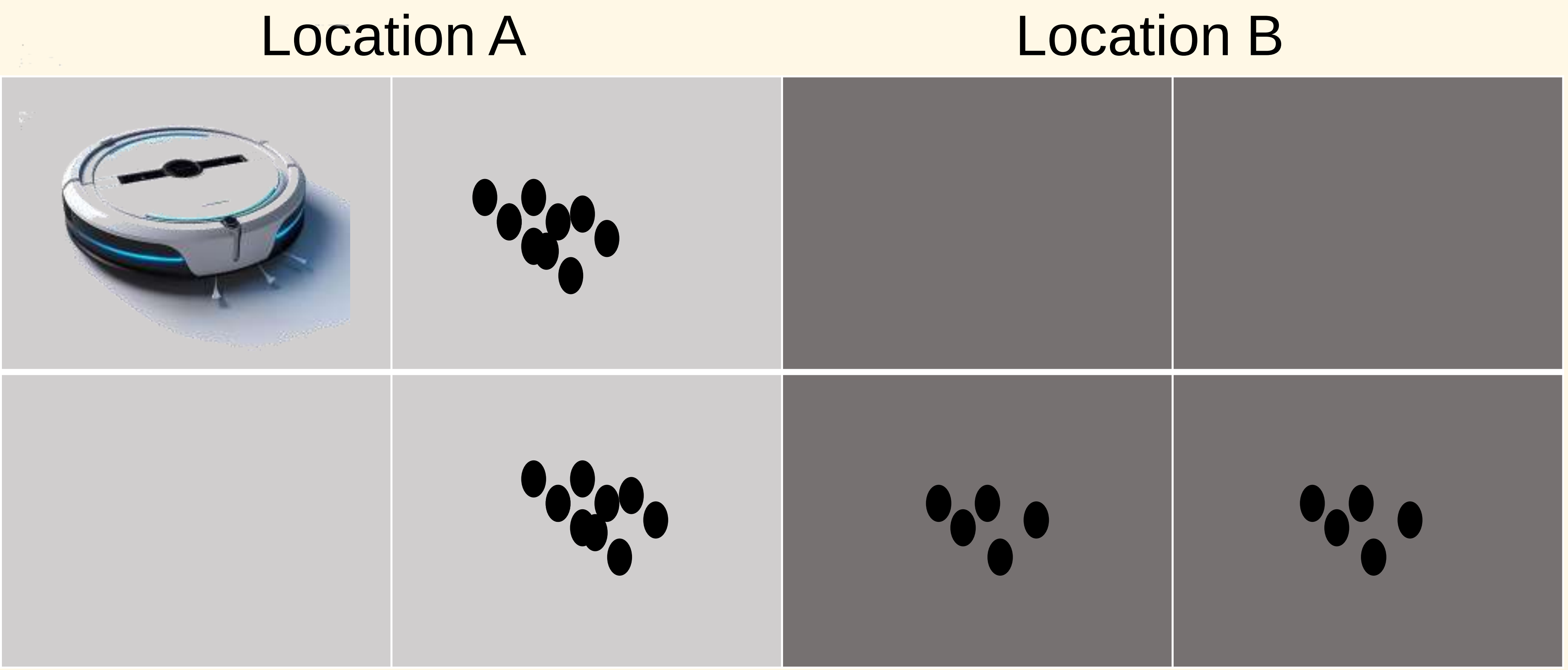
Example: vacuum world

States: two locations with or without dirt: 2 x 2^2=8 states

Actions: *Left*, *Right*, Up, Down, *Siphon*

Goal Test: Check whether squares are clean

Path Cost: Number of actions to reach goal



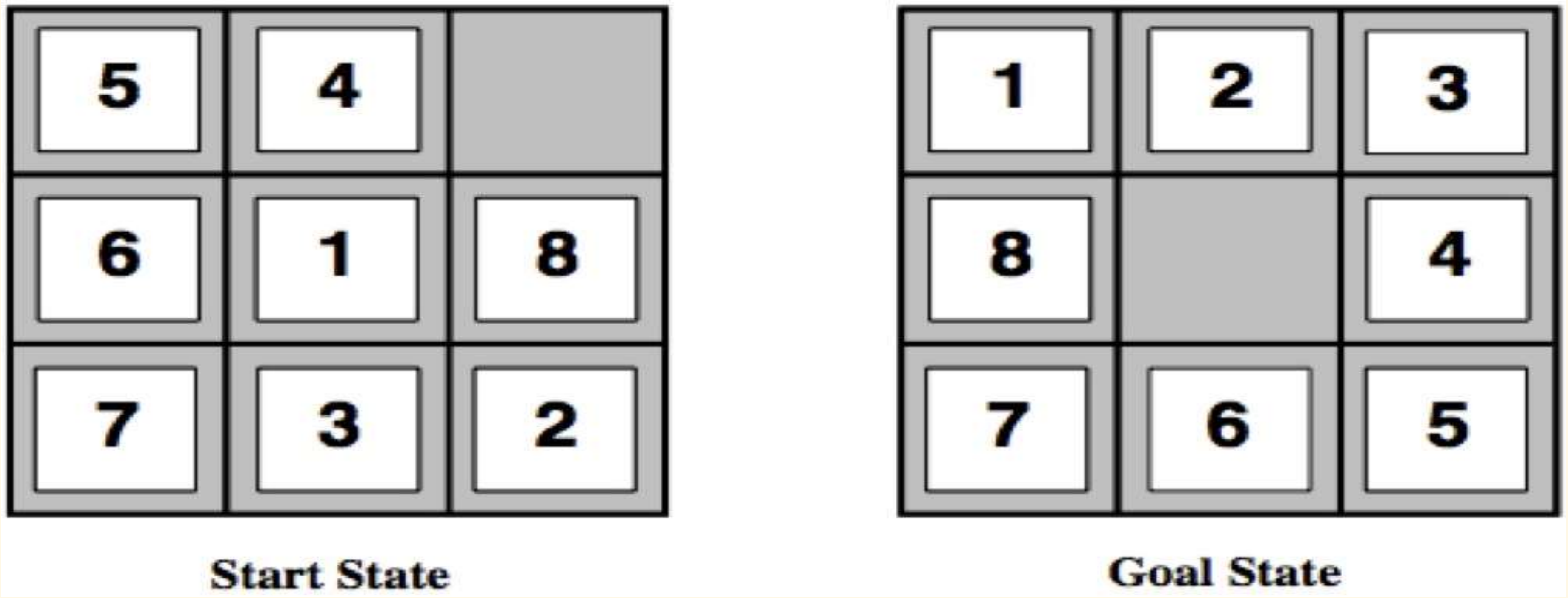
Example: 8-Puzzle

States: ?

Actions: ?

Goal Test: ?

Path Cost: ?



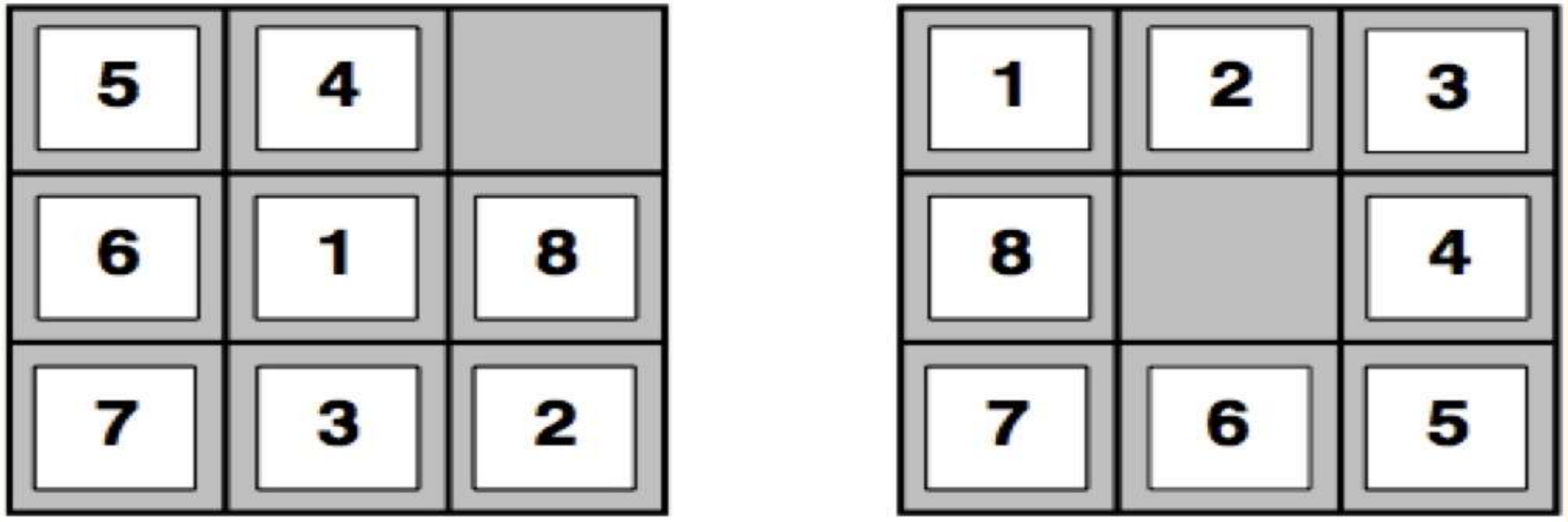
Example: 8-Puzzle

States: a state description specifies the location of each of the eight tiles in one of the nine squares.

Actions: blank moves left, right, up, or down

Goal Test: Does the state match the desired configuration?

Path Cost: each step costs 1, so the path cost is just the length of the path

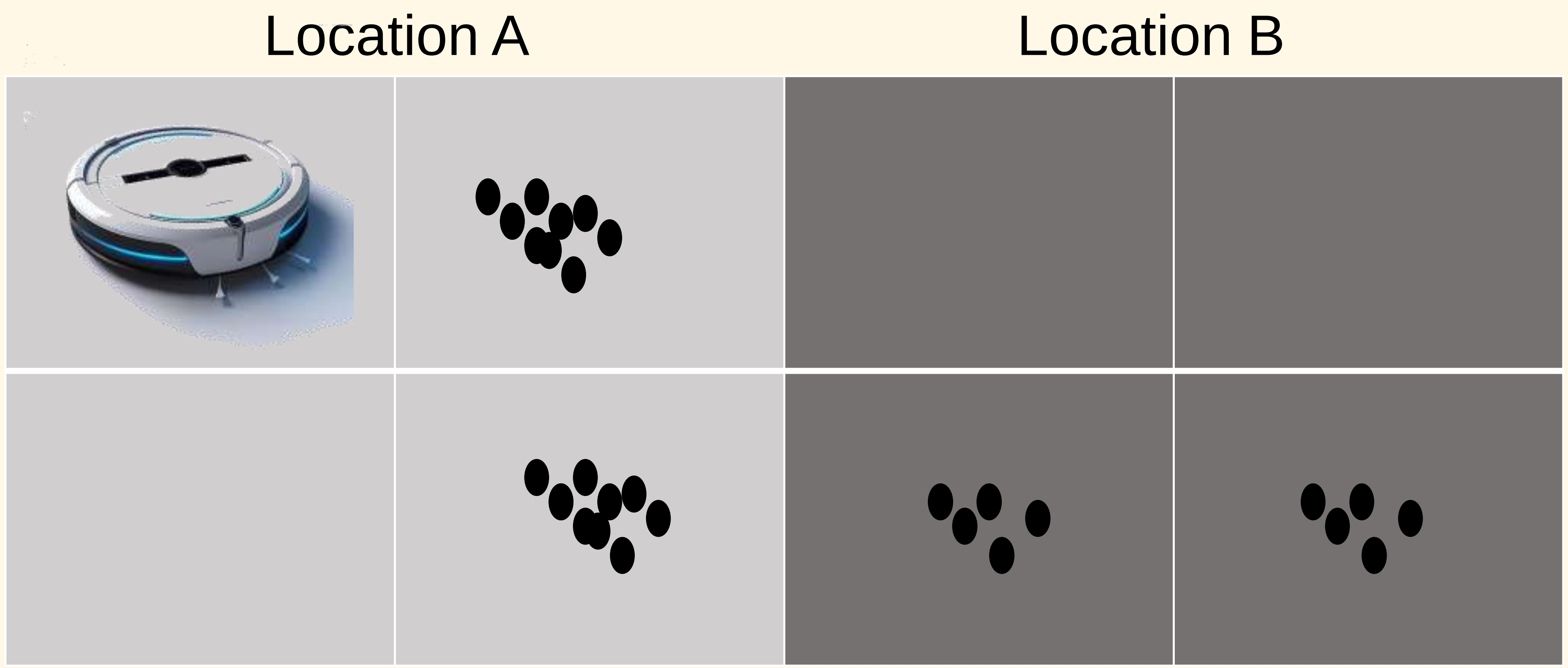


Start State

Goal State

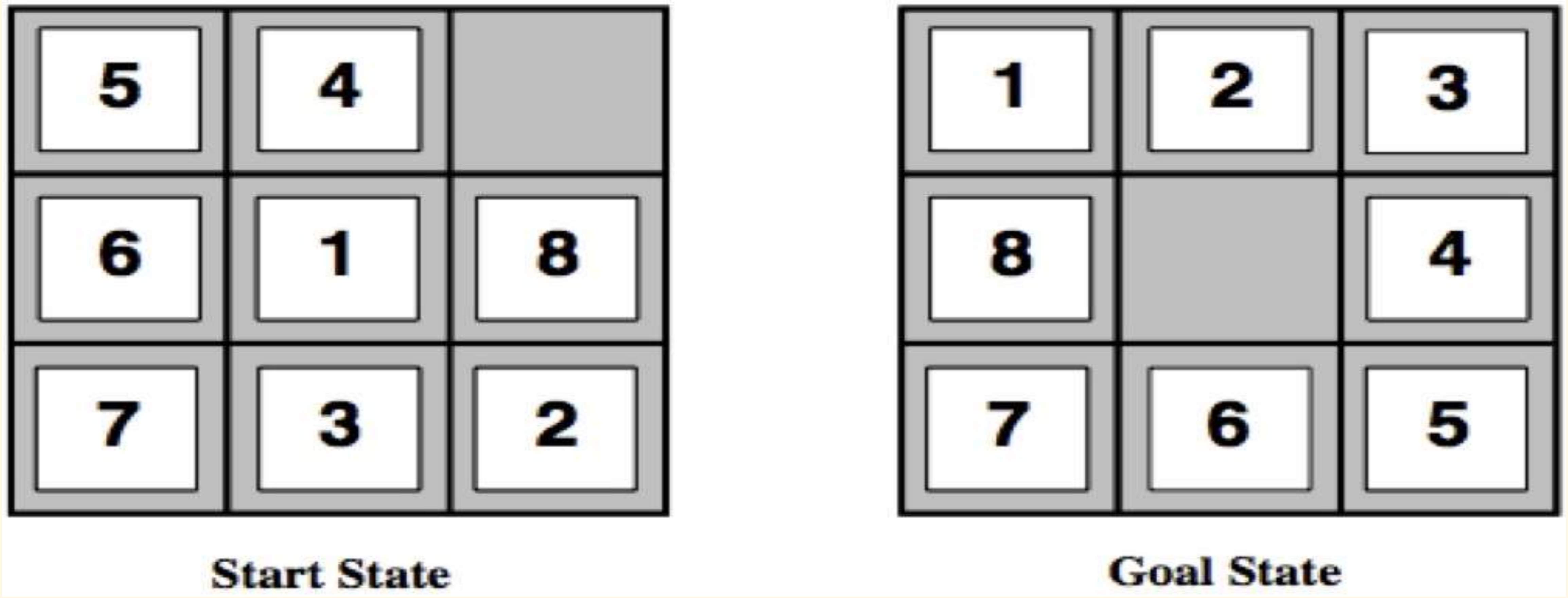
PART I -Assessment

Path cost: Each step costs 1. What is the total number of actions to reach the goal?



PART I -Assessment

Path Cost: Each step costs 1. What is the total number of actions to reach the goal?



Part II

types of problems in AI problem-solving

1. Uninformed (Blind) Search

- These algorithms do not have domain-specific knowledge beyond the problem definition. They explore the search space blindly.

2. Informed (Heuristic) Search

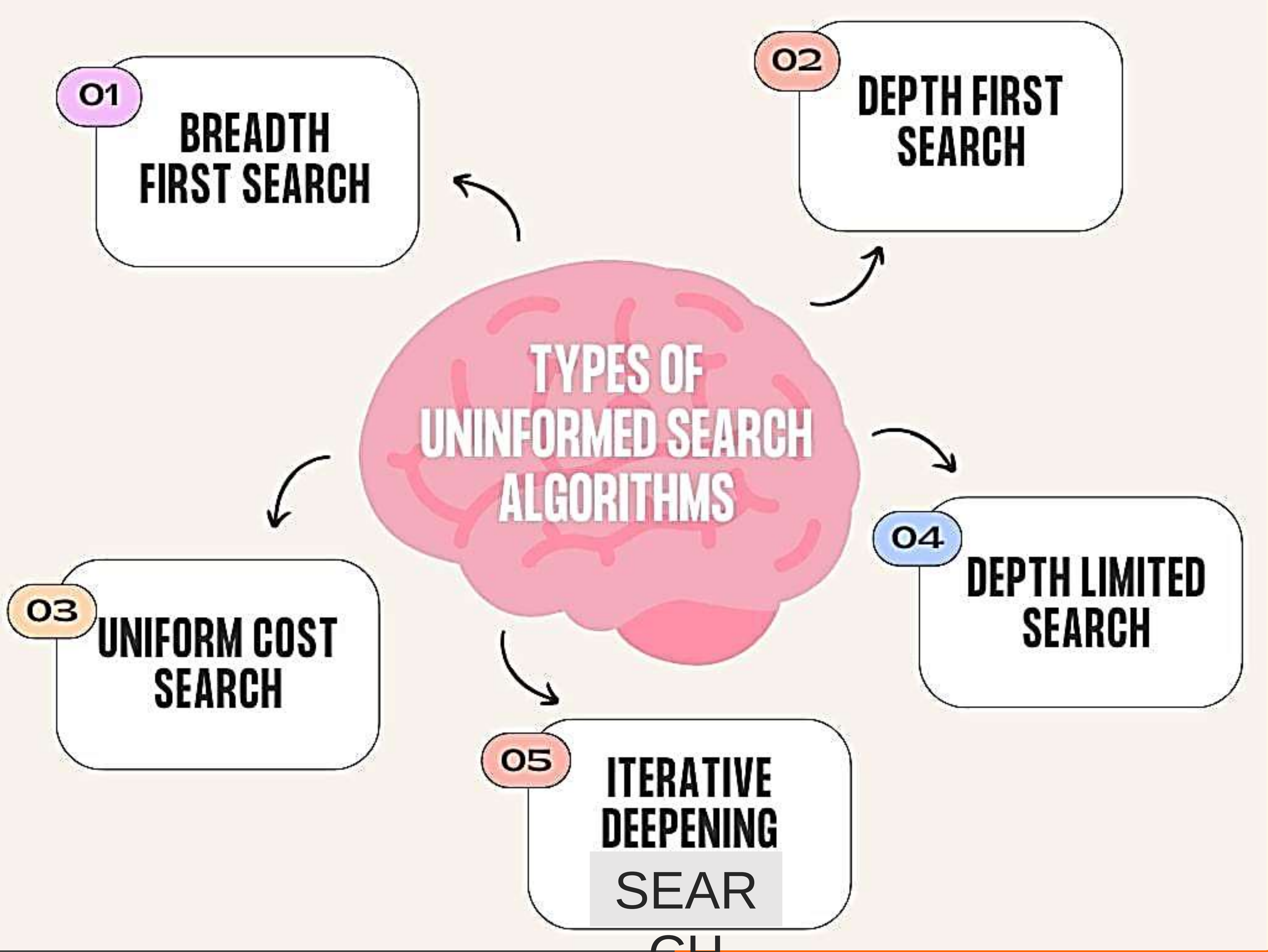
- Uses **heuristics** (problem-specific knowledge) to guide search, making it more efficient.

Analogy Example: Searching for a Friend’s House

Basic Search Algorithms in AI

- **Uninformed Search:**
  You only know the street map. You try every possible street until you (hopefully) arrive.
  “I don’t know which way is better—I’ll just explore everything systematically.”
- **Informed Search:**
  You not only have the map, but also a hint: *“My friend’s house is near the church tower you can see from far away.”*
  Now you can **prioritize streets that seem to head toward the church tower**, instead of checking all streets blindly.

Uninformed search algorithms



Uninformed search algorithms

1. **Breadth-First Search (BFS)**
   - Explores all nodes at the current depth before going deeper.
   - Guarantees the *shortest solution* if all step costs are equal.
   - Uses a **queue (FIFO)**.

   Example: Search Tree for the Vacuum World AI

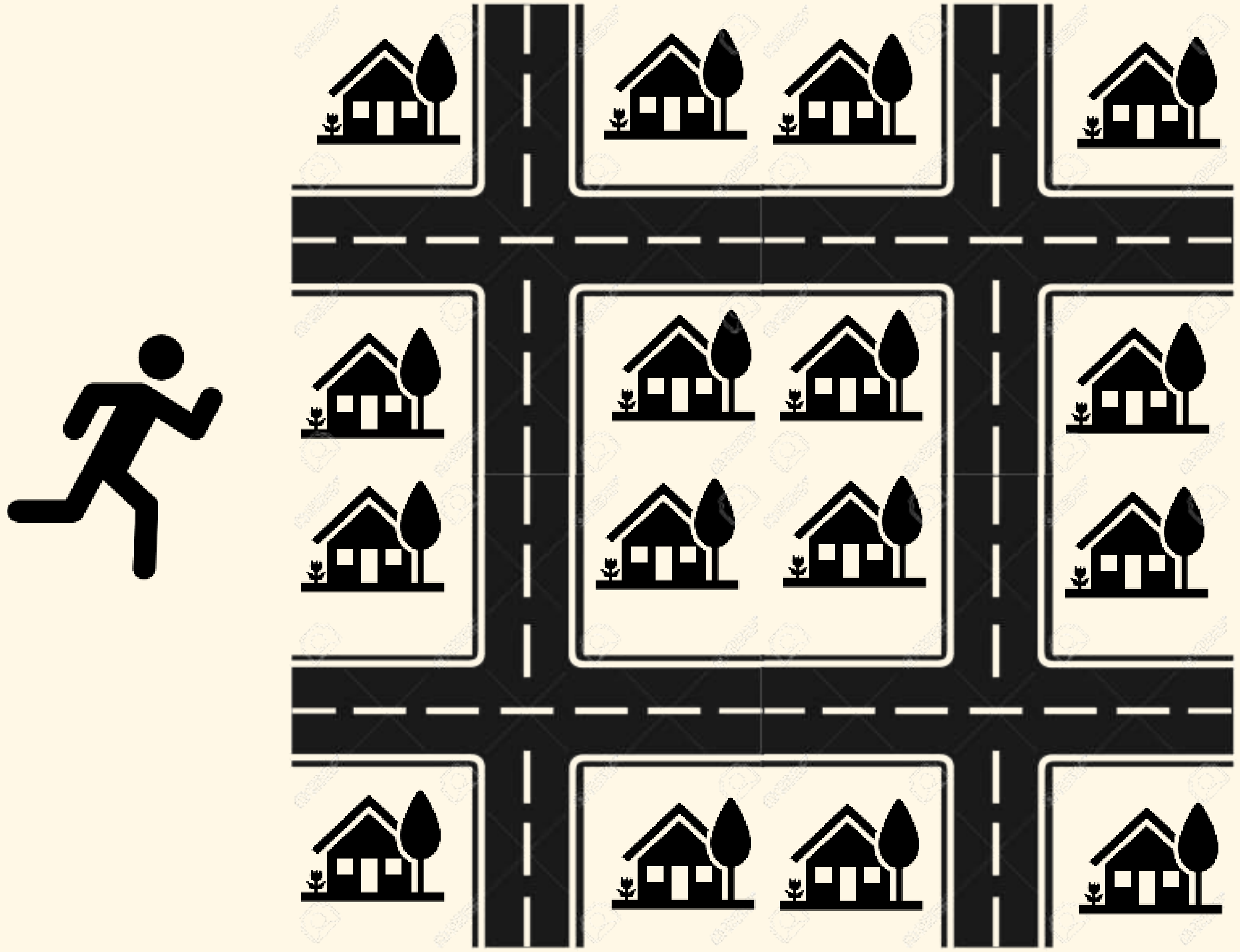
Characteristics:

- Expands all nodes level by level.
- Guarantees finding the shortest solution path (minimum number of actions).
- Requires high memory to store frontier.

Uninformed search algorithms

1. Breadth-First Search (BFS)

- **BFS = Layer-by-layer search.**
- Just like checking houses in a **stretch from your starting point**, BFS explores all possibilities closest first, ensuring the shortest path is found—but it can be slow and memory-hungry if the neighborhood is huge.



Uninformed search algorithms

1. Breadth-First Search (BFS

Initial State: (A_dirty, B_dirty), Agent at A

Level 0 (Root):

(A_dirty, B_dirty) [Agent at A]

Level 1 (Expand root):

Action: SIPHON → (A_clean, B_dirty)

Action: RIGHT → (A_dirty, B_dirty), Agent at B

Level 2:

From (A_clean, B_dirty):

SIPHON → (A_clean, B_dirty) [no effect]

RIGHT → (A_clean, B_dirty), Agent at B

From (A_dirty, B_dirty), Agent at B:

SIPHON → (A_dirty, B_clean)

LEFT → (A_dirty, B_clean), Agent at A

Level 3:

Eventually, BFS finds → (A_clean, B_clean) ✅ (Goal State)

```
(A_dirty, B_dirty, Agent=A)
              /             \
        SIPHON/                 \RIGHT
            /                        \
(A_clean, B_dirty, Agent=A)    (A_dirty, B_dirty, Agent=B)
        |                            |
      SIPHON                       SIPHON
        |                            |
      RIGHT                        LEFT
        |                            |
(A_clean, B_dirty, Agent=B)   (A_dirty, B_clean, Agent=A)
                    |
                  SIPHON
                    |
          (A_clean, B_clean) ✅ GOAL
```

Uninformed search algorithms

2. **Depth-First Search (DFS)**
    - Explores as far down one branch as possible, then backtracks.
    - Memory efficient but may get stuck in deep/looping paths.
    - Uses a **stack (LIFO)** or recursion.

    Example: File directory search.

Characteristics:

- Goes deep along one branch until it finds the goal.
- Uses less memory than BFS (only stores current path).
- May find a solution faster, but not guaranteed shortest path.
    - E.g. if DFS explored RIGHT first at root, it would take longer before reaching the goal.

Uninformed search algorithms

2. Depth-First Search (DFS)

- **DFS = Depth-first exploration.**
- Like walking down one street all the way without checking nearby streets first.
- It uses **less memory** than BFS but doesn't guarantee the **shortest path** (you might pass your friend's house early on but only find it after exploring a long dead-end first).

Uninformed search algorithms

2. Depth-First Search (DFS)

(A_dirty, B_dirty, Agent=A)
Step 1 (Expand root, pick first action):
SIPHON → (A_clean, B_dirty, Agent=A)
Step 2 (Expand child):
From (A_clean, B_dirty, A):

- First action = SIPHON → no change
- Second action = RIGHT → (A_clean, B_dirty, Agent=B)

Step 3 (Expand further):
From (A_clean, B_dirty, B):

- SIPHON → (A_clean, B_clean, Agent=B) ✅ **Goal Found**

```
(A_dirty, B_dirty, Agent=A)
        |
      SIPHON
        |
(A_clean, B_dirty, Agent=A)
        |
      RIGHT
        |
(A_clean, B_dirty, Agent=B)
        |
      SIPHON
        |
 (A_clean, B_clean, Agent=B) ✅ GOAL
```

Uninformed search algorithms

3. **Uniform Cost Search (UCS)**
   - Expands the node with the **lowest path cost** (not just shallowest).
   - Useful when step costs are not uniform.
   - Equivalent to **Dijkstra's Algorithm**.

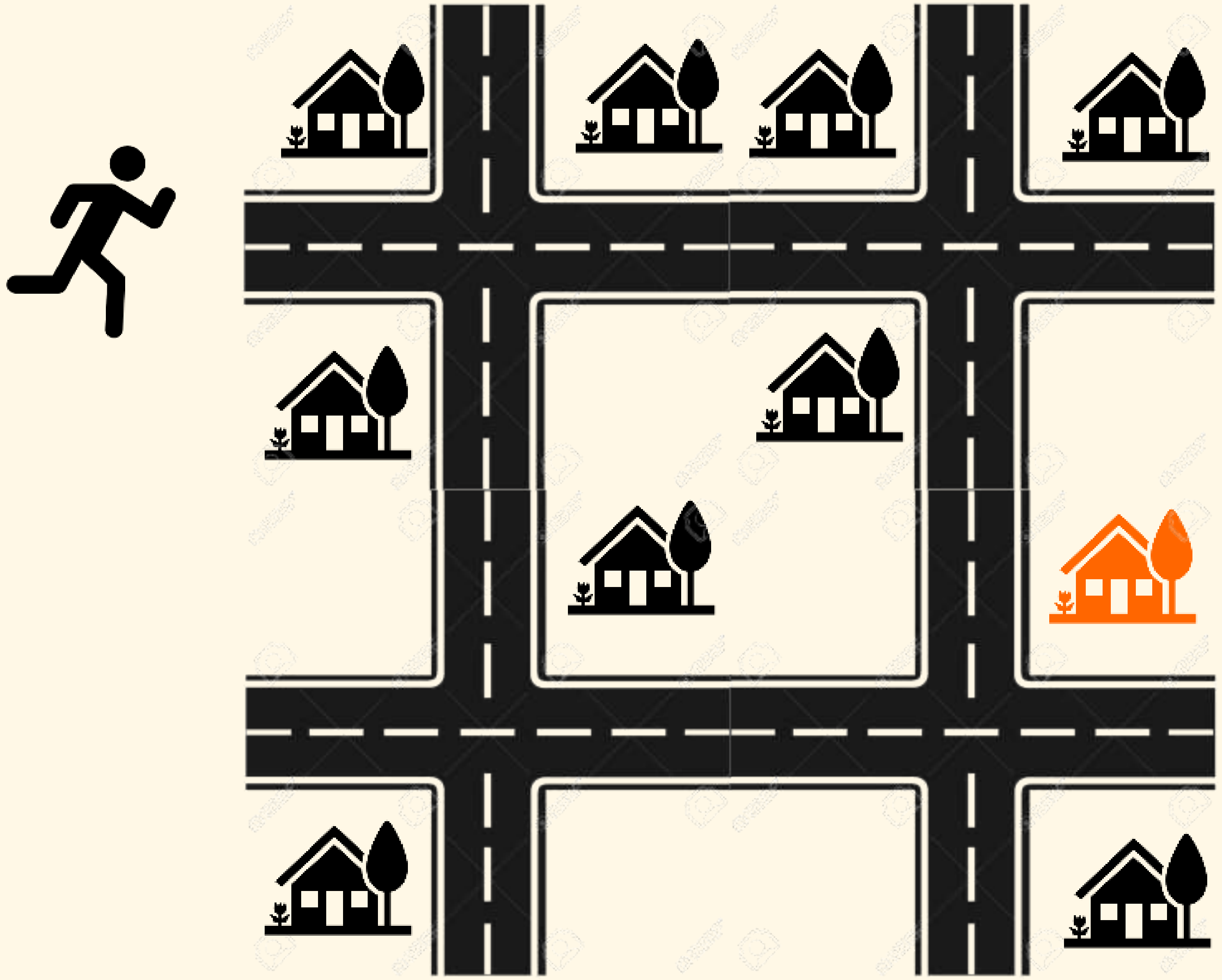
Example: Google Maps finding the fastest/cheapest route.

Characteristics:
- Expands the **lowest-cost node first** (not depth-based like BFS or DFS).
- Guarantees **optimal solution** if step cost > 0.
- Good for problems with **non-uniform costs**.

3. Uniform Cost Search (UCS)

- **UCS = expanding the cheapest path first.**
- Guarantees finding the **optimal (lowest cost) path** to the friend's house.
- Different from BFS (which minimizes number of steps) because UCS minimizes the **actual cost** of traveling.

Cost: Knocking house = 2, moving from 1 house to the next = 1



Uninformed search algorithms

3. Uniform Cost Search (UCS)

Vacuum World Example

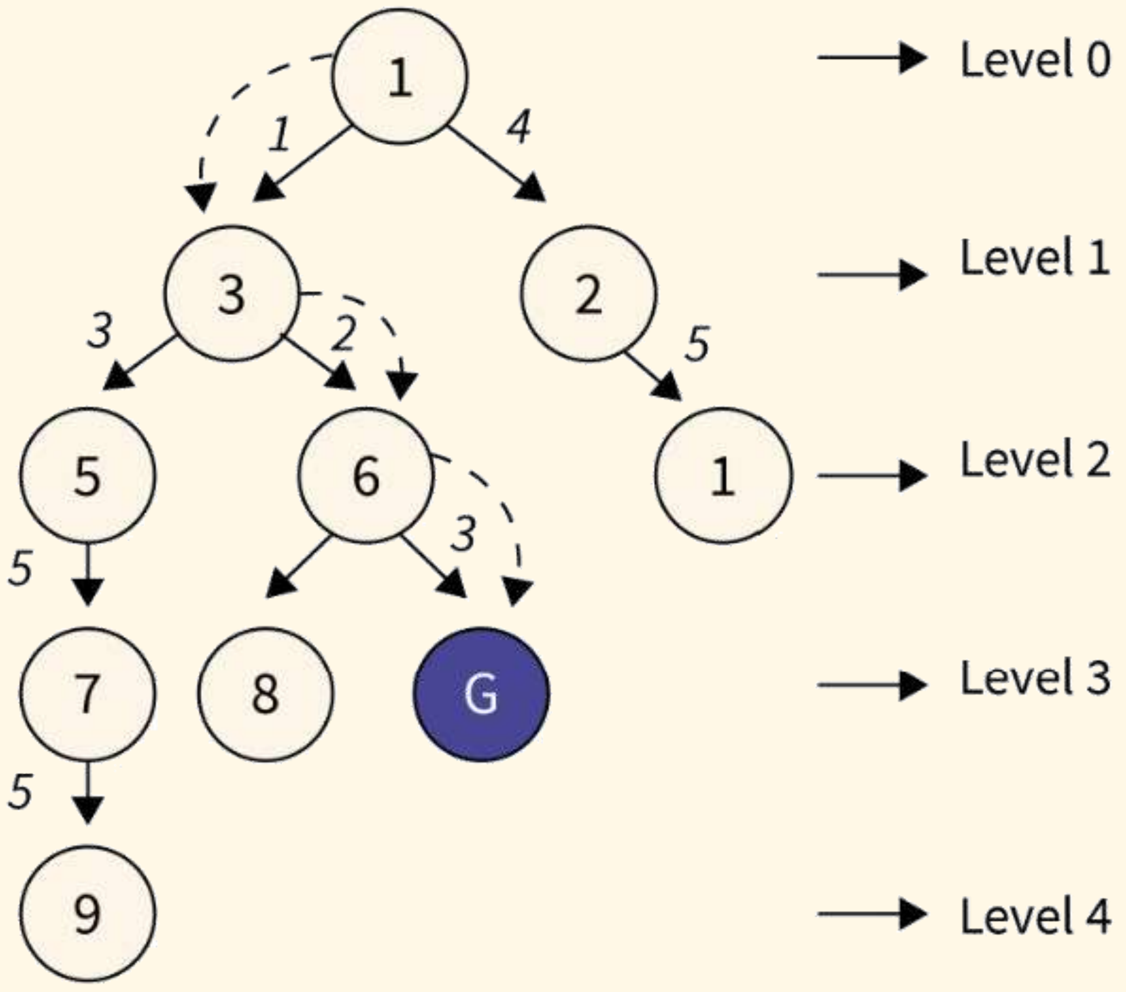
Suppose moving left/right = cost 1, cleaning = cost 2.

UCS will prioritize the sequence with **lowest cumulative cost** until reaching the clean goal state.

Initial State: [DirtyA, DirtyB, Vacuum at A]

Cost = 0

```
├── Clean A (Cost=2) → [CleanA, DirtyB, A]
│     └── Move Right (Cost=3) → [CleanA, DirtyB, B]
│           └── Clean B (Cost=5) → [CleanA, CleanB, B] ✅ Goal
│
└── Move Right (Cost=1) → [DirtyA, DirtyB, B]
      └── Clean B (Cost=3) → [DirtyA, CleanB, B]
            └── Move Left (Cost=4) → [DirtyA, CleanB, A]
                  └── Clean A (Cost=6) → [CleanA, CleanB, A] ✅ Goal
```



✓ UCS chooses the first path (total cost 5), which is **optimal**.

Uninformed search algorithms

4. Depth-Limited Search

- Like DFS but with a depth cut-off to prevent infinite loops.
- Useful when we know the approximate depth of a solution.

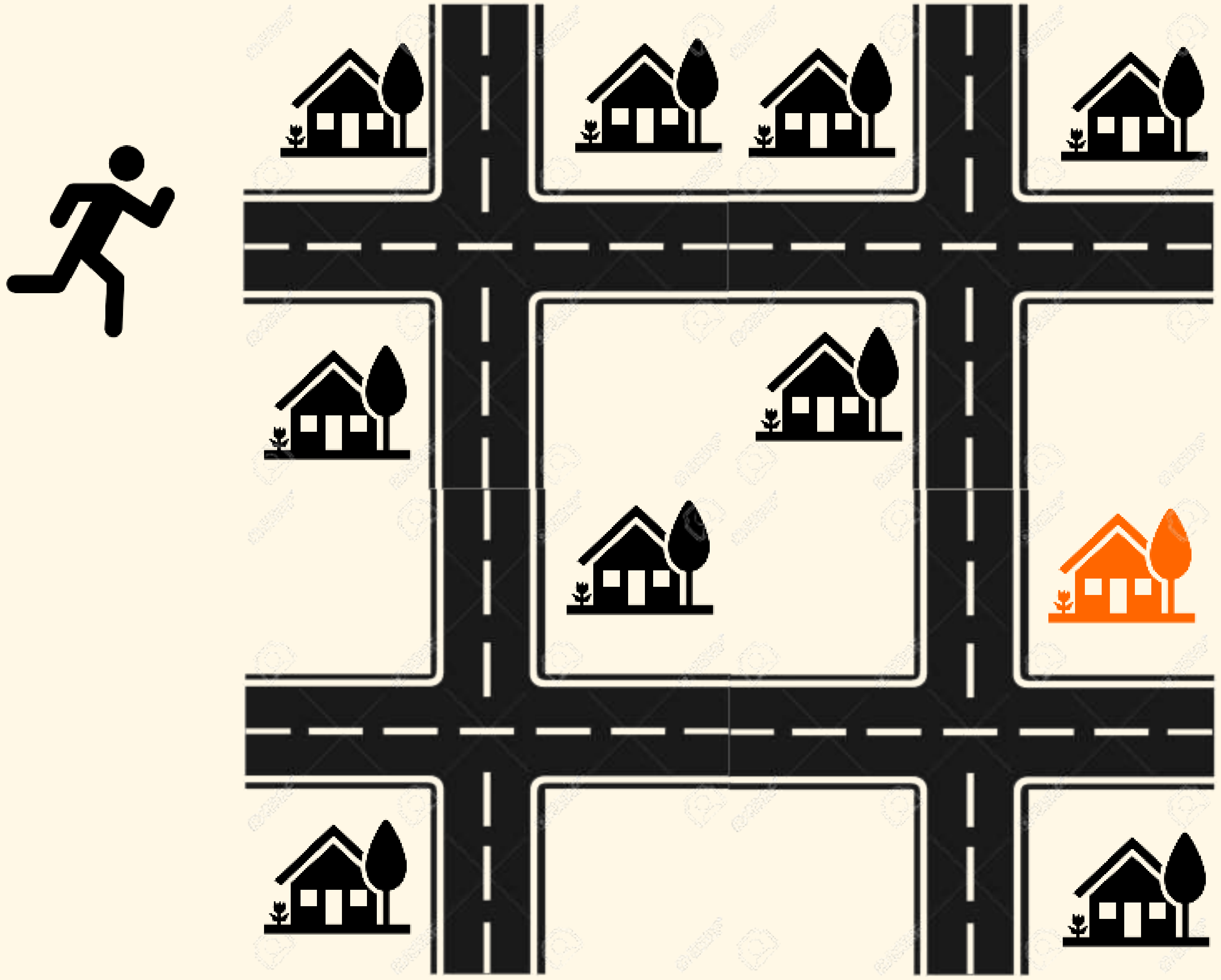
Characteristics:

- DFS but with a **cutoff depth L**.
- Avoids infinite loops.
- May miss solution if solution depth > L.

4. Depth-Limited Search

- DLS is like DFS with a cutoff point.
- Useful when the search tree is very deep or infinite (e.g., when you don't want to get lost forever).
- **Limitation**: If the goal is deeper than the limit, you will fail to find it.

Limit = 4



4. Depth-Limited Search

Vacuum World Example

Limit = 2.

Goal: both rooms clean.

Tree Example (Limit=2)

```
Depth 0: [DirtyA, DirtyB, A]
   |
Depth 1: ├── Clean A → [CleanA, DirtyB, A]
   |     └── Move Right → [DirtyA, DirtyB, B]
   |
Depth 2: ├── Move Right → [CleanA, DirtyB, B]
   |     └── Clean B → [DirtyA, CleanB, B]
```

✓ Since the **solution requires at least depth 3 (Clean A → Move Right → Clean B)**, it won't be found at L=2.

Uninformed search algorithms

5. Iterative Deepening Search (IDS)

- Combines BFS and DFS.
- Repeatedly applies DFS with increasing depth limits.
- Optimal like BFS, but memory efficient like DFS.

Characteristics:

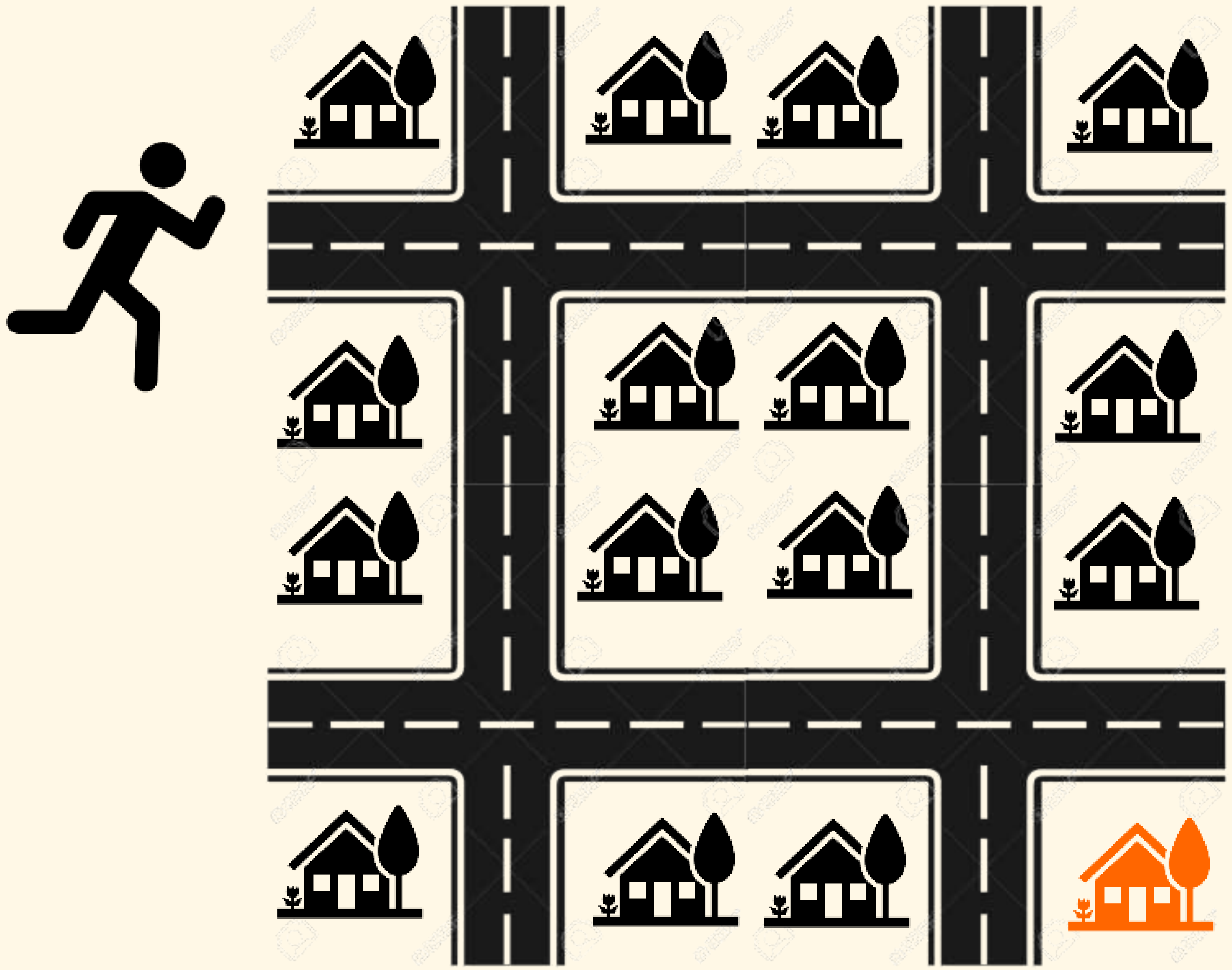
- Combines BFS completeness with DFS memory efficiency.
- Repeatedly runs DFS with increasing depth limit until solution is found.
- Finds **optimal solution in terms of depth**.

Uninformed search algorithms

5. Iterative Deepening Search (IDS)

- Like DFS, it explores deeply.
- Like BFS, it eventually guarantees finding the goal at the shallowest level.
- It avoids getting lost infinitely (DFS) and avoids memory explosion (BFS).
- The trade-off is that some paths are revisited multiple times, but overall it balances completeness and efficiency.

Depth = 4



5. Iterative Deepening Search (IDS)

Iteration L = 0 (Depth Limit = 0)

Only check root: [DirtyA, DirtyB, A]

Not goal → Stop

Iteration L = 1 (Depth Limit = 1)

Expand up to 1 step from root:

Action: Clean A → [CleanA, DirtyB, A]

Action: Move Right → [DirtyA, DirtyB, B]

Neither is the goal → Stop

Iteration L = 2 (Depth Limit = 2)

Expand paths up to 2 steps:

[DirtyA, DirtyB, A] → Clean A → [CleanA, DirtyB, A] → Move Right → [CleanA, DirtyB, B]

[DirtyA, DirtyB, A] → Move Right → [DirtyA, DirtyB, B] → Clean B → [DirtyA, CleanB, A]

Still no full goal (both clean) → Stop

```
(A_dirty, B_dirty, Agent=A)
           /              \
      SIPHON/                  \RIGHT
          /                         \
(A_clean, B_dirty, Agent=A)    (A_dirty, B_dirty, Agent=B)
       |                                |
     SIPHON                           SIPHON
       |                                |
     RIGHT                             LEFT
       |                                |
(A_clean, B_dirty, Agent=B)   (A_dirty, B_clean, Agent=A)
                      |
                   SIPHON
                      |
         (A_clean, B_clean) ✅ GOAL
```

5. Iterative Deepening Searc

Iteration L = 3 (Depth Limit = 3)

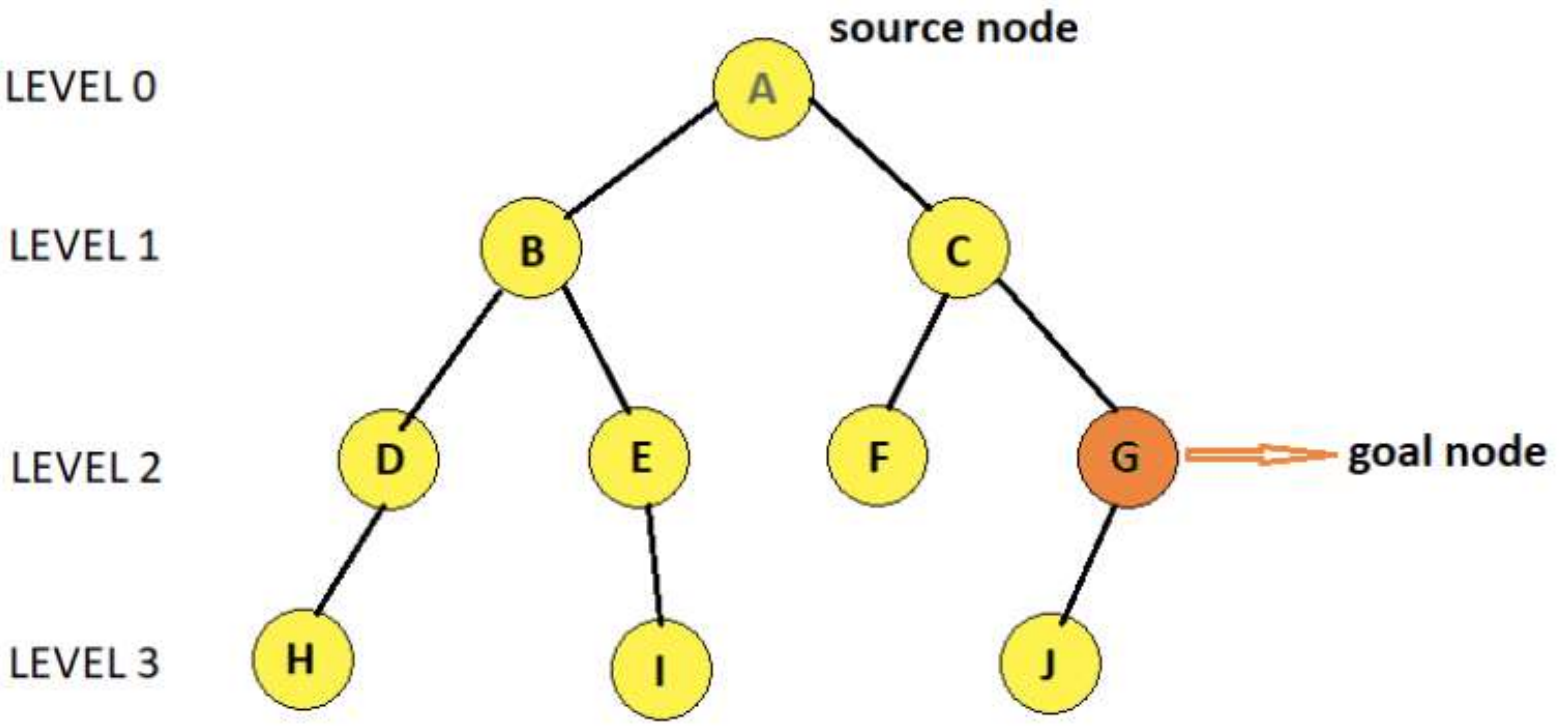
Expand further:

[DirtyA, DirtyB, A] → Clean A → [CleanA, DirtyB, A]
→ Move Right → [CleanA, DirtyB, B]
→ Clean B → [CleanA, CleanB, B] ✅ Goal Found
[DirtyA, DirtyB, A] → Move Right → [DirtyA, DirtyB, B]
→ Clean → [DirtyA, CleanB, B]
→ Move Left [DirtyA, CleanB, A]
Clean A → [CleanA, CleanB, A] ✅ Goal Found

```
(A_dirty, B_dirty, Agent=A)
          /           \
      SIPHON/                \RIGHT
        /                        \
(A_clean, B_dirty, Agent=A)    (A_dirty, B_dirty, Agent=B)
     |                               |
  SIPHON                           SIPHON
     |                               |
   RIGHT                            LEFT
     |                               |
(A_clean, B_dirty, Agent=B)   (A_dirty, B_clean, Agent=A)
                  |
               SIPHON
                  |
        (A_clean, B_clean) ✅ GOAL
```

- ✓ IDDFS eventually finds the solution at depth 3, same as BFS, but uses much less memory.

Iterative Deepening Search (IDS)



IDDFS with max depth-limit = 3
Note that iteration terminates at depth-limit=2
Iteration 0: A
Iteration 1: A->B->C
Iteration 2: A->B->D->E->C->F->G

Uninformed search algorithms

Search Strategy	Analogy (Friend’s House)	Key Behavior	Strength	Weakness
Breadth-First Search (BFS)	You check **all houses at distance 1**, then **all at distance 2**, and so on.	Explores level by level.	Guarantees **shortest path (fewest steps)**.	Memory heavy (stores many nodes).
Depth-First Search (DFS)	You pick one path and **walk straight until the end** before backtracking.	Explores deep before wide.	Low memory use.	May go too deep (get lost in wrong path).
Uniform Cost Search (UCS)	You check **the cheapest travel cost first**, not just the shortest path. Example: A longer road with fewer tolls may be cheaper.	Expands paths by **lowest cumulative cost**.	Guarantees **lowest-cost solution**.	Slower if costs are very close.
Depth-Limited Search (DLS)	You say, “I’ll only check up to **3 streets away**. If I don’t find the house, I stop.”	DFS with a fixed depth limit.	Prevents infinite loops.	Might miss the goal if limit is too shallow.
Iterative Deepening Search (IDS)	You search **1 street away**, then **2 streets**, then **3 streets**… repeating until you find the house.	Repeated DFS with increasing limits.	Finds shortest path, less memory than BFS.	Some repeated work (restarts search at each depth).

Uses **heuristics** (problem-specific knowledge) to guide search, making it more efficient.

informed (Heuristic) search algorithms

1. Greedy Best-First Search

- Expands the node that looks closest to the goal based on a heuristic **h(n)**.
- Fast but not guaranteed optimal.

Example: GPS navigation using straight-line distance as heuristic.

2. A*

- Combines UCS and Greedy Best-First.
- Expands node with the lowest **f(n) = g(n) + h(n)**
  - g(n) = cost so far
  - h(n) = heuristic estimate to goal
- Optimal and complete (if heuristic is admissible).

Example: Pathfinding in games, GPS navigation.

Greedy Best-First Search

Vacuum World Setup

- **States:** Position of the vacuum (A or B), and cleanliness of each room.
- **Actions:** Move Left, Move Right, Siphon.
- **Goal:** Both rooms clean.
- **Path Cost:** 1 per action.

Greedy Best-First Search (GBFS)

- **Strategy:** Chooses the node with the **lowest heuristic (h(n))**, ignoring the path cost.
- **Heuristic Example:**
  h(n)=number of dirty rooms
- Initial State: Vacuum at A, A = dirty, B = dirty
  - h(n)=2 (two dirty rooms)
- If vacuum siphons at A: Vacuum at A, A = clean, B = dir
  - h(n)=1
- Move right to B and siphon: Both clean
  - h(n)=0 → Goal

GBFS Path: Siphon(A) → Move Right → Siphon(B)

- Only cares about cleaning the dirtiest state quickly, not path cost.

```
Start (A dirty, B dirty) h=2
├── Siphon(A) → (A clean, B dirty, at A) h=1
│    └── Move Right → (A clean, B dirty, at B) h=1
│    └── Siphon(B) → (A clean, B clean) h=0
└── Move Right → (A dirty, B dirty, at B) h=2
     └── Siphon(B) → (A dirty, B clean) h=1
     └── Move Left → (A dirty, B clean, at A) h=1
     └── Siphon(A) → (A clean, B clean) h=0
```

A* Search

A* Search

•**Strategy:** Minimizes f(n)=g(n)+h(n), where:

- f(n) is the predicted total path cost from start → current node → goal
- g(n) = cost so far
- h(n) = estimated cost to goal (dirty room/s)

Example:

•Initial State: Vacuum at A, A = dirty, B = dirty

•g(n)=0, h(n) =2 → f(n)=2

•After **Siphon(A):** A = clean, B = dirty

•g(n)=1, h(n)=1 → f(n)=2

•After **Move Right:** Vacuum at B, A = clean, B = dirty

•g(n)=2, h(n)=1 → f(n)=3

•After **Siphon(B):** Both clean

•g(n)=3, h(n)=0 → f(n)=3

A* Path: Siphon(A) → Move Right → Siphon(B)

•Same as GBFS here, but A* **balances path cost + heuristic**, so if actions had different costs (e.g., moving costlier than siphoning), A* would choose the cheapest solution overall.

```
Start (A dirty, B dirty) f=2 (g=0,h=2)
 ├── Siphon(A) → (A clean, B dirty) f=2 (g=1,h=1)
 │      └── Move Right → (A clean, B dirty, at B) f=3 (g=2,h=1)
 │           └── Siphon(B) → Goal f=3 (g=3,h=0)
 └── OR Move Right → (A dirty, B dirty, at B) f=3 (g=1,h=2)
           └── Siphon(B) →(A dirty, B clean) f=3 (g=2,h=1)
            └── Move Left→(A dirty, B clean) f=4 (g=3,h=1)
            └── Siphon(A) → Goal f=4 (g=4,h=0)
```

In short:

- **GBFS** is fast but not always optimal (greedy).
- **A*** guarantees optimality if heuristic is admissible (never overestimates).

Basic Search Algorithms in AI

Algorithm	Behavior in Vacuum World
Greedy Best-First Search	Always rushes to the dirtiest room it sees first. May waste moves (e.g., moves to Room B, cleans, then goes back to A unnecessarily).
A*	Plans carefully by considering both moves already made and estimated moves left. Always finds the shortest sequence of moves to get both rooms clean.

Optimizing Cleaning in the Vacuum World

Post-Exercise: Uninformed vs. Informed Search (2-Room Vacuum World)

Scenario Setup

- Environment: 2 **rooms in a row** → A – B
- Initial state:
  - Agent starts at **Room A**
  - Room A = **dirty**
  - Room B = **clean**
- Goal: All rooms are clean.
- Actions: **Clean, Move Left, Move Right** (cost = 1 per action).
- Requirements: Apply all Uninformed and Informed Search Algorithms (show all expansions, including non-optimal path).

Thank You!

Any Questions?