Solving Problems By Searching

Declaration:

This PPT content has been prepared from the 4th and 3rd Editions of the Book *Artificial Intelligence: A Modern Approach by Russell and Norvig. So, Dr. Jana is giving full credit to the authors of this book. The images and examples used directly or indirectly in this PPT may or may not be from this book. Read suggestions are provided to verify the content from different standard books, articles, etc.*

By Dr. Gopal Chandra Jana, Assistant Professor, DCSE, SSCSE

Course Page:

https://www.gcjana.in/courses/shardauniversity/2502/PHD650/

Topics to be Covered:

- Recap – type of Agents
- Problem-Solving Agent
- Uninformed search algorithms
- Informed search algorithms

Recap:

- The reflex agents, which base their actions on a direct mapping from states to actions. Such agents cannot operate well in environments for which this mapping would be too large to store and would take too long to learn.

- **Goal-based agents**, on the other hand, consider future actions and the desirability of their outcomes.

Problem-solving agent (goal-based agent)**:**

- Problem-solving agents use **atomic** representations.
- Goal-based agents that use more advanced **factored** or **structured** representations are usually called **planning agents.**

- Let's see a scenario that illustrates how goal-based agents in real-life tourism decisions must balance long-term enjoyment with short-term deadlines, adapting their plans dynamically while aligning with the ultimate objective.

Think of a Scenario: A Touring Agent in Manali, India

An agent is on a leisure trip in **Manali**, Himachal Pradesh. The agent's **performance measure** includes several factors:

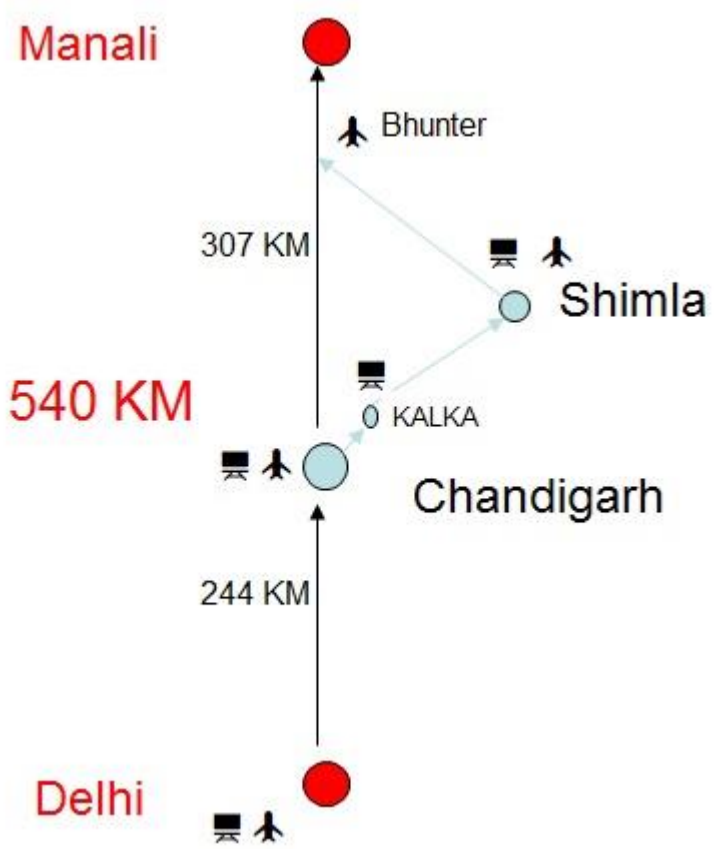
- Enjoy beautiful mountain views and improve mental well-being
- Try local Himachali food
- Learn some Hindi phrases and engage with locals
- Explore temples, waterfalls, and nearby trekking trails
- Enjoy local nightlife or cafes
- Avoid exhaustion or health issues due to altitude

Now suppose the agent has a **nonrefundable flight from Indira Gandhi International Airport, Delhi**, scheduled **the next day at 6 PM**.

Think of a Scenario: A Touring Agent in Manali, India

❑ Goal Adoption:

In this case, despite the tempting attractions in and around Manali, it becomes rational for the agent to **adopt the goal of reaching Delhi in time** to catch the flight.



Think of a Scenario: A Touring Agent in Manali, India

❑ Invalid Courses of Action (That Don't Reach Delhi on Time):

Examples of decisions that conflict with the goal and are suboptimal:

- **Deciding to take a detour to Kasol or Tirthan Valley** for extra sightseeing (adds 6–8 hours of travel)
- **Spending another night in Old Manali to enjoy a music festival** (risk of missing morning departure)
- **Taking a bus that departs late or is known for delays** (e.g., an overnight bus with uncertain arrival)
- **Choosing a scenic but long bike ride to Delhi** without time margin (risky and tiring)
- **Starting the journey to Delhi the next day morning** without accounting for roadblocks or weather delays

Think of a Scenario: A Touring Agent in Manali, India

❑ Valid Courses of Action

Actions consistent with the agent's goal:

- **Leaving Manali early in the morning** (e.g., by 6 AM) via a reliable Volvo or taxi
- **Booking a flight from Kullu to Delhi**, if available
- **Staying overnight in a midway city (e.g., Chandigarh)** to break the journey but still reach Delhi on time
- **Using a travel app to track real-time road conditions** to optimize route planning

Importance of Goals Adoption and Goal Formulation:

- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs **GOAL FORMULATION** to consider.

❖ **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem-solving.

Importance of Problem Formulation:

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

Clearly, we can say:

Problem formulation is the process of defining what actions, states, and goals an intelligent agent should consider in order to find an effective solution. It is a critical step in Artificial Intelligence (AI) and problem-solving, as it lays the foundation for how the problem will be approached and ultimately solved.

Properties of the environment:

- We assume that the environment is **observable**, so the agent always knows the current state.
- We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from.
- We will assume the environment is **known**, so the agent knows which states are reached by each action.
- Finally, we assume that the environment is **deterministic**, so each action has exactly one outcome.

Problem types

Deterministic, accessible $\Longrightarrow$ *single-state problem*

Deterministic, inaccessible $\Longrightarrow$ *multiple-state problem*

Nondeterministic, inaccessible $\Longrightarrow$ *contingency problem*

must use sensors during execution

solution is a *tree* or *policy*

often *interleave* search, execution

Unknown state space $\Longrightarrow$ *exploration problem* ("online")

Foundation for problem-solving agents:

The agent design process in AI, particularly focusing on the **Formulate–Search–Execute cycle**, which is **foundational for problem-solving agents**. Here's a simplified and clear breakdown of the key concepts:

1. Formulate

- The agent **defines a goal** based on its current state or percepts.
- It then **formulates a problem**, which includes:
  - Initial state
  - Possible actions
  - Transition model (what each action does)
  - Goal test
  - Path cost (if any)

2. Search

- The agent invokes a **search algorithm** to find a **sequence of actions** that leads from the initial state to the goal state.
- The output of this phase is called a **solution** (i.e., an action sequence or plan).

3. Execute

- The agent **executes the first action** in the plan, then moves to the next, until the goal is reached.
- During execution, the agent typically **does not reconsider** its percepts — it trusts the precomputed plan.
- Once the solution has been executed, the agent will formulate a new goal.

OPEN-LOOP:

Notice that while **the agent is executing the solution sequence it ignores its percepts when choosing an action because it knows in advance what they will be**.

An agent that **carries out its plans with its eyes closed**, so to speak, must be quite certain of what is going on. Control theorists call **this an** **open-loop system**, because ignoring the percepts breaks the loop between agent and environment.

- **Open-Loop System**
  - This is a system that **does not use feedback** from the environment during execution.
  - The agent assumes the environment behaves as expected, hence **ignores percepts during execution**.
  - It is efficient but risky in uncertain or dynamic environments.

Well-defined problems and solutions:

A problem can be defined formally **by five components**:

1. Initial state
2. **Actions**
3. **Transition model**
4. **Goal test**
5. **Path Cost**

1. Initial State

- The state in which the agent **starts**.
- Denoted as: initial_state

Example: In a navigation problem, the initial state might be the **starting location** of the agent.

Initial State: In(New Delhi) [if in case you are traveling to Jaipur from New Delhi]

Well-defined problems and solutions:

A problem can be defined formally **by five components**:

1. Initial state

2. Actions

3. Transition model

4. Goal test

5. Path Cost

2. Actions (or Successor Function)

- Describes the **possible actions** the agent can take in each state.
- Also called the **successor function**, which returns the set of possible next states.

Example: A description of the possible **actions** available to the agent. Given a particular state *s*, **ACTIONS(s)** returns the set of actions that can be executed in *s*. We say that each of these actions is **applicable** in *s*. For example, from the state ***In(New Delhi)***, the applicable actions are **{** ***Go(Gurugram), Go(Agra), Go(Jaipur)*** **}**.

Well-defined problems and solutions:

A problem can be defined formally **by five components**:

1. **Initial state**
2. **Actions**
3. **Transition model**
4. **Goal test**
5. **Path Cost**

3. Transition Model

- Describes what **state results** from performing a given action in a given state.
- It defines the **dynamics** of the environment.

Example: If you drive from Arad to Sibiu, the transition model tells you that the new state is being in Sibiu.

- A description of what each action does; the formal name for this is the **transition model**, specified by a function RESULT(s, a) that returns the state that results from doing action a in state s. We also use the term **successor** to refer to any state reachable from a given state by a single action.[2] For example, we have

$$\text{RESULT}(In(Arad), Go(Zerind)) = In(Zerind) .$$

Well-defined problems and solutions:

A problem can be defined formally **by five components**:

1. Initial state

2. Actions

3. Transition model

4. Goal test

5. Path Cost

4. Goal Test

- A function to determine whether a given state is a **goal state**.
- It returns true if the goal has been achieved.

Example: In a puzzle, the goal test checks if all tiles are in the correct order.

- The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set $\{In(Bucharest)\}$.

Well-defined problems and solutions:

A problem can be defined formally **by five components**:

1. Initial state

2. Actions

3. Transition model

4. Goal test

5. Path Cost

5. Path Cost

- A **numeric cost** associated with a path (a sequence of actions).
- Used to evaluate the **efficiency** of the solution.

Example: In a travel route, the path cost could be total distance, time, or fuel consumption.

Well-defined problems and solutions:

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Ref Book: Page No. 67, 3rd-Edition_AI_Russell-and-Norvig

Problem-solving approaches:

1. Standardized Problems (Toy problem)

- **Purpose**: Designed mainly for *illustrating* or *testing* problem-solving methods.
- **Features**:
  - ✓ Have a **clear, concise, and exact description**.
  - ✓ Easy to replicate for comparison across different algorithms.
  - ✓ Often used as **benchmarks** in research.
- **Examples**:
  - **8-puzzle** (sliding tile puzzle).
  - **Travelling Salesman Problem**.
  - **Rubik's Cube**.
  - **Sudoku**.

2. Real-World Problems

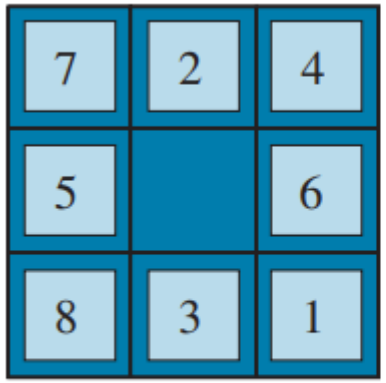
- **Purpose**: Problems whose solutions are **actually used in real life**.
- **Features**:
  - ✓ **Not standardized** — problem definition changes depending on context.
  - ✓ **Data sources and constraints differ** (e.g., hardware, environment).
  - ✓ More complex and messy than neat benchmark problems.
- **Examples**:
  - Robot navigation (different robots → different sensors → different data).
  - Self-driving cars adapting to different road rules.
  - Medical diagnosis using patient-specific data.

Note: The concepts of **standardized problems** and **toy problems** are the same. In *Artificial Intelligence: A Modern Approach by Russell and Norvig*, the **4th edition** uses the term **standardized problems**, while the **3rd edition** uses the term **toy problems**.

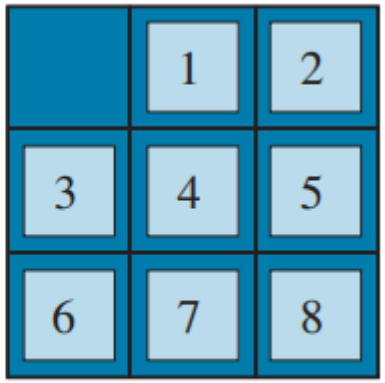
Standardized Problems (Toy problem):

❖Grid world problem

A **grid world problem** is a two-dimensional rectangular array of square cells in which agents can move from cell to cell. Typically, the agent can move to any obstacle-free adjacent cell— horizontally or vertically, and in some problems diagonally. Cells can contain objects, which the agent can pick up, push, or otherwise act upon; a wall or other impassible obstacle in a cell prevents an agent from moving into that cell.



Start State



Goal State

The vacuum world can be formulated as a grid world problem as follows:

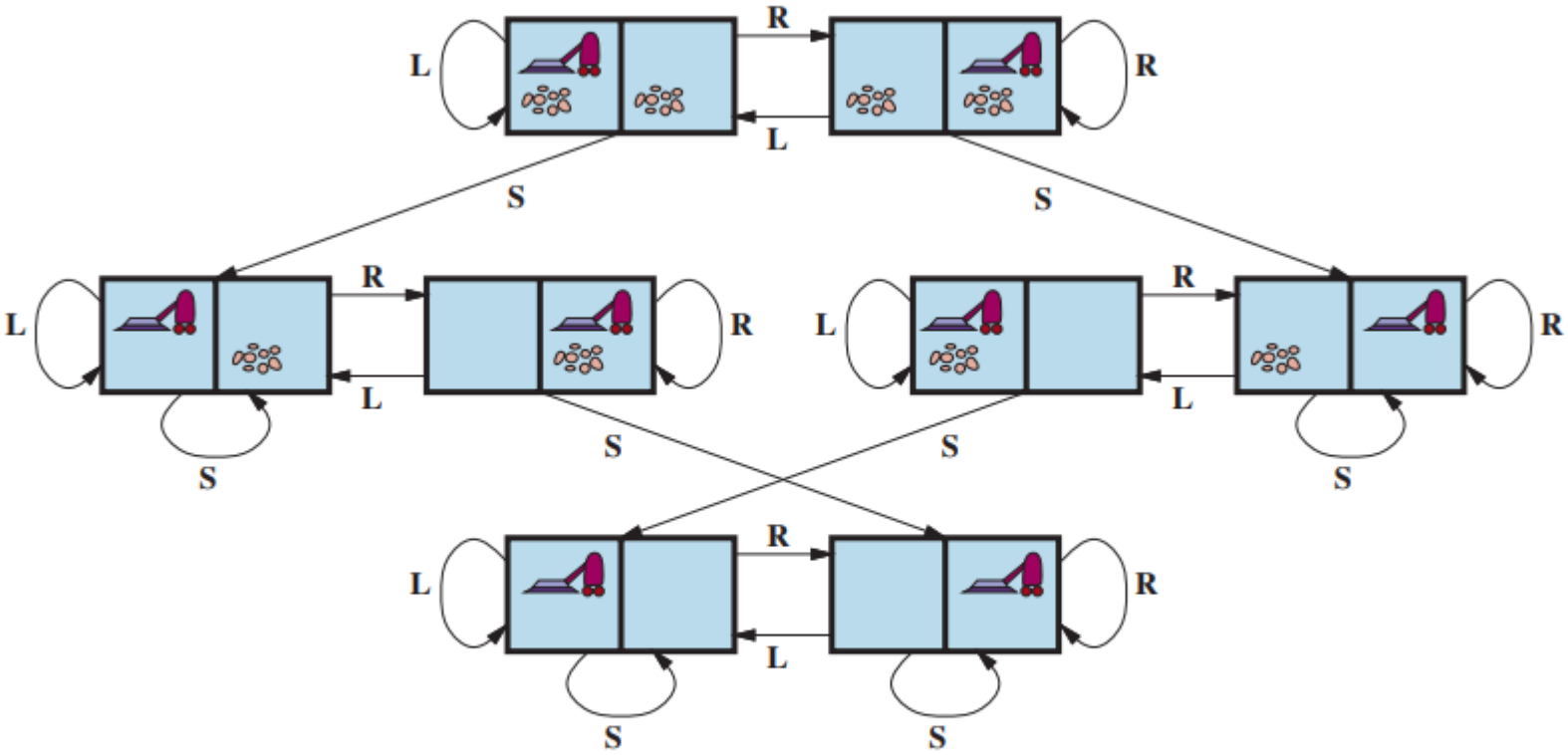


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Ref Book: Page No. 85, 4th-Edition_AI_Russell-and-Norvig

The vacuum world can be formulated as a grid world problem as follows:

- **States**: A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. In the simple two-cell version, the agent can be in either of the two cells, and each cell can either contain dirt or not, so there are $2 \cdot 2 \cdot 2 = 8$ states (see Figure 3.2). In general, a vacuum environment with n cells has $n \cdot 2^n$ states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In the two-cell world we defined three actions: *Suck*, move *Left*, and move *Right*. In a two-dimensional multi-cell world we need more movement actions. We could add *Upward* and *Downward*, giving us four **absolute** movement actions, or we could switch to **egocentric actions**, defined relative to the viewpoint of the agent—for example, *Forward*, *Backward*, *TurnRight*, and *TurnLeft*.
- **Transition model**: *Suck* removes any dirt from the agent's cell; *Forward* moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect. *Backward* moves the agent in the opposite direction, while *TurnRight* and *TurnLeft* change the direction it is facing by 90°.
- **Goal states**: The states in which every cell is clean.
- **Action cost**: Each action costs 1.

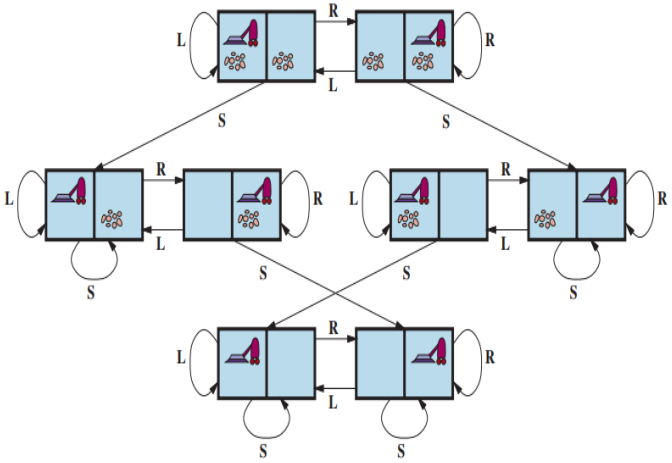


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Ref Book: Page No. 85, 4th-Edition_Artificial Intelligence: A Modern Approach by Russell-and-Norvig

Real-World Problems:

- We have already seen how the route-finding problem is defined in terms of specified locations and transitions along edges between them. Route-finding algorithms are used in a variety of applications.

- Consider the airline travel problems that must be solved by a travel-planning website.

- **States**: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.
- **Initial state**: The user's home airport.
- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **Transition model**: The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new time.
- **Goal state**: A destination city. Sometimes the goal can be more complex, such as "arrive at the destination on a nonstop flight."
- **Action cost**: A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on.

Real-World Problems:

- **Touring problems:** Traveling salesperson problem (TSP) - is a touring problem in which every city on a map must be visited. The aim is to find a tour with cost < C (or in the optimization version, to find a tour with the lowest cost possible).

- **VLSI layout problem:** The layout problem comes after the logical design phase and is usually split into two parts: **cell layout** and **channel routing**.

- **Robot navigation**

- **Automatic assembly sequencing**

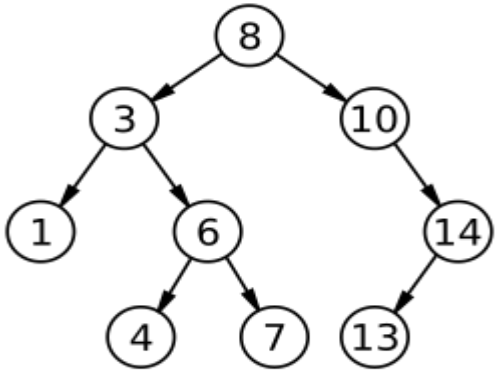
Search Algorithms:

A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure.

We will consider algorithms that superimpose a **search tree** over the **state space graph**, forming various paths from the **initial state**, trying to find a **path** that reaches a **goal state**.

- Each **node** in the search tree corresponds to **a state in the state space.**
- The **edges** in the search tree correspond to **actions**.
- The **root** of the tree corresponds to the **initial state of the problem**.

Search: Starting at the root, comparing keys to navigate left or right until the node is found.



Classical AI Search Problems:

Problem	Algorithmic Approach/Teaches
Water Jug	State space & operators
8-Puzzle	Heuristics, A*
Wolf–Goat–Cabbage	Constraints
Missionaries & Cannibals	Safe state pruning
Tower of Hanoi	Recursion, exponential states
Bridge & Torch	Optimal cost search
N-Queens	CSP, backtracking
Blocks World	Planning
Sliding Block (Klotski)	Heuristic necessity
Maze Solving	Pathfinding algorithms
Cryptarithmetic	CSP with arithmetic constraints

Water Jug Problem:

The objective is to measure a specific quantity of water by performing operations like filling a jug, emptying a jug, or transferring water between the two jugs. The problem can be stated as follows:

- You are given two jugs, one with a capacity of **X** liters and the other with a capacity of **Y** liters.
- You need to measure exactly **Z** liters of water using these two jugs.

- The allowed operations are:
  - Fill one of the jugs.
  - Empty one of the jugs.
  - Pour water from one jug into another until one jug is either full or empty.

The Water Jug Problem is an excellent example to introduce key AI concepts such as **state space, search algorithms, and heuristics**.

Water Jug Problem:

State Space Representation:

In AI terms, the Water Jug Problem can be described using a **state space** representation, where:

- Each state is represented by a tuple **(a, b)**, where **a** is the amount of water in the first jug and **b** is the amount of water in the second jug.
- The initial state is **(0, 0)**, meaning both jugs are empty.
- The goal state is any configuration **(a, b)** where **a** or **b** equals the desired amount **Z**.
- Transitions between states occur when one of the allowed operations is performed.

Water Jug Problem:

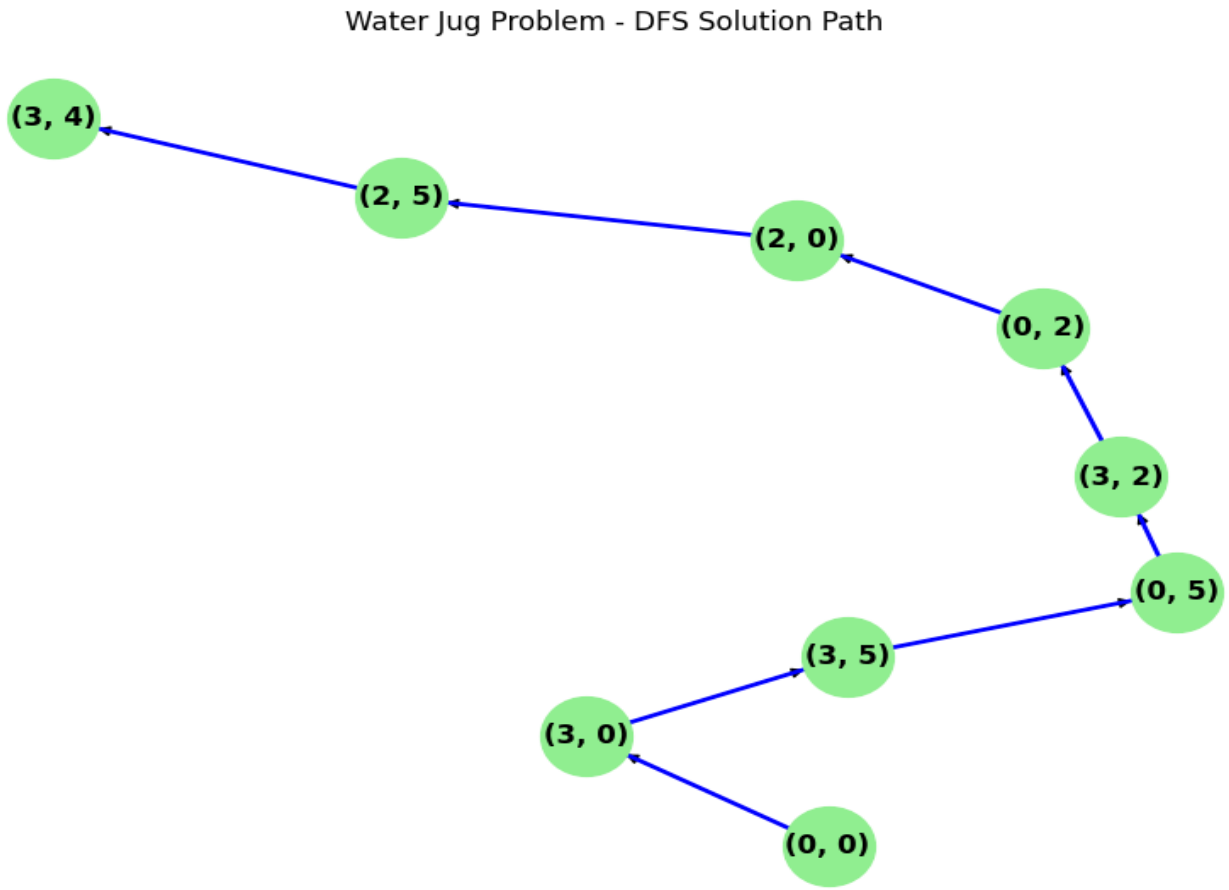
Search Algorithms to Solve the Water Jug Problem

To solve the Water Jug Problem using AI techniques, we can apply search algorithms like

> **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**.

These algorithms systematically explore the state space to find the optimal sequence of operations that leads to the goal state.

Water Jug Problem:



Water Jug Problem: Clear Explanation

Problem Statement:

You are given:

- A **4-liter jug**
- A **3-liter jug**
- Unlimited water supply
- No measurement markings on jugs

Goal: Measure exactly **2 liters** of water.

How AI Sees This Problem:

- AI does **not** see jugs.
- AI sees **states** and **actions**.

❖ State Representation:

A **state** is written as: **(x, y)**

Where:

x= water in 4L jug

y= water in 3L jug

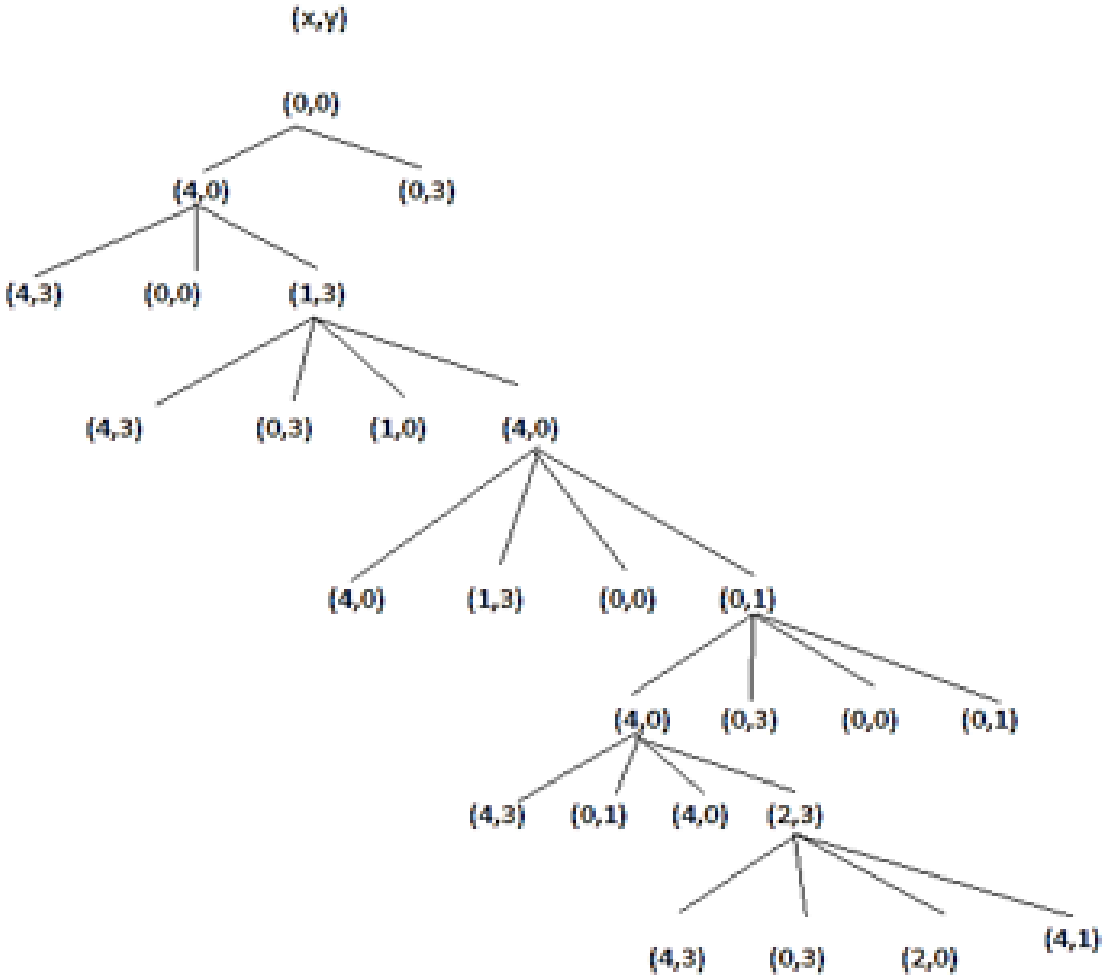
Example:

$(4,0)$→4L jug full, 3L jug empt

Allowed Operations: Fill a jug, Empty a jug, and Pour water from one jug to another (until one is empty or the other is full)

Start state: (0,0)

Goal state:
(2,0) => 4L jug must contain 2 liters (y can be anything)



State Space Search: Water Jug Problem

"You are given two jugs, a 4-litre one and a 3-litre one.
Neither has any measuring markers on it. There is a
pump that can be used to fill the jugs with water. How
can you get exactly 2 litres of water into 4-litre jug."

State Space Search: Water Jug Problem

- State: (x, y)

  x = 0, 1, 2, 3, or 4 y = 0, 1, 2, 3

- Start state: (0, 0).
- Goal state: (2, n) for any n.
- Attempting to end up in a goal state.

(i) Empty a jug

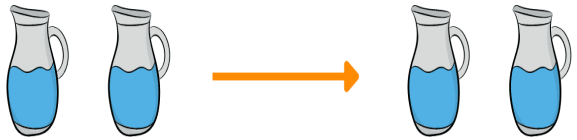
(ii) Fill a jug

(iii) Transfer

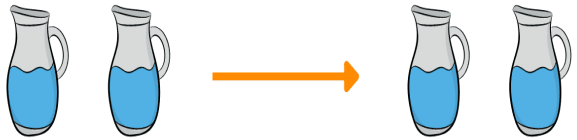
Step	State (4L,3L)	Action Performed	Explanation
1	(0,0)	Start	Both jugs empty
2	(4,0)	Fill 4L	Filled 4L jug
3	(1,3)	Pour 4L → 3L	3L fills, 1L left in 4L
4	(1,0)	Empty 3L	Emptied 3L jug
5	(0,1)	Pour 4L → 3L	Transfer 1L to 3L
6	(4,1)	Fill 4L	Fill 4L again
7	(2,3)	Pour 4L → 3L	3L fills (needs 2L), **2L left in 4L**

Goal reached: 4L jug has **2 liters**

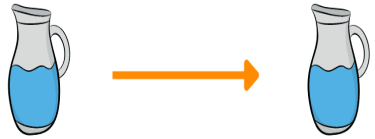
(i) Empty a jug



(ii) Fill a jug



(iii) Transfer



1	5L	3L
2	5L - 3L = 2L	3L
3	2L	0L
4	5L	2L
5	5L - 1L = 4L	2L + 1L = 3L
6	4L	0L
7	4L	0L

How does General Search work?

```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

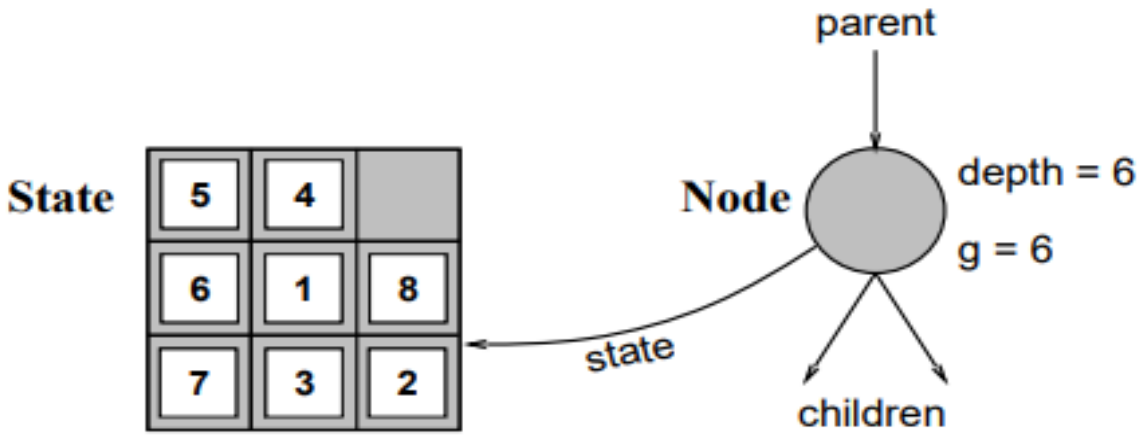
Implementation of search algorithms

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

    nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
    loop do
        if nodes is empty then return failure
        node ← REMOVE-FRONT(nodes)
        if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
        nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
    end
```

Implementation contd: states vs. nodes

A *state* is a (representation of) a physical configuration
A *node* is a data structure constituting part of a search tree
includes *parent*, *children*, *depth*, *path cost* $g(x)$
States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

Search strategies

A 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?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory
- optimality—does it always find a least-cost solution?

Time and space complexity are measured in terms of

- b—maximum branching factor of the search tree
- d—depth of the least-cost solution
- m—maximum depth of the state space (may be ∞)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

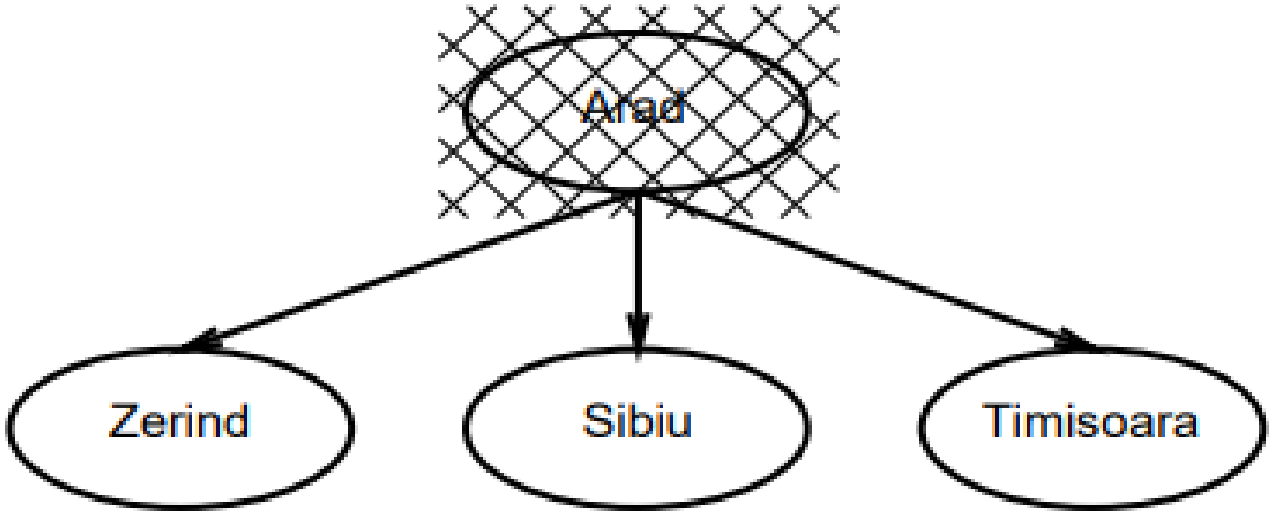
Iterative deepening search

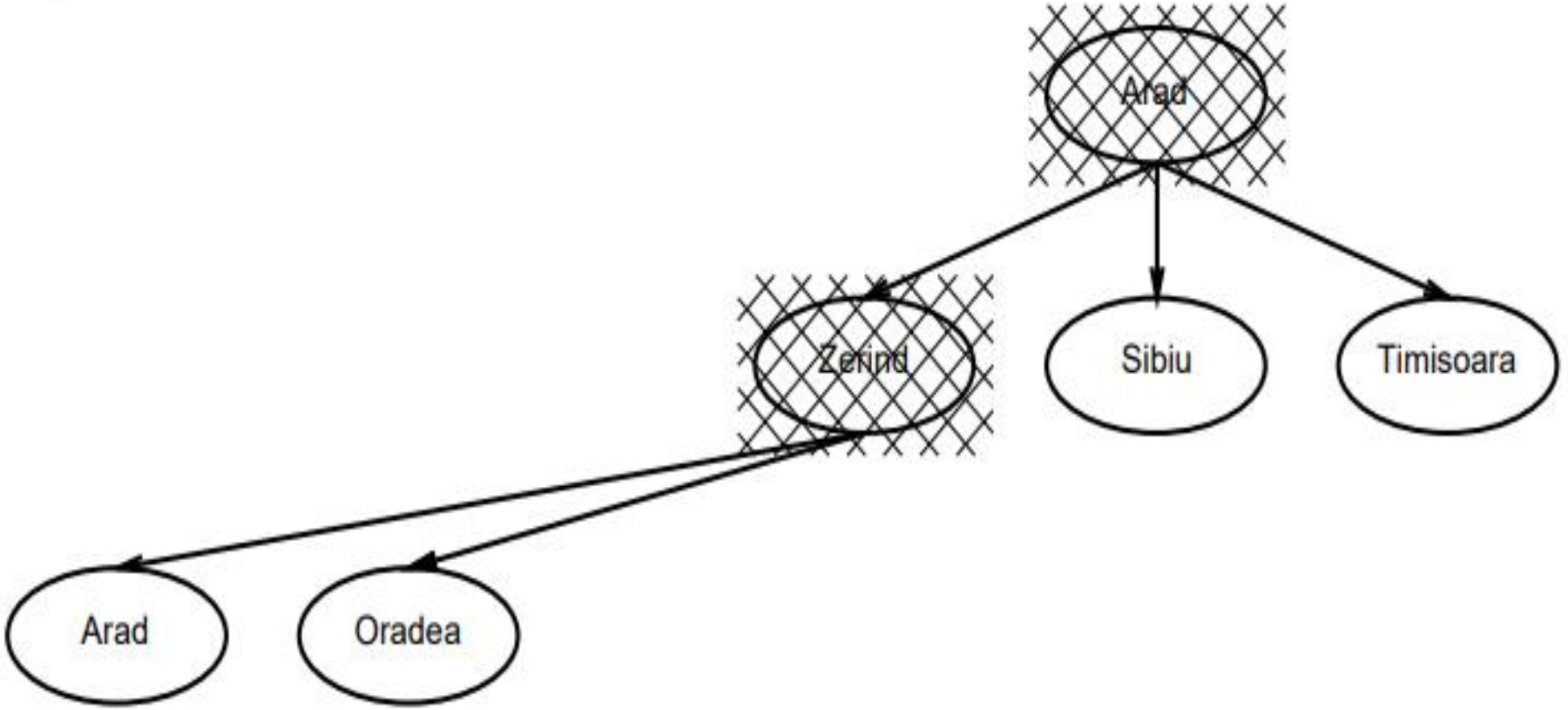
Breadth-first search

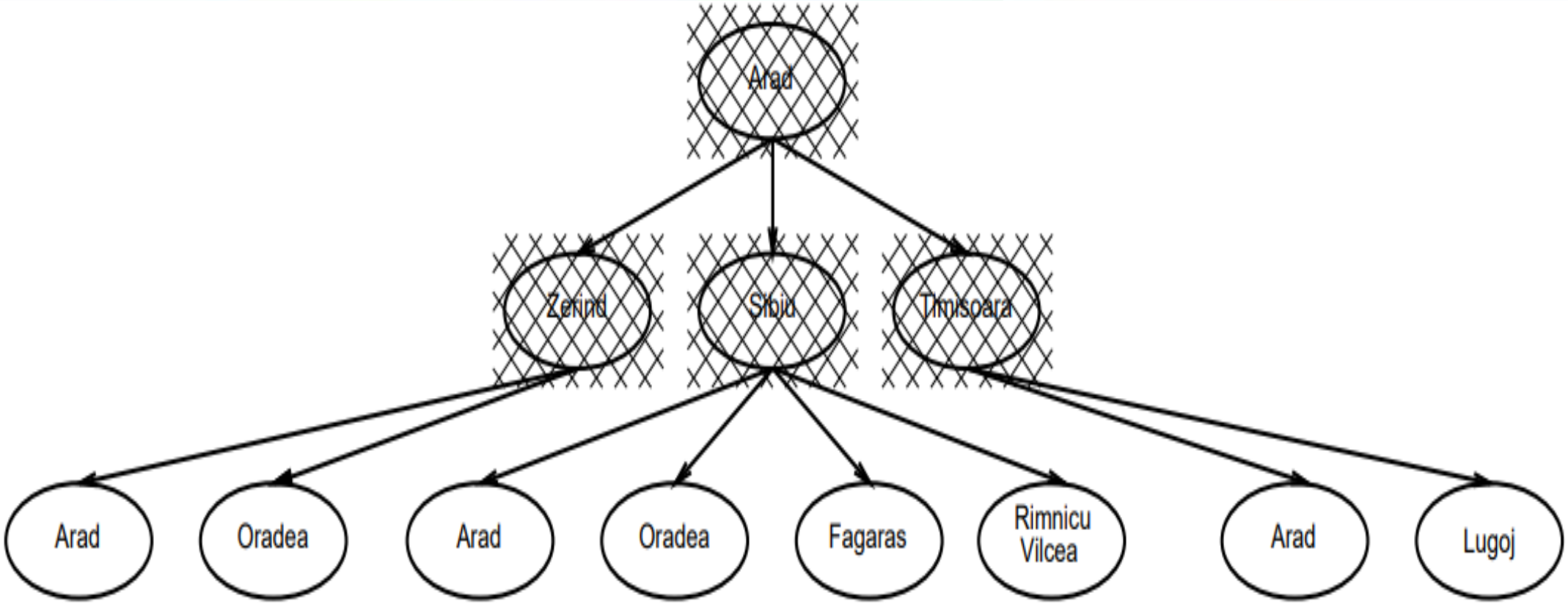
Expand shallowest unexpanded node

Implementation:

QUEUEINGFN = put successors at end of queue


Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Rimnicu Vilcea
Arad
Lugoj

Properties of breadth-first search

Complete??

Time??

Space??

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exponential in d

Space?? $O(b^d)$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

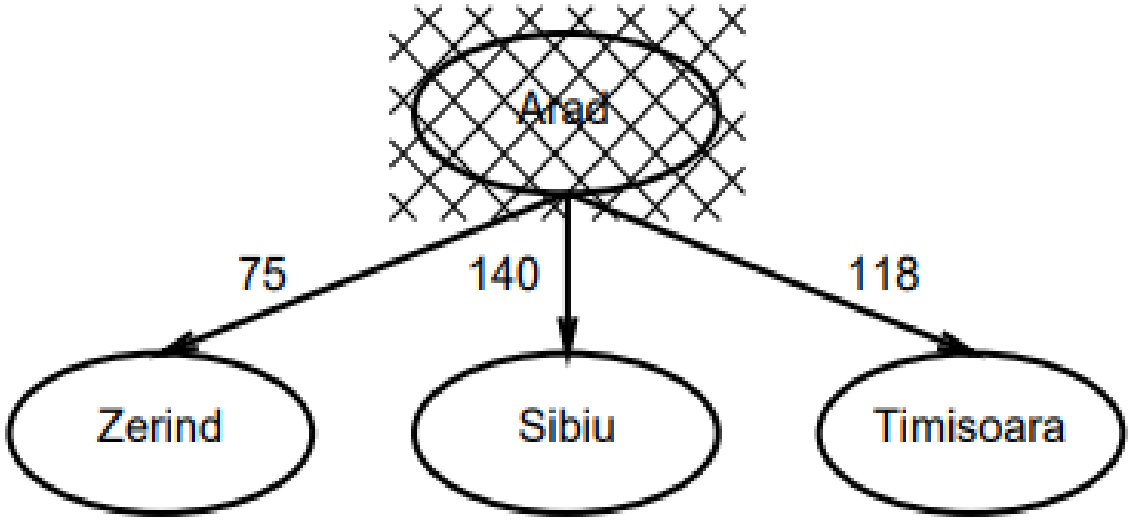
Space is the big problem; can easily generate nodes at 1MB/sec
so 24hrs $=$ 86GB.

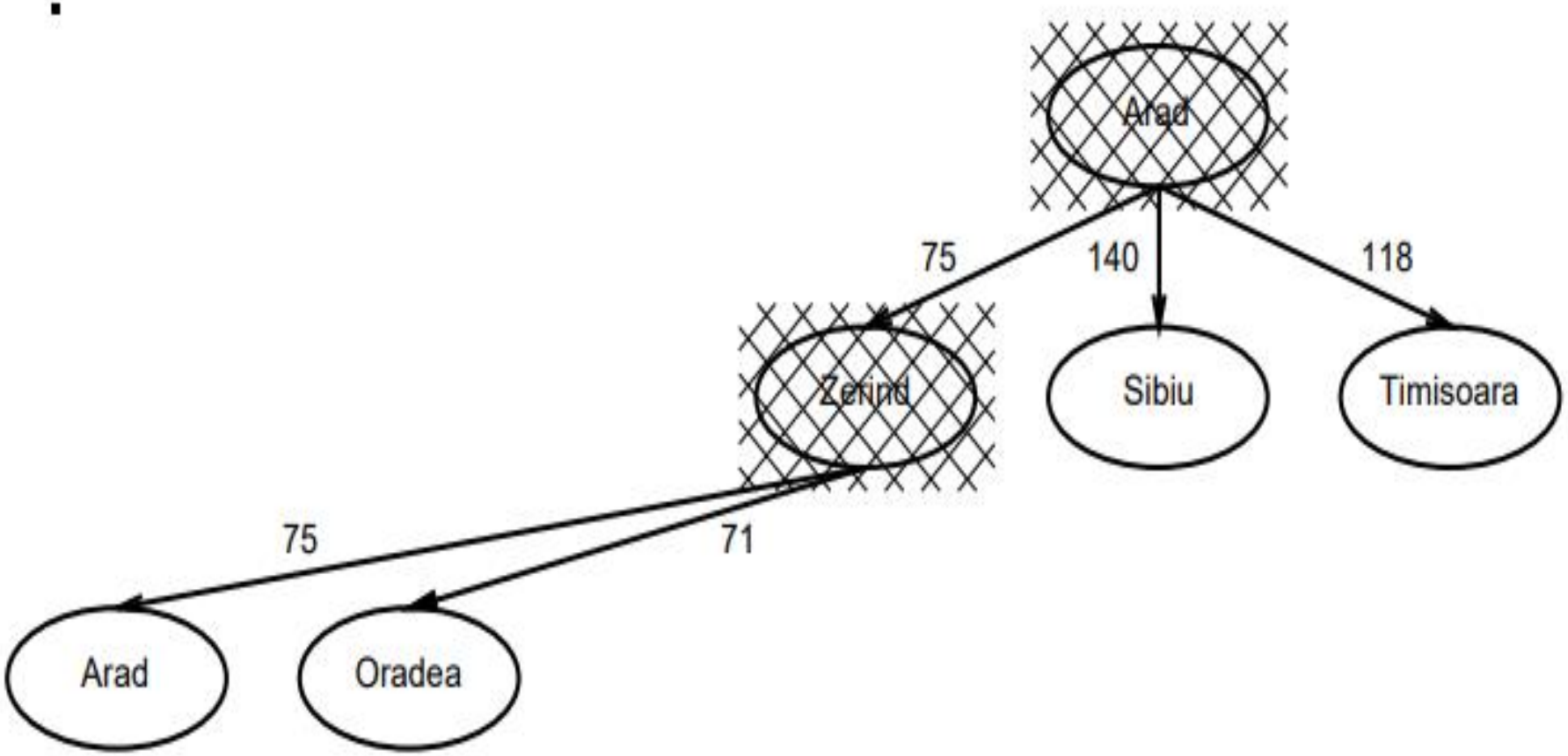
Uniform-cost search

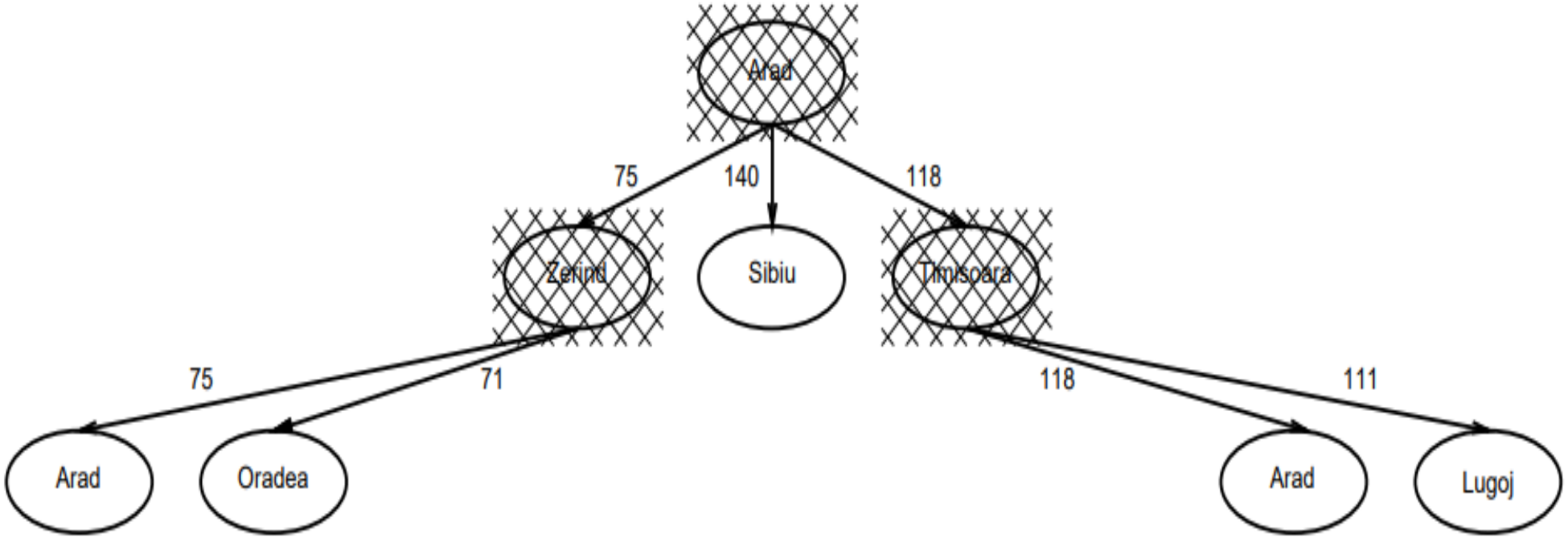
Expand least-cost unexpanded node

Implementation:

QUEUEINGFN = insert in order of increasing path cost


Arad
75
140
118
Zerind
Sibiu
Timisoara


Arad
75
140
118
Zerind
Sibiu
Timisoara
75
71
Arad
Oradea


Arad
75
140
118
Zerind
Sibiu
Timisoara
75
71
118
111
Arad
Oradea
Arad
Lugoj

Properties of uniform-cost search

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution

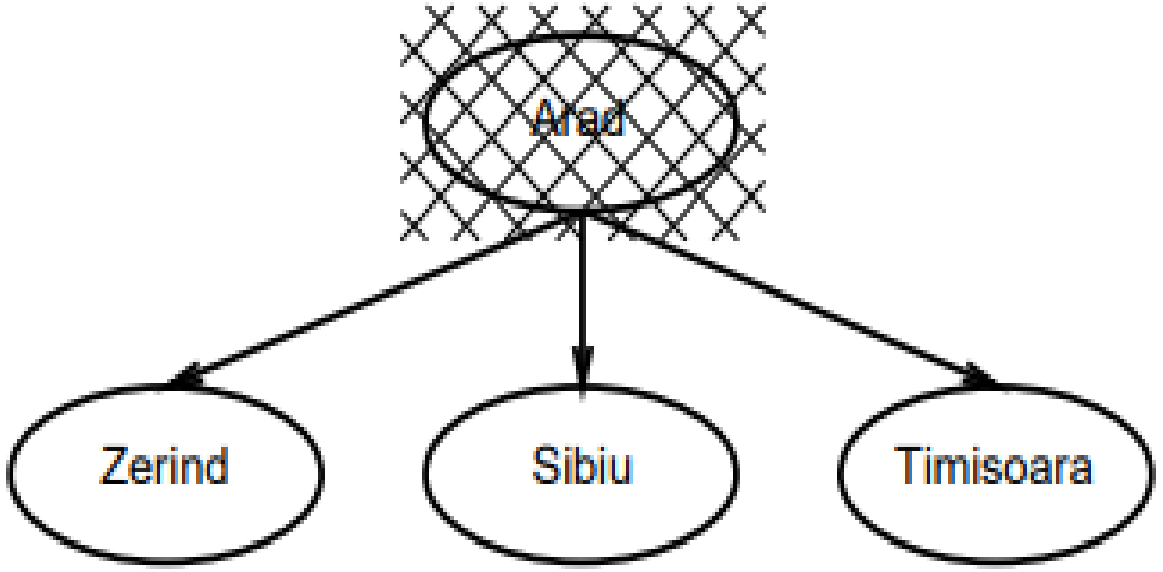
Optimal?? Yes

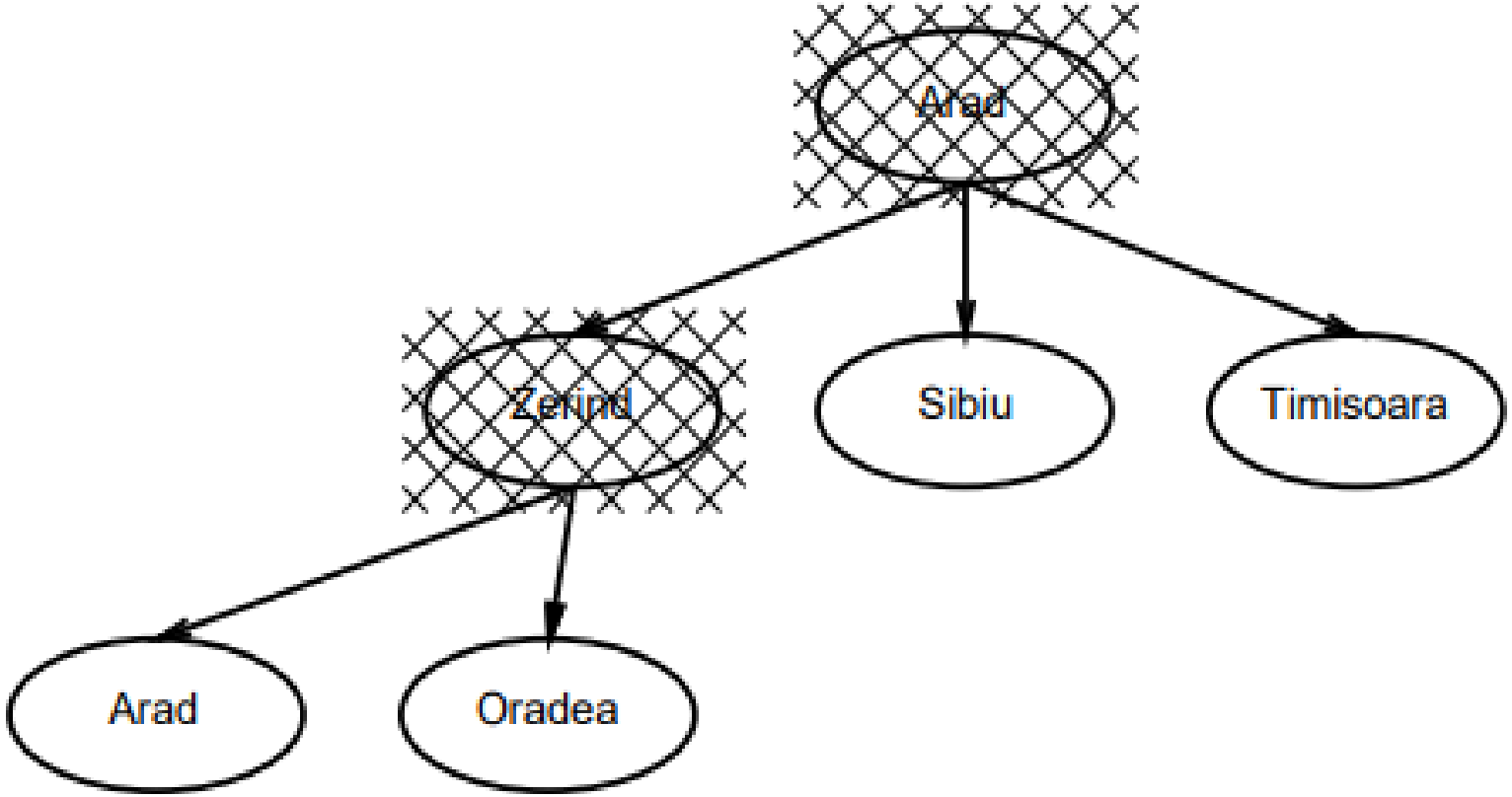
Depth-first search

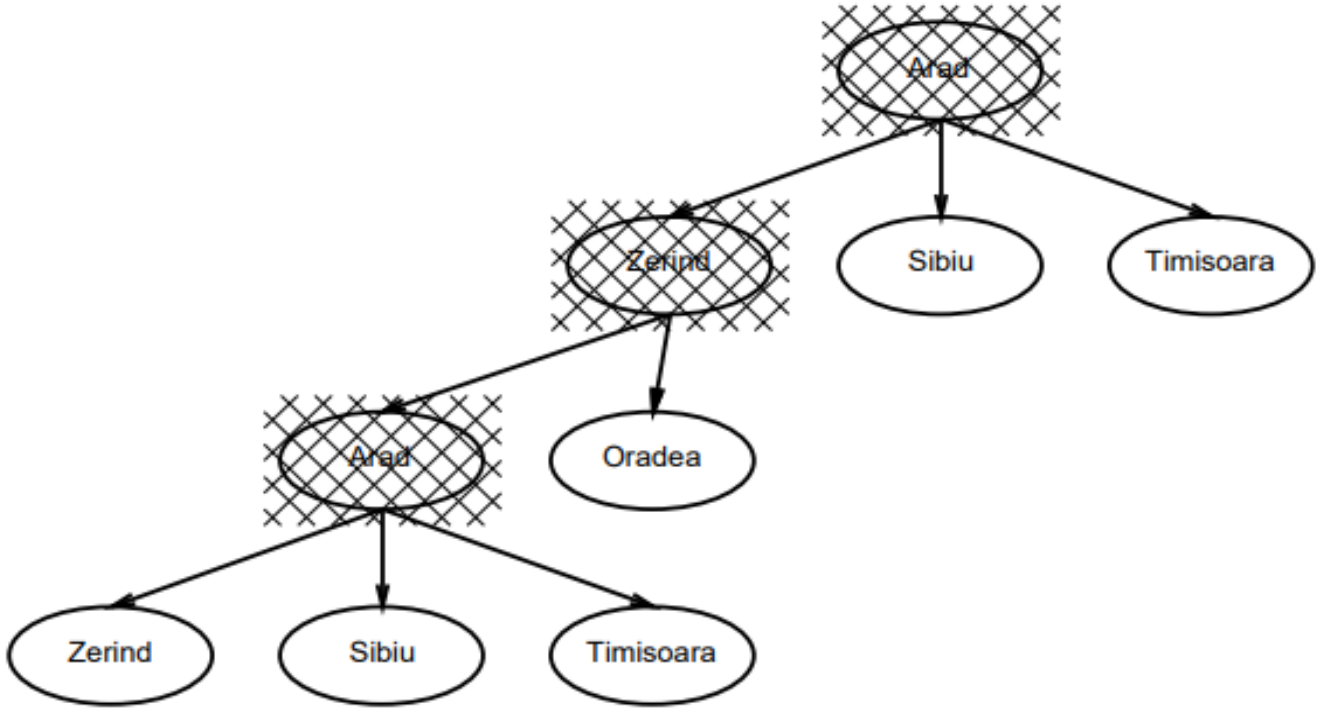
Expand deepest unexpanded node

Implementation:

QUEUEINGFN = insert successors at front of queue

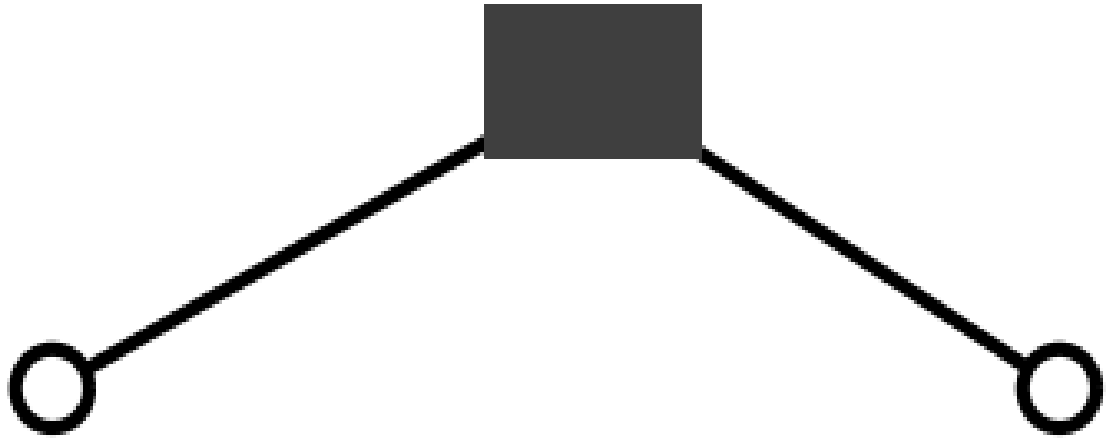

Arad
Zerind
Sibiu
Timisoara

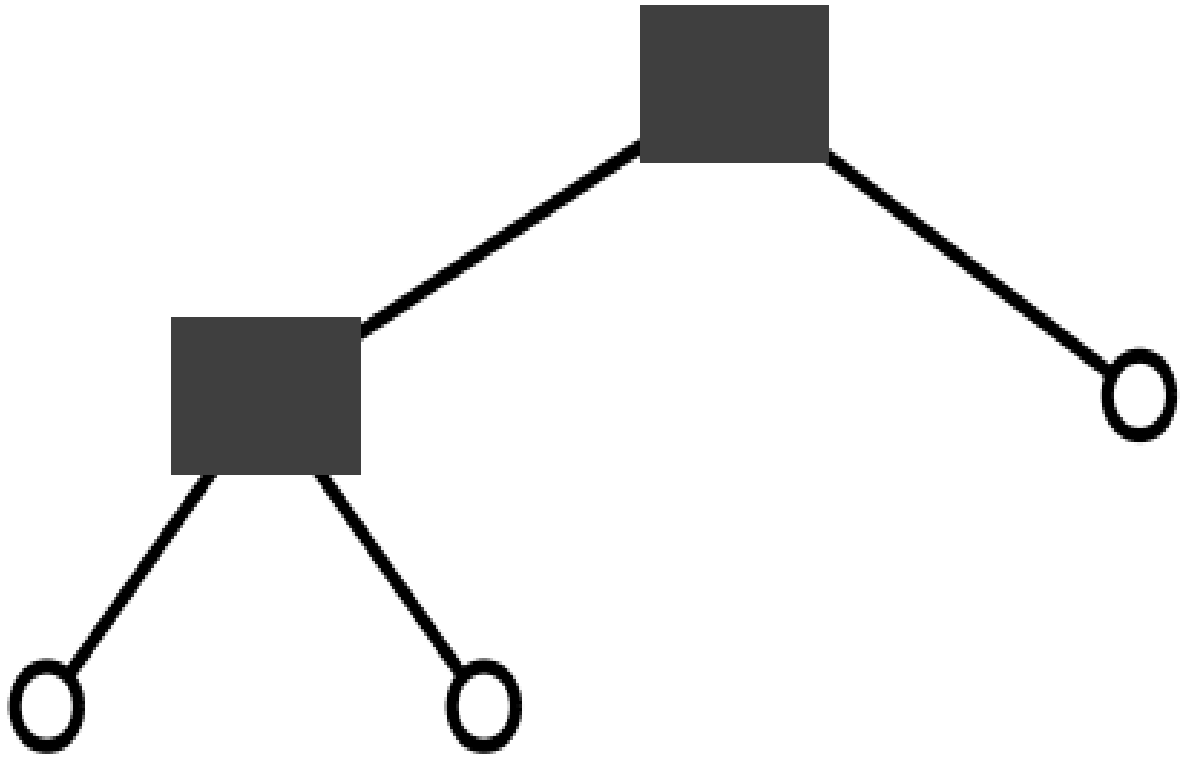

Arad
Zerind
Sibiu
Timisoara
Arad
Oradea

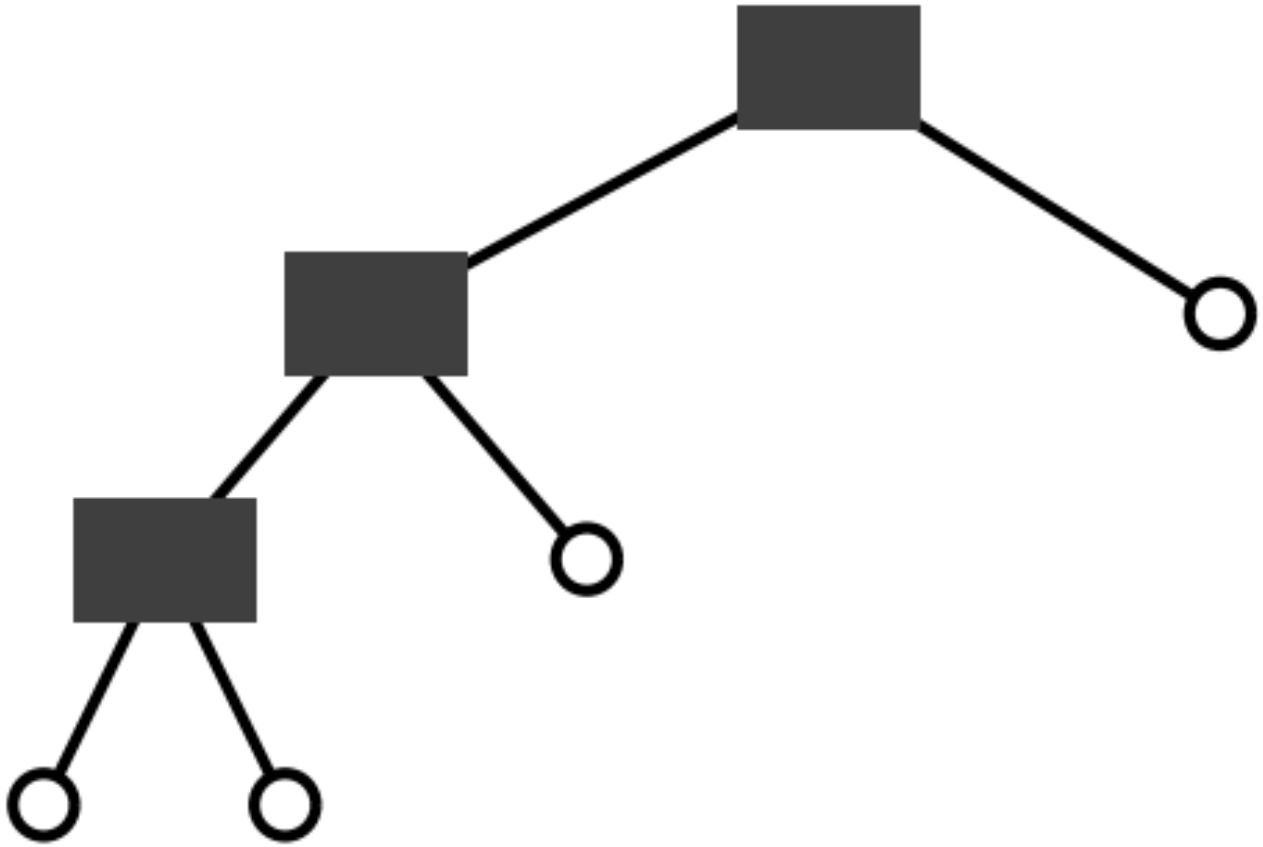


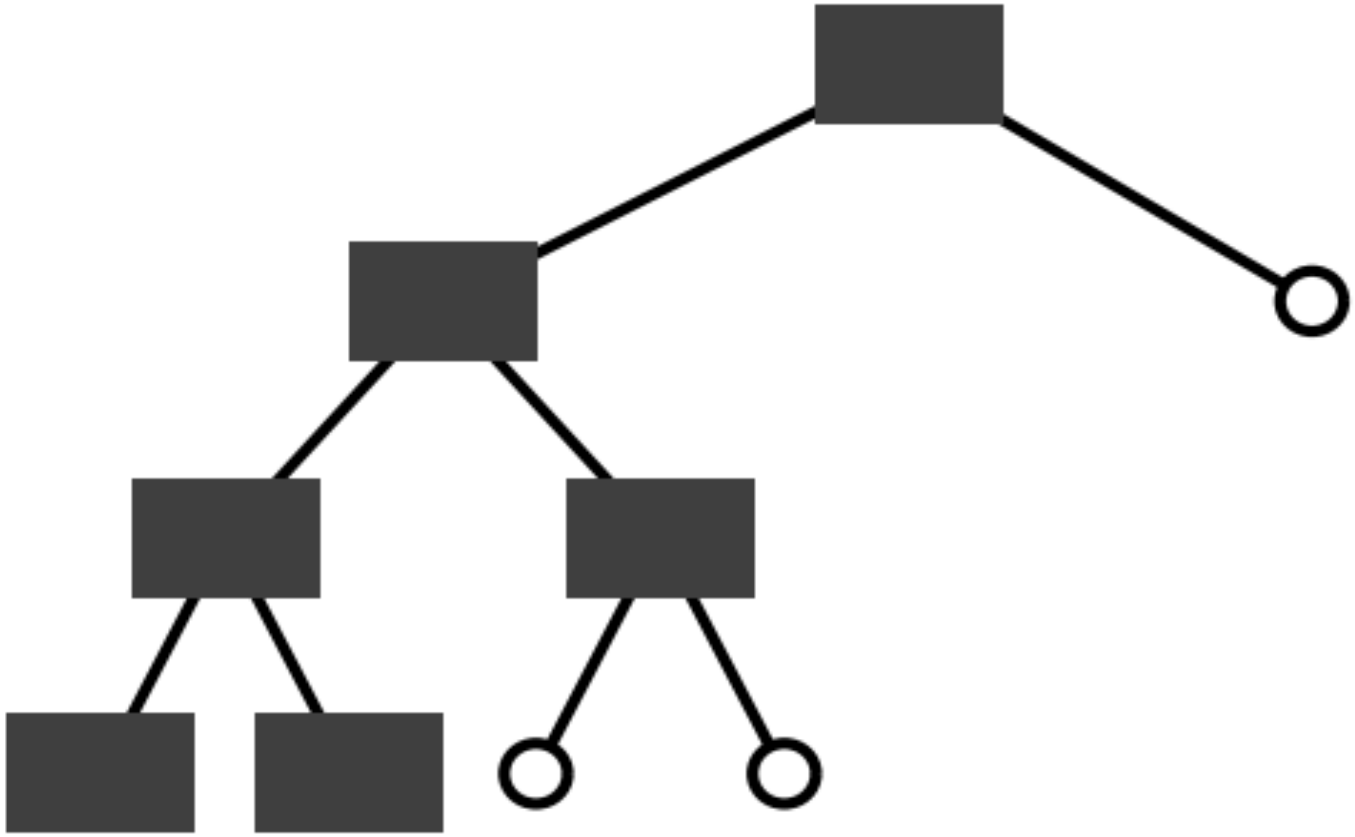
I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)

DFS on a depth-3 binary tree

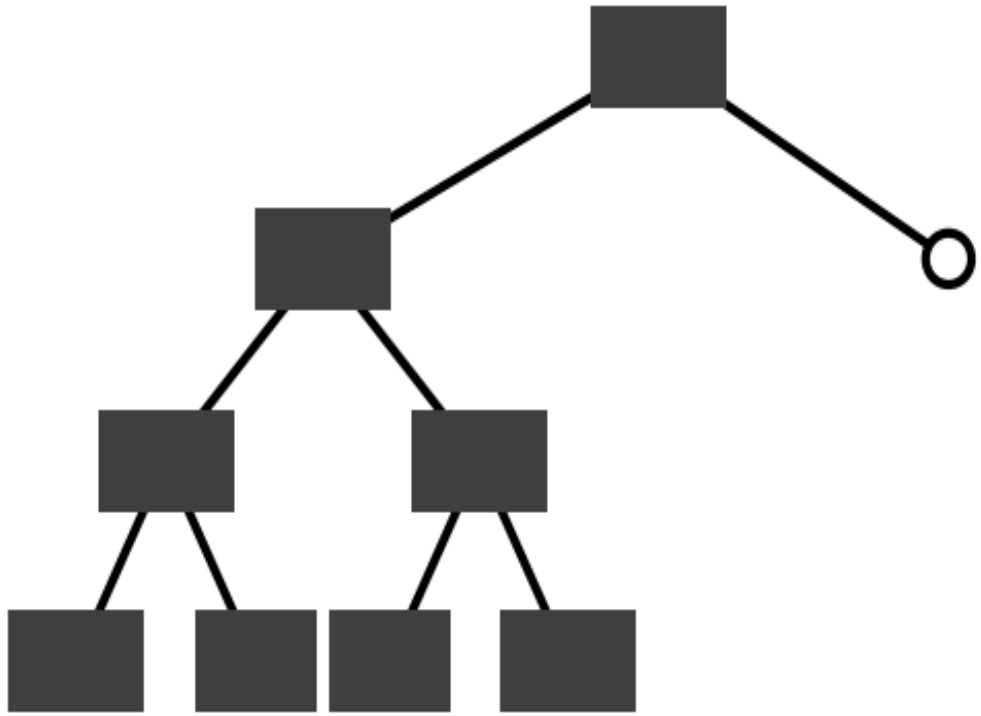


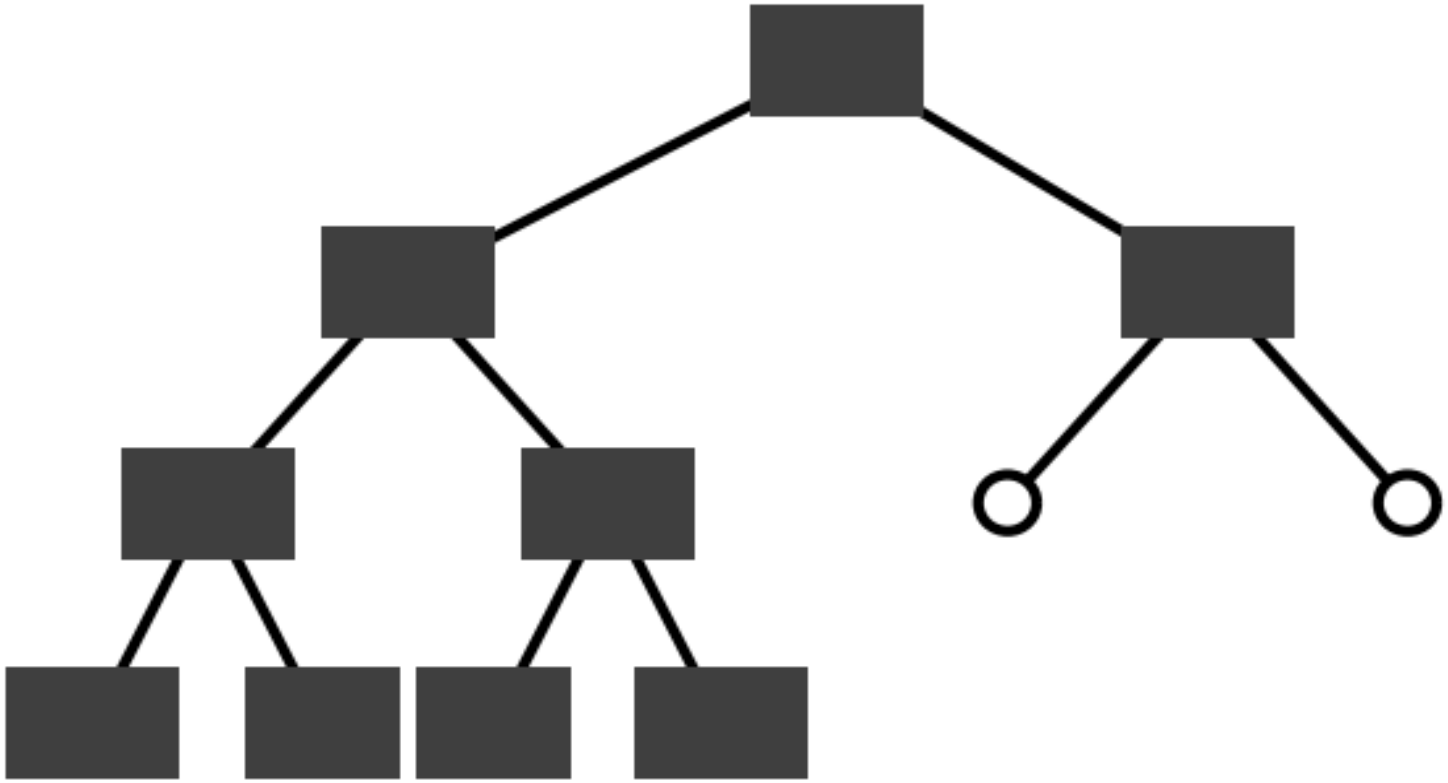


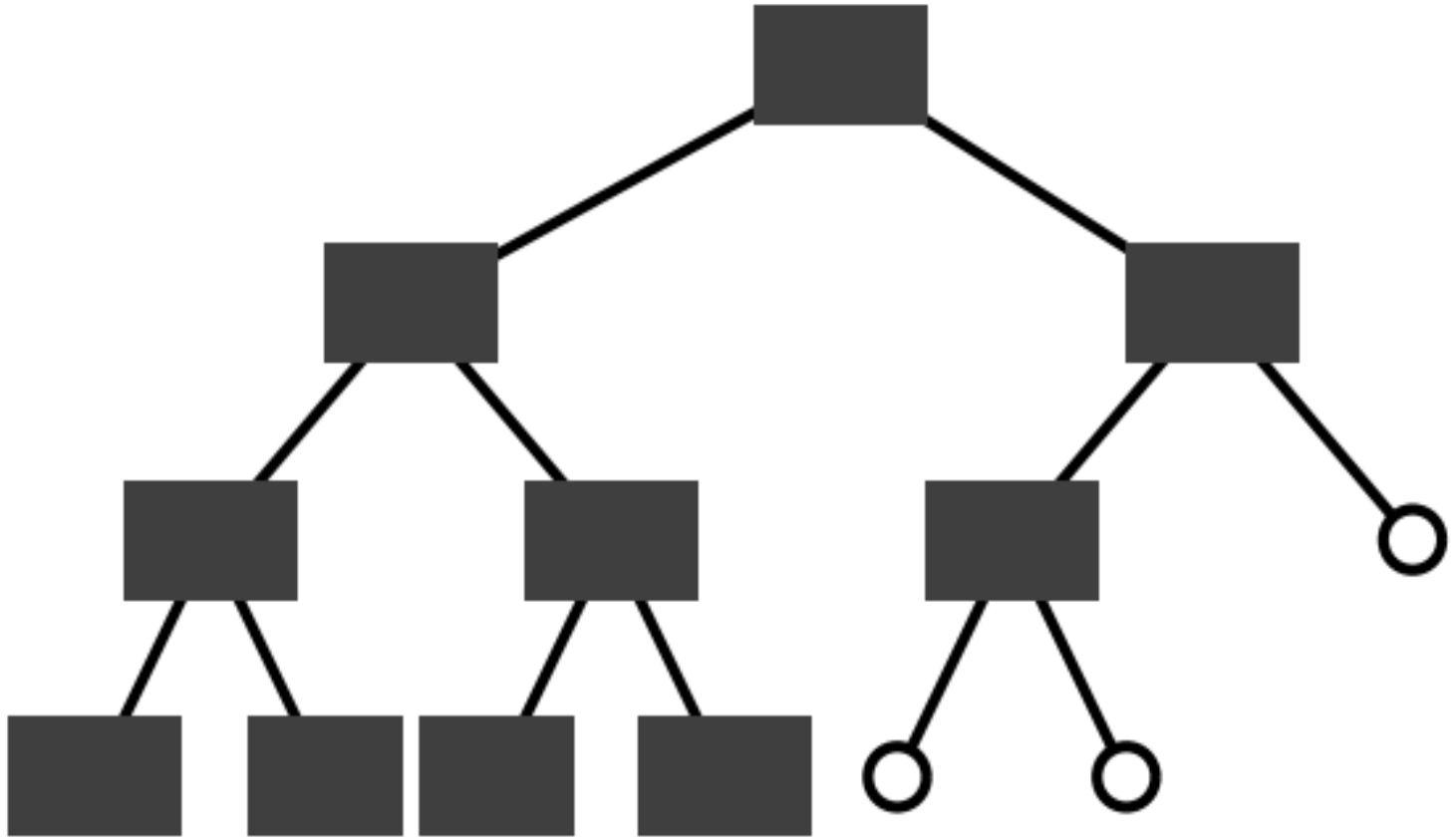


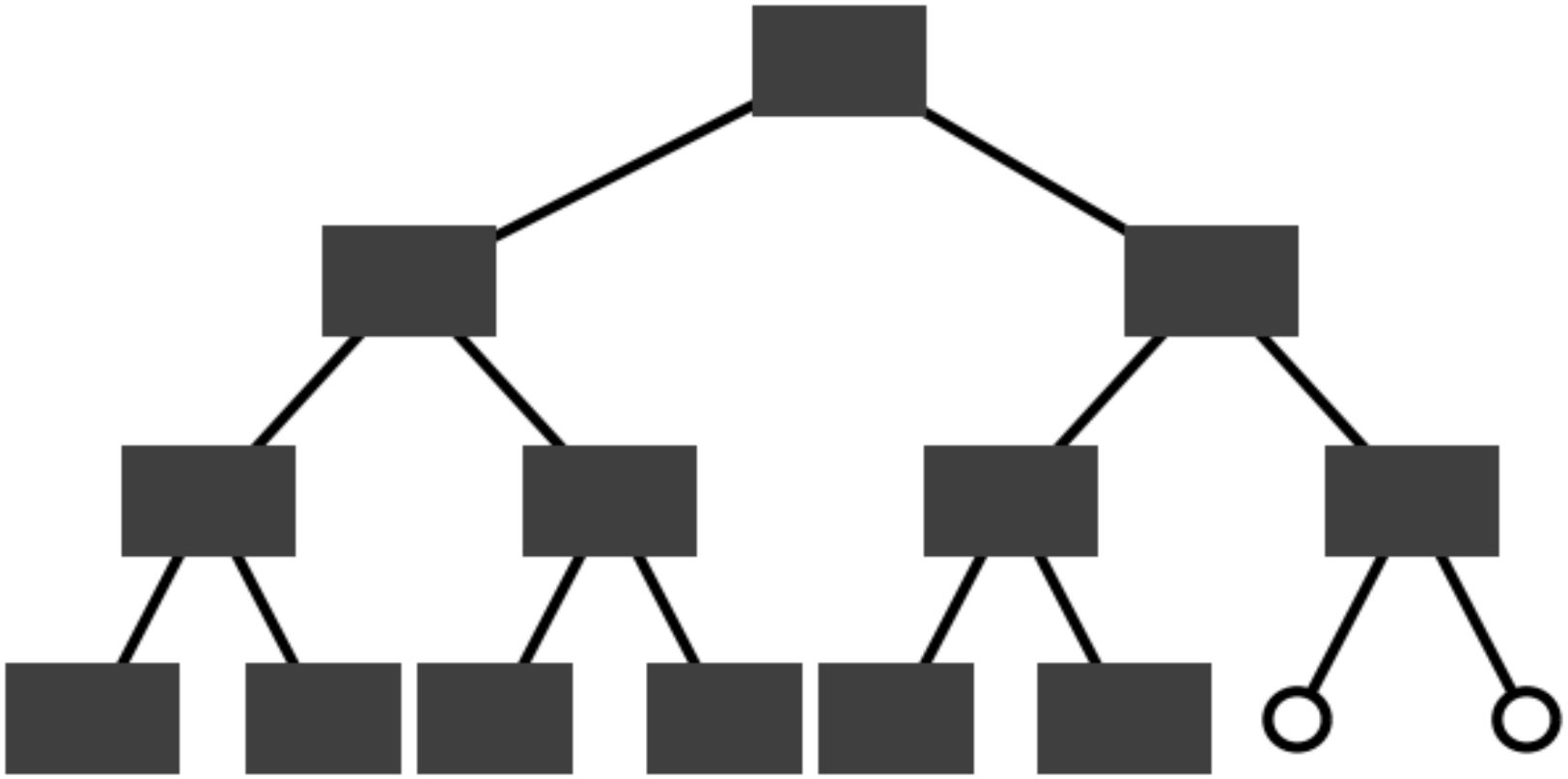


DFS on a depth-3 binary tree, contd.









Properties of depth-first search

Complete??

Time??

Space??

Optimal??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Depth-limited search

= depth-first search with depth limit l

Implementation:

Nodes at depth l have no successors

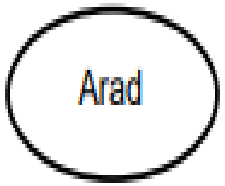
Iterative deepening search

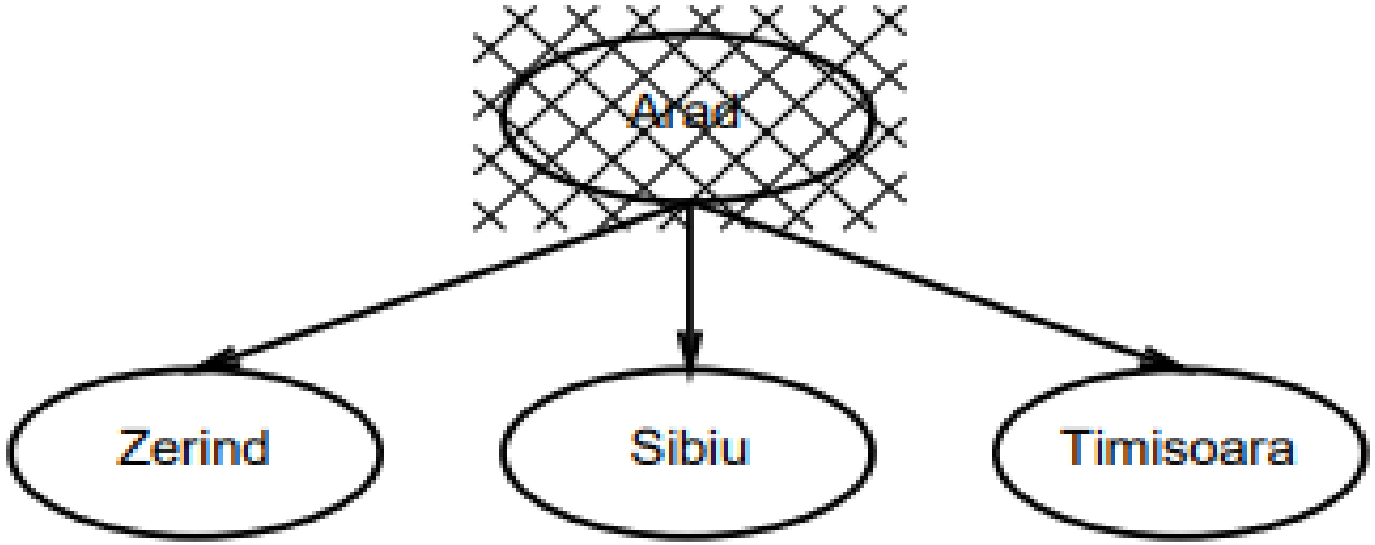
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution sequence
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

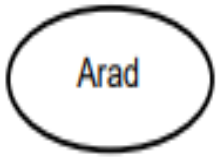
Iterative deepening search $l = 0$

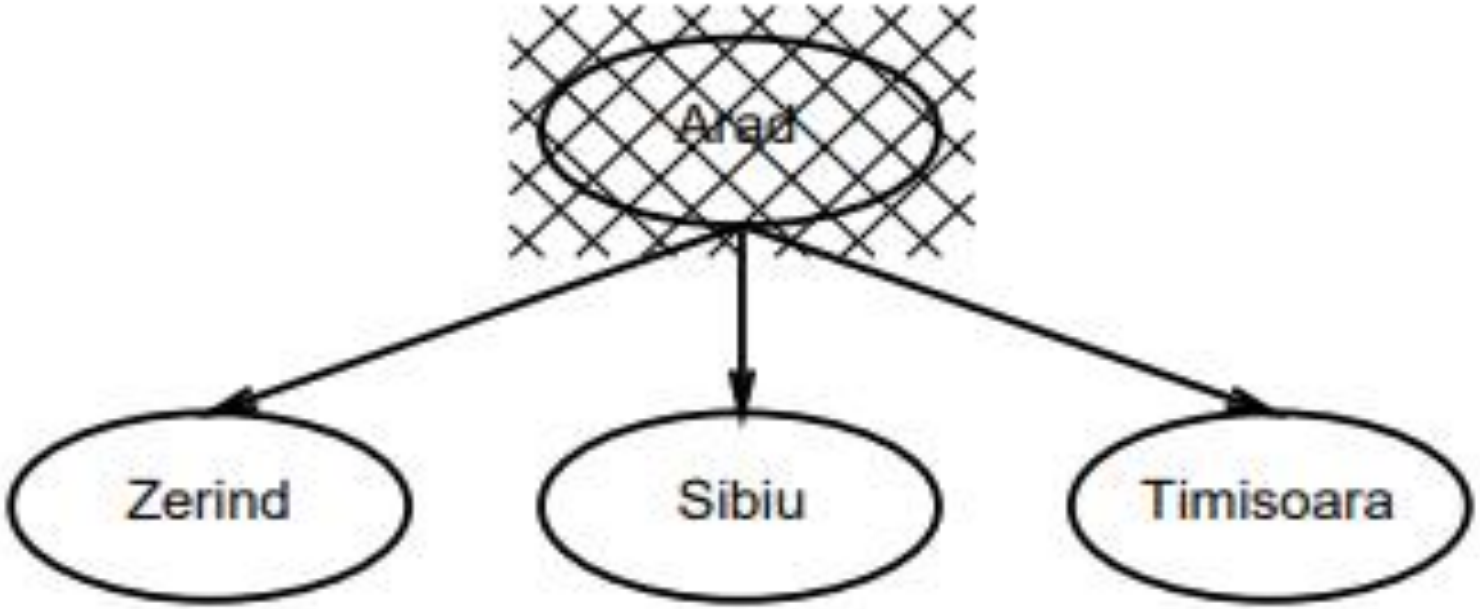
Iterative deepening search $l = 1$

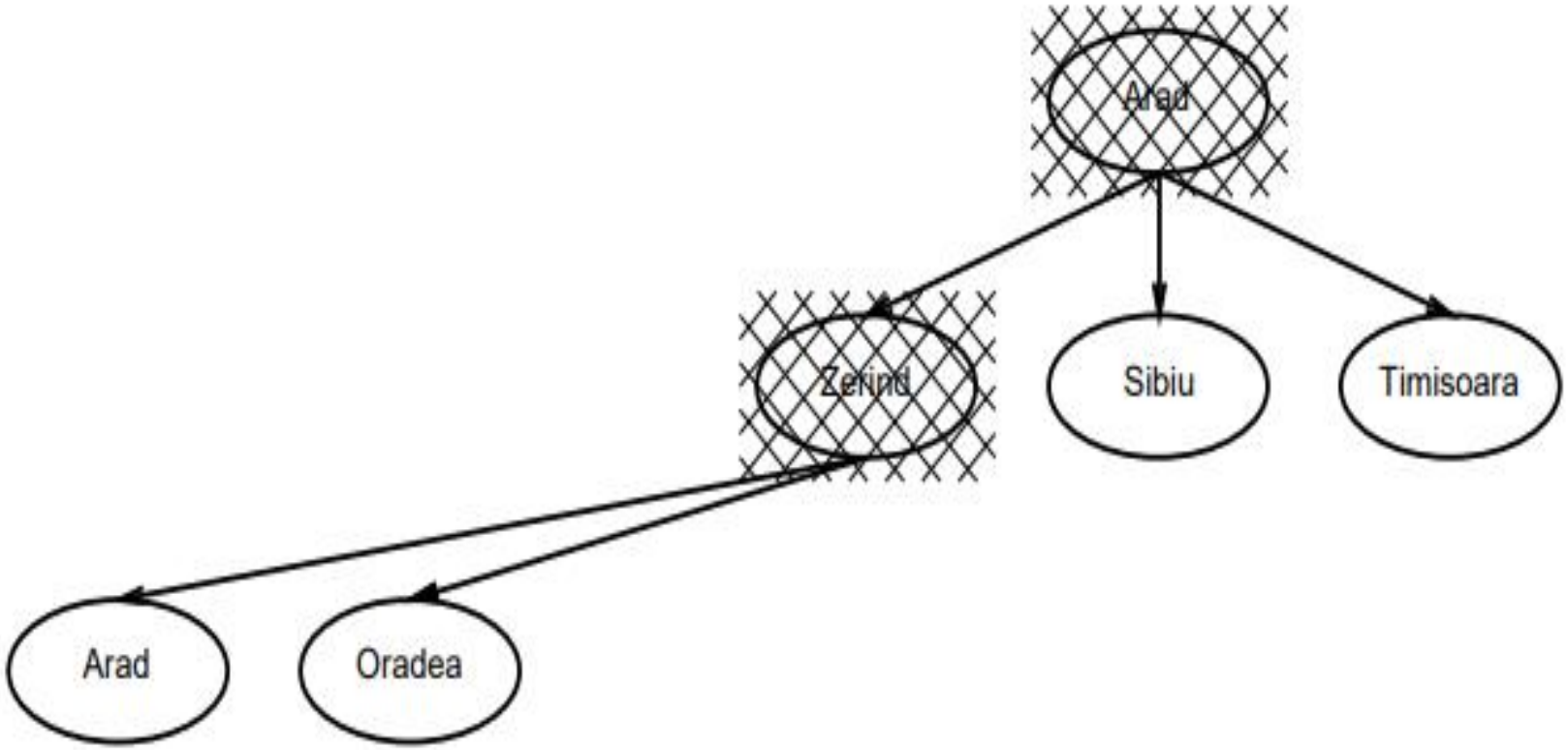


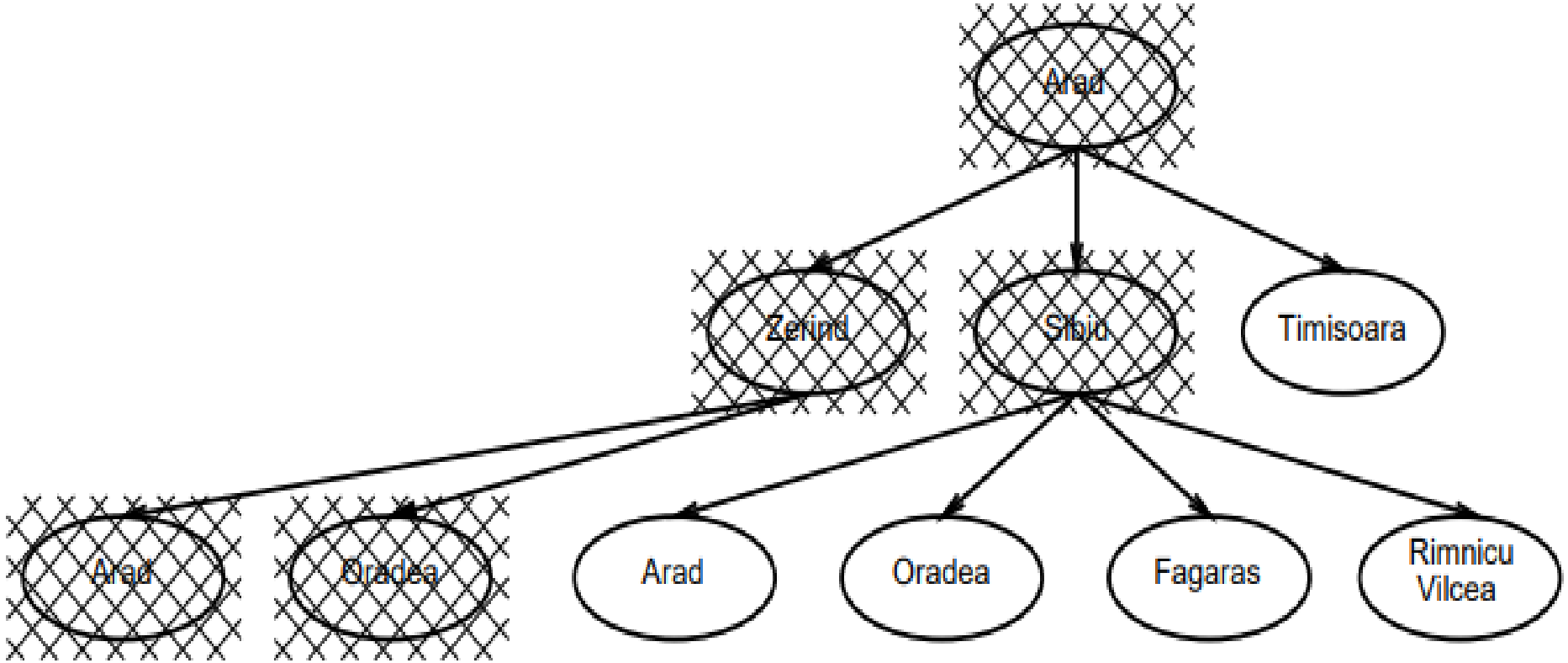

Arad
Zerind
Sibiu
Timisoara

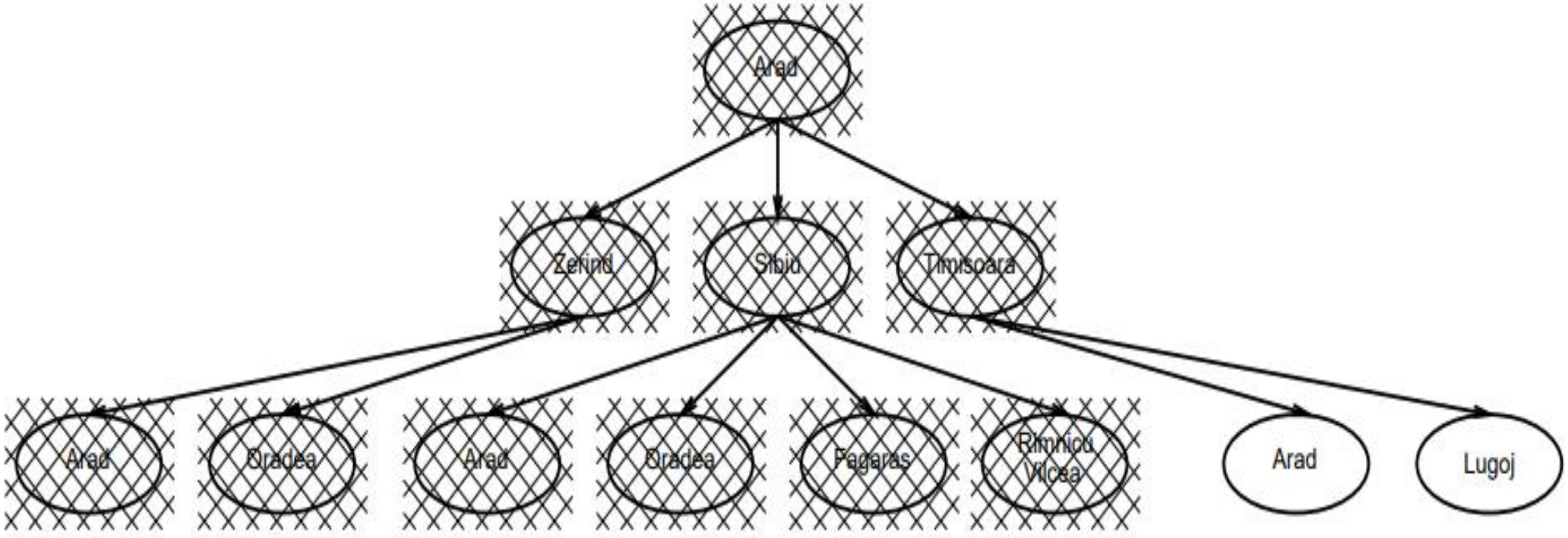
Iterative deepening search $l = 2$




Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Rimnicu Vilcea


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Rimnicu Vilcea
Arad
Lugoj

Properties of iterative deepening search

Complete??

Time??

Space??

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$

Can be modified to explore uniform-cost tree

Uninformed (Blind) Search

Uninformed (Blind) Search Strategies in AI

Uninformed search strategies use **only the information given in the problem definition**:

- Initial state
- Operators (actions)
- Goal test
- Path cost

They **do not use any heuristic knowledge** about how close a state is to the goal.

These strategies systematically explore the state space.

Uninformed (Blind) Search Strategies in AI

Uninformed search strategies use **only the information given in the problem definition**:

- Initial state
- Operators (actions)
- Goal test
- Path cost

"Uninformed search is about **how** to search, not **where** to search."

They **do not use any heuristic knowledge** about how close a state is to the goal.

These strategies systematically explore the state space.

Uninformed (Blind) Search Strategies in AI

- Best-first search
- A* search
- Heuristics
- Hill-climbing
- Simulated annealing

Best-first search

Idea: use an *evaluation function* for each node
– estimate of "desirability"

⇒ Expand most desirable unexpanded node

Implementation:
QUEUEINGFN = insert successors in decreasing order of desirability

Special cases:
greedy search
A* search

Greedy search

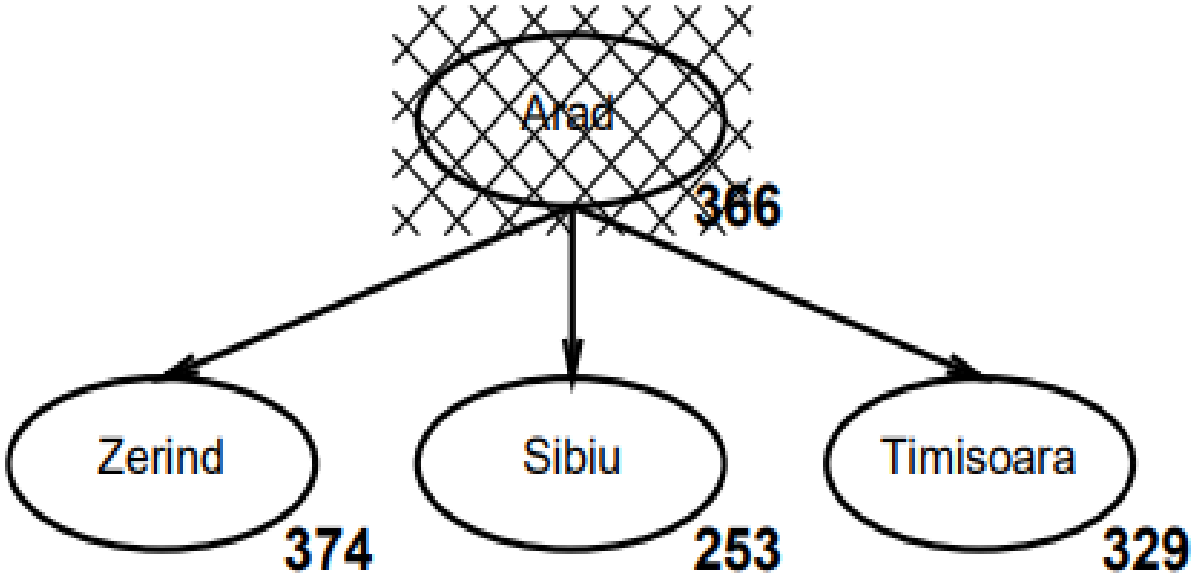
Evaluation function $h(n)$ (heuristic)

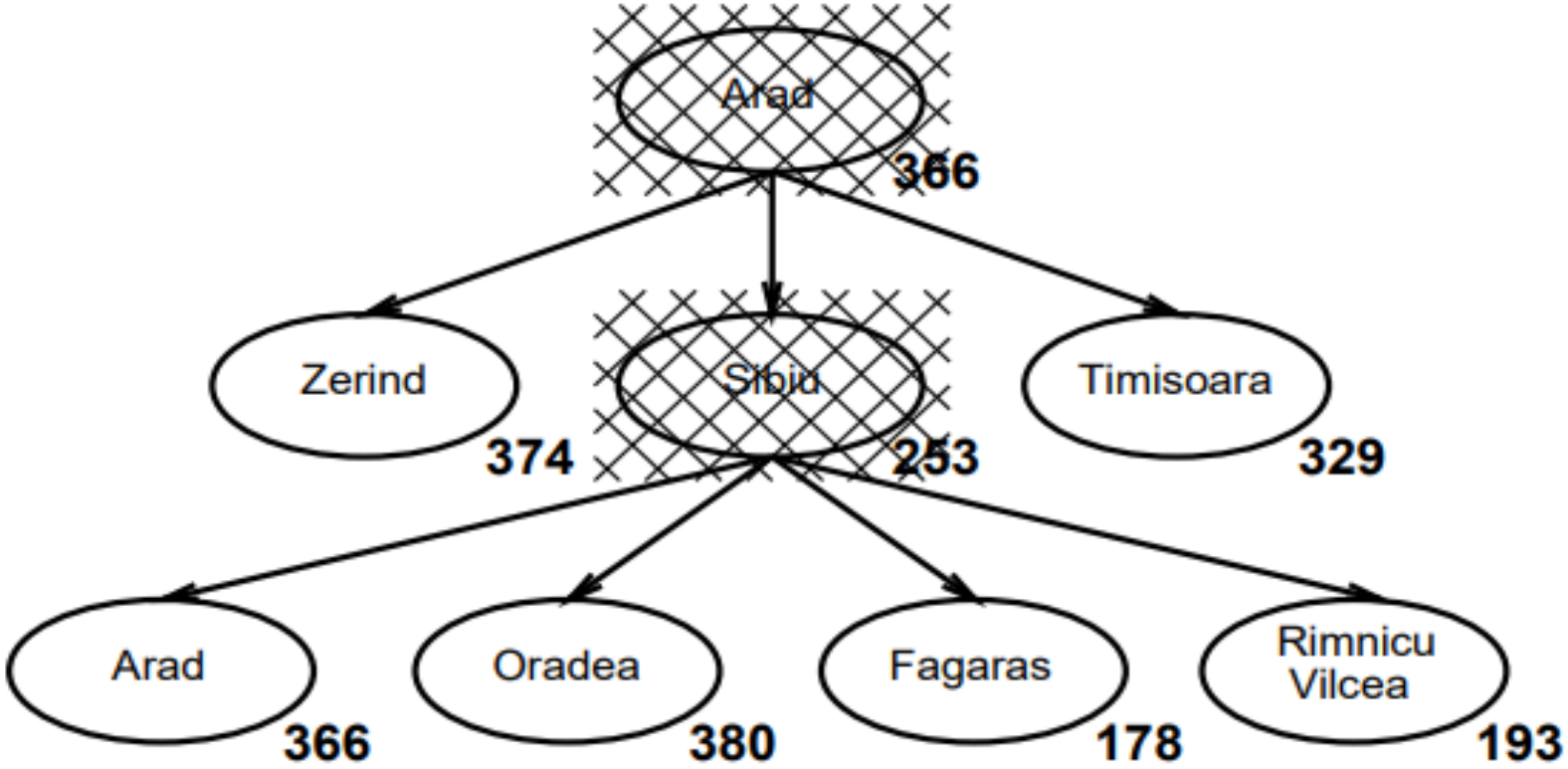
$=$ estimate of cost from n to $goal$

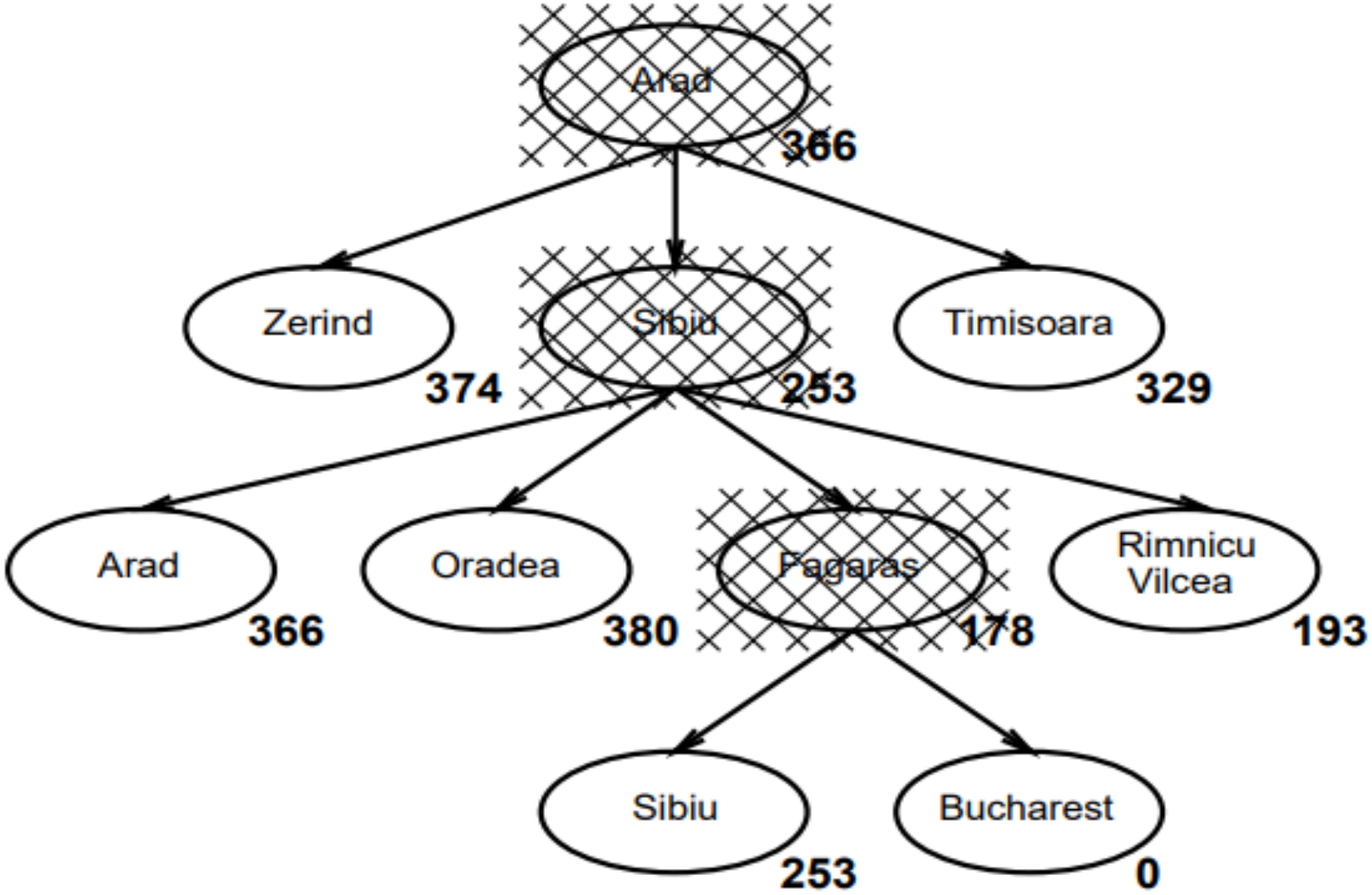
E.g., $h_{\mathrm{SLD}}(n) =$ straight-line distance from n to Bucharest

Greedy search expands the node that $appears$ to be closest to goal

Greedy search example


Arad
366
Zerind
374
Sibiu
253
Timisoara
329


Arad
366
Zerind
374
Sibiu
253
Timisoara
329
Arad
366
Oradea
380
Fagaras
178
Rimnicu
Vilcea
193


Arad
366
Zerind
374
Sibiu
253
Timisoara
329
Arad
366
Oradea
380
Fagaras
178
Rimnicu Vilcea
193
Sibiu
253
Bucharest
0

Properties of greedy search

Complete??

Time??

Space??

Optimal??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,

Iasi → Neamt → Iasi → Neamt →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$—keeps all nodes in memory

Optimal?? No

Informed (Heuristic) Search Strategies

Informed search strategy—one that uses domain-specific hints about the location of goals—can find solutions more efficiently than an uninformed strategy.

The hints come in the form of a heuristic function, denoted **h(n)**:

h(n) = estimated cost of the cheapest path from the state at node **n** to a **goal state**.

A* (A-Star) Search Algorithm

A* (A-Star) is an informed search algorithm used to find the **shortest path** from a start node to a goal node. It is widely used in: Route finding (like GPS), Robotics, Game AI, and Path planning problems.

A* combines:

- ❖ Uniform Cost Search (actual path cost) / Dijkstra's Algorithm
- ❖ Greedy Best First Search (heuristic estimate)

Key components

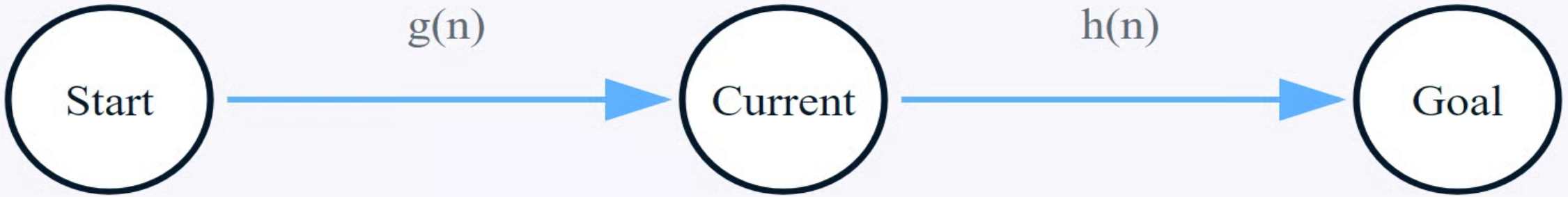
To understand A* algorithm, you need to be familiar with these fundamental concepts:

- **Nodes**: Points in your graph (like intersections on a map)
- **Edges**: Connections between nodes (like roads connecting intersections)
- **Path Cost**: The actual cost of moving from one node to another
- **Heuristic**: An estimated cost from any node to the goal
- **Search Space**: The collection of all possible paths to explore

Key Concepts in A* Search

The A* algorithm's efficiency comes from its smart evaluation of paths using three key components: **g(n), h(n), and f(n).** These components work together to guide the search process toward the most promising paths.

A* algorithm Cost Function: $f(n) = g(n) + h(n)$



https://www.datacamp.com/tutorial/a-star-algorithm

A* search

Idea: avoid expanding paths that are already expensive

Evaluation function $f(n) = g(n) + h(n)$

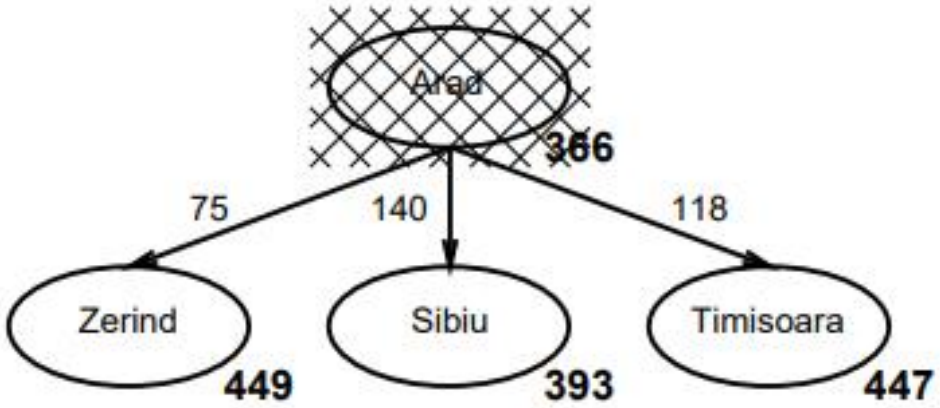
$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal

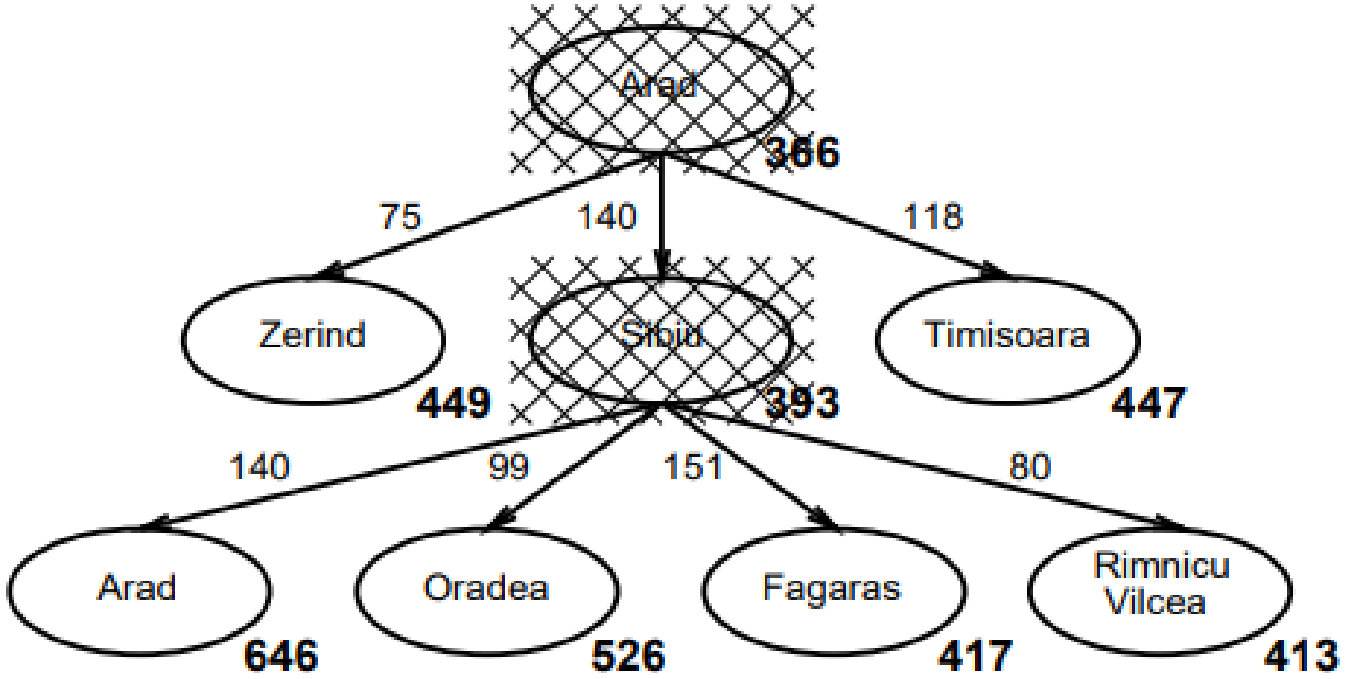
A* search uses an *admissible* heuristic
i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n.

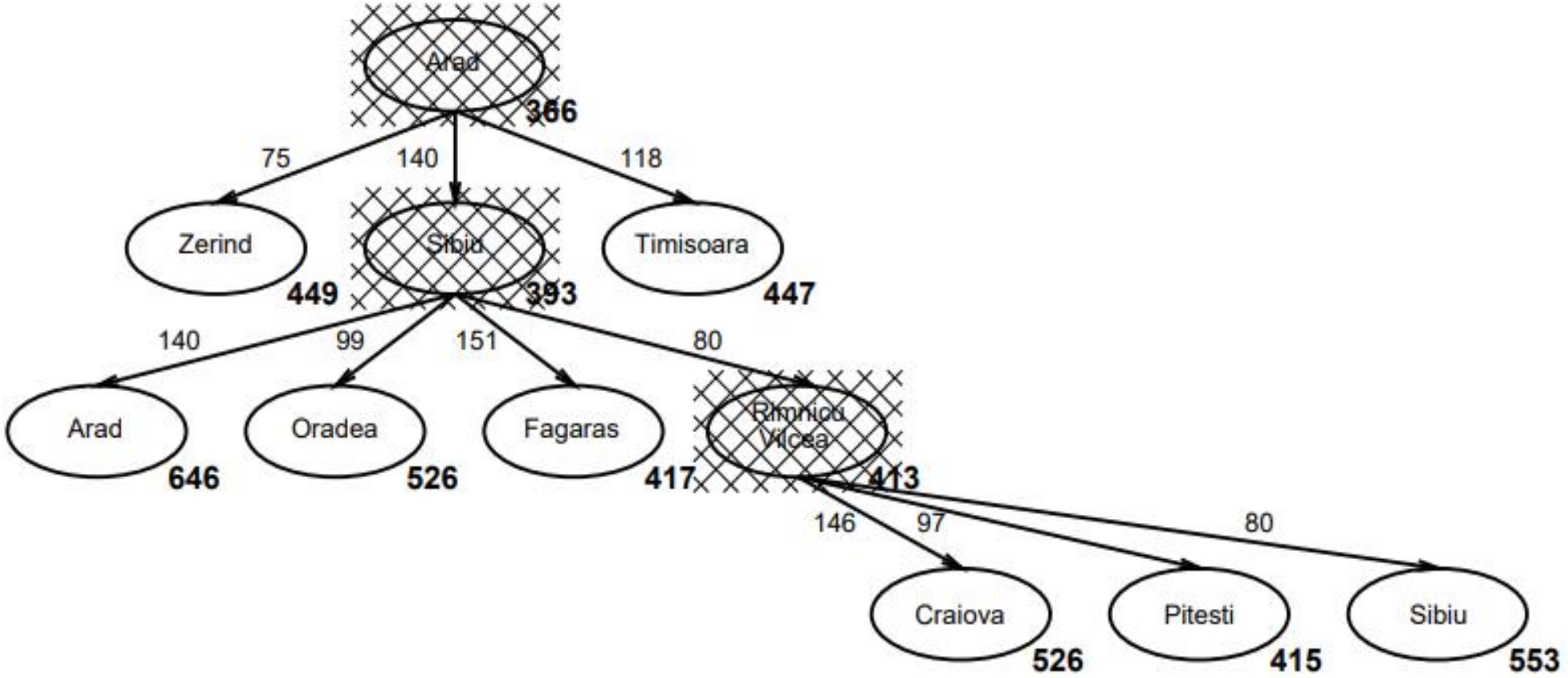
E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance

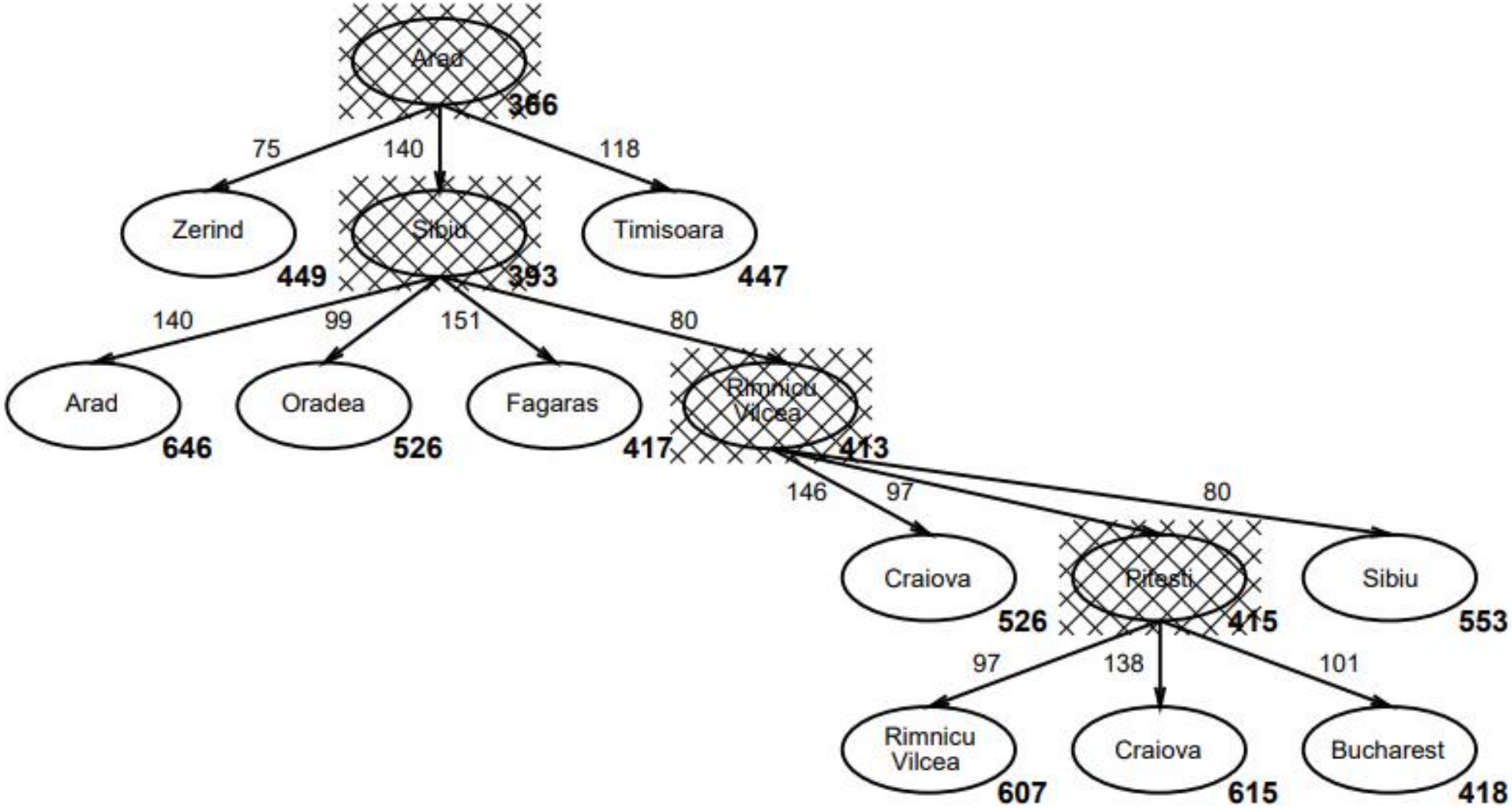
Theorem: A* search is optimal

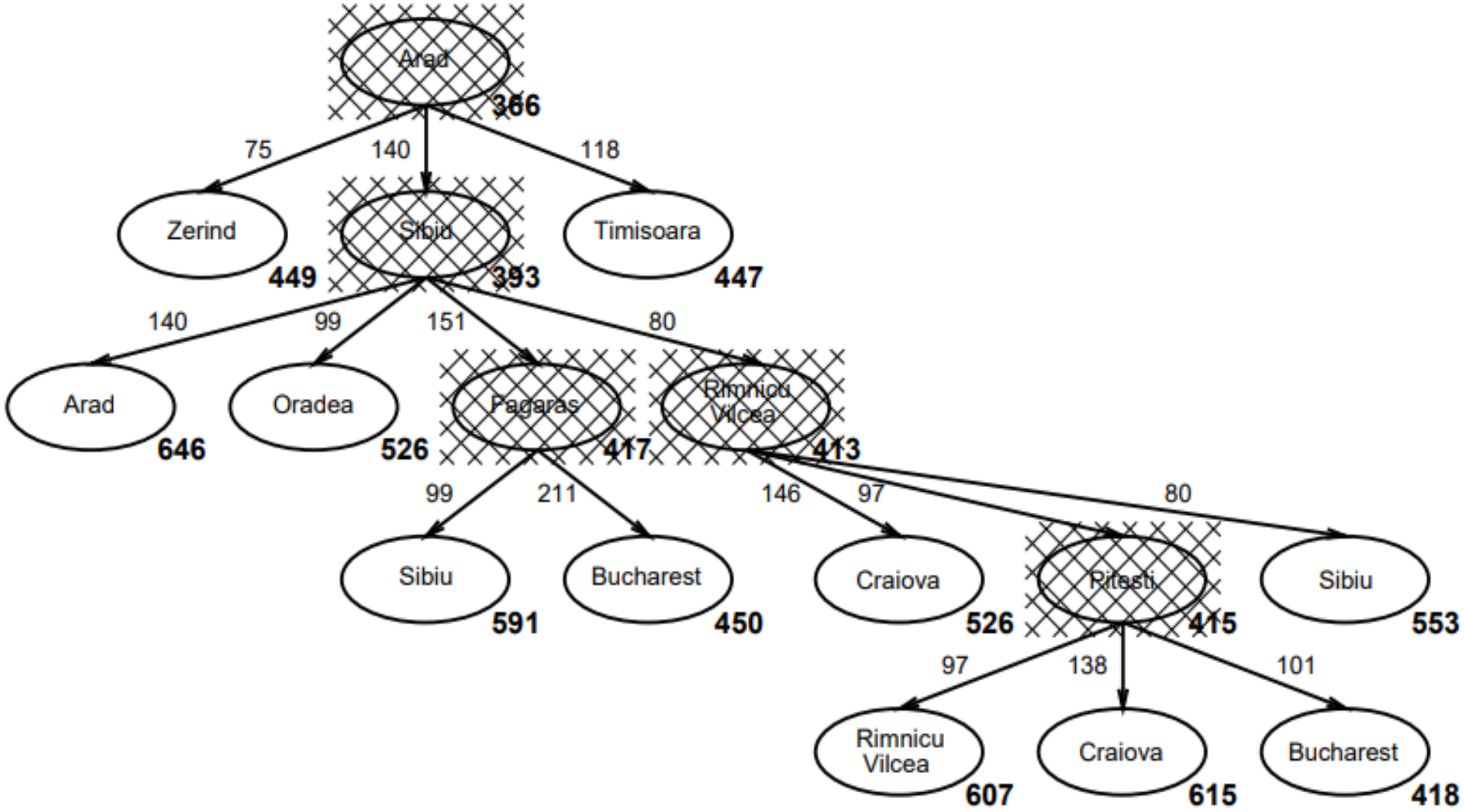
A* search example


Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447


Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447
140
99
151
80
Arad
646
Oradea
526
Fagaras
417
Rimnicu
Vilcea
413

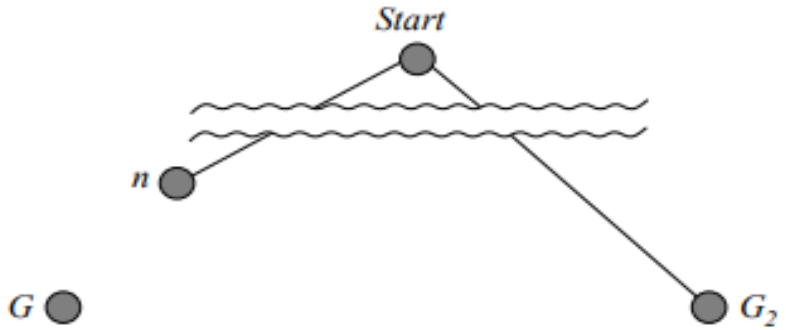

Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447
140
99
151
80
Arad
646
Oradea
526
Fagaras
417
Rimnicu Vilcea
413
146
97
80
Craiova
526
Pitesti
415
Sibiu
553


Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447
140
99
151
80
Arad
646
Oradea
526
Fagaras
417
Rimnicu Vilcea
413
146
97
80
Craiova
526
Pitesti
415
Sibiu
553
97
138
101
Rimnicu Vilcea
607
Craiova
615
Bucharest
418


Arad
366
75
140
118
Zerind
449
Sibiu
393
Timisoara
447
140
99
151
80
Arad
646
Oradea
526
Pagaras
417
Rimnicu Vilcea
413
99
211
146
97
80
Sibiu
591
Bucharest
450
Craiova
526
Pitesti
415
Sibiu
553
97
138
101
Rimnicu Vilcea
607
Craiova
615
Bucharest
418

Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1.



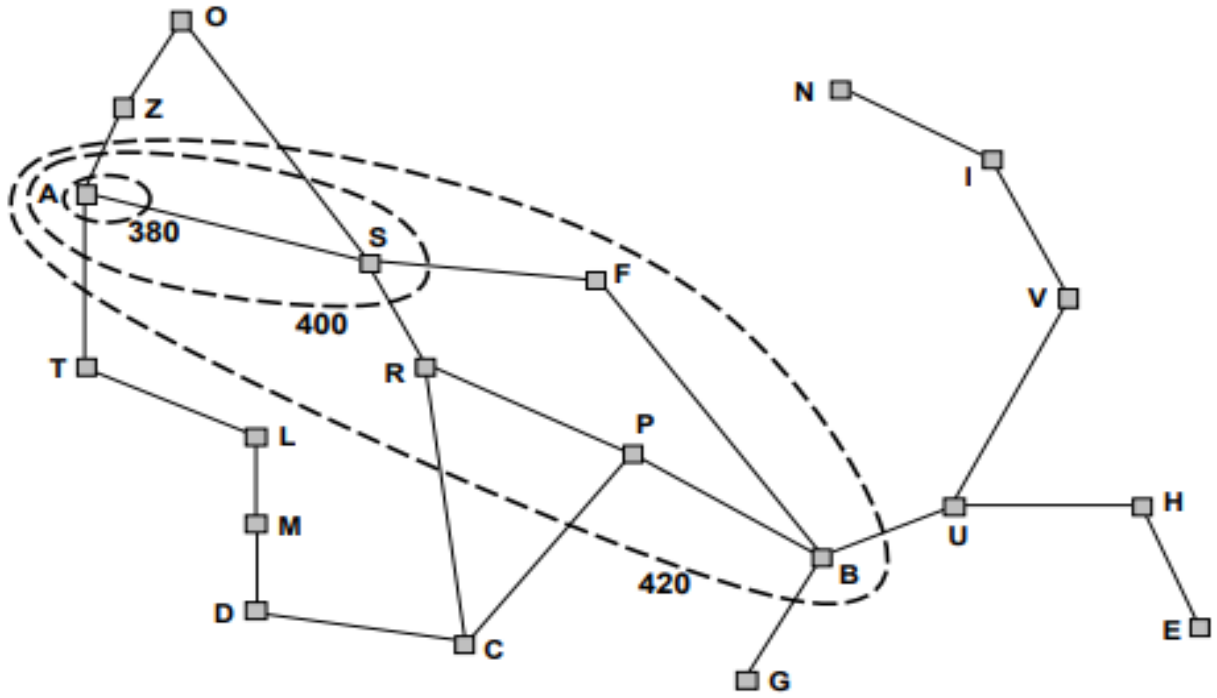
$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing f value

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Heuristics are functions:

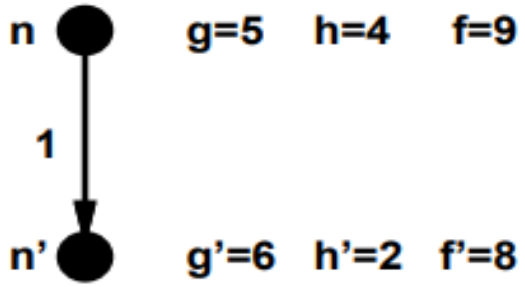
In **A* Search Algorithm**, **heuristics** are functions $h(n)$that estimate the cost from a node $n$to the goal. Different types of heuristics influence performance and optimality.

Here are the main **types of heuristics used in A***:

Proof of lemma: Pathmax

For some admissible heuristics, f may *decrease* along a path

E.g., suppose n' is a successor of n



But this throws away information!
$f(n) = 9 \Rightarrow$ true cost of a path through n is ≥ 9
Hence true cost of a path through n' is ≥ 9 also

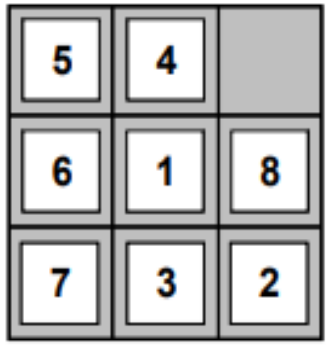
Pathmax modification to A*:
Instead of $f(n') = g(n') + h(n')$, use $f(n') = max(g(n') + h(n'), f(n))$

With pathmax, f is always nondecreasing along any path

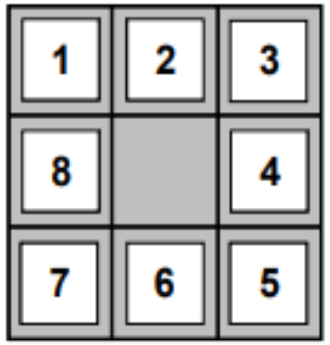
Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



Start State



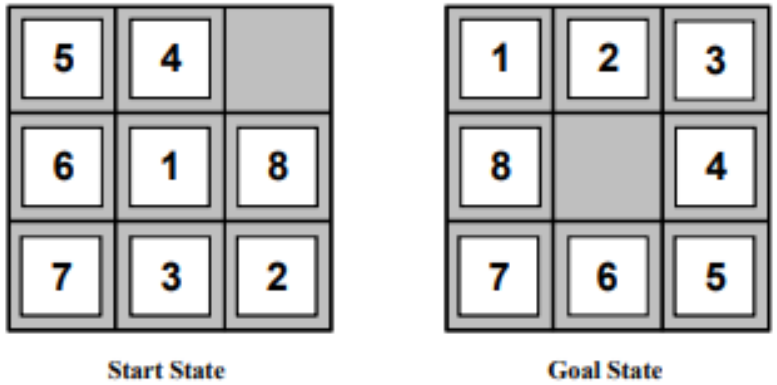
Goal State

$h_1(S)$ =??
$h_2(S)$ =??

Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



$h_1(S)$ =?? 7
$h_2(S)$ =?? 2+3+3+2+4+2+0+2 = 18

1. Admissible Heuristic

- **Definition:** Never overestimates the true cost to reach the goal.
- Condition:

$$h(n) \leq h^*(n)$$

- where $h^*(n)$is the actual cost.
- **Property:** Guarantees optimal solution in A*.

Example:

- Straight-line distance in maps.

2. Consistent (Monotonic) Heuristic

- **Definition:** Satisfies triangle inequality.
- Condition:

$$h(n) \leq c(n, n') + h(n')$$

- **Property:** Ensures no node is re-expanded; improves efficiency.

Note:

All consistent heuristics are admissible.

3. Inadmissible Heuristic

- **Definition:** May overestimate the true cost.
- **Property:** Faster but **may not give optimal solution**.

Use case:

- When speed is more important than accuracy.

4. Exact Heuristic

- **Definition:** Equals the actual cost to reach the goal.

$$h(n) = h^*(n)$$

- **Property:** A* becomes extremely efficient (like direct path).

Limitation:

Usually impossible to compute in real problems.

5. Dominant Heuristic

• **Definition:** One heuristic is better than another if it gives higher values without overestimating.

• If:

$$h_1(n) \geq h_2(n)$$

• and both are admissible → h_1 is better.

Property:

Expands fewer nodes → more efficient.

6. Common Practical Heuristics (Distance-Based)

(a) Manhattan Distance

• Used in grid-based movement (4 directions)

$$| x_1 - x_2 | + | y_1 - y_2 |$$

b) Euclidean Distance

• Straight-line distance

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

c) Chebyshev Distance

• Used when diagonal movement allowed, **max(|x1−x2|,|y1−y2|)**

Hill-climbing (or gradient ascent/descent)

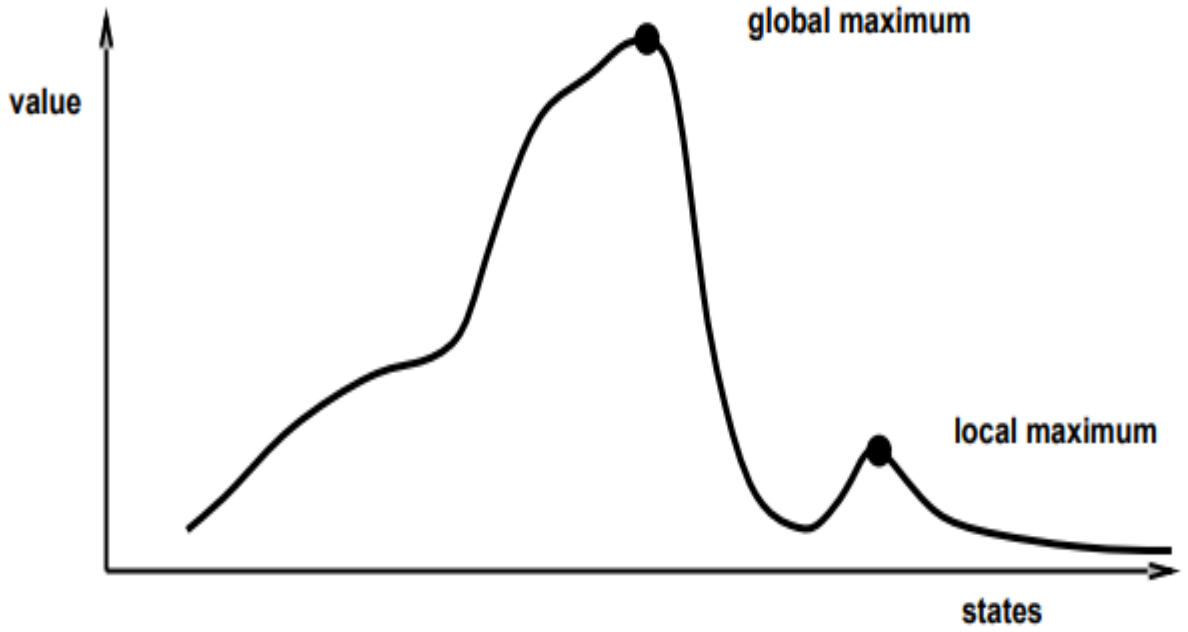
"Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING( problem) returns a solution state
    inputs: problem, a problem
    local variables: current, a node
                     next, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        next ← a highest-valued successor of current
        if VALUE[next] < VALUE[current] then return current
        current ← next
    end
```

Hill-climbing contd.

Problem: depending on initial state, can get stuck on local maxima



Simulated annealing

Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their size and frequency

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if $T=0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

Properties of simulated annealing

At fixed "temperature" T, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

T decreased slowly enough $\Longrightarrow$ always reach best state

Is this necessarily an interesting guarantee??

Devised by Metropolis et al., 1953, for physical process modelling

Widely used in VLSI layout, airline scheduling, etc.