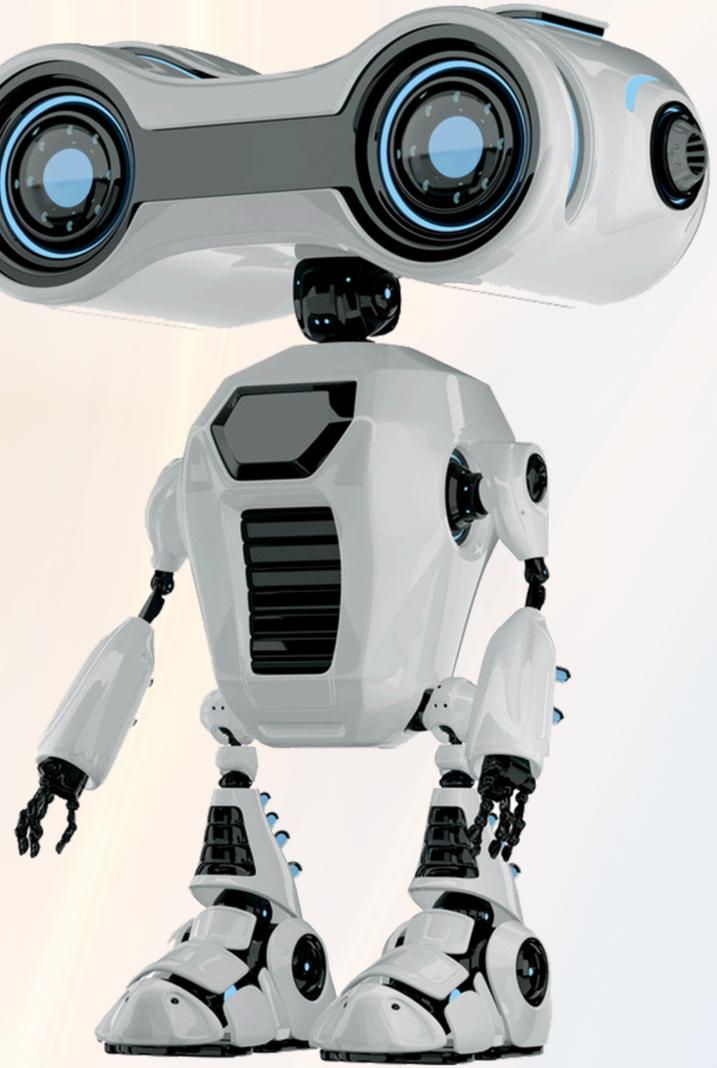
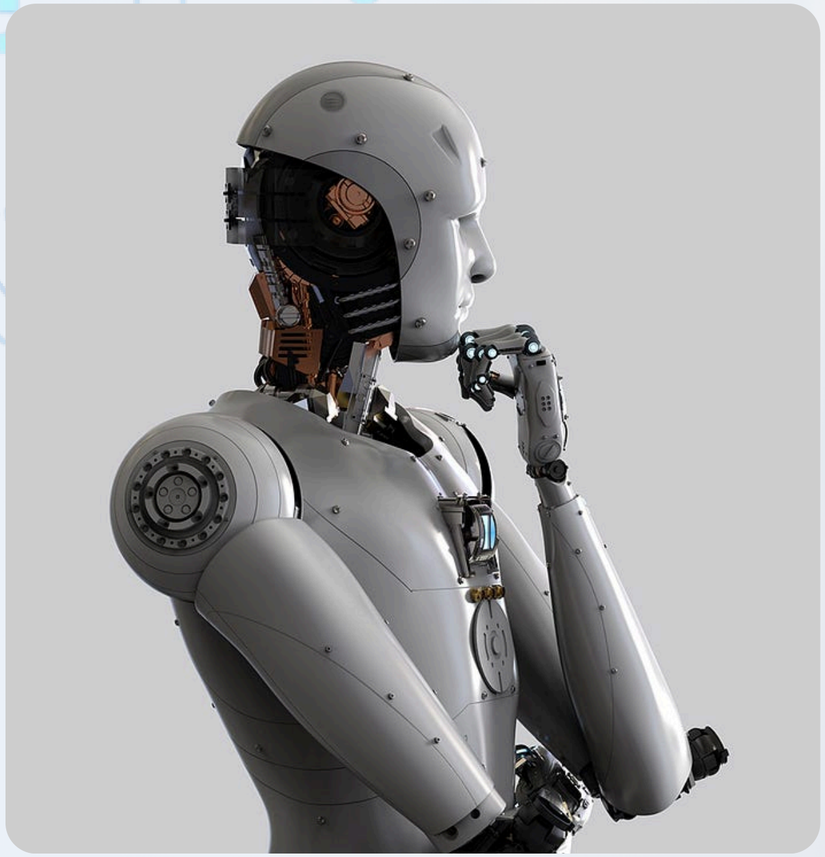
Artificial Intelligence: Solving problems by Searching



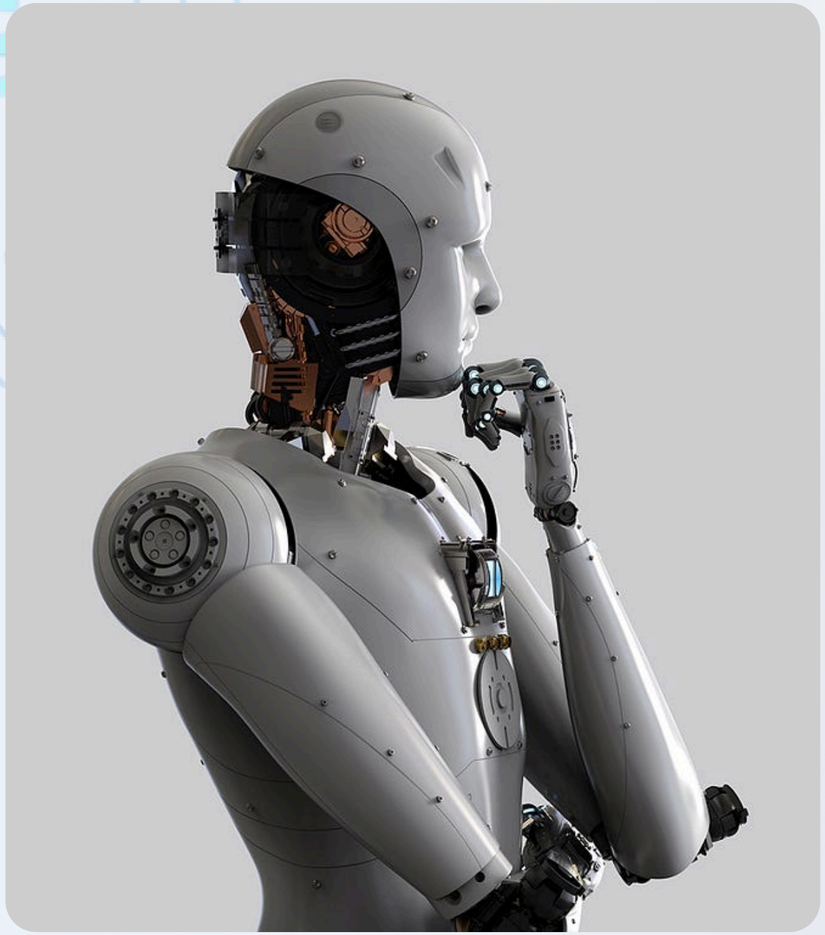
Problem-Solving Agents: Basic Terminology



🎯 **Goals** – a desired state. Goals organize an agent's behavior by limiting the objectives that an agent tries to achieve and hence the actions it needs to consider. **Goal formulation**

💭 **Problem formulation –** the process of deciding what actions and states to consider, given a goal

Problem-Solving Agents: Basic Terminology



🔎 **Search** - looking for a sequence of actions that reaches the goal. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence.

▶ **Execution -** carrying out actions recommended in the solution step-wise

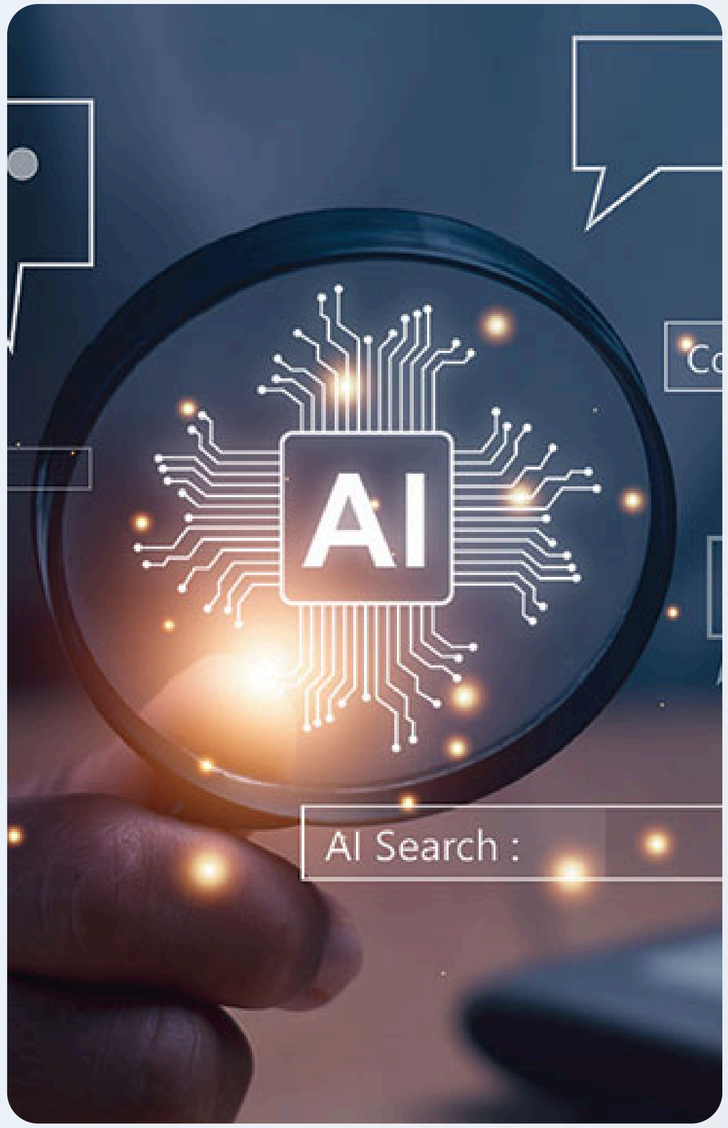
➡**Open-loop system -** an agent ignores its percepts while executing the actions, breaking the loop between itself and the environment


FORMULATE


SEARCH


Execute
Shift

Components of a Problem



01 **Initial State** an agent starts in. **Set space -** a set of possible states reachable from the initial state.

02 **Possible actions** available to the agent.

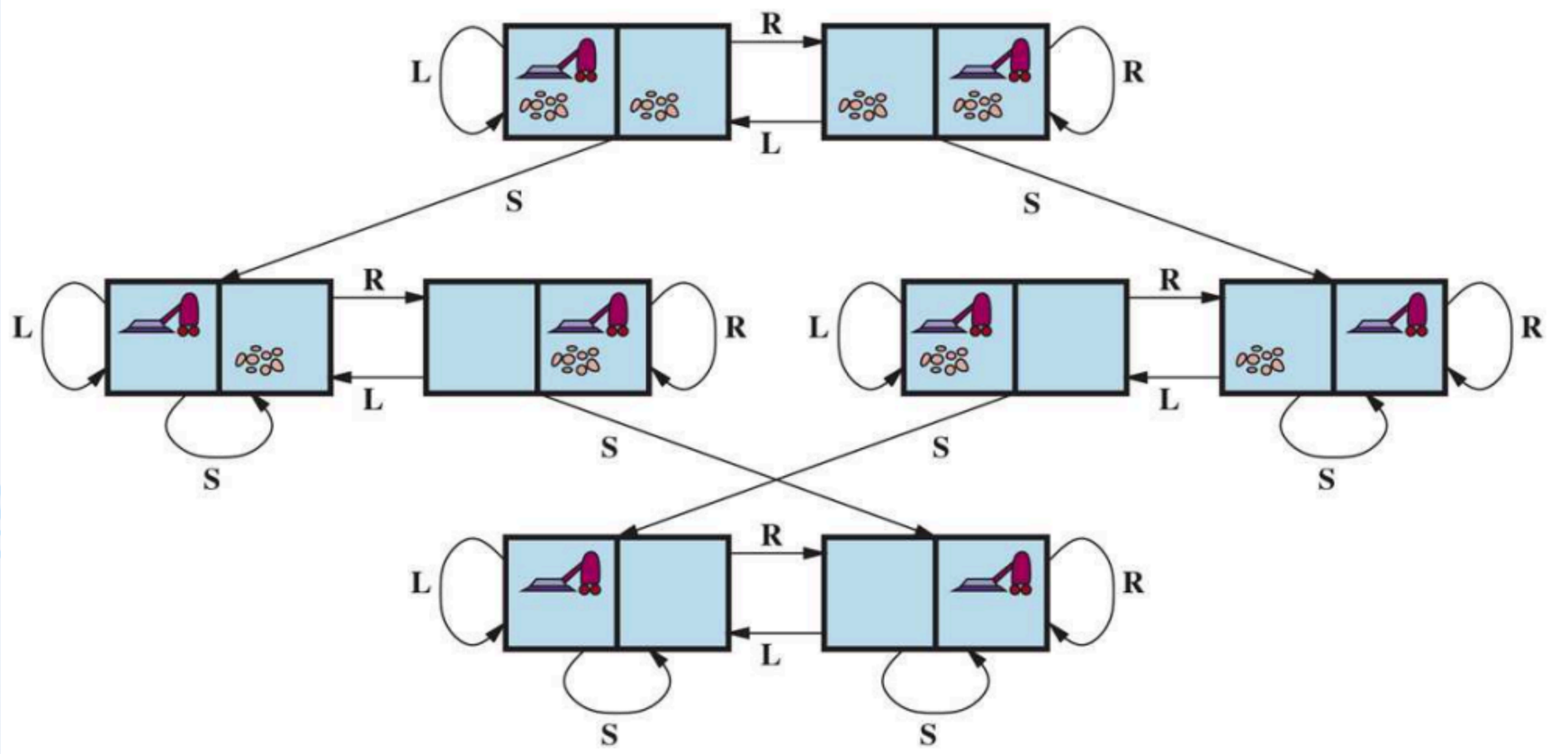
03 **Transition model -** description of what each action does. **Successor -** any state reachable from a given state by a single action.

04 A set of **goal states** and a **goal test** to determine whether a given state is a goal state.

05 **Path/action cost function -** assigns a numeric cost to each path

Solution - an action sequence that leads from the initial state to a goal state.
Optimal Solution - has lowest path cost.

Toy Problems



States - 8, any being **Initial State**

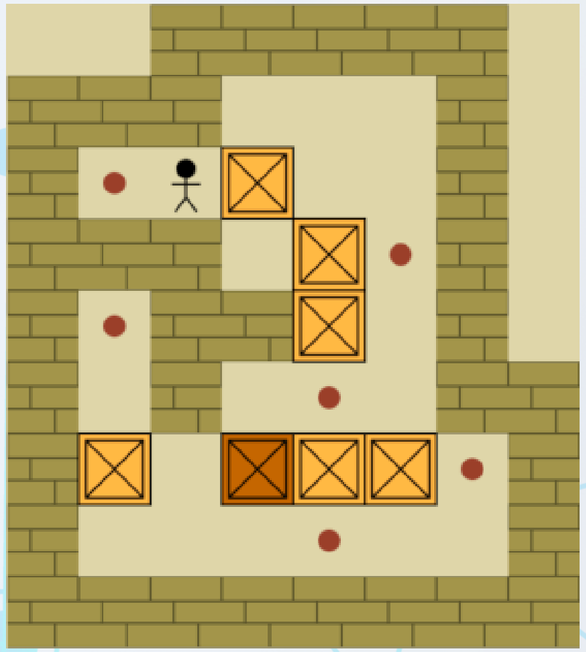
Actions - Left, Right, Suck

Goal test - all squares are clean

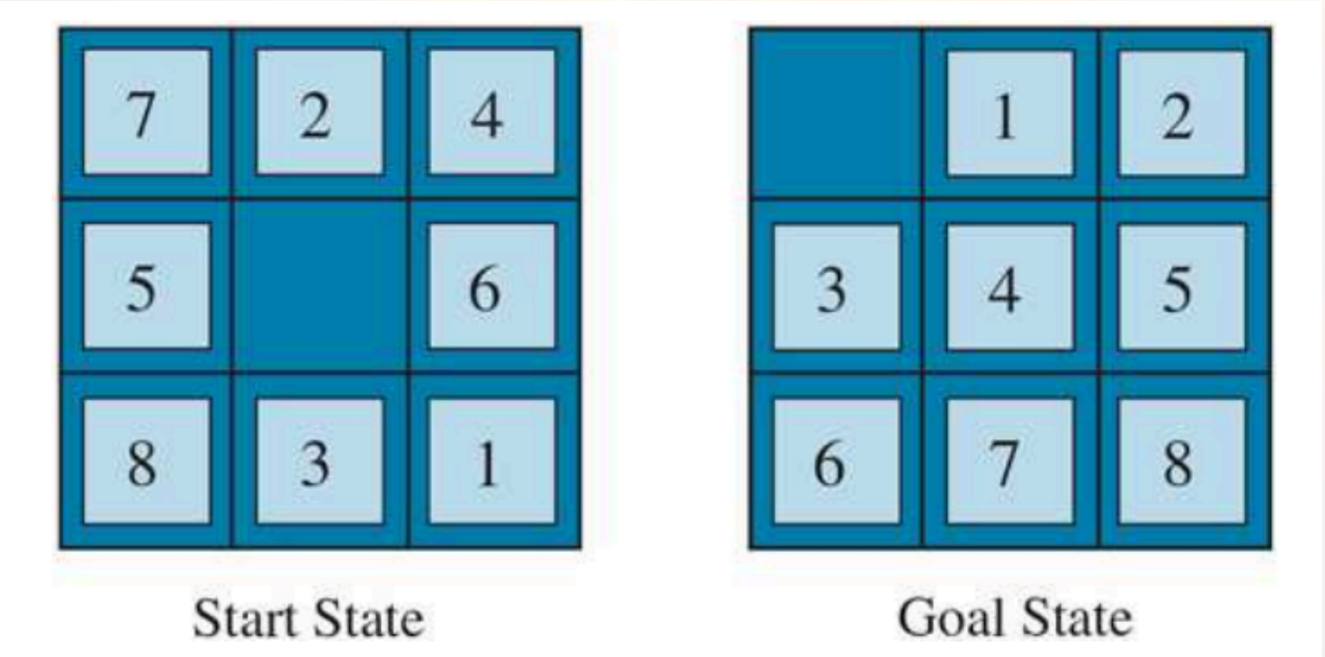
Transition model - each action has effects, except *l* in L, *r* in R and *s* in a clean square

Path function - each step costs 1

Toy Problems

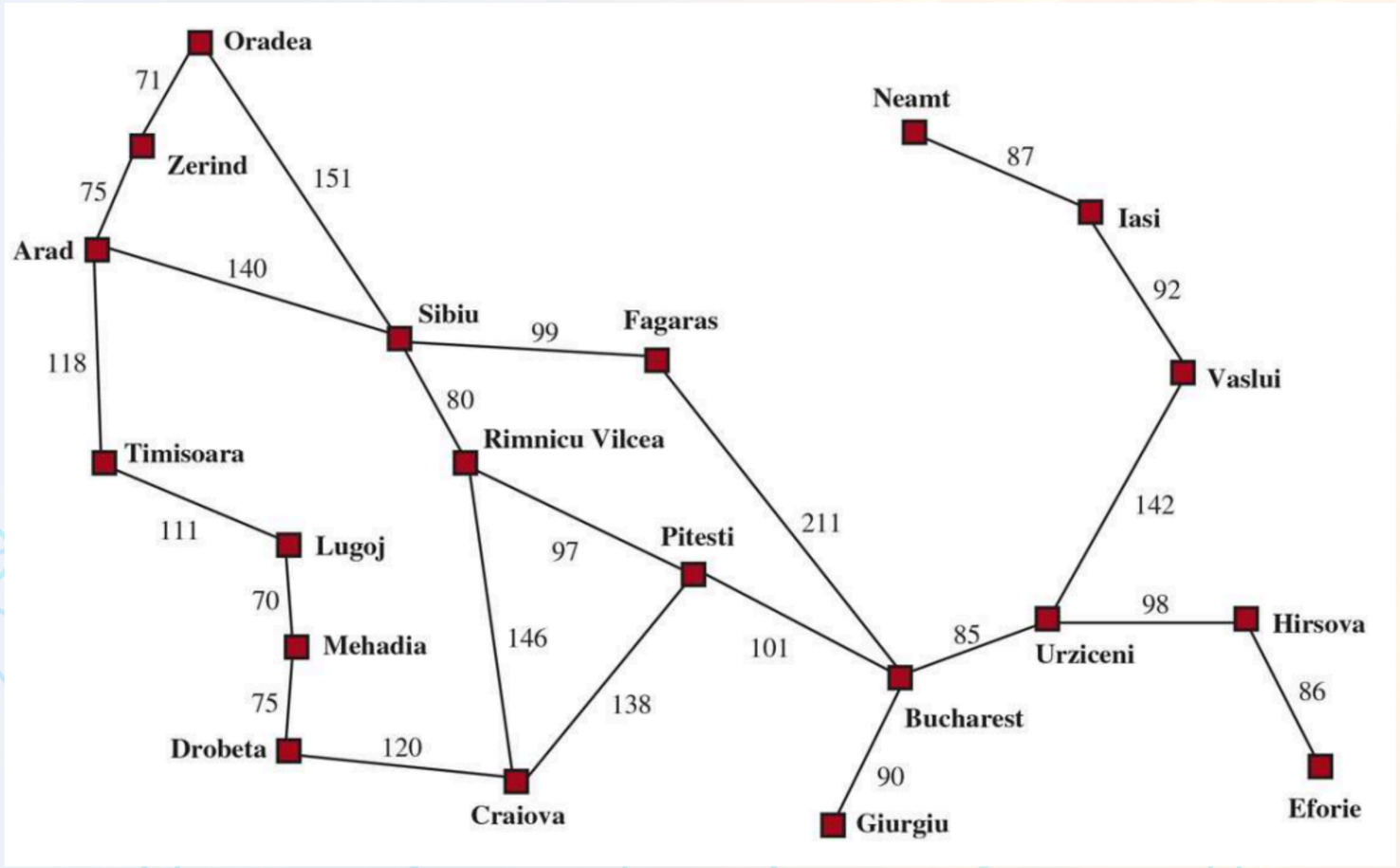


Sokoban Puzzle



8*8 Puzzle

Real World Problem



States - 20 cities, any being **Initial State**

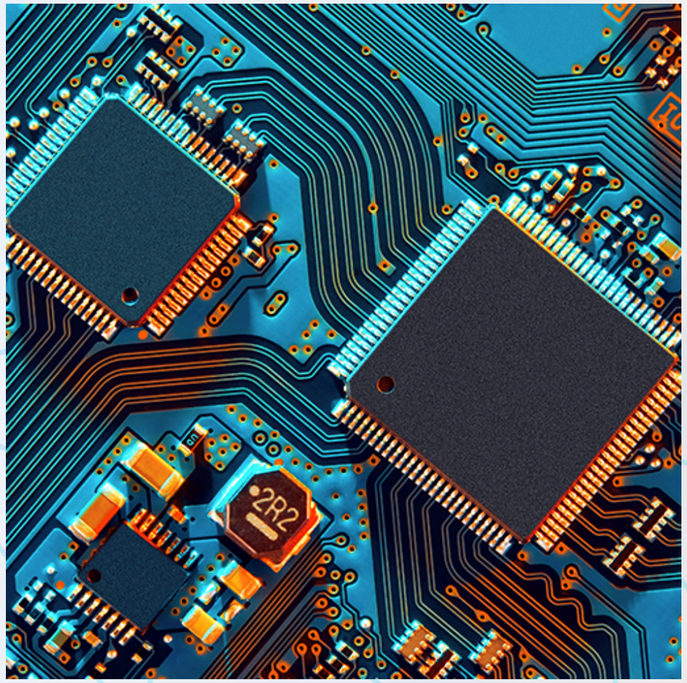
Actions - move to the next city

Goal test - at destination

Transition model - change of current city

Path function - defined in each path

Real World Problems



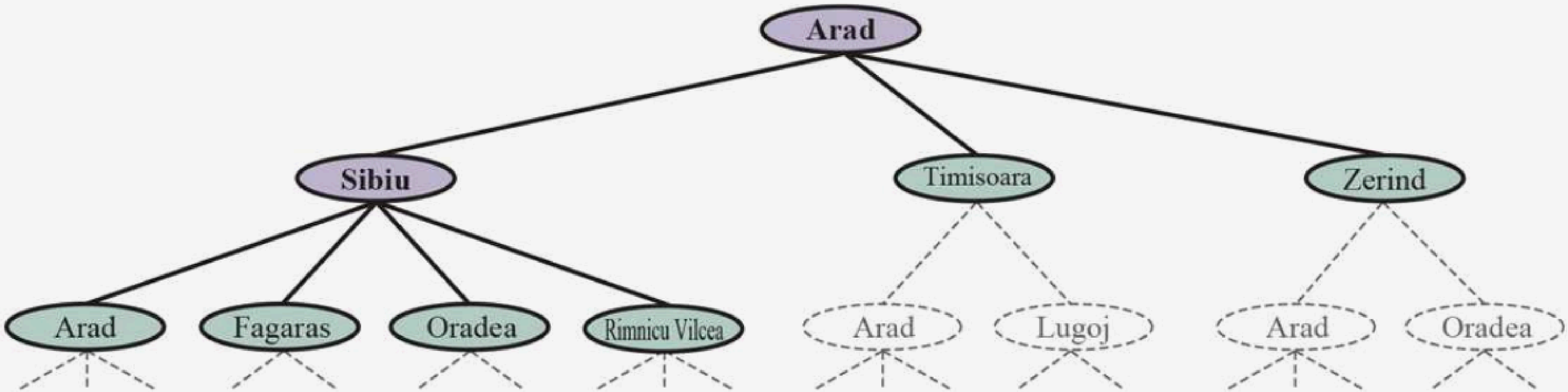
VLSI design

Automatic Assembly Sequencing

Search Algorithms

- Solution: Sequence of actions .
- Initial State: Starting point of possible action sequences.
- Search Tree: Possible action sequence from the root
- Nodes: State spaces of the problem.
- Edges: Represent actions.

Moving Along the Tree



Terms Defn

- Expanding: Considering available actions for a state.
- Generating: Creating child nodes based on available actions.
- Parent Node: The node before the current node.
- Leaf Node: A node with no children.
- Frontier: Set of all leaf nodes available for expansion.

Redundant Paths

Redundant paths: When a node can be reached in more than one way.

To handle redundant paths:

- Maintain an explored set to track expanded nodes.
- Not worrying about repetitions (Tree Search).
- Detect cycles by checking parent chains (Graph Search).

Search Data Structure

Node Representation:

- node.STATE: The state of the node.
- node.PARENT: The parent node that generated it.
- node.ACTION: Action applied to reach this node.
- node.PATH-COST: Total cost from the initial state.

Frontier Data Structure

Mainly uses “queue” data structure.

Frontier Operations:

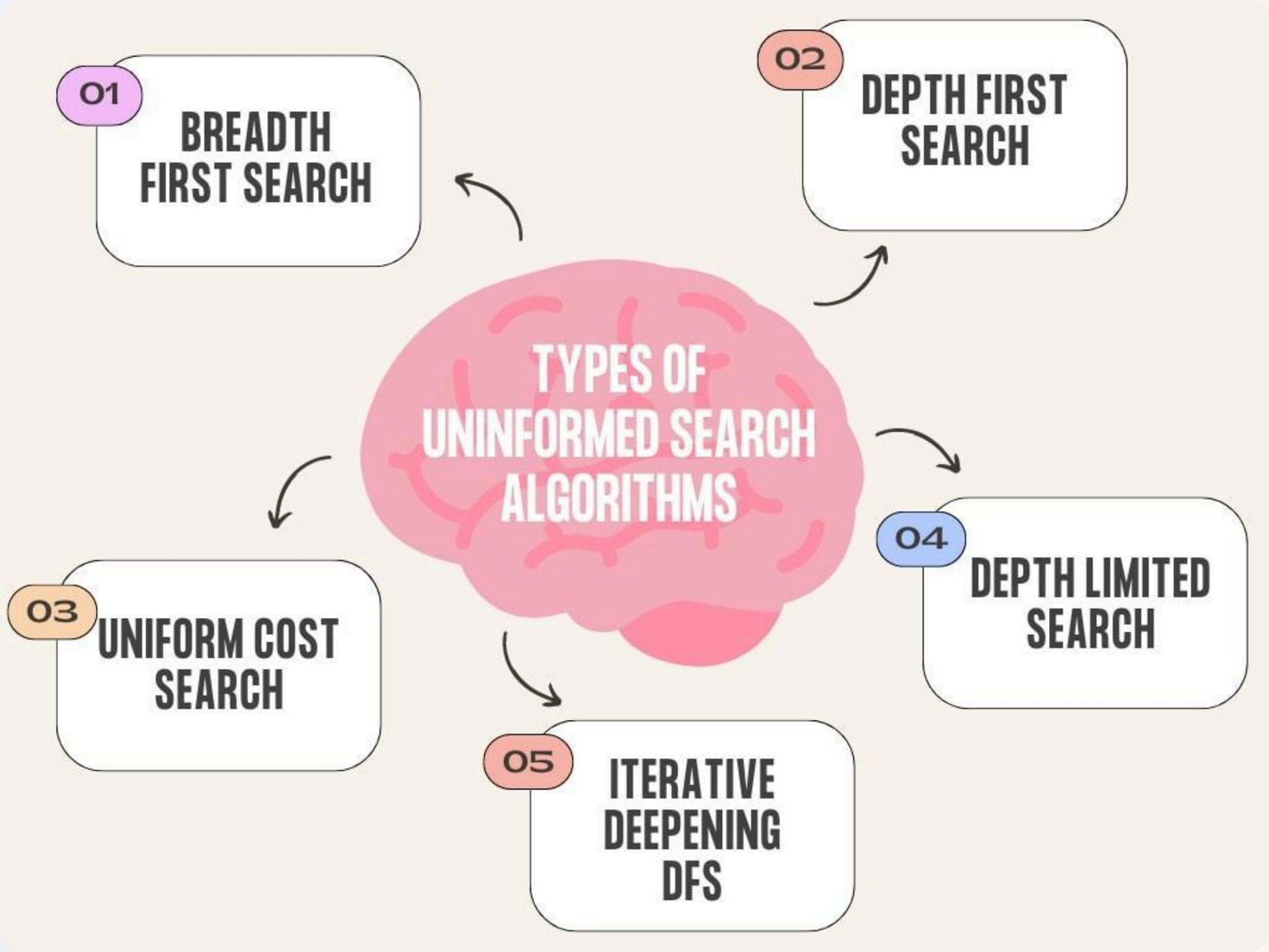
- IS-EMPTY: Checks if the frontier is empty.
- POP: Removes and returns the top node.
- TOP: Returns the top node without removing it.
- ADD: Inserts node in the queue properly.

Evaluating Problem-solving Perfomance

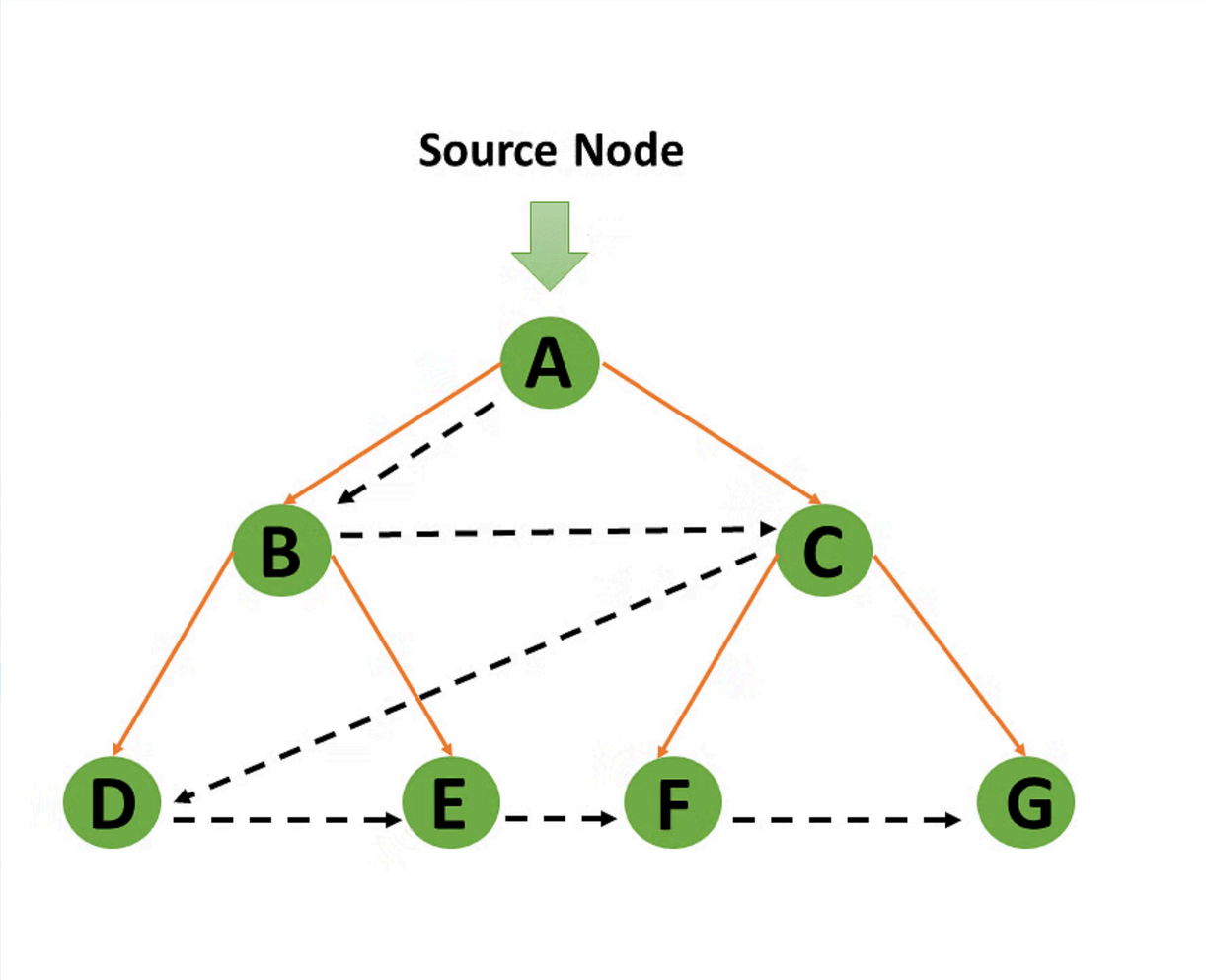
UNINFORMED SEARCH STRATEGIES

- Uninformed search strategies are algorithms that explore the search space without any prior knowledge of how close a given state is to the goal.
- They do not use extra information that could guide the search more efficiently but rather explore the state space systematically.

STRATEGIES


TYPES OF UNINFORMED SEARCH ALGORITHMS
01
BREADTH FIRST SEARCH
02
DEPTH FIRST SEARCH
03
UNIFORM COST SEARCH
04
DEPTH LIMITED SEARCH
05
ITERATIVE DEEPENING DFS

Breadth-First Search (BFS)



- BFS is a type of uninformed search that explores all nodes at the current depth before moving deeper into the search space, expanding the root node first, then all its children.

- It systematically explores all possible states of a problem level by level, ensuring that the shortest path in terms of the number of steps is found when all actions have the same cost.

How BFS Works

- Start with the root (initial node) and add it to a FIFO queue.
- Repeat until the queue is empty:
- Remove (dequeue) the front node from the queue.
- Process the node (check if it’s the goal).
- Mark the node as visited so that we don’t visit it again.
- Add all unvisited neighbours (child nodes) to the back of the queue.
- Since nodes are processed in order of insertion, all nodes at depth d are expanded before any nodes at depth d+1.
- An early goal test checks whether a node is a goal as soon as it is generated, preventing unnecessary expansions.

Properties of BFS

Completeness

- BFS is complete, meaning it will always find a solution if one exists.
- Even in infinite state spaces, BFS guarantees finding a solution if the branching factor is finite.

Optimality (Cost-Optimality)

- BFS guarantees to find the shortest path (least number of actions) when all actions have the same cost.
- If action costs vary, BFS may not be optimal; Uniform-Cost Search (UCS) is preferable in that case.

Properties of BFS

Time Complexity

- In a problem with a branching factor of b, where each state has b successors, the total number of nodes at depth d is: 1+b+b2+b3+...+bd=O(b^d)
- BFS requires expanding all nodes up to depth d, leading to exponential time complexity, which grows rapidly for large d.

Space Complexity

- BFS stores all generated nodes in memory, leading to an exponential space requirement of O(b^d).
- Memory becomes a limiting factor, often making BFS infeasible for large-scale problems.

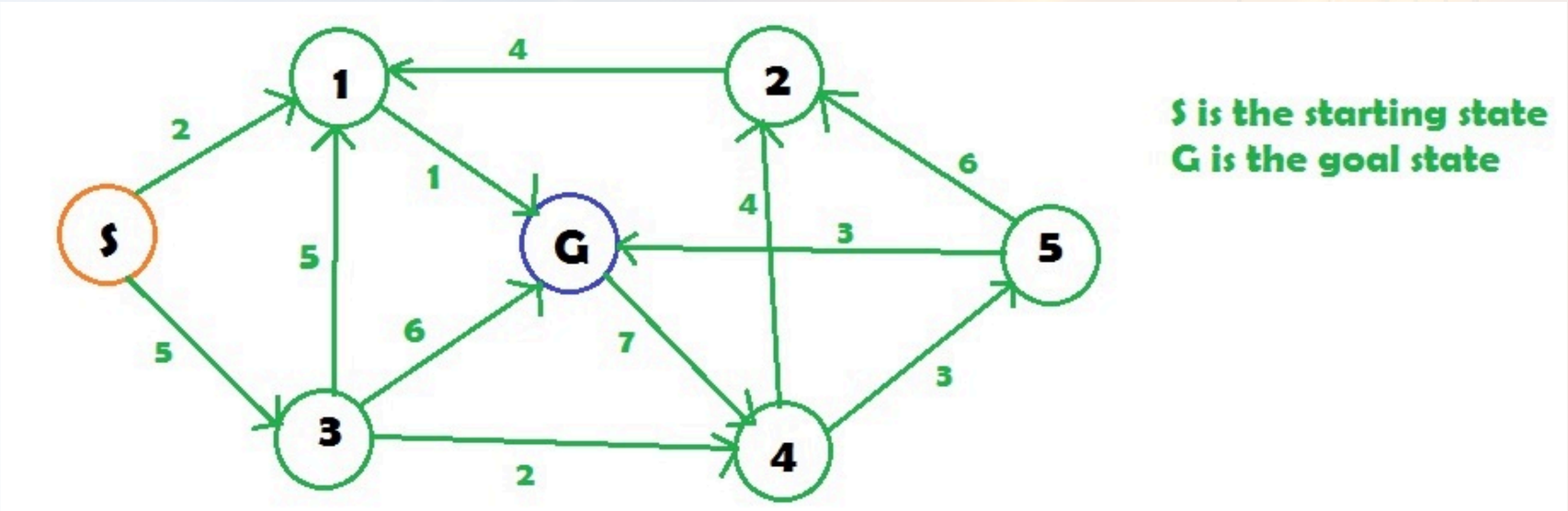
BFS Algorithm (Pseudocode)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure:
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node

    frontier ← FIFO queue with node as an element
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do:
      node ← POP(frontier)
      for each child in EXPAND(problem, node) do:
        s ← child.STATE
        if problem.IS-GOAL(s) then return child
        if s is not in reached then:
            add s to reached
            add child to frontier
    return failure
```

Uniform-Cost Searching

- **Uniform-cost search** is an uninformed search strategy that always expands the node with the **lowest total path cost** from the start node, rather than expanding nodes by their depth (as in Breadth-First Search).
- It is sometimes called **Dijkstra's Algorithm** when applied to graphs without negative edge costs.



Key Concepts

1. **Path Cost; *g(n)***
   - Each node ***n*** in the search has a path cost, ***g(n)***, representing how much it costs to get from the start node to ***n***.
   - Uniform-cost search uses a priority queue (often called **the frontier**) ordered by ***g(n)*** in **ascending order**. Nodes with lower ***g(n)*** are expanded first.

Key Concepts (cont'd)

2. Expansion Order

- In each step, the algorithm removes (expands) the node with the **lowest path cost** from the frontier.
- When a node is expanded, its **successors** are generated, and their path costs are calculated.
- Any new successor node goes into the priority queue, ordered by its **total cost** so far.

Key Concepts (cont'd)

3. Goal Test on Expansion

- The **goal test** is applied when a node is **about to be expanded**, not when it is first generated.
- This differs from **Breadth-First Search**, where we might check the goal right when a node is generated.
- In Uniform-cost search, this ensures we do not mistakenly stop on a **suboptimal path**.

Key Concepts (cont'd)

4. Updating Paths

- If the algorithm finds a **better path** (lower cost) to a node already on the frontier, it **updates** that node's path cost.
- This prevents discarding a **potentially better route** discovered after the node was first generated.

Key Concepts (cont'd)

5. Completion

- Uniform-cost search is **complete** if every action costs at least some **small positive constant** (ε/epsilon).
- If there are **zero-cost actions**, the algorithm can **get stuck** exploring an **infinite sequence** of those actions.

How It Works: Simple Algorithm Illustration

1. Initial State:

- You begin with a **start node** (e.g, a city if you're navigating a map).
- The **frontier** is the priority queue containing just this start node with cost; ***g(start) = 0***.

Algorithm Illustration (cont'd)

2. Expansion Steps:

- Select the node ***n*** from the frontier with the smallest ***g(n)***.
- If ***n*** is a goal node, return the **solution**.
- Otherwise, expand ***n*** by generating all its **successors** (neighboring nodes).
- For each successor, calculate its path cost ***g(successor)=g(n)+step_cost***.
- **Insert** or **update** this successor in the priority queue, ensuring the queue remains **ordered by lowest path cost** first.

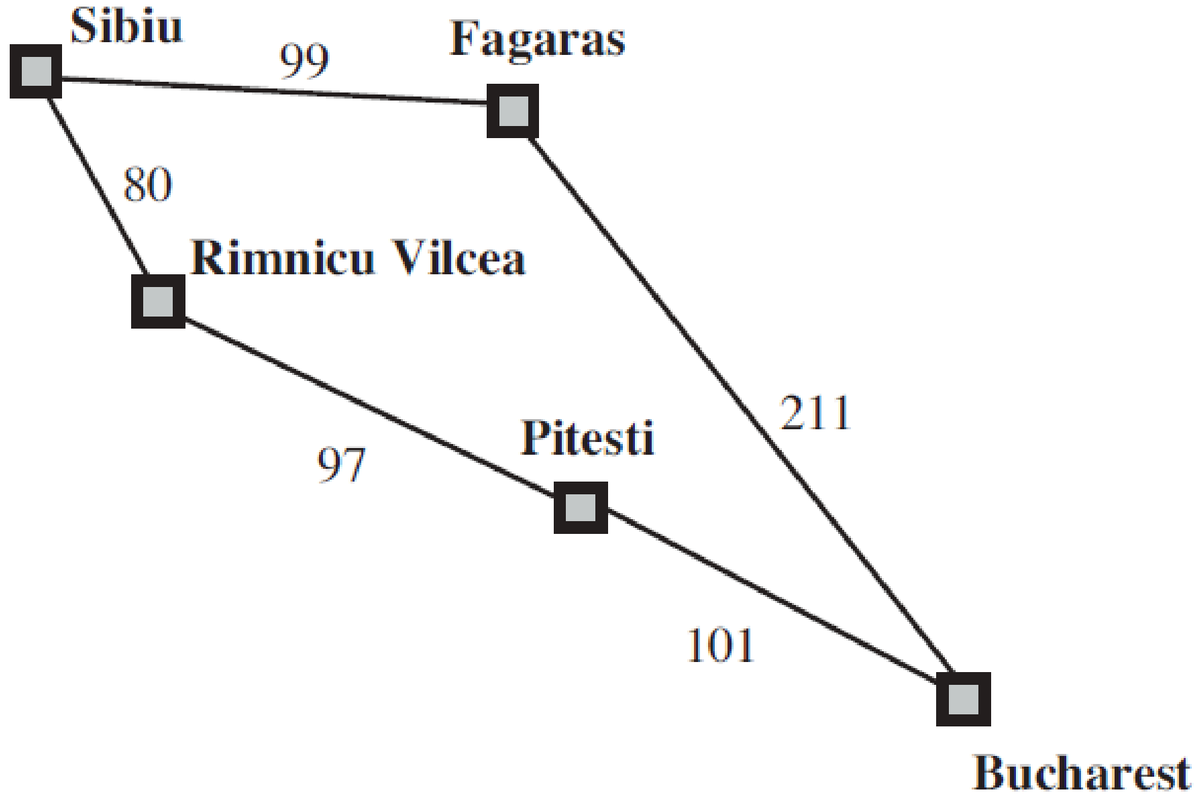
Algorithm Illustration (cont'd)

3. Termination:

- Eventually, the **lowest-cost path** to the goal will emerge from the priority queue and be expanded.
- Once you expand the goal, you know it's the **optimal solution**.

Example Scenario

- Consider traveling in **Romania** from **Sibiu** to **Bucharest**, where each road has a certain distance (cost):



Part of the Romania state space, selected to illustrate uniform-cost search.

Steps

1. Start at **Sibiu *(g=0)***. Frontier = **{Sibiu(0)}**
2. Expand **Sibiu**:
    - Successors: **Rimnicu Vilcea** (cost 80) and **Fagaras** (cost 99).
    - Frontier = **{Rimnicu(80), Fagaras(99)}**
3. Next, expand the lowest-cost node; **Rimnicu(80)**:
    - Successor: **Pitesti** with total cost; 80 + 97 = 177.
    - Frontier = **{Fagaras(99), Pitesti(177)}**

Steps (cont'd)

4. Next, expand the lowest-cost node: **Fagaras(99)**:

- Successor: **Bucharest** with total cost; 99+211=310.
- Frontier = **{Pitesti(177), Bucharest(310)}**
  - We have a path to Bucharest, but we keep going because we might find **a cheaper path**.

5. Next, expand **Pitesti(177)**:

- Successor: **Bucharest** with total cost; 177+101=278.
  - This new path to Bucharest is **cheaper** (278 < 310), so we **update** the cost of Bucharest in the frontier.
- Frontier = **{Bucharest(278)}** (the 310 cost path is discarded)

Steps (conclusion)

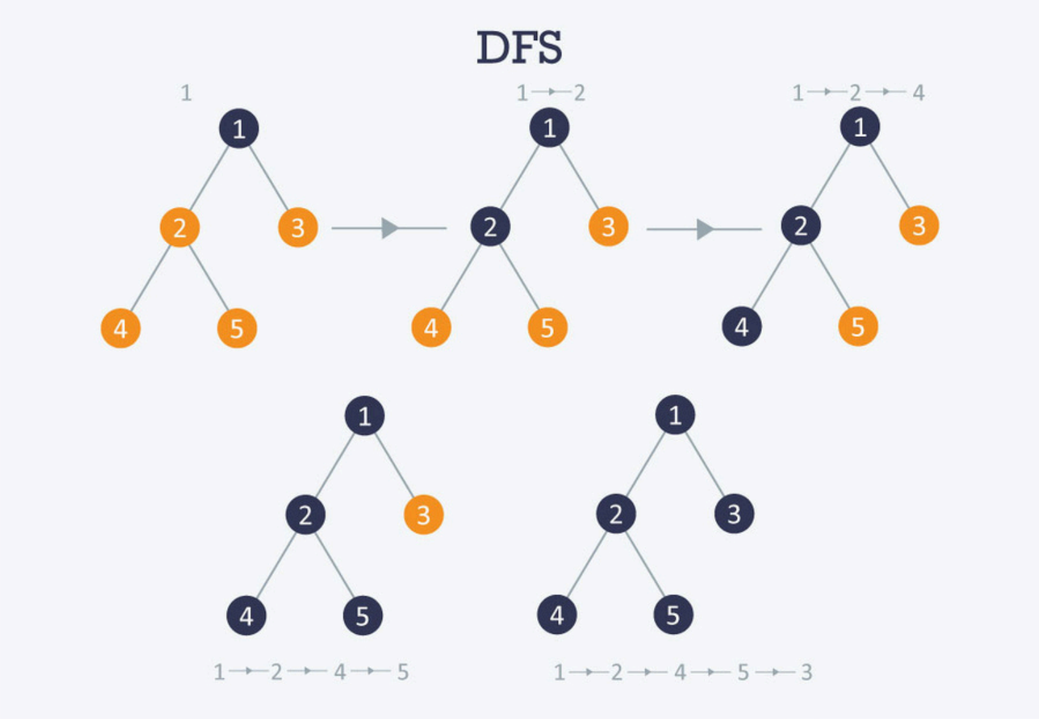
6. Next, expand **Bucharest(278)**:
    - Goal found and expanded with cost 278. **Return this solution**.

- Thus, we see that the first time we expand Bucharest is **not** necessarily **the optimal path**.
- Uniform-cost search ensures we keep searching until the **absolute cheapest path** is the next to expand.

Key Takeaways

- Uniform-cost search expands paths in order of **increasing path cost**, $g(n)$.
- It is **optimal** when all costs are **non-negative**.
- It can handle **different step costs**, unlike **BFS** which assumes all steps have the **same cost**.
- Uses a **priority queue** based on path cost, not on depth.
- Checks for **better paths** to already discovered nodes, ensuring no cheaper path is overlooked.

Depth First Searching (DFS)



- The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking
- This means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse

Advantages

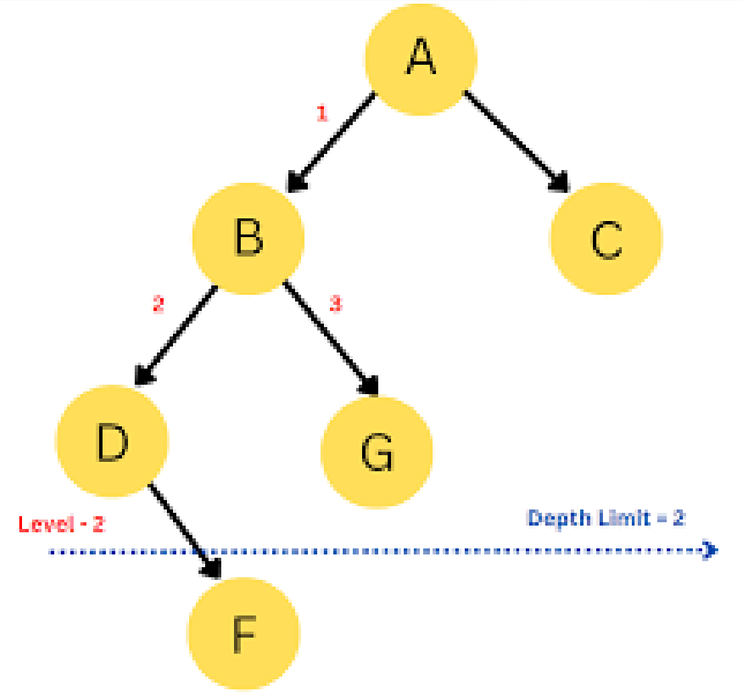
- Simple to implement (recursive)
- Memory efficient in deep search spaces (since it only needs to store a single path at a time)

Disadvantages

- May get stuck in infinite loops (in cyclic graphs) unless cycle checks are implemented
- Not guaranteed to find the shortest path
- Can be inefficient in large search spaces (it may explore many nodes without finding the solution)

DFS is useful for problems where the goal is located deep in the tree or when memory usage is a critical factor.

Introducing Depth Limited Search (DLS)



- The DFS method can be inefficient when dealing with large or infinite trees, as it may explore deep branches that do not contain the goal, leading to wasted time and resources
- Therefore Depth Limited Search is a modified version of DFS that imposes a limit on the depth of the search.

Advantages

- Prevents infinite recursion, especially in infinite-depth spaces
- Allows control over the search depth

Disadvantages

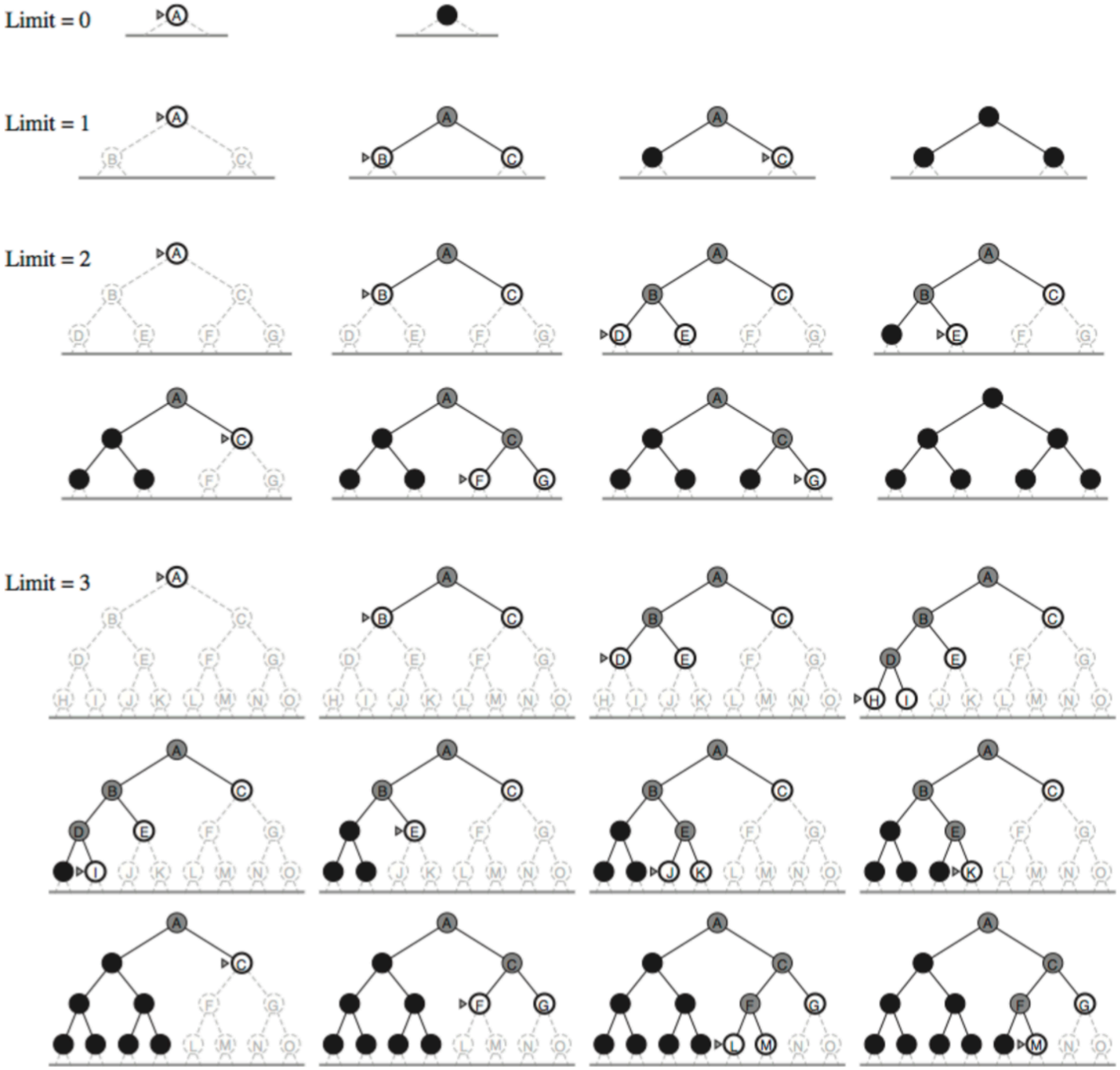
- If the goal is deeper than the given depth limit, the search will fail to find it
- The optimal solution might be overlooked if it is beyond the depth limit

DLS is used when there is a known maximum depth for solutions, or when exploring parts of a large search space where solutions are believed to be close to the root

Iterative Deepening Search (IDS)

- Iterative Deepening Search (IDS) is an iterative graph searching strategy that takes advantage of the completeness of the Breadth-First Search (BFS) strategy but uses much less memory in each iteration (similar to Depth-First Search).

- IDS achieves the desired completeness by enforcing a depth-limit on DFS that mitigates the possibility of getting stuck in an infinite or a very long branch. It searches each branch of a node from left to right until it reaches the required depth. Once it has, IDS goes back to the root node and explores a different branch that is similar to DFS.


Limit = 0
Limit = 1
Limit = 2
Limit = 3

Advantages

- Completes search optimally like BFS (finding the shallowest solution) without consuming too much memory (similar to DFS)
- Always finds the goal if it exists, even if the depth is unknown
- Guarantees that the first solution found is the shortest one

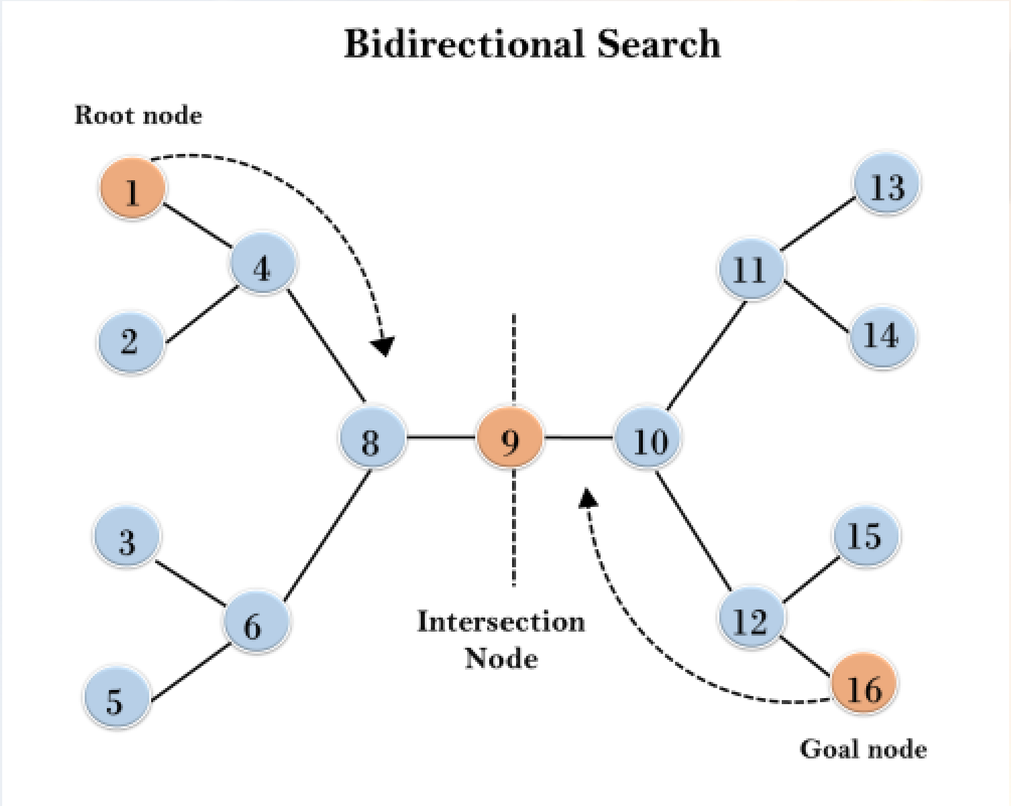
Disadvantages

- Redundant work: many nodes are visited multiple times, leading to inefficiency
- Might require more computation time compared to plain DFS or BFS

Very useful in situations where the search space is large and unknown. IDS is particularly popular in problems like puzzles (e.g., the 8-puzzle), where the depth of the solution isn’t known ahead of time

Bi-directional Searching

- Is an effective search technique used for finding the shortest path between an initial and a goal state.
- It operates by simultaneously running two separate search processes—one forward from the initial state and the other backward from the goal state.



How Bidirectional Searching Works

- **Initial Setup:** Initialize two searches. One starts from the initial state and expands forward. The other starts from the goal state and expands backward.
- **Node Expansion:** Both searches alternately expand the nearest unexplored node. For each node, all possible successors (in the forward direction) or predecessors (in the backward direction) are generated.
- **Checking Intersections:** After each expansion, check if any of the newly generated nodes are present in the frontier of the opposite search.
- **Meeting Point:** Once a common node is discovered, this node acts as the meeting point, and the optimal path can be constructed by joining the paths from the initial state to the meeting point and from the meeting point to the goal state.

Performance evaluation

- **Completeness :** Bidirectional search is complete if BFS is used in both searches.
- **Optimality :** It is optimal if BFS is used for search and paths have uniform cost.
- **Time and Space Complexity :** Time and space complexity is O(bˆ(d/2))

Benefits to BDS

- **Benefits** to bidirectional search include:

1. **Efficiency**: By simultaneously searching from both ends towards the middle, it reduces the search space dramatically, often leading to quicker results.
2. **Optimality**: When used with uniform-cost search strategies, bidirectional search is guaranteed to find an optimal solution if one exists.
3. **Reduced Memory Footprint:** It often requires less memory than traditional algorithms like BFS, especially in densely connected graphs or large search spaces.

Challenges faced in using BDS

- **Challenges** faced in using bidirectional search include:

1. **Implementation Complexity:** Managing two simultaneous searches and ensuring they meet optimally can be complex compared to unidirectional strategies.
2. **Memory Management:** While generally less memory-intensive than some alternatives, bidirectional search still requires careful management of both search frontiers.
3. **Applicability:** It's not suitable for all problems. For instance, it's most effective in searchable spaces where both forward and backward expansion are feasible and meaningful.

Comparison of uninformed search techniques

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]

Thank You!