CS2109S: Introduction to AI and Machine Learning

Lecture 2:
Solving Problems by Searching

21 January 2024

DO <u>NOT</u> CLOSE YOUR POLLEVERYWHERE APP

There will be activities ahead

New Midterm Timing

- Date/Time: Tuesday, 4 March (Week 7), 6.30pm - 8.00pm
- Venue: MPSH 2A & 2B

Consultation

- **Walk-in:**
  - Office: COM2-02-06
    - If it's open, you can come in
    - If not and the light is on, you can knock
  - Highest availability: Tuesday, 12-3pm
- **Appointment:**
  - Send an email to rizki@nus.edu.sg

Muhammad **Rizki** Maulana

rizki@nus.edu.sg

Deep Reinforcement Learning

Consultation

- **Walk-in:**
  - Office: COM2-04-29
  - Monday, 2-4pm

- **Appointment:**
  - Send an email to cqtfpr@nus.edu.sg

Patrick Rebentrost
cqtfpr@nus.edu.sg
Quantum Machine Learning

For those who are just enrolled

- Please watch the recording or read lecture slides for Lecture 1
- **Important:** Course Policies (also available in Canvas)

Outline

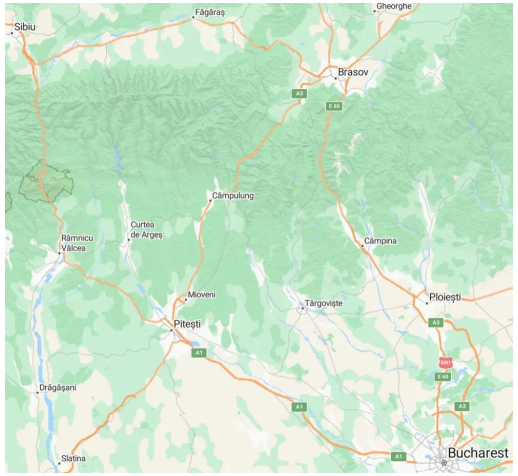
- Designing an Agent
  - Problem solving agents
  - Problem formulation
- Search
  - Uninformed Search
    - Breadth-first, depth-first, and uniform-cost search
    - Depth-limited and iterative-deepening search
  - Informed Search
    - A* search
    - Heuristics

Outline

- **Designing an Agent**
  - Problem solving agents
  - Problem formulation
- Search
  - Uninformed Search
    - Breadth-first, depth-first, and uniform-cost search
    - Depth-limited and iterative-deepening search
  - Informed Search
    - A* search
    - Heuristics

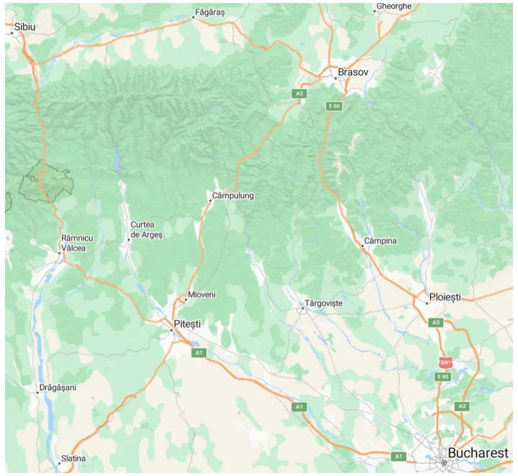
Preliminary: Search Problems

- A search problem refers to a type of problem where the goal is to find a state (or a path to a state) from a set of possible states by exploring various possibilities.
- Examples:
  - Path finding
  - Puzzle solving



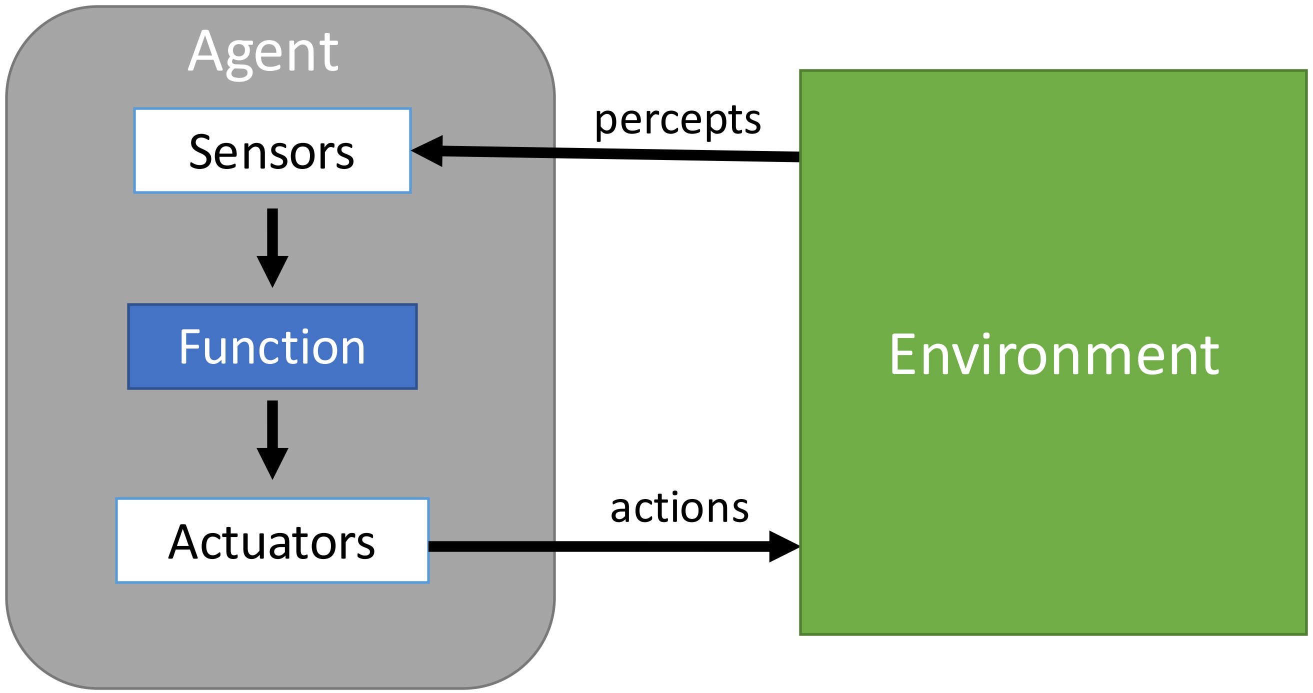
Preliminary: Search Algorithms

- A search algorithm takes in a search problem as input and returns a solution / failure.
- A solution of a search could be a sequence of actions or final state
  - An optimal solution is a solution with least cost, or least number of steps if there is no cost associated with actions
- Search algorithms
  - Linear search (bruteforce)
  - Djikstra (CS2040S)
  - …



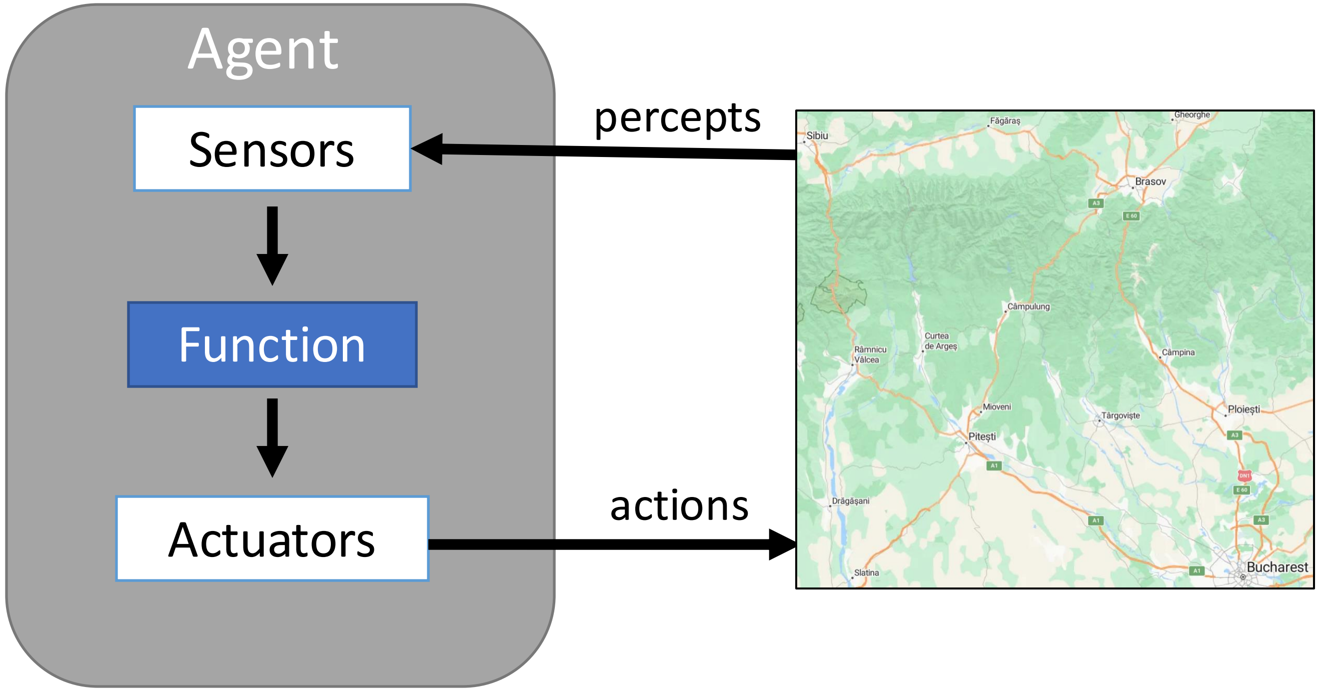
9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Designing an Agent



Designing an Agent for problems that can be solved by **searching**. E.g., path finding

Goal-based agent



Fully-observable
Deterministic
Static
Discrete

Only need to look once

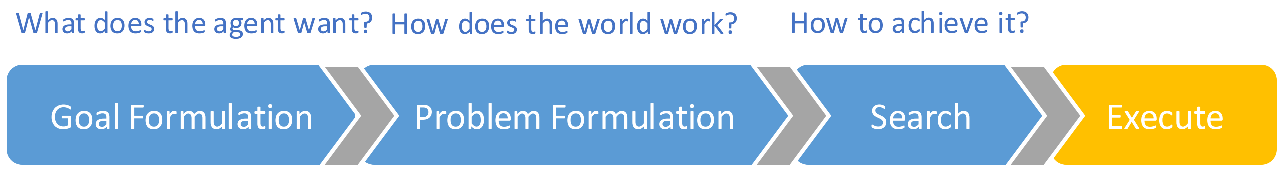
How is search used by the agent?

Designing an Agent

To solve a problem using search, the agent needs to have:

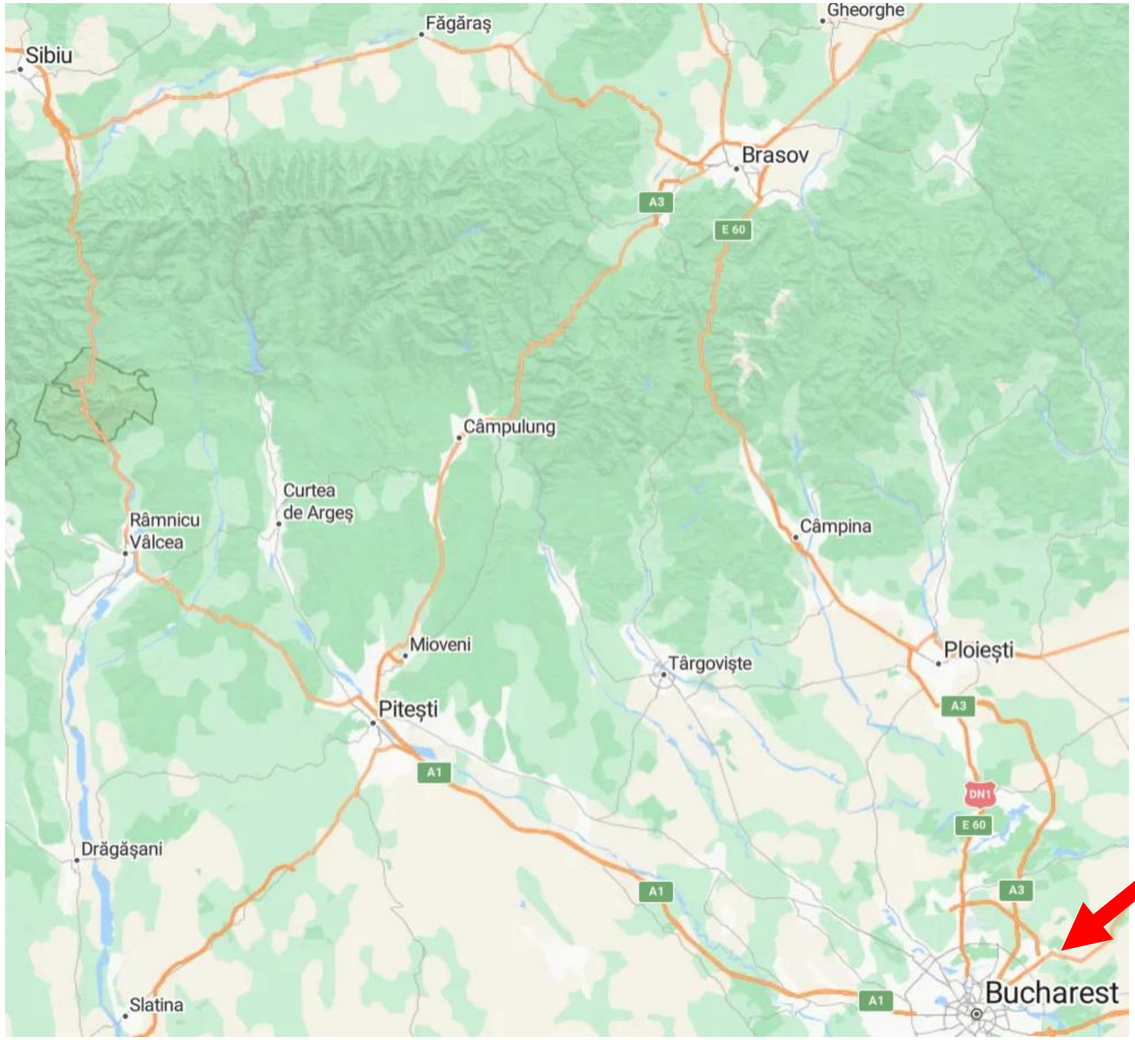
- A goal, or a set of goals
- **A model of the environment**
- **A search algorithm**

Problem Solving Steps



Example: Romania

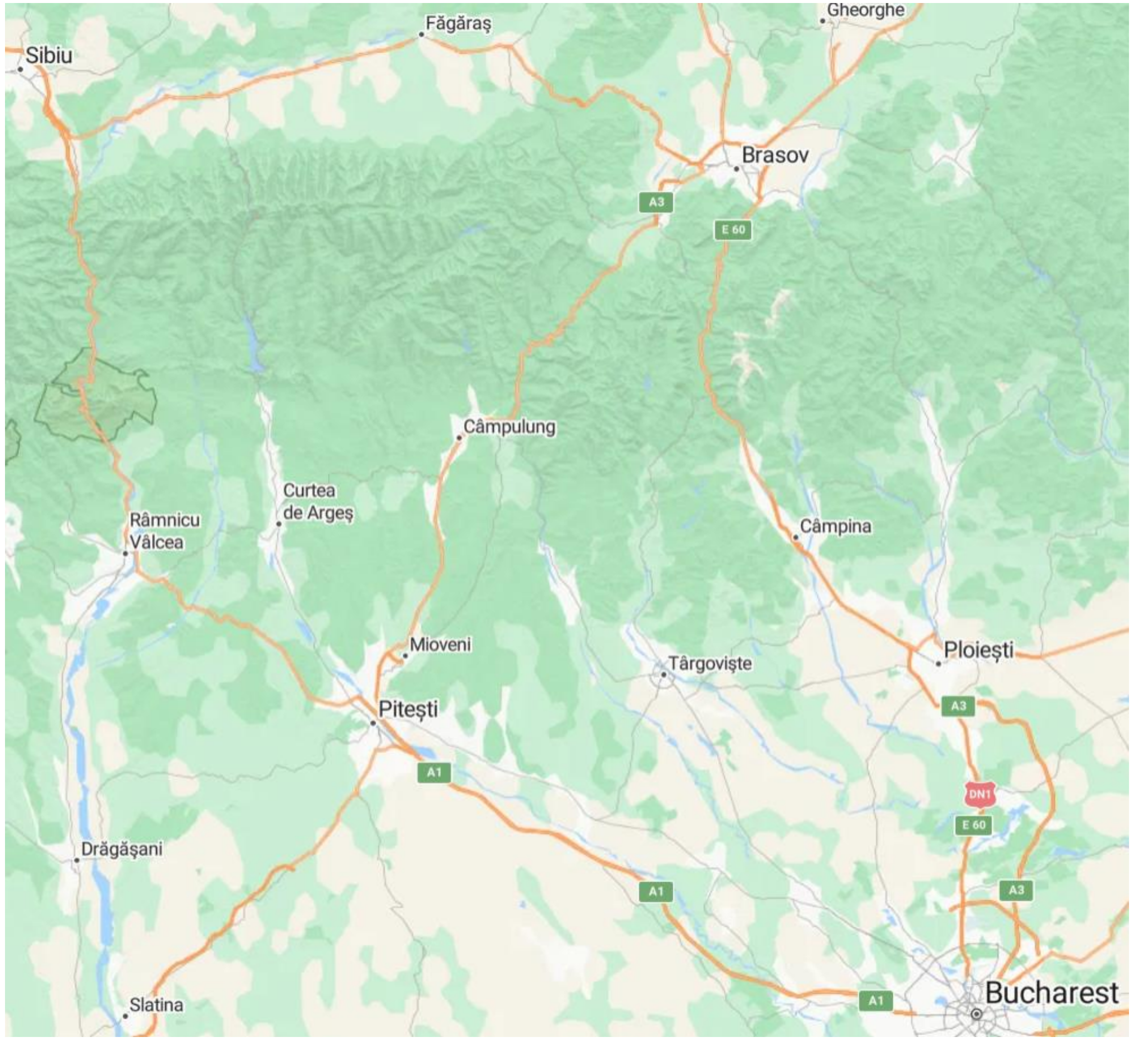
Goal Formulation
Problem Formulation
Search
Execute

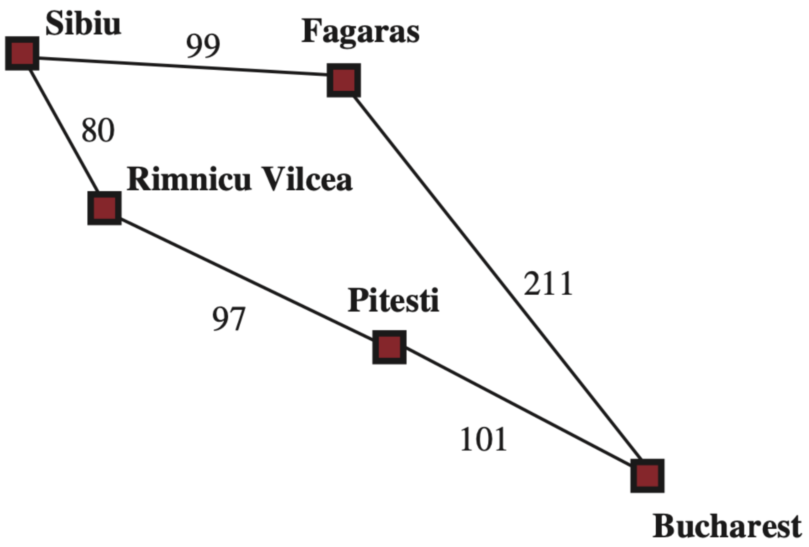


Go to Bucharest

Example: Romania

Goal Formulation → **Problem Formulation** → Search → Execute



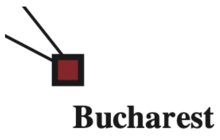


Problem Formulation

- States — Cities: {at Sibiu, at Pitesti, ...}
- Initial state — at Sibiu
- Goal state(s)/test — at Bucharest
- Actions — Go to a neighboring city
- Transition model — Current state = selected city
- Action cost function — See edges

Important facts

Representation invariant: ensure that the *abstract states* have corresponding *concrete states*.



Credit: Passport & Plates

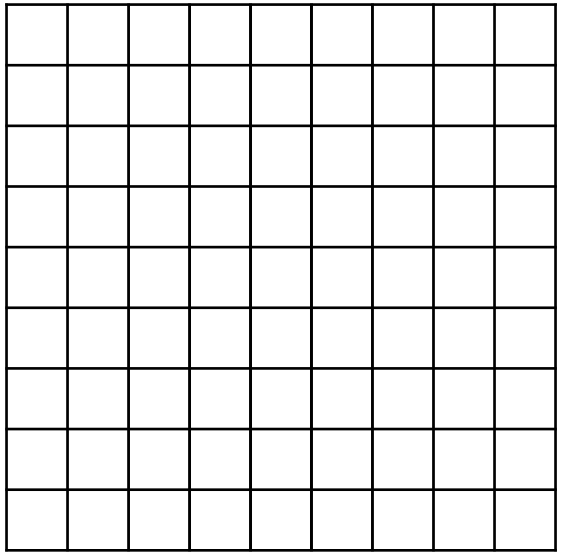
Goal test: Goal defined via a function such as is_goal(state)

Actions: a set actions(state) with size |actions(state)|≤ b (branching factor)

Transition model: *new_state = transition(state, action)*

Example: Sudoku

Goal Formulation → **Problem Formulation** → Search → Execute



9x9 Integer Matrix

Problem Formulation

- States: All possible assignments of numbers to a 9x9 matrix
- Initial state: Partially filled matrix
- Goal state(s)/test: Check if all constraints are satisfied
- Actions: Fill matrix
- Transition model: Current state = filled matrix
- Action cost function: 1

Outline

- Designing an Agent
  - Problem solving agents
  - Problem formulation
- **Search**
  - Uninformed Search
    - Breadth-first, depth-first, and uniform-cost search
    - Depth-limited and iterative-deepening search
  - Informed Search
    - A* search
    - Heuristics

Review: Data structures

Required operations

- Add element (“add")
- Remove element ("pop")
- Is empty

Credit: freepik.com

Queue

First-in-first-out (FIFO)

Credit: freepik.com

Priority Queue

High-priority-first-out

Credit: iStock

Stack

Last-in-first-out (LIFO)

Search

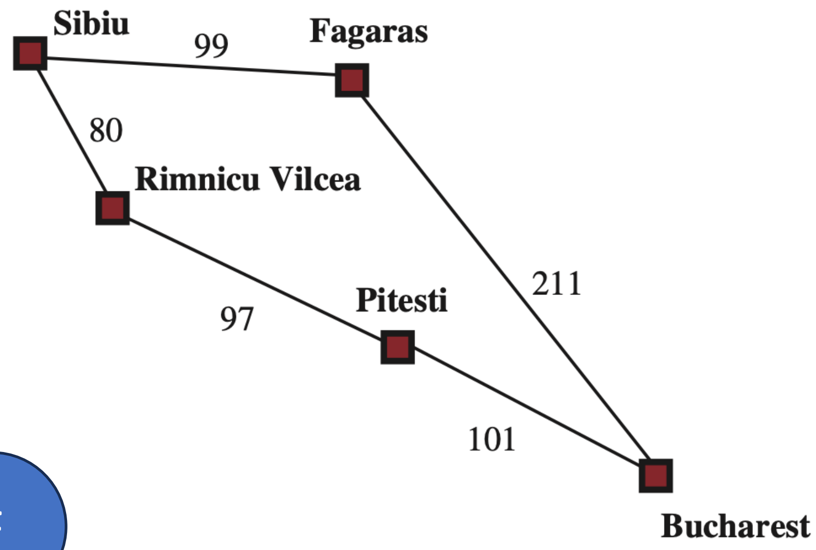
AIAMA: tree search

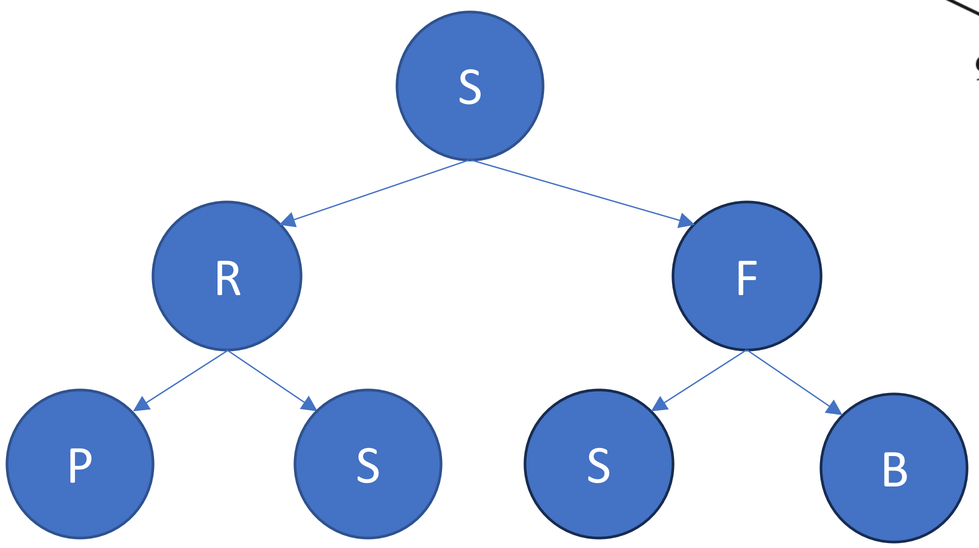
Goal Formulation > Problem Formulation > **Search** > Execute

```
create frontier

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution


   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

Can be queue, priority queue, stack

Frontier defines the search

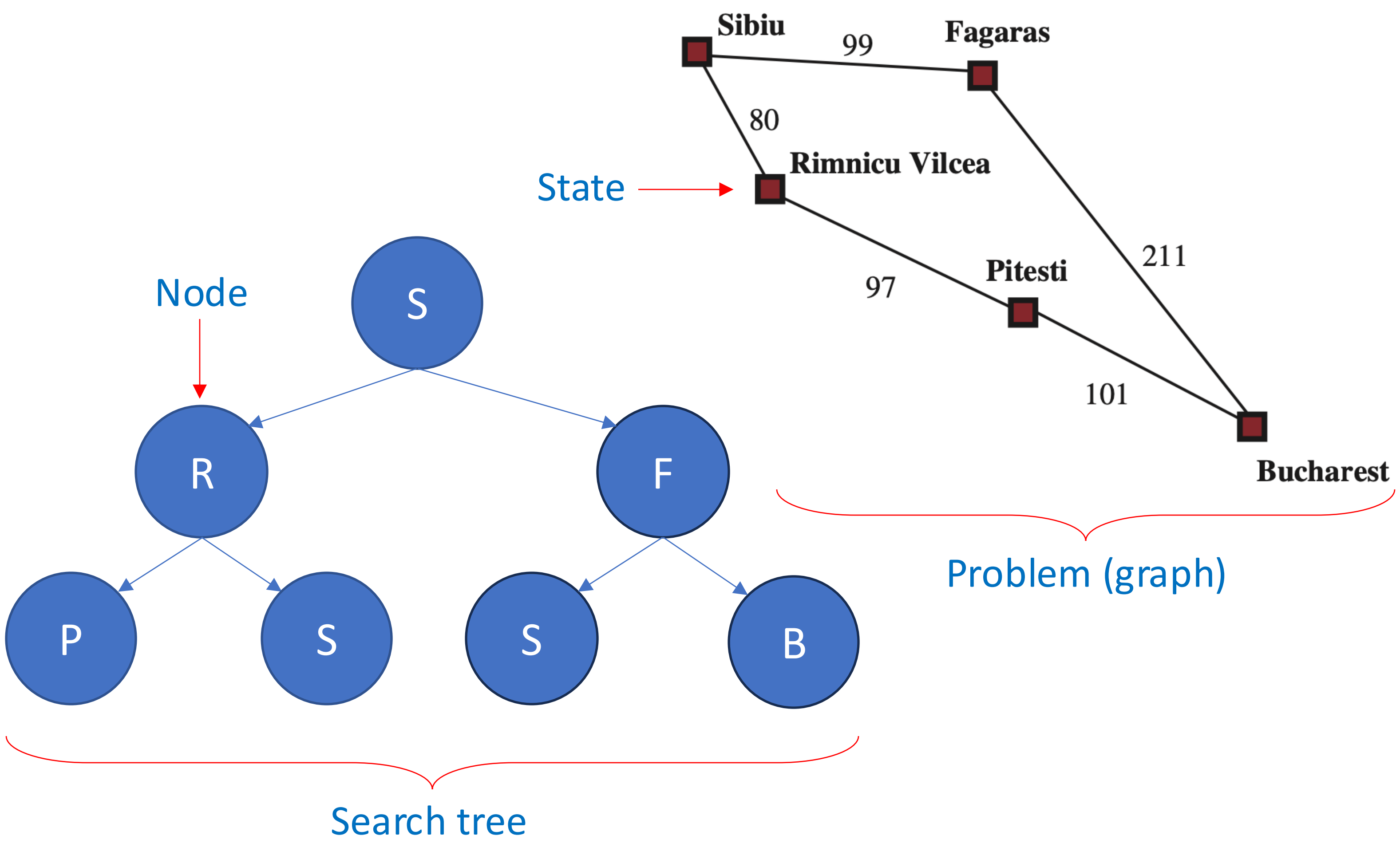




Search Terms

Path cost is the cost of a path from any state to any state

Optimal path cost is the cost of the lowest-cost path from any state to any state



Evaluation Criteria

(Worst-case) Complexity:

- Time complexity: number of nodes generated or expanded
- Space complexity: maximum number of nodes in memory
- Measure: branching factor, optimal depth, maximum depth, …

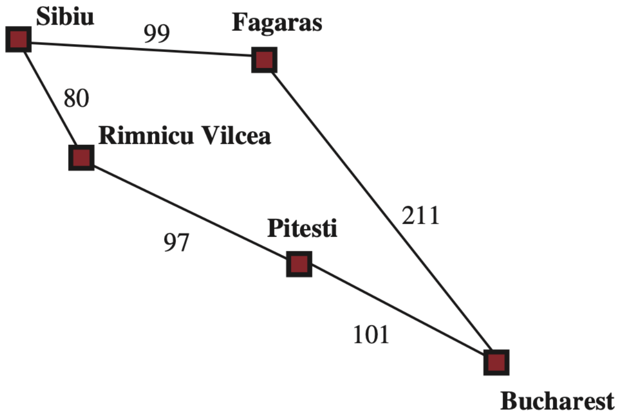
Disclaimer: CS2109S is not an algorithm-oriented course, so we will not discuss this in detail

Others:

- Completeness
- Optimality

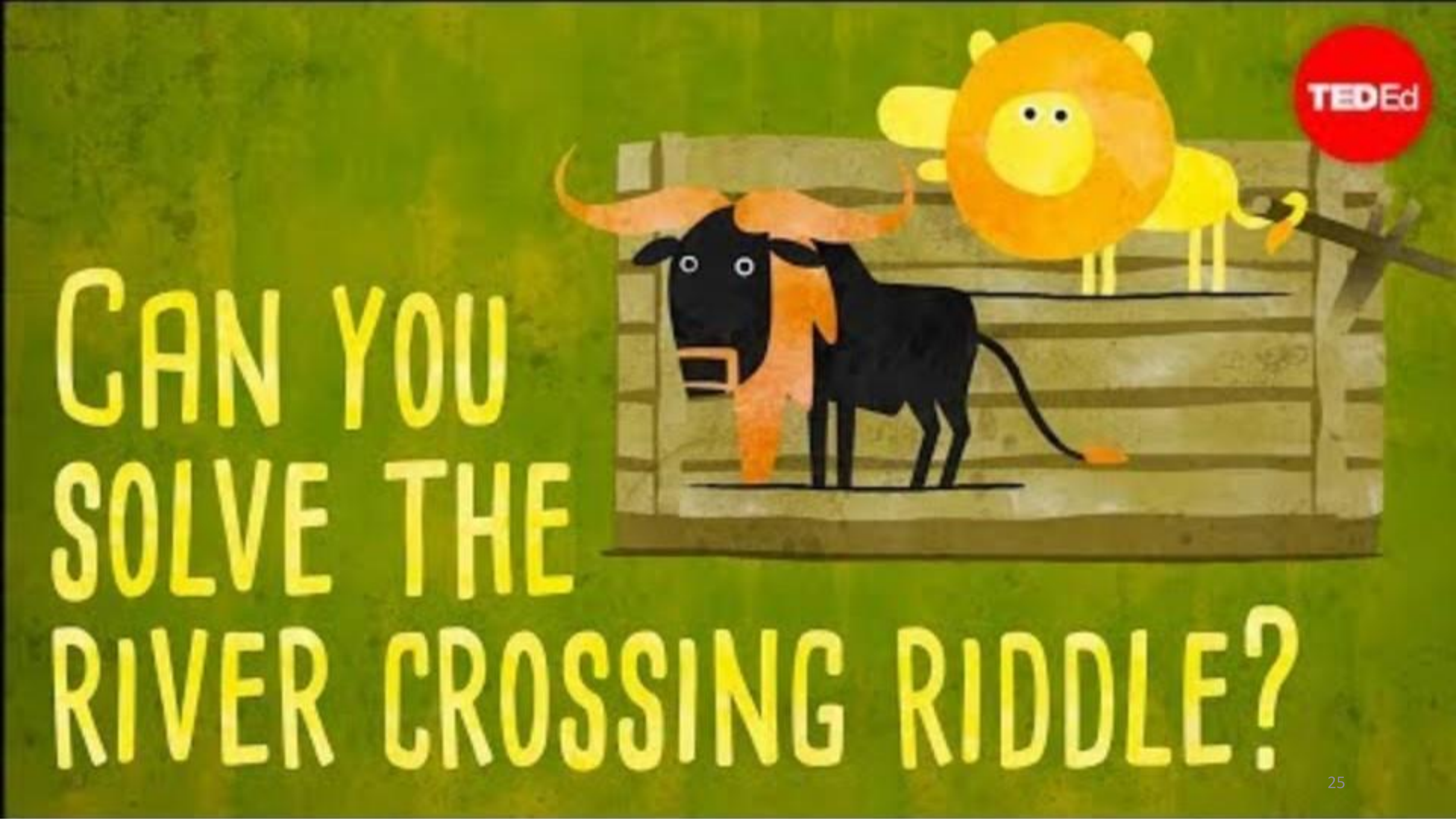
Completeness and Optimality

- **Complete**: An algorithm is complete if, for every problem instance, it will find a solution if one exists. Conversely, an algorithm is incomplete if there exists at least one instance with a solution that the algorithm fails to find.
- **Optimal**: An algorithm is optimal if, for every instance where it produces a solution, that solution is the best possible. In other words, if the algorithm outputs a solution, it is guaranteed to be optimal.
- **Optimal and Incomplete**: An algorithm can be both optimal and incomplete. This means that there exists at least one instance where the algorithm fails to find a solution, even though, when it does find a solution, that solution is guaranteed to be optimal.




```
def search(state):
    if state == "Fagaras":
        return "go to Bucharest"
    return None
```

Break

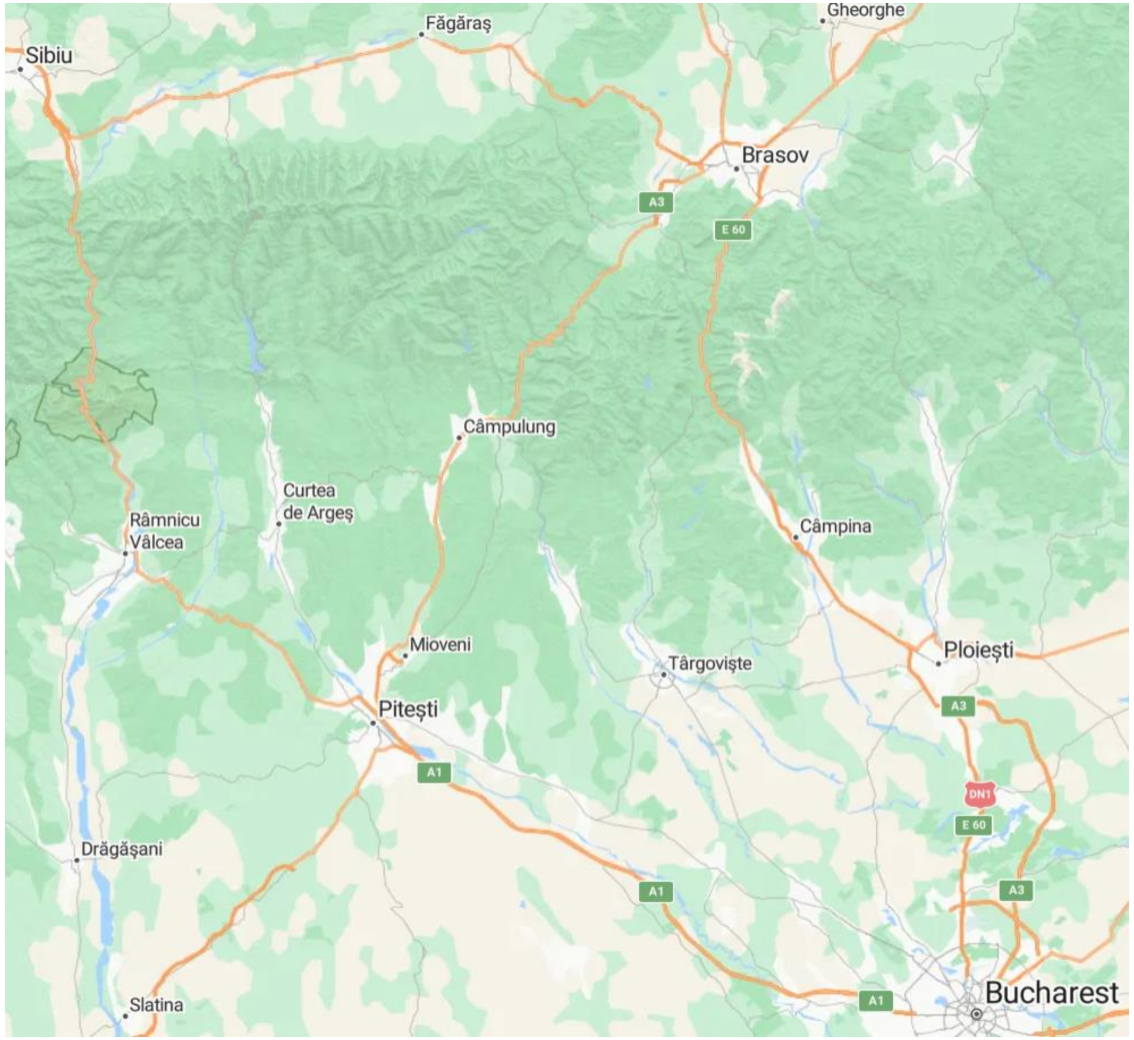

TED Ed
CAN YOU
SOLVE THE
RIVER CROSSING RIDDLE?

Outline

- Designing an Agent
  - Problem solving agents
  - Problem formulation
- Search
  - **Uninformed Search**
    - **Breadth-first, depth-first, and uniform-cost search**
    - Depth-limited and iterative-deepening search
  - Informed Search
    - A* search
    - Heuristics

Uninformed Search Algorithms

No information that could guide the search



Blind search

No clue how good a state is,

e.g., how close it is to the goal

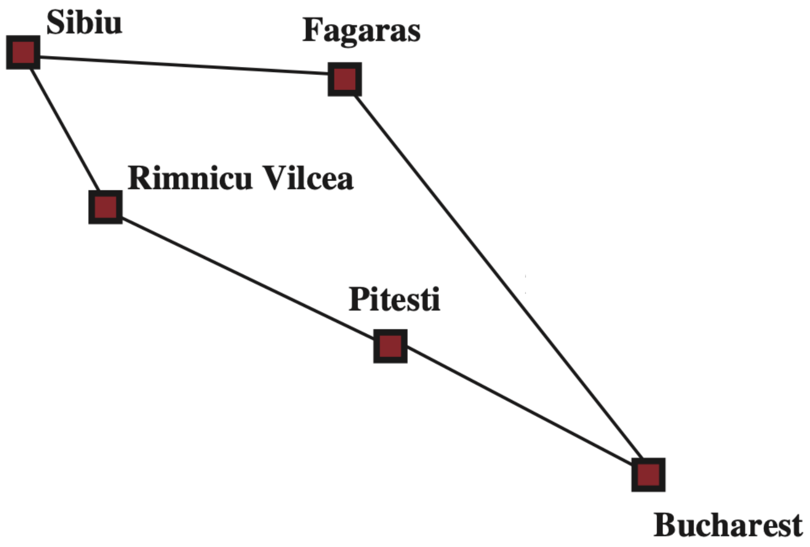
Search

AIAMA: tree search

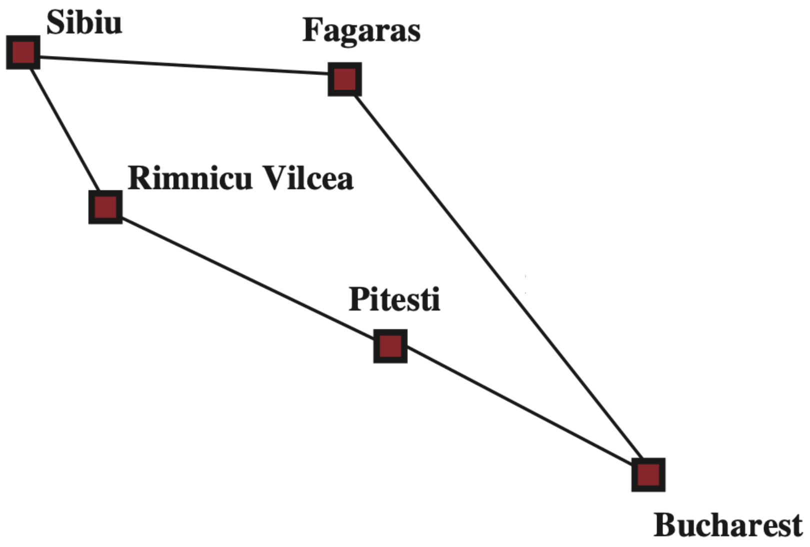
```
create frontier


insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal: return solution



    for action in actions(node.state):
        next_state = transition(node.state, action)
        frontier.add(Node(next_state))
return failure
```

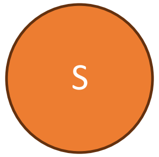
Breadth-first Search (BFS)

```
create frontier : queue

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```



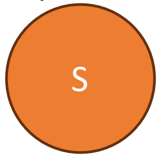
Breadth-first Search (BFS)



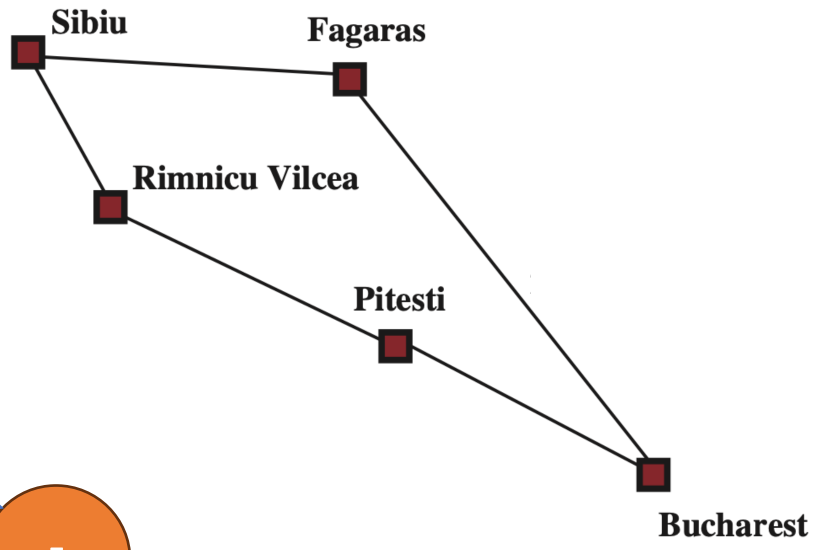


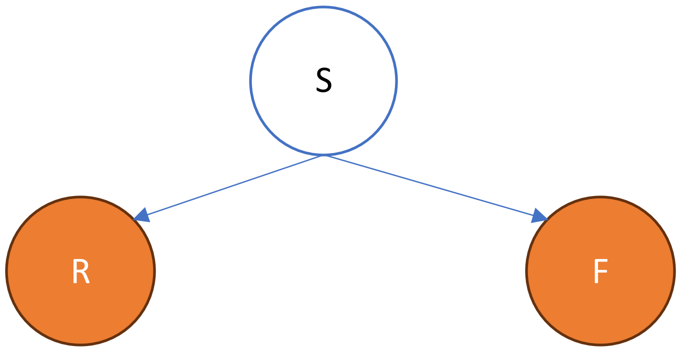

```
create frontier : queue

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

Queue:



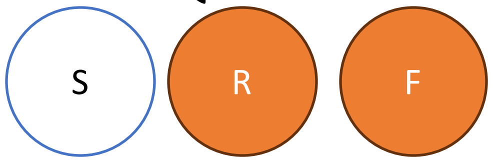
Breadth-first Search (BFS)



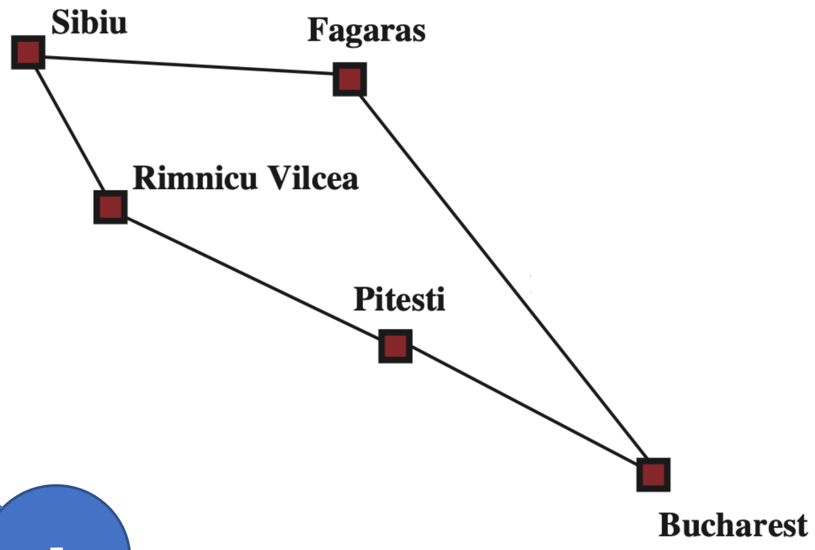



```
create frontier : queue

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

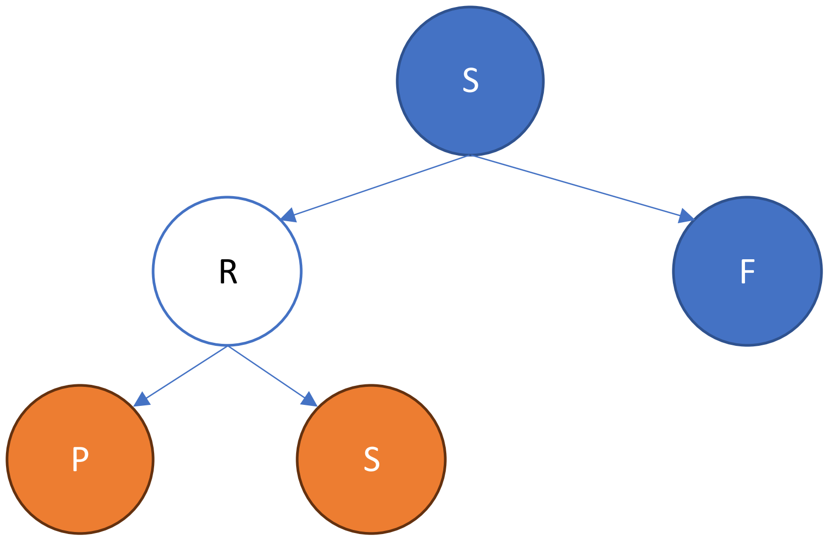
Queue:



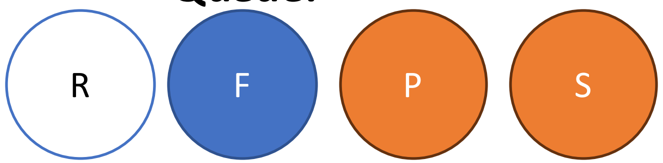
Breadth-first Search (BFS)



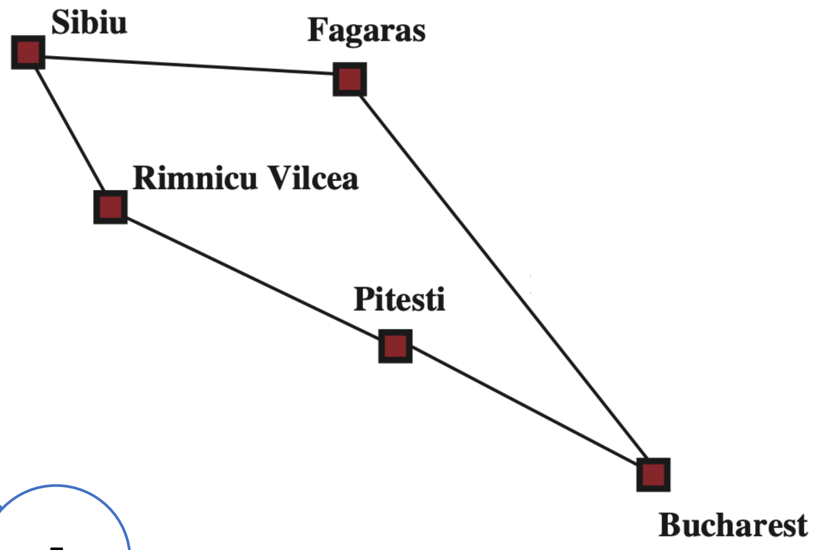

```
create frontier : queue

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution


   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```



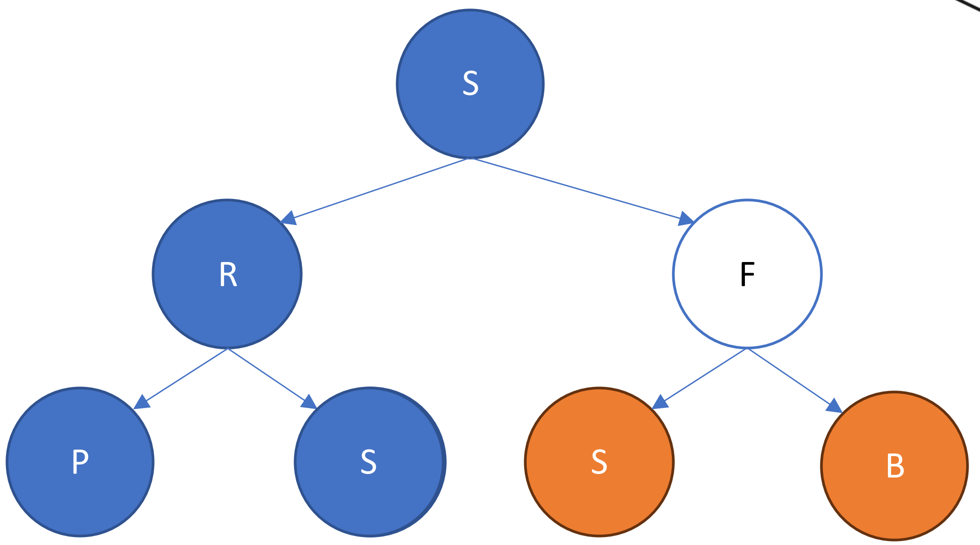
Queue:



Breadth-first Search (BFS)

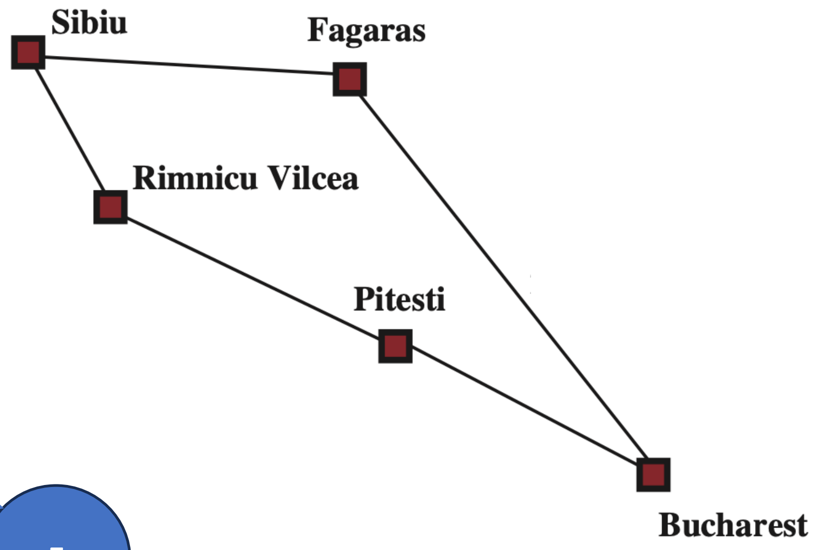



```
create frontier : queue

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution


   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

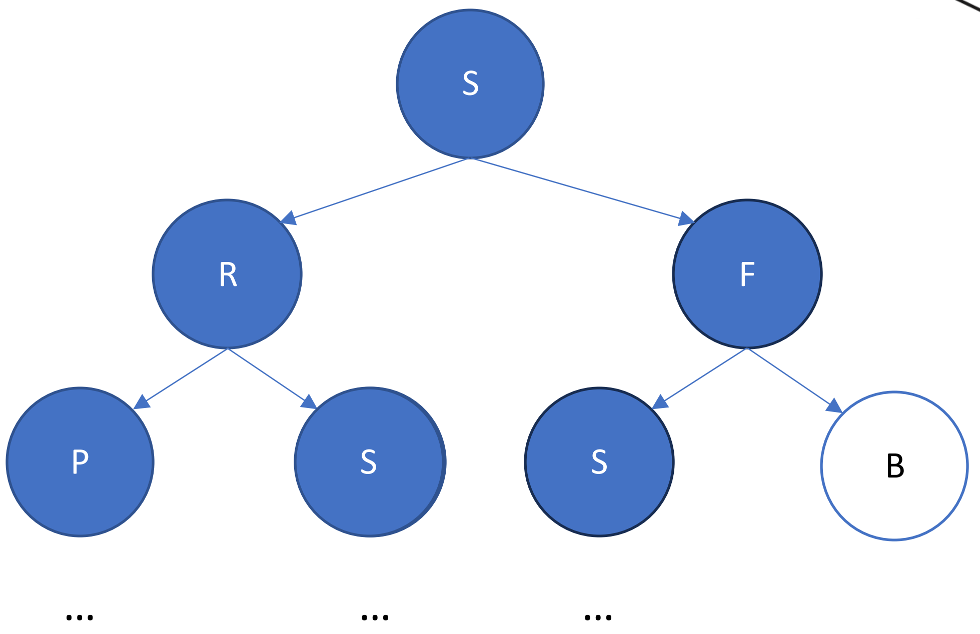


Queue:

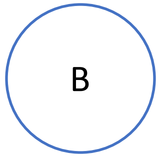
Breadth-first Search (BFS)




```
create frontier : queue

insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal: return solution


    for action in actions(node.state):
        next_state = transition(node.state, action)
        frontier.add(Node(next_state))
return failure
```



Queue:

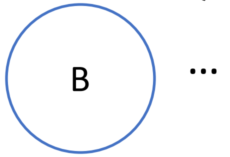


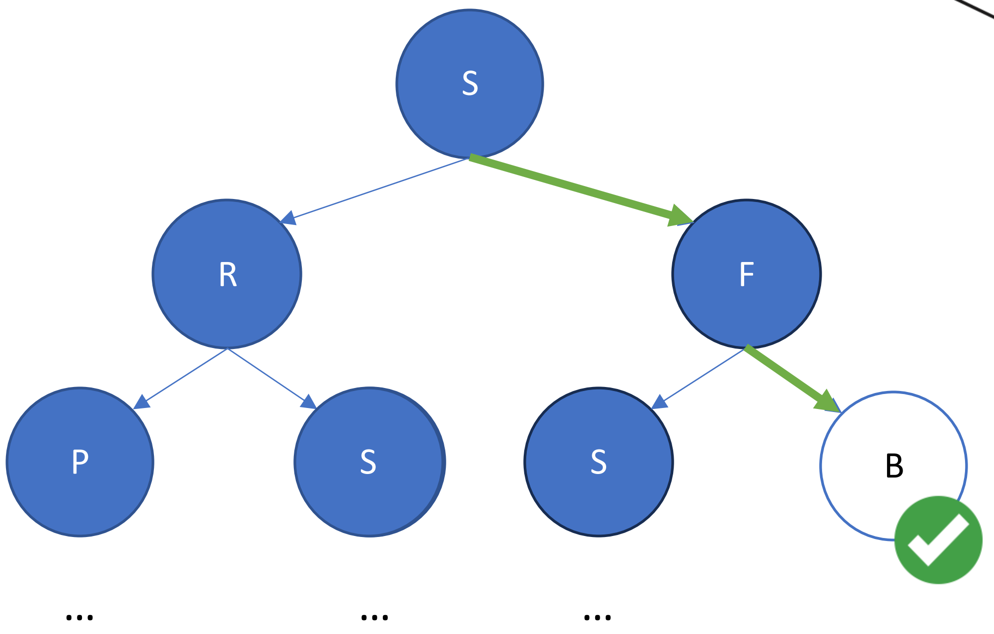
…

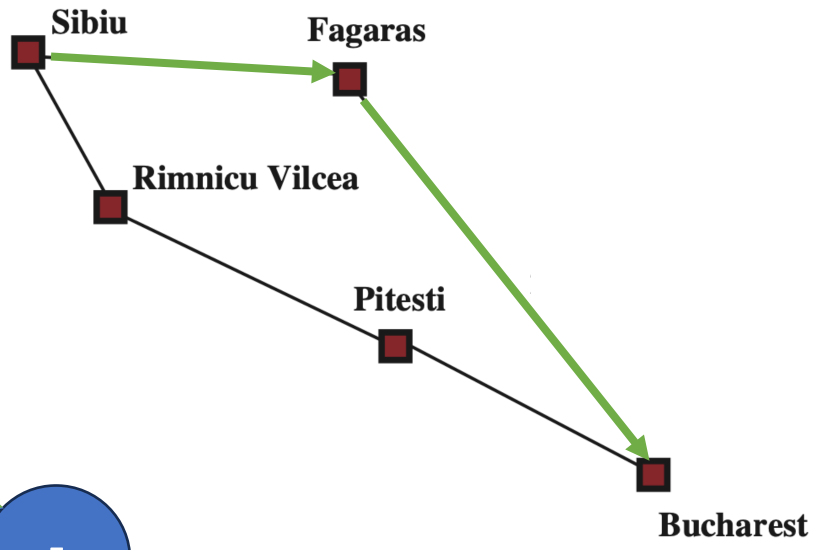
Breadth-first Search (BFS)

```
create frontier : queue

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

Queue:







Breadth-first Search (BFS) - Analysis

```
create frontier : queue

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution


   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

Time complexity: (# nodes generated)
Exponential w.r.t. the depth of the optimal solution

Space complexity: (size of frontier)
Exponential w.r.t. the depth of the optimal solution

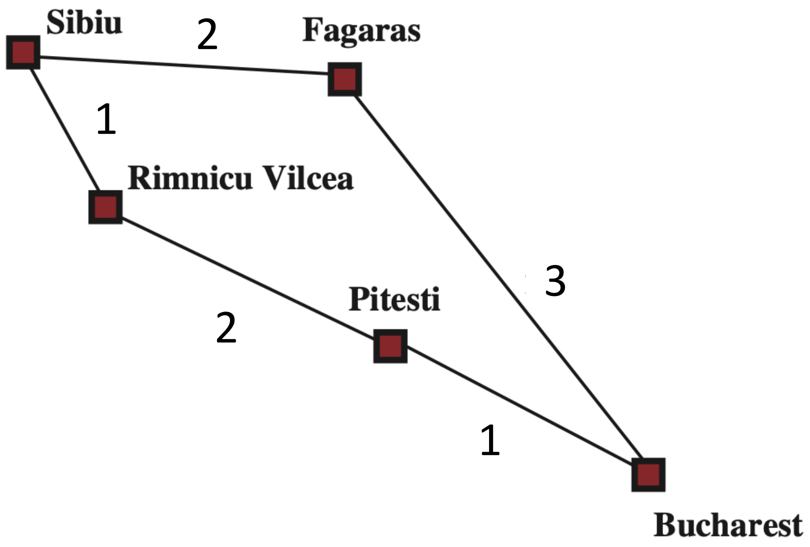
Complete: Yes, if b is finite

Optimal: Yes, if step cost is the same everywhere

Uniform-cost Search (UCS)

Cost from root to a state

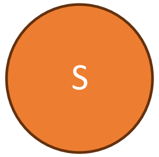
```
create frontier : priority queue (path cost)


insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

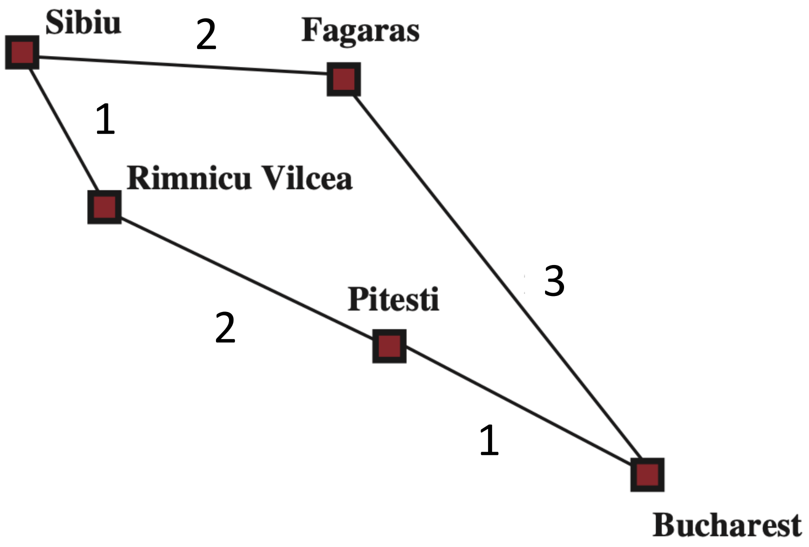


Uniform-cost Search (UCS)

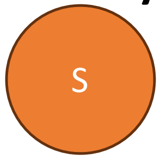
Cost from root to a state

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```





Priority Queue:

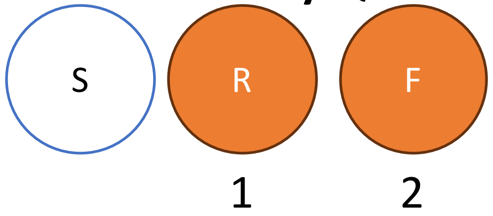


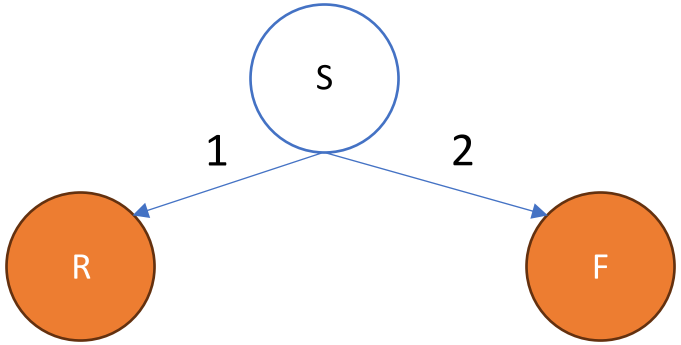
Uniform-cost Search (UCS)

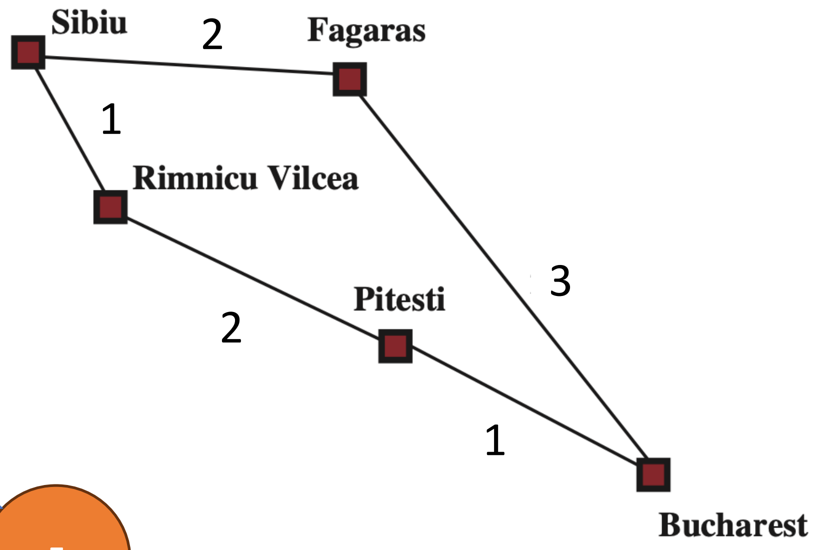
Cost from root to a state

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution


   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

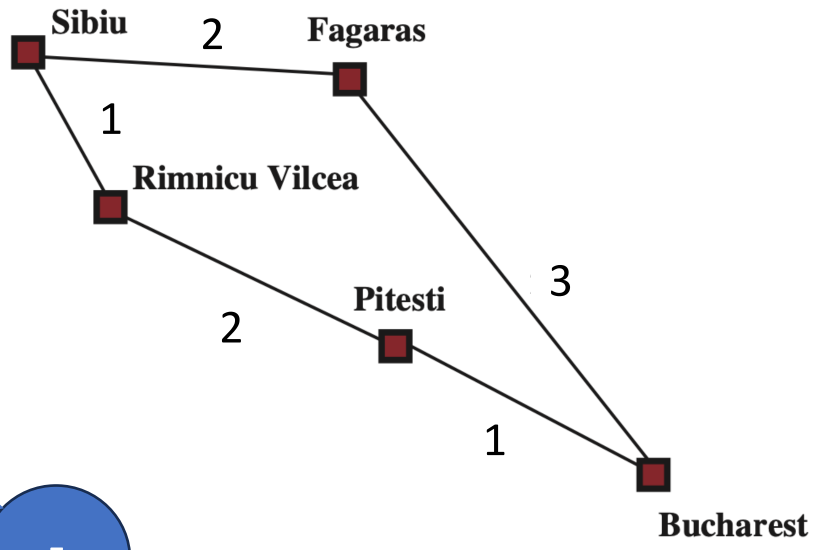
Priority Queue:





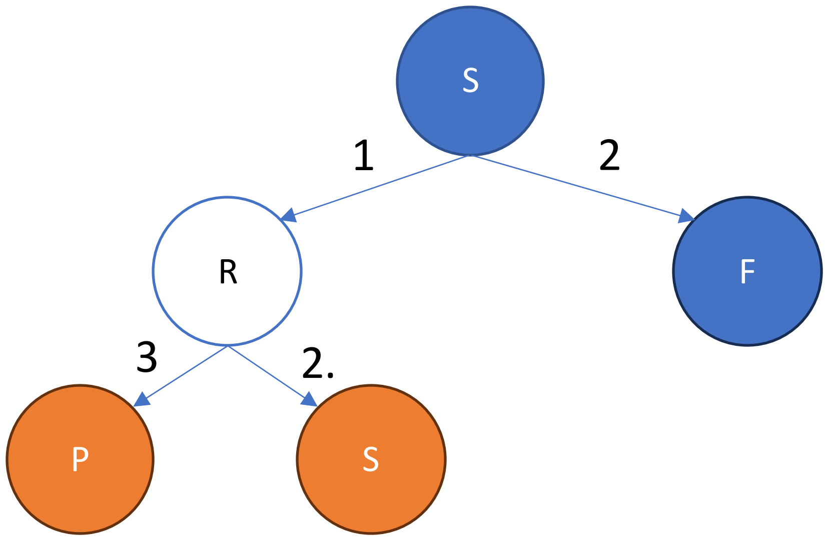


Uniform-cost Search (UCS)

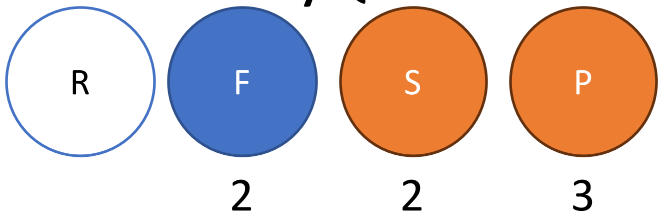


Cost from root to a state

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution


   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```



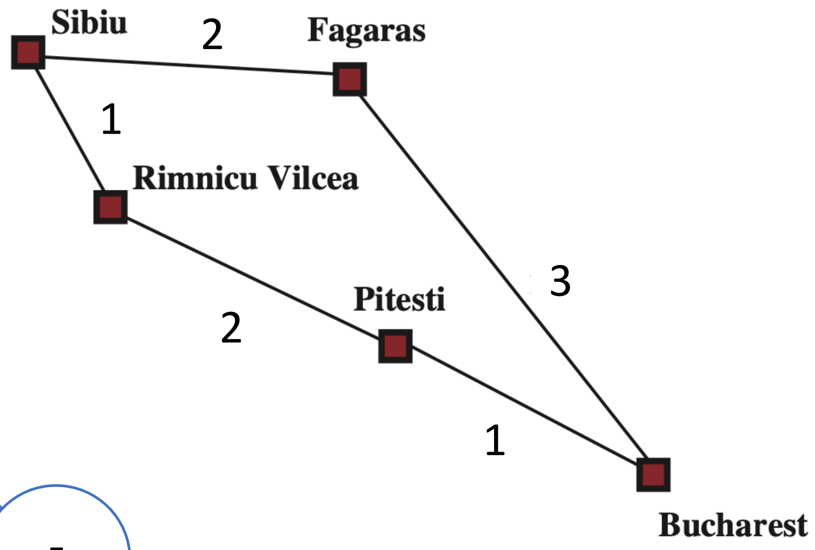
Priority Queue:

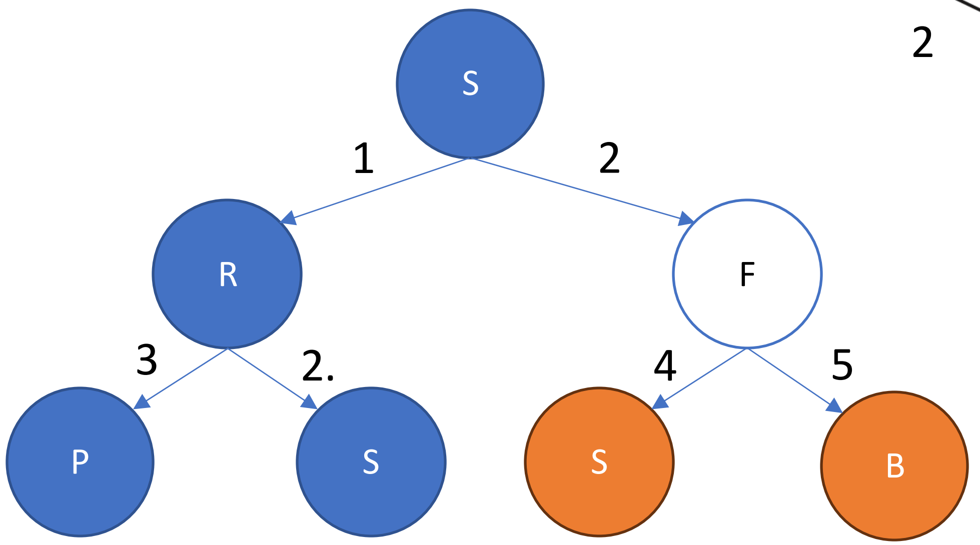


Uniform-cost Search (UCS)

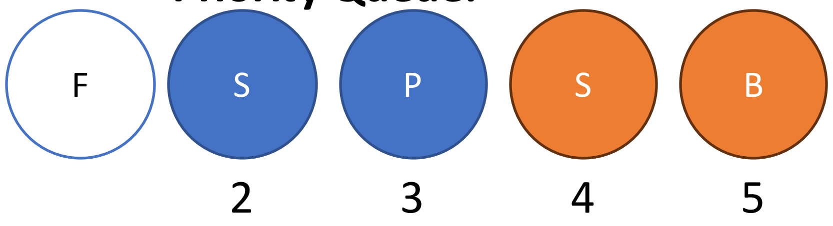
Cost from root to a state

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```





Priority Queue:

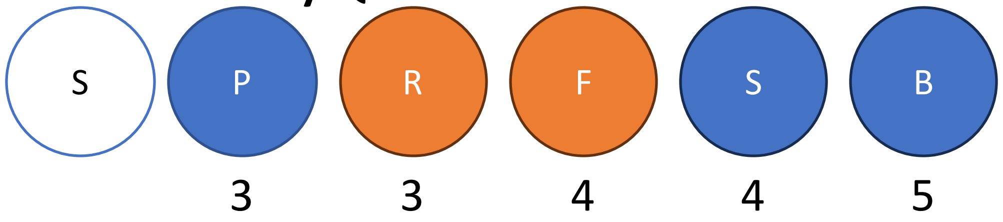


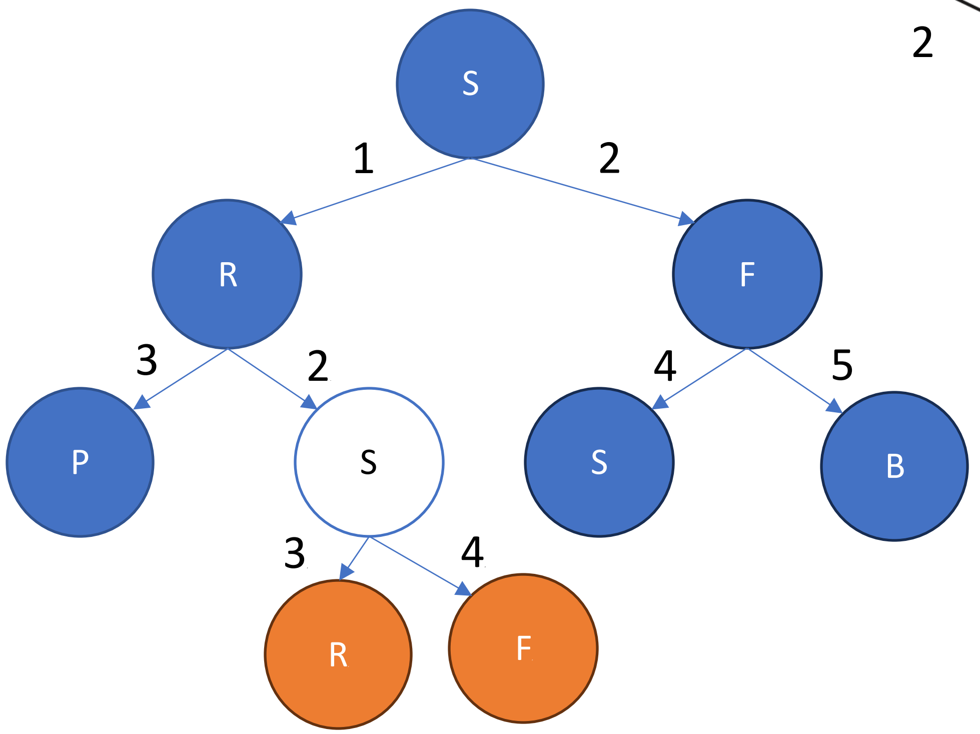
Uniform-cost Search (UCS)

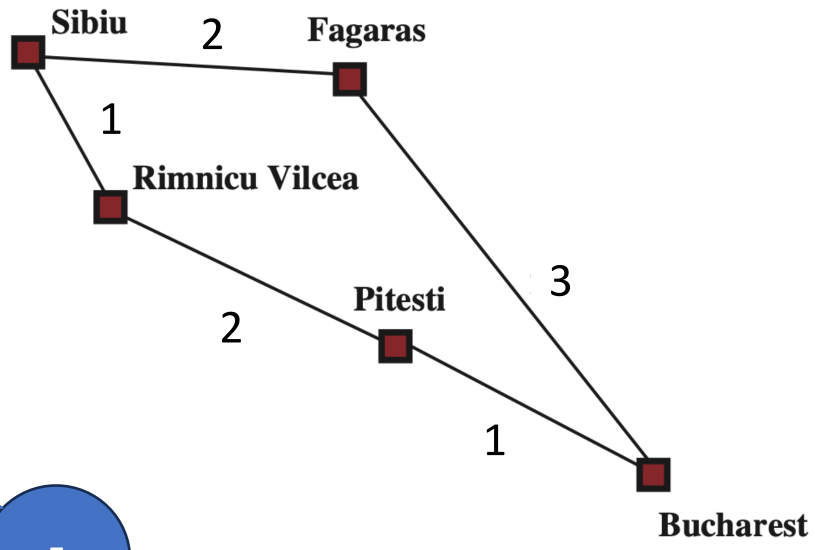
Cost from root to a state

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

Priority Queue:



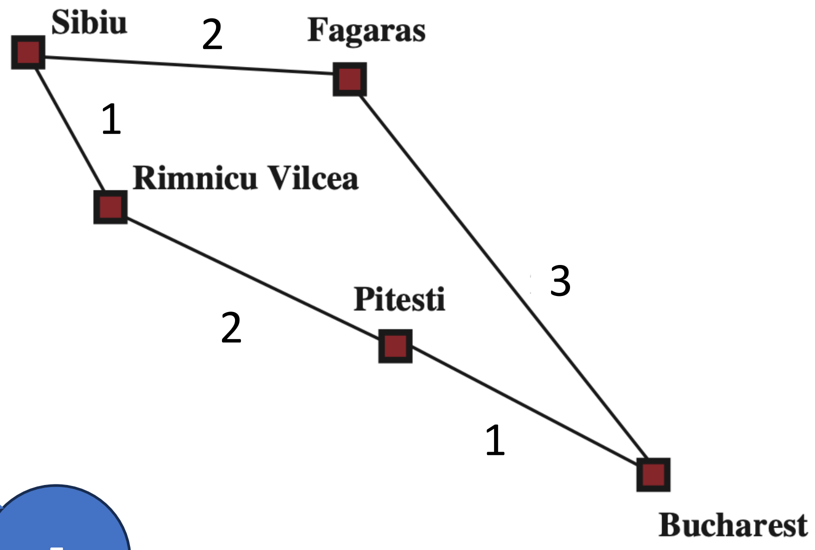


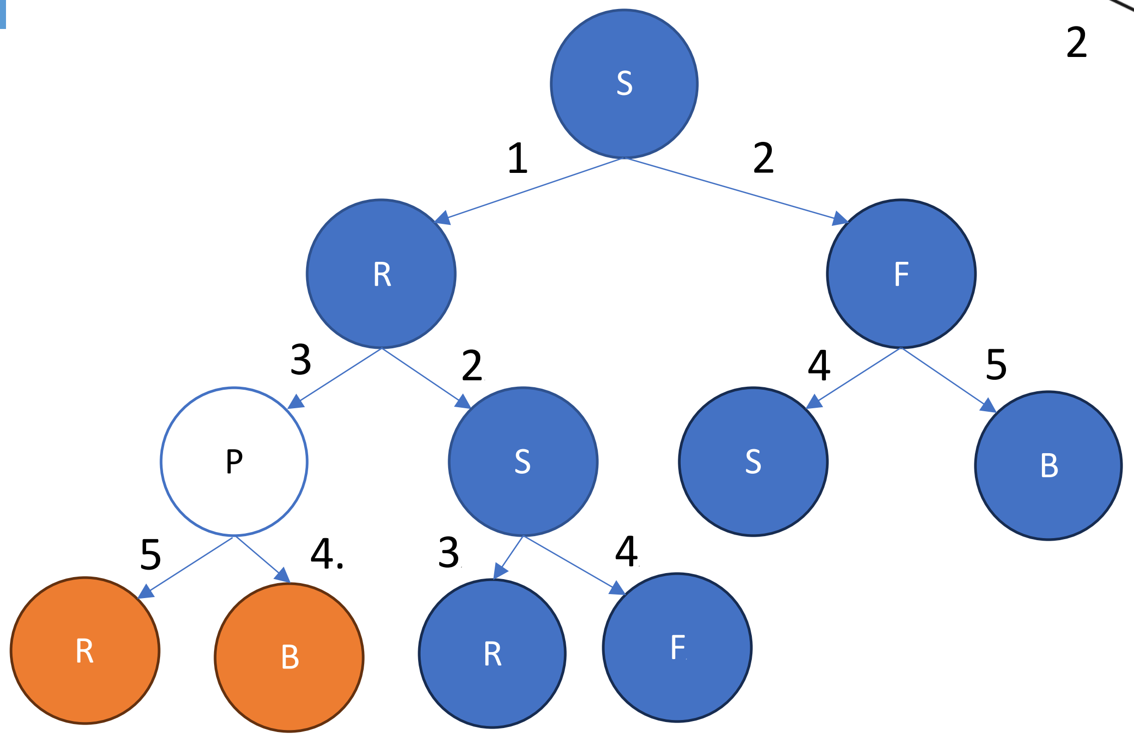


Uniform-cost Search (UCS)

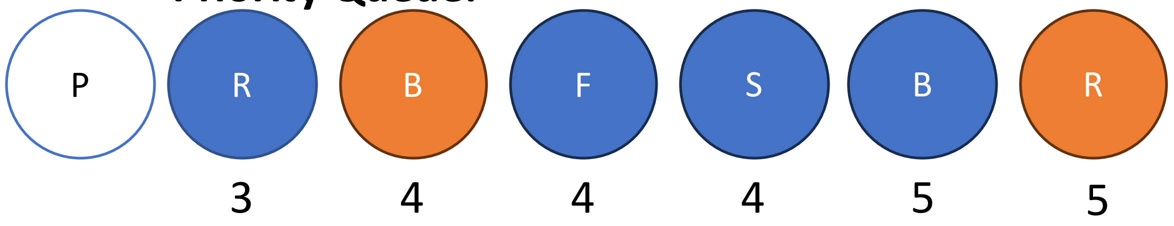
Cost from root to a state

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution

   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```





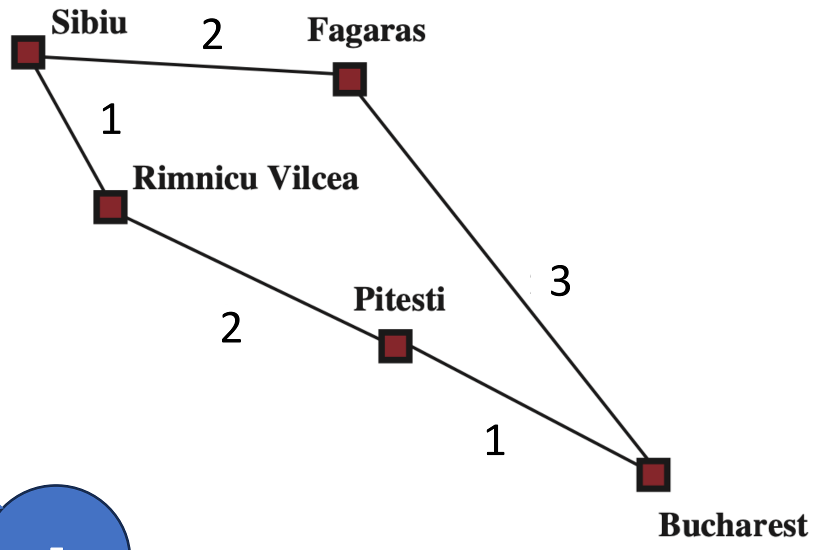
Priority Queue:

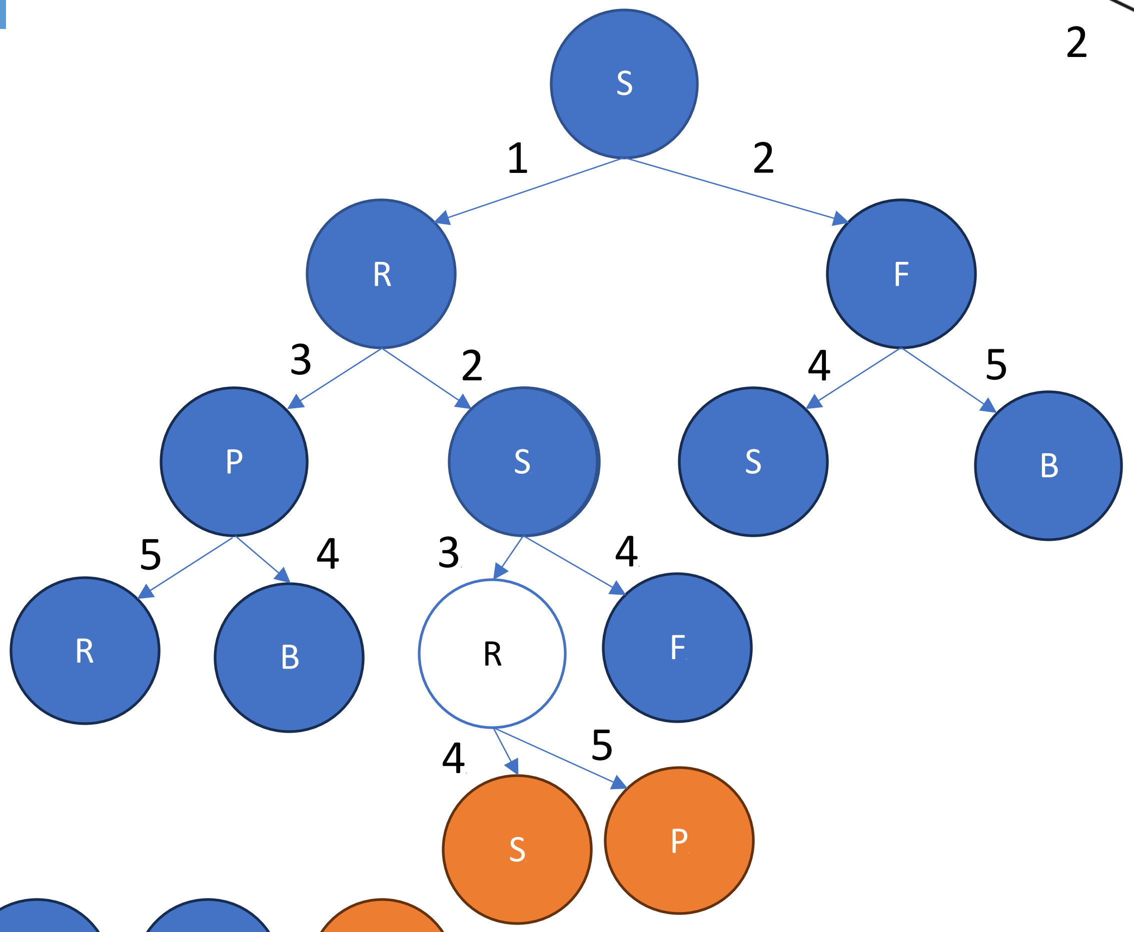


Uniform-cost Search (UCS)

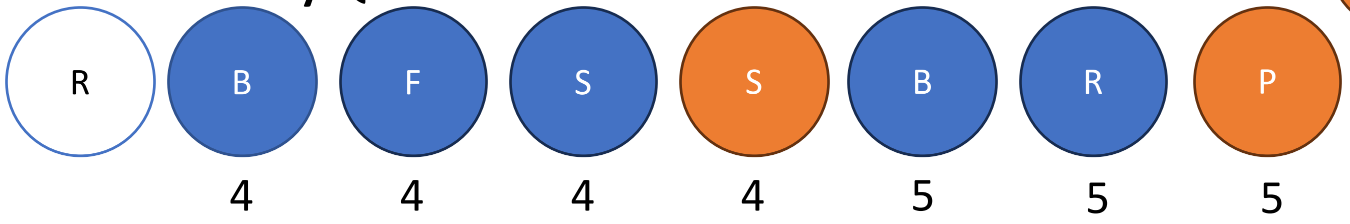
Cost from root to a state

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal: return solution


    for action in actions(node.state):
        next_state = transition(node.state, action)
        frontier.add(Node(next_state))
return failure
```





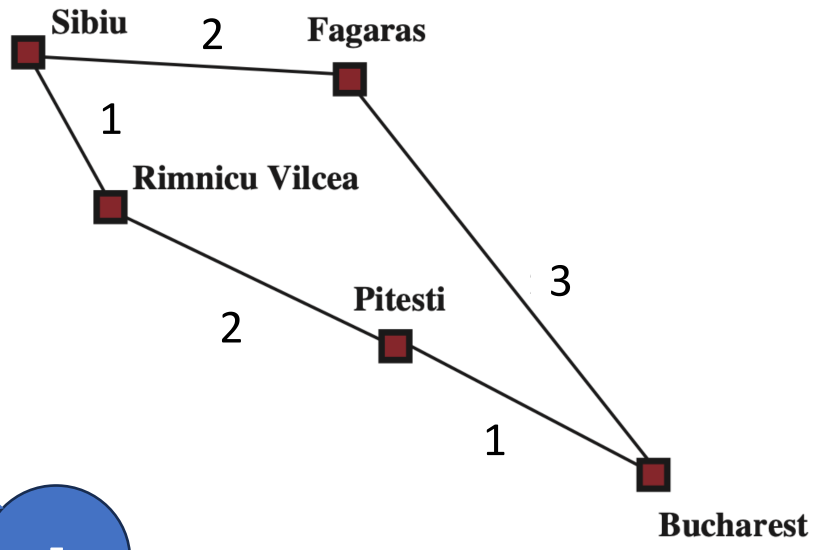
Priority Queue:

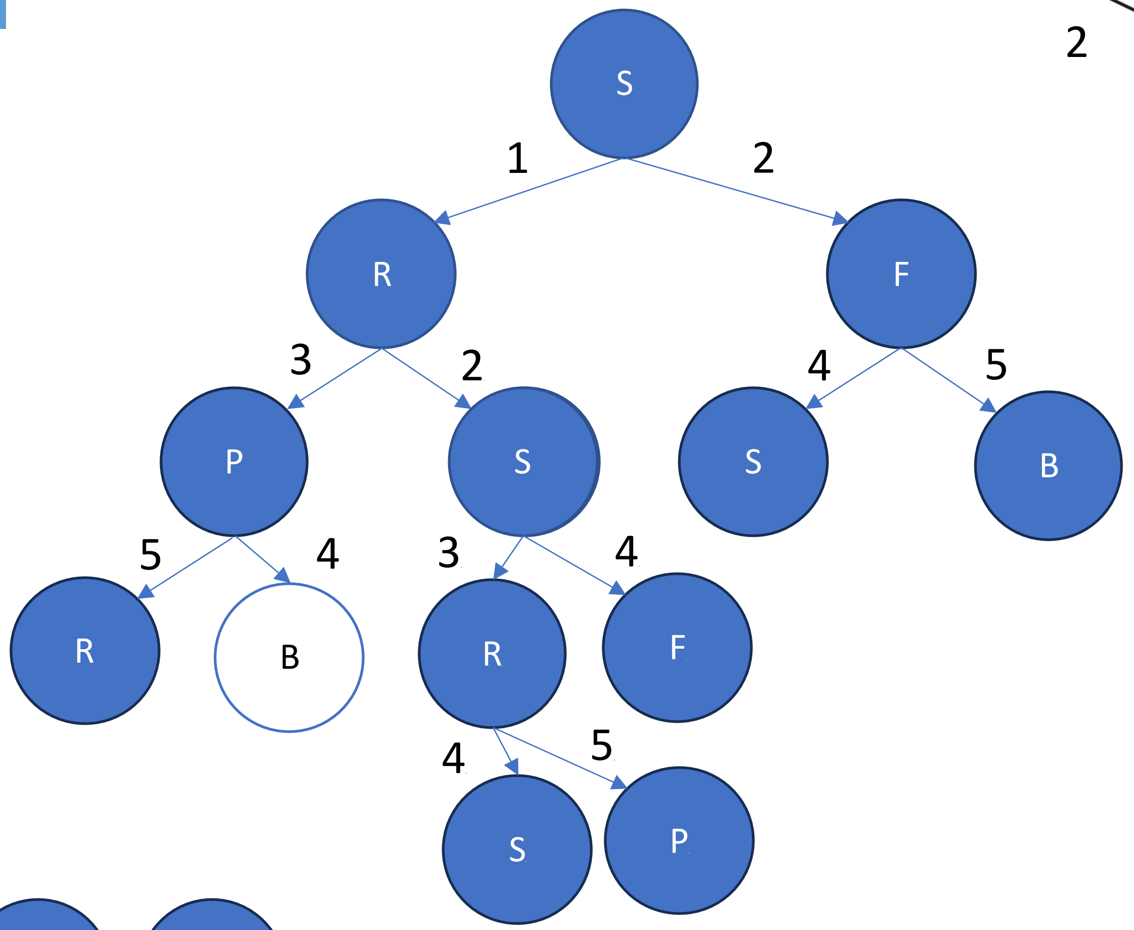


Uniform-cost Search (UCS)

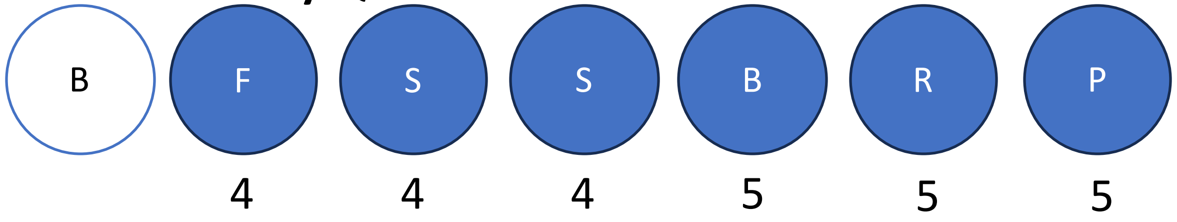
Cost from root to a state

```
create frontier : priority queue (path cost)


insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal: return solution



    for action in actions(node.state):
        next_state = transition(node.state, action)
        frontier.add(Node(next_state))
return failure
```





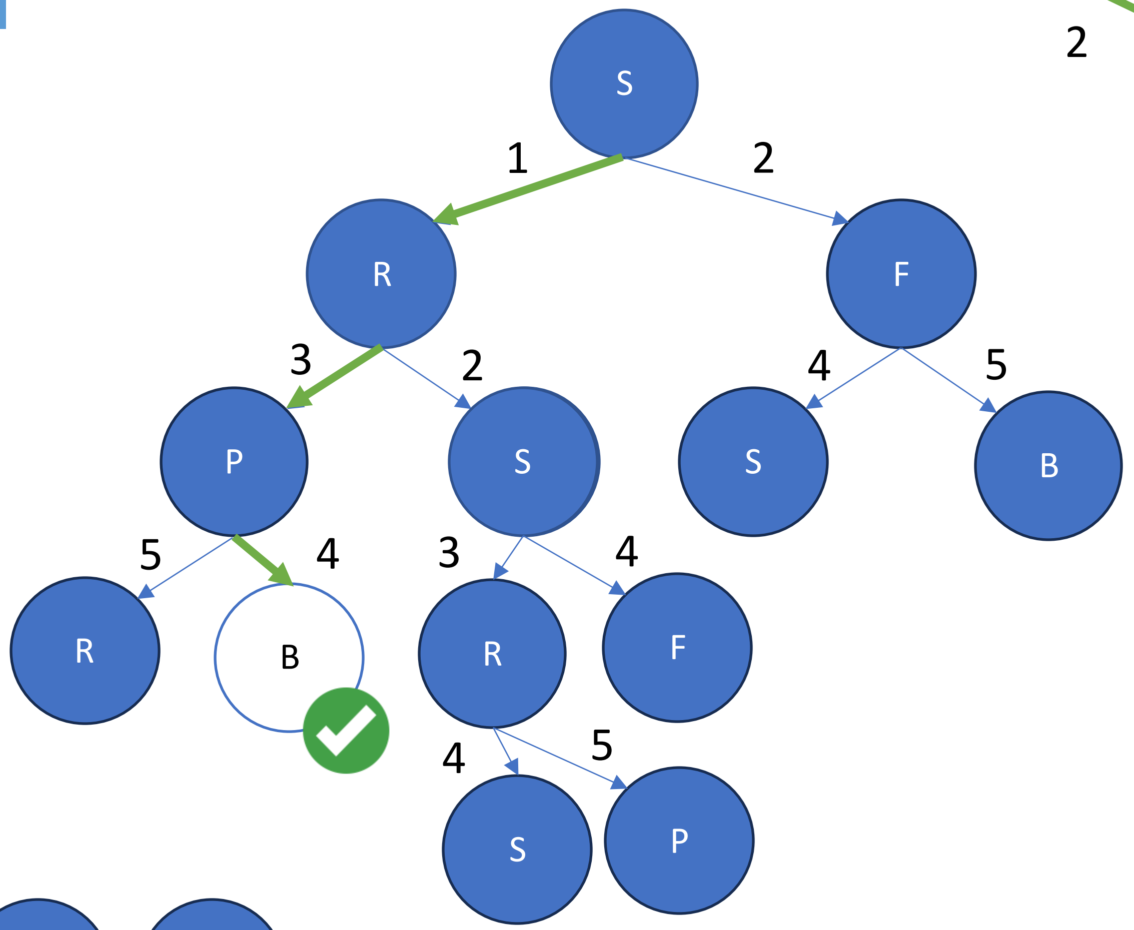
Priority Queue:

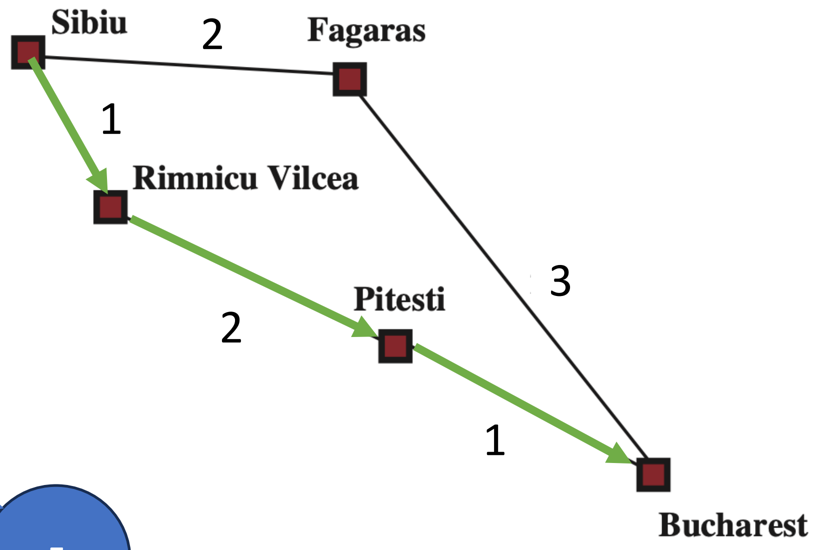


Uniform-cost Search (UCS)

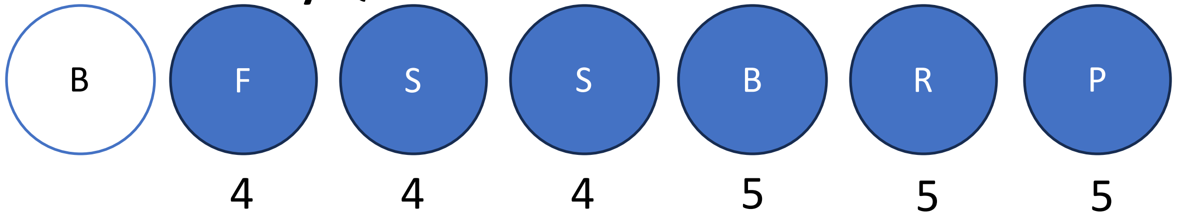
Cost from root to a state

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal: return solution


    for action in actions(node.state):
        next_state = transition(node.state, action)
        frontier.add(Node(next_state))
return failure
```





Priority Queue:



Uniform-cost Search (UCS)

Cost from root to a state

create **frontier : priority queue** (path cost)

insert **Node**(initial_state) to **frontier**

while **frontier** is not empty:

  node = **frontier**.pop()

  if node.state is goal: return solution

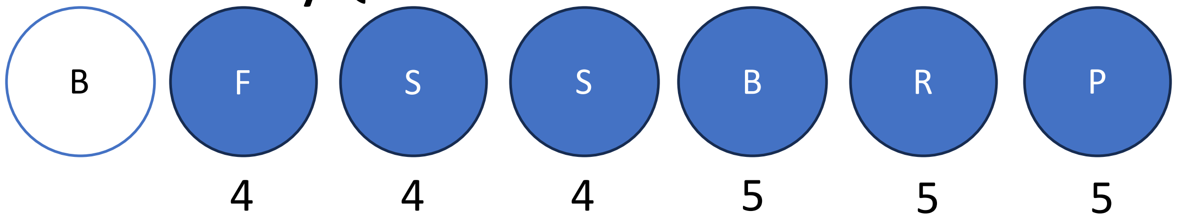
  for action in actions(node.state):

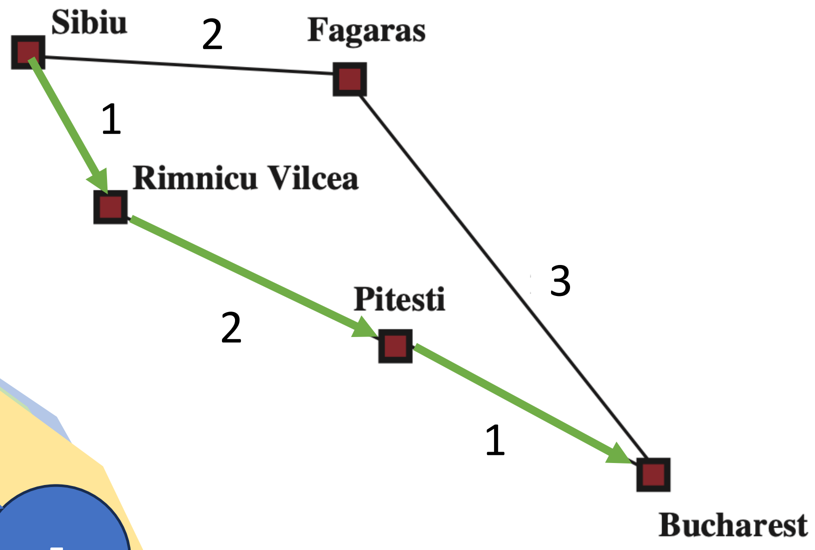
    next_state = transition(node.state, action)

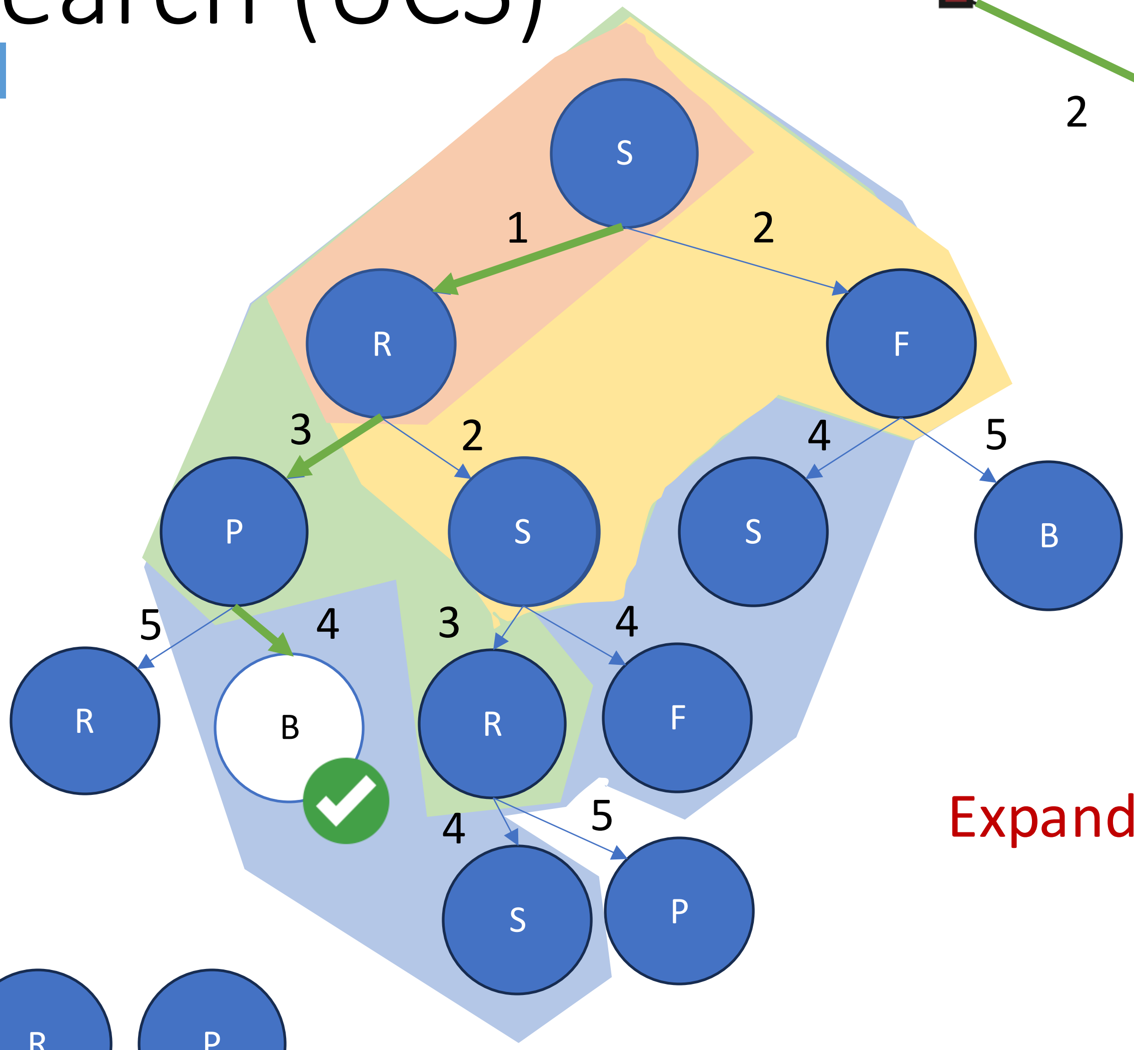
    frontier.add(**Node**(next_state))

return failure

Priority Queue:







Expands "tier" by "tier"

Uniform-cost Search (UCS) - Analysis

Cost from root to a state

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal: return solution


    for action in actions(node.state):
        next_state = transition(node.state, action)
        frontier.add(Node(next_state))
return failure
```

Time complexity: (# nodes generated)
Exponential w.r.t. the "tier" of the optimal solution

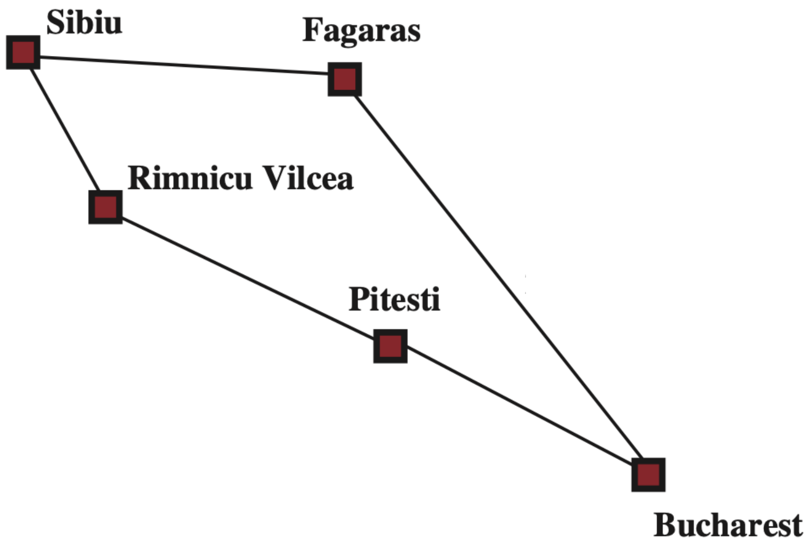
Space complexity: (size of frontier)
Exponential w.r.t. the "tier" of the optimal solution

Complete: Yes, if positive step cost everywhere and finite total cost

Optimal: Yes, if positive step cost everywhere

Depth-first Search (DFS)

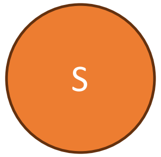
```
create frontier : stack


insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

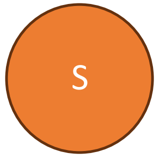


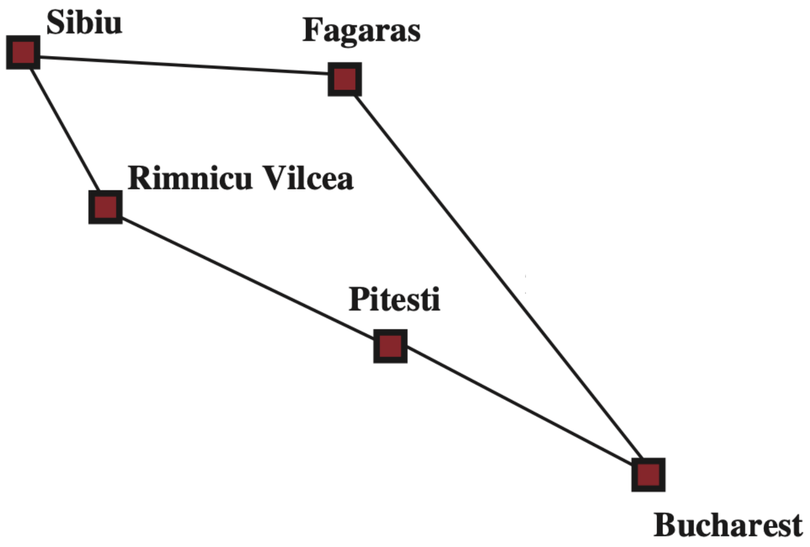
Depth-first Search (DFS)

```
create frontier : stack

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

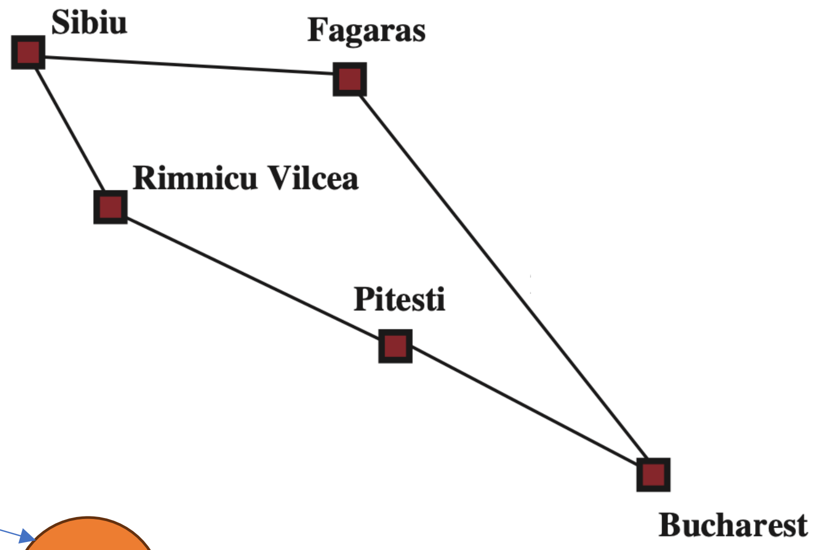
Stack:



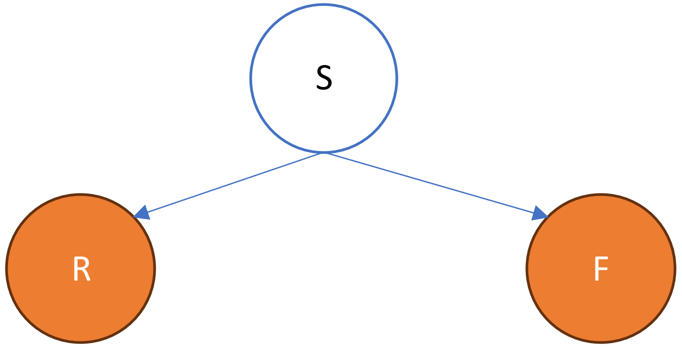




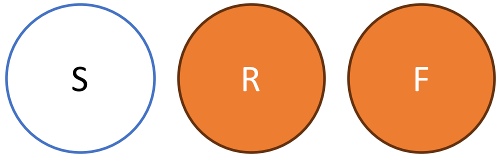
Depth-first Search (DFS)




```
create frontier : stack

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```



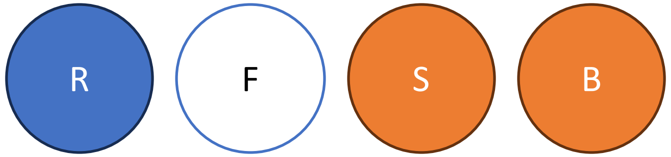
Stack:

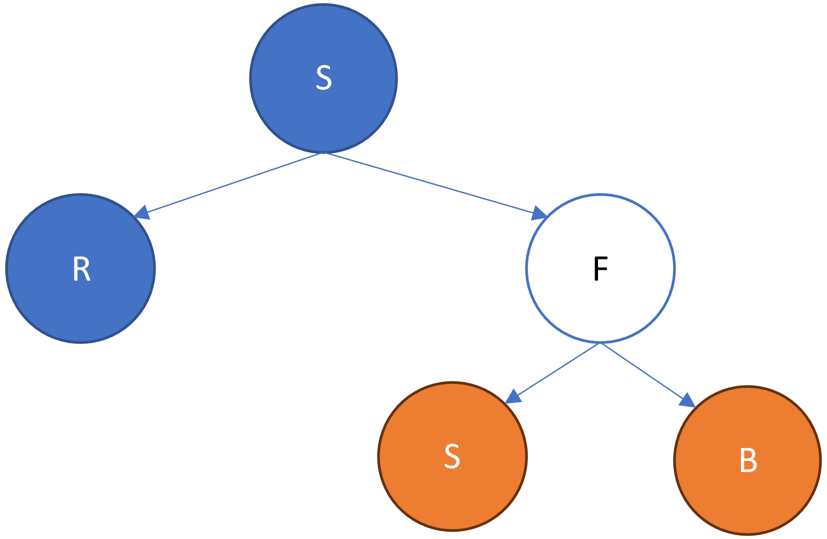


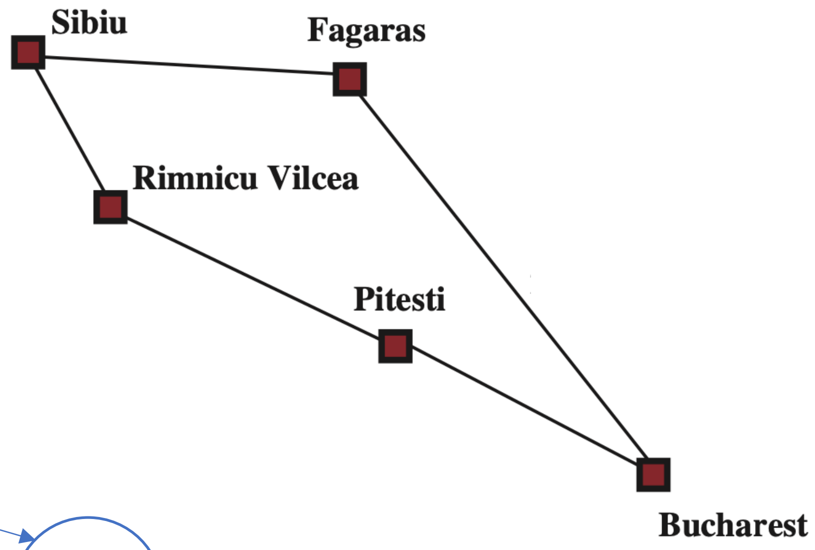
Depth-first Search (DFS)

```
create frontier : stack

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution


   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

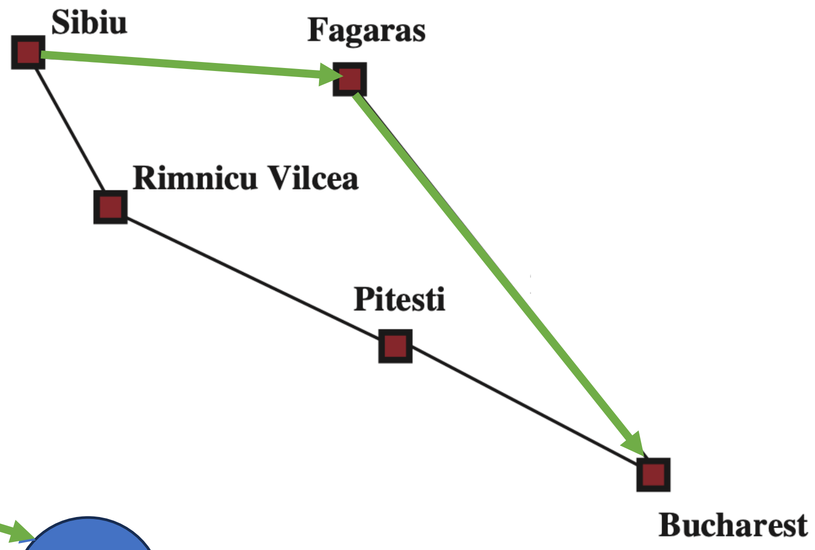
Stack:



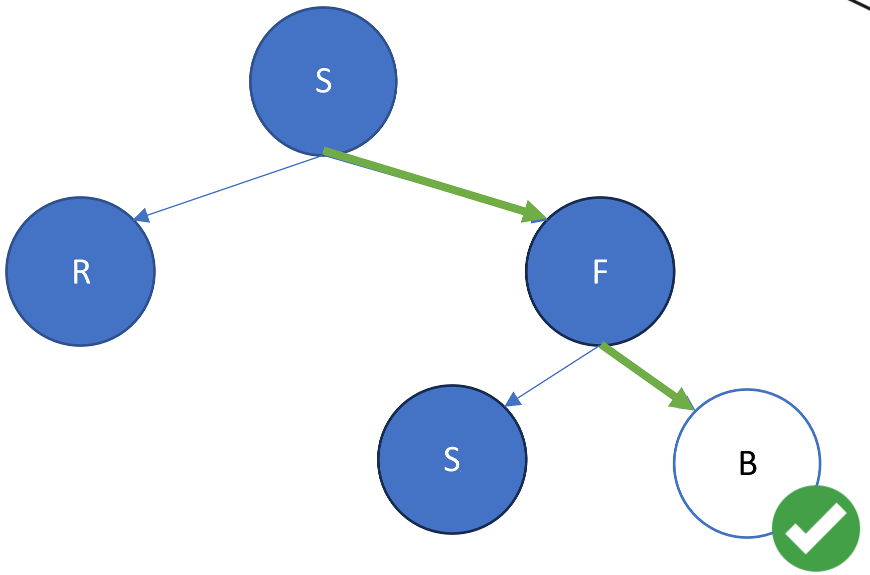




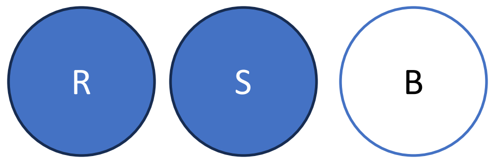
Depth-first Search (DFS)




```
create frontier : stack

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution


   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```



Stack:



Depth-first Search (DFS) - Analysis

```
create frontier : stack

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution


   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

Time complexity: (# nodes generated)
Exponential w.r.t. the maximum depth of the search tree

Space complexity: (size of frontier)
Polynomial w.r.t. the maximum depth of the search tree

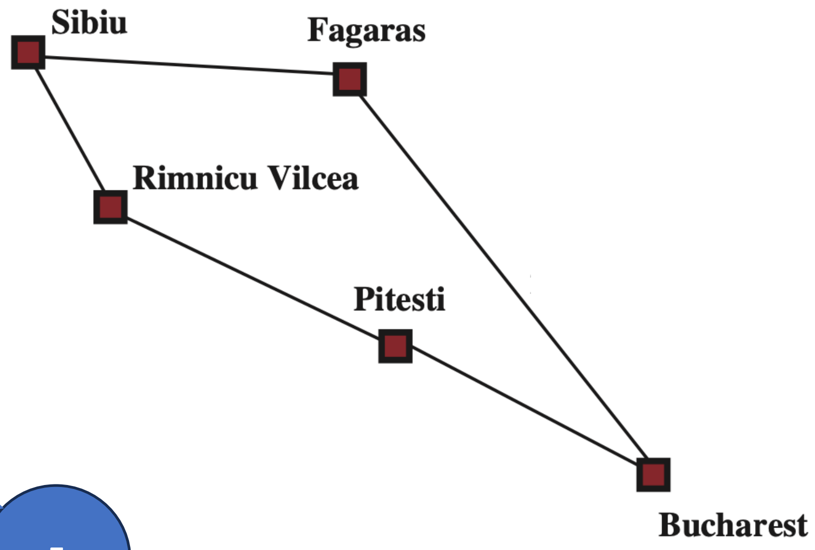
Complete: No, when the depth of the search tree is infinite (e.g., when the action is reversible, causing the state to go back and forth)

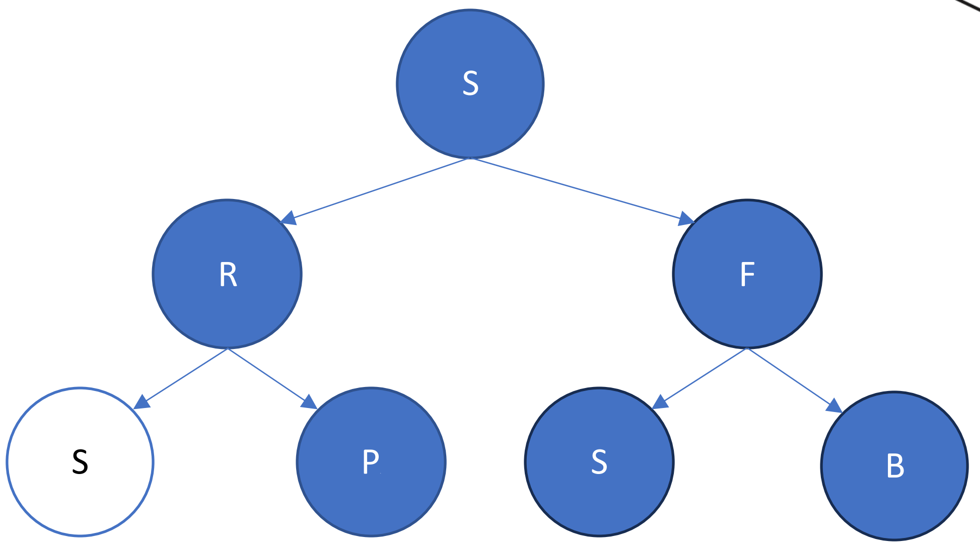
Optimal: No, the optimal solution may be in a shallower depth

Search with visited memory

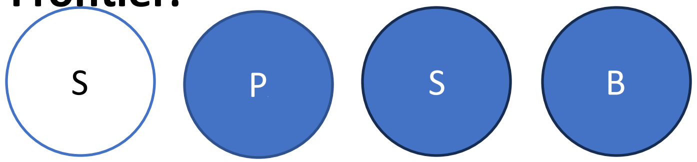
AIAMA: graph search

```
create frontier
create visited
insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution
   if node.state in visited: continue
   visited.add(state)
   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

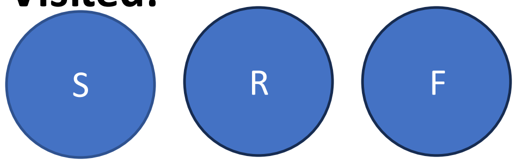




Frontier:



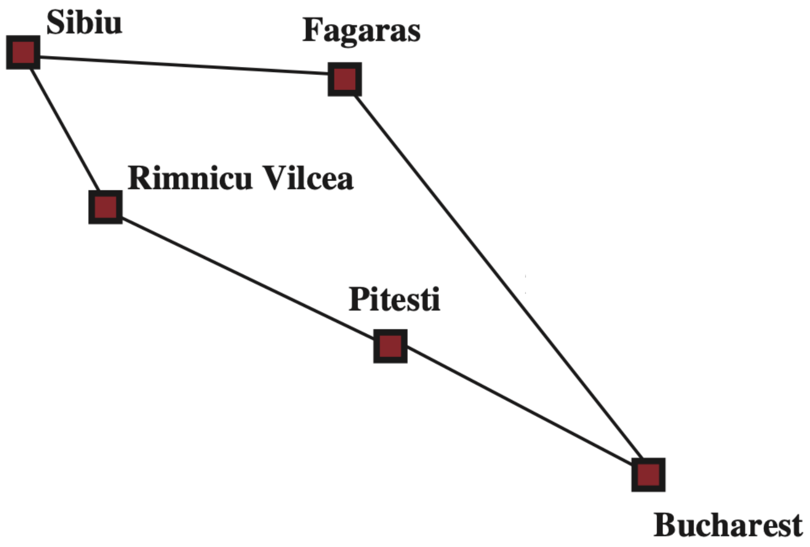
Visited:



Outline

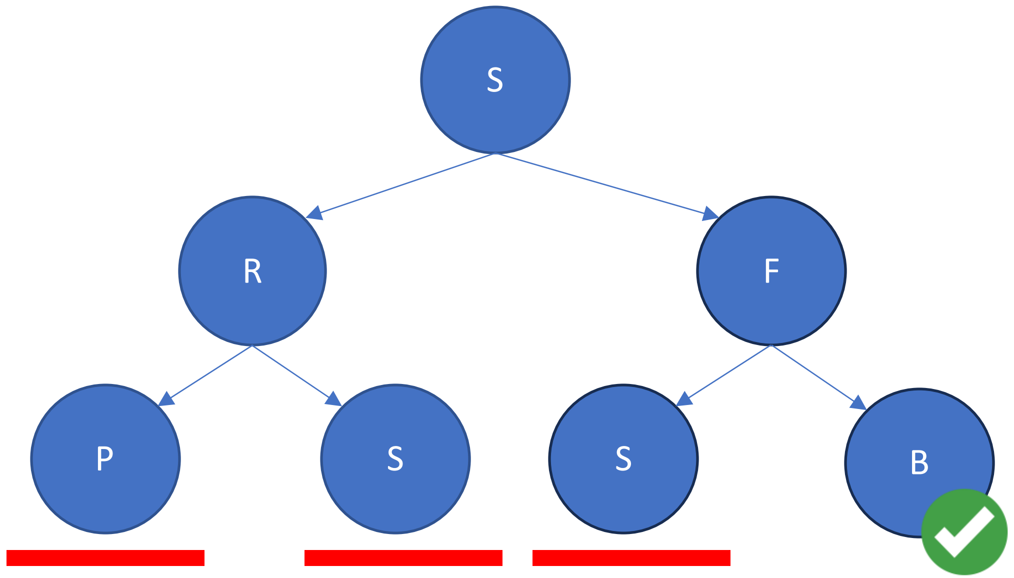
- Designing an Agent
  - Problem solving agents
  - Problem formulation
- Search
  - **Uninformed Search**
    - Breadth-first, depth-first, and uniform-cost search
    - **Depth-limited and iterative-deepening search**
  - Informed Search
    - A* search
    - Heuristics

Depth-limited Search (DLS)

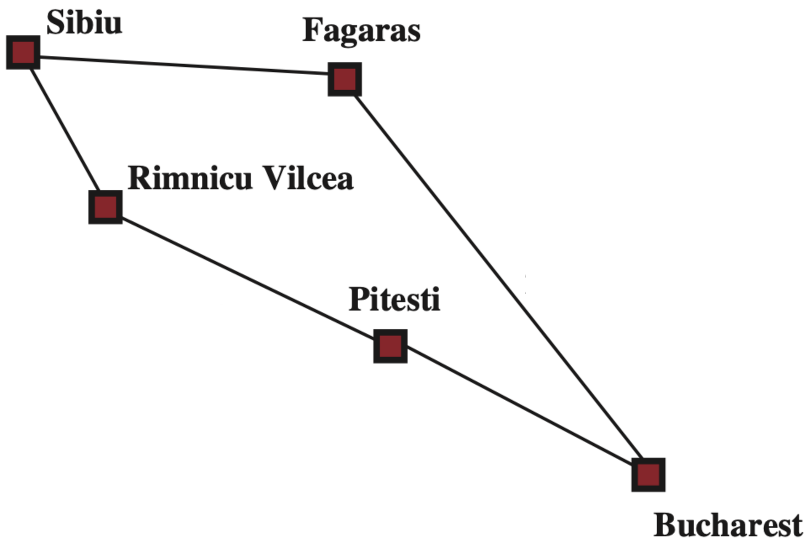


Limit the search to depth l

Backtrack when limit is hit



Depth-limited Search (DLS)



Limit the search to depth l

Backtrack when limit is hit

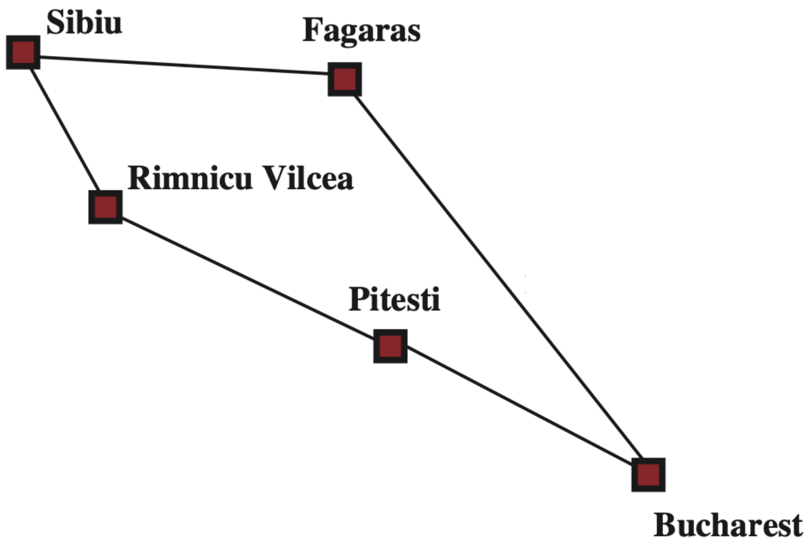
Time complexity: (# nodes generated)
Exponential w.r.t. the depth limit

Space complexity: (size of frontier)
Polynomial w.r.t. the depth limit if used with DFS

Complete: No

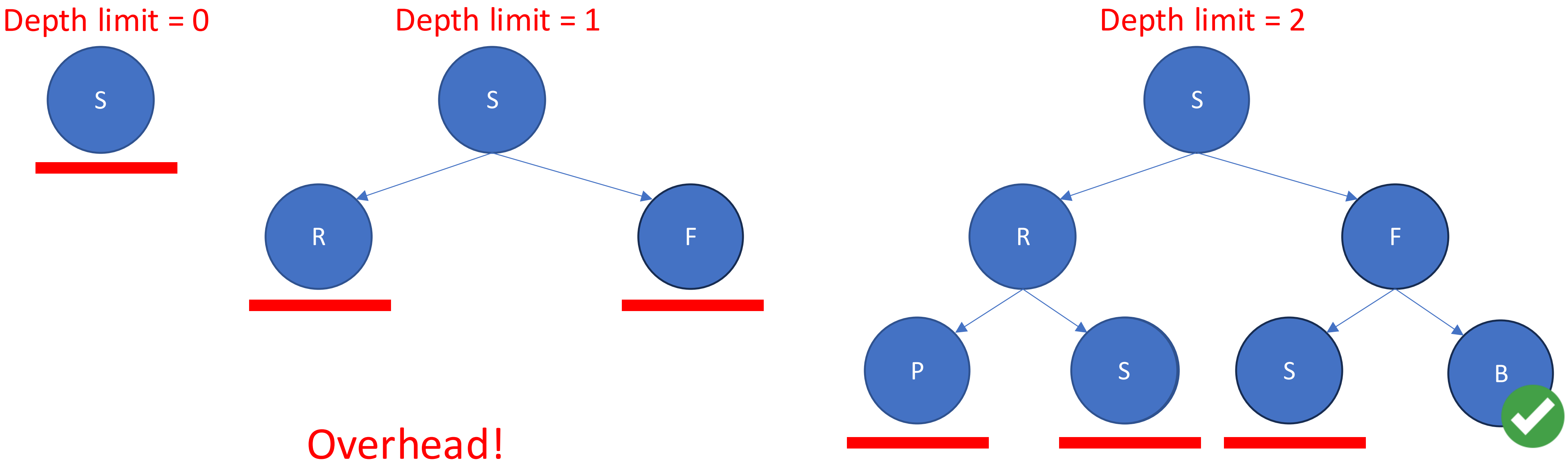
Optimal: No, if used with DFS

Iterative Deepening Search (IDS)

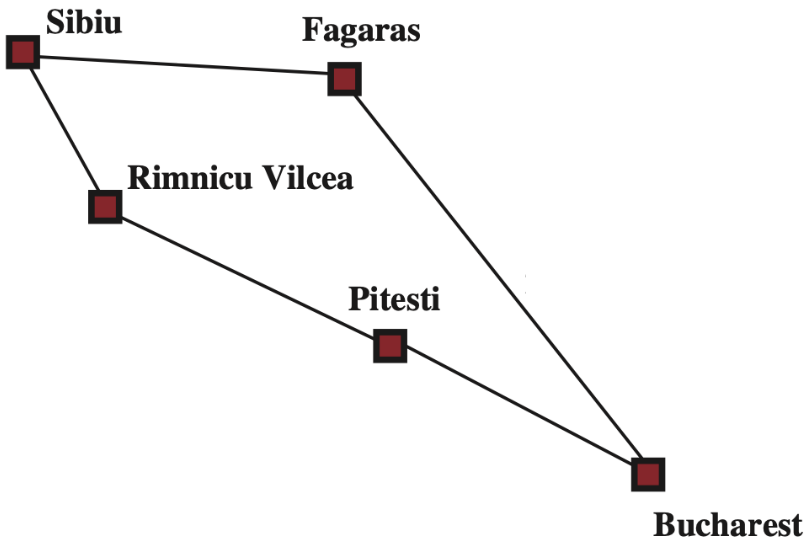


Search with depth limit $0, \ldots, \infty$

Return solution if found



Iterative Deepening Search (IDS)



Search with depth limit $0, \ldots, \infty$

Return solution if found

Time complexity: (# nodes generated)
Exponential w.r.t. the depth of optimal solution (with overhead)

Space complexity: (size of frontier)
Polynomial if used with DFS

Complete: Yes

Optimal: Yes, if step cost is the same everywhere

Summary: Uninformed Search Algorithms

- Search algorithms
  - Breadth-first search – **queue**, explore layer by layer
  - Uniform-cost search – **priority queue** (path cost)
  - Depth-first search – **stack**, go deep first then backtrack
- Variants:
  - Depth limited search – **limit max depth** of the search
  - Iterative deepening search – try DLS with depth limit $0, \ldots, \infty$

Summary: Uninformed Search Algorithms

Worst-case

Name	Time Complexity*	Space Complexity*	Complete?	Optimal?
Breadth-first Search	Exponential	Exponential	Yes	Yes
Uniform-cost Search	Exponential	Exponential	Yes	Yes
Depth-first Search	Exponential	Polynomial	No	No
Depth-limited Search	Exponential	Polynomial**	No**	No**
Iterative Deepening Search	Exponential	Polynomial**	Yes	Yes

*) In terms of some notion of depth/tier
**) If used with DFS

Further Reading (Optional)

- Full analysis of BFS, UCS, DFS, DLS, and IDS (AIAMA 3.4)
- Backward Search (AIAMA 3.4.6)
- Bidirectional Search (AIAMA 3.4.6)

Activity:

Given a maze where the agent is in the middle and there are two goals at the ends, which search algorithm(s) with visited memory is/are **optimal** and the most **space** and **time** efficient (in worst-case)?

Note: we care about overhead in space and time complexities

Name	Time Complexity*	Space Complexity*	Complete?	Optimal?
Breadth-first Search	Exponential	Exponential	Yes	Yes
Uniform-cost Search	Exponential	Exponential	Yes	Yes
Depth-first Search	Exponential	Polynomial	No	No
Depth-limited Search	Exponential	Polynomial**	No**	No**
Iterative Deepening Search	Exponential	Polynomial**	Yes	Yes

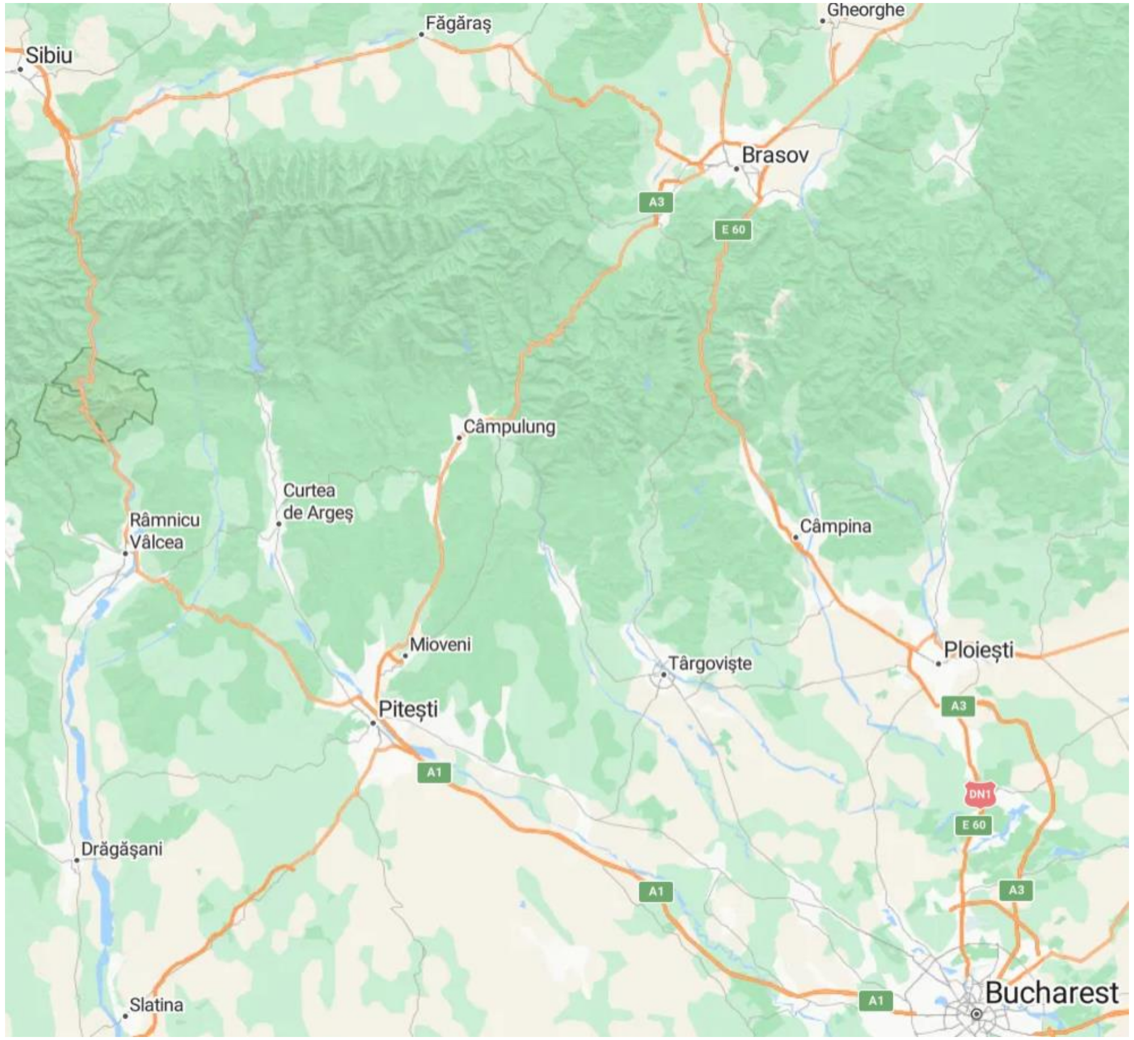
Note: we care about overhead in space and time complexities

Outline

- Designing an Agent
  - Problem solving agents
  - Problem formulation
- Search
  - Uninformed Search
    - Breadth-first, depth-first, and uniform-cost search
    - Depth-limited and iterative-deepening search
  - **Informed Search**
    - A* search
    - Heuristics

Uninformed Search Algorithms

No information that could guide the search



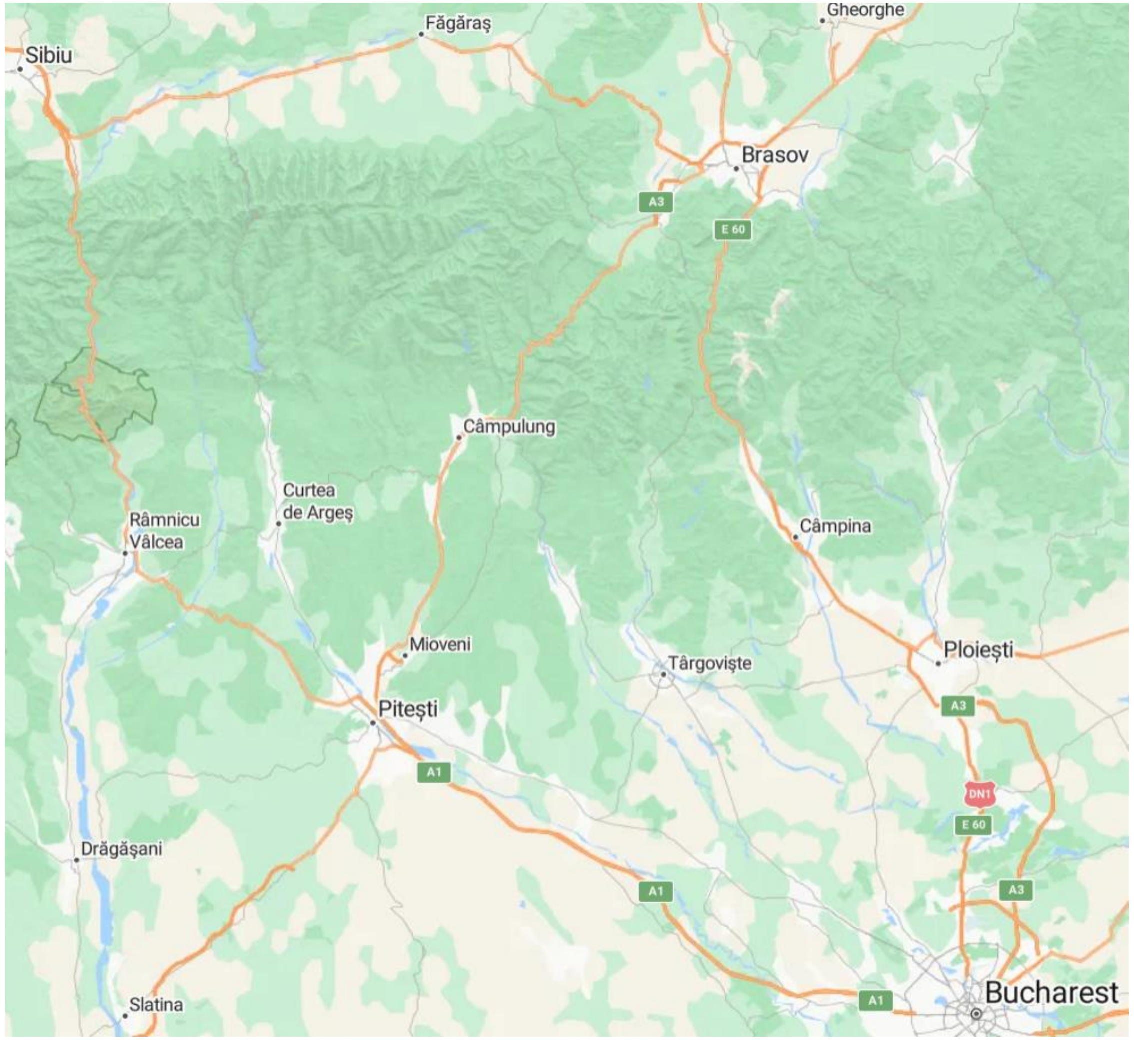
Blind search

No clue how good a state is,

e.g., how close it is to the goal

Informed Search Algorithms

Use information to guide the search



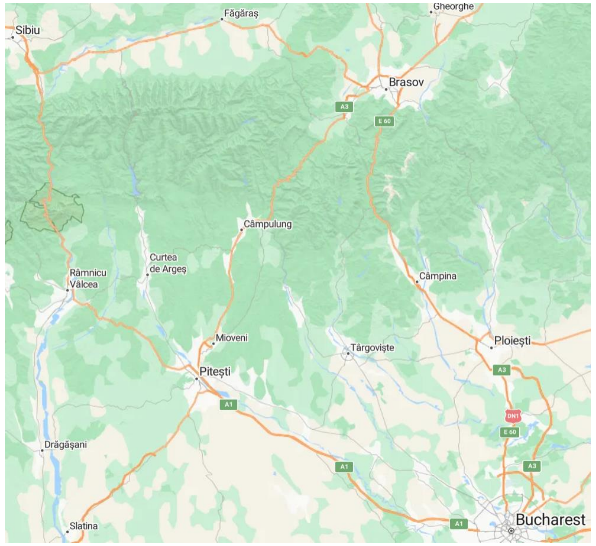
Search with extra info

<u>Has</u> a clue how good a state is,

e.g., how close it is to the goal, estimated using a heuristic function

Heuristic

- Recall that:
  - Path cost is the cost of a path from any state to any state
  - Optimal path cost is the cost of the lowest-cost path from any state to any state
- A heuristic is an estimate of the optimal path cost from any state to the goal state
- Examples (in path finding):
  - Straight-line distance h_{SLD}
  - Manhattan distance h_{MD}

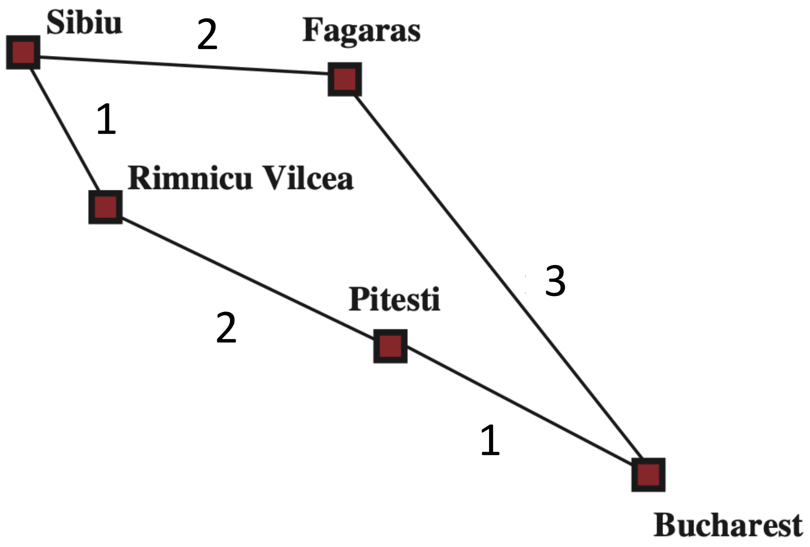


Best-first Search

Evaluation function

```
create frontier : priority queue ( f(n) )

insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal: return solution



    for action in actions(node.state):
        next_state = transition(node.state, action)
        frontier.add(Node(next_state))
return failure
```

$f(n)$ estimates how good a state is



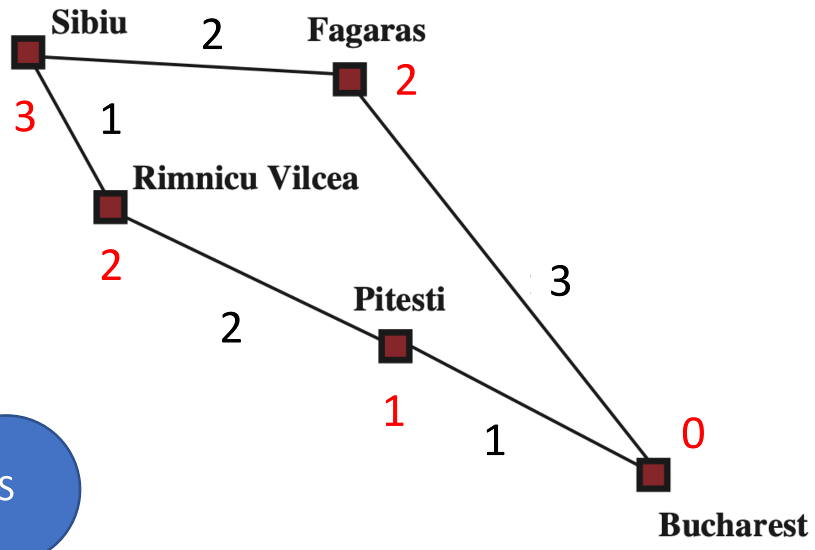
A* Search

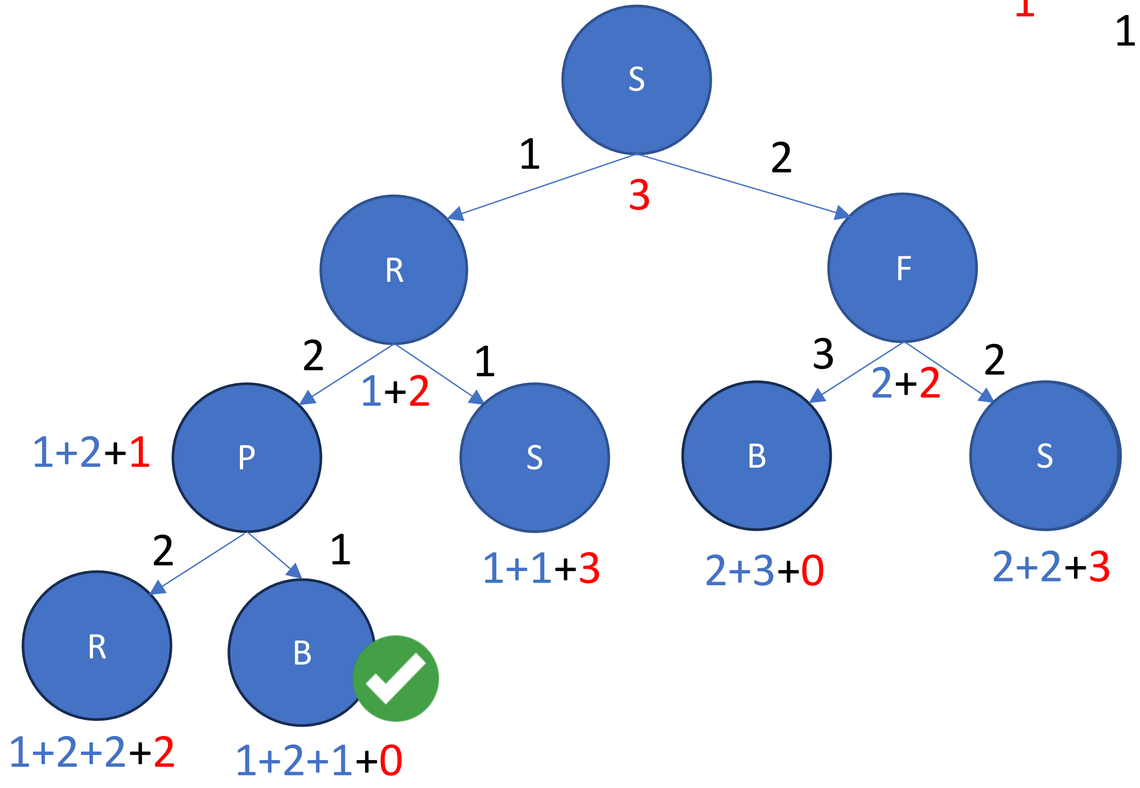
Evaluation function

```
create frontier : priority queue ( f(n) )

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution



   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

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

Cost to reach n — Heuristic: estimated cost from n to goal





A* Search - Analysis

Evaluation function

```
create frontier : priority queue ( f(n) )

insert Node(initial_state) to frontier
while frontier is not empty:
   node = frontier.pop()
   if node.state is goal: return solution

   for action in actions(node.state):
      next_state = transition(node.state, action)
      frontier.add(Node(next_state))
return failure
```

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

Cost to reach n   Heuristic: estimated cost from n to goal

Time complexity: (# nodes generated)
Exponential, good heuristic can improve

Space complexity: (size of frontier)
Exponential

Complete:
Yes, if edge costs are positive and branching factor is finite
(Hart, Nilsson and Raphael, 1968)

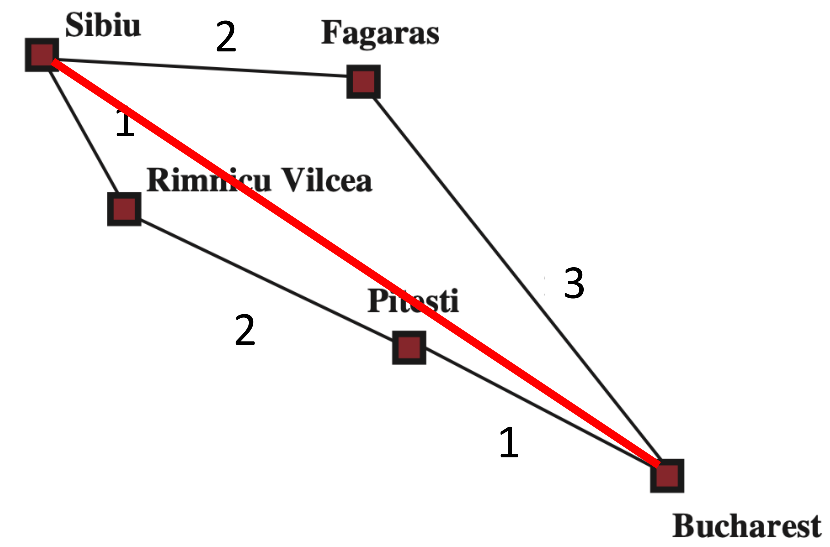
Optimal: Depends on the heuristic

Admissible Heuristics

A heuristic h(n) is **admissible** if for every node n, h(n) ≤ h*(n), where h*(n) is the **optimal path cost** to reach the goal state from n.

An admissible heuristic **never over-estimates** the cost to reach the goal, i.e., it is an **optimistic estimate.**

$h_{SLD}(Sibiu)$

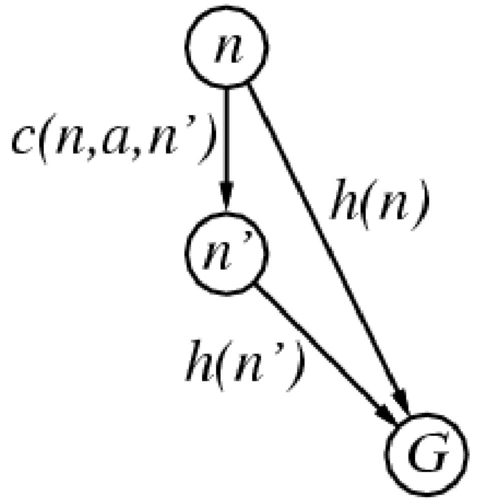


Example: $h_{SLD}(n)$ never overestimates the actual road distance

Theorem (Hart, Nilsson and Raphael, 1968):
if h(n) is admissible, A* **search (without visited memory)** is optimal

Consistent Heuristics

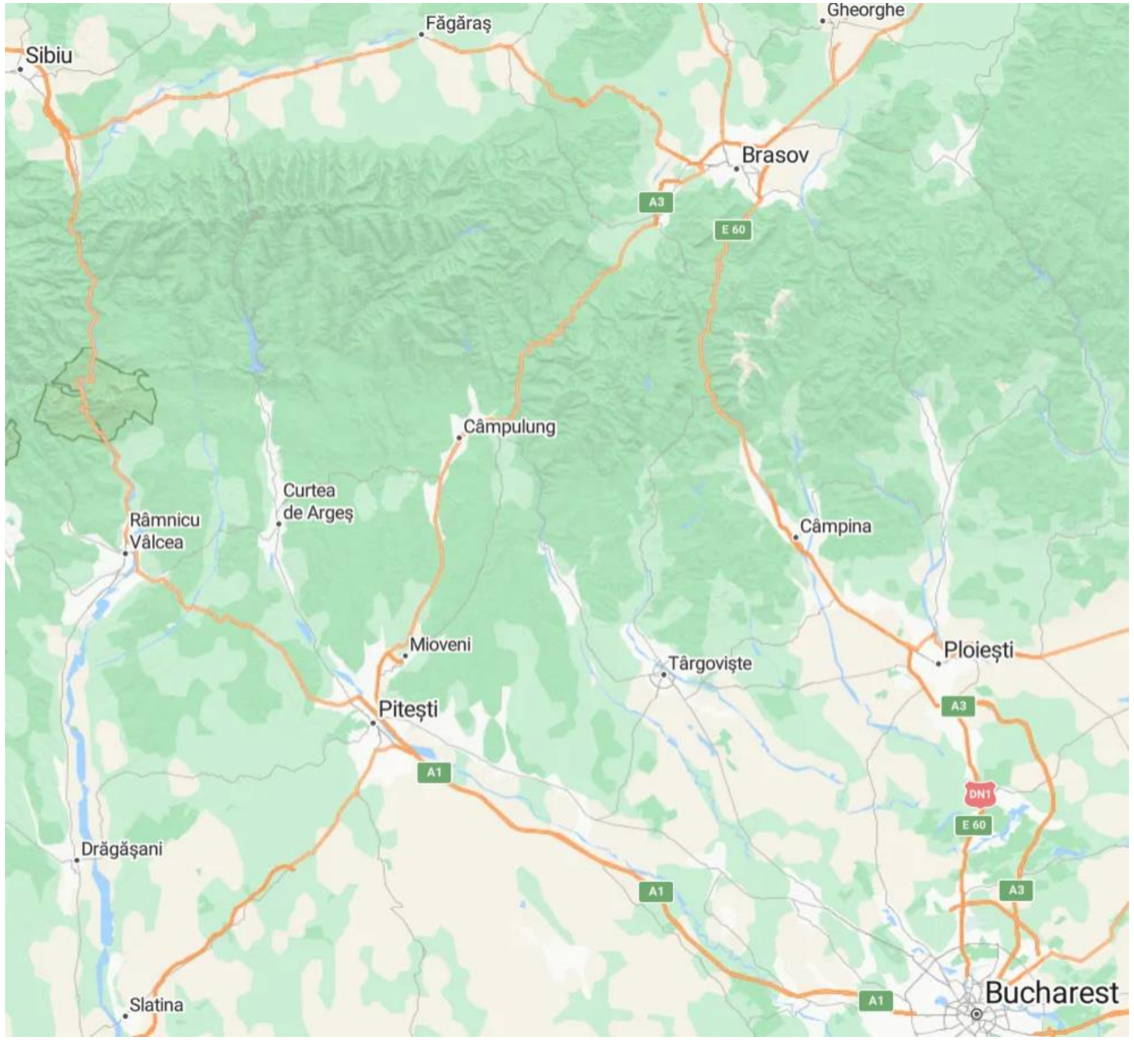
A heuristic h(n) is **consistent** if for every node n, every successor n' of n generated by any action a, h(n) ≤ c(n,a,n') + h(n’), and h(G) = 0.



Theorem (Hart, Nilsson and Raphael, 1968)**:**

If h(n) is consistent, A* **search with visited memory** is optimal

“Inventing” Admissible Heuristics



A problem with **fewer restrictions** on the actions is called a **relaxed problem**. The cost of an optimal solution to a relaxed problem is an **admissible heuristic** for the original problem.

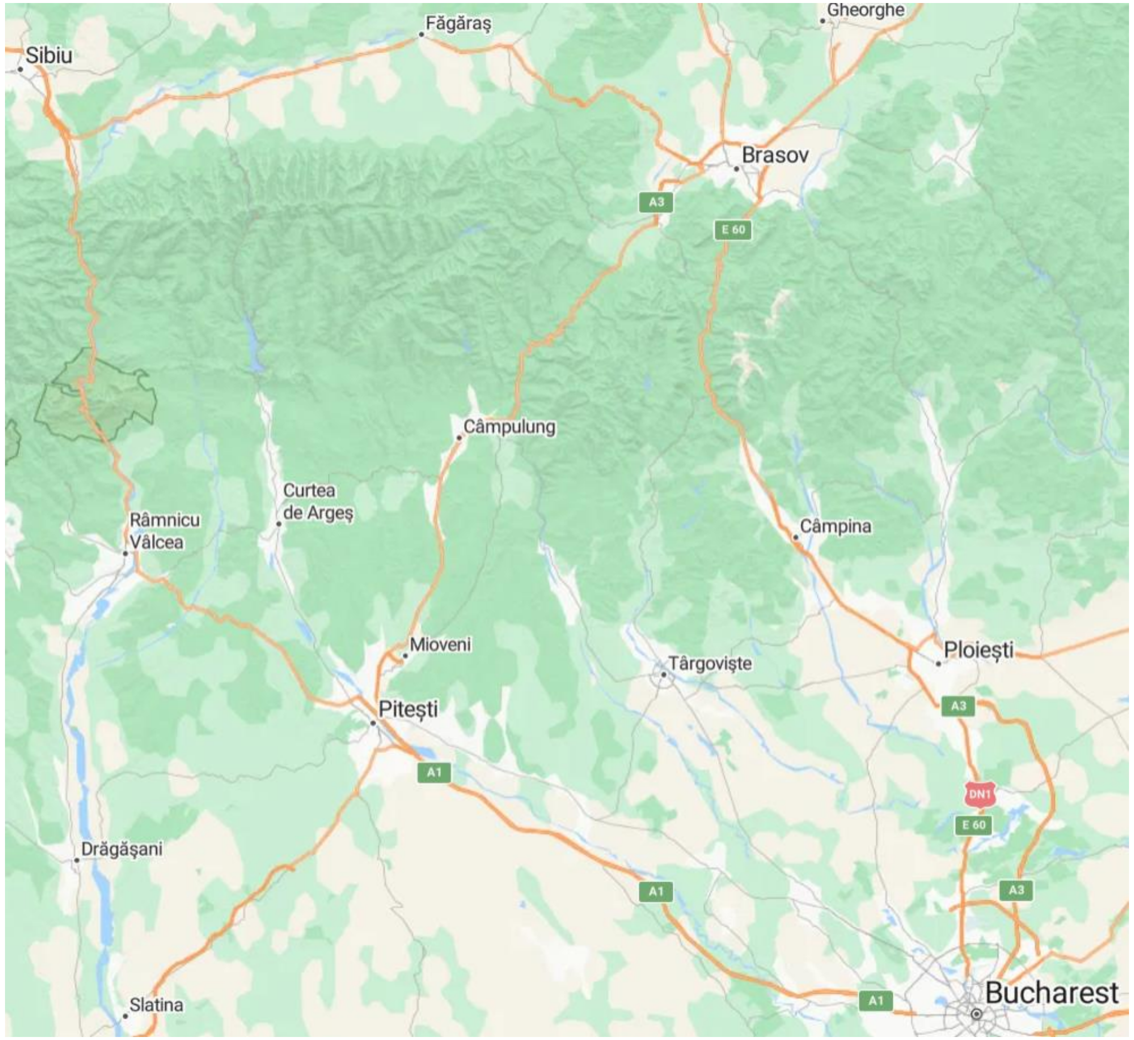
Original:

An agent can only move along the road

Relaxations:

- An agent can move off-road (fly)
  - h_1 straight line distance
- An agent can teleport
  - h_2 = 1

Dominance



If $h_1(n) \geq h_2(n)$ for all n, then h_1 dominates h_2.

- h_1 is better for search if admissible.

Heuristics

- h_1 straight-line distance
- h_2 = 1

h_1 dominates h_2

Summary: Informed Search Algorithms

- Informed search: <u>guide</u> search with extra information
- Best-first search
  - A* search
    - $f(n) = g(n) + h(n)$, cost so far + heuristic
- Heuristics
  - Admissible: $h(n) \leq h^*(n)$
  - Consistent: $h(n) \leq c(n,a,n') + h(n')$
  - Dominant: if $h1(n) \leq h2(n)$, h2 dominant
- Creating admissible heuristic: **optimal path cost** of the **relaxed problem**

Further Reading (Optional)

(Hart, Nilsson and Raphael, 1968)

- Proof: admissible heuristic → optimal search <u>without</u> visited memory
- Proof: consistent heuristic → optimal search with visited memory (AIAMA 3.5.2)
- Greedy best-first search (AIAMA 3.5.1)
- Variants of A* search (AIAMA 3.5.3)
  - Iterative Deepening A* (IDA*)
  - Simplified Memory-bounded A* (SMA*)

Summary

- Problem solving agents:
  - Goal formulation -> problem formulation -> search -> execute
- Uninformed search
  - Breadth-first search (BFS)
  - Uniform-cost Search (UCS)
  - Depth-first Search (DFS)
  - Depth-limited & Iterative deepening search (DLS & IDS)
- Informed search
  - A* Search
  - Heuristics: admissible, consistent, dominant, relaxed problem

Coming Up Next Week

- Local search
  - Hill climbing
  - …
- Adversarial search
  - Games
  - Minimax
  - Alpha-beta pruning
  - …

To Do

- **Lecture Training 2**
  - +250 EXP
  - +100 Early bird bonus
- **Problem Set 1**

Will be released later today!