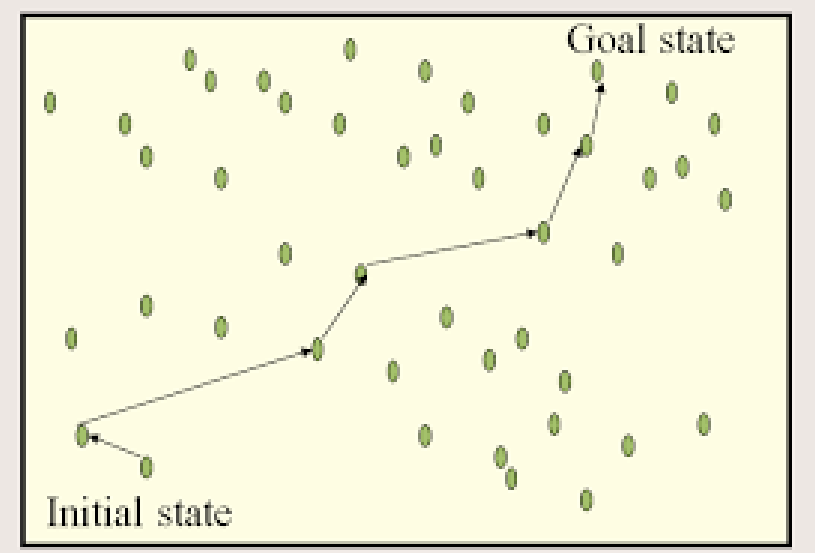
Introduction to Artificial Intelligence

Subject 4: Solving Problems by Searching

Dr. Mohammed Shambour

Problem-Solving Agent

A problem-solving agent is an AI system that plans ahead — it searches through possible sequences of actions to find a path from an initial state to a goal state.



Why We Need Problem-Solving Agents

Some intelligent agents can act **immediately** based on percepts — for example, a thermostat turning the heater on when the room is cold.

But in many real-world situations:

- There's **no single action** that reaches the goal.
- The agent must think **several steps ahead**.
- The agent needs to find a **plan** — a *sequence* of actions.

For example:

A robot that wants to reach a charging station must plan a path around obstacles — not just move forward randomly.

That's what makes it a **problem-solving agent**.

How It Works (The Process)

A problem-solving agent works in **four main phases**:

Step 1: Goal Formulation

The agent first defines its goal—the desired state of the world it wants to reach (e.g., "be in city A").

Step 2: Problem Formulation (State Space Representations**)**

The agent defines:

- The **initial state** (where it starts),
- The **goal state** (what it wants to achieve),
- The **possible actions** (what it can do),
- The **transition model** (what happens when it performs an action),
- The **path cost** (how to evaluate or compare different action sequences).

Step 3: Search

The agent uses **search algorithms** to explore possible **sequences of actions** that can lead from the initial state to the goal state.

Common algorithms:

- Uninformed search (e.g., Breadth-First, Depth-First)
- Informed search (e.g., A*, Greedy Best-First)

Step 4: Execute Solution

Once it finds a path (plan), it executes the actions one by one to reach the goal.

Example: Automated Taxi Driver as a Problem-Solving Agent

Step 1. Goal Formulation

Be at the Airport.

Step 2. Problem Formulation

- ❑ **Initial state:**
  - Taxi at Main Street, passenger waiting at City Center.
- ❑ **Goal state:**
  - Passenger dropped off at Airport.
- ❑ **Possible actions:**
  - Drive north/south/east/west, pick up passenger, drop off passenger.
- ❑ **Transition actions:**
  - Describes how the taxi's location changes after each move.
- ❑ **Path cost:**
  - The action sequences can be evaluated by a predefined measurement such as: time consuming, distance, and energy.

Step 3. Search

The agent searches for the best route (among sequences of actions) that leads from the current location to the destination (airport) using a suitable search algorithm.

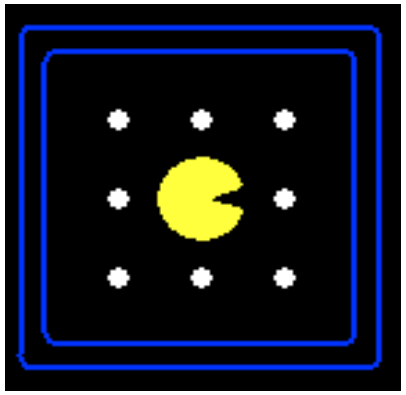
Step 4. Execute Solution

The agent performs the sequence of actions (e.g., Drive East, Pick up passenger, Drive West then North, Drop off passenger) until the Goal Test is satisfied, thus achieving the goal.

Example: Pac-Man: Concrete Search Problem

Step 1. Goal Formulation

To find and eat the nearest large power pellet.



Step 2. Problem Formulation

- ❑ **Initial state:**
  - Pac-Man is at grid coordinate (X=1, Y=3).
- ❑ **Goal state:**
  - Number of remaining dots is zero.
- ❑ **Possible actions:**
  - Move Up, Move Down, Move Left, Move Right.
- ❑ **Transition actions:**
  - If Pac-Man is at (1, 3) and performs Move_Right, the new state is (2, 3).
- ❑ **Path cost:**
  - Each movement action (Move Right, Move Up, etc.) has a cost of 1.

Step 3. Search

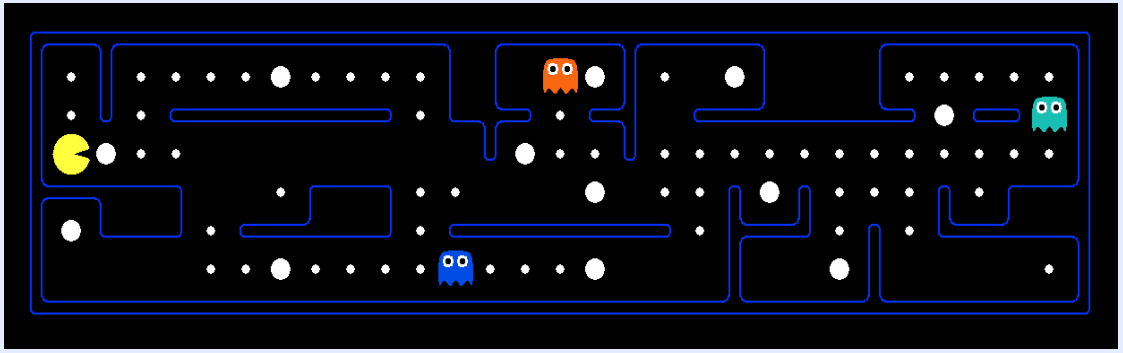
- The agent uses an algorithm (e.g., BFS) to systematically explore the maze.
- It identifies the sequence of actions that leads to the power pellet with the minimum total Path Cost.

Step 4. Execute Solution

The agent performs the sequence of actions (e.g., Move_Right, Move_Left, Move_Left, Move_Up, Move_Down, etc.) until the Goal Test is satisfied, thus achieving the goal.

Pac-Man Example: What's in a State Space?

The world state includes every last detail of the environment



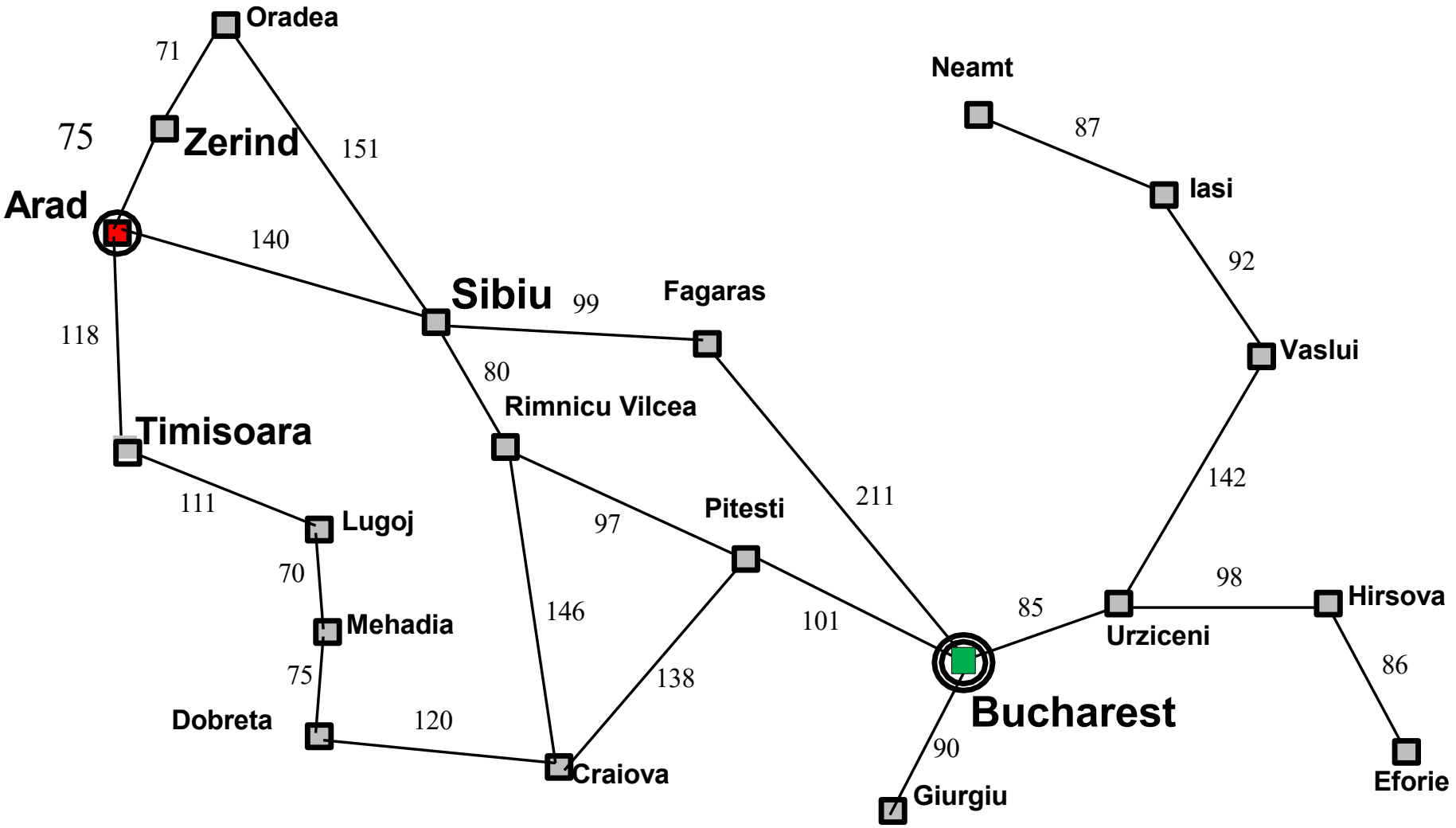
A search state keeps only the details needed for planning (abstraction)

Problem: Pathing	Problem: Eat-All-Dots
• States: (x,y) location • Actions: NSEW • Successor: update location only • Goal test: is (x,y)=END	• States: {(x,y), dot Booleans • Actions: NSEW • Successor: update location and possibly a dot boolean • Goal test: dots all false

World state vs Search state

Feature	World State	Search State
Definition	Complete real configuration of the environment	Simplified or abstracted representation used for problem solving
Who uses it?	The environment (objective reality)	The agent (for search/planning)
Visibility	May include hidden or unknown information that the agent cannot fully perceive	Includes only known or relevant information needed for decision-making.
Example (8-puzzle)	The actual physical or visual arrangement of all tiles on the 3×3 board.	The state representation used in the algorithm (e.g., storing the tile order as numbers in a list or array)
Example (Robot)	The full environment: detailed house map, positions of all furniture, people, dirt, and the robot, plus time and battery information.	The simplified model used in planning: the robot's current position, which rooms are clean or dirty, and its remaining battery level.

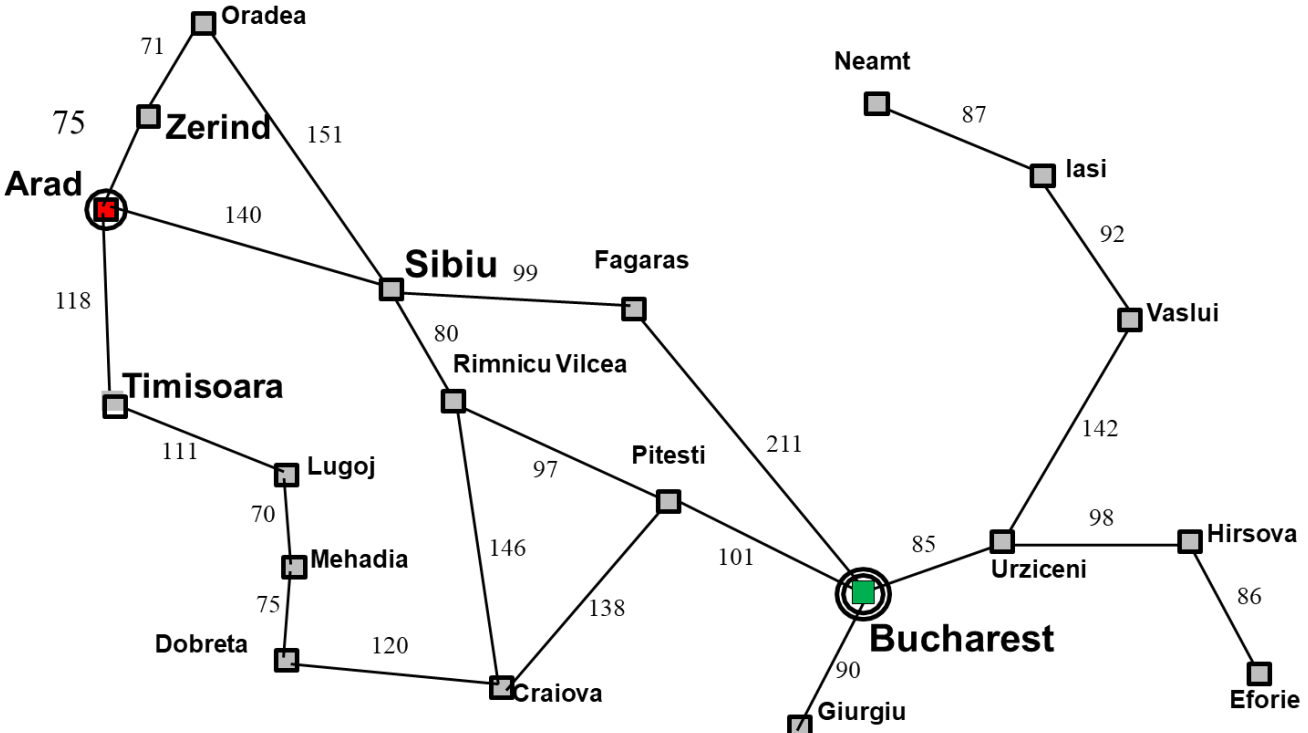
classic example of how a problem-solving agent works in AI
Example: Romania



classic example of how a problem-solving agent works in AI
Example: Romania

1. Scenario Overview

- You (the agent) are **on holiday in Romania**, currently in **Arad**. Your **goal** is to reach **Bucharest** (to catch your flight).
- The agent must **plan a route** — it cannot just perform one single action to get there. It must **search** for a *sequence of cities* to pass through.



classic example of how a problem-solving agent works in AI Example: Romania

2. Step-by-Step Explanation

Step 1: Goal Formulation

- The goal is **to be in Bucharest**.
- This is the *destination* or the *goal state*.

Step 2: Problem Formulation

To solve the problem, the agent must define:

- **States:** each city (e.g., Arad, Sibiu, Fagaras, Bucharest).
- **Actions:** driving between connected cities (edges on the map).
- **Transition model:** what happens after an action — e.g., if the agent drives from Arad to Sibiu, the next state becomes "in Sibiu."
- **Goal test:** check if the agent is in Bucharest.
- **Path cost:** total distance traveled (the numbers on the map are distances in KM).

So, the problem can be formally stated as:

Find a sequence of cities (a path) from Arad to Bucharest with minimal cost (distance).

classic example of how a problem-solving agent works in AI
Example: Romania

3. Search for a Solution

This is where **search algorithms** come in.
The agent explores possible routes such as:

- Arad → Sibiu → Fagaras → Bucharest
- Arad → Timisoara → Lugoj → Mehadia → Dobreta → Craiova → Pitesti → Bucharest
- Arad → Zerind → Oradea → Sibiu → Fagaras → Bucharest

The **search algorithm** (like Breadth-First, Uniform Cost, or A*) will evaluate which path leads to the goal efficiently (lowest cost).

So, the **search** is the process of building and exploring a **search tree** — starting from Arad, generating possible next states (cities), and expanding them until Bucharest is found.

classic example of how a problem-solving agent works in AI Example: Romania

4. Execution

Once the agent finds a solution (a sequence of actions or path), it executes them:
If the found path is:

- Arad → Sibiu → Fagaras → Bucharest

Then the **actions** are:

- Drive from Arad to Sibiu
- Drive from Sibiu to Fagaras
- Drive from Fagaras to Bucharest

Example of ACTIONS(Arad):
ACTIONS(Arad) = {ToSibiu, ToTimisoara, ToZerind}
That means — from Arad, the agent can choose one of these three roads as its next move.

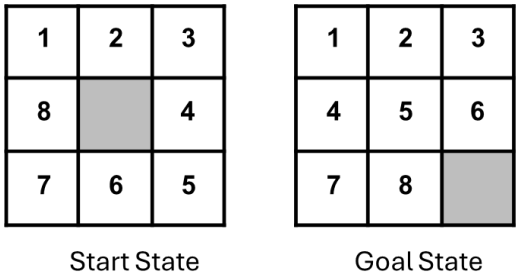
Single-state problem formulation

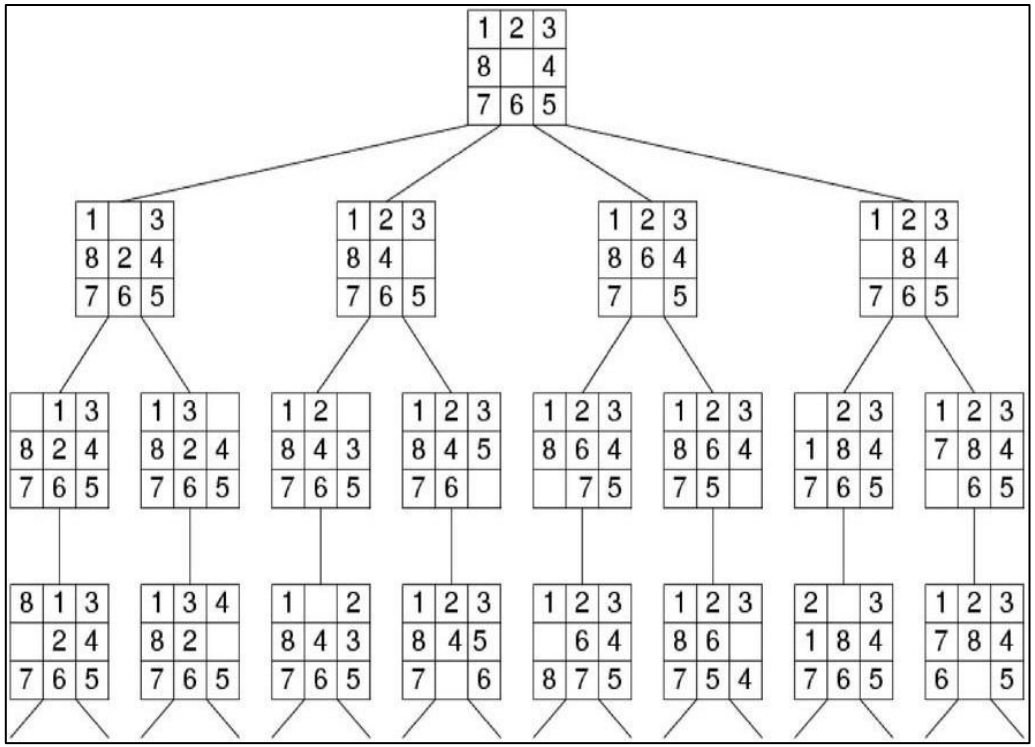
- A problem is defined by five items:

1. initial state e.g., "at Arad"
2. actions: Actions(s) returns applicabile actions in s: (Go(Sibiu), Go(Timisoara), Go(Zerind)
3. transition model - set of action–state pairs (succesor function Result(s,a)): e.g., Result(In(Arad), Go(Zerind))=In(Zerind)
4. goal test - determines if whether a given state is a goal state
   explicit, e.g., *xs* = "at Bucharest"
   implicit, e.g., *xs* = checkmate
5. path cost (additive) - reflects agent's own performance measure
   e.g., sum of distances, number of actions executed, etc.
   $c(s, a, s')$ is the step cost, assumed to be ≥ 0

- A solution is a sequence of actions leading from the initial state to a goal state

Problem Formulation Example: 8-puzzle





1. Initial State :
   - Any permutation of 1-8 numbers with a blank
2. Actions :
   - Movement of blank space either by Left, Right, Up or Down
3. Transition Model :
   - The resulting state after moving the blank space will replace the digit
4. Goal Test :
   - Check whether the state matches the goal configuration
5. Path cost :
   - Each step costs 1, so the path cost is total number of steps

Outline

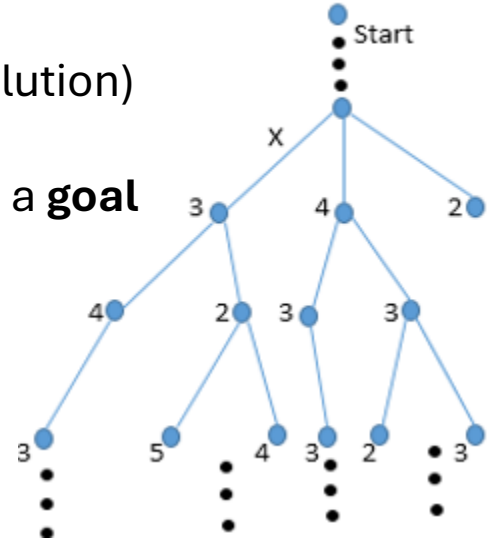
- Problem-solving agents
- Problem formulation
- **Uninformed search algorithms**
- Informed (heuristic) search strategies

Tree search algorithms

A tree search algorithm is a method used to explore possible sequences of actions — represented as a tree structure — in order to reach a goal state.

The Basic Idea

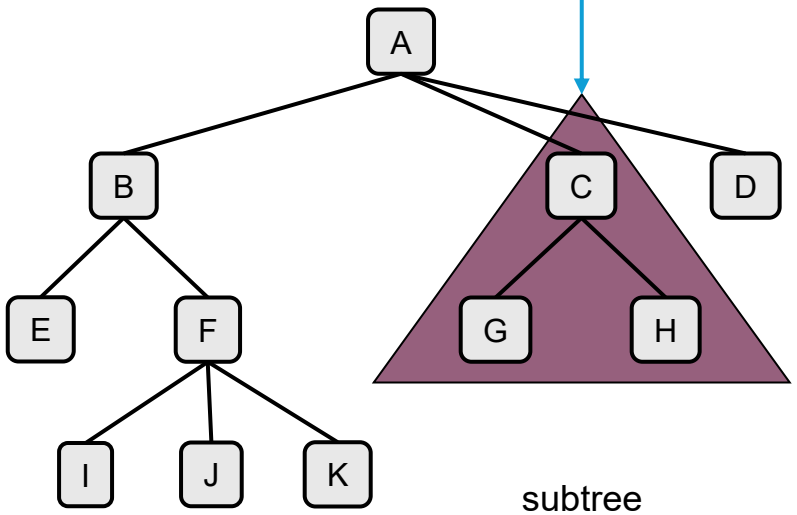
1. Start from the **initial state** (the root of the tree).
2. E**xpand** the current node → generate all possible **next states** (successors).
3. For each successor, check:
   - Is it the **goal state**? (then stop — you found a solution)
   - If not, expand it further → keep growing the tree.
4. Continue until you find a **path** from the initial state to a **goal state**.



Tree Terminology

- **Root**: node without parent (A)
- **Internal node**: node with at least one child (A,B, C, F)
- **External node** (a.k.a. **leaf**): node without children (E, I, J, K, G, H, D)
- **Ancestors of a node**: parent, grandparent, grand-grandparent, etc. → the ancestors of node J are F, B, and A
- **Depth of a node**: number of ancestors → the depth of the node K is 3
- **Height of a tree**: maximum depth of any node → in this tree the height is (3)
- **Sibling** Nodes in the tree that are children of the same parent are said to be siblings. For example, B → C → D
- **Path**: A path is an ordered list of nodes that are connected by edges. For example, B → F → k

- **Subtree**: tree consisting of a node and its descendants → example for a subtree of size 3

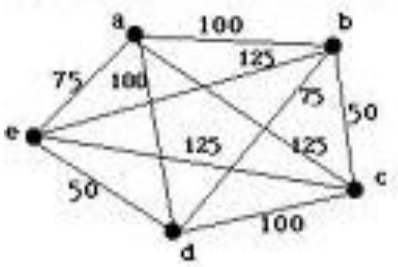


Relation Between State Space and Search Tree

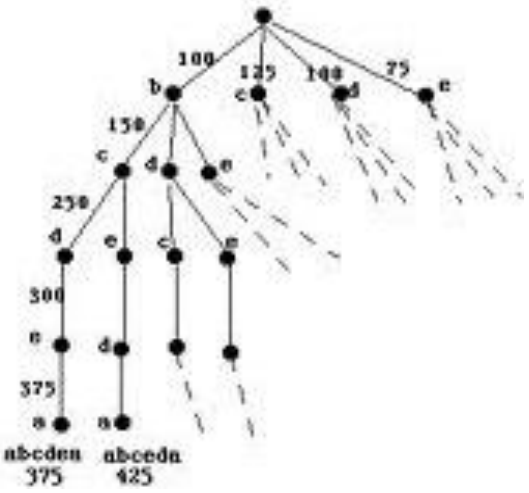
To understand this, we must distinguish between two key concepts:

Concept	Description
State space	All possible configurations (states) of the environment. It's like a map of every possible situation.
Search tree	The structure built by the agent while exploring the state space. It's how the agent "searches" possible paths to reach the goal.

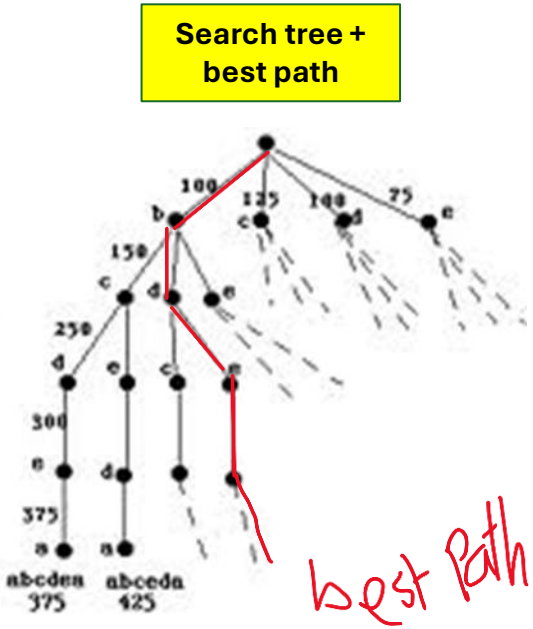
State space (all possible connections)



Search tree (agent's exploration)



Search tree + best path

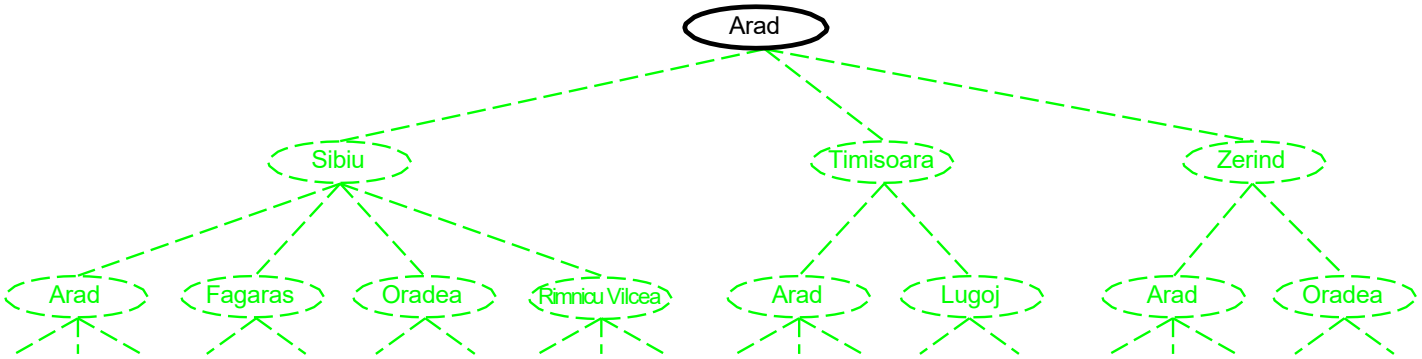


Example (Romania Map Problem)

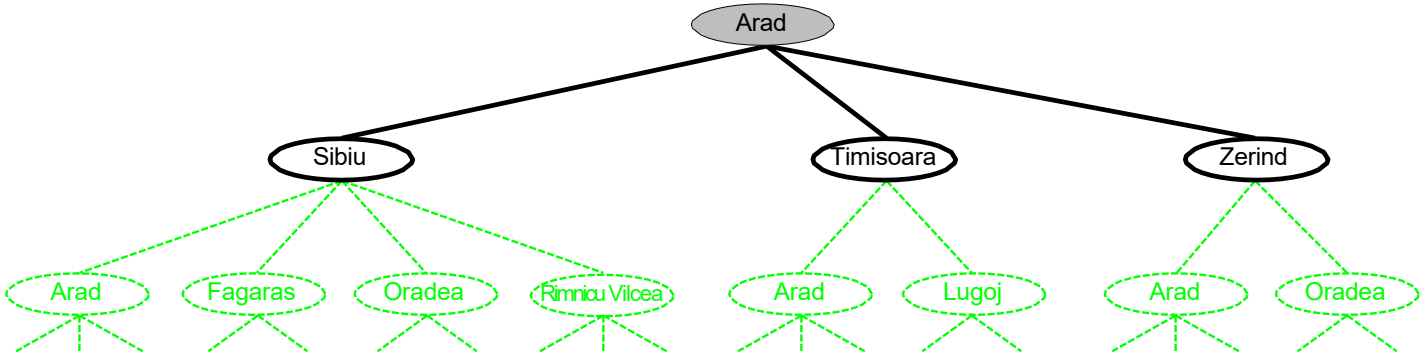
Imagine the **agent is in Arad** and wants to reach **Bucharest**.

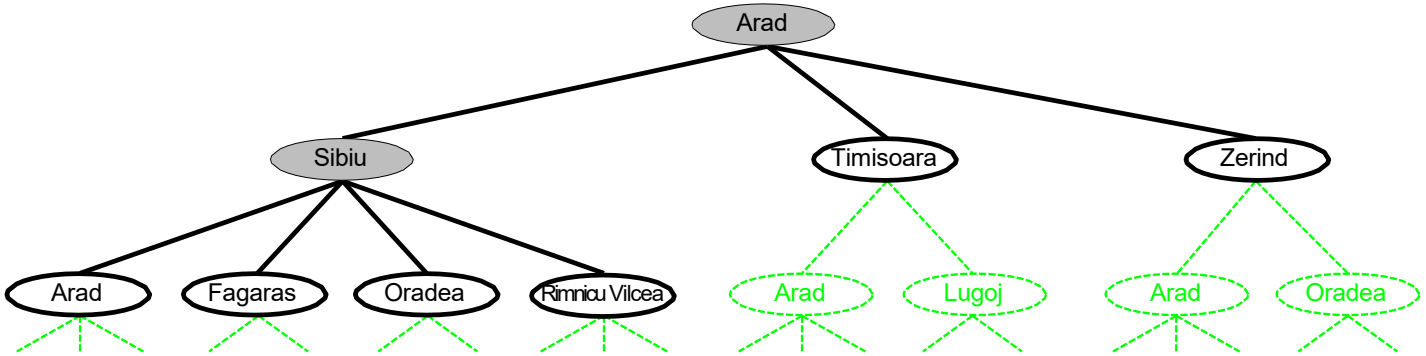
- **Initial state:** Arad
- **Goal state:** Bucharest
- **Actions:** Drive to neighboring cities

- ❑ The **state space** (Romania map) shows all cities and roads.
- ❑ The **search tree** begins with "Arad" as the root.
- ❑ Each node is a state (city), and each edge is an action (driving).



Tree search example





.....

.....

.....

.....

Search strategies

A **search strategy** defines *how* the agent chooses which node to expand next — that is, the **order of node expansion**.

- **Evaluation Criteria**

  Search strategies are compared using these key dimensions:

  - **Completeness:**
    Does the strategy always find a solution if one exists?
  - **Time Complexity:**
    How many nodes are generated or expanded?
  - **Space Complexity:**
    What is the maximum number of nodes stored in memory at any time?
  - **Optimality:**
    Does it always find the least-cost (best) solution?

- **Cost Measures**

  - **Search Cost:** The amount of time taken by search.
  - **Solution Cost:** The total length of the found path.
  - **Total Cost = Search Cost + Solution Cost**

Families of Search Algorithms in AI

Search algorithms in Artificial Intelligence are broadly divided into two main families (or categories):

1. Uninformed Search Family (also called Blind Search)

- These algorithms **do not use any heuristic** or problem-specific knowledge.
- They only rely on **the structure of the problem itself** — for example, the start state, goal test, and successor function.
- **Example**: BFS expands nodes in increasing order of their depth — it doesn't know which node is closer to the goal; it just tries all nodes level by level.

2. Informed Search Family (Heuristic Search)

- These algorithms use **heuristic functions** (denoted as h(n)) that **estimate the cost or distance** from the current state n to the goal.
- **Example:** A* uses the function f(n) = g(n) + h(n) where:
  g(n) = cost from start to node n
  h(n) = estimated cost from n to goal
  It expands nodes that seem *closest to the goal* based on these costs.

Families of Search Algorithms in AI

- **Key Characteristics of Uninformed Search Family**
  - They have **no information** about how far a state is from the goal.
  - They explore the search space **systematically** (e.g., by levels or costs).
  - They **guarantee** completeness and sometimes **optimality** if costs are uniform.
  - Usually have **high time and space complexity**.

- **Key Characteristics of Informed Search Family**
  - They are **goal-directed** (guided by heuristic knowledge).
  - Use **domain-specific information** to decide which node to explore next.
  - Usually **faster** and more **efficient** than uninformed searches.
  - May or may **not guarantee** optimality (depends on the heuristic and algorithm).

Uninformed search strategies

Uninformed search strategies are methods that **do not use any domain-specific knowledge**. They explore the search space using **only the information given in the problem definition**, such as:

- The initial state
- The goal test
- The available actions (successors)
- The cost of actions (if any)

These strategies don't "know" which direction leads to the goal — they **blindly** explore all possible paths.

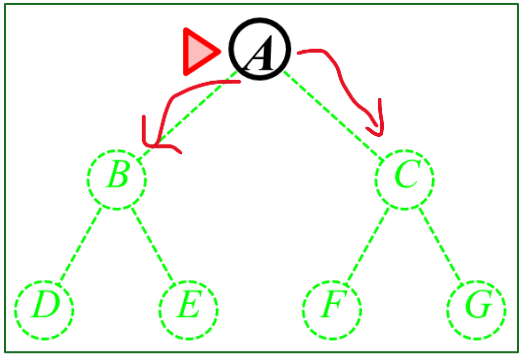
Uninformed Algorithm Examples:

A. **Breadth-first search**
B. **Uniform-cost search**
C. **Depth-first search**
D. **Depth-limited search**
E. **Iterative deepening search**

A. Breadth-first Search (BFS)

🧩 **How it works:**

- Expands the shallowest (closest to the root) node first.
- Explores all nodes at depth 1 before moving to depth 2, and so on.



🧠 **Example (Romania Map):**

From **Arad**, BFS first explores:

Arad → {Zerind, Timisoara, Sibiu}

Then all their neighbors and continues until reaching **Bucharest**.

✅ **Pros:**

- **Complete:** Will find a solution if one exists.
- **Optimal:** Finds the **shortest path** (in number of steps).

⚠️ **Cons:**

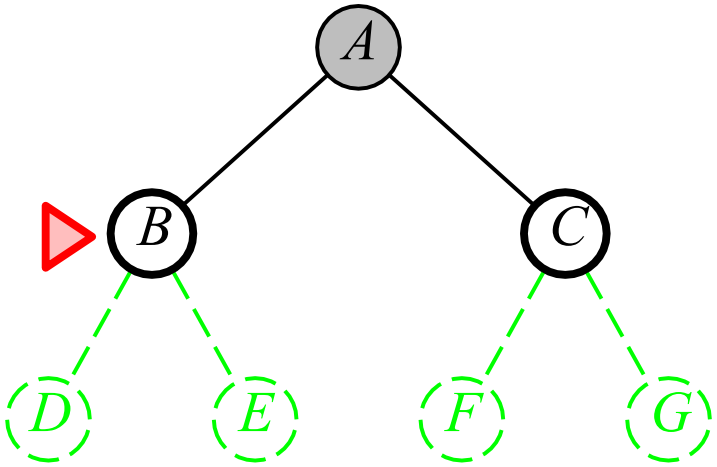
- **High memory and time usage:** $O\left(b^{d+1}\right)$, where b = branching factor (average number of successors per node), d = depth of solution (from root to goal).

A. Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



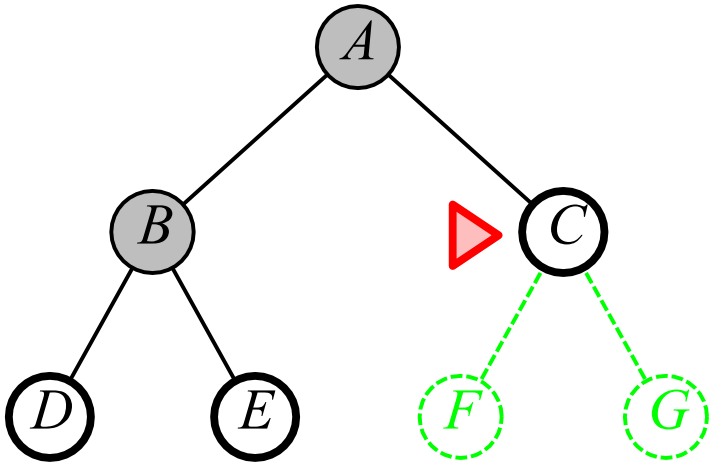
FIFO means
First In First Out

A. Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

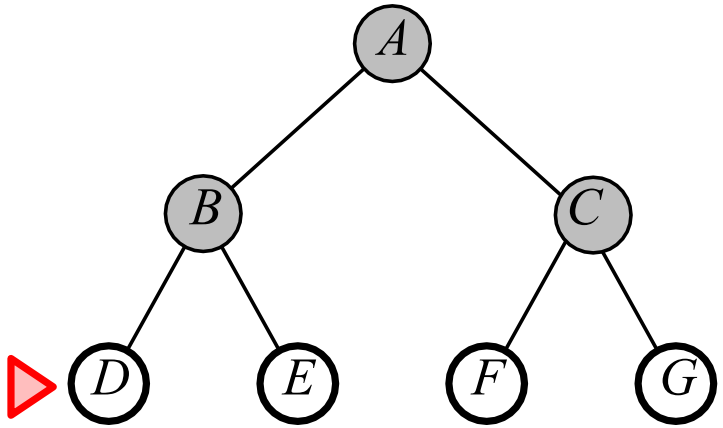


A. Breadth-first search

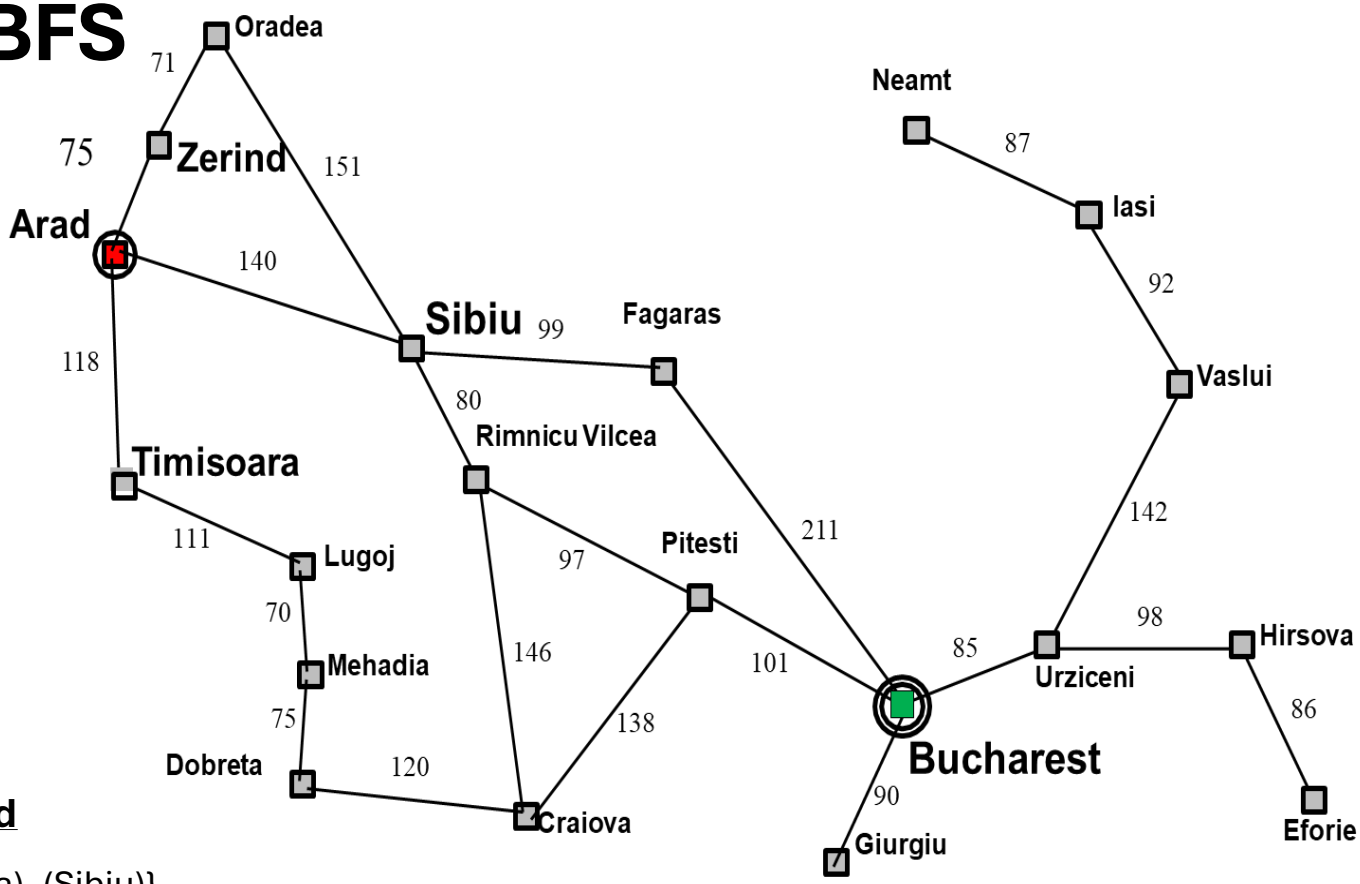
Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



Example: BFS



Example: Start From Arad

- **Step1**. {(Zerind), (Timisoara), (Sibiu)}
- **Step2**. {(Timisoara), (Sibiu), (Oradea)}
- **Step3.** {(Sibiu), (Oradea), (Lugoj)}
- **Step4.** {(Oradea), (Lugoj), (Fagaras), (Rimnicu Vilcea)}
- **Step5.** {(Lugoj), (Fagaras), Rimnicu Vilcea)}
- **Step6.** {(Fagaras), (Rimnicu Vilcea), (Mehadia)}
- **Step7.** {(Rimnicu Vilcea), (Mehadia), (Bucharest)}

A. Breadth-first search on a graph

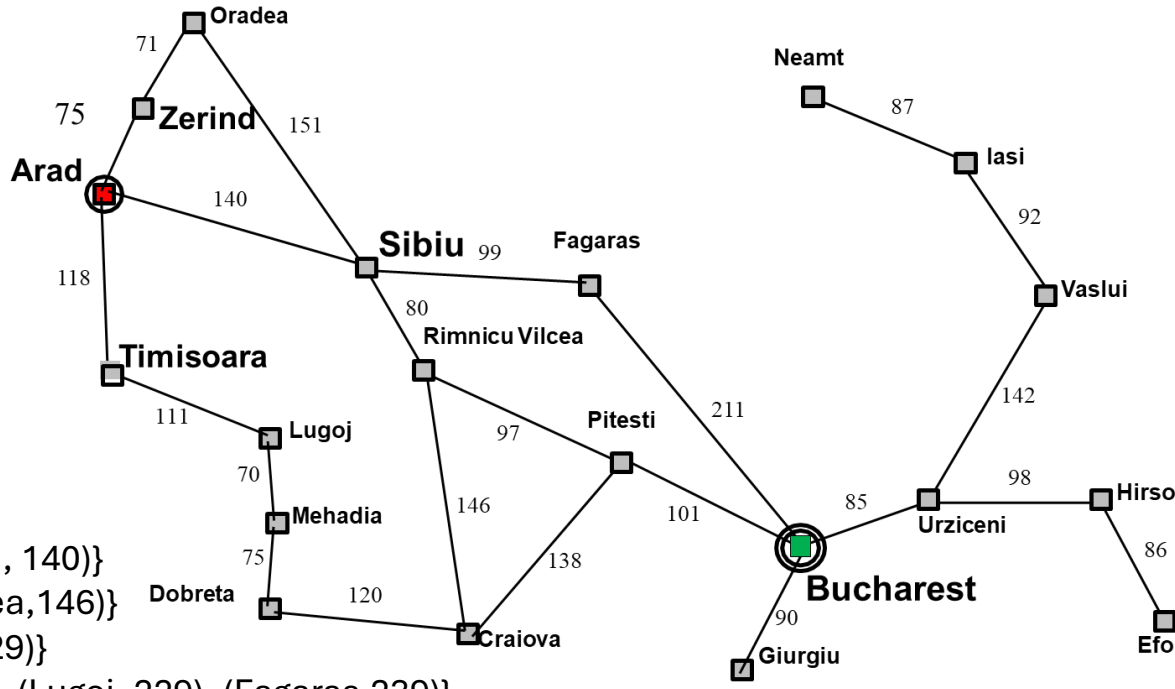
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)

function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

B. Uniform-cost search (UCS)

🧩 **How it works:**

- Expands the node with the lowest total path cost $g(n)$.Unlike BFS, it considers costs of paths (not just depth).



✅ **Pros:**

- **Complete** (if all costs ≥ 0)
- **Optimal** (finds least-cost solution)

⚠️ **Cons:**

- **Can be slow** and memory-intensive because it explores many nodes.

Example: Start From Arad:

Step1. {(Zerind, 75), (Timisoara, 118), (Sibiu, 140)}
Step2. {(Timisoara,118), (Sibiu,140), (Oradea,146)}
Step3. {(Sibiu,140), (Oradea,146), (Lugoj,229)}
Step4. {(Oradea,146), (Rimnicu Vilcea,220), (Lugoj, 229), (Fagaras,239)}
Step5. {(Rimnicu Vilcea,220), (Lugoj,229), (Fagaras,239), (Sibiu,297)}
Step6. {(Lugoj,229), (Fagaras,239), (Sibiu,297), (Pitesti,317), (Craiova,366)}
Step7. {(Fagaras,239), (Mehadia,299), (Sibiu,297 --> ignored), (Pitesti,317), (Craiova, 366)}
Step8. {(Mehadia,299), (Pitesti,317), (Craiova, 366), (Bucharest, 450)}
Step9. {(Pitesti,317), (Craiova, 366), (Drobeta, 374), (Bucharest,450)}
Step10.{(Craiova, 366), (Drobeta, 374), (Bucharest,418), (Bucharest,450--> ignored)}
Step11.{(Drobeta, 374), (Bucharest,418), (Bucharest,450--> ignored)}
Step12.{(Bucharest,418), (Bucharest,450--> ignored)}

B. Uniform-cost search

Algorithm Steps (as shown in the diagram)

1.Initialize:

Frontier = {Start}, with cost = 0.

2.Expand the lowest-cost node (the one with smallest *g(n)*).

For each neighbor, compute the new cumulative cost and insert it into the frontier.

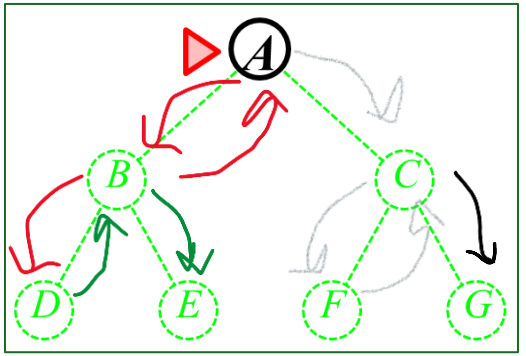
3.Repeat:

1. If the goal is expanded → stop (solution found).
2. Otherwise, continue expanding nodes in increasing order of total cost.

C. Depth-first Search (DFS)

🧩 **How it works:**

- Always expands the deepest unexpanded node first.
- It goes deep into one branch before backtracking.



🧠 **Example (Romania Map):**

From **Arad**, DFS first explores:

Arad → Sibiu → Fagaras → Bucharest (if lucky).Otherwise, it might go deep into the wrong path before finding the goal.

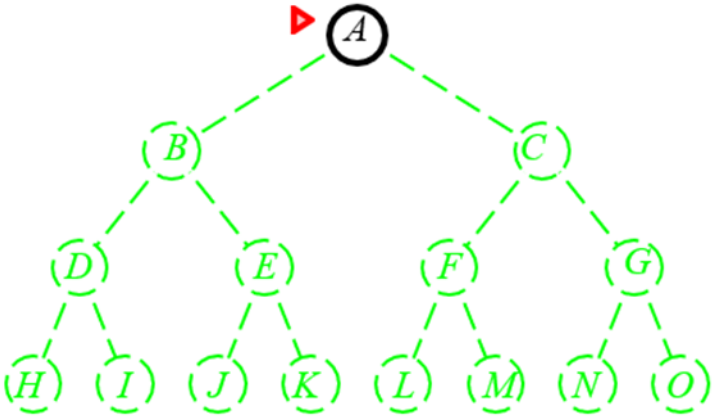
✅ **Pros:**

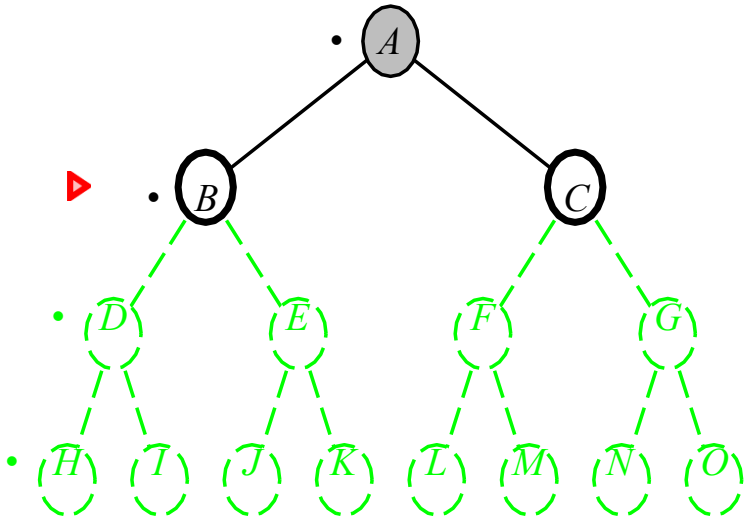
- **Low memory:** Can find a solution quickly (if the goal is deep on one branch), $O(bm)$ where m is the max depth.
- **Optimal:** Can find a solution quickly (if the goal is deep on one branch)

⚠️ **Cons:**

- **Incomplete:** $O\left(b^{d+1}\right)$: can get stuck in infinite paths.
- **Not optimal**

C. Depth-first search

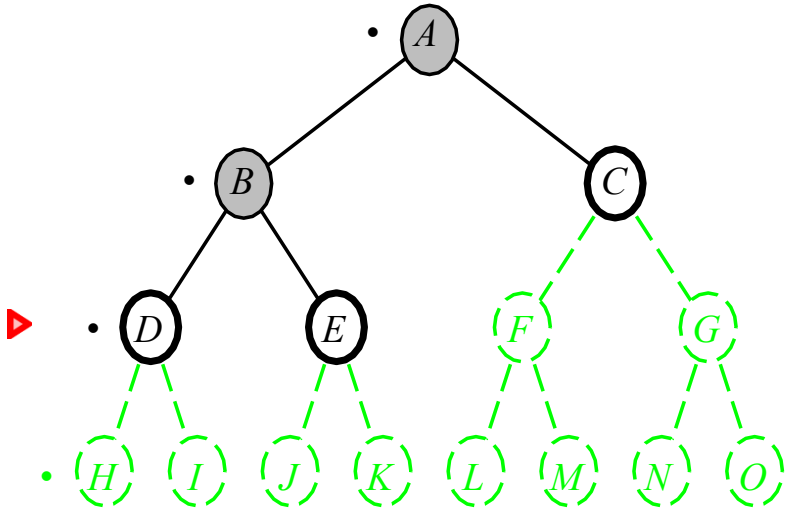




C. Depth-first search

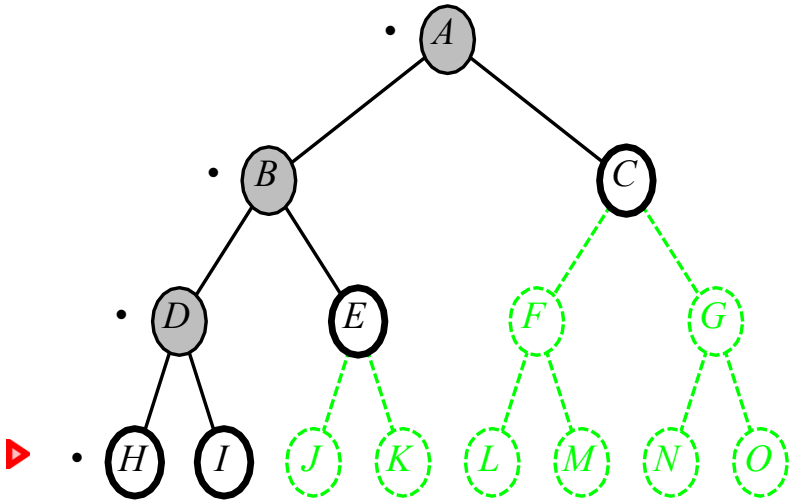
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

LIFO:
Last In
First Out



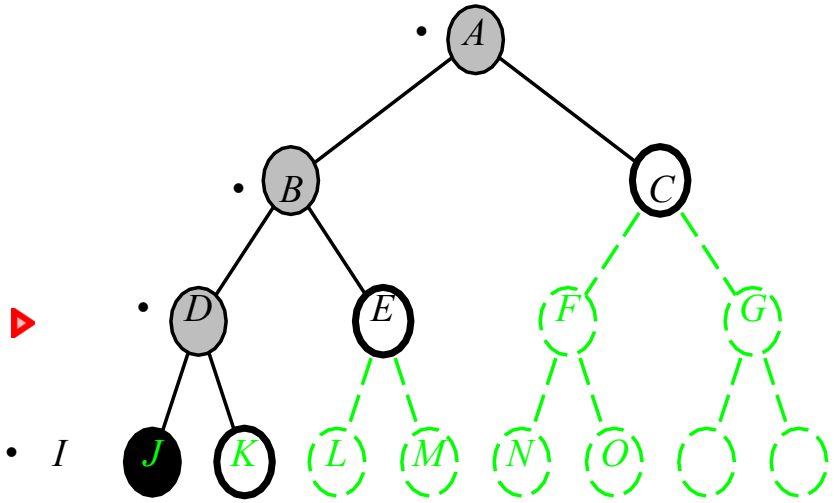
C. Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



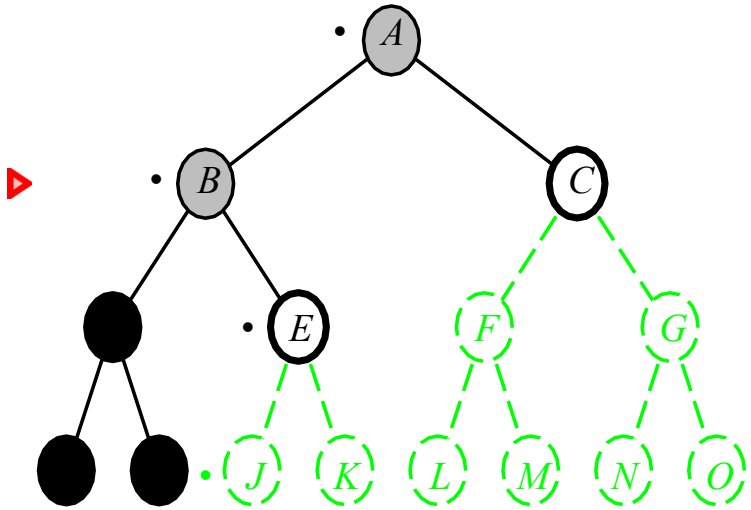
C. Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



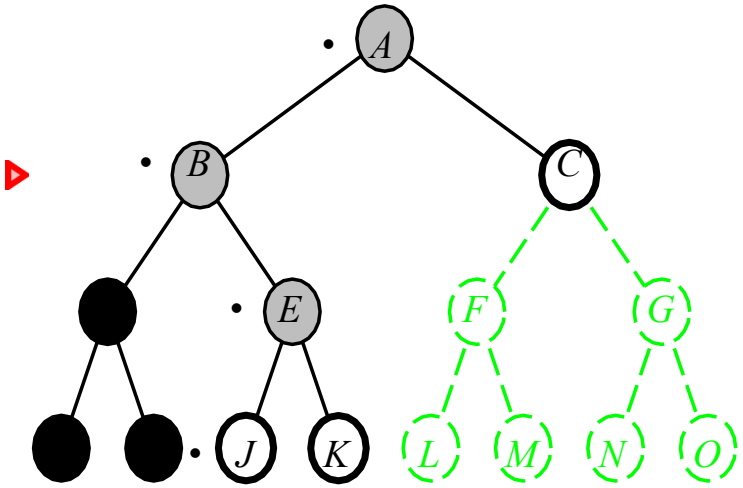
C. Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



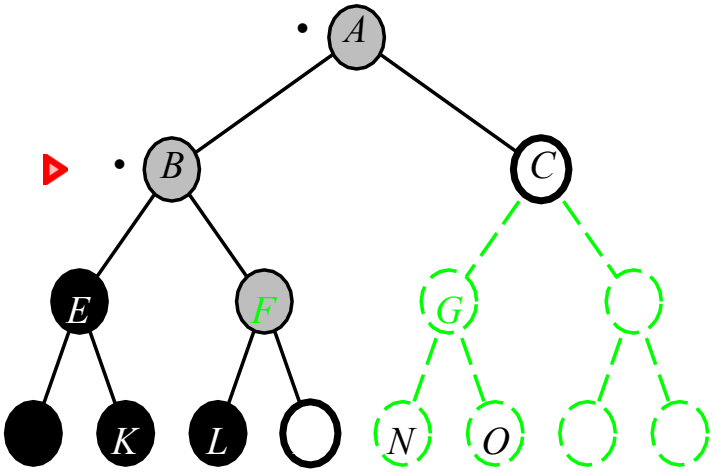
C. Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



C. Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

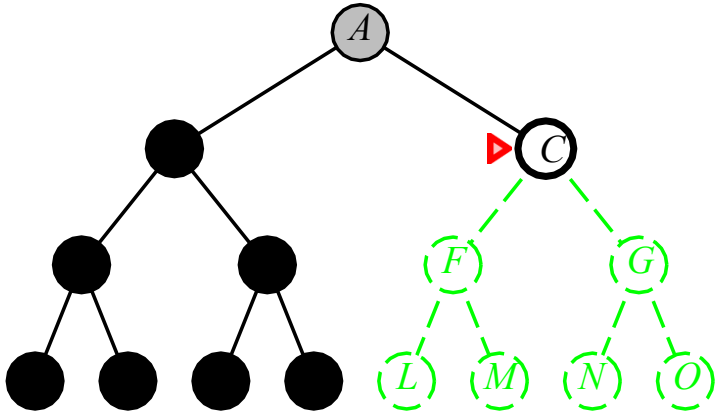


C. Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

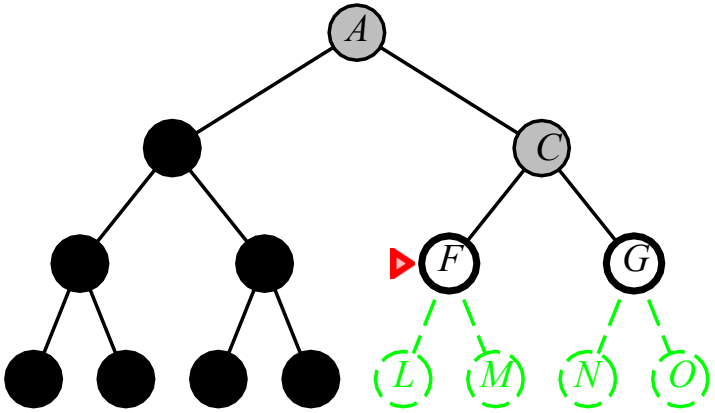


C. Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

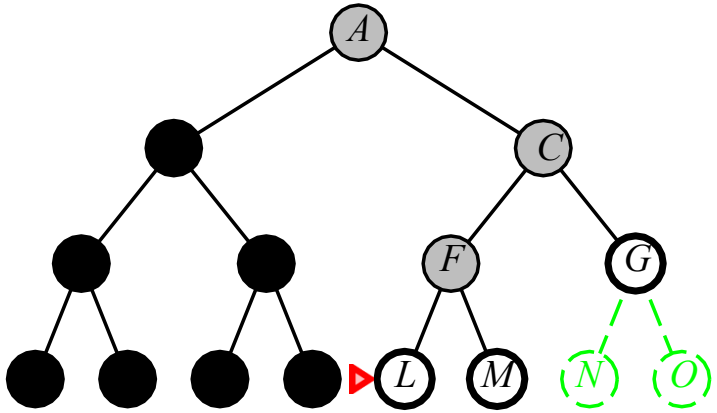


C. Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

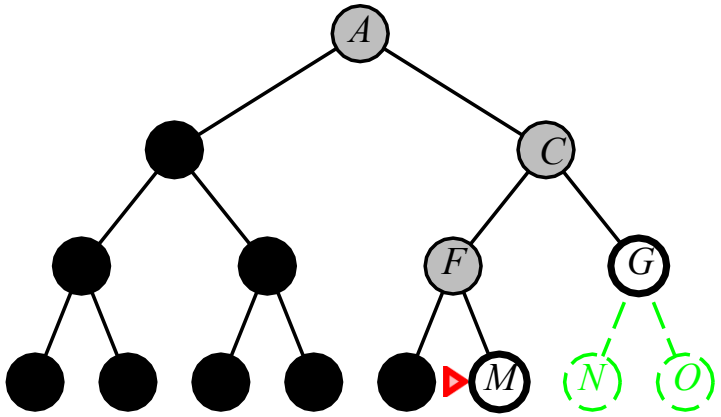


C. Depth-first search

Expand deepest unexpanded node

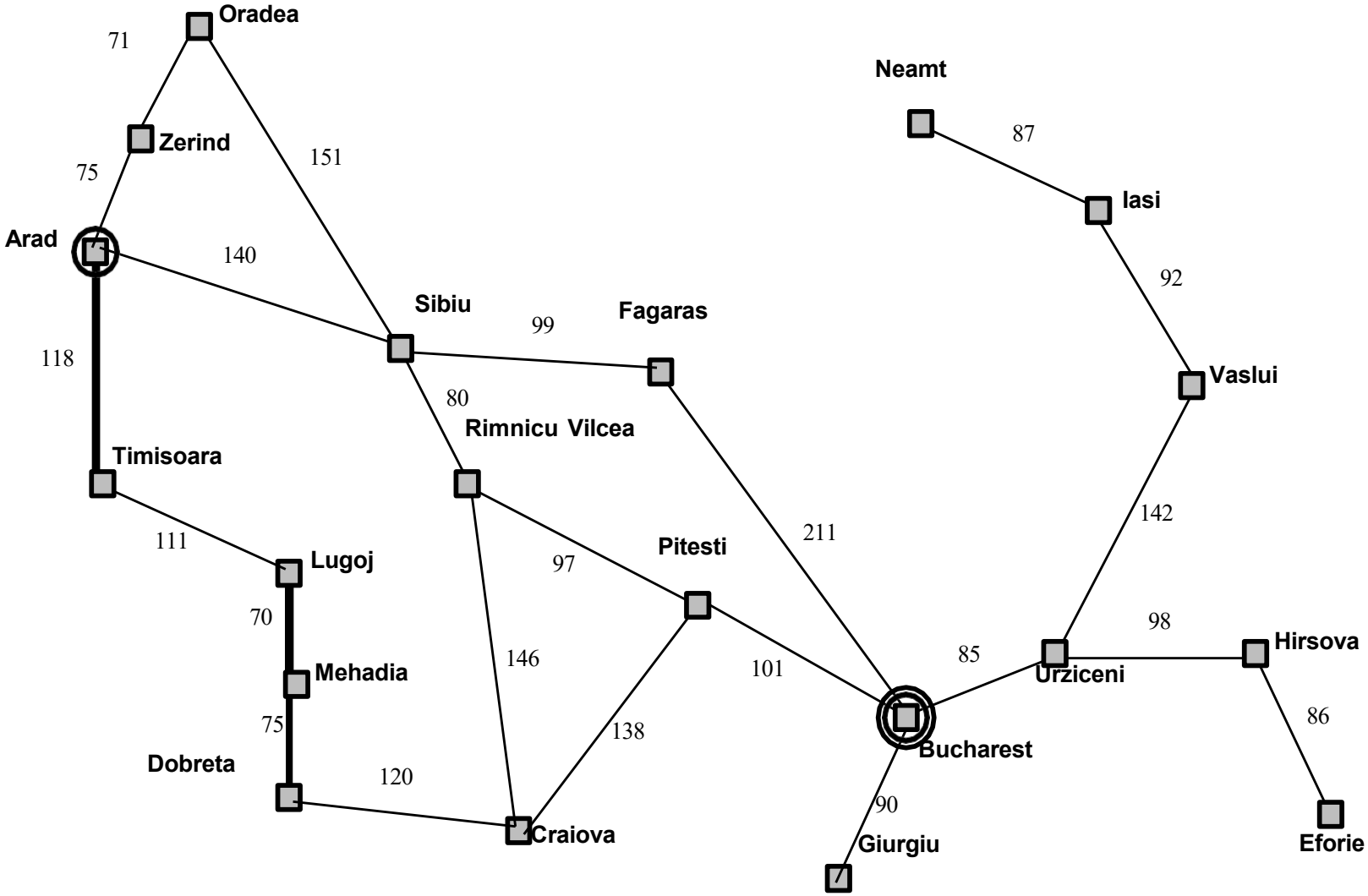
Implementation:

fringe = LIFO queue, i.e., put successors at front



Properties of depth-first search

Complete??



D. Depth-Limited Search (DLS)

🧩 **How it works:**

- Like DFS, but limits the depth of exploration to L levels.
- If the goal is deeper than L, it won't be found.

🧠 **Example:**

Limit depth to 3 — the agent only searches 3 moves away from Arad.

✅ **Pros:**

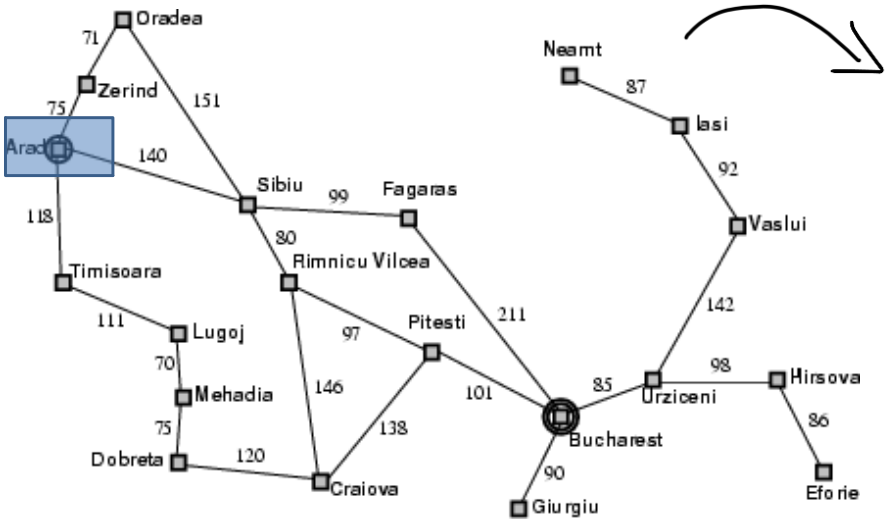
- **Avoids infinite loops.**
- **Requires less memory.**

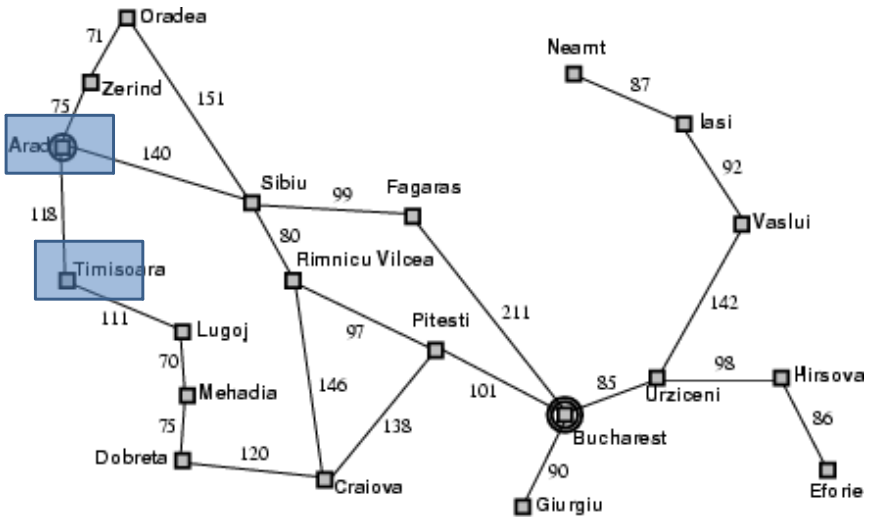
⚠️ **Cons:**

- **Incomplete (if goal is below depth L)**
- **Not optimal**

D. Depth-Limited Search (DLS)Example

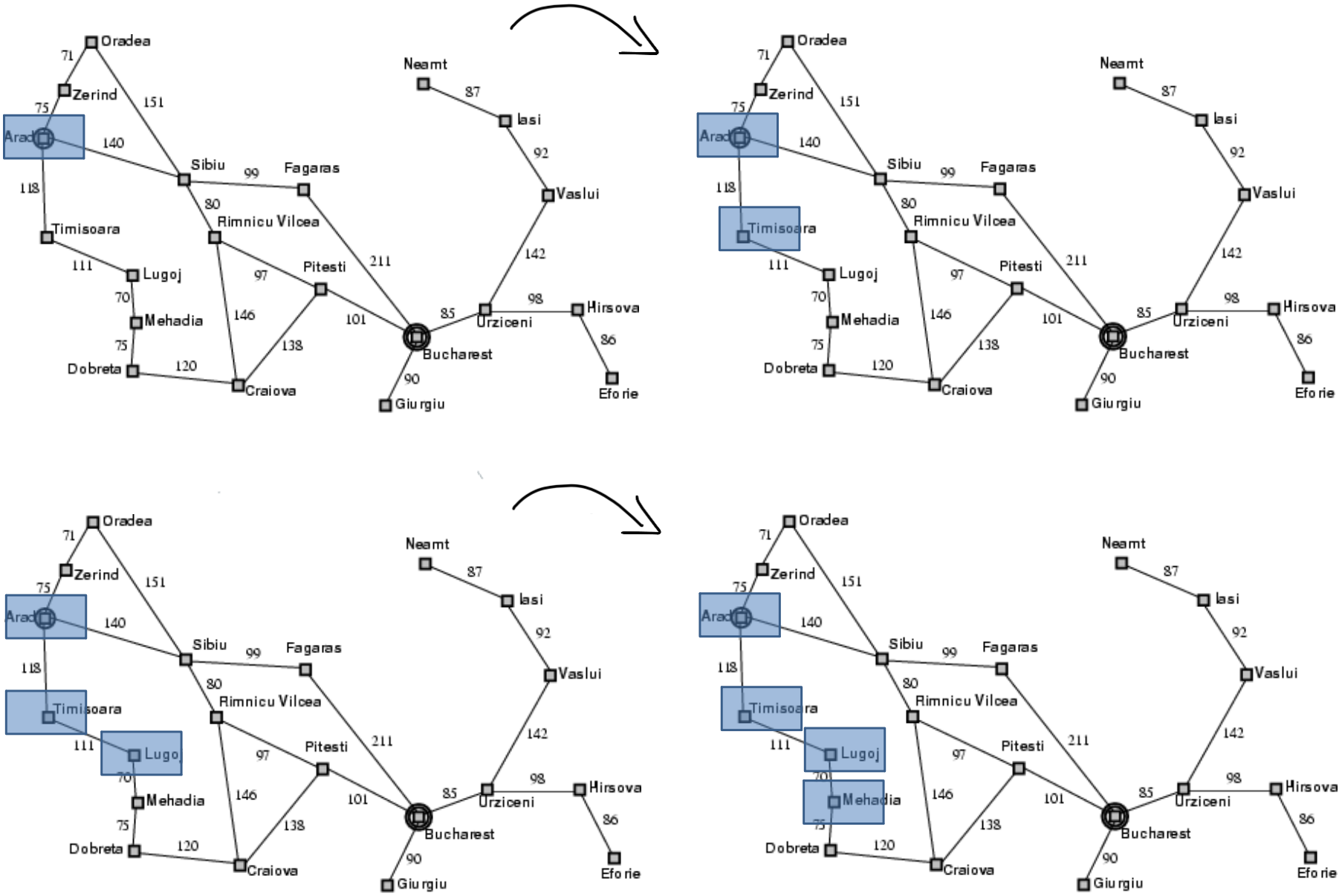
Max depth = 3





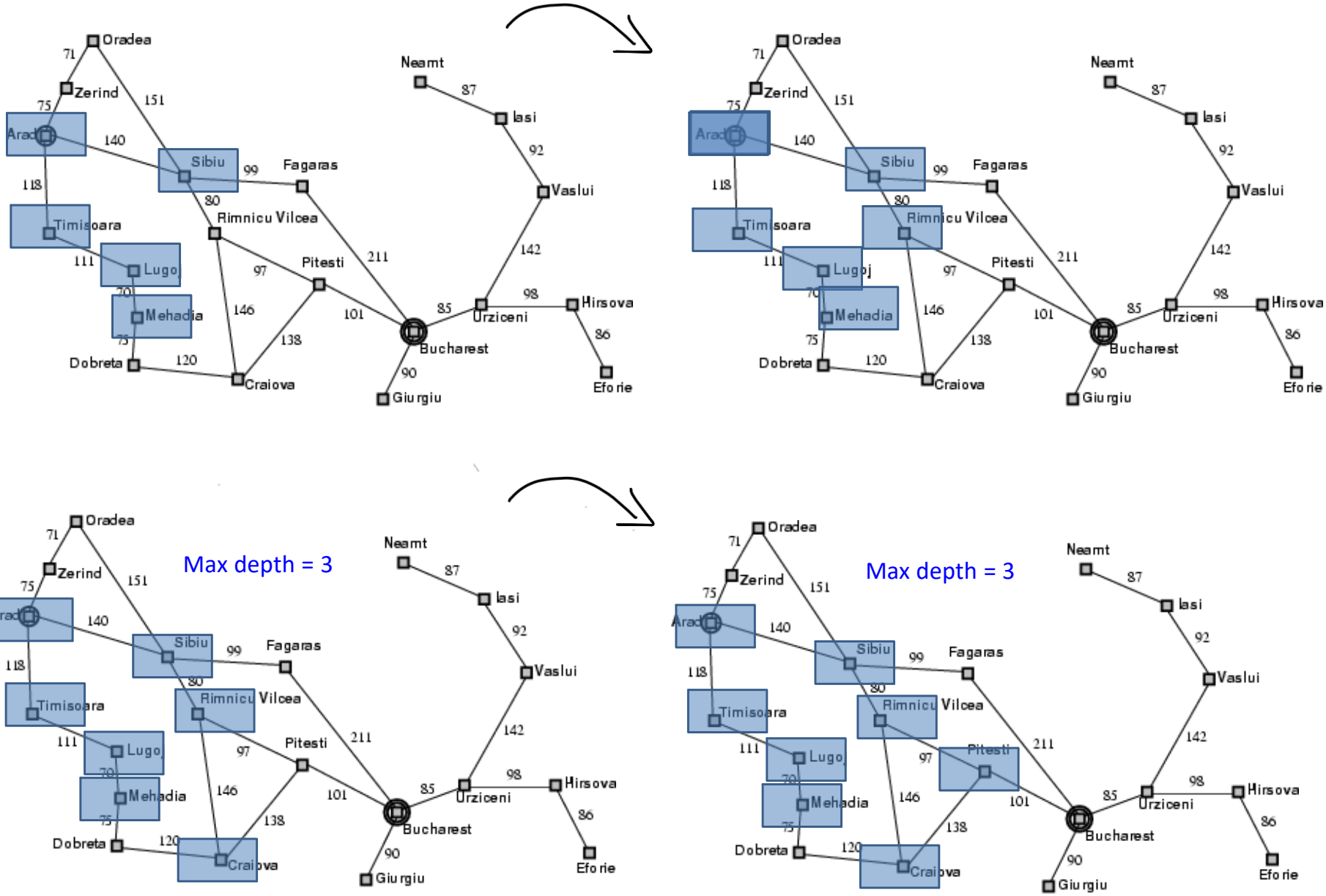
Max depth = 3

D. Depth-Limited Search (DLS)Example

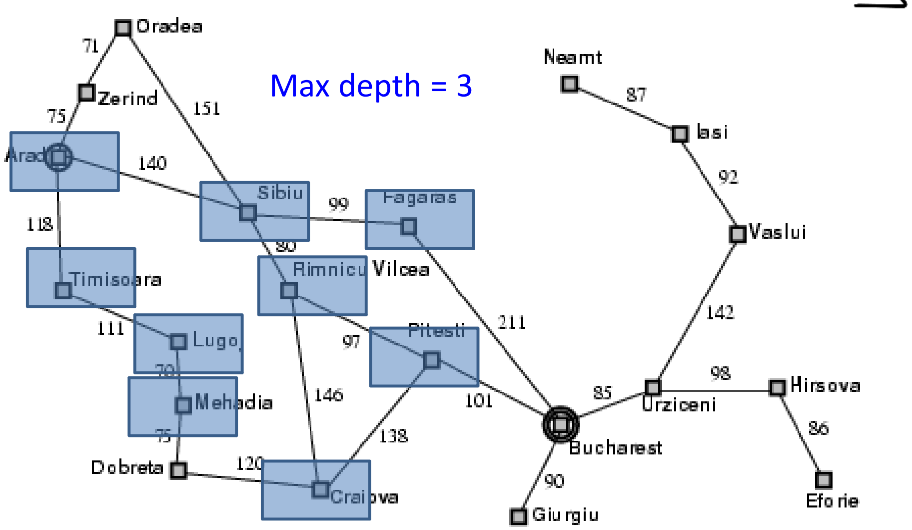


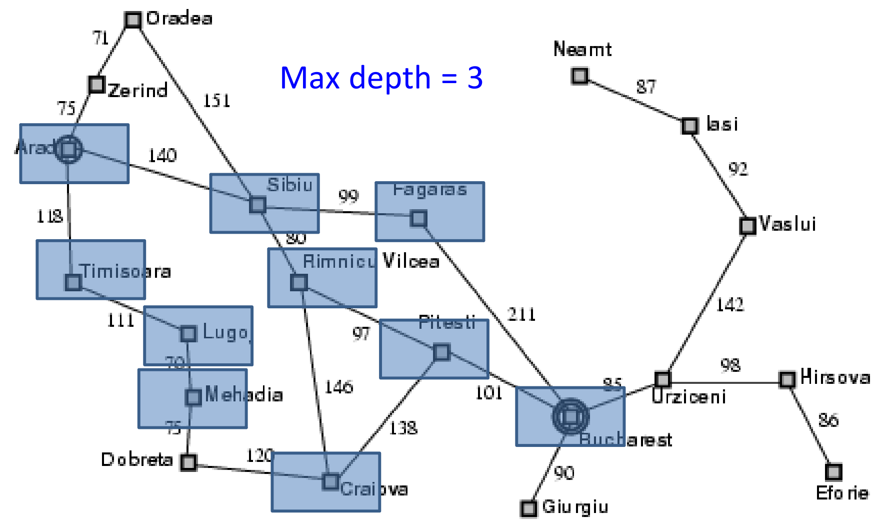
Max depth = 3

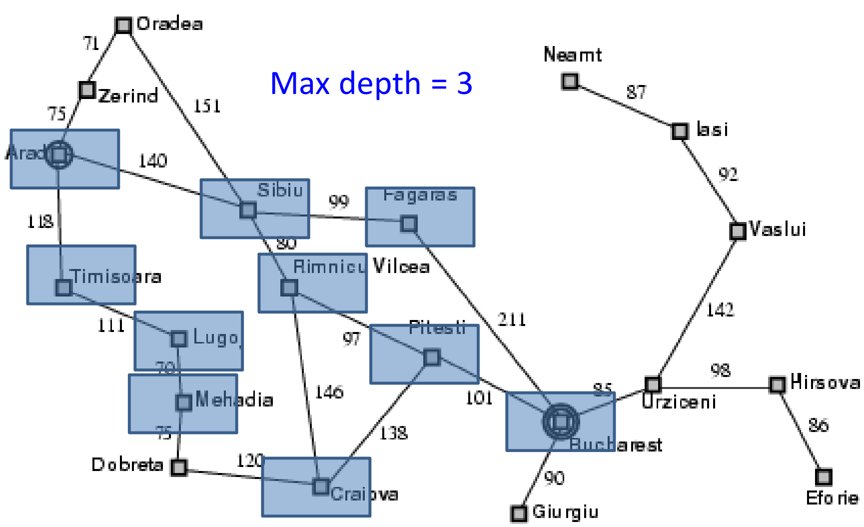
D. Depth-Limited Search (DLS)Example



D. Depth-Limited Search (DLS)Example







E. Iterative Deepening Search (IDS)

🧩 **How it works:**

•Combines DFS and BFS advantages.

•Repeatedly runs Depth-Limited Search with increasing limits (L = 0, 1, 2, 3…)

•When goal is found, it stops.

🧠 **Example:**

Search depth 0 → 1 → 2 → 3… until it finds Bucharest.

✅ **Pros:**

•Complete

•Optimal

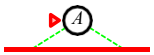
•Memory efficient

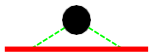
⚠️ **Cons:**

•Some repeated work, but overall efficient for large search spaces.

E. Iterative deepening search

Limit = 0



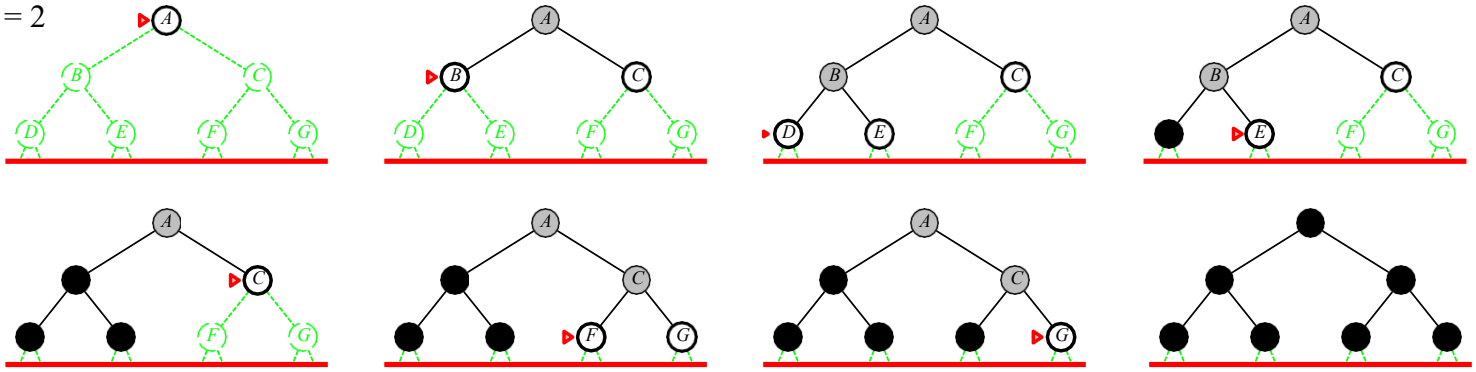


E. Iterative deepening search *l* = 1

Limit = 1

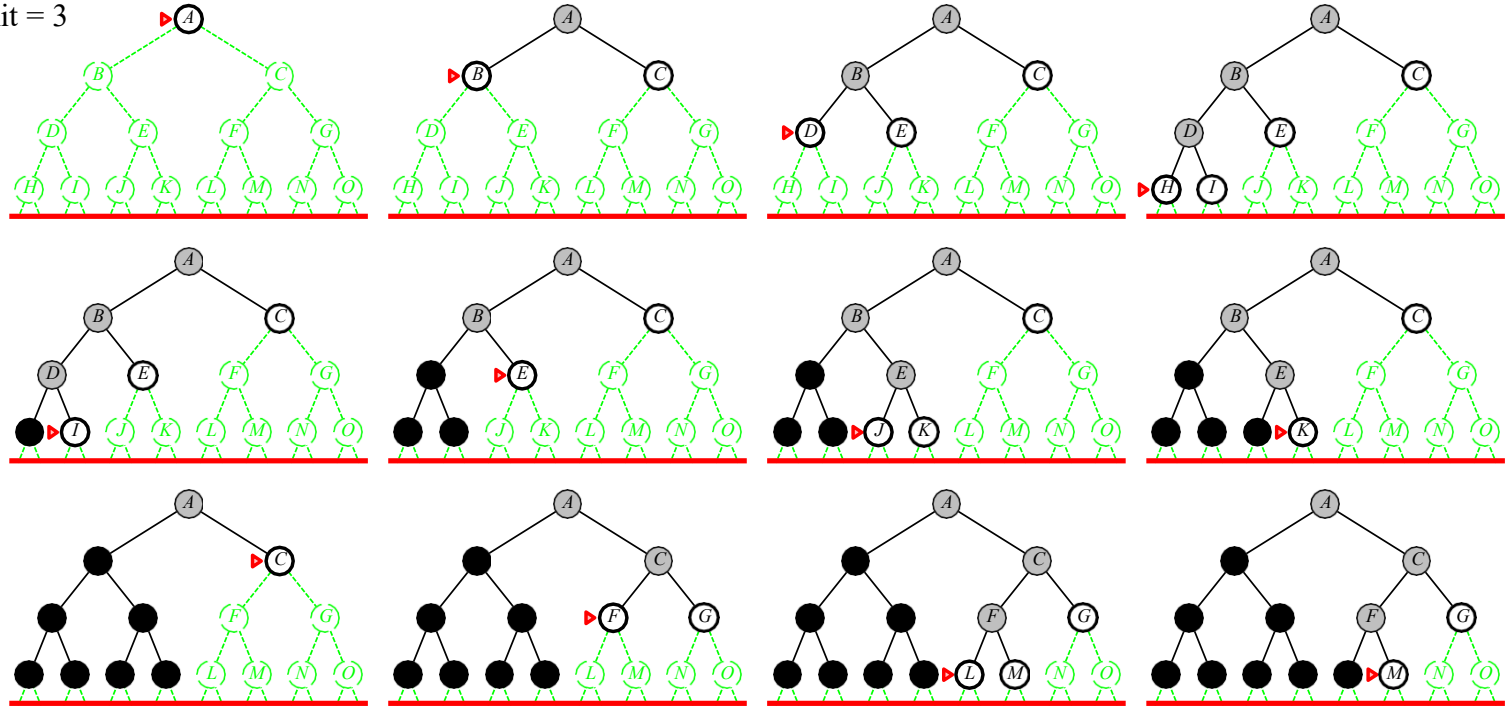
E. Iterative deepening search *l* = 2

Limit = 2

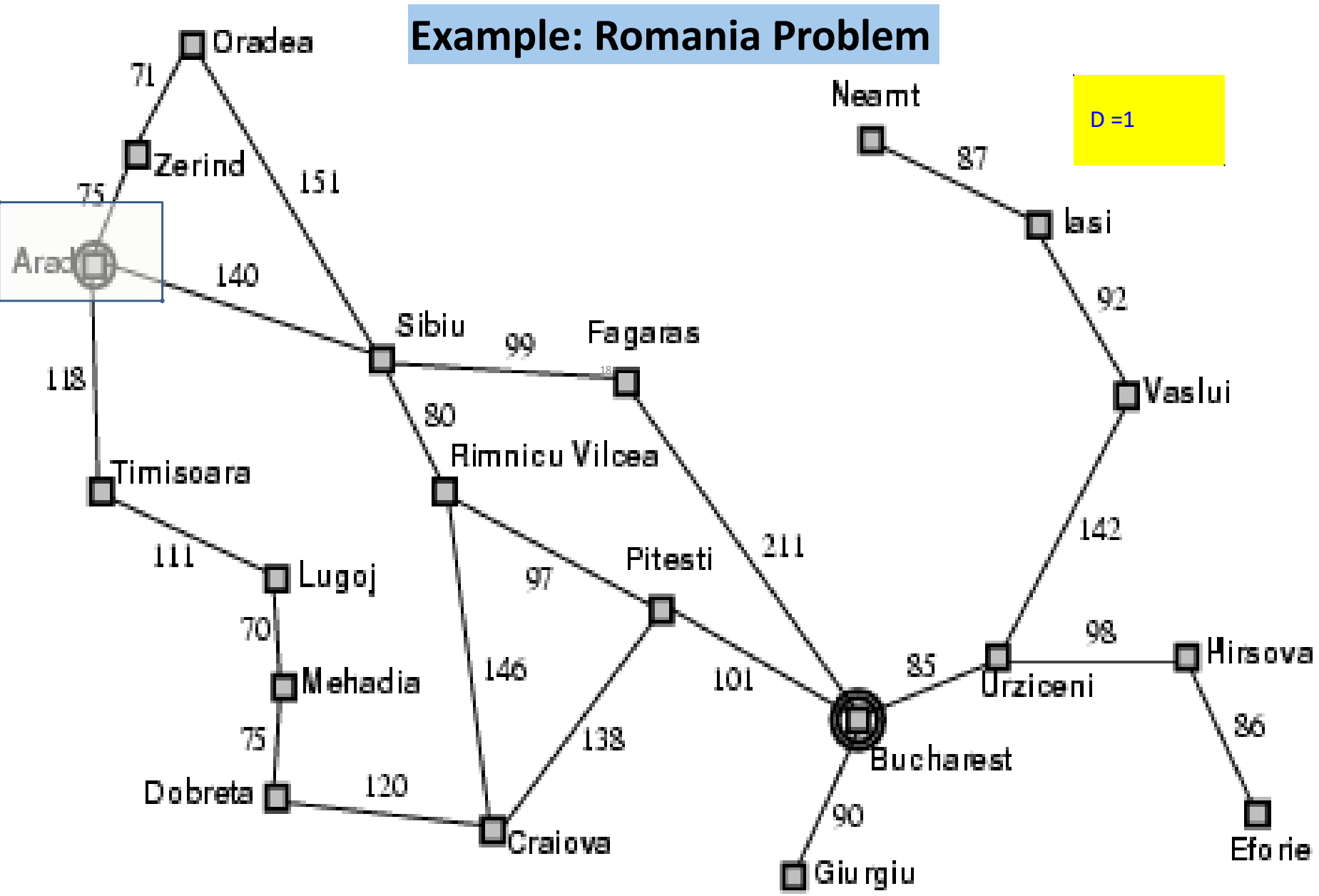


E. Iterative deepening search *l* = 3

Limit = 3

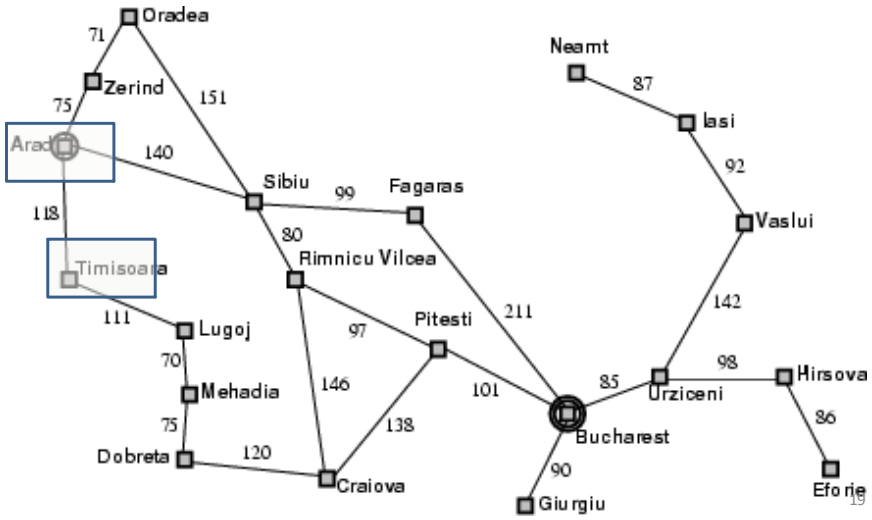


E. Iterative deepening search (IDS) Example



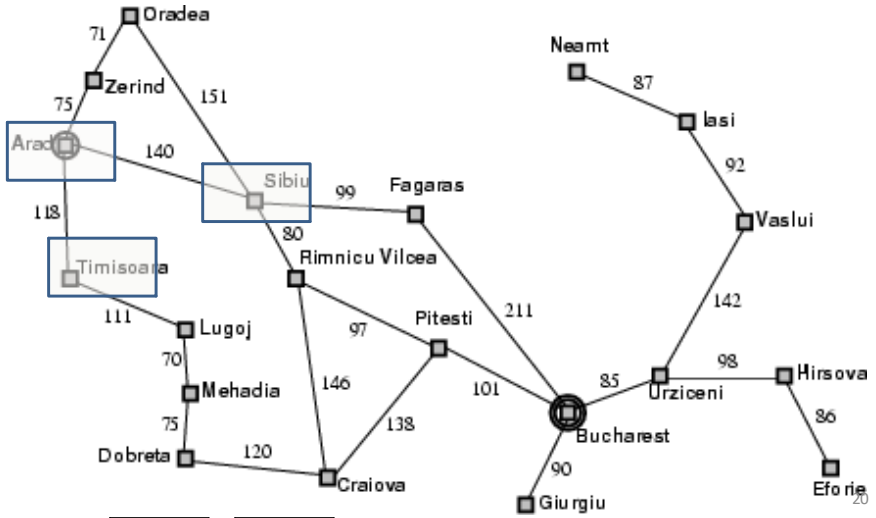
E. Iterative deepening search (IDS) Example

D =1



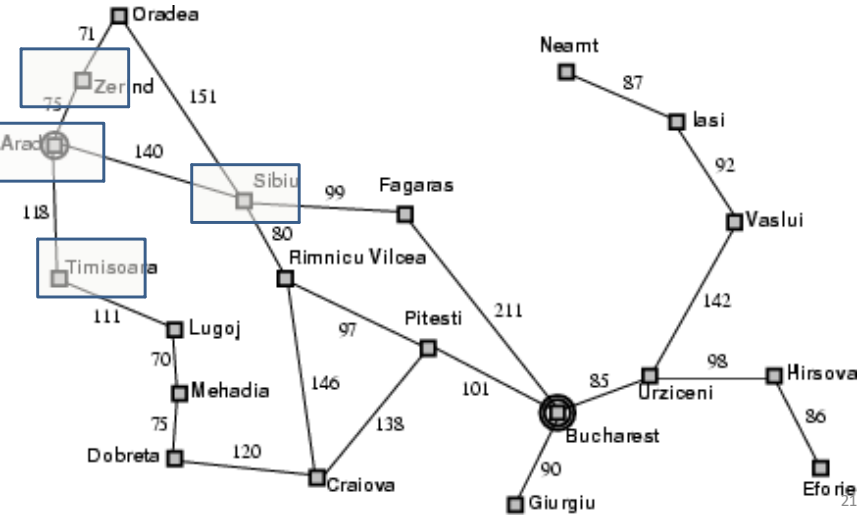
19

D =1



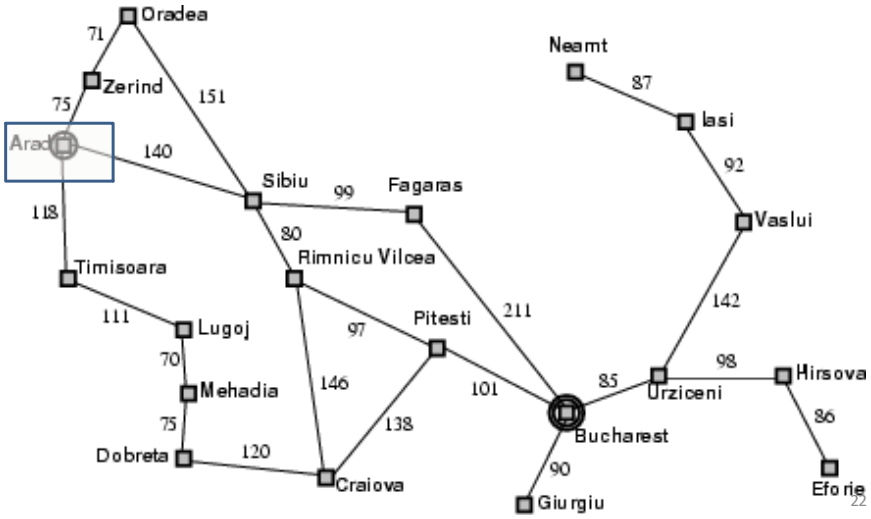
20

D =1



21

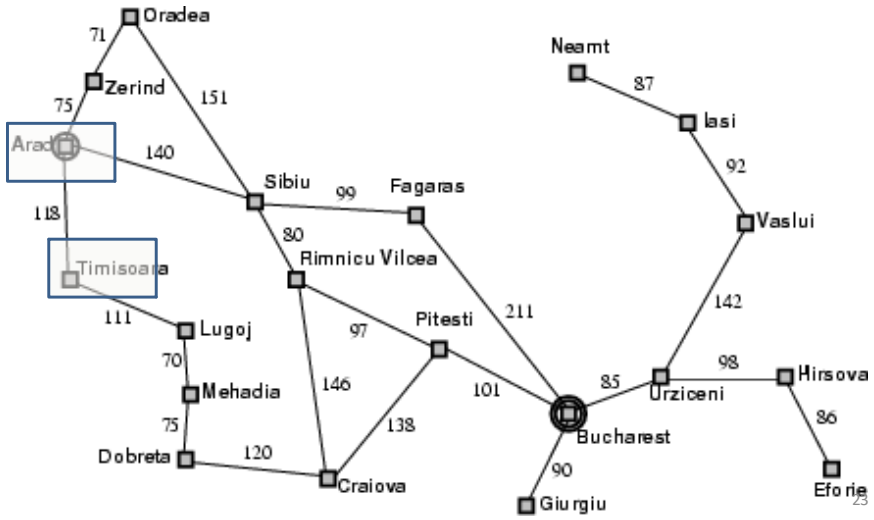
D =1 D =2



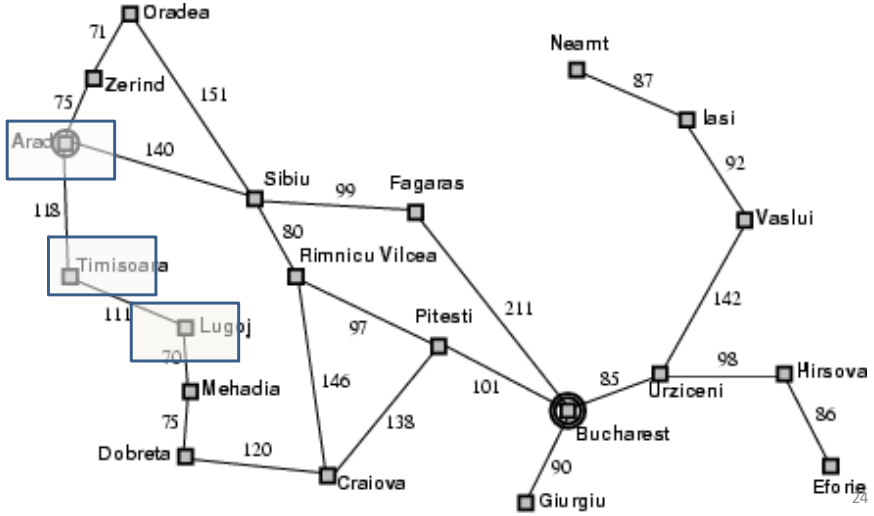
22

E. Iterative deepening search (IDS) Example

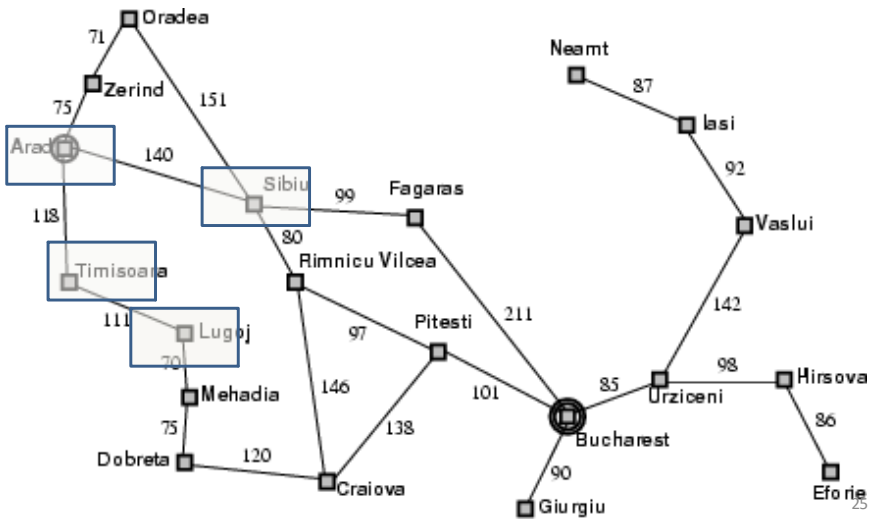
D =1 D =2



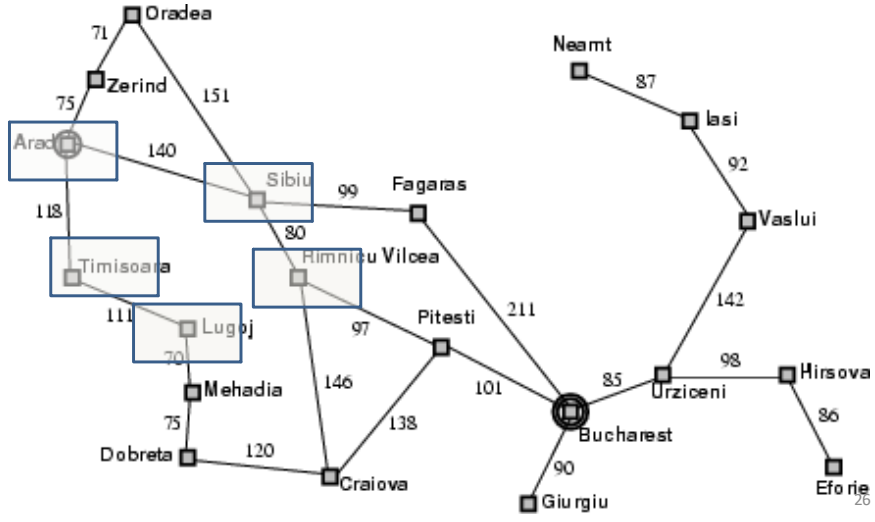
D =1 D =2



D =1 D =2

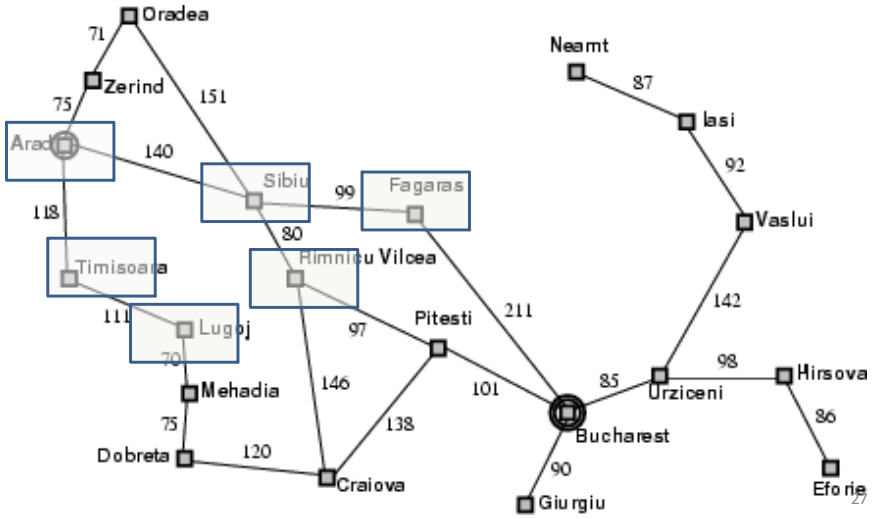


D =1 D =2

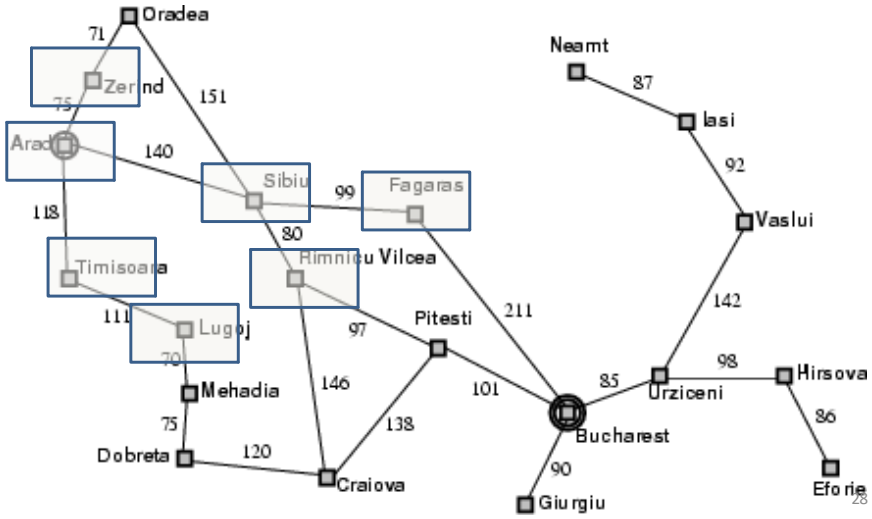


E. Iterative deepening search (IDS) Example

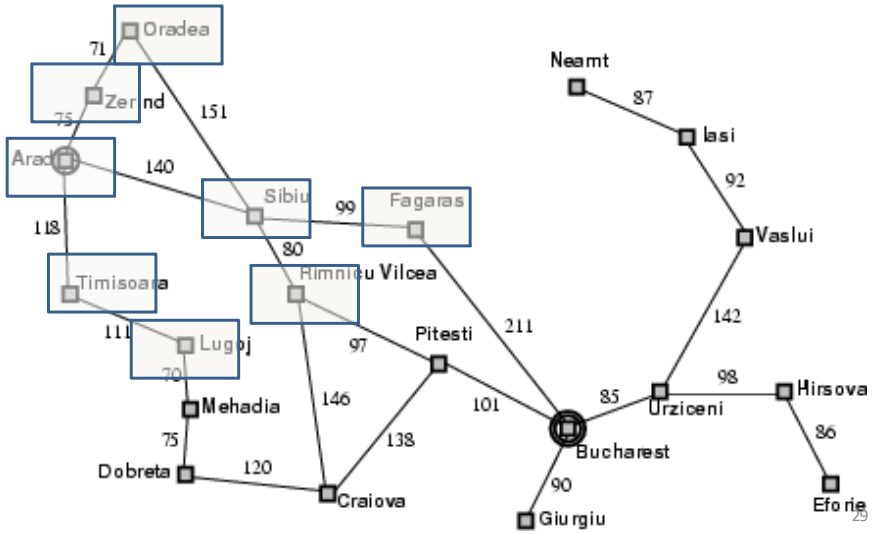
D =1 D =2



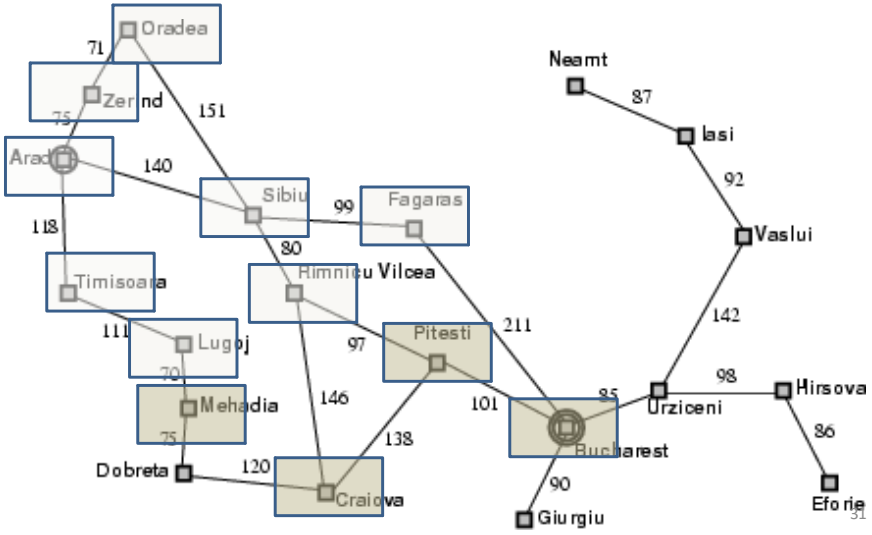
D =1 D =2



D =1 D =2



D =1 D =2 D =3



Outline

- Problem-solving agents
- Problem formulation
- Basic search algorithms
- **Informed (heuristic) search strategies**

Informed (heuristic) search strategies

It's a **search algorithm** that tries to reach the goal as quickly as possible by **always expanding the node that seems closest to the goal** — based on a heuristic.

💡 Heuristic function h(n)

- The **heuristic** is an estimate of how close a node is to the goal.
- It's **not** the actual cost, just a **guess**.

Informed Algorithm Examples:

A. **A* Search**
B. **Memory- bounded heuristic search**
C. **Hill Climbing**

Informed (heuristic) search strategies

🧩 **How it works:**

1. Start at a node (say **Arad**).
2. Look at all neighboring cities (like Sibiu, Zerind, Timisoara).
3. Choose the one with the **lowest heuristic value h(n)** — i.e., the city that *appears* closest to Bucharest.
4. Repeat until you reach the goal.

✅ **Pros:**

- **Very fast (low search cost) — it often finds a solution quickly.**
- **Memory use is low — doesn't need to store many nodes.**

⚠️ **Cons:**

- **Not optimal:** it may find a path, but not the shortest one.

A*search Algorithm

A* search combines **Greedy Best-First Search** and **Uniform Cost Search**.
It balances between:

- **Exploring the cheapest path so far (actual cost)**, and
- **Heading toward the goal (estimated cost)**.

So it's both **informed** and **optimal** (when the heuristic is admissible).

🧩 How it works:

For every node n, A* computes: $f(n) = g(n) + h(n)$ where:

g(n) = the **cost so far** from the start node to n

h(n) = the **heuristic estimate** from n to the goal (e.g., straight-line distance)

🪄 A* chooses the node with the **lowest f(n)** value to expand next.

🧩 Step-by-step example (conceptually)

Suppose we're trying to go from **Arad → Bucharest**.

Node	g(n)	h(n)	f(n)=g+h
Arad	0	366	366
Sibiu	140	253	393
Fagaras	239	176	415
Bucharest	450	0	450

- A* expands nodes based on the smallest f(n) value.
- It continues until it reaches the goal node (Bucharest).

A*search Algorithm

✅ **Pros:**

- **Optimal –** Usually finds the shortest (least-cost) path
- **Complete –** Always finds a solution if one exists.
- **Efficient –** Explores fewer nodes than Uniform Cost or BFS, due to guidance from the heuristic.

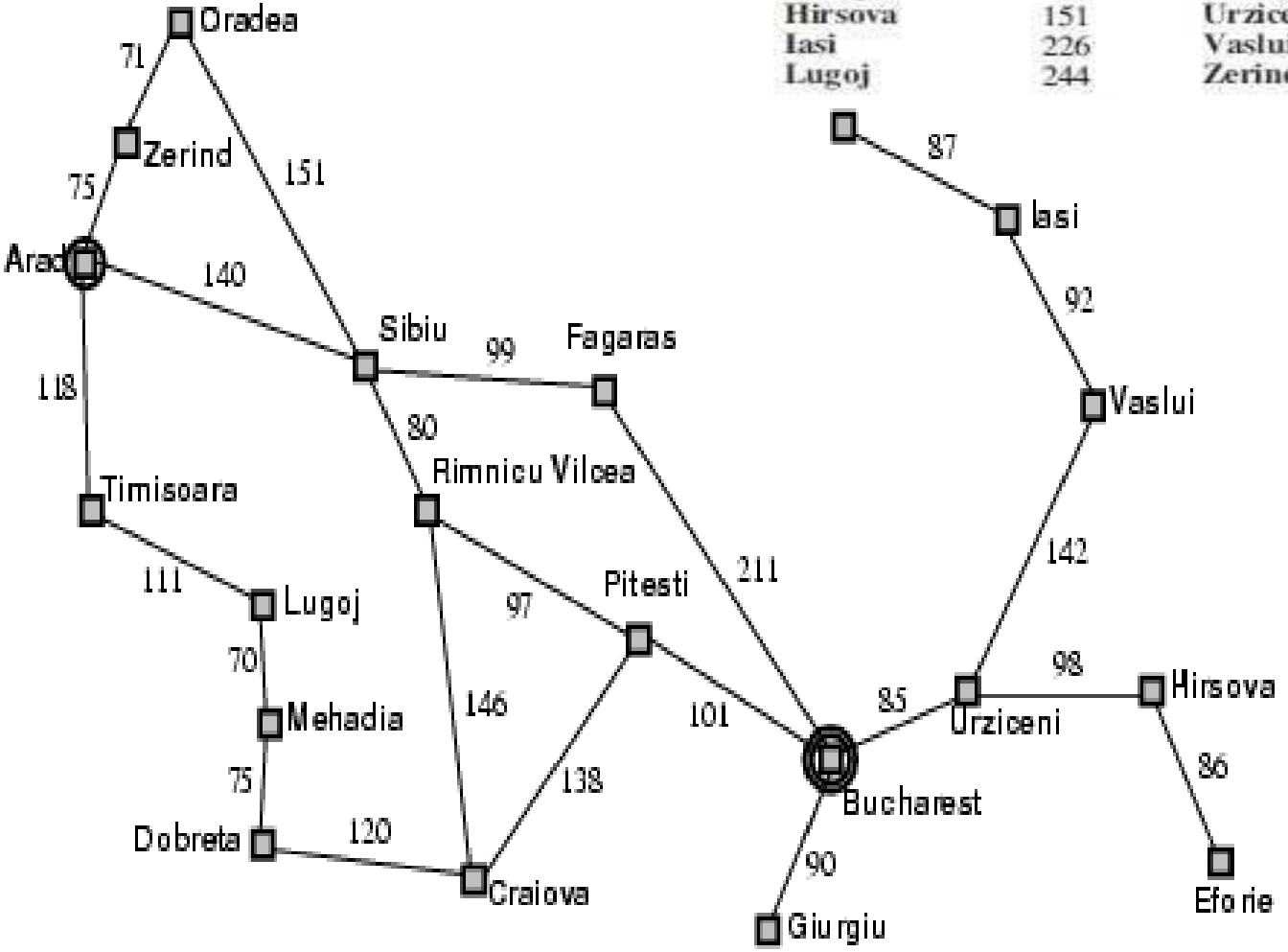
⚠️ **Cons:**

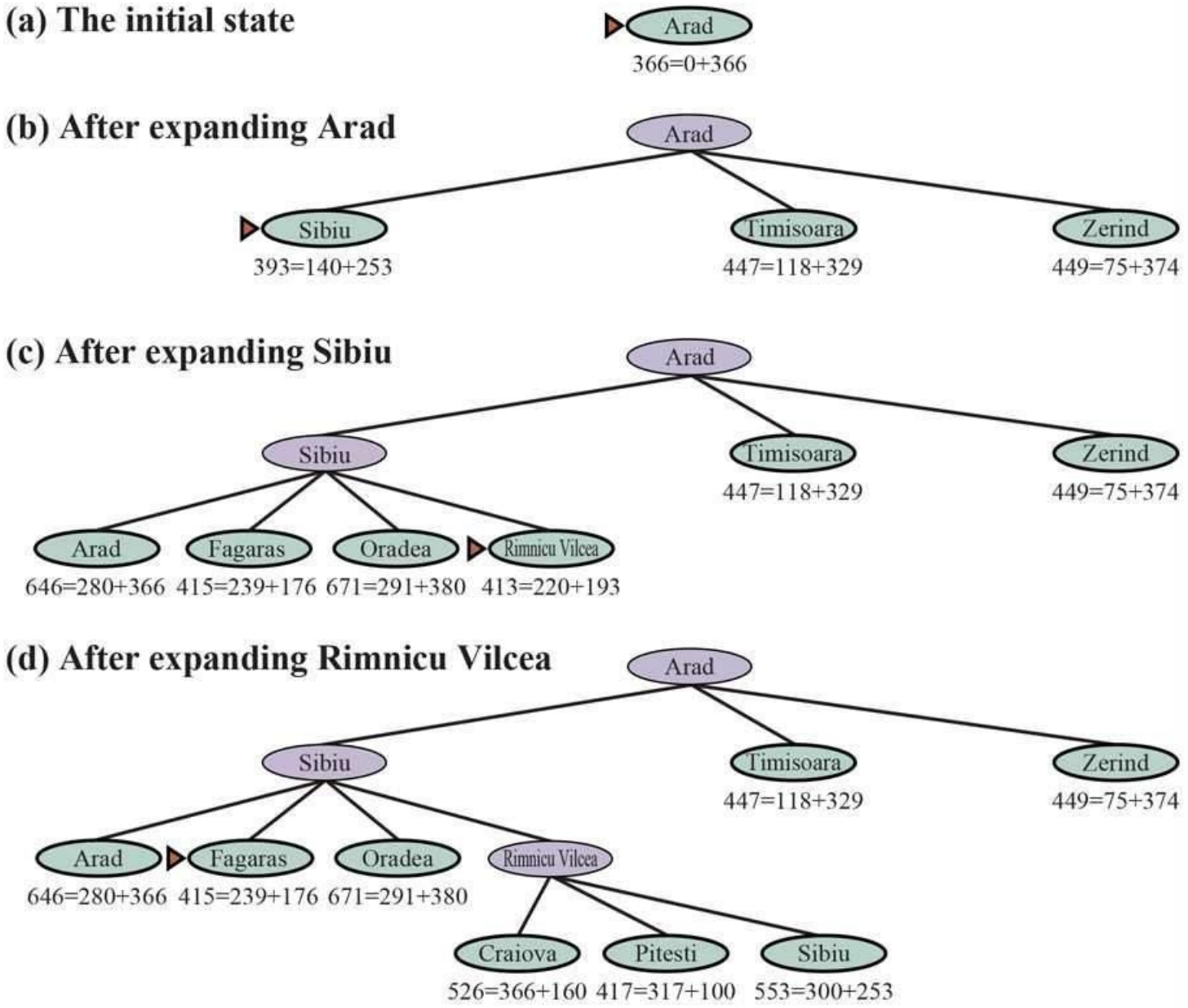
- **High memory use** – Must keep all visited nodes in memory (can be exponential).

A* search: minimising the total estimated solution cost

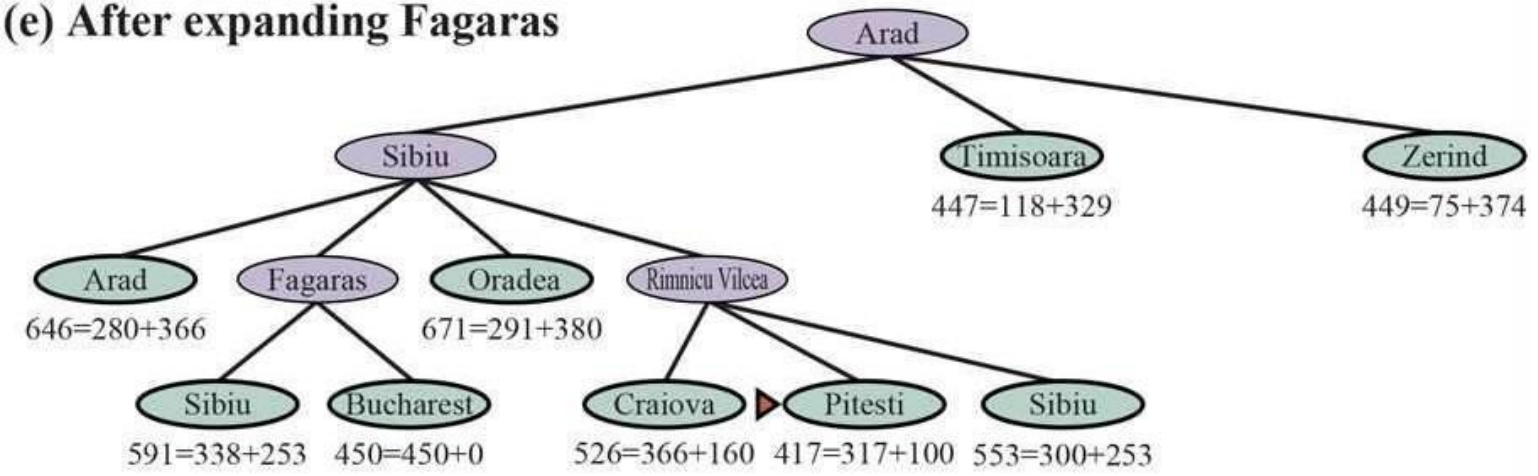
g(n) - the cost to reach the solution
h(n) - the cost to get from the node to the goal
f(n)= g(n)+ h(n)

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

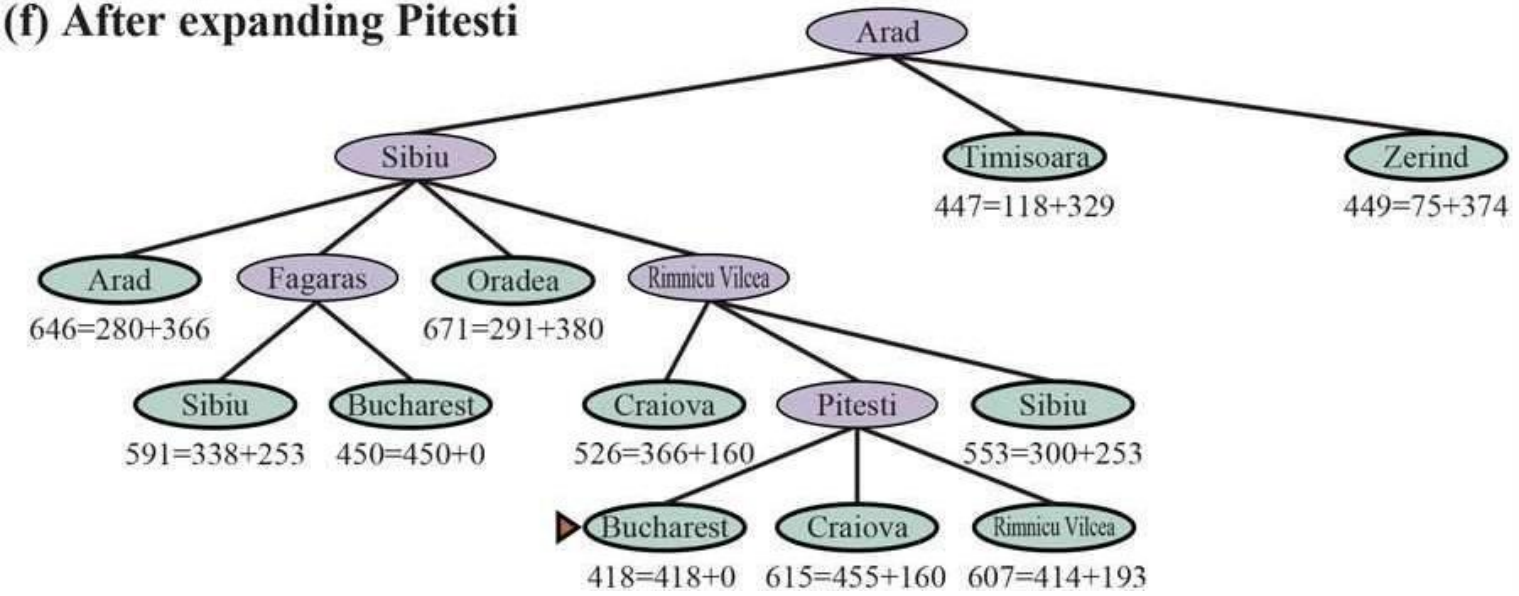



(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
Timisoara
Zerind
393=140+253
447=118+329
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
Craiova
Pitesti
Sibiu
526=366+160
417=317+100
553=300+253

(e) After expanding Fagaras



(f) After expanding Pitesti



Memory-bounded heuristic search 1

1. Goal of RBFS

Recursive Best-First Search (RBFS) is designed to:

•behave like **Best-First Search (specifically A*)**,

•but use only **linear memory**, like **Depth-First Search (DFS)**.

👉 So it's a *memory-efficient* version of A*, ideal when you can't store the entire open list.

🔹 **2. How It Works (Conceptually)**

RBFS works **recursively**, just like DFS:

•It goes down one path at a time (depth-first style).

•But it keeps track of the **f-values** (`f = g + h`) — which represent estimated total cost from start to goal — to decide when to backtrack.

🔹 **3. The Key Idea — the f-limit**

During the recursive search, RBFS maintains an **f-limit**, which is:

The best (lowest) f-value among the unexplored alternative paths from ancestors.

•Think of it as a **cost threshold** that tells the algorithm when to stop exploring a path that looks too expensive.

If the f-value of the current path **exceeds** this f-limit:

→ RBFS **backs up** (returns) to an earlier node that had a better alternative.

🔹 **4. Backtracking and Updating f-values**

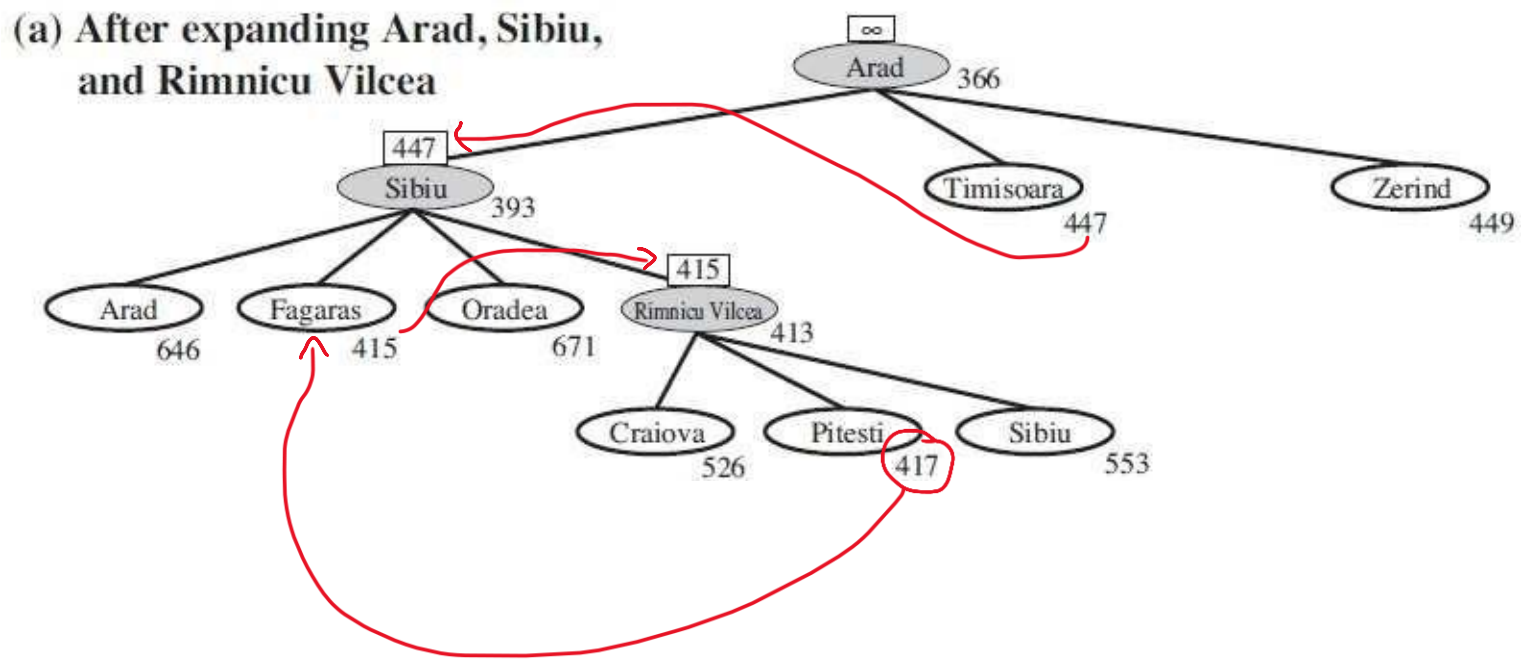
When RBFS backtracks, it doesn't forget everything — it updates the **f-value** of each node along the way with the **best f-value among its children**.

That way:

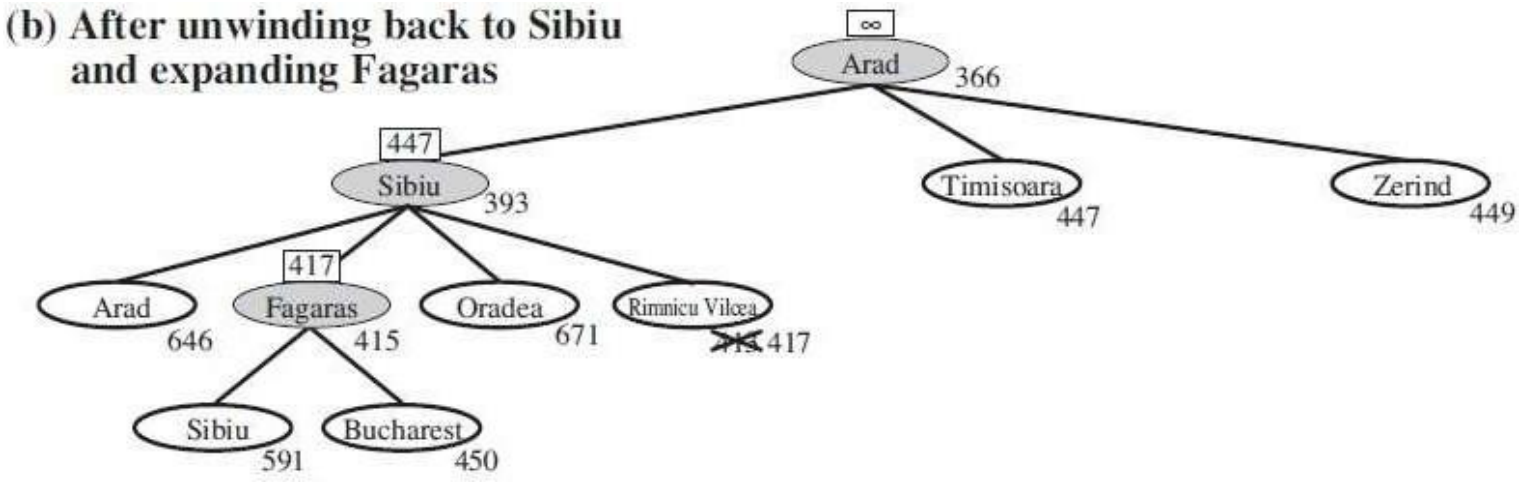
•RBFS "remembers" how good (or bad) a pruned subtree was.

•If a better path is later found, RBFS can **re-expand** that subtree efficiently.

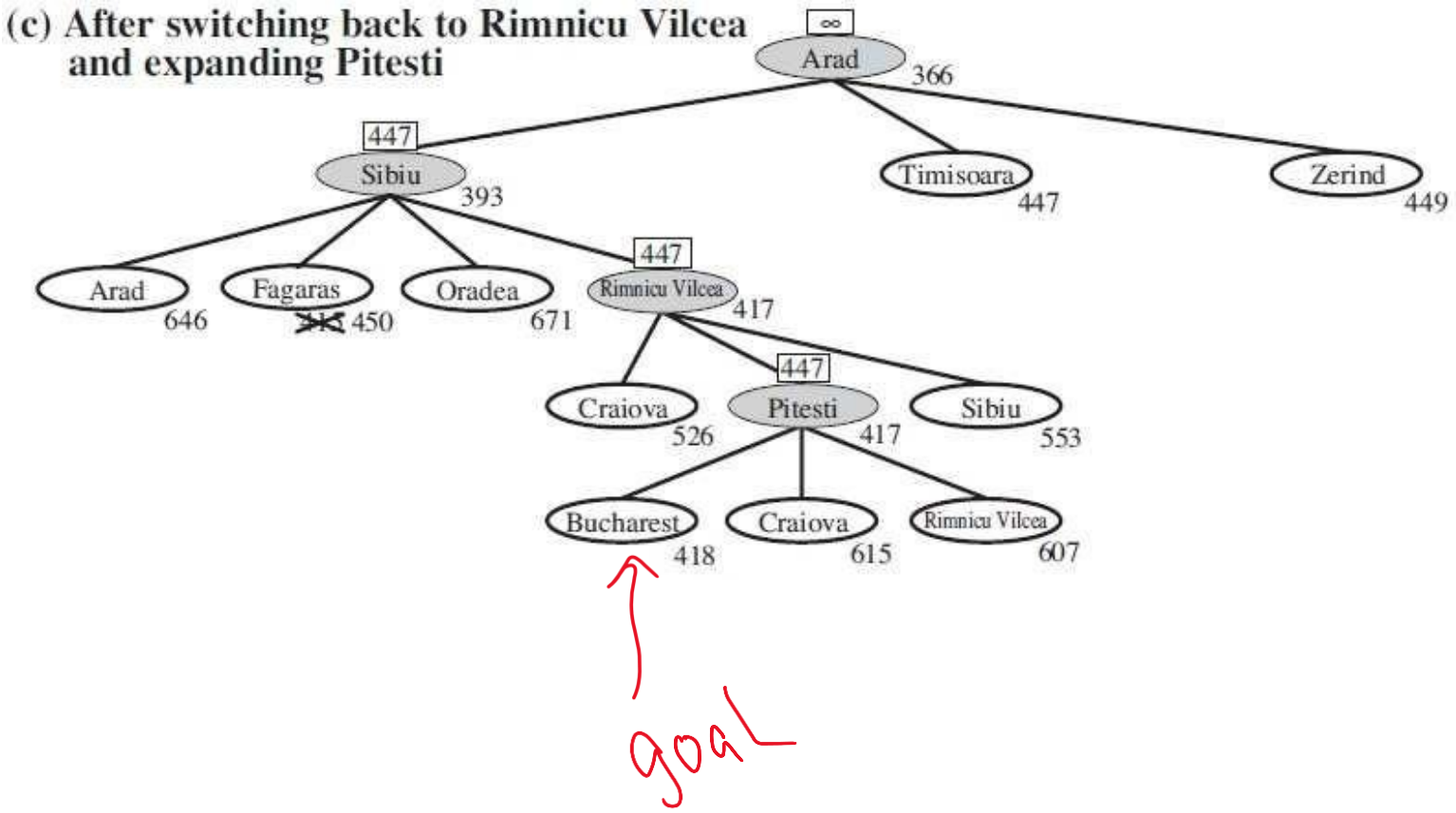
Memory-bounded heuristic search 1



Memory-bounded heuristic search 2



Memory-bounded heuristic search 3



C. Hill Climbing Algorithm

Hill Climbing is a heuristic search algorithm that continuously moves in the direction of increasing value (or improving cost function) — similar to climbing a hill until you reach the top.

- **Core Idea**
  - Start with an **initial state** (current solution).
  - Evaluate all **neighboring states** using a **heuristic function h(n)**.
  - Move to the neighbor with the **best (lowest or highest)** heuristic value.
  - Repeat until:
    - No better neighbor exists → **local maximum/minimum** reached, or
    - The goal is found.

C. Hill Climbing Algorithm

➢ Advantages

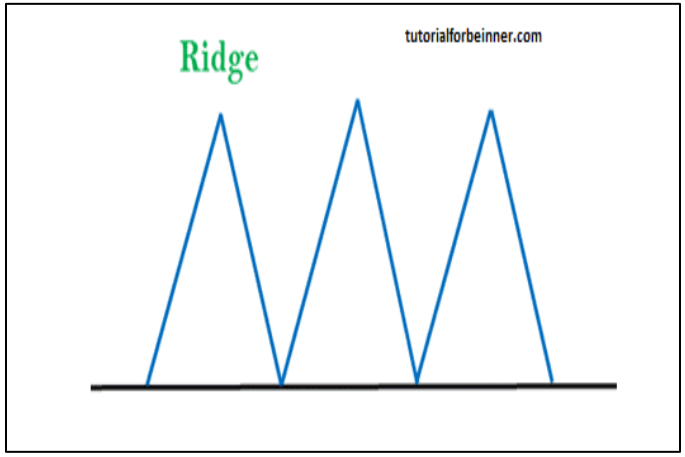
✅ Simple and easy to implement.

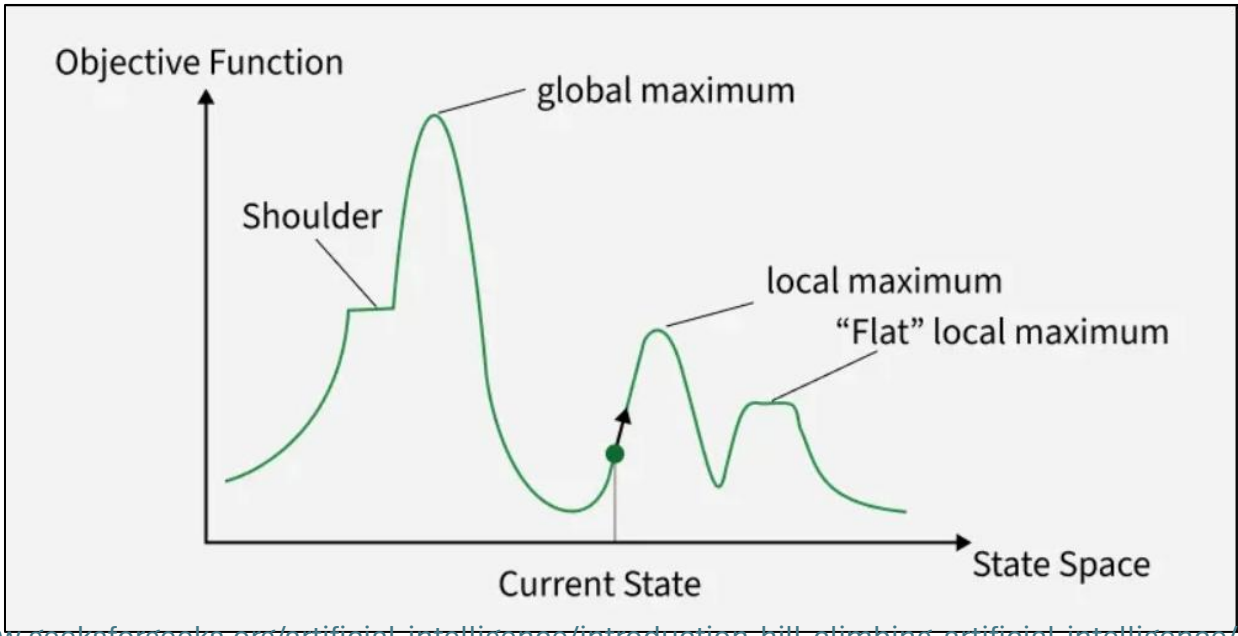
✅ Requires little memory.

➢ Disadvantages

❌ Can get stuck in:

- **Local maxima** – a peak lower than the global maximum.
- **Plateaus** – flat areas where all neighbors have equal value.
- **Ridges** – narrow paths that are hard to climb without sideways moves.





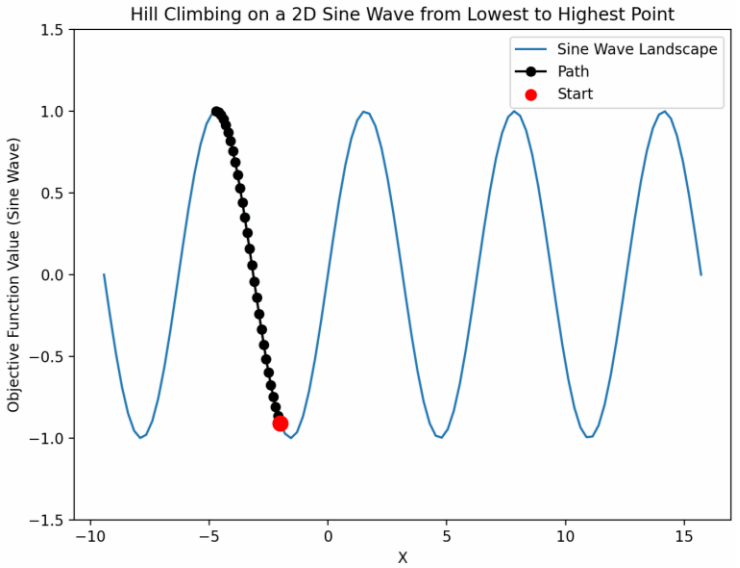
https://www.geeksforgeeks.org/artificial-intelligence/introduction-hill-climbing-artificial-intelligence/

C. Hill Climbing Algorithm

➢ **Applications of Hill Climbing in AI**

- **Pathfinding**: It is used in AI systems that need to navigate or find the shortest path between points such as in robotics or game development.
- **Optimization**: It can be used for solving optimization problems where the goal is to maximize or minimize a particular objective function such as scheduling or resource allocation problems.
- **Game AI:** In certain games, AI uses hill climbing to evaluate and improve its position relative to an opponent's.
- **Machine Learning:** It is sometimes used for hyperparameter tuning where the algorithm iterates over different sets of hyperparameters to find the best configuration for a machine learning model.

https://www.geeksforgeeks.org/artificial-intelligence/introduction-hill-climbing-artificial-intelligence/



Summary

- **Search** methods that an agent can use to select actions in environments that are deterministic, observable, static, and completely known. In such cases, the agent can construct sequences of actions that achieve its goals - search.
- Before an agent can start searching for solutions, a goal must be identified, and a wel- defined **problem must be formulated**.
- A problem: 1) **initial state**, 2) a set of **actions**, 3) a **transition model** describing the results of those actions, 4) a **goal test function**, and 5) a **path cost function**.
- Search algorithms: completeness, optimality, time complexity, and space complexity.
- **Uninformed search**: breadth-first, uniform cost, depth-first, iterative deepening.
- **Informed (heuristic) search**: greedy-best first, A^*, memory bounded

That’s all for Today