جامعة الفيصل
Alfaisal University

كلية الهندسة والحوسبة المتقدمة
College of Engineering and Advanced Computing

Introduction to Artificial Intelligence (SE 444)

Lecture 3: Uniformed and Informed Search Algorithms

Prof. Anis Koubaa

akoubaa@alfaisal.edu

This lecture dives deep into the core of AI problem-solving: **search algorithms**. You'll uncover how AI agents navigate complex spaces to find optimal solutions, differentiating between **uninformed methods** like Breadth-First and Depth-First Search, and more efficient **informed techniques** such as A* Search. We will explore the strengths and weaknesses of each, equipping you with the fundamental skills to tackle real-world challenges through intelligent search strategies.

🎓 Week 3 Lecture Plan

Today we'll explore how 🤖 AI systems find 🎯 solutions through 🔍 search techniques:

01

🧩 Problem Formulation & Uninformed Search

Define search problems and examine blind search algorithms ⚡

02

💡 Introduction to Informed Search

Learn how heuristics 🗺️ guide search toward 🎯 solutions

03

⚖️ Comparison & Takeaways

Analyze the tradeoffs between different search 🔍 approaches

04

📦 Wrap-Up & Preview 🔜

Prepare for next lecture's implementation lab 💻

Taxonomy of Search Algorithms

1

By Search Strategy

Uninformed (Blind Search): Operates without domain-specific knowledge.

- Depth-First Search (DFS) - uses Stack frontier
- Breadth-First Search (BFS) - uses Queue frontier
- Uniform-Cost Search (UCS) - uses Priority Queue frontier

Informed (Heuristic Search): Leverages domain-specific knowledge to guide the search.

- Greedy Best-First Search - expands node closest to goal (heuristic only)
- A* Search - combines cost so far + heuristic

2

By Completeness & Optimality

- **Complete:** Guarantees finding a solution if one exists (e.g., BFS, A*).
- **Optimal:** Ensures finding the best/shortest solution (e.g., UCS, A*).

3

By Data Structure for Frontier

- Stack → DFS
- Queue → BFS
- Priority Queue → UCS, A*

💡 Key Idea: The Frontier

The "Frontier" represents the boundary of explored and unexplored nodes in the search space. The choice of data structure for managing this frontier fundamentally defines the search algorithm's behavior.

🔑 Key Terminologies in Search Algorithms

To effectively understand search algorithms, it's crucial to grasp these fundamental concepts and their roles in navigating the search space.

Frontier

The boundary between explored and unexplored nodes in the search space. It contains candidates for expansion.

- **Stack** → DFS
- **Queue** → BFS
- **Priority Queue** → UCS, A*

Explored Set

A collection of nodes that have already been visited. This prevents revisiting nodes and avoids infinite loops in the search process.

Parent Map

A data structure that stores the "parent" of each node. This information is critical for reconstructing the optimal path once the goal node is found.

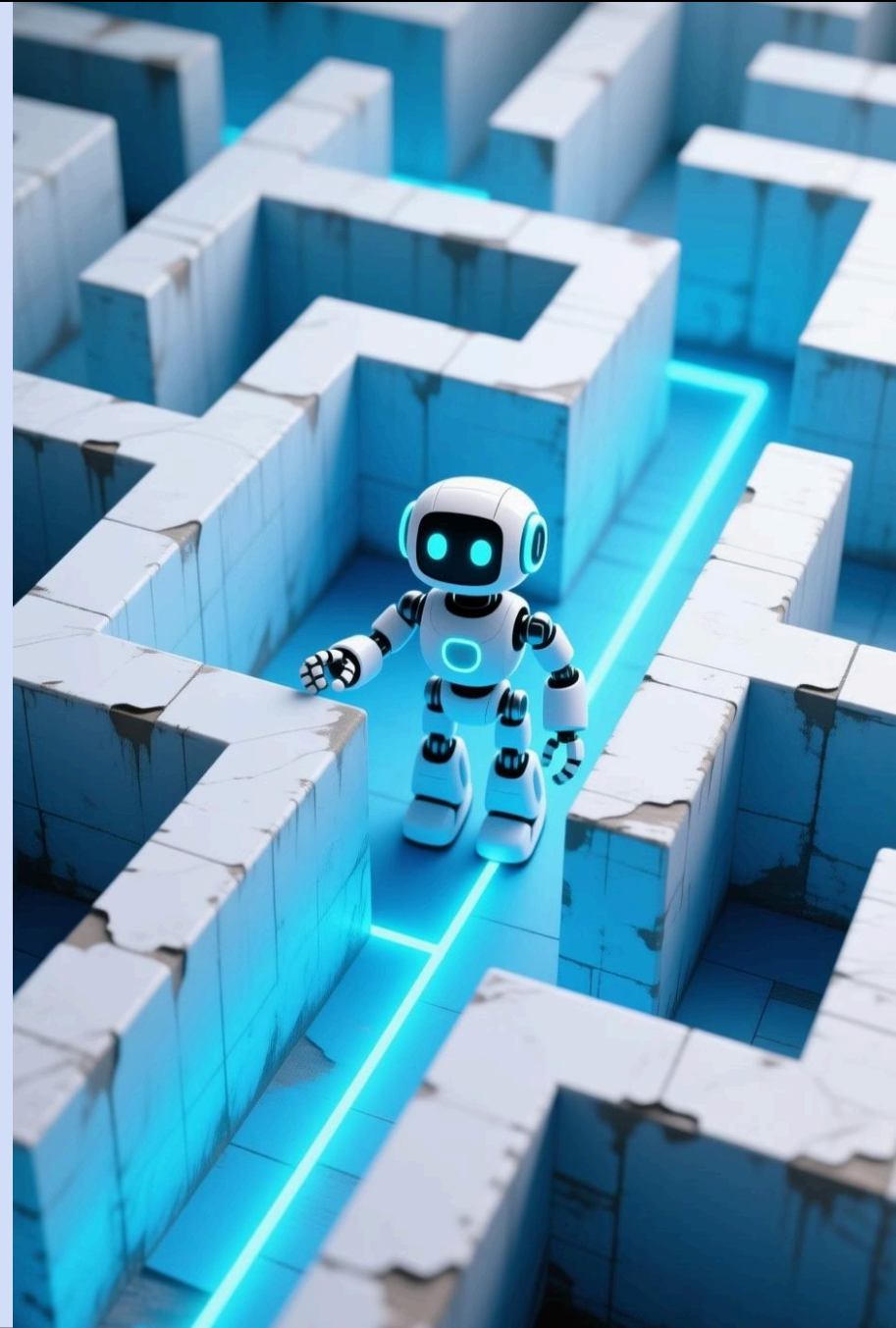
Cost-So-Far

Applicable in weighted searches, this is a dictionary that tracks the best-known cumulative cost to reach each node from the starting point. It's continually updated as cheaper paths are discovered.

Reconstruct Path

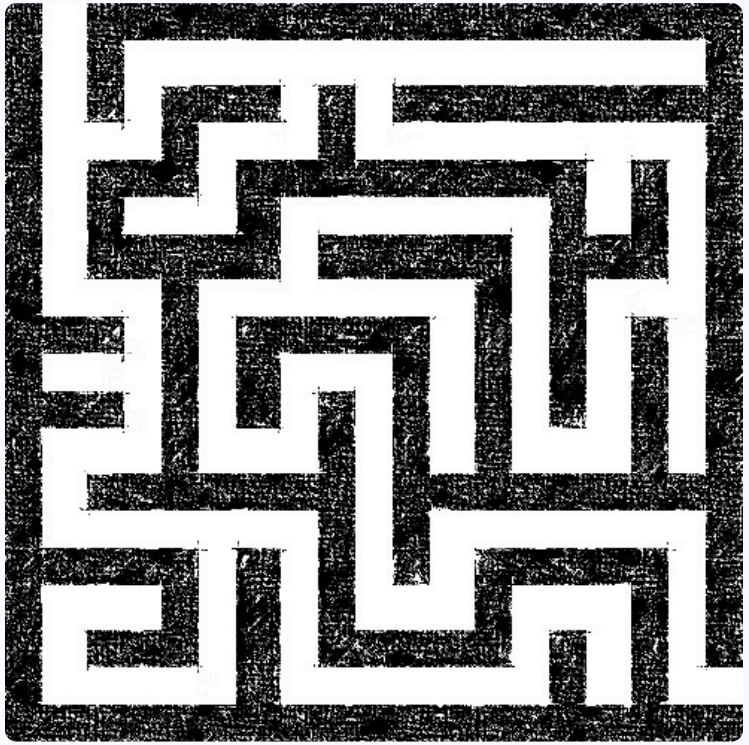
The process of tracing back from the found goal node to the start node using the information stored in the parent map, thereby revealing the solution path.

🧩🔍 Part 1: Problem Formulation & Uninformed Search



🎯 Definition of Search Problems

- Consider the problem of designing 🤖 **goal-based agents** in **known**, **fully observable**, and **deterministic** environments.
- A 🤖 **goal-based agen**t is an intelligent system that determines a sequence of actions to achieve a specific target state.
- In these scenarios, the agent's primary task is **to find a 🗺️ path from its current state to a desired 🎯 goal state**.
- This involves navigating through a defined **"🧩 state space"**, which represents all possible configurations of the environment.

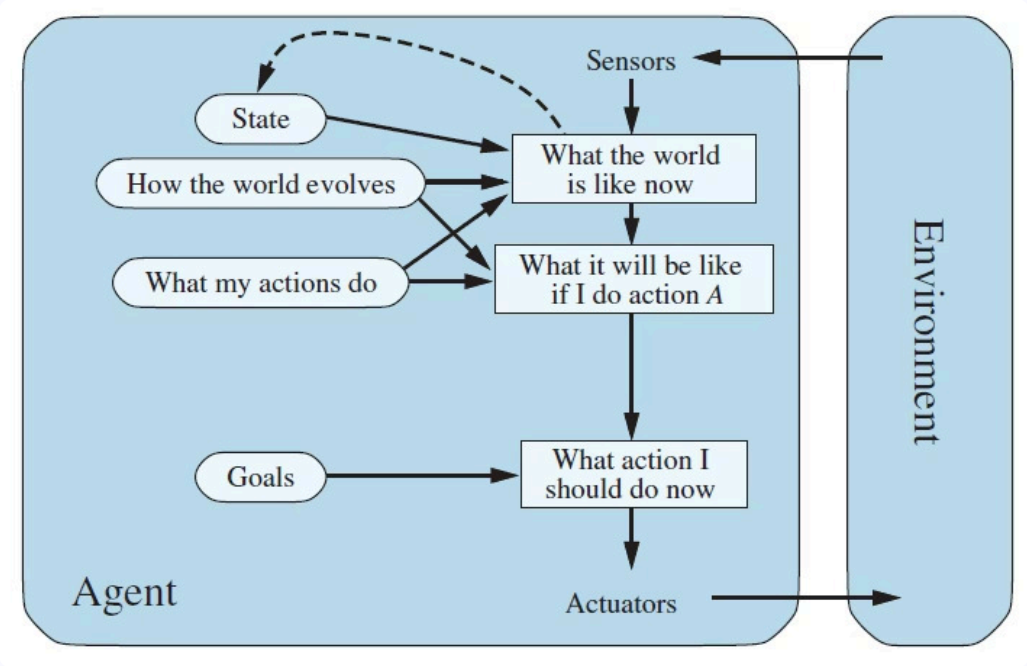


🤖🎯 Goal-based Agent

The agent has the task to reach a defined 🎯 **goal state**.

It uses 🔍 **search algorithms** ⚡ to plan a sequence of actions that lead to the goal.

The performance measure is typically the 💰 **cost** to reach the goal.



The agent observes its environment 🌐, determines possible actions, and selects the one that brings it closer to the goal ✨.

🧩 Search Problem Components

🚀 Initial State

The starting position or configuration

⚡ Actions

Possible moves from each state (e.g., N, E, S, W)

🔄 Transition Model

How actions change the state

🎯 Goal Test

Determines if a goal state has been reached

💰 Path Cost

Function that assigns a cost to each path

The **optimal solution** is the sequence of actions that reaches the goal with the lowest path cost.

Example: Romania Vacation

You're on ✈️ vacation in Saudi Arabia, currently in Riyadh, and need to reach Makkah for your trip tomorrow.

- 📍 **Initial state:** Riyadh
- 🚗 **Actions:** Drive from one city to another
- 🔄 **Transition model:** If you go from city A to B, you end up in B
- 🎯 **Goal state:** Makkah
- 📏 **Path cost:** Sum of distances between cities

This 🗺️ map represents cities in Saudi Arabia with distances in miles.

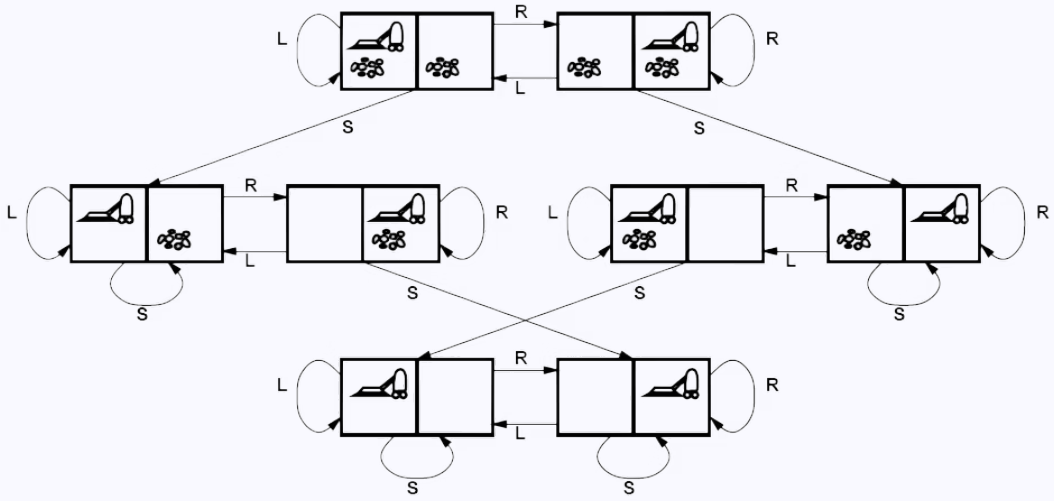
The 🔍 search problem is to find the 🛣️ shortest path from Riyadh to Makkah.

🧹 Example: Vacuum World

A 🤖 vacuum cleaner agent must clean dirty locations. 🧼

- ✨ **Initial state:** Agent location and dirt locations
- 🕹️ **Actions:** Left, Right, Suck
- 🔄 **Transition model:** Moving changes location, sucking removes dirt
- 🎯 **Goal state:** All locations are clean 🌟
- 💰 **Path cost:** Number of actions taken

⚛️ The state space has 8 possible states (2 positions × 2 dirt conditions for each location).

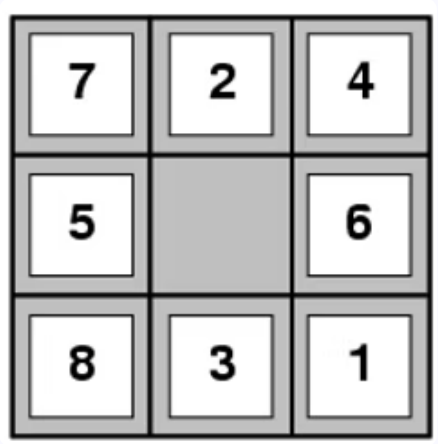


🧩 Example: 8-Puzzle

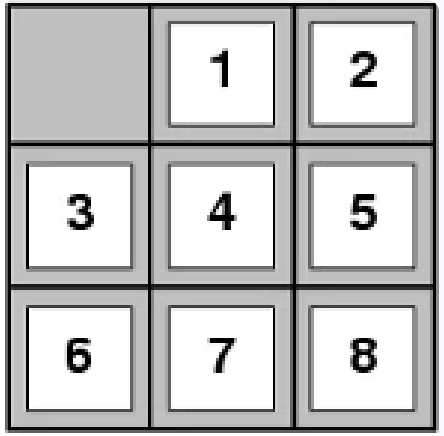
A classic sliding tile puzzle 🧩 with numbered tiles and one empty space.

- ➡️ **Initial state:** Any configuration of tiles
- 🚶 **Actions:** Move blank Left, Right, Up, Down
- 🔄 **Transition model:** Swapping tiles with the blank space
- 🎯 **Goal state:** Tiles arranged in order (1-8) with blank at position 9
- 👣 **Path cost:** One per move

➡️ Initial State



🎯 Goal State

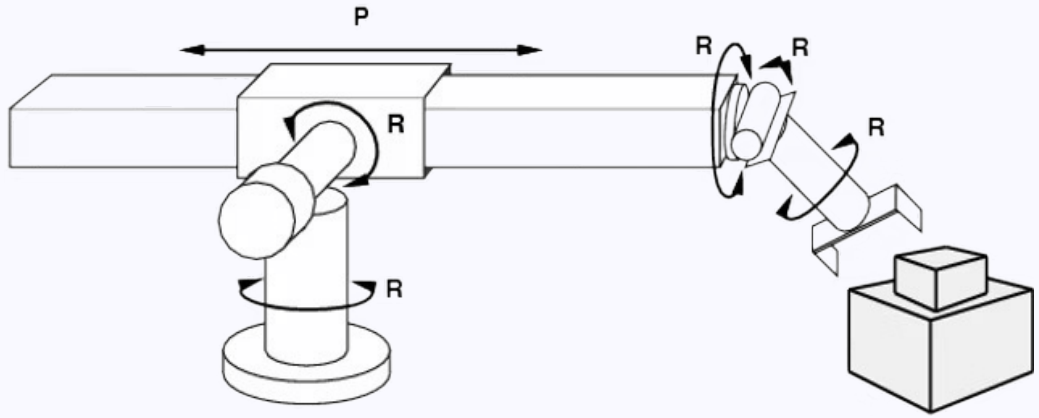


🤖 Example: Robot Motion Planning

A robot arm must move from its current position to grasp an object.

- 📍 **Initial state:** Current arm position
- ➡️ **Actions:** Continuous motions of robot joints
- 🧪 **Transition model:** Physics of arm movement
- 🎯 **Goal state:** Object is grasped
- ⏱️ **Path cost:** Time, energy, smoothness

This problem has a ♾️ continuous state space (infinite possible positions).



🔍 Solving Search Problems

The Main Question

How do we find the **optimal solution** (sequence of **actions/states**)?

The Answer

🌳 Construct a **search tree** for the **state space graph**!

Key Benefits of Search Trees

- Systematically explore possible paths through the state space.
- Expand nodes by applying available actions.
- Create branches representing different choices.

🔍 Solving Search Problems

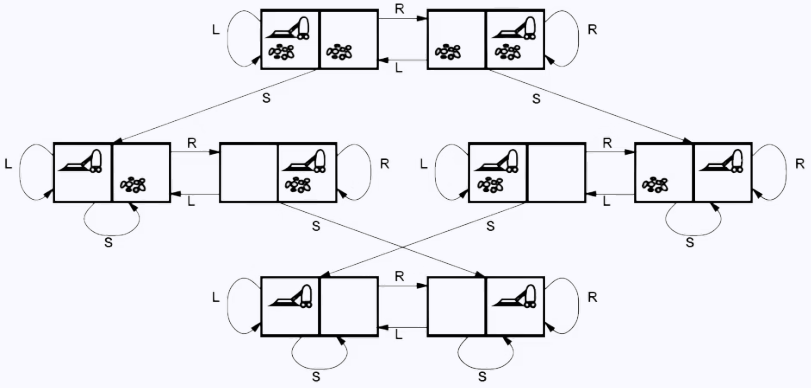
How do we find the optimal solution (sequence of actions/states)?

🌳 Construct a search tree for the state space graph!

A 🔍 search tree allows us to systematically explore possible 🗺️ paths through the state space.

Starting from the 📍 initial state, we expand nodes by applying available actions, creating 🌱 branches that represent different choices.

The search continues until we find a path to a 🎯 goal state or exhaust all possibilities.



🌳 Search Tree

💡 A search tree is a "what if" exploration of possible actions and outcomes:

📍 Root Node

Represents the initial state

🔗 Edges

Represent actions that transition between states

🌱 Child Nodes

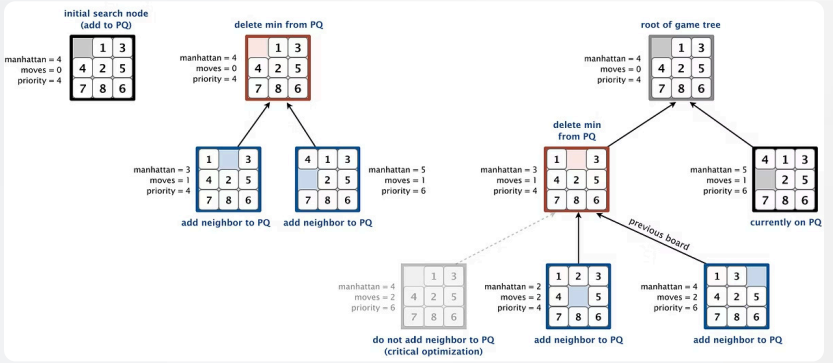
States reached by applying actions to parent states

🗺️ Path

A sequence of actions (edges) connecting nodes

🎯 Solution

A path from the root to a goal node



🌳 Trees must handle 🔄 cycles (paths that return to a previously visited state) and ♻️ redundant paths (multiple ways to reach the same state).

🔍🤖 AI Search vs. Typical Tree Search

🤖🔍 AI Tree/Graph Search

- 🔍 Search space too large 🌌 to fit into memory
  - 🛠️ Builds parts of the tree dynamically as needed
  - 💾 Memory management is critical
- 🔄 Must handle cycles efficiently to minimize loops
- ⚠️ Must check and detect redundant paths

🌳🔍 Typical Tree Search

- 🌳 Assumes a given tree that fits in memory
- 🚫🔄 Trees have no cycles by definition
- ✅ No redundant paths to worry about
- 🗺️ Can traverse the entire structure

In AI search, we're often dealing with state spaces 🌌 that are too large to represent explicitly, so we must 🌱 generate and explore the search tree incrementally.

⚡ Tree Search Algorithm Outline

1. ➕ Initialize the **frontier** 🗺️ (set of unexplored known nodes) using the starting state/root node.
2. 🔄 While the frontier is not empty:
   a. 🔍 Choose a frontier node to expand according to **search strategy**.
   b. 🎯 If the node represents a **goal state**, return it as the solution.
   c. ✨ Else **expand** the node (apply all possible actions) and add its children 🗺️ to the frontier.

💡 The **search strategy** determines which node to expand next, and is what differentiates various search algorithms.

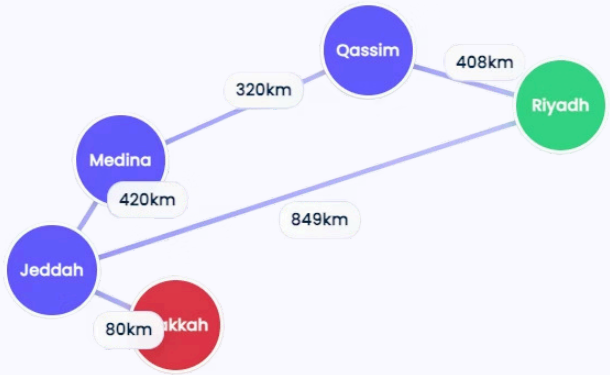
🗺️🔍 Tree Search Example: Saudi Arabia Map

🔍 Starting our search from Riyadh to find Makkah:

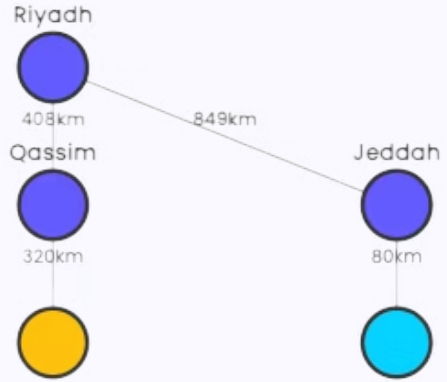
1. 🆕 Initialize **frontier** with Riyadh
2. ➕ Expand Riyadh, adding its neighbors (Qassim, Jeddah) to the frontier
3. 🤔 Choose a node from the frontier based on our search strategy
4. 🎯 Continue expanding nodes until we reach Makkah

⚡ Different **search strategies** will explore the map in different orders.

Saudi Arabia Graph Map



Tree State Space



The ⚡ **search strategy** determines which of these nodes we expand next.

🔍 Search Strategies: Evaluation Criteria

A **search strategy** is defined by picking the **order of node expansion**.

✅ Completeness

Does it always find a solution if one exists?

🏅 Optimality

Does it always find a least-cost solution?

⏱️ Time Complexity

How long does it take to find a solution?

💾 Space Complexity

How much memory does it need?

Complexity is often measured in terms of:

- 🌲 **b:** Maximum branching factor (number of successors per node)
- ⬇️ **d:** Depth of the optimal solution
- ♾️ **m:** Maximum depth of the state space (may be ∞)

📏 State Space Complexity Metrics

When the state space is too large to define explicitly, complexity is measured using these key metrics:

⬇️ Depth of Optimal Solution (d)

The number of actions required to reach the least-cost goal state.

This is crucial for evaluating solution length.

♾️ Maximum Path Length (m)

The maximum number of actions in any path in the state space. It can be infinite if the search space contains cycles.

🌲 Maximum Branching Factor (b)

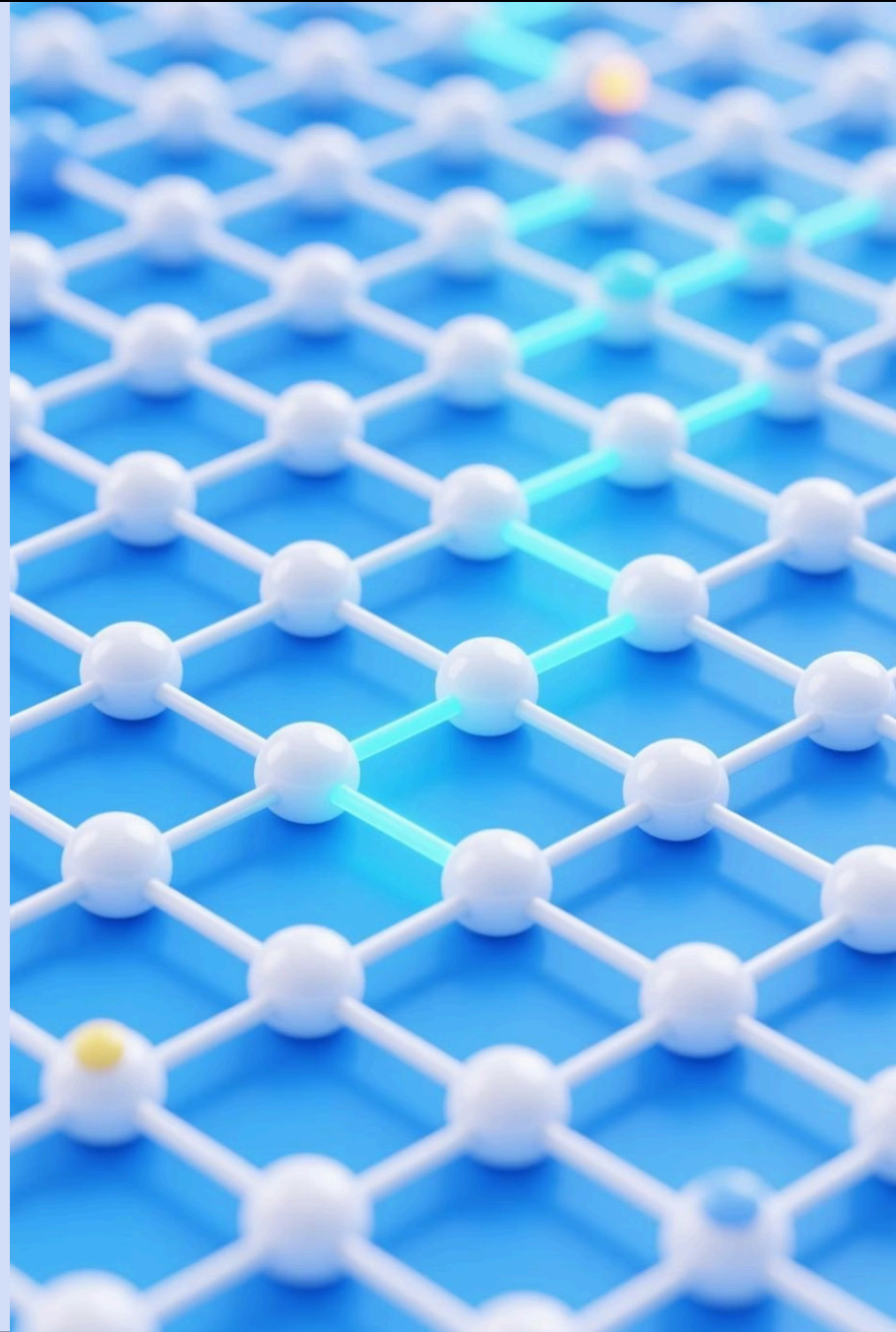
The highest number of successor nodes a single node can have.

It directly impacts the growth rate of the search tree.

These metrics help quantify the **worst-case time and space complexity** of **search algorithms** in an implicitly defined state space.

🗺️ State Space Representation

💡 Understanding the problem space is crucial for effective 🔍 search.



🌐 State Space

The state space represents all possible 🧩 states the agent and environment can be in:

🌐 Complete State Space

All theoretically possible states

➡️ Reachable State Space

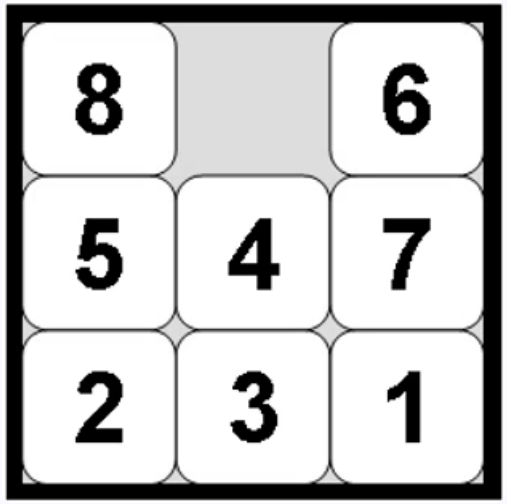
States that can be reached 🎯 from the initial state

🔍 Search Tree

Structure we build to explore the state space

A single state in the state space may be represented by multiple nodes in the search tree 🌳 if there are different paths to reach that state.

📏 State Space Size Examples

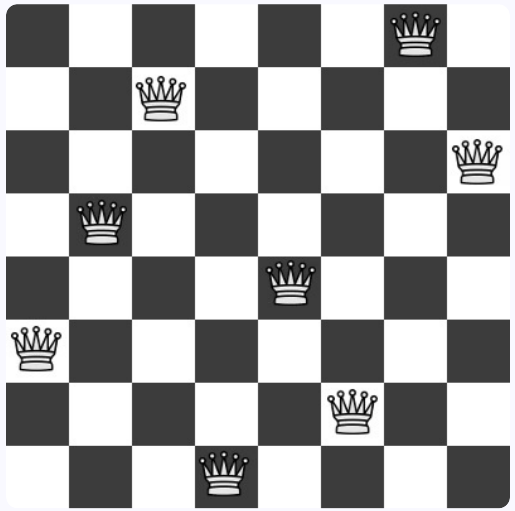


🧩 8-Puzzle

The 8-puzzle has 181,440 reachable states (9!/2).

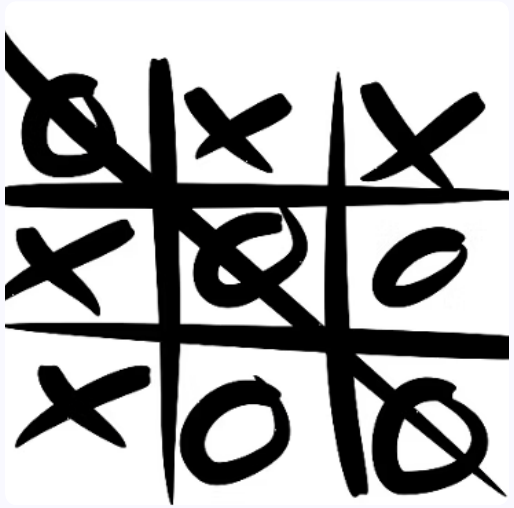
🧩 Maze

The state space for an m×n maze is 2^(m×n), as each cell can be a wall or an open space.



🧩 8-Queens

There are 92 distinct solutions for placing 8 non-attacking queens on an 8x8 board, out of approximately 4.4 billion possible arrangements.



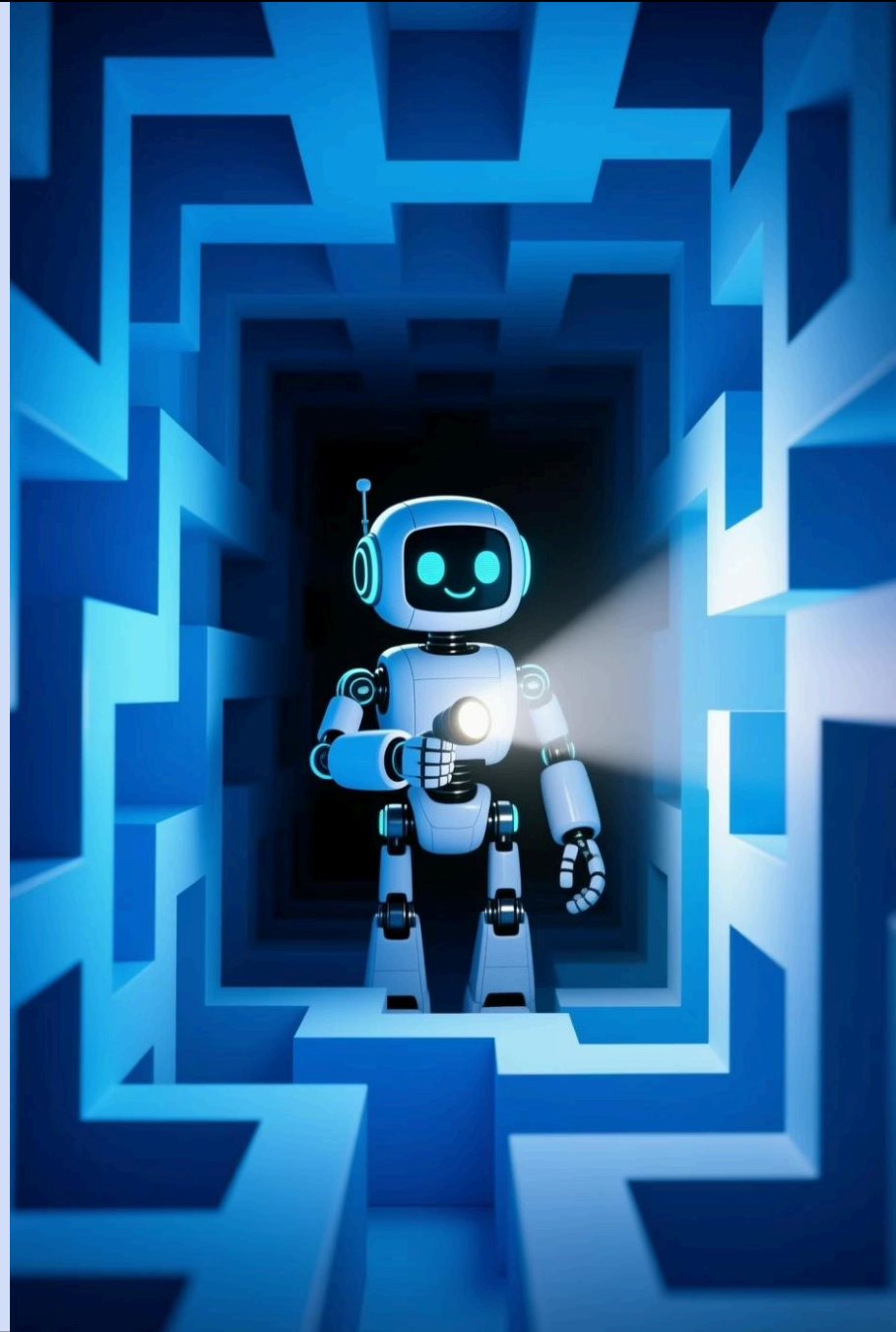
🎮 Tic-tac-toe

Tic-tac-toe has $3^9 = 19{,}683$ possible board configurations, though many are unreachable in gameplay.

💡 The size of the state space gives us an indication of how difficult the search problem will be.

🔍 Uninformed Search Algorithms ⚡

These ⚡ algorithms have no additional 💡 information about states beyond the problem 🎯 definition.



🤖🔍 Uninformed Search Strategies

These ⚡ algorithms **have no additional** 💡 **information about states** beyond the problem 🎯 definition.

The ⚡ search algorithm is **not** provided ❓ information about how close a state is to the 🎯 goal state.

It 🙈 **blindly searches** following a **simple strategy** until it **finds the goal state** 🎲 **by chance**.

🌊 Breadth-First Search (BFS)

Expands the shallowest unexpanded node first ➡️

🌲 Depth-First Search (DFS)

Expands the deepest unexpanded node first ⬇️

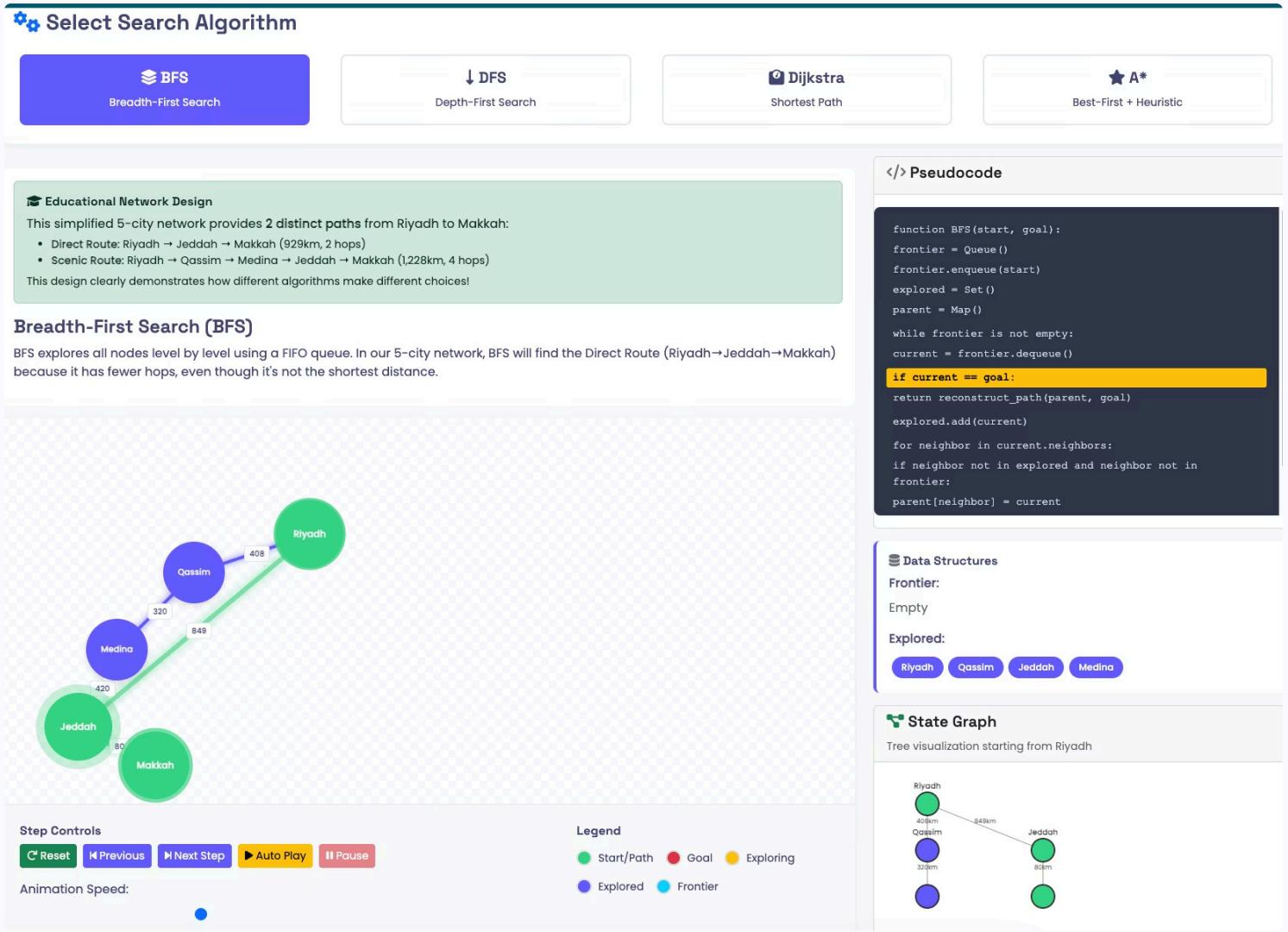
💰 Uniform-Cost Search (UCS)

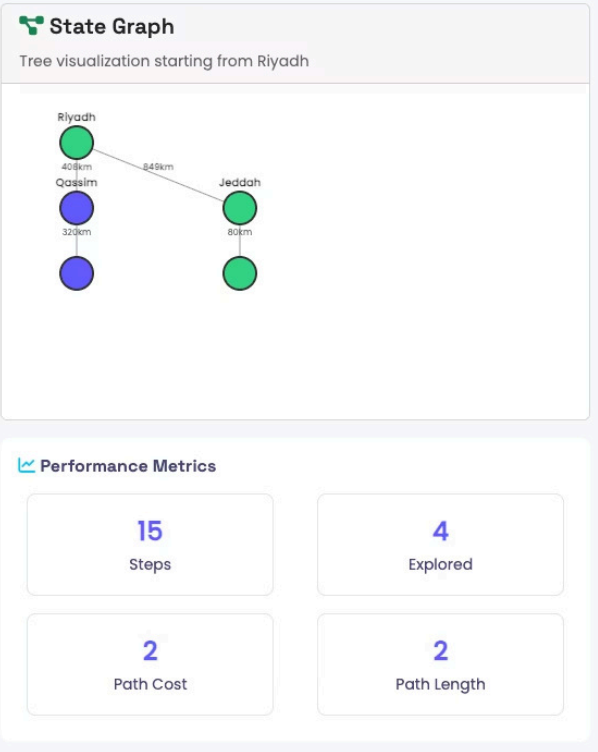
Expands the node with the lowest path cost first ✨

🔄 Iterative Deepening Search (IDS)

Repeatedly runs depth-limited DFS with increasing depth limits 📈

Breadth-First Search (BFS)





⚡ 1. Breadth-First Search (BFS)

➡️ **Expansion rule:** Expand **shallowest unexpanded node** in the frontier (**FIFO** queue)

🗂️ **Data Structures:**

- 🟩 **Frontier:** Queue holding nodes to be expanded (green)
- 🌫️ **Reached:** Set tracking all visited nodes (gray + green)

🌳 BFS builds a tree layer by layer, guaranteeing that the shallowest goal node will be found first.

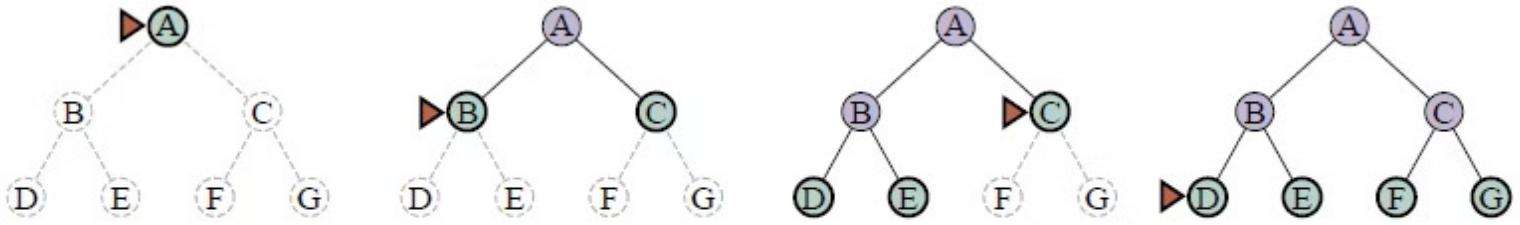


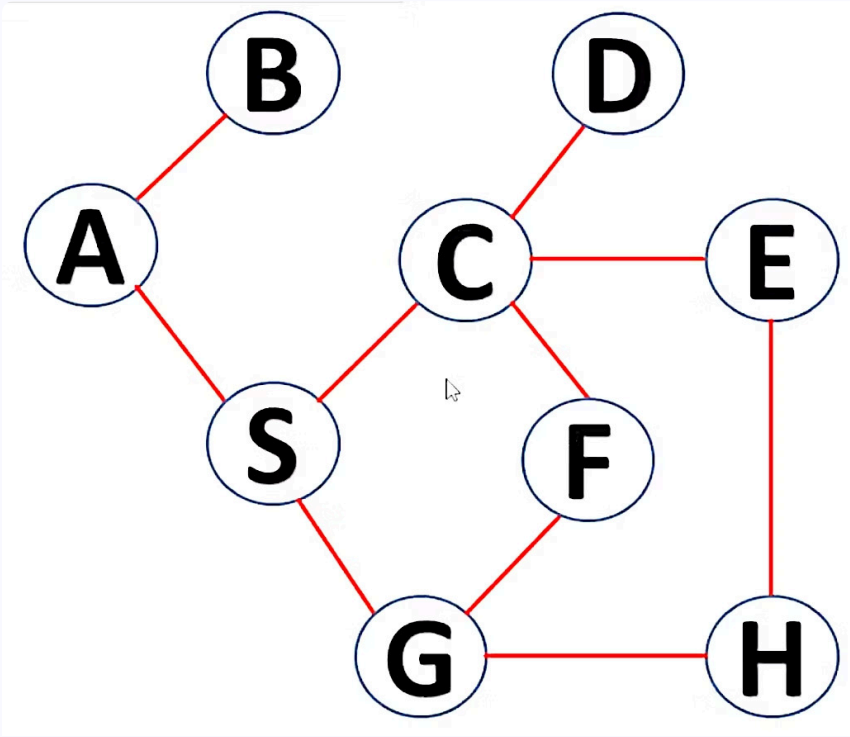
Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

🔍 BFS Example

🪜 BFS explores **all nodes** at the **current depth** before moving to the next level:

1. ➡️ Start at A (level 0)
2. 🗺️ Explore all neighbors of A: B and S (level 1)
3. 🗺️ Explore all neighbors of B, then all neighbors of S (level 2)
4. 🎯 Continue until a goal is found

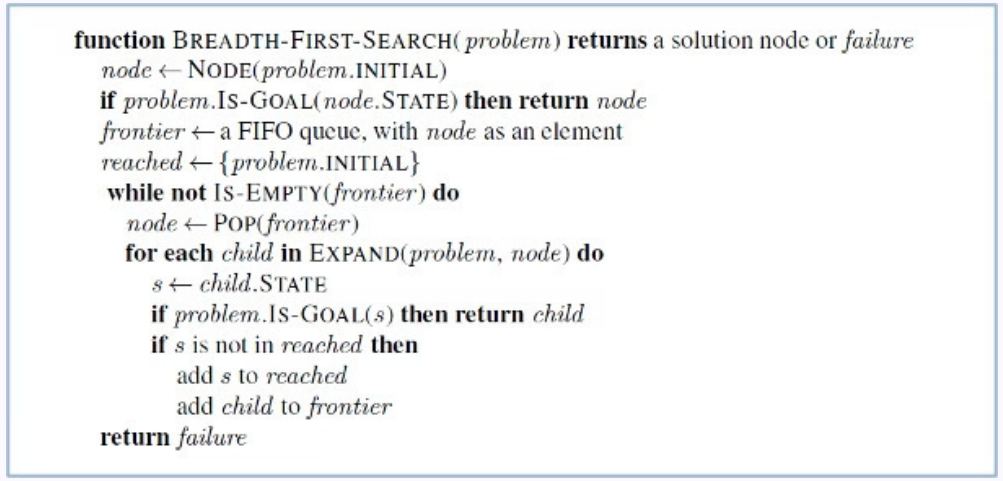
➡️ Output sequence: A → B → S → C → G → D → E → F → H



🎯 BFS finds G, the shallowest goal node, after exploring all nodes at depths 0, 1, and some at depth 2.

🔍 BFS Implementation

```
function BFS(problem):
    node = Node(problem.initial_state)
    if problem.is_goal(node.state): return node

    frontier = Queue() # FIFO queue
    frontier.add(node)
    reached = {node.state} # Set of visited states

    while not frontier.is_empty():
        node = frontier.pop()

        for action in problem.actions(node.state):
            child = child_node(problem, node, action)

            if child.state not in reached:
                if problem.is_goal(child.state): return child
                frontier.add(child)
                reached.add(child.state)

    return failure
```



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 ← a 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*

🔍 BFS Implementation - Key Concepts

```
function BFS(problem):
    node = Node(problem.initial_state)
    if problem.is_goal(node.state): return node

    frontier = Queue() # FIFO queue
    frontier.add(node)
    reached = {node.state} # Set of visited states

    while not frontier.is_empty():
        node = frontier.pop()

        for action in problem.actions(node.state):
            child = child_node(problem, node, action)

            if child.state not in reached:
                if problem.is_goal(child.state): return child
                frontier.add(child)
                reached.add(child.state)

    return failure
```

Key Concepts

- **problem** → Encapsulates search space (initial state, goal test, actions).
- **node** → A wrapper for a state (with parent, action, path cost).
- **frontier (Queue)** → "Line" of nodes waiting to be explored.
- **reached (Set)** → Keeps track of already visited states (avoids loops).
- **child_node()** → Generates a new node by applying an action to the current node.
- **while loop** → Expands nodes in FIFO order (level by level).
- **return child** → As soon as we hit the goal, we stop and return the solution.

👉 **Intuition:**

- Start at the root node.
- Use a **queue** to explore breadth-first.
- Keep a **set** of visited states.
- First solution found = shortest path.

🔍 BFS Implementation Explanation

```
function BFS(problem):
    node = Node(problem.initial_state)
    if problem.is_goal(node.state): return node

    frontier = Queue() # FIFO queue
    frontier.add(node)
    reached = {node.state} # Set of visited states

    while not frontier.is_empty():
        node = frontier.pop()

        for action in problem.actions(node.state):
            child = child_node(problem, node, action)

            if child.state not in reached:
                if problem.is_goal(child.state): return child
                frontier.add(child)
                reached.add(child.state)

    return failure
```

BFS – Step by Step

1. **Problem** → defines the search space:
- `initial_state`, `goal_test`, and possible `actions`.
- **Start Node** → create from *initial state* (starting point). If it's already the goal → return.
- **Frontier (Queue)** → list of nodes to explore, in **FIFO order** (first in, first out).
- **Reached (Set)** → keeps track of **visited states** to avoid repeats.
- **Loop** → while frontier not empty:
- Pop the **first node** from frontier.
- For each **action** from this state → create a **child node** (new state).
- If child state not in *Reached*:
  - If goal → return child (solution found).
  - Else add child to **Frontier** and mark in **Reached**.
- **End** → if frontier empty → return failure (no solution).

🎓 Properties of BFS

✅ Completeness

✅ **Yes** - Will 🎯 find a solution if one exists, as long as the branching factor is finite

🏆 Optimality

✅ **Yes** - If all step costs are identical, finds the 👣 shallowest goal (fewest actions)

⏱️ Time Complexity

$O(b^d)$ - Must 🌳 generate all nodes at depths 1, 2, ..., d

💾 Space Complexity

$O(b^d)$ - Must 📦 store all generated nodes

Where **b** is the branching factor and **d** is the depth of the shallowest solution.

The ⚠️ memory requirements of BFS are its biggest limitation for large problems.

Complexity Notation Variables:

- **b:** Branching factor (average number of successors per node)
- **d:** Depth of the shallowest goal
- **V:** Number of vertices (in a graph formulation)
- **E:** Number of edges (in a graph formulation)

⏳ Time and Memory Requirements in BFS

BFS systematically explores the search space, level by level, ensuring that all nodes at a given depth are expanded before moving to the next. This systematic approach has distinct implications for its time and space requirements.

⏱️ Time Complexity

BFS explores nodes uniformly. In AI search (tree model), where nodes can be regenerated, BFS expands all nodes up to depth 'd' in the worst case.

Nodes Generated: $O(b^d)$
Time Complexity: $O(b^d)$

In graph traversal (graph model), where each node is visited only once, the complexity relates directly to the graph's structure.

Time Complexity: $O(V + E)$

💾 Space Complexity

BFS maintains both the frontier (nodes to visit) and the 'reached' set (visited nodes). Its memory consumption can be significant.

For AI search (tree model), the frontier can store all nodes at the deepest level 'd'.

Space Complexity: $O(b^d)$

For graph traversal (graph model), the memory needed is bounded by the total number of vertices.

Space Complexity: $O(V)$

Understanding these complexities is crucial for selecting the appropriate search algorithm, especially in large-scale problems where resources are limited.

⏳ Time and Memory Requirements in BFS

Depth	Nodes	Time	Memory
2	110	0.11 ms	107 KB
4	11,110	11 ms	10.6 MB
6	106	1.1 sec	1 GB
8	108	2 min	103 GB
10	1010	3 hrs	10 TB
12	1012	13 days	1 PB

🤔 **Assumes:** branching factor b = 10, 1 million nodes/second, 1000 bytes/node

⚠️ Memory requirements **grow exponentially with depth**, making BFS impractical for deep searches.

⚡ 2. Depth-First Search (DFS)

⚙️ **Expansion rule:** Expand the **deepest unexpanded node** in the frontier (**LIFO stack**)

💡 **Key characteristics:**

- 🌲 Explores a single path all the way down before backtracking
- 💾 Uses less memory than BFS (only stores the current path)
- ⚠️ May get stuck in infinite paths without cycle detection
- ❌ Not guaranteed to find the optimal solution

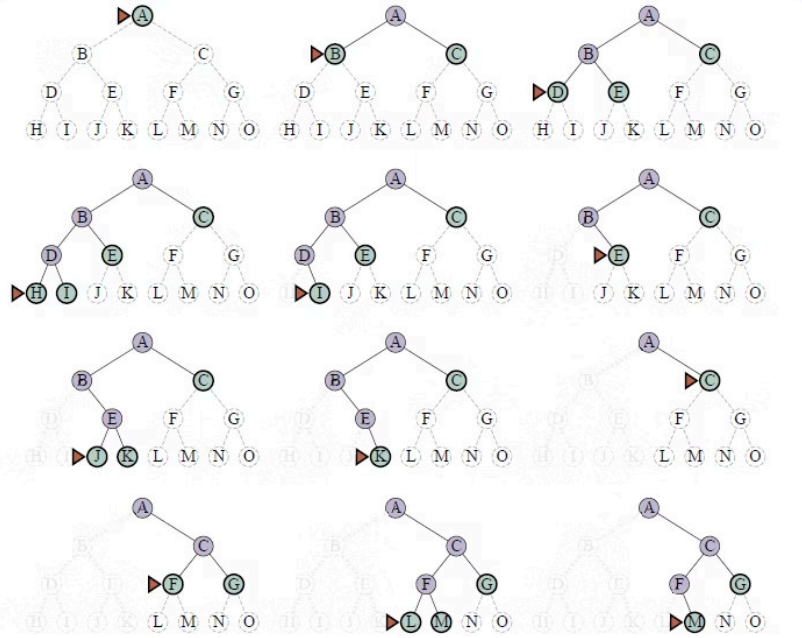


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

⚡ DFS Implementation

```
function DFS(problem):
    node = Node(problem.initial_state)
    if problem.is_goal(node.state): return node

    frontier = Stack() # LIFO stack
    frontier.add(node)
    reached = {node.state} # Set of visited states

    while not frontier.is_empty():
        node = frontier.pop()

        for action in problem.actions(node.state):
            child = child_node(problem, node, action)

            if child.state not in reached:
                if problem.is_goal(child.state): return child
                frontier.add(child)
                reached.add(child.state)

    return failure
```

⚡ DFS Implementation

```
function DFS(problem):
    node = Node(problem.initial_state)
    if problem.is_goal(node.state): return node

    frontier = Stack() # LIFO stack
    frontier.add(node)
    reached = {node.state} # Set of visited states

    while not frontier.is_empty():
        node = frontier.pop()

        for action in problem.actions(node.state):
            child = child_node(problem, node, action)

            if child.state not in reached:
                if problem.is_goal(child.state): return child
                frontier.add(child)
                reached.add(child.state)

    return failure
```

🔑 Key Concepts (DFS)

- **problem** → Defines the search task: initial state, goal test, and actions.
- **node** → Represents a state (with parent link, action taken, path cost).
- **frontier (Stack)** → Stores nodes **to be explored**, but in **LIFO order** (Last-In, First-Out).
- **reached (Set)** → Keeps track of visited states to prevent revisiting.
- **child_node()** → Creates a new node by applying an action to the current node's state.
- **while loop** → Repeatedly pops from the stack, exploring **deep into one path** before backtracking.
- **return child** → The first time the goal is found, it immediately returns the solution

⚡ DFS Implementation

```
function DFS(problem):
    node = Node(problem.initial_state)
    if problem.is_goal(node.state): return node

    frontier = Stack() # LIFO stack
    frontier.add(node)
    reached = {node.state} # Set of visited states

    while not frontier.is_empty():
        node = frontier.pop()

        for action in problem.actions(node.state):
            child = child_node(problem, node, action)

            if child.state not in reached:
                if problem.is_goal(child.state): return child
                frontier.add(child)
                reached.add(child.state)

    return failure
```

🔑 DFS – Step by Step

- **Problem** → defines the search space:
  - `initial_state`, `goal_test`, and possible `actions`.
- **Start Node** → create from *initial state* (starting point). If it's already the goal → return.
- **Frontier (Stack)** → list of nodes to explore, in **LIFO order** (last in, first out).
- **Reached (Set)** → keeps track of **visited states** to avoid repeats.
- **Loop** → while frontier not empty:
  - Pop the **top node** from frontier.
  - For each **action** from this state → create a **child node** (new state).
  - If child state not in *Reached*:
    - If goal → return child (solution found).
    - Else add child to **Frontier** and mark in **Reached**.
- **End** → if frontier empty → return failure (no solution).

👉 **Key Idea:** DFS explores **deep down one path before backtracking**.

⚡ Properties of DFS

🚫 Completeness

No - Can get stuck in infinite paths (unless modified with cycle detection 🔄)

👎 Optimality

No - First goal found may be far from optimal 🎯

⏱️ Time Complexity

O(bm) - Worst case 😱, must search the entire tree 🌳

💾 Space Complexity

O(bm) - Only stores a single path from root to the current node ⬇️, plus unexpanded siblings 📦

Where **b** is the branching factor 🌱 and **m** is the maximum depth 📏 of the search tree.

DFS trades optimality ⚖️ and completeness ✅ for better memory efficiency 🧠.

🔍 Uniform-Cost Search (UCS) / Dijkstra's Algorithm

> 🌍 **Key Idea:** **UCS** expands nodes based on their accumulated **path cost**, $g(n)$, always prioritizing the node with the **lowest cost**. It employs a **priority queue** (**min-heap**) to ensure it finds the **optimal path** when all **edge costs are positive**.

🧩 How It Works

1. Start at the **source node**.
2. Insert the starting node into a **priority queue** with a **path cost** of 0.
3. Repeatedly remove the node with the smallest **path cost** from the **priority queue**.
4. Expand its **neighbors**, calculating their **path costs**. Update costs if a better path is found, and add/re-add them to the **priority queue**.
5. The algorithm stops when:
   - **UCS:** The **goal node** is dequeued from the **priority queue**.
   - **Dijkstra: All nodes** have been processed (if exploring the entire graph for **shortest paths** to all nodes).

Teaching Note:
👉 **Breadth-First Search (BFS)** can be considered a special case of **UCS** where all **edge costs are uniform** (e.g., cost = 1).
👉 **Dijkstra's Algorithm** is essentially **Uniform-Cost Search** applied to finding **shortest paths** from a single source to all other nodes in a graph.

⚡ UCS Implementation

```
function UCS(problem):
    node = Node(problem.initial_state, path_cost=0)
    if problem.is_goal(node.state): return node

    frontier = PriorityQueue()
    frontier.add(node, node.path_cost)

    reached = {node.state: node.path_cost}

    while not frontier.is_empty():
        node = frontier.pop() # Get node with smallest path_cost

        if problem.is_goal(node.state): return node

        for action in problem.actions(node.state):
            child = child_node(problem, node, action)
            if child.state not in reached or child.path_cost < reached[child.state]:
                reached[child.state] = child.path_cost
                frontier.add(child, child.path_cost)

    return failure
```

🔑 UCS – Step by Step

- **Problem** → defines the search space:
  - `initial_state`, `goal_test`, and edge/path costs.
- **Start Node** → begin from ***initial state*** with path cost = 0.
- **Frontier (Priority Queue / Min-Heap)** → stores nodes ordered by their **lowest path cost g(n)**.
- **Reached (Dict / Map)** → tracks the **cheapest cost so far** to each visited state.
- **Loop** → while frontier not empty:
  - Pop the **node with smallest cost**.
  - If it is the **goal** (UCS) → return solution.
  - For each neighbor (via an action):
    - Compute new path cost.
    - If this path is cheaper than known:
      - Update cost in *Reached*.
      - Add/re-add child to **Frontier**.
- **End** →
  - **UCS:** when goal is removed → solution found (optimal).
  - **Dijkstra:** continue until all nodes are processed.

🔍 Uniform-Cost Search (UCS)

➡️ **Expansion rule:** Expand the node with the **lowest path cost** so far

⚙️ Implementation

Uses a priority queue ordered by path cost g(n)

↔️ Differences from BFS

Considers edge costs, may explore deeper before shallow paths if cheaper

✅ Optimality

Guaranteed to find the lowest-cost path to the goal

🎓 UCS is equivalent to Dijkstra's algorithm from graph theory.

🤔 Unlike BFS, UCS may explore nodes at any depth, depending on their cumulative path cost.

⚡ Iterative Deepening Search (IDS)

✨ Combines the benefits of BFS and DFS:

1 🎯 **Run DFS with a depth limit of 0**

Check if the initial state is a goal.

2 🗺️ **Run DFS with a depth limit of 1**

Explore all nodes at depth 1.

3 🗺️ **Run DFS with a depth limit of 2**

Explore all nodes up to depth 2.

4 ✅ **Continue increasing the depth limit**

Until a solution is found or the maximum depth is reached.

👍 IDS gets DFS's good memory footprint while preserving BFS's completeness and optimality (for uniform costs).

⚡ IDS Example

⚡ Although IDS repeats work from previous iterations, its time complexity is still O(bd), dominated by the final iteration.

💡 Most nodes are in the bottom level of the tree, so earlier iterations are relatively quick.

⚡ Properties of IDS

✅ Completeness

Yes - Will find a solution if one exists (like BFS) 🎯

🏆 Optimality

Yes - If step costs are identical (like BFS) ✨

⏱️ Time Complexity

O(bd) - Despite repeating work, asymptotically the same as BFS 🚀

💾 Space Complexity

O(bd) - Only stores the current path (like DFS) 🧠

💡 IDS combines the best features of BFS and DFS:

- 🏆 Optimal like BFS (for uniform costs)
- 💾 Memory-efficient like DFS

🔍 Part 2: Informed Search

💡 Using domain knowledge to guide the search process

🔍 Informed Search

Informed search algorithms use **additional knowledge** 💡 **about the problem** to 🗺️ **guide the search process** more 🚀 efficiently.

> While uninformed search algorithms are "blind," informed search uses a 💡 **heuristic function** to 📏 estimate how close a state is to the 🎯 goal.

💡 Heuristic Function

h(n) = estimated cost from node n to the 🎯 goal

🚀 Better Heuristics

Lead to more ⚡ efficient search by prioritizing ✅ promising paths

⚡ Key Algorithms

Greedy Best-First Search and A* Search

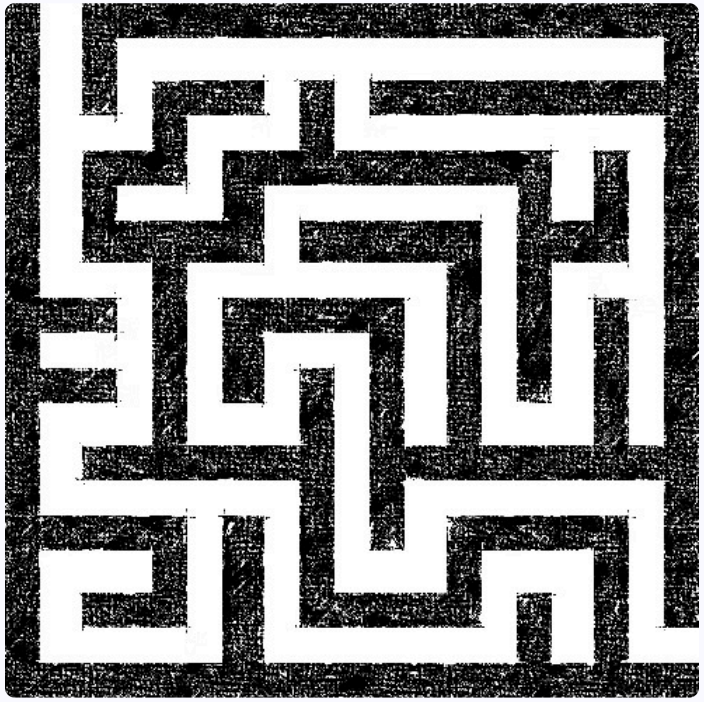
Informed search algorithms can dramatically 📉 reduce the number of nodes explored compared to uninformed search. ✨

💡 Heuristic Function

📏 Euclidean Distance

Straight-line distance between current position and 🎯 goal:

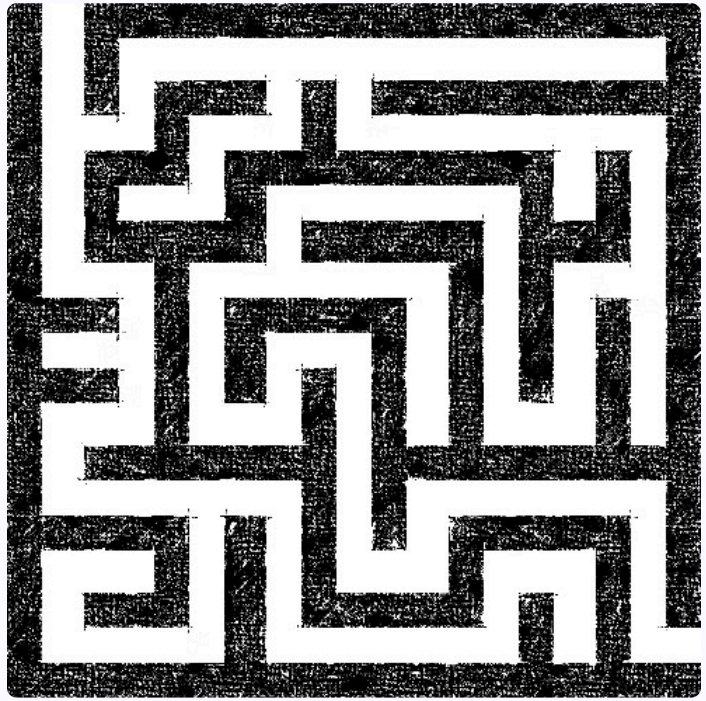
$$h(n) = \sqrt{[(xn - xgoal)^2 + (yn - ygoal)^2]}$$the



🚶 Manhattan Distance

Sum of ⬆️ horizontal and ➡️ vertical distances:

$$h(n) = |xn - xgoal| + |yn - ygoal|$$



A good heuristic provides a reasonable ✅ estimate of the remaining cost to the goal without overestimating it.

Heuristic Functions

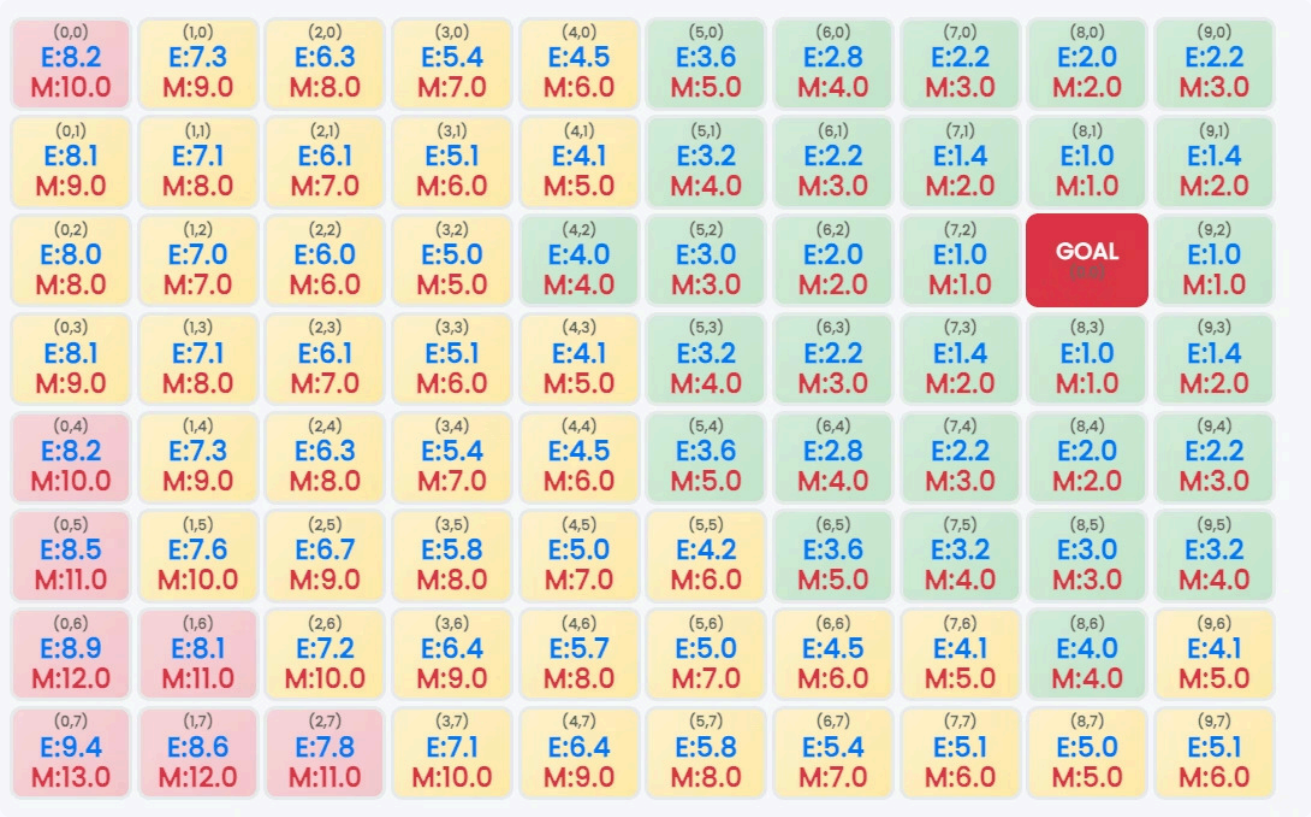
Euclidean Distance Heuristic

(0,0) 8.2	(1,0) 7.3	(2,0) 6.3	(3,0) 5.4	(4,0) 4.5	(5,0) 3.6	(6,0) 2.8	(7,0) 2.2	(8,0) 2.0	(9,0) 2.2
(0,1) 8.1	(1,1) 7.1	(2,1) 6.1	(3,1) 5.1	(4,1) 4.1	(5,1) 3.2	(6,1) 2.2	(7,1) 1.4	(8,1) 1.0	(9,1) 1.4
(0,2) 8.0	(1,2) 7.0	(2,2) 6.0	(3,2) 5.0	(4,2) 4.0	(5,2) 3.0	(6,2) 2.0	(7,2) 1.0	GOAL (8,2)	(9,2) 1.0
(0,3) 8.1	(1,3) 7.1	(2,3) 6.1	(3,3) 5.1	(4,3) 4.1	(5,3) 3.2	(6,3) 2.2	(7,3) 1.4	(8,3) 1.0	(9,3) 1.4
(0,4) 8.2	(1,4) 7.3	(2,4) 6.3	(3,4) 5.4	(4,4) 4.5	(5,4) 3.6	(6,4) 2.8	(7,4) 2.2	(8,4) 2.0	(9,4) 2.2
(0,5) 8.5	(1,5) 7.6	(2,5) 6.7	(3,5) 5.8	(4,5) 5.0	(5,5) 4.2	(6,5) 3.6	(7,5) 3.2	(8,5) 3.0	(9,5) 3.2
(0,6) 8.9	(1,6) 8.1	(2,6) 7.2	(3,6) 6.4	(4,6) 5.7	(5,6) 5.0	(6,6) 4.5	(7,6) 4.1	(8,6) 4.0	(9,6) 4.1
(0,7) 9.4	(1,7) 8.6	(2,7) 7.8	(3,7) 7.1	(4,7) 6.4	(5,7) 5.8	(6,7) 5.4	(7,7) 5.1	(8,7) 5.0	(9,7) 5.1

Manhattan Distance Heuristic

(0,0) 10.0	(1,0) 9.0	(2,0) 8.0	(3,0) 7.0	(4,0) 6.0	(5,0) 5.0	(6,0) 4.0	(7,0) 3.0	(8,0) 2.0	(9,0) 3.0
(0,1) 9.0	(1,1) 8.0	(2,1) 7.0	(3,1) 6.0	(4,1) 5.0	(5,1) 4.0	(6,1) 3.0	(7,1) 2.0	(8,1) 1.0	(9,1) 2.0
(0,2) 8.0	(1,2) 7.0	(2,2) 6.0	(3,2) 5.0	(4,2) 4.0	(5,2) 3.0	(6,2) 2.0	(7,2) 1.0	GOAL (8,2)	(9,2) 1.0
(0,3) 9.0	(1,3) 8.0	(2,3) 7.0	(3,3) 6.0	(4,3) 5.0	(5,3) 4.0	(6,3) 3.0	(7,3) 2.0	(8,3) 1.0	(9,3) 2.0
(0,4) 10.0	(1,4) 9.0	(2,4) 8.0	(3,4) 7.0	(4,4) 6.0	(5,4) 5.0	(6,4) 4.0	(7,4) 3.0	(8,4) 2.0	(9,4) 3.0
(0,5) 11.0	(1,5) 10.0	(2,5) 9.0	(3,5) 8.0	(4,5) 7.0	(5,5) 6.0	(6,5) 5.0	(7,5) 4.0	(8,5) 3.0	(9,5) 4.0
(0,6) 12.0	(1,6) 11.0	(2,6) 10.0	(3,6) 9.0	(4,6) 8.0	(5,6) 7.0	(6,6) 6.0	(7,6) 5.0	(8,6) 4.0	(9,6) 5.0
(0,7) 13.0	(1,7) 12.0	(2,7) 11.0	(3,7) 10.0	(4,7) 9.0	(5,7) 8.0	(6,7) 7.0	(7,7) 6.0	(8,7) 5.0	(9,7) 6.0

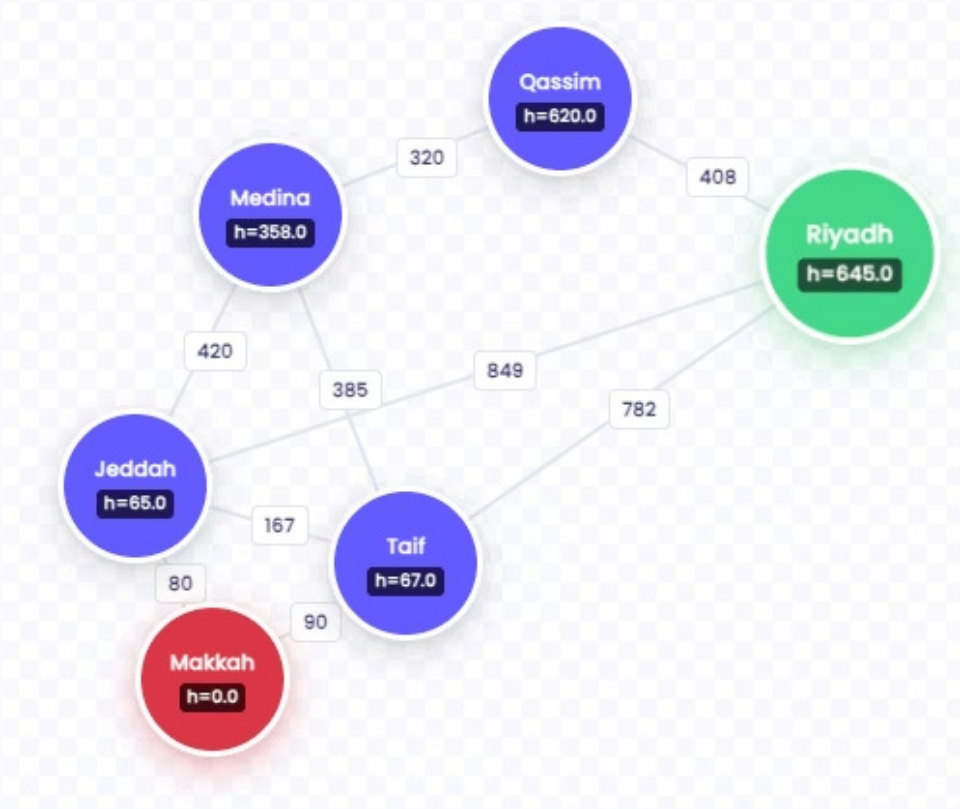
Comparing Both Heuristics



🗺️ Heuristic for the Saudi Arabia Problem

🗺️ For the Saudi Arabia travel problem, we can use the straight-line distance to **Makkah** as our heuristic, our final destination:

City	h(n) to Makkah
Riyadh	645
Qassim	620
Taif	356
Jeddah	55
Medina	358
Makkah	0



🎯 This heuristic gives us an **estimate of how far each city is from our goal (Makkah)** within the Saudi Arabian road network.

⚡ 1. Greedy Best-First Search

➡️ **Expansion rule:** Expand the node that has the **lowest value of the heuristic function h(n)**.

Greedy Best-First Search always moves toward the state that appears closest to the goal, regardless of the path cost so far. It prioritizes nodes based solely on how close the heuristic estimates they are to the goal.

🔍 Greedy Best-First Search Example: Saudi Arabia

Let's trace the execution on the Saudi Arabia map, aiming for **Makkah (h=0)**. We'll use the straight-line distance heuristics previously defined:

- **Start at Riyadh** (h = 645 to Makkah)
- From Riyadh, consider neighbors:
  - To **Medina** (h = 358)
  - To **Taif** (h=67)
  - To **Jeddah** (h = 55)
- Greedy selects **Jeddah** (h=80) because it has the lowest heuristic value.
- From Jeddah, expand to **Makkah** (h = 0, goal!).

Path found: Riyadh → Jeddah → Makkah **(Not the shortest)**

This path illustrates how **Greedy Best-First Search** makes choices based on the most promising immediate next step according to the heuristic, not considering the actual path cost from the start.

⚡ Greedy Best-First Search

➡️ **Expansion rule:** Expand the node that has the **lowest value of the heuristic function h(n)**.

Greedy Best-First Search always moves toward the state that appears closest to the goal, regardless of the path cost so far.

It **prioritizes nodes** based solely on **how close the heuristic estimates** they are to the goal.

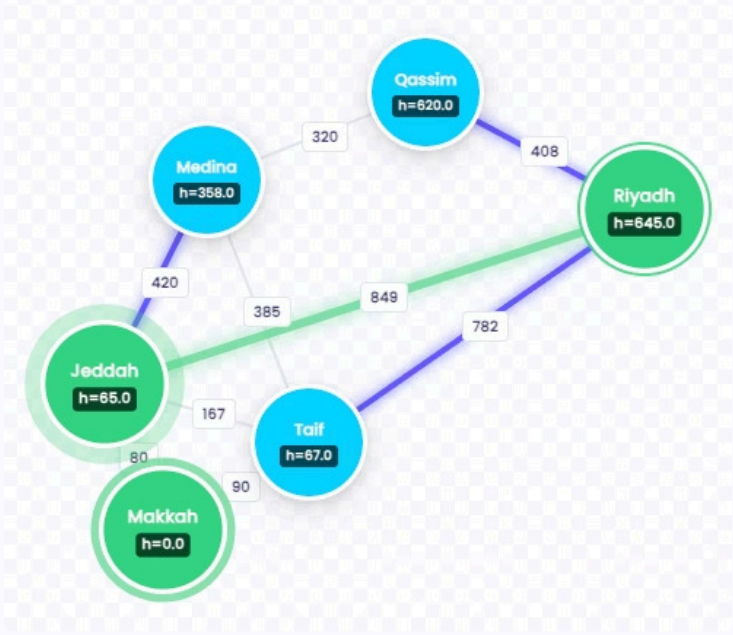
🔍 Greedy Best-First Search Example: Saudi Arabia

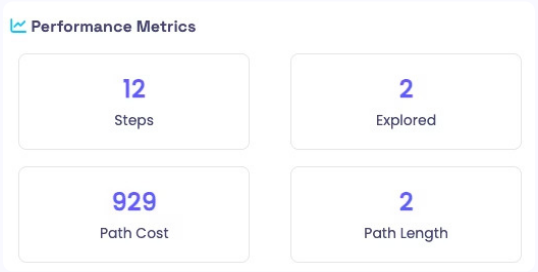
Let's trace the execution on the Saudi Arabia map, aiming for **Makkah (h=0)**. We'll use the straight-line distance heuristics previously defined:

- **Start at Riyadh** (h = 780 to Makkah)
- From Riyadh, consider neighbors:
  - To **Medina** (h = 480)
  - To **Jeddah** (h = 80)
- Greedy selects **Jeddah** (h=80) because it has the lowest heuristic value.
- From Jeddah, expand to **Makkah** (h = 0, goal!).

Path found: Riyadh → Jeddah → Makkah

This path illustrates how **Greedy Best-First Search** makes choices based on the most promising immediate next step according to the heuristic, not considering the actual path cost from the start.





⚡ Properties of Greedy Best-First Search

Completeness

✅ **Yes** - In finite spaces with cycle detection

Optimality

❌ **No** - May find suboptimal paths

⏱️ Time Complexity

O(bm) - In worst case, may explore all paths

💾 Space Complexity

O(bm) - Must store all generated nodes

While **Greedy Best-First Search** often works well in practice, **it's not optimal.**

For example, our path (450 miles) isn't the shortest. A better route 🛣️ would be:

⚡ Greedy Best-First Search Implementation

```
function GREEDY-BEST-FIRST-SEARCH(problem):
  node = Node(problem.initial_state)
  frontier = PriorityQueue(ordered by h)
  frontier.add(node, h(node))
  reached = {node.state: node}

  while not frontier.is_empty():
    node = frontier.pop()
    if problem.is_goal(node.state): return node

    for action in problem.actions(node.state):
      child = child_node(problem, node, action)
      s = child.state

      if s not in reached or child.path_cost < reached[s].path_cost:
        reached[s] = child
        frontier.add(child, h(child))

  return failure
```

Greedy Best-First Search – Step by Step

- **Problem** → defines the search space:
  - `initial_state`, `goal_test`, and actions.
- **Start Node** → begin from *initial state*.
- **Frontier (Priority Queue / Min-Heap)** → stores nodes ordered by **heuristic value $h(n)$** (estimate of closeness to goal).
- **Reached (Dict / Map)** → records the best node found so far for each visited state.
- **Loop** → while frontier not empty:
  - Pop the **node with lowest $h(n)$**.
  - If it is the **goal** → return solution.
  - For each neighbor (via an action):
    - Create child node.
    - If state is new or has a cheaper path cost:
      - Update *Reached*.
      - Add child to **Frontier**, prioritized by $h(child)$.
- **End** → if frontier empty → return failure (no solution).

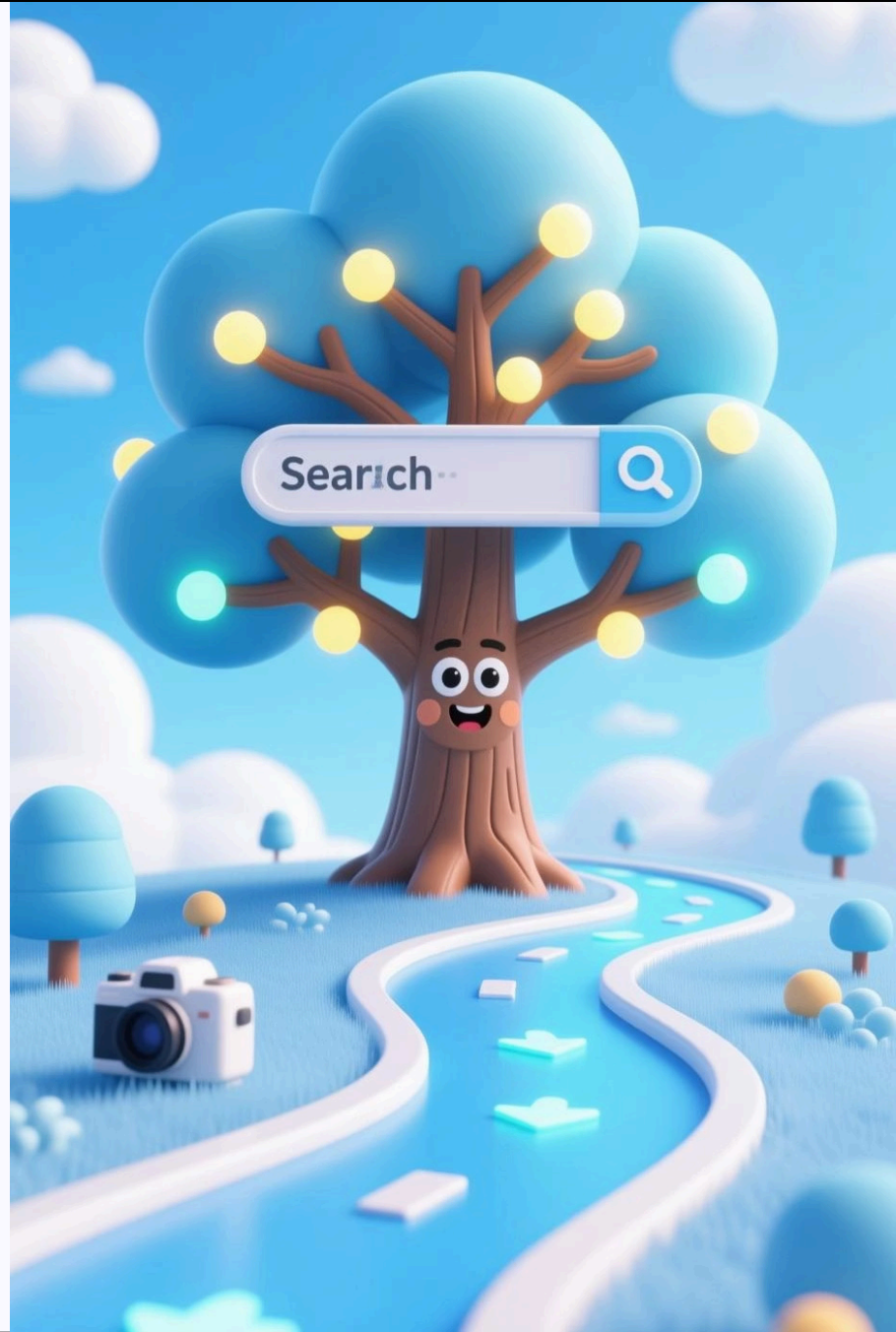
🎯 The Optimality Problem with Greedy Search

Greedy search has a fundamental ⚡ flaw: **it considers only the estimated distance to the goal (h(n)) and ignores the cost already incurred (g(n)).**

This can lead to 🗺️ **suboptimal paths** when a slightly longer initial path leads to a much shorter overall path.

We need an 💡 algorithm that balances:

- 🗺️ The path cost so far
- 🎯 The estimated cost to the goal



⚡ 2. A* Search

➡️ **Expansion rule:** Expand the node that has the **lowest value of the evaluation function f(n) = g(n) + h(n)**.

A* balances the path cost incurred so far (g(n)) with the estimated cost to the goal (h(n)), leading to an efficient and optimal solution when using an admissible heuristic.

💡 A* combines the best of both worlds:

🔢 Evaluation Function

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

where **g(n)** = cost so far to reach n

and **h(n)** = estimated cost from n to goal

⚙️ Expansion Rule

Expand the node with the lowest f(n) value

💡 Key Insight

Balances the cost incurred so far with the estimated remaining cost

✅ A* is optimal if h(n) is **admissible** (never overestimates the true cost to the goal).

⚡ 2. A* Search Example: Saudi Arabia

Using the Saudi Arabia map and defined **heuristics (h = o for Makkah)**, let's trace A*:

- **Start at Riyadh:** g = 0, h = 645.
  f(Riyadh) = 0 + 645 = 645.

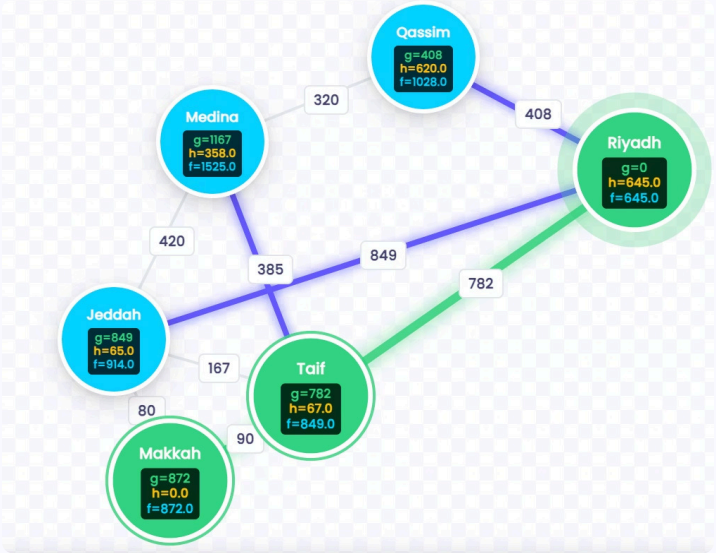
From Riyadh, consider neighbors:

- **Qassim:** g = 408, h = 620. f(Qassim) = 408 + 620 = 1028.
- **Taif:** g = 782, h = 67. f(Taif) = 782 + 67 = 849.
- **Jeddah:** g = 849, h = 55. f(Jeddah) = 849 + 55 = 904.
- **Medina:** g = 1167, h = 358. f(Medina) = 1167 + 358 = 1525.

A* selects Taif (f = 849) as it has the lowest f(n).

From Taif, expand to Makkah:

g(Makkah) = 782 (Riyadh–Taif) + 90 (Taif–Makkah) = 872.
h(Makkah) = 0. f(Makkah) = 872 + 0 = 872 (goal reached!).



Path found: Riyadh → Taif → Makkah
Total cost: 872 (shortest)

⚡ A* Search Implementation

```
function A_STAR_SEARCH(problem):
  node = Node(problem.initial_state)
  frontier = PriorityQueue(ordered by f)
  frontier.add(node, f(node))   # f(n) = g(n) + h(n)
  reached = {node.state: node}

  while not frontier.is_empty():
    node = frontier.pop()
    if problem.is_goal(node.state): return node

    for action in problem.actions(node.state):
      child = child_node(problem, node, action)
      s = child.state

      if s not in reached or child.path_cost < reached[s].path_cost:
        reached[s] = child
        frontier.add(child, f(child))   # f = g + h

  return failure
```

A* Best-First Search – Step by Step

- **Problem** → defines the search space:
  - `initial_state`, `goal_test`, actions, and path costs.
- **Start Node** → begin from *initial state* with cost = 0.
- **Frontier (Priority Queue / Min-Heap)** → stores nodes ordered by **$f(n) = g(n) + h(n)$** (actual cost + heuristic).
- **Reached (Dict / Map)** → records the best (lowest-cost) node found so far for each visited state.
- **Loop** → while frontier not empty:
  - Pop the **node with lowest $f(n)$**.
  - If it is the **goal** → return solution.
  - For each neighbor (via an action):
    - Create child node.
    - If state is new or has a cheaper path cost:
      - Update *Reached*.
      - Add child to **Frontier**, prioritized by $f(child) = g(child) + h(child)$.
- **End** → if frontier empty → return failure (no solution).

⚡ Properties of A* Search

✅ Completeness

Yes - If there is a solution 🎯 and the branching factor 🌳 is finite

🏆 Optimality

Yes - If h(n) is admissible (never overestimates) 👍

⏱️ Time Complexity

📈 Exponential in the worst case, but often much better ✨ in practice

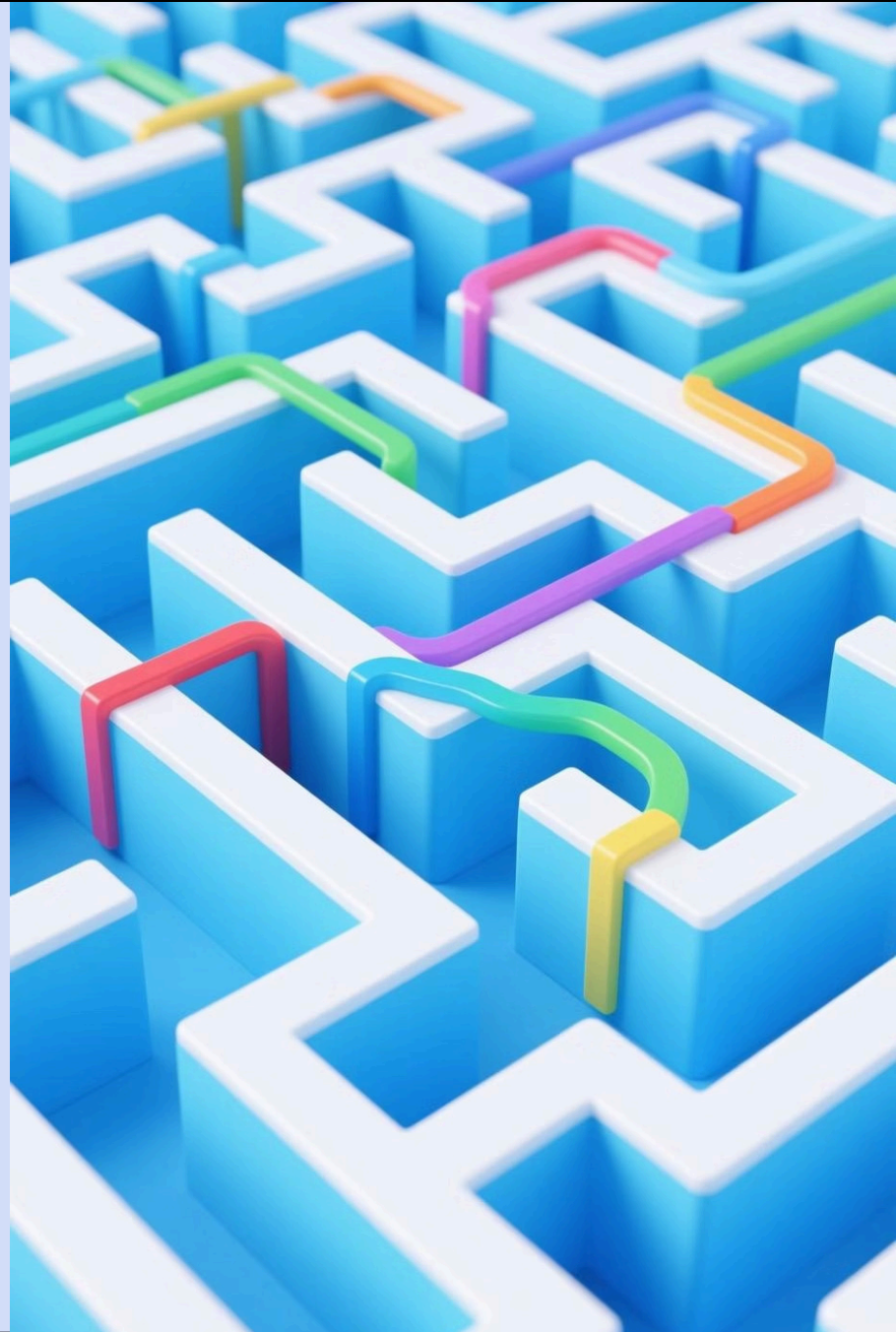
💾 Space Complexity

Must store 🧠 all generated nodes (like BFS) 🕸️

A* is the most ⭐ popular informed search algorithm because it combines efficiency ⚡ with optimality 🏆 guarantees.

The quality of the heuristic function 🎯 greatly impacts A*'s performance 📈.

💡 Part 3: Comparison & Takeaways



🔍⚡ Search Algorithm Comparison

Algorithm	Complete?	Optimal?	Time	Space
BFS	✅ Yes	✅ Yes*	O(bd)	O(bd)
UCS	✅ Yes	✅ Yes	O(bC*/ε)	O(bC*/ε)
DFS	❌ No	❌ No	O(bm)	O(bm)
IDS	✅ Yes	✅ Yes*	O(bd)	O(bd)
Greedy	✅ Yes**	❌ No	O(bm)	O(bm)
A*	✅ Yes**	✅ Yes***	Exponential	Exponential

💡 * If step costs are identical

💡 ** If space is finite

💡 *** If heuristic is admissible

💡 Key Takeaways

🧩 Problem Formulation is Critical

Properly defining the state space, actions, and 🎯 goal test is the first ⚙️ step in solving any search problem

🙈 Uninformed Search = Blind

Explores everything systematically without ❓ knowledge of where the 🗺️ goal is

🧭 Informed Search = Guided

Uses 🧠 domain knowledge (heuristics) to 🔎 focus the search toward promising areas

⚖️ Algorithm Selection Involves Tradeoffs

Choose based on ✅ completeness, ✨ optimality, ⏱️ time, and 💾 space requirements for your specific problem

😊 Thank You!

🧑‍💻 See you in the next lab session

💡 Remember to bring your laptops 💻 and be prepared to implement these search algorithms 🔍.