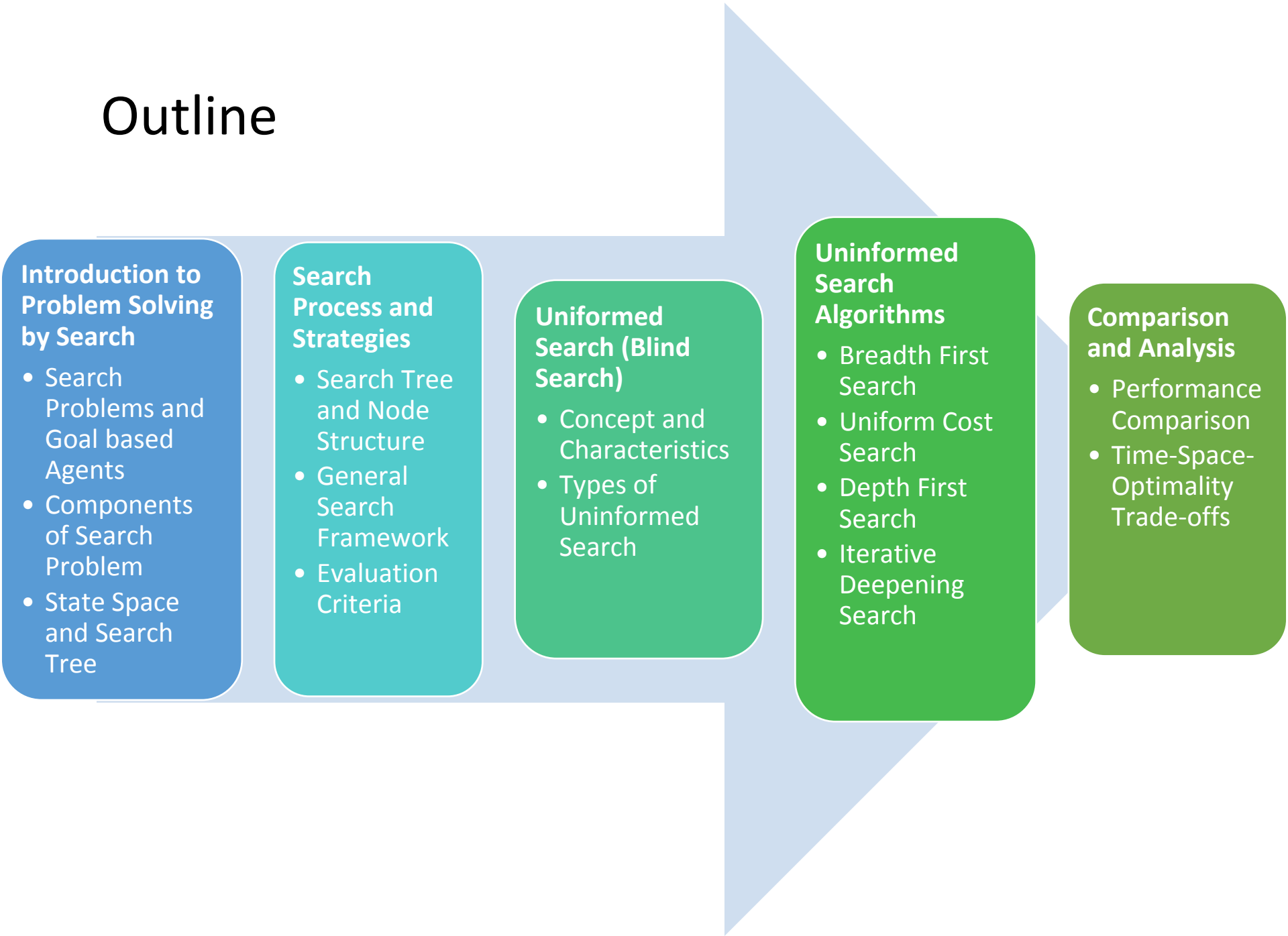
Artificial Intelligence (CS280)

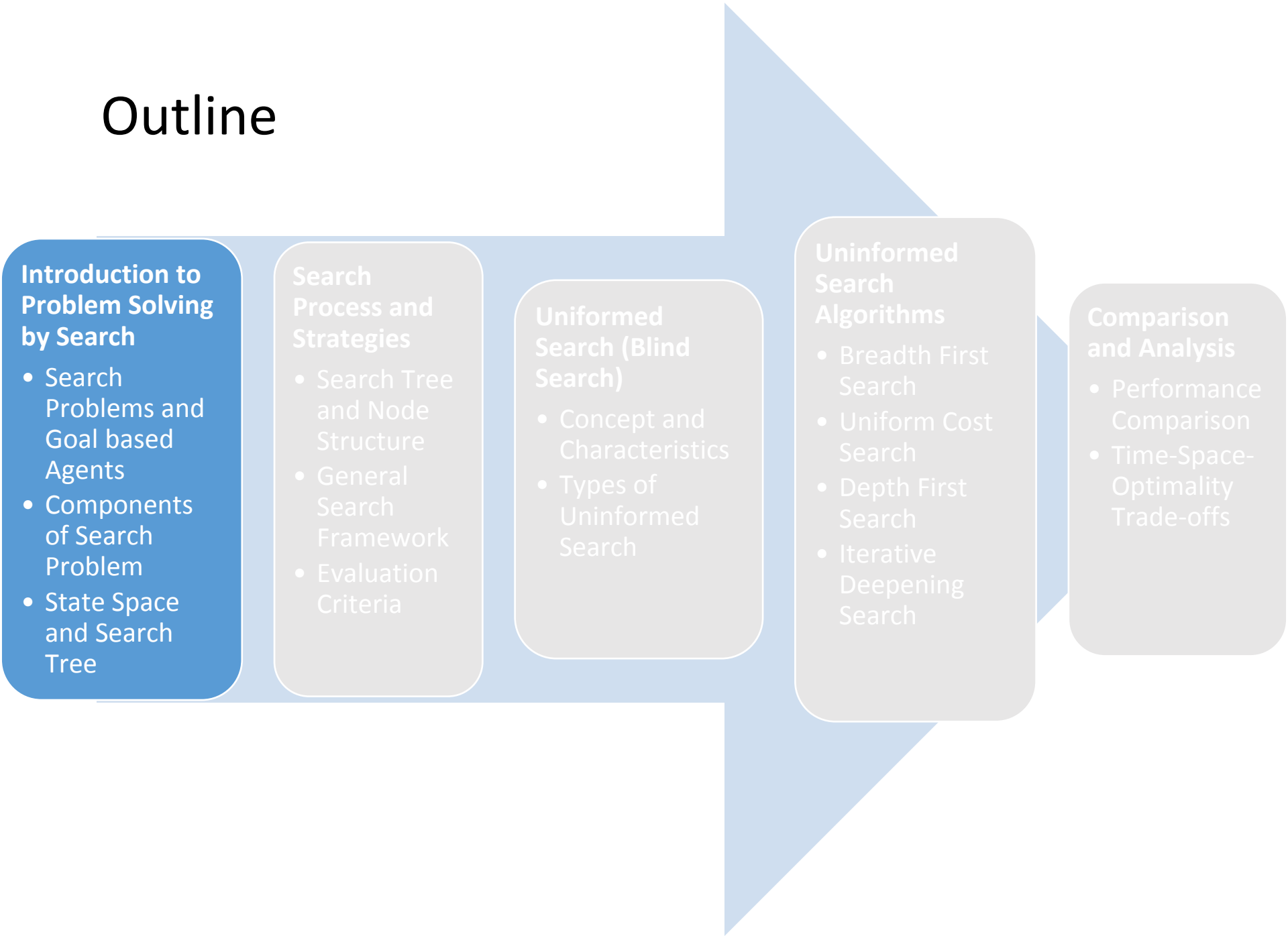
Noushin Saba

Lecture 4
Problem Solving by Searching: Uninformed Search Algorithms

Outline

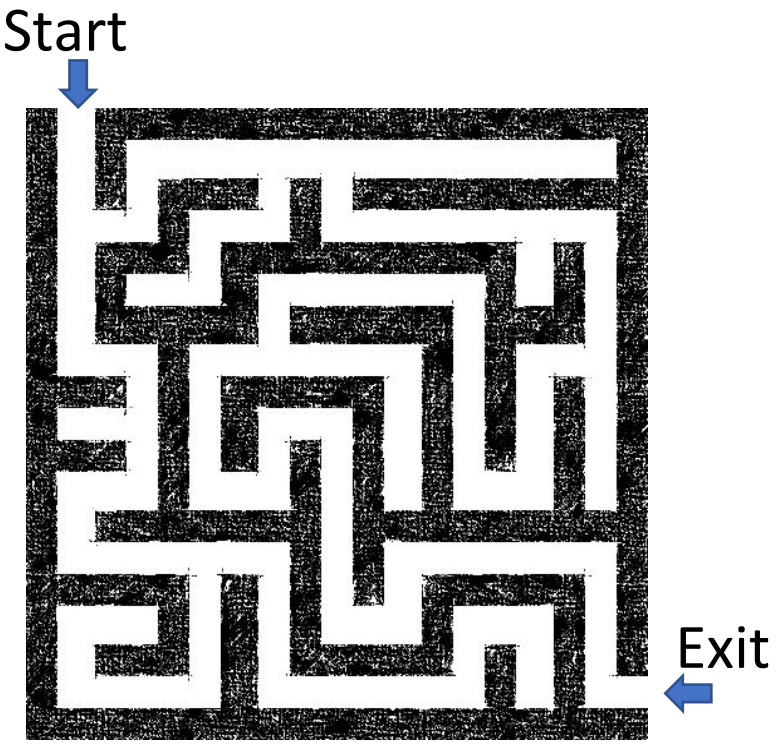

Introduction to Problem Solving by Search
• Search Problems and Goal based Agents
• Components of Search Problem
• State Space and Search Tree
Search Process and Strategies
• Search Tree and Node Structure
• General Search Framework
• Evaluation Criteria
Uniformed Search (Blind Search)
• Concept and Characteristics
• Types of Uninformed Search
Uninformed Search Algorithms
• Breadth First Search
• Uniform Cost Search
• Depth First Search
• Iterative Deepening Search
Comparison and Analysis
• Performance Comparison
• Time-Space-Optimality Trade-offs

Outline


Introduction to Problem Solving by Search
• Search Problems and Goal based Agents
• Components of Search Problem
• State Space and Search Tree
Search Process and Strategies
• Search Tree and Node Structure
• General Search Framework
• Evaluation Criteria
Uniformed Search (Blind Search)
• Concept and Characteristics
• Types of Uninformed Search
Uninformed Search Algorithms
• Breadth First Search
• Uniform Cost Search
• Depth First Search
• Iterative Deepening Search
Comparison and Analysis
• Performance Comparison
• Time-Space-Optimality Trade-offs

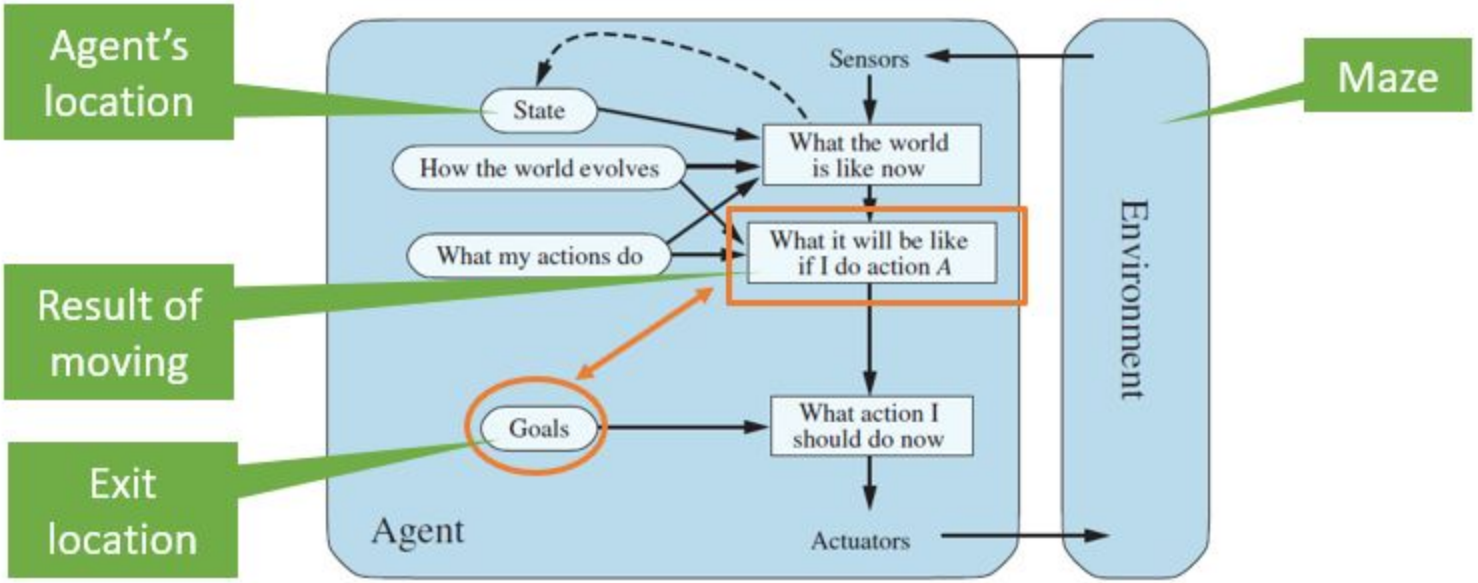
Problem Solving by Search

- Goal-based agents choose actions to achieve a specific goal.
- The agent searches for a sequence of actions leading from the initial state to the goal state.
- Environment is assumed to be:
  - Known (transition model available)
  - Fully observable (agent knows current state)
  - Deterministic (actions have predictable outcomes)
- Search = Planning before acting



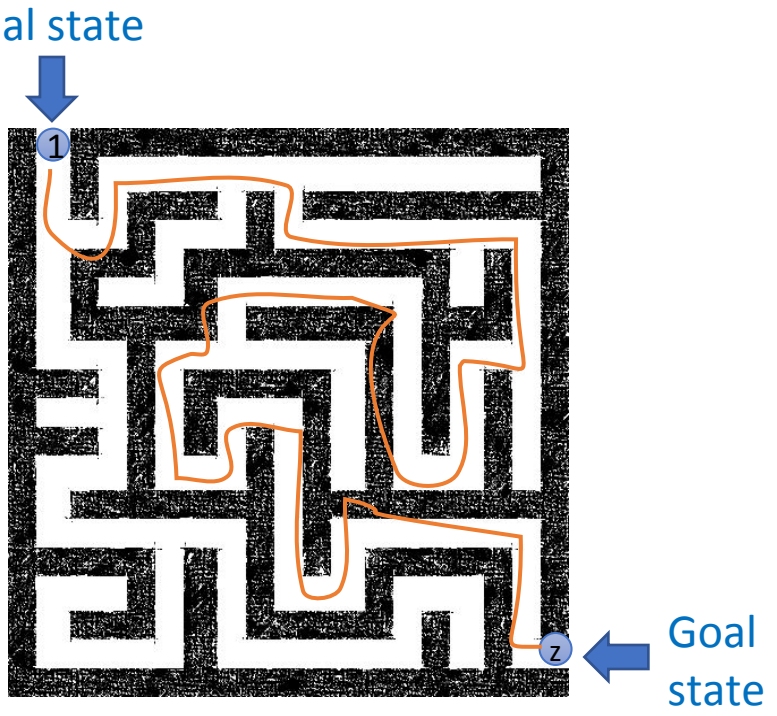
Goal-Based Agents

- The agent has the task to reach a defined **goal state**.
- The agent needs to move towards the goal. It can use **search algorithms** to plan actions that lead to the goal.
- The performance measure is typically the cost to reach the goal.
- Agent operates in a **discrete state space**.



Search Problems

- For now, we consider only a discrete environment using an **atomic state representation** (states are just labeled 1, 2, 3, …).
- The **state space** is the set of all possible states of the environment and some states are marked as **goal states**.
- The **optimal solution** is the sequence of actions (or equivalently a sequence of states) that gives the lowest path cost for reaching the goal.

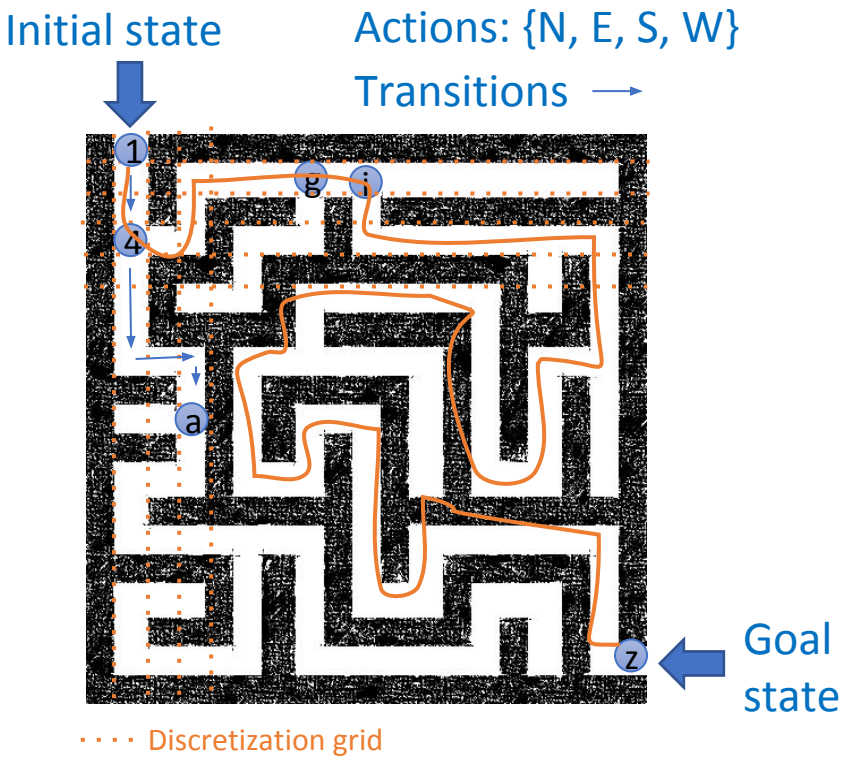


Phases:

1) **Search/Planning**: the process of looking for the **sequence of actions** that reaches a goal state. Requires that the agent knows what happens when it moves!
2) **Execution**: Once the agent begins executing the search solution in a deterministic, known environment, it can ignore its percepts (**open-loop system**).

Components of a Search problem

- **Initial state:** state description
- **Actions:** set of possible actions A
- **Transition model:** a function that defines the new state resulting from performing an action in the current state
- **Goal state:** state description
- **Path cost:** the sum of *step costs*



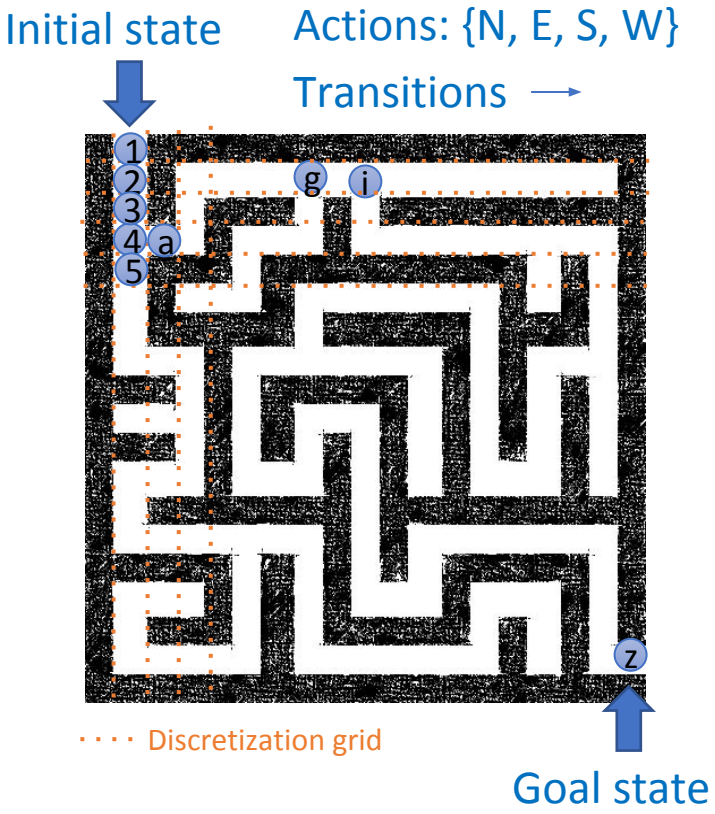
Important: The **state space** is typically too large to be enumerated or it is continuous. Therefore, the problem is defined by initial state, actions and the transition model and not the set of all possible states.

State Space and Search Tree

- **State space**: all possible configurations the agent can be in.
- **Search tree**: tree built during search; nodes represent states, edges represent actions.
- **Root**: initial state
- **Goal nodes**: satisfy goal test
- **Solution**: path from root to a goal node
- **Optimal solution**: path with minimum cost.

Transition Function and Available Actions

Original Description



- As an action schema:
  $Action(go(dir))$
  PRECOND: no wall in direction dir
  EFFECT: change the agent's location according to dir
- As a function:
  $f: S \times A \rightarrow S$ or $s' = result(a, s)$

Function implemented as a table representing the state space as a graph.

1	S	2
2	N	1
2	S	3
…	…	…
4	E	a
4	S	5
4	N	3
…	…	…

- Available actions in a state come from the transition function. E.g.,
  $actions(4) = \{E, S, N\}$

Note: Known and deterministic is a property of the transition function!

Example: Romania Vacation

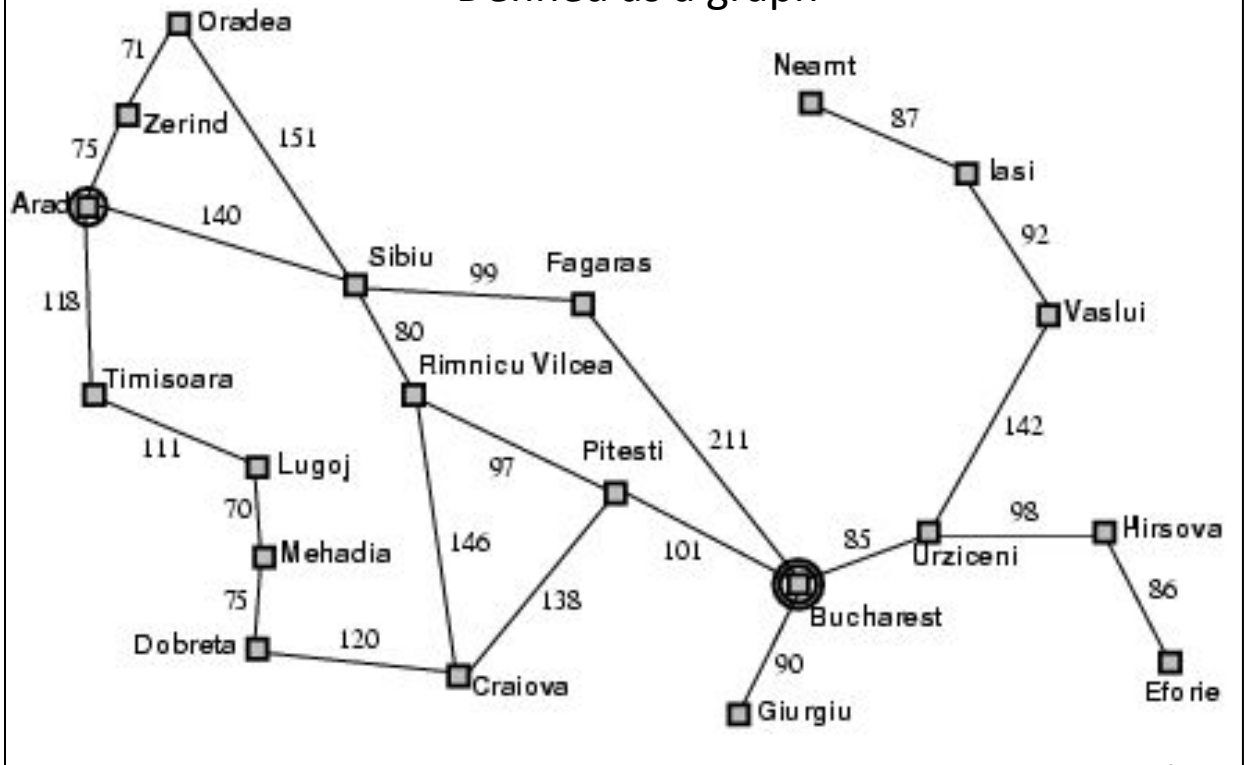
- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

Original Description

- **Initial state:** Arad
- **Actions:** Drive from one city to another.
- **Transition model and states:** If you go from city A to city B, you end up in city B.
- **Goal state:** Bucharest
- **Path cost:** Sum of edge costs.

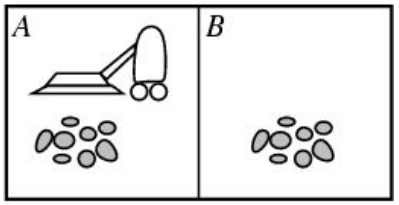
State Space/Transition model

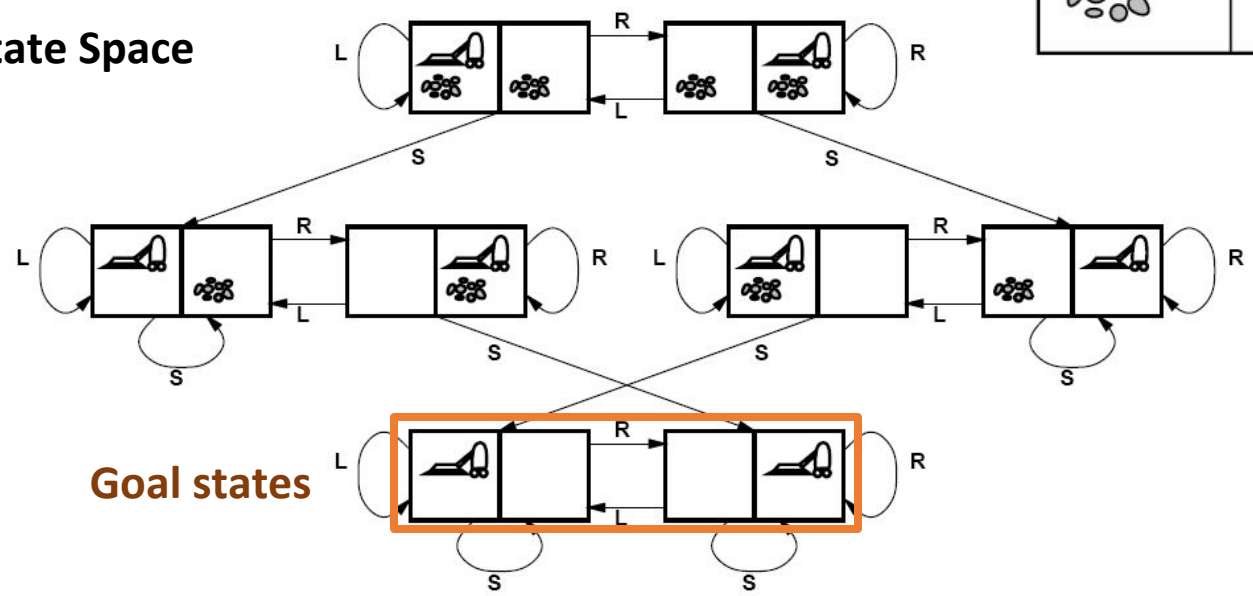
Defined as a graph



Distance in miles

Example: Vacuum world



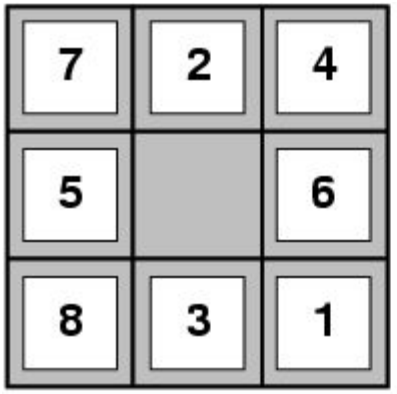


- **Initial State:** Defined by agent location and dirt location.
- **Actions:** Left, right, suck
- **Transition model:** Clean a location or move.
- **Goal state:** All locations are clean.
- **Path cost:** E.g., number if actions

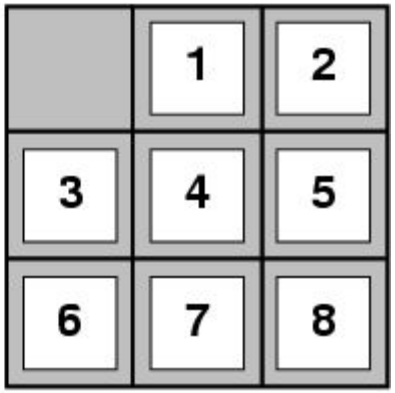
There are 8 possible atomic states of the system.
Why is the number of states for n possible locations $n(2^n)$?

Example: Sliding-tile puzzle

- **Initial State:** A given configuration.
- **Actions:** Move blank left, right, up, down
- **Transition model:** Move a tile
- **Goal state:** Tiles are arranged empty and 1-8 in order
- **Path cost:** 1 per tile move.



Start State



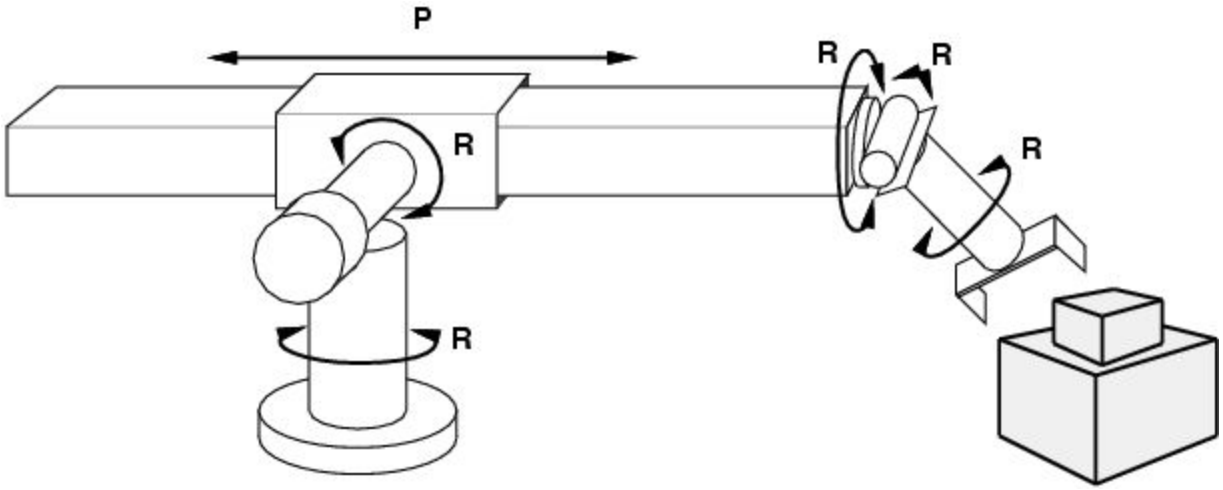
Goal State

State space size

Each state describes the location of each tile (including the empty one). ½ of the permutations are unreachable.

- 8-puzzle: $9!/2 = 181{,}440$ states
- 15-puzzle: $16!/2 \approx 10^{13}$ states
- 24-puzzle: $25!/2 \approx 10^{25}$ states

Example: Robot motion planning



- **Initial State:** Current arm position.
- **States:** Real-valued coordinates of robot joint angles.
- **Actions:** **Continuous** motions of robot joints.
- **Goal state:** Desired final configuration (e.g., object is grasped).
- **Path cost:** Time to execute, smoothness of path, etc.

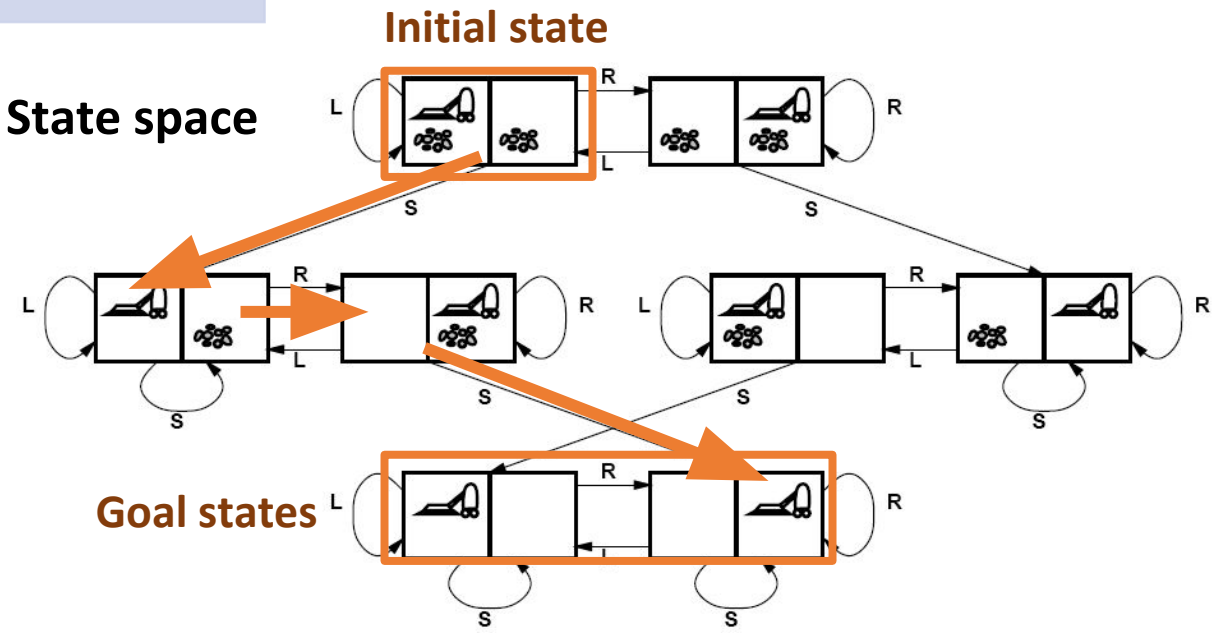
Solving search problems

Given a search problem definition

- Initial state
- Actions
- Transition model
- Goal state
- Path cost

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

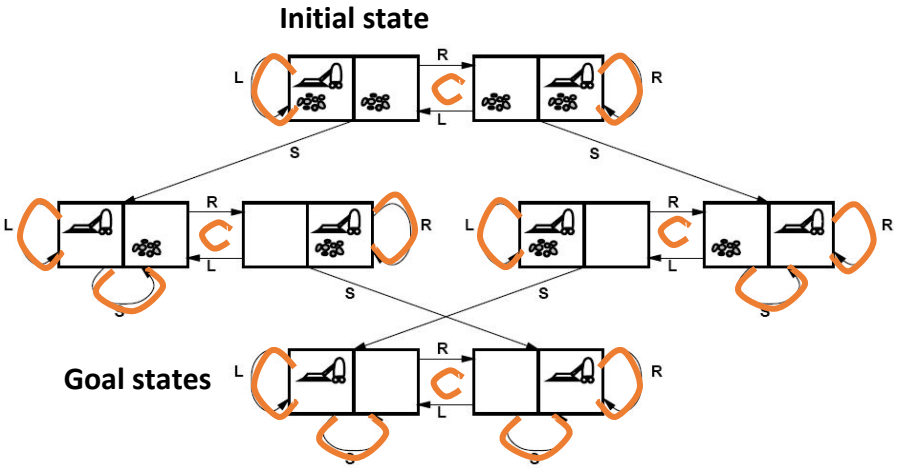
Construct a search tree for the state space graph!



Issue: Transition Model is Not a Tree! It can have Redundant Paths

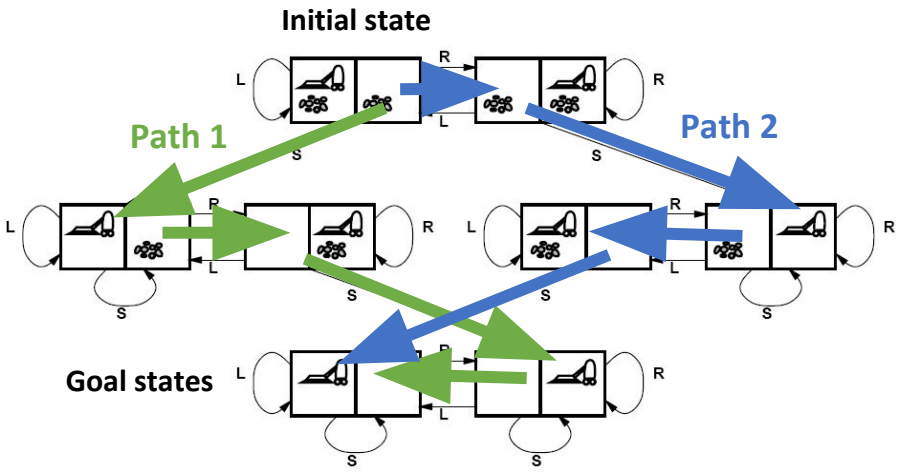
Cycles

Return to the same state. The search tree will create a new node!



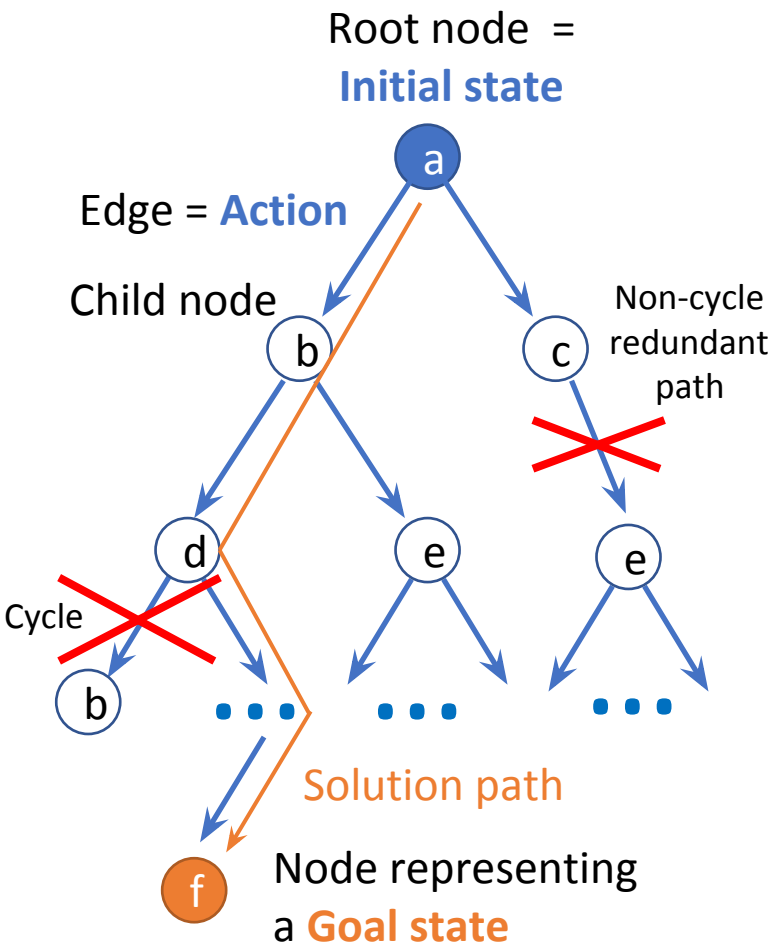
Non-cycle redundant paths

Multiple paths to get to the same state



Search tree

- Superimpose a “what if” tree of possible actions and outcomes (states) on the state space graph.
- The **Root node** represents the initial stare.
- An action child node is reached by an **edge** representing an action. The corresponding state is defined by the transition model.
- Trees cannot have **cycles (loops)** or **multiple paths to the same state.** These are called redundant paths. Cycles in the search space must be broken to prevent infinite loops. Removing other redundant paths improves search efficiency.
- A **path** through the tree corresponds to a sequence of actions (states).
- A **solution** is a path ending in a node representing a goal state.
- **Nodes vs. states:** Each tree node represents a state of the system. If redundant path cannot be prevented then state can be represented by multiple nodes.



Differences between typical Tree search and AI search

Typical tree search

- Assumes a given tree that fits in memory.

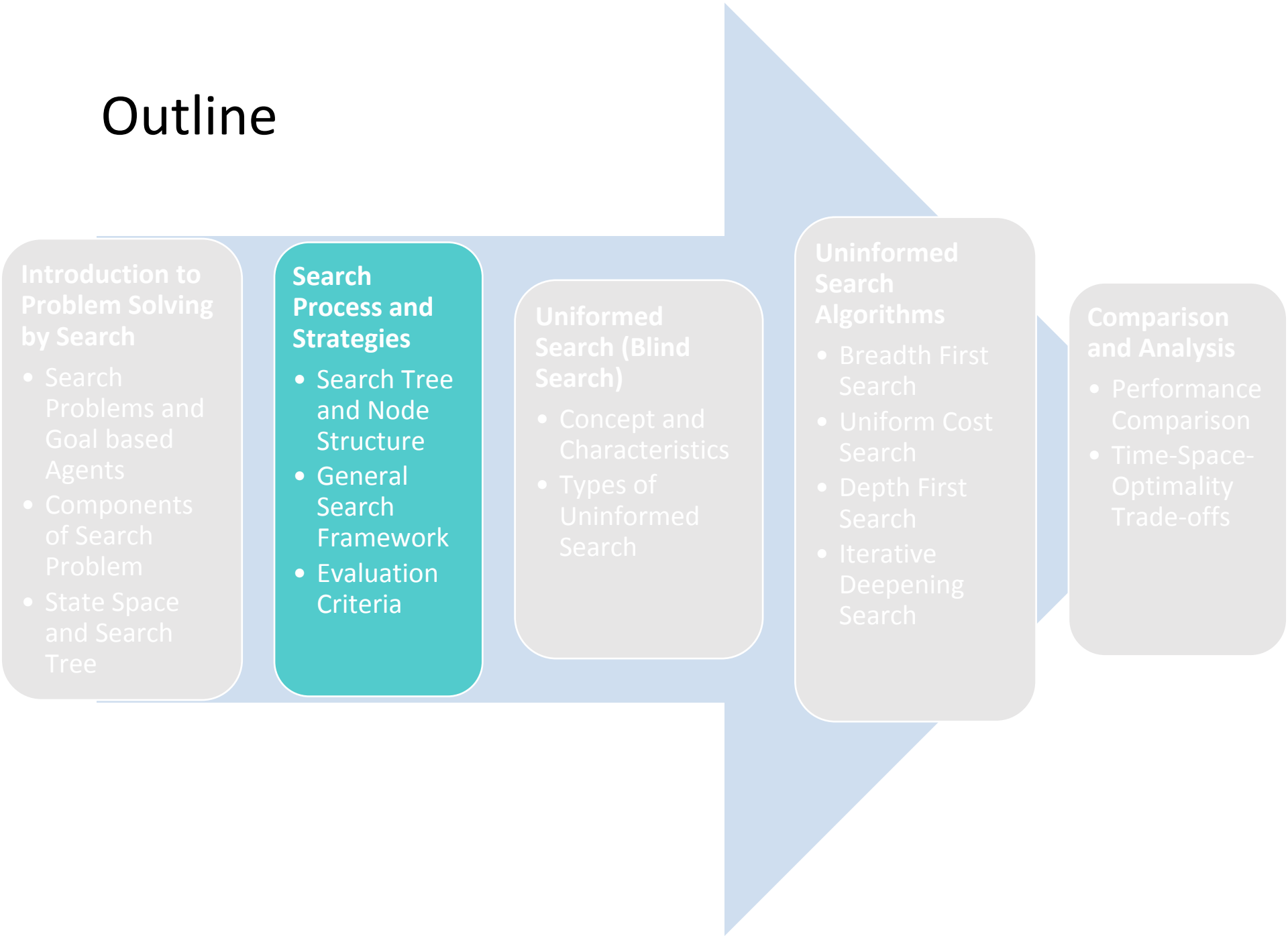
- Trees have by construction no cycles or redundant paths.

AI tree/graph search

- The search tree is too large to fit into **memory**.
  a. **Builds parts of the tree** from the initial state using the transition function representing the graph.
  b. **Memory management** is very important.

- The search space is typically a very large and complicated graph. Memory-efficient **cycle checking** is very important to avoid infinite loops or minimize searching parts of the search space multiple times.
- Checking redundant paths often requires too much memory and we accept searching the same part multiple times.

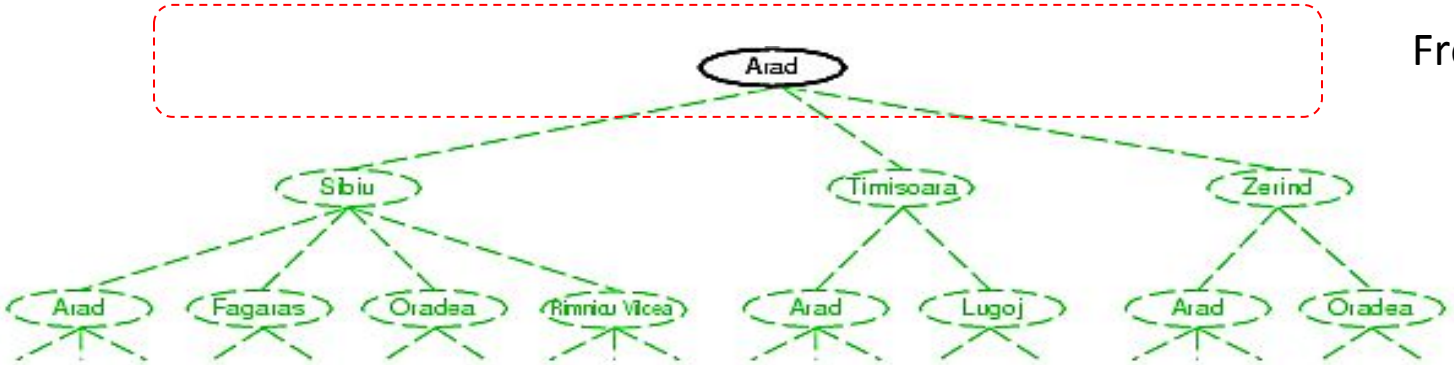
Outline


Introduction to Problem Solving by Search
• Search Problems and Goal based Agents
• Components of Search Problem
• State Space and Search Tree
Search Process and Strategies
• Search Tree and Node Structure
• General Search Framework
• Evaluation Criteria
Uniformed Search (Blind Search)
• Concept and Characteristics
• Types of Uninformed Search
Uninformed Search Algorithms
• Breadth First Search
• Uniform Cost Search
• Depth First Search
• Iterative Deepening Search
Comparison and Analysis
• Performance Comparison
• Time-Space-Optimality Trade-offs

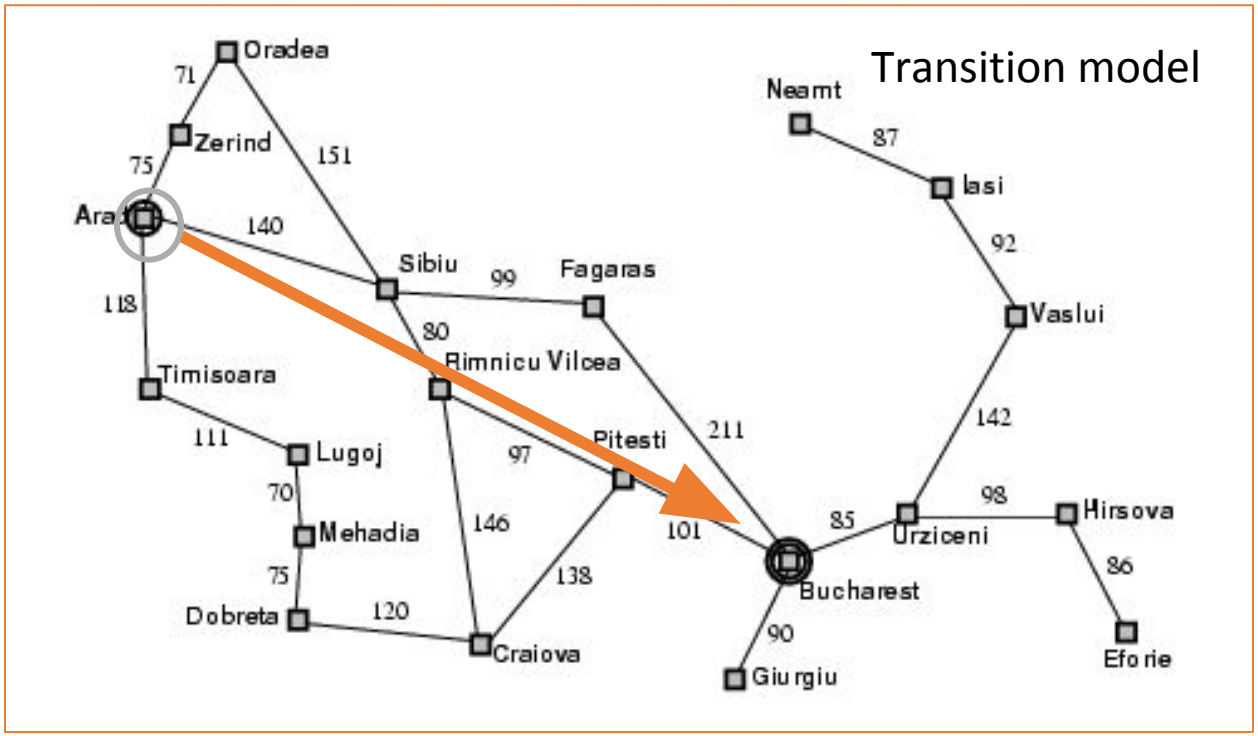
Tree Search Algorithm Outline

1. Initialize the **frontier** (set of unexplored know nodes) using the **starting state/root node.**
2. While the frontier is not empty:
   a) Choose next 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 (i.e., apply all possible actions to the transition model) and add its children nodes representing the newly reached states to the frontier.

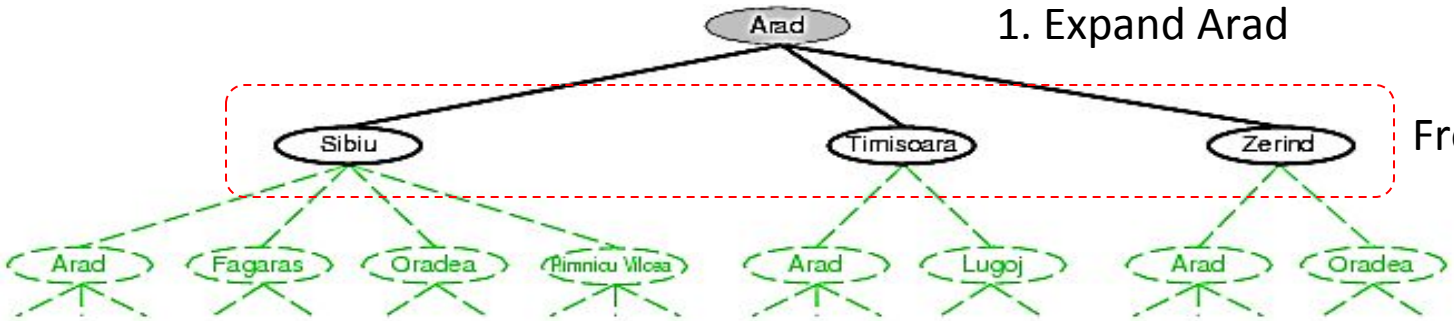
Tree search example



Frontier

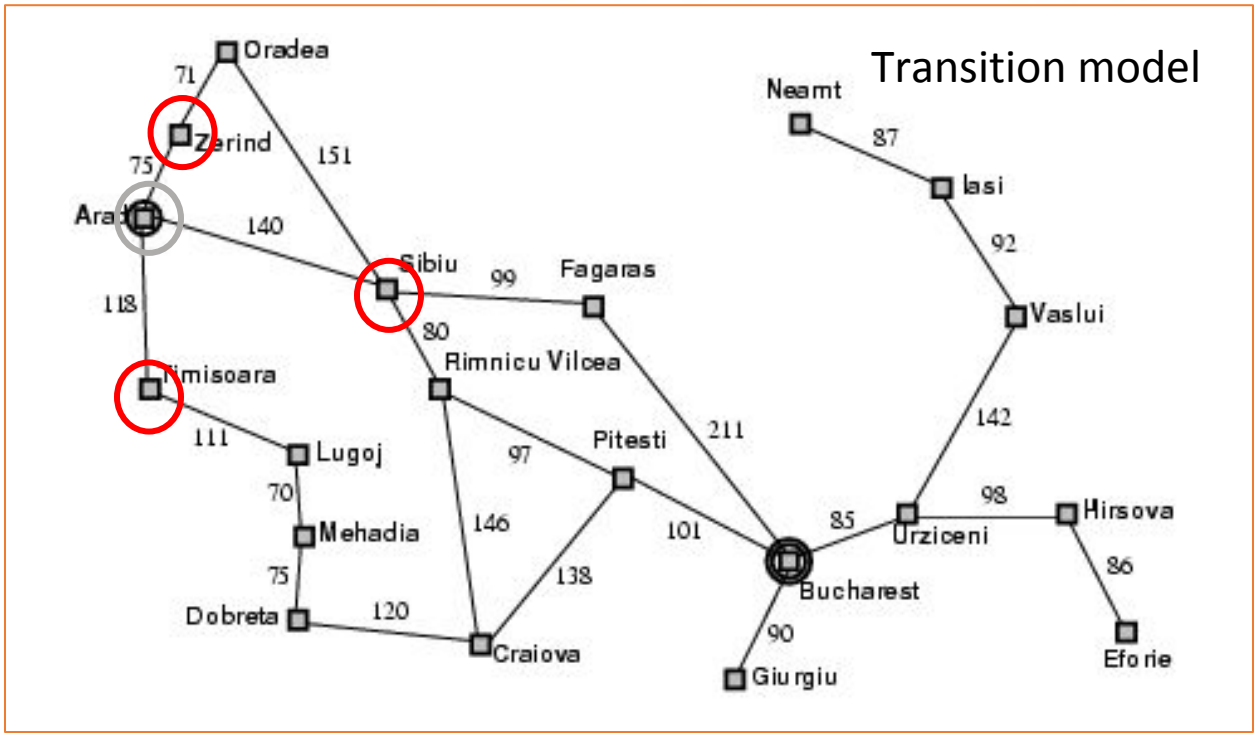


Tree search example



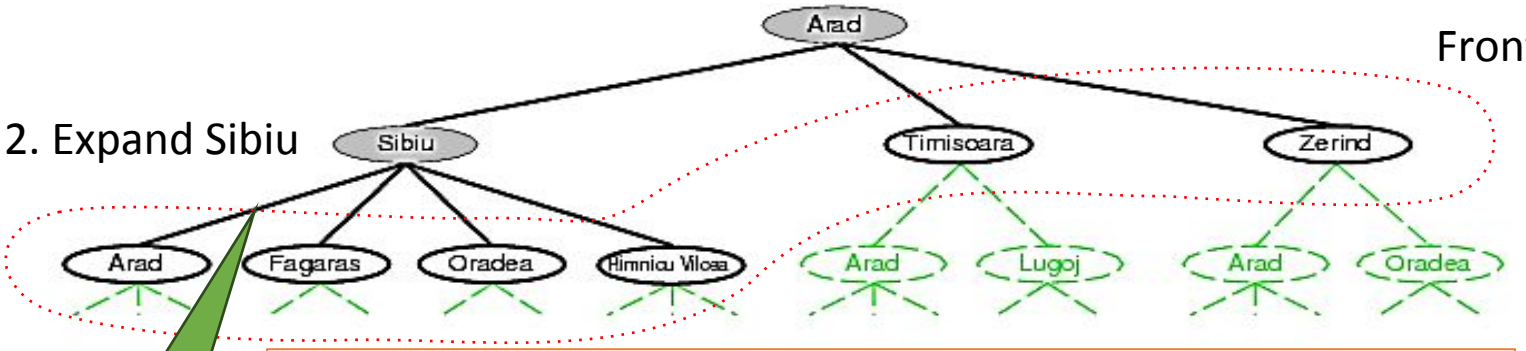
1. Expand Arad

Frontier



Transition model

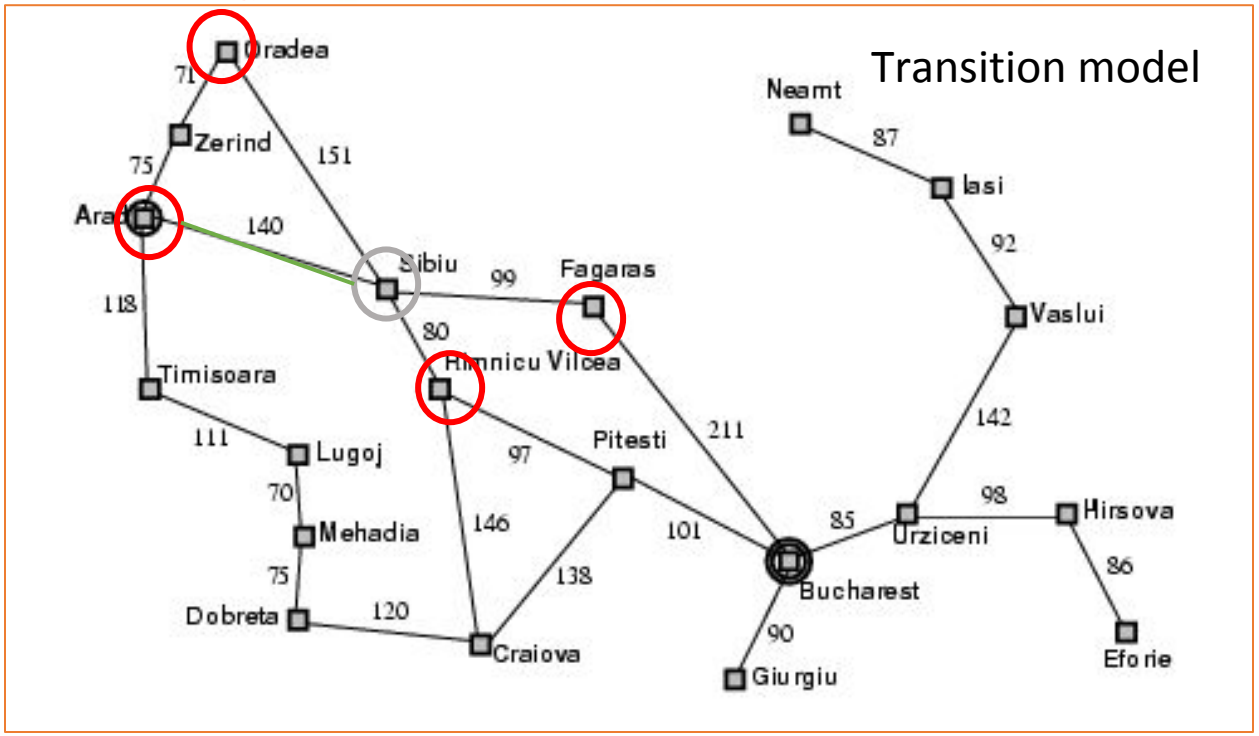
Tree search example



Frontier

2. Expand Sibiu

Example of a cycle



We could have also expanded Timisoara or Zerind!

Search Node Structure

- **Search algorithms** require a data structure to keep track of the search tree.
- A node in the tree is represented by a data structure with four components:
  - **STATE:** the state to which the node corresponds
  - **PARENT:** the node in the tree that generated this node
  - **ACTION:** the action that was applied to the parent's state to generate this node
  - **PATH-COST**: the total cost of the path from the initial state to this node
- Queue : Data structure to store frontier
  - Operations on a frontier are:
    - **IS-EMPTY(frontier)** returns true only if there are no nodes in the frontier.
    - **POP(frontier)** removes the top node from the frontier and returns it.
    - **TOP(frontier)** returns (but does not remove) the top node of the frontier.
    - **ADD(node, frontier)** inserts node into its proper place in the queue

Search Strategies

- A **search strategy** is defined by picking the **order of node expansion**.
- Strategies are evaluated along the following dimensions:
  - **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?
  - **Space complexity:** how much memory does it need?
- Worst case time and space complexity are measured in terms of the **size of the state space *n*** (= number of nodes in the search tree).
- Time and space complexity often depend on:
  - **b**: branching factor (number of successors per state)
  - **d**: depth of the shallowest goal node
  - **m**: maximum depth of the search tree

Outline

Introduction to Problem Solving by Search

- Search Problems and Goal based Agents
- Components of Search Problem
- State Space and Search Tree

Search Process and Strategies

- Search Tree and Node Structure
- General Search Framework
- Evaluation Criteria

Uniformed Search (Blind Search)

- Concept and Characteristics
- Types of Uninformed Search

Uninformed Search Algorithms

- Breadth First Search
- Uniform Cost Search
- Depth First Search
- Iterative Deepening Search

Comparison and Analysis

- Performance Comparison
- Time-Space-Optimality Trade-offs

What is Uninformed (Blind) Search?

- The agent has **no additional information** about which state leads to the goal.
- Search proceeds **purely based on the problem definition**.
- All nodes are treated **equally** — no heuristic or estimate of goal distance.
- Also called **blind search**, as it explores without "guidance."
- **Key Idea** ? "Search first, then evaluate — not guided by knowledge about the goal."

Characteristics of Uninformed Search

- Uses **only the problem definition** (initial state, actions, goal test, path cost).
- **Systematic exploration** of the search space.
- No domain-specific knowledge.
- May explore **many unnecessary paths**.
- Guarantees a solution **only if search space is finite and well-defined.**
- **Analogy** ? Like exploring a city **without a map** — you try every possible street until you find your destination.

Outline

Introduction to Problem Solving by Search

- Search Problems and Goal based Agents
- Components of Search Problem
- State Space and Search Tree

Search Process and Strategies

- Search Tree and Node Structure
- General Search Framework
- Evaluation Criteria

Uniformed Search (Blind Search)

- Concept and Characteristics
- Types of Uninformed Search

Uninformed Search Algorithms

- Breadth First Search
- Uniform Cost Search
- Depth First Search
- Iterative Deepening Search

Comparison and Analysis

- Performance Comparison
- Time-Space-Optimality Trade-offs

Types of Uninformed Search Algorithms

- **Breadth-First Search (BFS)** – Explores level by level (shallowest node first).
- **Uniform Cost Search (UCS)** – Expands the least-cost node first.
- **Depth-First Search (DFS)** – Expands deepest unexpanded node first.
- **Iterative Deepening Search (IDS)** – Combines DFS's memory efficiency with BFS's completeness.
- Each algorithm follows the **general search framework**, differing only in **node selection strategy**.

Breadth-first search (BFS)

Expansion rule: Expand shallowest unexpanded node in the frontier (=**FIFO**).

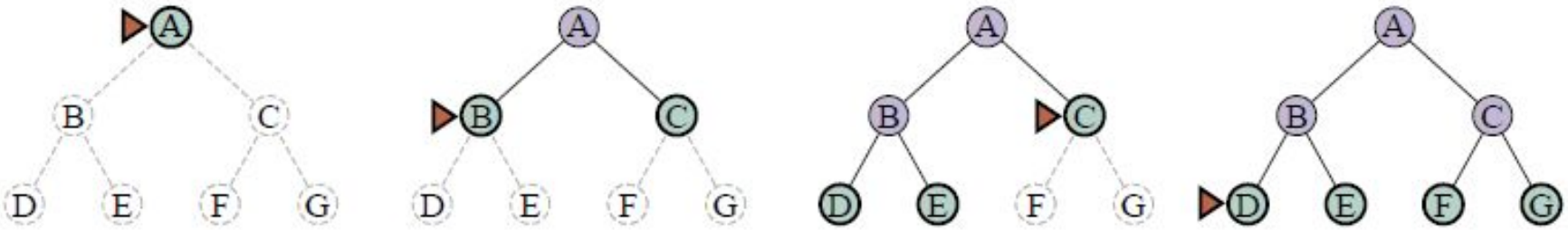


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Data Structures

- **Frontier** data structure: holds references to the green nodes (green) and is implemented as a FIFO **queue**.
- **Reached** data structure: holds references to all visited nodes (gray and green) and is used to prevent visiting nodes more than once (redundant path checking).
- Builds a **tree** with links between parent and child.

Implementation: BFS

```
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
```

Expand adds the next level below node to the frontier.

reached makes sure we do not visit nodes twice (e.g., in a cycle or other redundant path). Fast lookup is important.

Implementation: Expanding the search tree

- AI tree search creates the search tree while searching.
- The EXPAND function tries all available actions in the current node using the transition function (RESULTS). It returns a list of new nodes for the frontier.

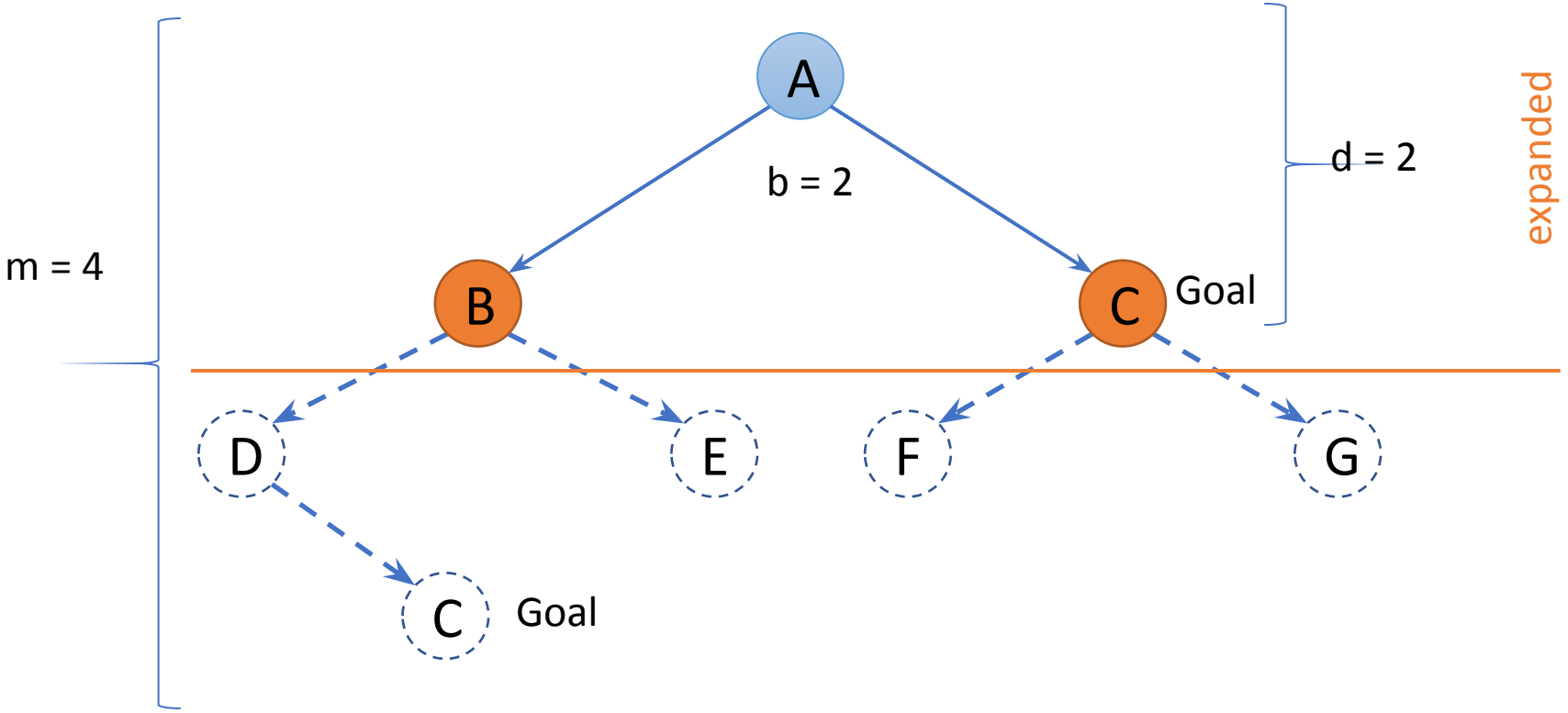
```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Transition function

Node structure for the search tree. Yield can also be implemented by returning a list of Nodes.

Time and Space complexity
Breadth-first search

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor



$\mathcal{O}(b^d)$ - all paths to the depth of the goal are expanded

Properties of Breadth-first search

- **Complete?**

  Yes (if branching factor b finite)

- **Optimal?**

  Yes – if cost is the same per step (action).

- **Time?**

  expands all nodes up to depth d $\rightarrow O(b^d)$

- **Space?**

  must store all frontier nodes $\rightarrow O(b^d)$

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

Note:

- Very **memory-intensive** for large *d*.
- Used when **path cost = depth** (e.g., shortest path in unweighted graph).
- Foundation for more advanced algorithms (Uniform Cost, IDS).

Uniform-cost search

- **Expansion rule**: Expand node in the frontier with **the least path cost** from the initial state.
- Implementation: **best-first search** where the frontier is a **priority queue** ordered by lower $f(n)$ = **path cost** (cost of all actions starting from the initial state).
- Equivalent to **Dijkstra's shortest path algorithm**.
- Suitable when **step costs vary** between actions.
- Ensures the **optimal solution** when all costs ≥ 0.

Implementation: Best-First Search Strategy

```
function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

The order for expanding the frontier is determined by *f(n)* = path cost from the initial state to node *n*.

This check is the difference to BFS! It visits a node again if it can be reached by a better (cheaper) path.

See BFS for function EXPAND.

Properties of Uniform-cost search

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

- **Complete?**

  Yes, if all step cost is greater than some small positive constant $\varepsilon > 0$

- **Optimal?**

  Yes – nodes expanded in increasing order of path cost

- **Time?**

  Number of nodes with path cost ≤ cost of optimal solution (C^*) is $O(b^{1+C^*/\varepsilon})$.

- **Space?**

  frontier may grow large ⍰ $O(b^{1+C^*/\varepsilon})$

Note:

- More **efficient than BFS** when path costs vary.
- Still suffers from **high memory usage**.
- Foundation for **A*** algorithm (informed search).

Depth-first search (DFS)

- **Expansion rule**: Expand deepest unexpanded node in the frontier (last added).
- **Frontier**: **stack** (LIFO) **(Last-In, First-Out)** structure
- When a dead end is reached, the algorithm **backtracks**.
- Explores **one complete path** before moving to the next.
- Simple to implement and **requires little memory**.

Depth-first search (DFS)

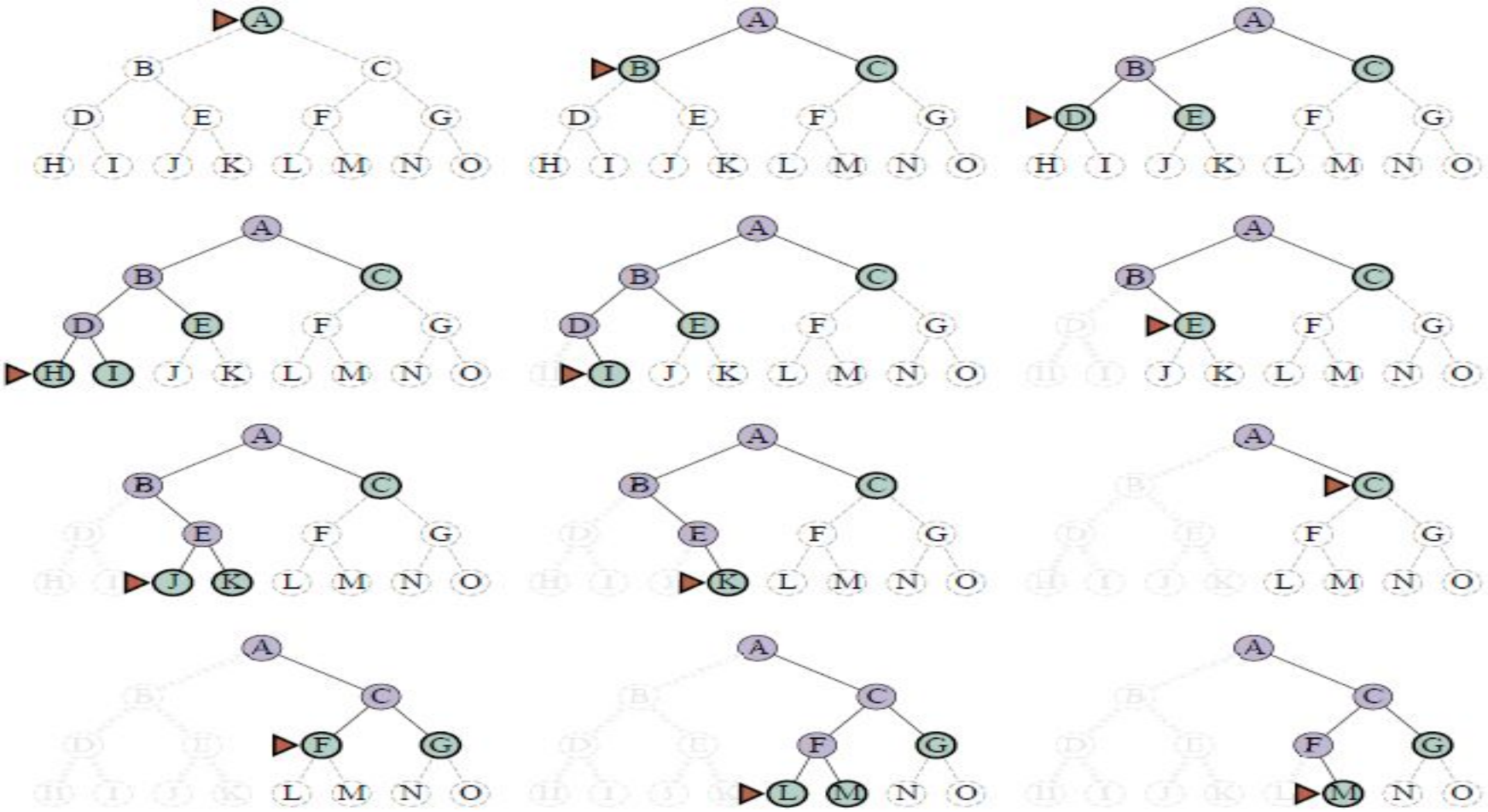


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint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

Implementation: DFS

- DFS could be implemented like BFS/Best-first search and just taking the last element from the frontier (LIFO).
- However, to reduce the space complexity to $O(bm)$, the reached data structure needs to be removed! Options:
  - ~~Recursive implementation~~ (cycle checking is a problem leading to infinite loops)
  - Iterative implementation: Build tree and abandoned branches are removed from memory. Cycle checking is only done against the current path. This is similar to Backtracking search.

DFS uses $\ell = \infty$

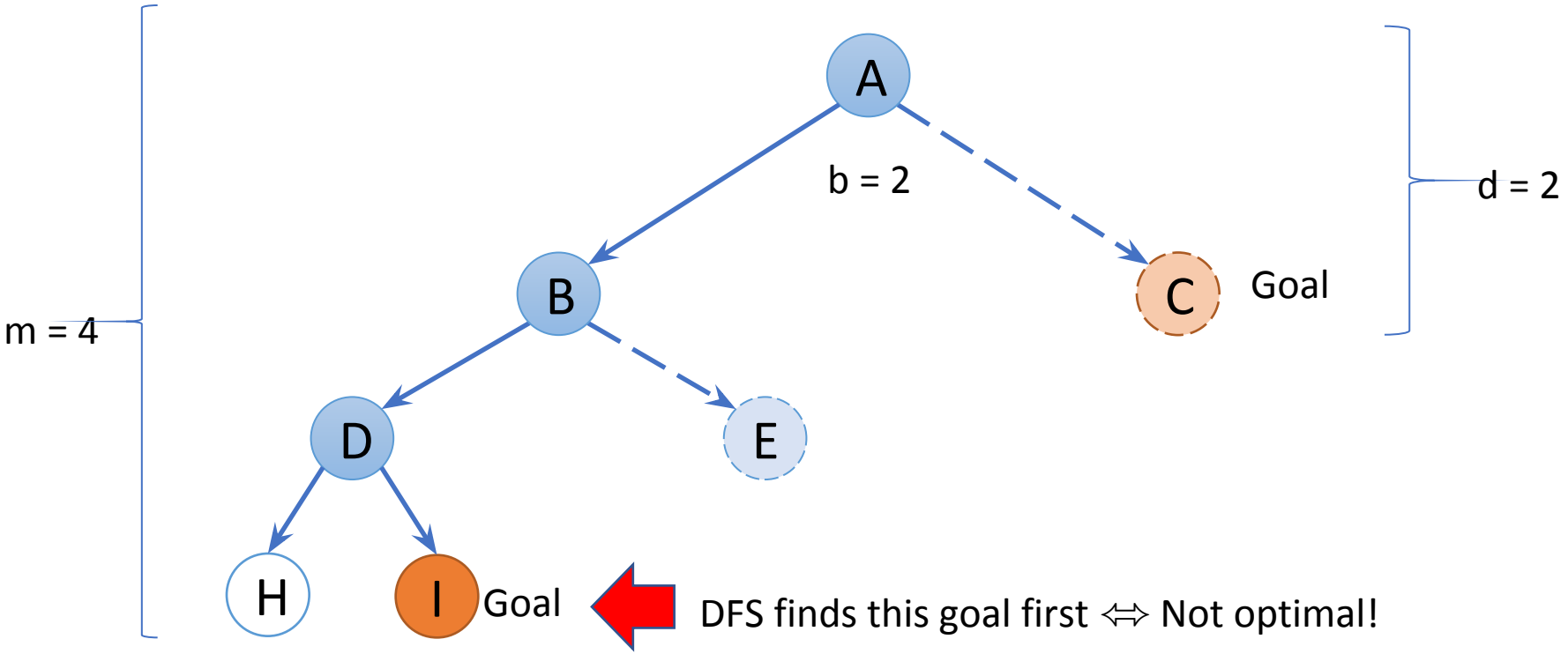
```
function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

If we only keep the current path from the root to the current node in memory, then we can only check against that path to prevent cycles, but we cannot prevent other redundant paths. We also need to make sure the frontier does not contain the same state more then once!

See BFS for function EXPAND.

Time and Space Complexity
Depth-first search

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor



- Time: $O(b^m)$ – worst case is expanding all paths.
- Space: $O(bm)$ - if it only stores the frontier nodes and the current path.

Properties of depth-first search

- **Complete?**
  - No (infinite paths → may never terminate)
- **Optimal?**

  No (does not guarantee shortest path)
- **Time?**

  may explore all depths m → $O(b^m)$
- **Space?**

  $O(bm)$ ⇔ stores only current path + siblings

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

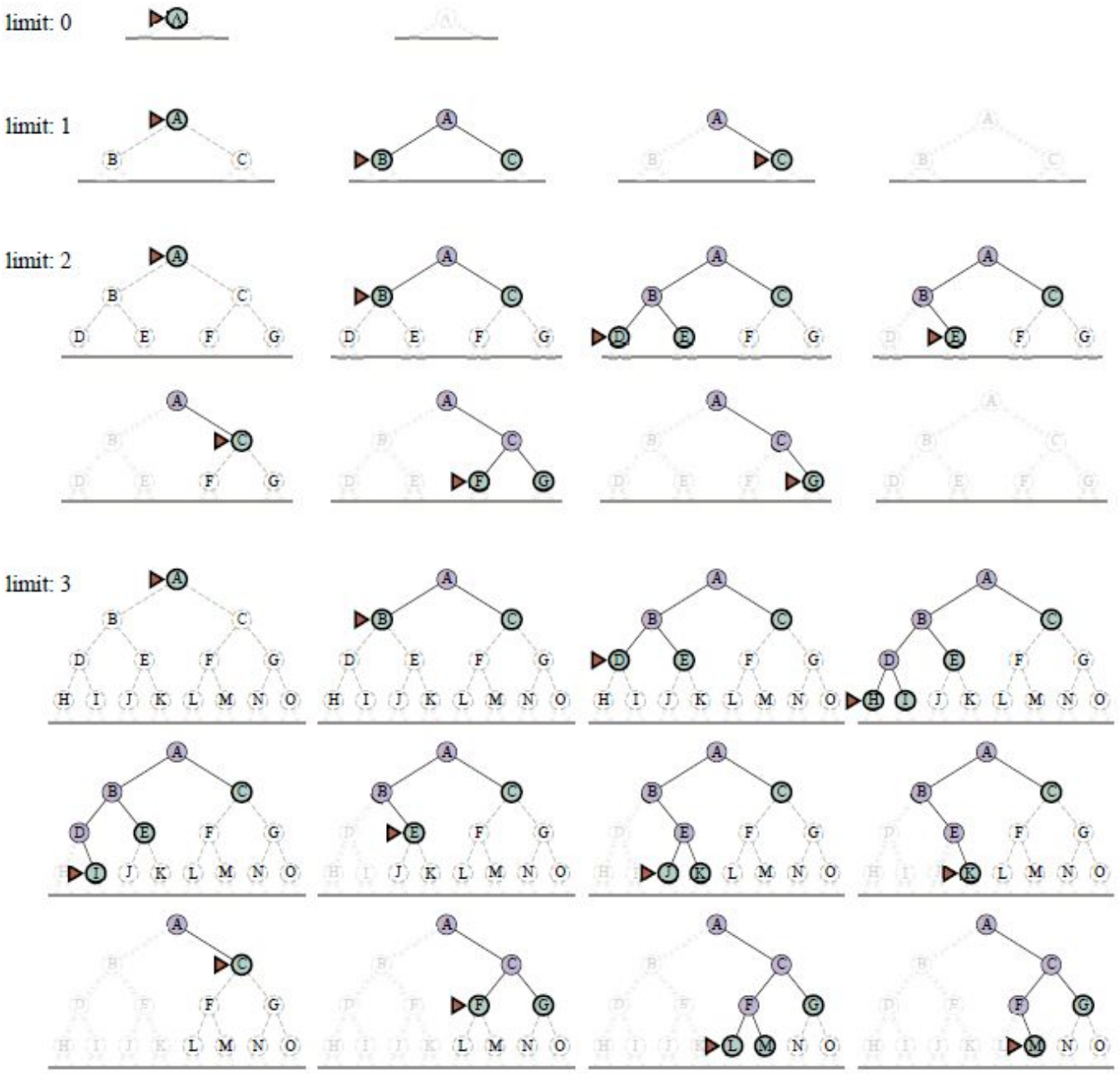
Note:

- Very **memory-efficient**, practical for large but finite spaces.
- **Fails** on infinite or looping paths (e.g., recursive state spaces).
- Variants like **Depth-Limited Search** and **Iterative Deepening** overcome these issues.

Iterative deepening search (IDS)

- Combines the **completeness of BFS** with the **space efficiency of DFS**.
- Repeatedly performs **Depth-Limited Search (DLS)** with increasing depth limit *l*.
- Each iteration deepens the search one level further.
- The first time the goal is found → it is the **shallowest (optimal)** one.
- Useful when **search depth is unknown**.

Iterative deepening search (IDS)



Implementation: IDS

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

```
function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

See BFS for function EXPAND.

Properties of iterative deepening search

- **Complete?**

  Yes (if branching factor *b* finite)

- **Optimal?**

  Yes, if step cost = 1

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

- **Time?**

  Consists of rebuilding trees up to d times

  $d\,b^1 + (d-1)b^2 + \ldots + 1b^d = O(b^d)$ ⇔ Slower than BFS, but the same complexity!

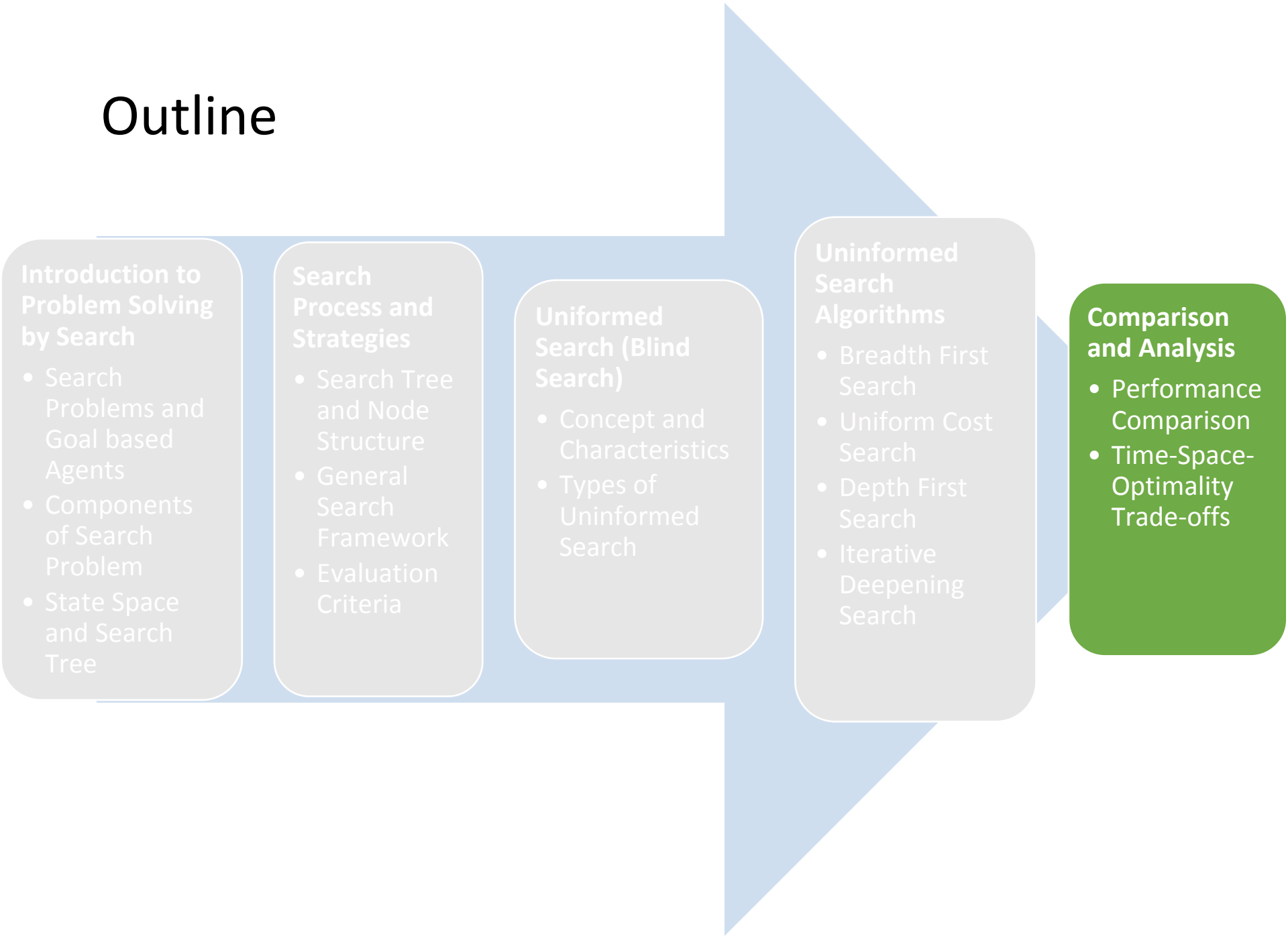
- **Space?**

  ***O*(*bd*) ⇔ linear space. Even less than DFS since m<=d.** Cycles need to be handled by the depth-limited DFS implementation.

Note:

- **Preferred uninformed search method** when the search space is large and depth unknown.
- Slight **repetition** of nodes, but overhead is small.
- Used as a **basis for A*** variants and **real-world problem solvers**.

Outline


Introduction to Problem Solving by Search
• Search Problems and Goal based Agents
• Components of Search Problem
• State Space and Search Tree
Search Process and Strategies
• Search Tree and Node Structure
• General Search Framework
• Evaluation Criteria
Uniformed Search (Blind Search)
• Concept and Characteristics
• Types of Uninformed Search
Uninformed Search Algorithms
• Breadth First Search
• Uniform Cost Search
• Depth First Search
• Iterative Deepening Search
Comparison and Analysis
• Performance Comparison
• Time-Space-Optimality Trade-offs

Comparison of Uninformed Search Algorithms

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor
C*: cost of optimal solution

Algorithm	Completeness	Optimality	Time Complexity	Space Complexity
Breadth-First Search (BFS)	Yes	Yes (equal costs)	O(b^d)	O(b^d)
Uniform Cost Search (UCS)	Yes	Yes	O(b^(1 + ⌊C*/ϵ⌋))	O(b^(1 + ⌊C*/ϵ⌋))
Depth-First Search (DFS)	No	No	O(b^m)	O(b × m)
Iterative Deepening Search (IDS)	Yes	Yes (equal costs)	O(b^d)	O(b × d)

Trade-offs Among Uninformed Searches

- **BFS**: Simple & optimal for unit costs → Memory heavy
- **UCS**: Best for varying costs → Slower but optimal
- **DFS**: Memory efficient but not complete; may loop indefinitely
- **IDS**: Best overall balance → Complete, optimal, space-efficient

- ***"IDS combines the space efficiency of DFS with the completeness and optimality of BFS."***

Summary

- We explored
  - **Problem Solving by Search:** How intelligent agents plan sequences of actions to reach a goal using state-space search.
  - **Search Representation:** Defined problems through **states, actions, transition models, goals, and path costs**, visualized as a **search tree**.
  - **Search Strategies:** Examined how algorithms expand nodes and evaluate performance via **completeness, optimality, time, and space**.
  - **Uninformed Search Methods:** Explored **BFS, UCS, DFS, and IDS** — algorithms that search **without heuristic knowledge**, differing in order of node expansion.
- **Trend:**
  These algorithms form the **foundation of goal-based AI**, later enhanced by **informed (heuristic) searches** like A*, which introduce *knowledge-guided exploration*.
- *"Uninformed search teaches us how to explore; informed search teaches us where to explore — together, they define the essence of intelligent problem solving."*