Module 2

Problem-Solving Agents

- A problem-solving agent decides what to do by searching for a sequence of actions that lead to a goal.
- It works well in known, fully observable environments where actions and outcomes are predictable.
- The agent formulates a problem, searches for a solution, and executes the plan.
- This process is called the "think → search → act" cycle.
- Example: A route-finding agent planning a trip from city A to city B.

1. Well-Defined Problems and Solutions

- A problem is well-defined if it includes all the following:
  1. **Initial state –** where the agent starts.
  2. **Actions –** what moves are allowed.
  3. **Transition model –** what happens when an action is taken.
  4. **Goal test –** how to check if the goal has been reached.
  5. **Path cost –** numerical value to evaluate how good a solution is (e.g., distance, time).
- A solution is a sequence of actions from the initial state to a goal state.
- The optimal solution has the lowest path cost among all possible solutions.

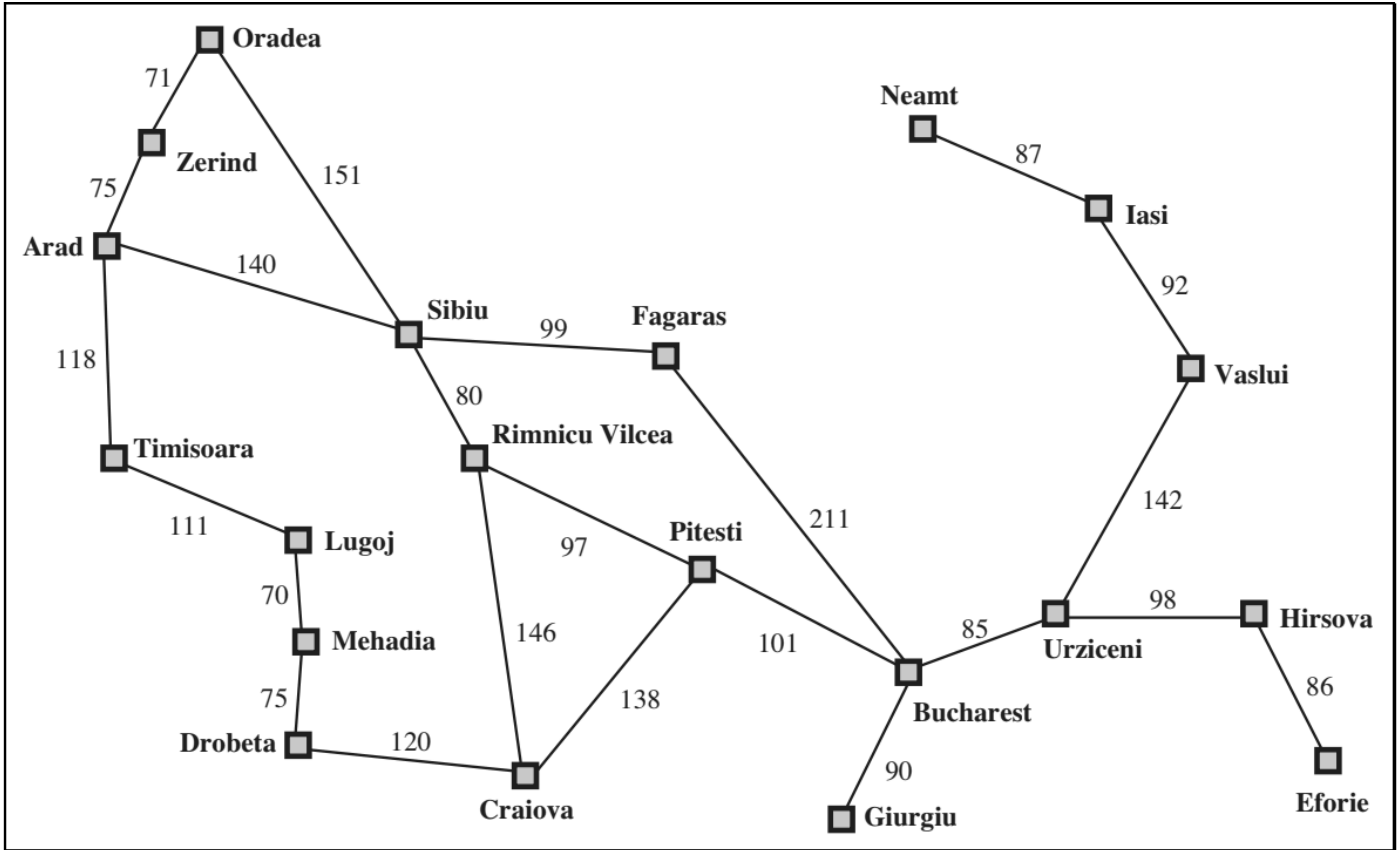


Figure 3.2 A simplified road map of part of Romania.

2. Formulating Problems

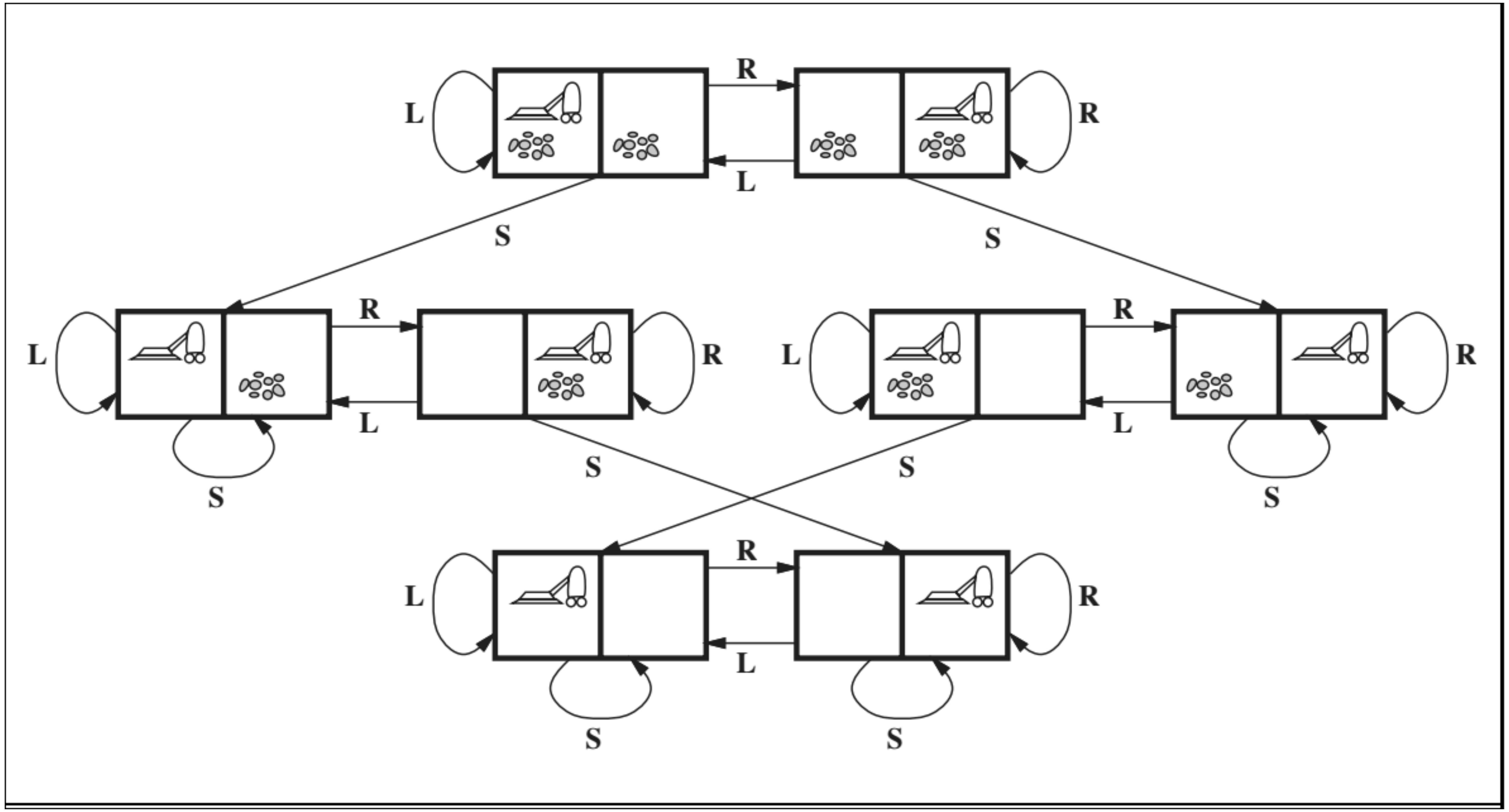
- Problem formulation is the process of defining the search problem in a way the agent can solve.
- Good problem formulation helps the agent focus on relevant details and ignore unnecessary complexity.
- It requires:
    - Clearly specifying the initial state, available actions, and goal.
    - Understanding the rules of how actions affect the world (transition model).
- Example: In the vacuum world, the initial state is room A dirty, the action is “suck,” and the goal is both rooms clean.

Example Problems

Toy Problem: Vacuum World

- **States**:
  - The state depends on the agent’s location (A or B) and whether each square has dirt.
  - Total possible states = $2 \times 2^2 = 8$ (2 locations × 2 dirt conditions).
  - For n locations, the total states = $n \times 2^n$.
- **Initial State**:
  - Any of the 8 states can be the starting point of the agent.
  - Example: Agent at A, A is dirty, B is clean.
- **Actions**:
  - The agent can perform Left, Right, or Suck.
  - In larger environments, more actions like Up and Down may be included.
- **Transition Model**:
  - Actions do what you'd expect: Left/Right moves the agent, Suck cleans dirt.
  - Exceptions:Moving Left from leftmost square = no effect.
    - Moving Right from rightmost square = no effect.
    - Suck in a clean square = no effect.

- **Goal Test**:
  - The goal is reached when both squares are clean.
- **Path Cost**:
  - Each action costs 1 step.
  - Total cost = number of steps taken to reach the goal.



8-Puzzle Problem

States:

- Each state is a 3×3 grid with 8 numbered tiles and 1 empty space.
- The position of each tile and the blank determines the state.
- There are 9! = 362,880 possible configurations, but only half are solvable.

Initial State:

- Any random arrangement of tiles and the blank can be the initial state.

Actions:

- The blank tile can move Up, Down, Left, or Right, if the move is legal (i.e., not off the edge).
- Each move swaps the blank with a neighboring tile.

Transition Model:

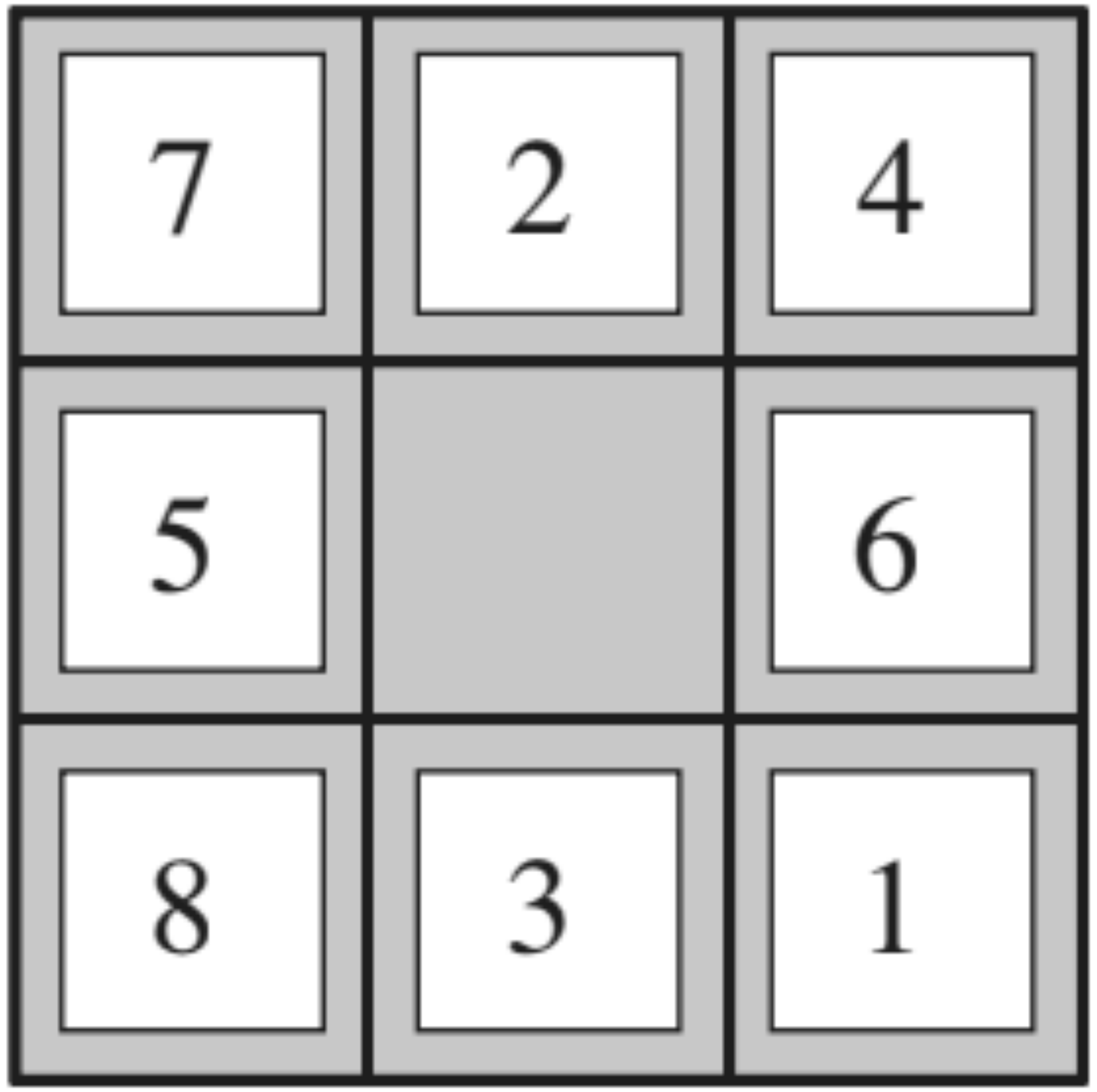
- The model defines the resulting state after a legal move (i.e., how tiles are rearranged).
- Example: Moving blank up swaps it with the tile above.

Goal Test:

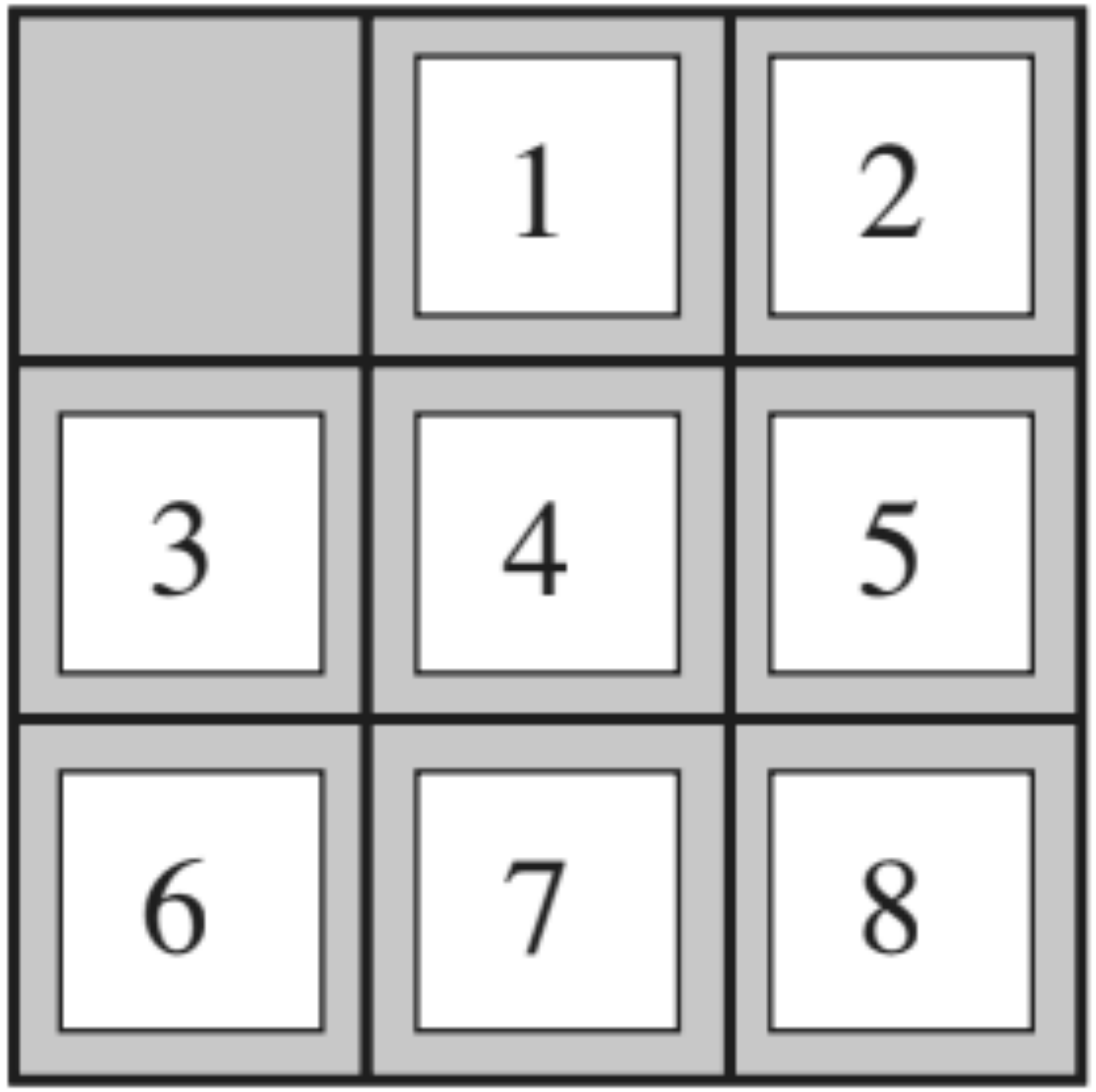
- The agent checks whether the puzzle is in the goal configuration:

Path Cost:

- Each move costs 1 step, so the total cost = number of moves taken to reach the goal.
- Often used to find the shortest (optimal) solution.



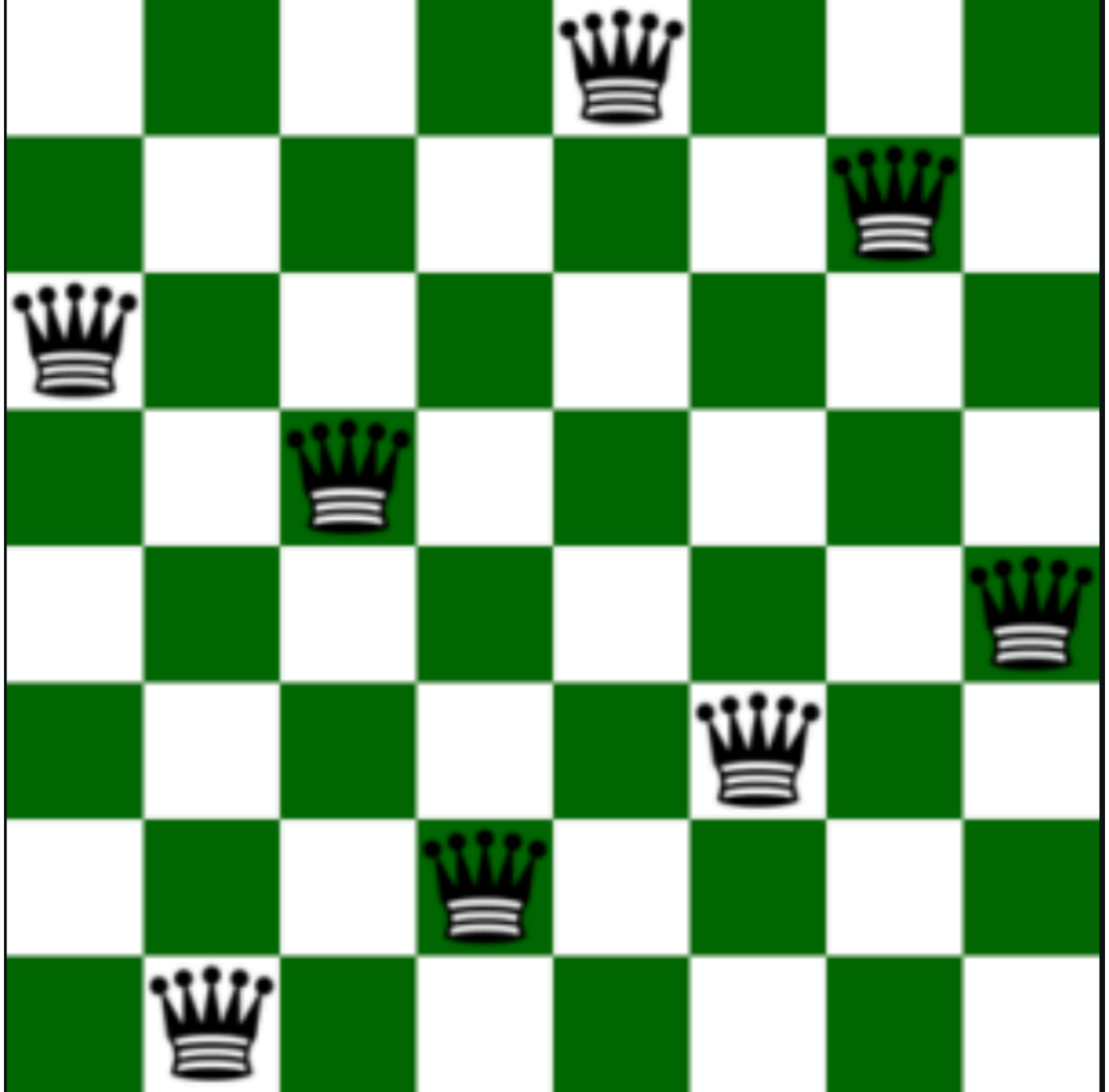
Start State



Goal State

8-Queens Problem

- **States**:
  - Each state is a configuration of queens on a chessboard.
  - A valid state places 1 queen in each row (or column), trying to avoid attacks.
  - Total possible configurations: 64 choose 8, but many are invalid.
- **Initial State**:
  - The board starts with 0 to 8 queens placed arbitrarily (often row by row).
  - Can also start with a blank board and place queens one at a time.
- **Actions**:
  - Add a queen to a row in a non-attacking column.
  - In local search versions: move a queen to a different square in the same column.
- **Transition Model**:
  - Describes the result of placing or moving a queen.
  - In local search (e.g., hill climbing), it defines a neighboring configuration.
- **Goal Test**:
  - Exactly 8 queens on the board, with no pair attacking each other (i.e., no two queens share the same row, column, or diagonal).
- **Path Cost**:
  - Often irrelevant for this problem, as any valid solution is acceptable (not a path-finding problem).
  - In some formulations, path cost = number of moves or violations fixed.



Real-World Problems

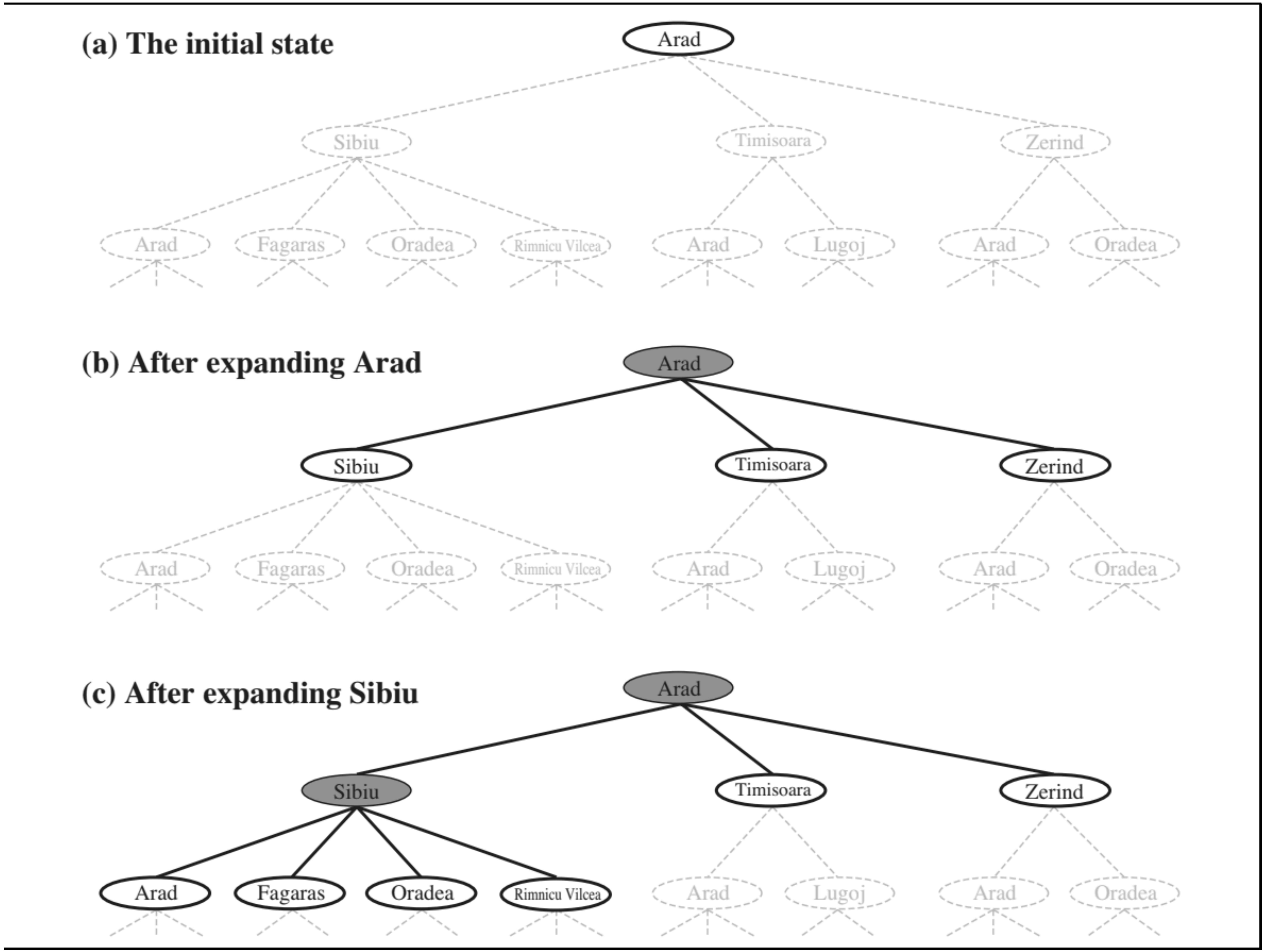
- **States:**
  - Represent complex, real-world situations (e.g., map location, robot position, car speed).
  - Often involve many variables and continuous values (not just grid positions or tiles).
  - Examples: GPS coordinates, fuel level, weather condition, traffic state.
- **Initial State:**
  - Starting condition in the real-world environment.
  - Example: A robot starting at a charging station or a delivery truck starting from a warehouse.
- **Actions:**
  - Realistic and domain-specific actions like drive forward, turn, pick up object, make a payment, etc.
  - Some actions might have uncertain outcomes (e.g., slippery road may affect turning).
- **Transition Model:**
  - Defines how the world changes after an action is performed.
  - Can be deterministic or stochastic (involving uncertainty).
  - Example: Moving a car changes its position and possibly fuel level.
- **Goal Test:**
  - Determines if the desired real-world goal is achieved (e.g., package delivered, destination reached).
  - May involve multiple conditions (e.g., deliver all packages + return home).
- **Path Cost:**
  - Often based on real-world resources: time, fuel, money, distance, energy.
  - Optimal solutions minimize the total cost.

Examples of Real-World Problems

- Route-finding (GPS navigation)
- Autonomous driving
- Robot delivery systems
- Flight planning
- Logistics and supply chain scheduling
- Travelling Salesman Problem
- VLSI layout
- Automatic Assembly Sequencing
- Protein Design

Searching for Solutions

1. After formulating a problem, the agent searches for a sequence of actions that leads from the initial state to a goal state.
2. The result of the search is called a solution, which is a complete plan or path the agent can follow.
3. If no path leads to the goal, the agent returns failure, meaning the problem has no solution under the given conditions.
4. Search involves systematically exploring possible states using available actions.
5. Different search strategies determine which paths to explore first (e.g., depth-first, breadth-first, etc.).


(a) The initial state
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea
(b) After expanding Arad
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

Infrastructure for Search Algorithms

- A search tree is like a map of all the possible moves the agent can make.
- Each node represents a state (where the agent is at a point in time).
- Nodes are connected via actions; a child node is generated from its parent node through an action.
- The frontier is the list of next possible states to explore.
- The explored set keeps track of states already visited to avoid going in circles.
- The search algorithm decides which node to explore next based on the method used.

Measuring Problem-Solving Performance

When comparing search algorithms

1. **Completeness** – Will it always find a solution if there is one?
2. **Time** – How fast is it? How many steps or nodes does it take?
3. **Space** – How much memory does it need to run?
4. **Optimality** – Will it give the best or shortest solution?

Uninformed Search Strategies

- These algorithms search without any knowledge of how close a state is to the goal.
- Also called blind search — they only use the information in the problem definition (initial state, actions, goal test, path cost).
- They explore the search space systematically, but do not use heuristics.

1. **Breadth-First Search (BFS)**

- Explores all nodes at one level before going deeper.
- Uses a queue (FIFO) to expand nodes in order of depth.
- Always finds the shallowest (fewest steps) solution.
- Complete, Optimal (if all step costs are equal).
- Can be slow and memory-heavy.

2. **Uniform-Cost Search (UCS)**

- Expands the node with the lowest path cost first.
- Uses a priority queue sorted by path cost.
- Always finds the cheapest solution, even if it takes more steps.
- Complete, Optimal (for any cost structure).
- Can be slow, especially with small cost differences.

3. Depth-First Search (DFS)

- Explores as deep as possible before backtracking.
- Uses a stack (LIFO) structure.
- Memory-efficient, Not guaranteed to find shortest or cheapest solution.
- Can get stuck in infinite paths if not careful (in infinite trees).
- Fast if solution is deep and leftmost.

4. Depth-Limited Search

- Like DFS but with a depth limit to prevent infinite descent.
- Useful in infinite or very deep search spaces.
- Not always complete if the solution is below the depth limit.
- Not optimal, but more controlled than DFS.

5. Iterative Deepening Depth-First Search (IDDFS)

- Combines depth-limited search and BFS.
- Repeats DFS with increasing depth limits.
- Complete, Optimal (like BFS), Uses less memory.
- Best for large search trees with unknown depth.
- May repeat nodes, but overall efficient.

6. Bidirectional Search

- Searches forward from start and backward from goal simultaneously.
- Stops when the two searches meet in the middle.
- Can reduce time from O(b^d) to O(b^(d/2)).
- Very fast if reverse actions are easy to compute.
- Needs to store both frontiers; not always applicable.

Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^{a}	Yes^{a,b}	No	No	Yes^{a}	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^{c}	Yes	No	No	Yes^{c}	Yes^{c,d}