CS2109S: Introduction to AI and Machine Learning

Lecture 2:
Solving Problems by Searching

25 August 2023

Admin: Assessments

- **Midterm (35%)**
  - Date/Time: Friday, 6 October (Week 7), 10:00 AM
  - Venue: Multipurpose Sports Hall (MPSH) (to be confirmed)
- **Final (35%)**
  - Date/Time: Saturday, 18 November, 8:00 PM - Sunday, 19 November, 11:59 PM

Admin: Next Week Lecture

Next week (Friday, 1st September) is Polling Day

- There will be **no in-class lecture**
- Lecture will be recorded and published online
- You are expected to watch the lecture before the weekend is over

Admin: Tutorial

Tutorial starts <u>next week</u>!

- Tutorial allocations is available in Coursemology
- Tutorial swaps is allowed (see latest announcement)

Do not appeal through EduRec!

Recap

Credit: Cooking with AMC

- What is AI?
- History lesson
- **PEAS**: **P**erformance Measure, **E**nvironment, **A**ctuators, **S**ensors
- Properties of the task environment
  - Fully observable, deterministic, episodic, static, discrete, single-agent
- Agents
  - Common: reflex, model-based, goal-based, utility-based, learning

Outline

- Problem-solving agents
- Search algorithms
- Uninformed search algorithms
    - Breadth-first Search (BFS)
    - Uniform-cost search
    - Depth-first Search (DFS)
- Variants of uninformed search algorithms
    - Depth-limited search
    - Iterative deepening search
    - Bidirectional search
- Dealing with repeated states
- Informed search algorithms
    - Greedy best-first search
    - A* search
    - Heuristics
- Variants of A*

Outline

- **Problem-solving agents**
- Search algorithms
- Uninformed search algorithms
  - Breadth-first Search (BFS)
  - Uniform-cost search
  - Depth-first Search (DFS)
- Variants of uninformed search algorithms
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search
- Dealing with repeated states
- Informed search algorithms
  - Greedy best-first search
  - A* search
  - Heuristics
- Variants of A*

Problem-Solving Agents

When the correct action to take is not immediately obvious, an agent may need to **plan-ahead**: to consider a *sequence* of actions that form a path to a goal state. Such an agent is called a **problem-solving agent**, and the computational process it undertakes is called **search**.

Credit: Passport & Plates

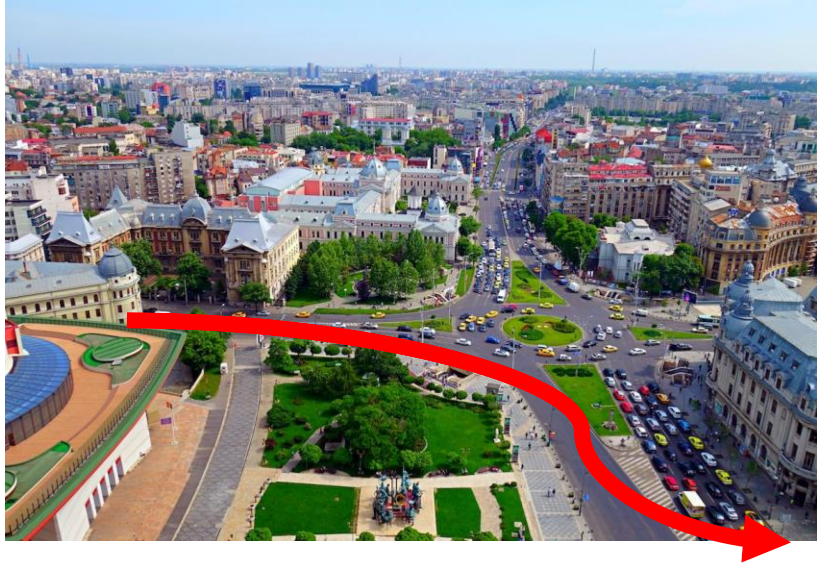
Want to go to Bucharest, how?

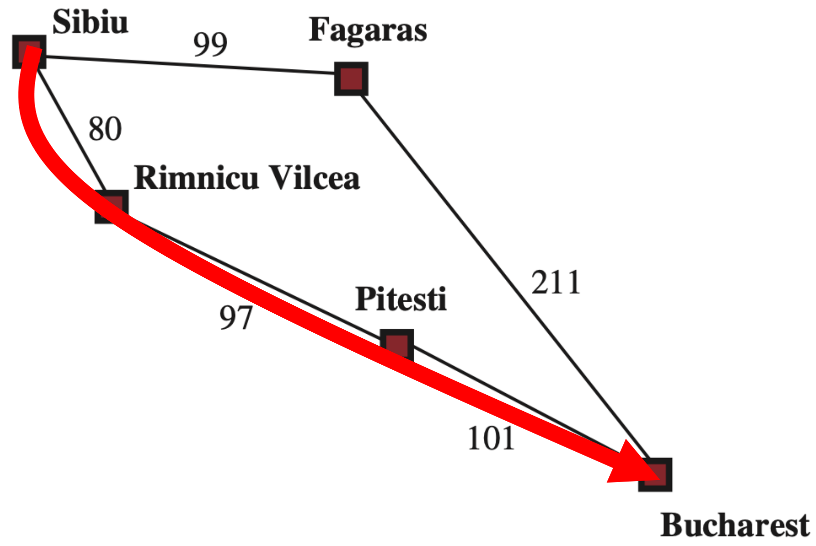
Problem-Solving Process

- **Goal formulation**
- **Problem formulation**
  - Create an *abstract* model of the relevant parts of the world
- **Search**
  - Simulates sequence of actions using the model, search until goal is reached
- **Execution**
  - Execute actions in the solution, one at a time

We assume that the task environment is **fully observable** and **deterministic**.

Thus, the **solution** is **a fixed sequence of actions**.

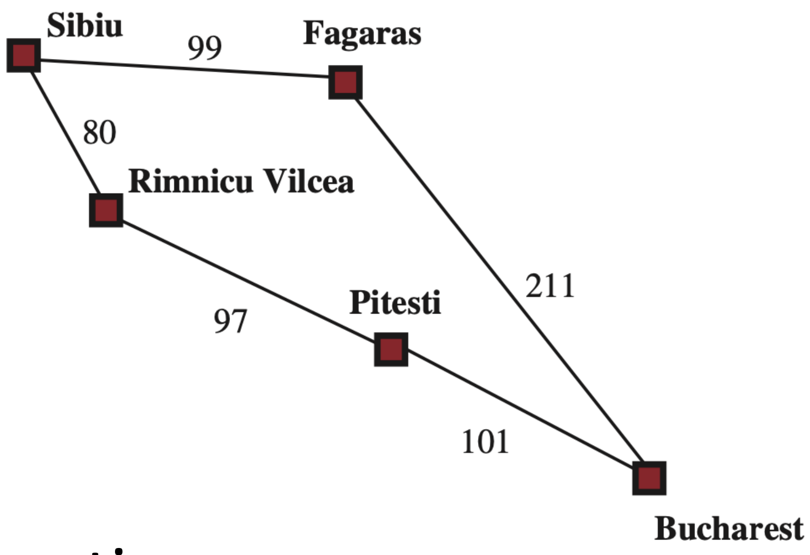




Problem Formulation

Formally, a search problem can be formulated as follows.

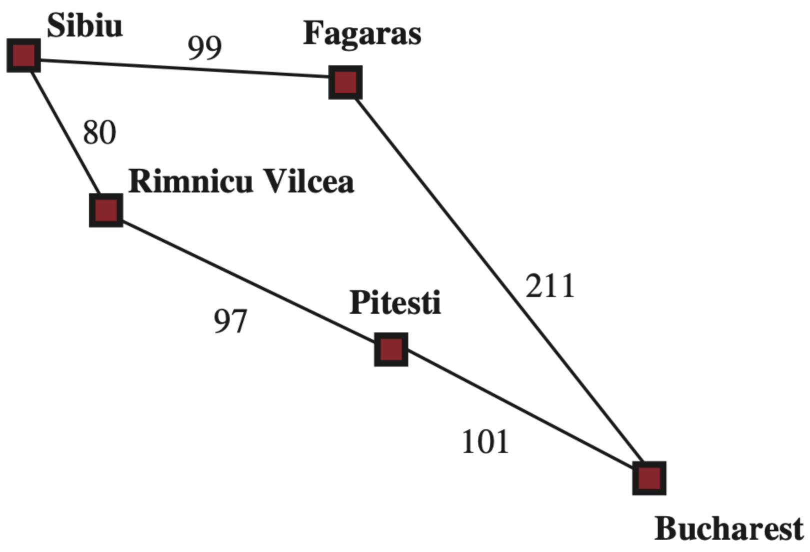
- **States (state space)**.
- **Initial state**: the initial state of the agent
- **Goal state(s)/test**
- **Actions**: things that the agent can do
- **Transition model**: what each action does
- **Action cost function**: the cost of performing an action



A sequence of action form a **path/trajectory**. **Solution** is a path to a goal.

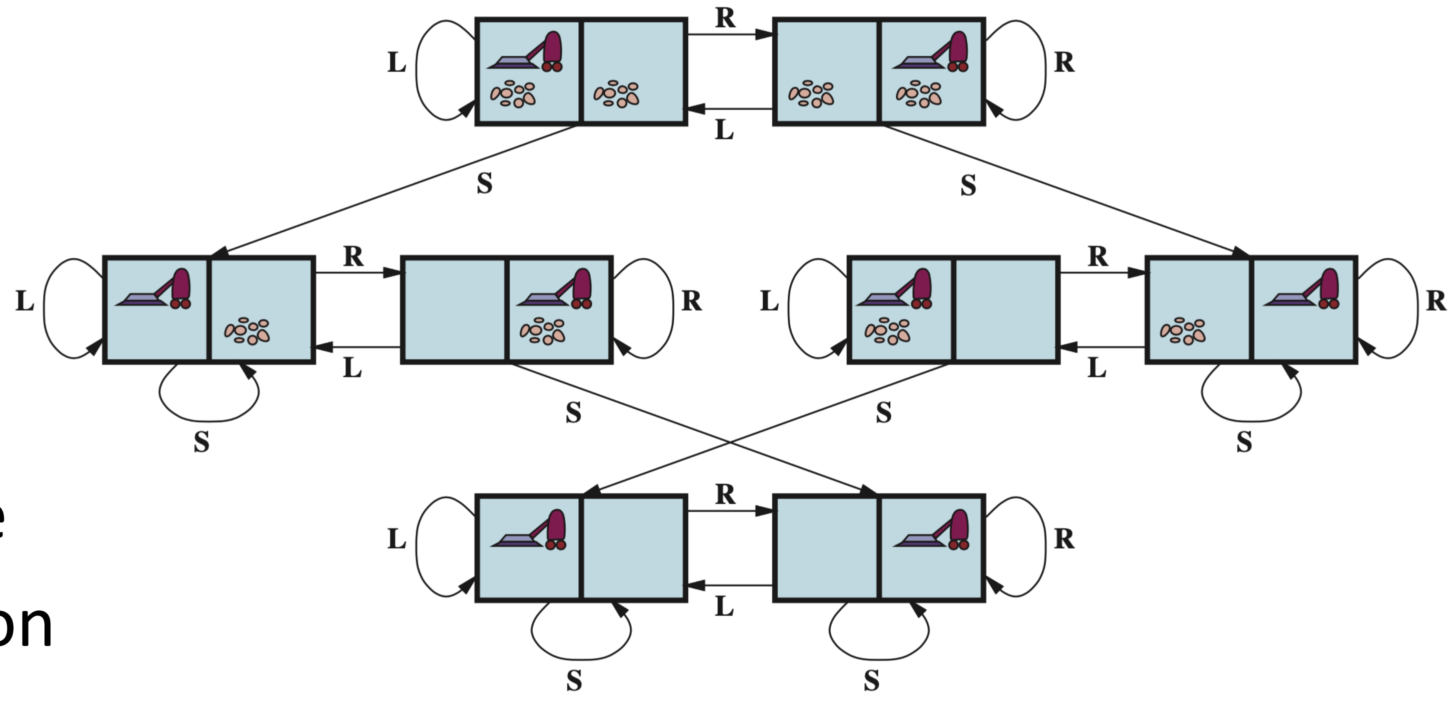
Problem Formulation: Romania

- **States:** {at Sibiu, at Fagaras, …, at Bucharest}
- **Initial state**: at Sibiu
- **Goal state(s)/test**: at Bucharest
- **Actions:** go to neighboring city x
- **Transition model**: move to target city
- **Action cost function**: distance



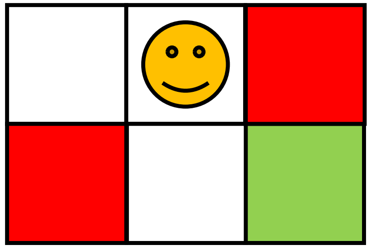
Problem Formulation: Vacuum World

- **States:** see image
- **Initial state**: any state
- **Goal state(s)/test**: all clean
- **Actions:** suck, left, right
- **Transition model**: see image
- **Action cost function**: 1/action



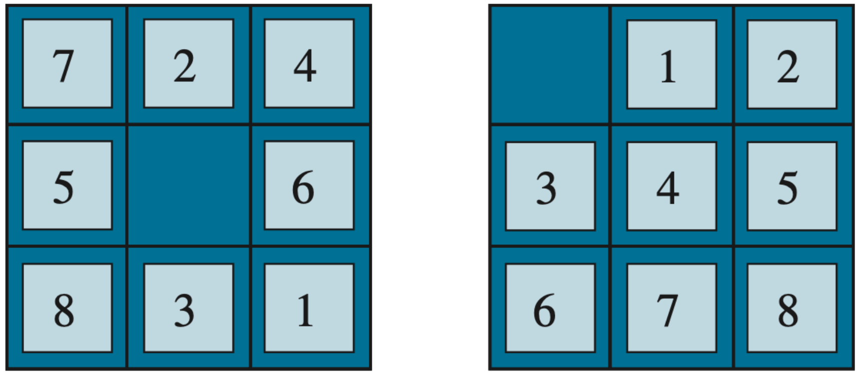
Problem Formulation: Grid World

- **States:** positions of agent
- **Initial state**: (0,1)
- **Goal state(s)/test**: (1,2)
- **Actions:** up, down, left, right
- **Transition model**: move if there is no wall, die on lava
- **Action cost function**: -1



Problem Formulation: 8-Puzzle

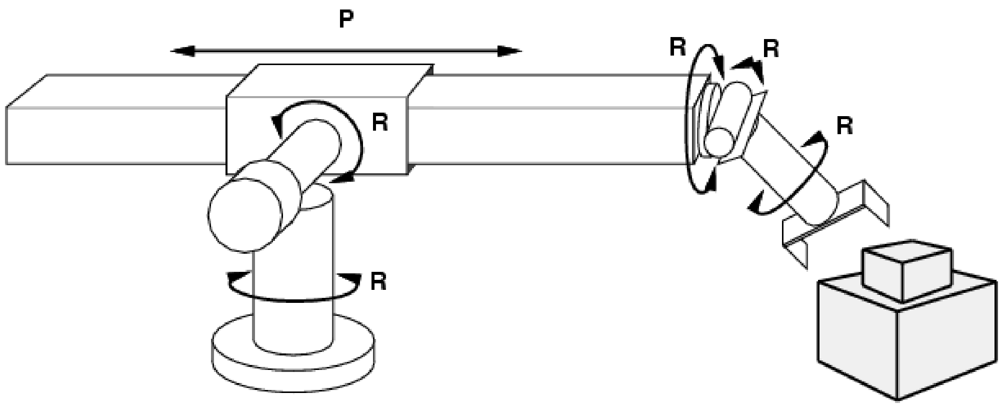
- **States:** location of tiles
- **Initial state**: see image
- **Goal state(s)/test**: see image
- **Actions:** move blank left, right, up, down
- **Transition model**: move blank
- **Action cost function**: 1/move



Start State
Goal State

Problem Formulation: Robot Assembly

- **States:** robot joints positions and angles
- **Initial state**: item not assembled
- **Goal state(s)/test**: item assembled
- **Actions:** see image
- **Transition model**: move/rotate joint
- **Action cost function**: rotation/displacement



Outline

- Problem-solving agents
- **Search algorithms**
- Uninformed search algorithms
  - Breadth-first Search (BFS)
  - Uniform-cost search
  - Depth-first Search (DFS)
- Variants of uninformed search algorithms
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search
- Dealing with repeated states
- Informed search algorithms
  - Greedy best-first search
  - A* search
  - Heuristics
- Variants of A*

Search Algorithms

A search algorithm takes in a search problem as input and returns a solution / failure. It is defined by the **order of node expansion**.

Evaluation criteria:

- **Time complexity:** number of nodes expanded
- **Space complexity:** maximum number of nodes in memory
- **Completeness:** does it return a solution if it exists?
- **Optimality:** does it always find the least-cost solution?

Measure: branching factor (b), depth (d), maximum depth (m)

Tree Search

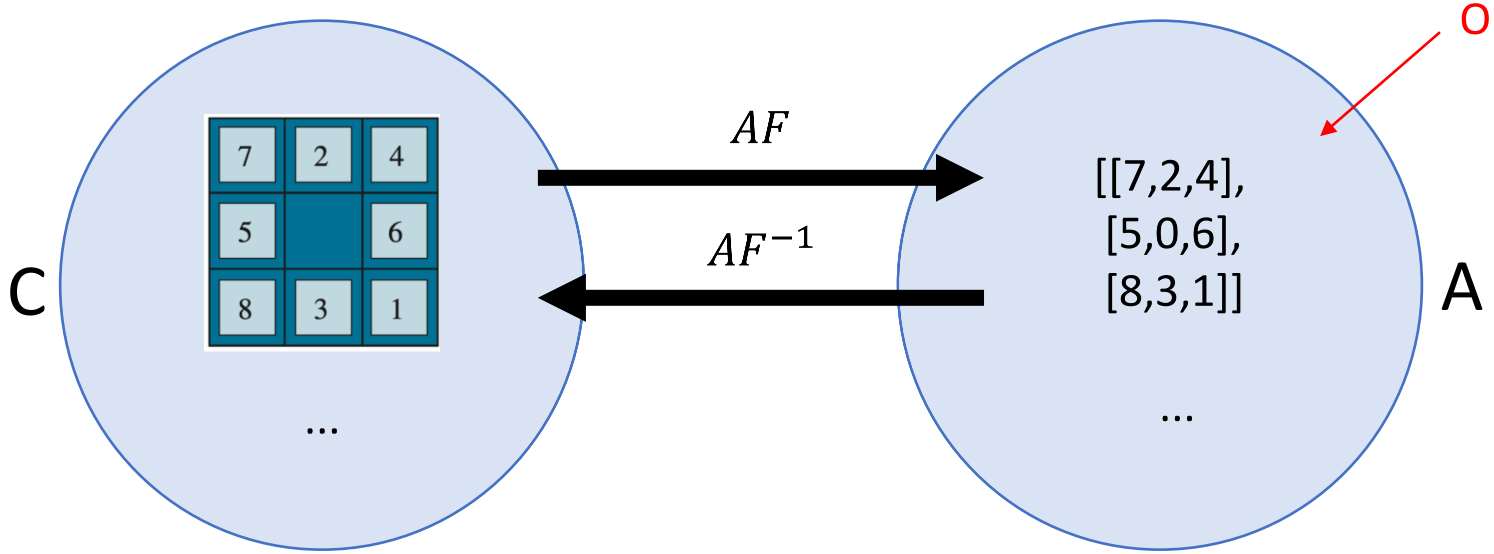
Define order of node expansion

```
create frontier


insert initial state
while frontier is not empty:
    state = frontier.pop()
    for action in actions(state):
        next state = transition(state, action)


        if next state is goal: return solution
        frontier.add(next state)
return failure
```

Representation Invariant

An **abstraction function** AF maps a real-world (concrete) state $c \in C$ to an **abstract representation** $a \in A$.

Our search operates on this



$I\left(\begin{matrix}[[7,4,2],\\ [5,0,6],\\ [8,3,1]]\end{matrix}\right) = false$

Invalid state!

Representation invariant I is a function such that $I(a) = true$ for all legitimate representations $a \in A$. If $I(a) = true$, then $AF^{-1}(a) \in C$.

We need to make sure that our transition function satisfies the representation invariant i.e., produce valid states

Outline

- Problem-solving agents
- Search algorithms
- **Uninformed search algorithms**
  - Breadth-first Search (BFS)
  - Uniform-cost search
  - Depth-first Search (DFS)
- Variants of uninformed search algorithms
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search
- Dealing with repeated states
- Informed search algorithms
  - Greedy best-first search
  - A* search
  - Heuristics
- Variants of A*

Uninformed Search Algorithms

An uninformed search algorithm is given no clue about how close a state is to the goal(s).

- Breadth-first search (BFS)
- Uniform-cost search
- Depth-first search (DFS)

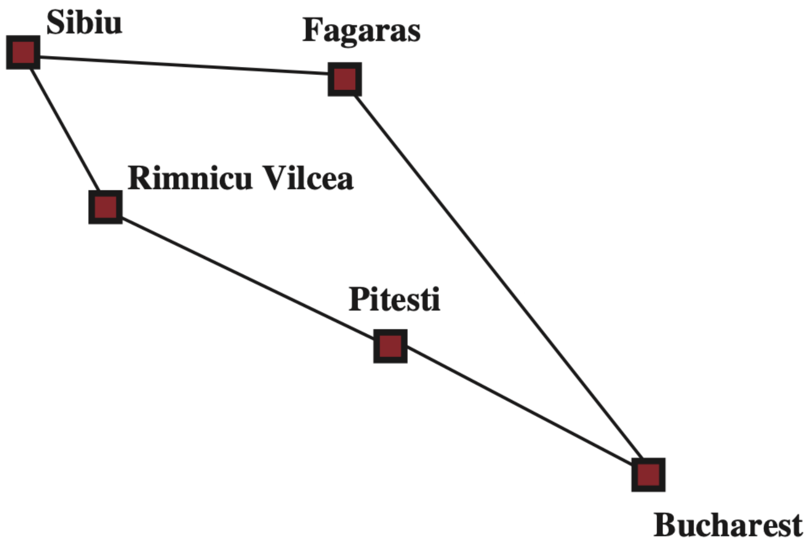
Condition	Algorithm	Time Complexity
No Negative Weight Cycles	Bellman-Ford Algorithm	$O(VE)$
On Unweighted Graph (or equal weights)	BFS	$O(V + E)$
No Negative Weights	Dijkstra's Algorithm	$O((V + E) log\, V)$
On Tree	BFS / DFS	$O(V)$
On DAG	Topological Sort	$O(V + E)$

CS2040S focus was on finding the shortest path (SSSP, APSP)

CS2109S: might <u>not care</u> about shortest path, or even the path

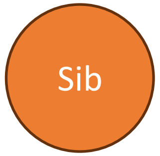
Breadth-first Search

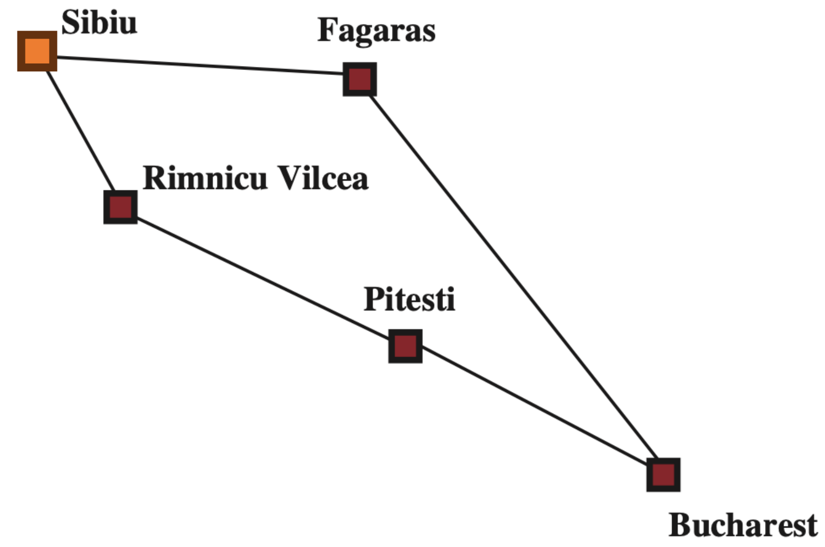
```
create frontier : queue


insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)


      if next state is goal: return solution
      frontier.add(next state)
return failure
```



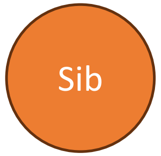
Breadth-first Search

```
create frontier : queue


insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)


      if next state is goal: return solution
      frontier.add(next state)
return failure
```



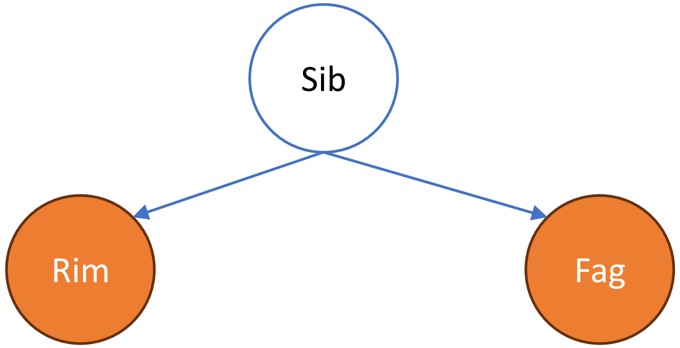


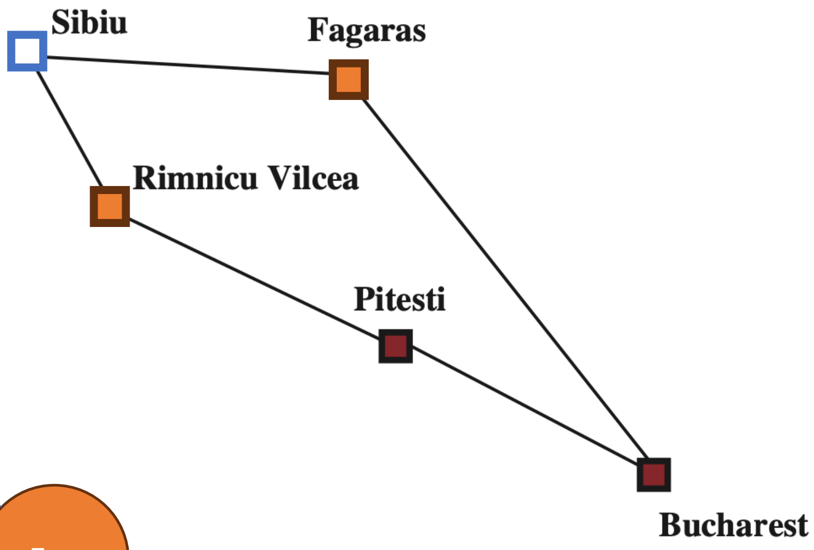
Queue:



Breadth-first Search

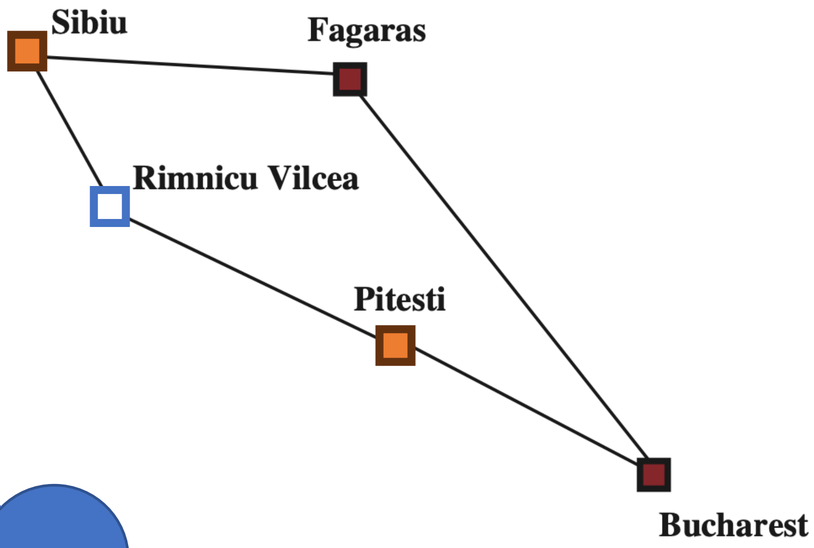
```
create frontier : queue

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```



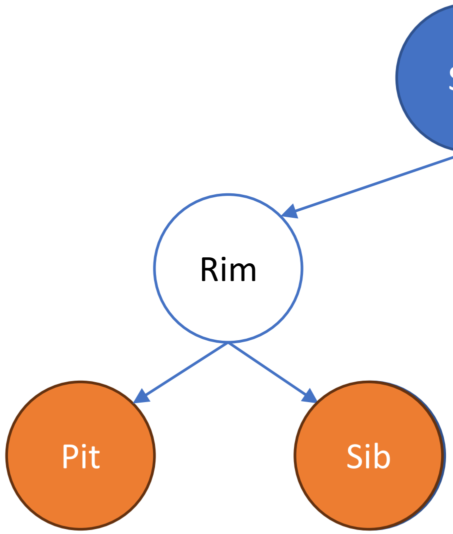


Queue:

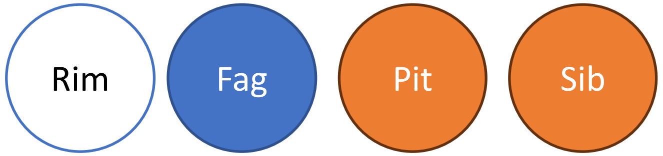
Breadth-first Search



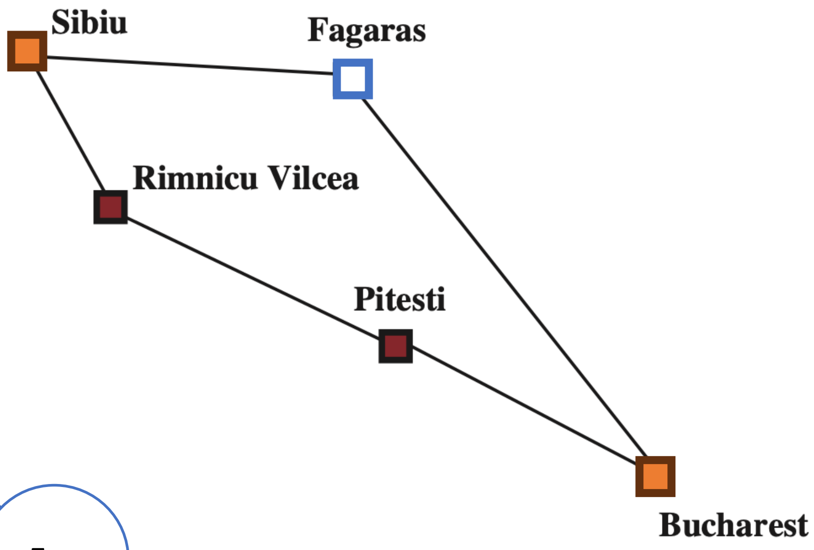

```
create frontier : queue

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```



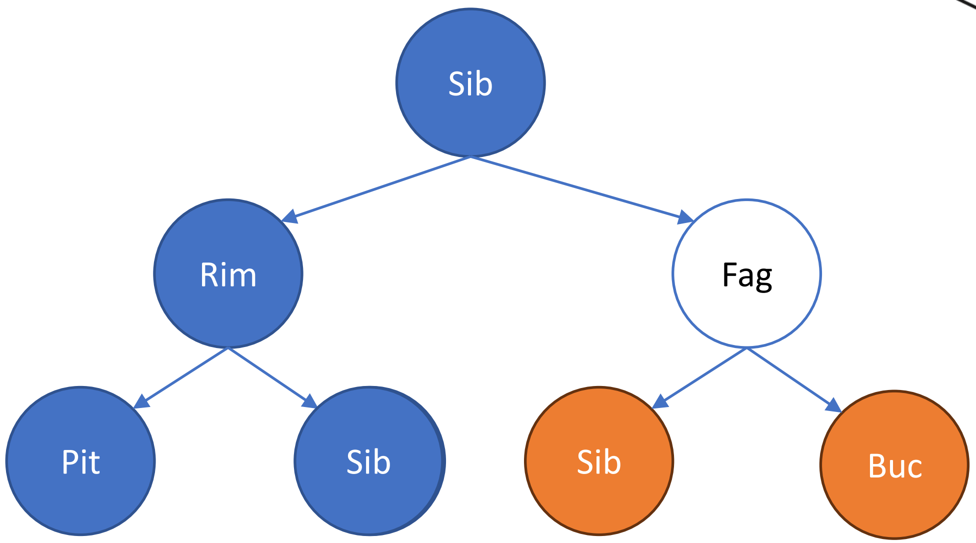
Queue:



Breadth-first Search




```
create frontier : queue

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```

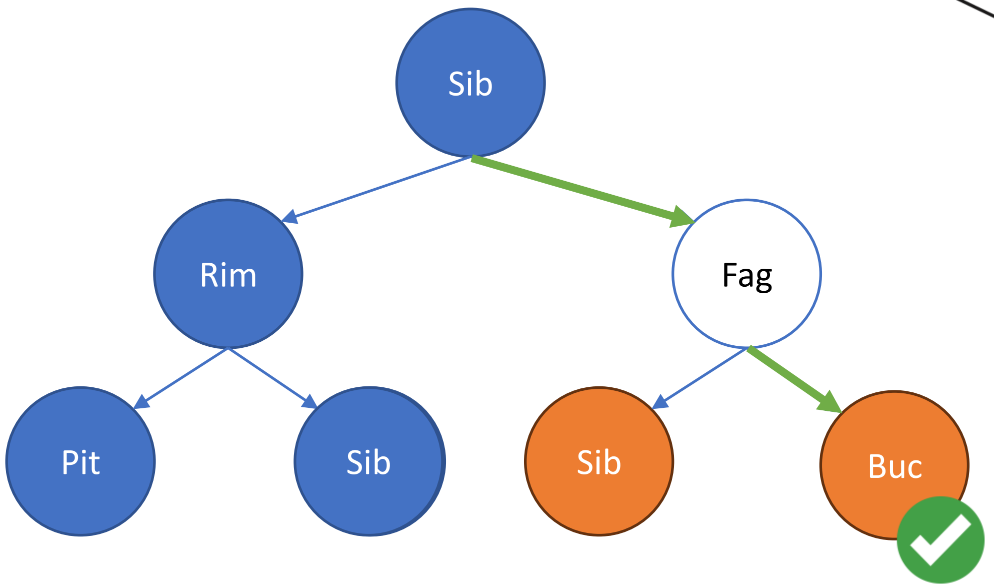


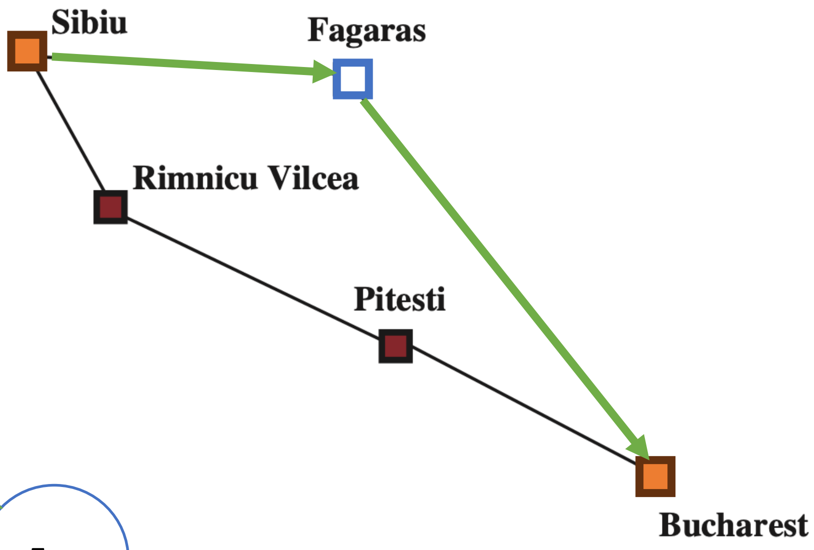
Queue:

Breadth-first Search

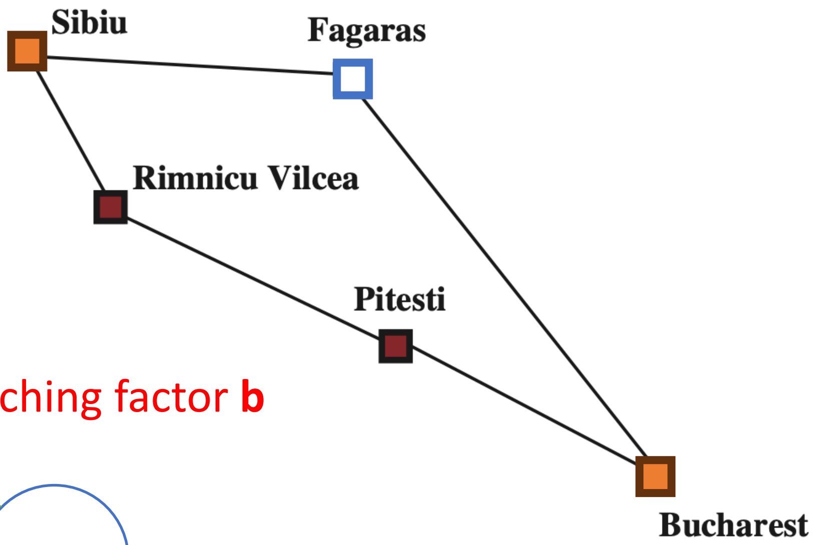
```
create frontier : queue

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```

Queue:

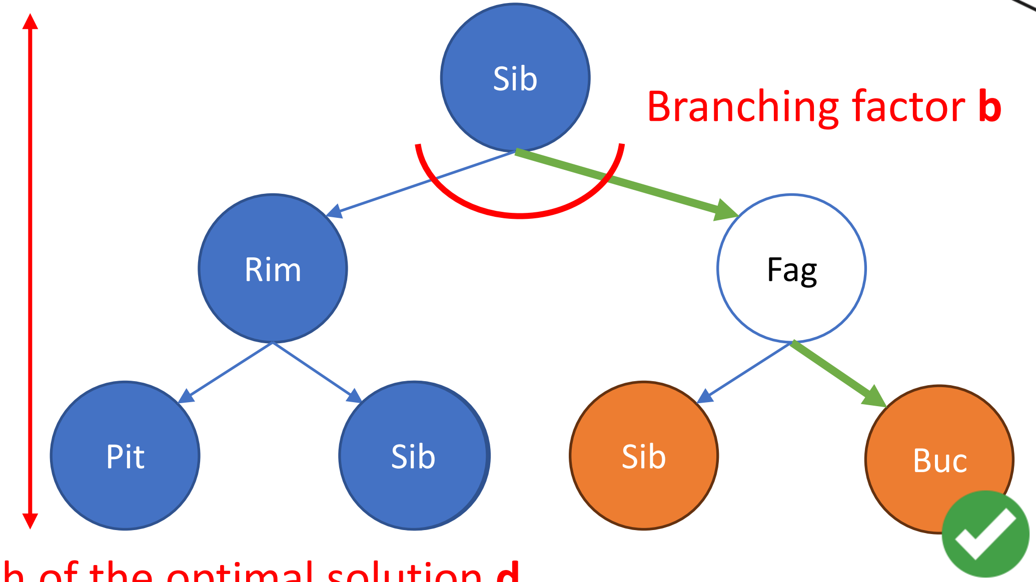




Breadth-first Search




```
create frontier : queue

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```

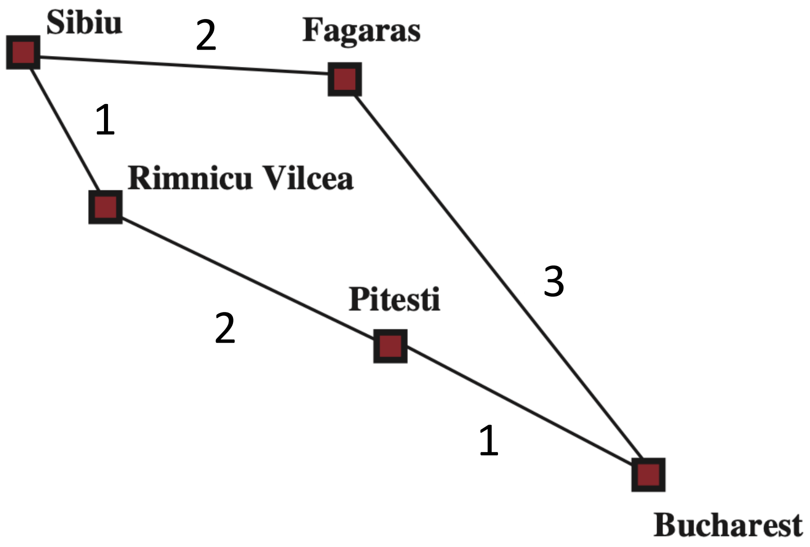


Queue:

- **Time complexity (# nodes expanded)?**
  - $1 + b + b^2 + \cdots + b^d = O\left(b^d\right)$
- **Space complexity?**
  - $O\left(b^d\right)$, worst case: expand the last child in a branch
- **Complete?** Yes, if B is finite
- **Optimal?** Yes, if uniform cost

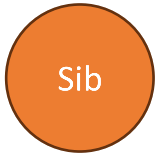
Uniform-cost Search

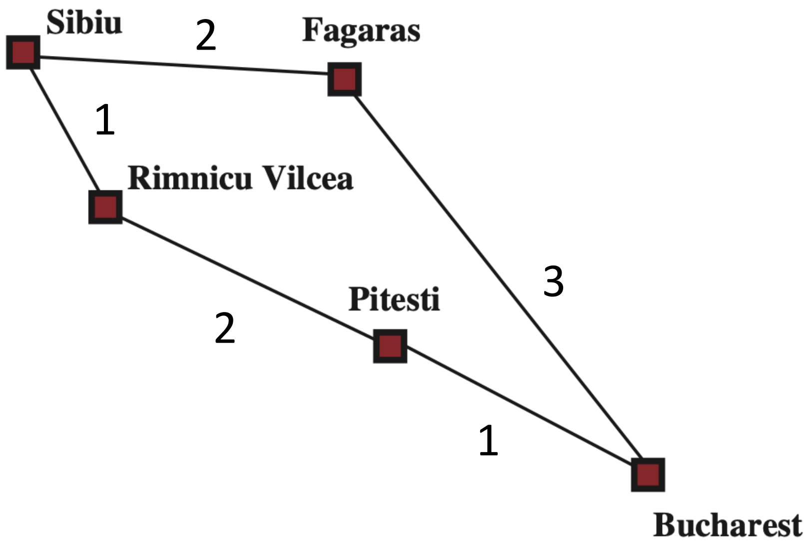
```
create frontier : priority queue (path cost)


insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):


      next state = transition(state, action)
      frontier.add(next state)
return failure
```



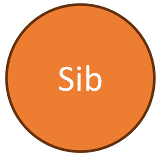
Uniform-cost Search

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


insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):


      next state = transition(state, action)
      frontier.add(next state)
return failure
```



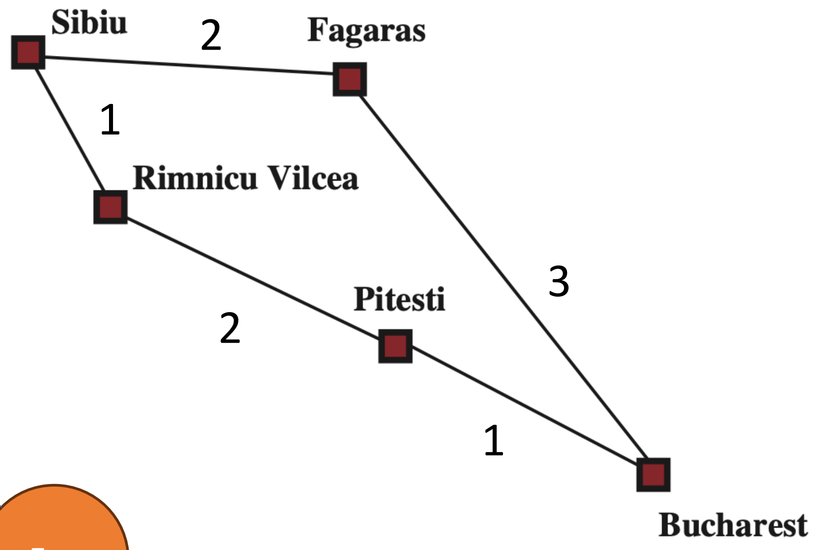


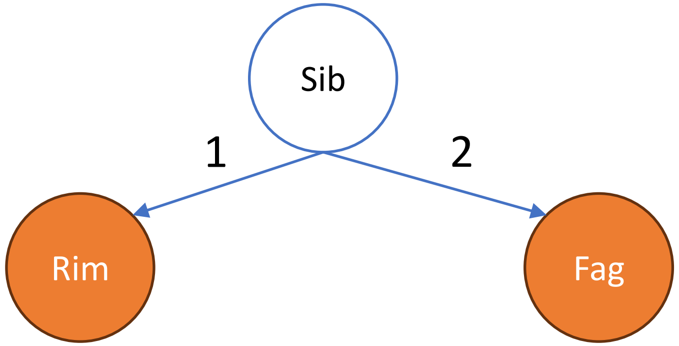
Priority Queue:



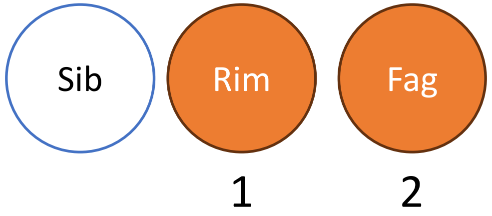
Uniform-cost Search

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

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

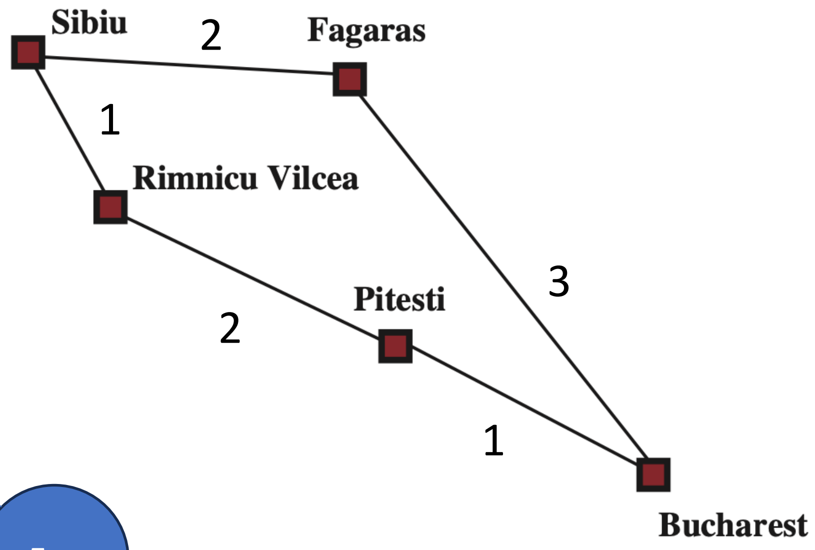




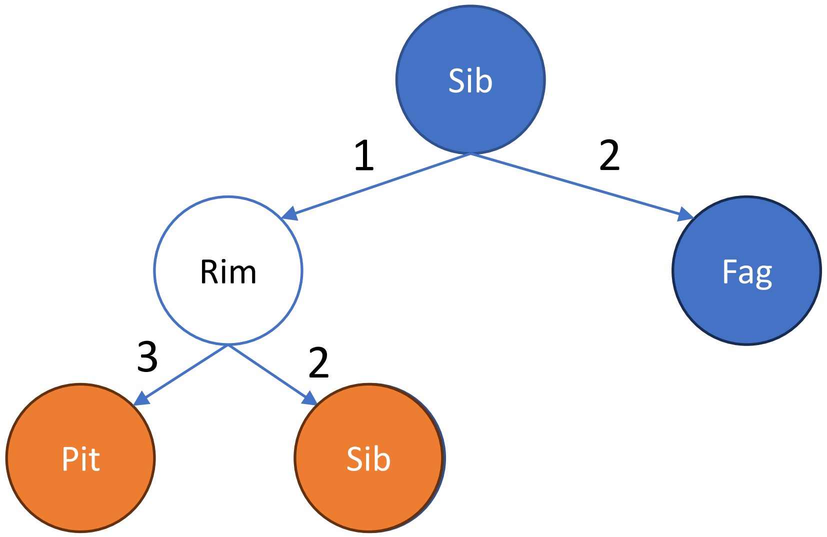
Priority Queue:



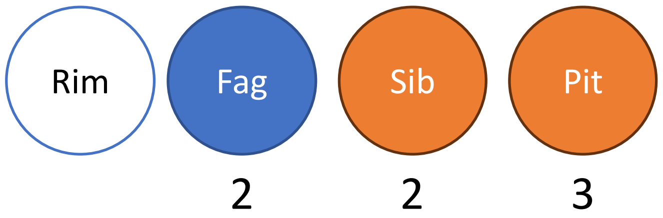
Uniform-cost Search



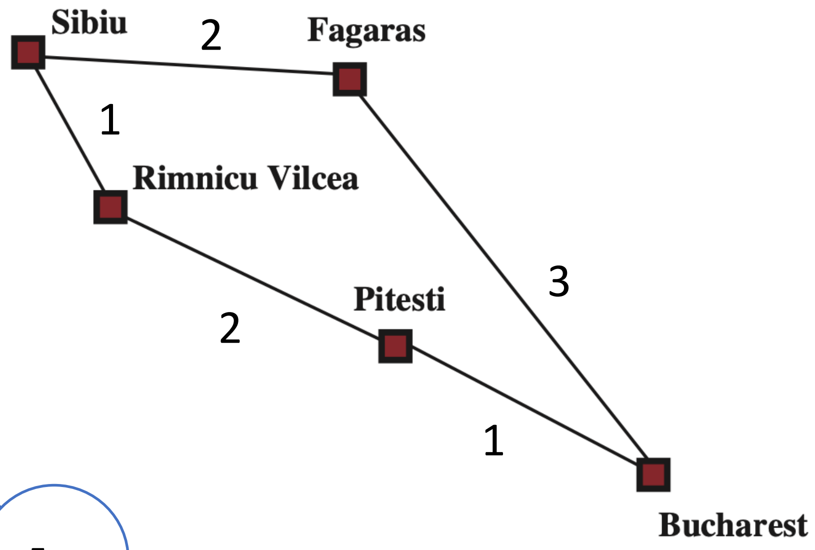

```
create frontier : priority queue (path cost)

insert initial state
while frontier is not empty:
    state = frontier.pop()
    if state is goal: return solution
    for action in actions(state):

        next state = transition(state, action)
        frontier.add(next state)
return failure
```



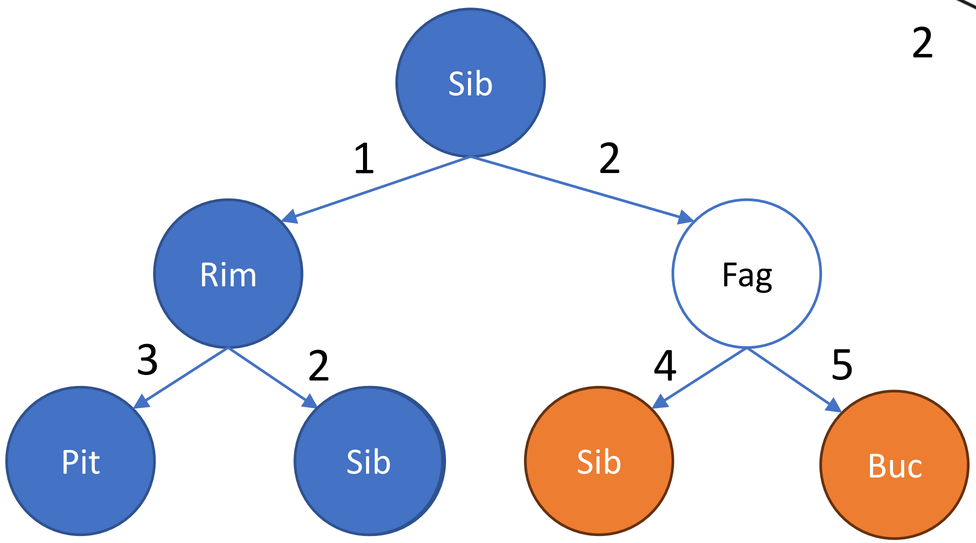
Priority Queue:



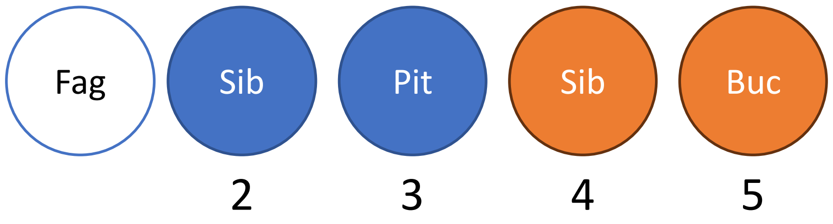
Uniform-cost Search




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

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

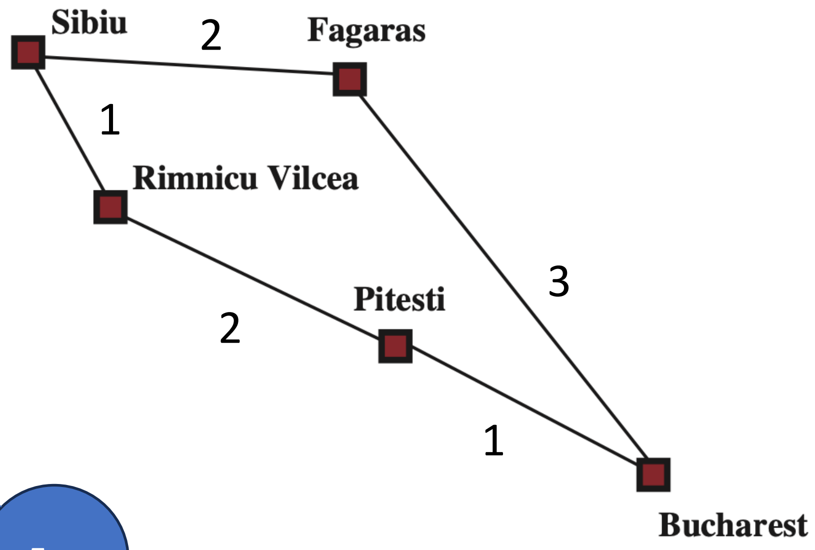


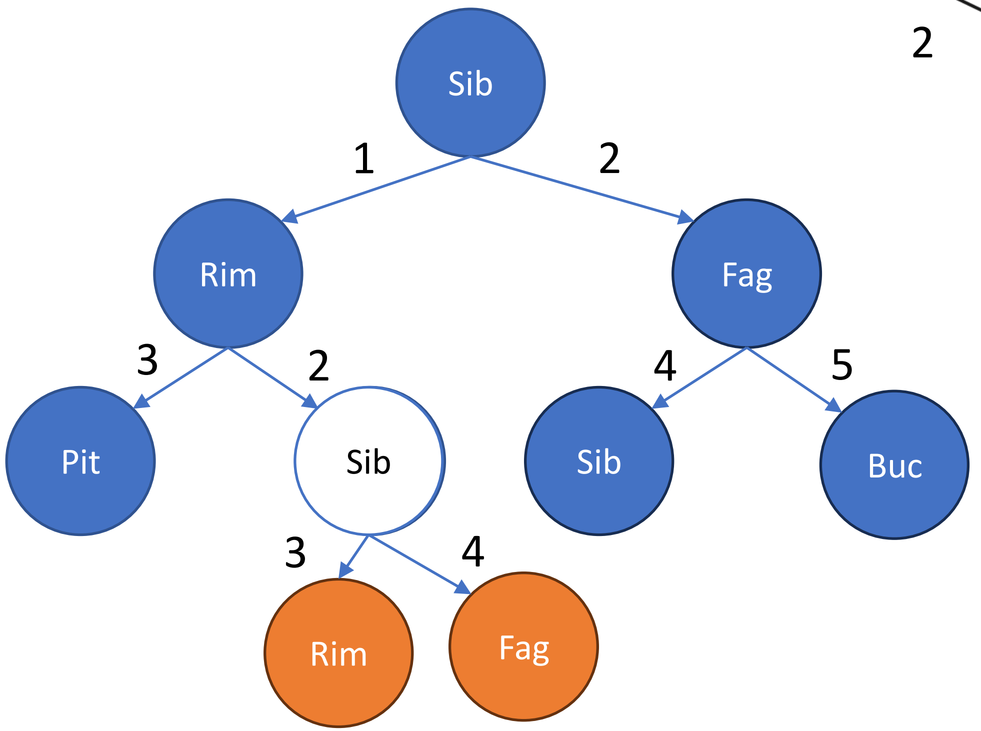
Priority Queue:



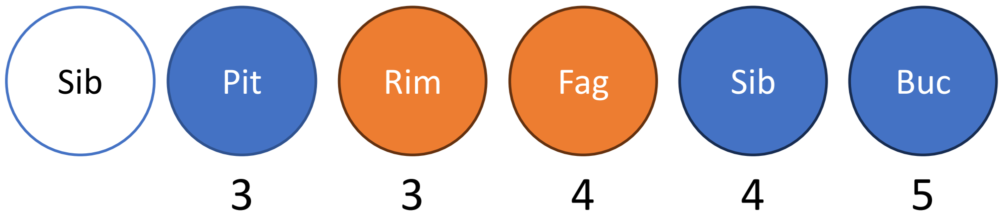
Uniform-cost Search

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


insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):


      next state = transition(state, action)
      frontier.add(next state)
return failure
```

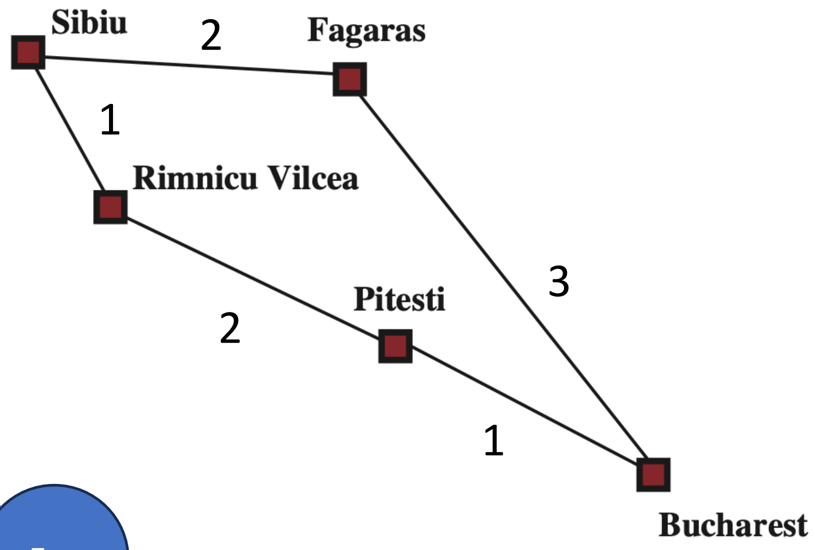




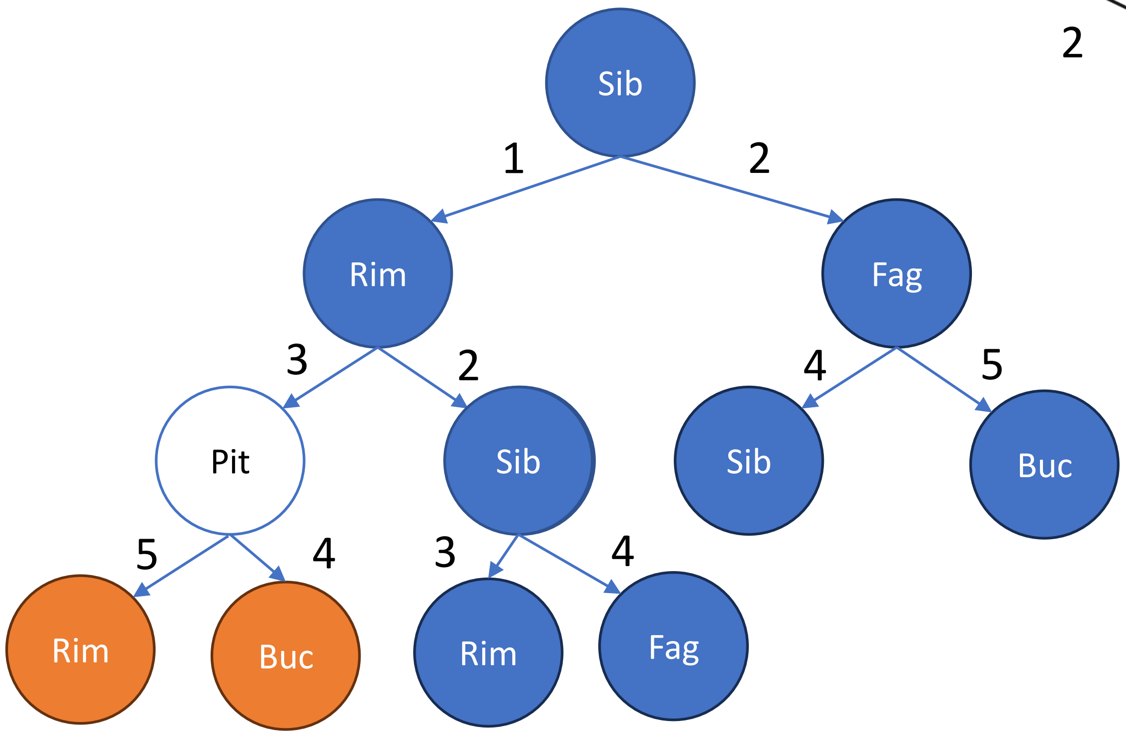
Priority Queue:



Uniform-cost Search



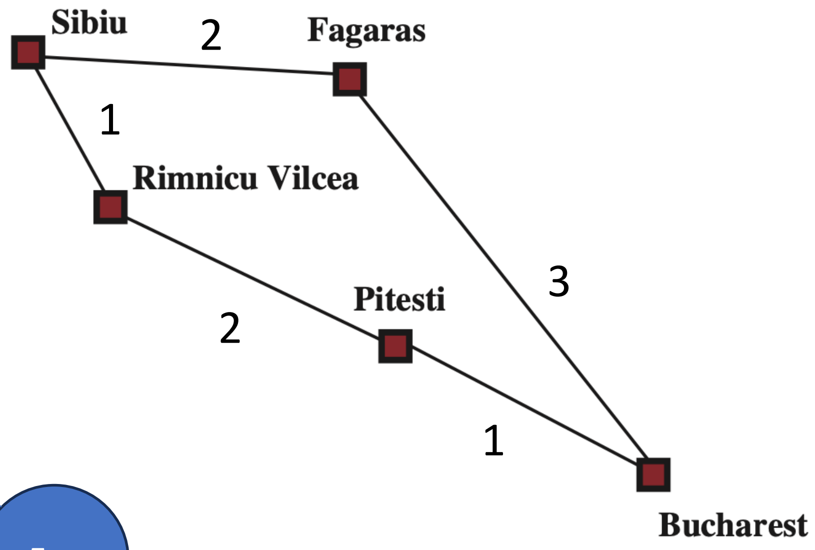

```
create frontier : priority queue (path cost)

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

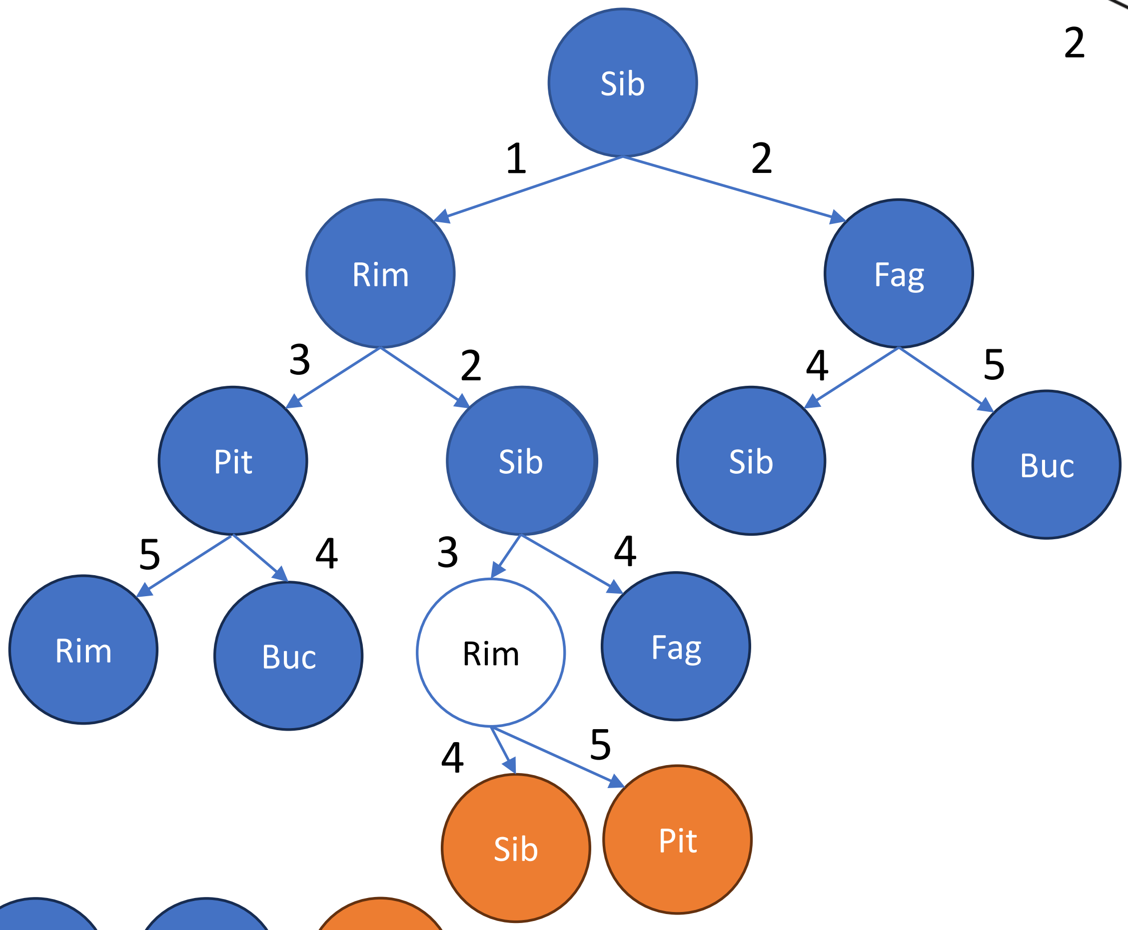


Priority Queue:

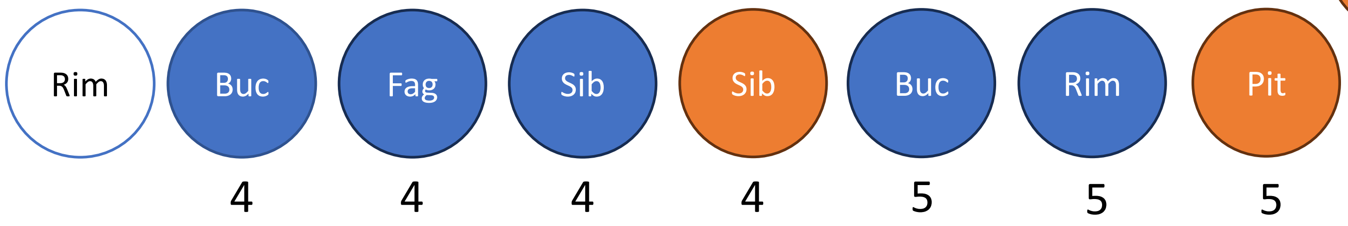
Uniform-cost Search

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

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```



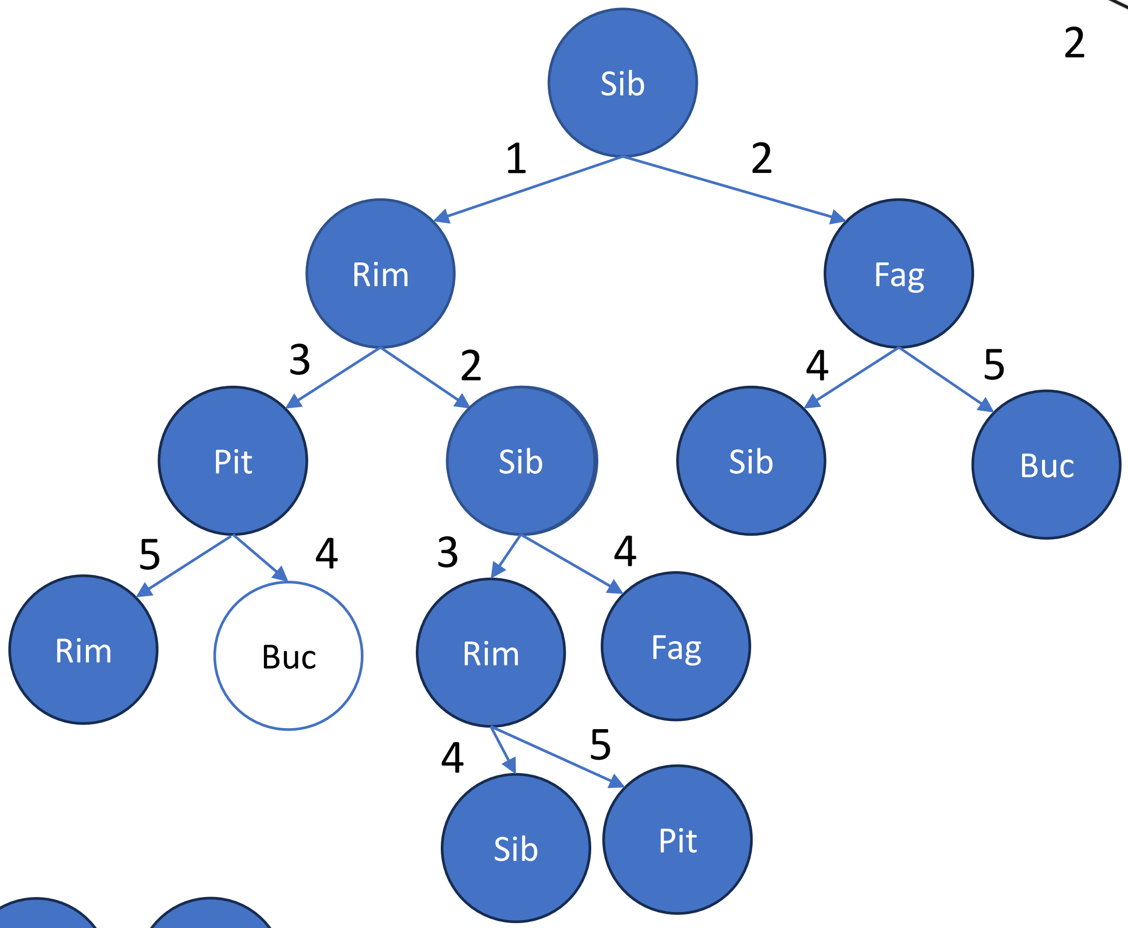


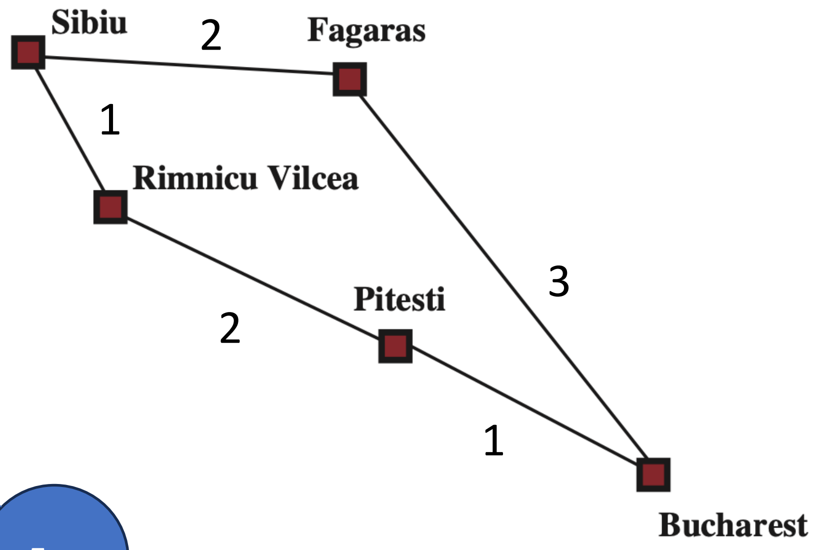
Priority Queue:



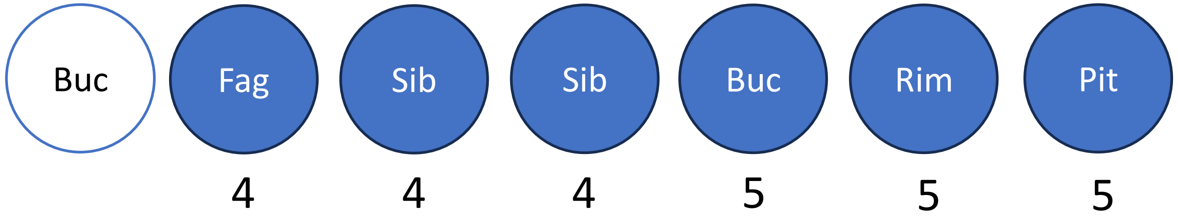
Uniform-cost Search

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


insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):


      next state = transition(state, action)
      frontier.add(next state)
return failure
```

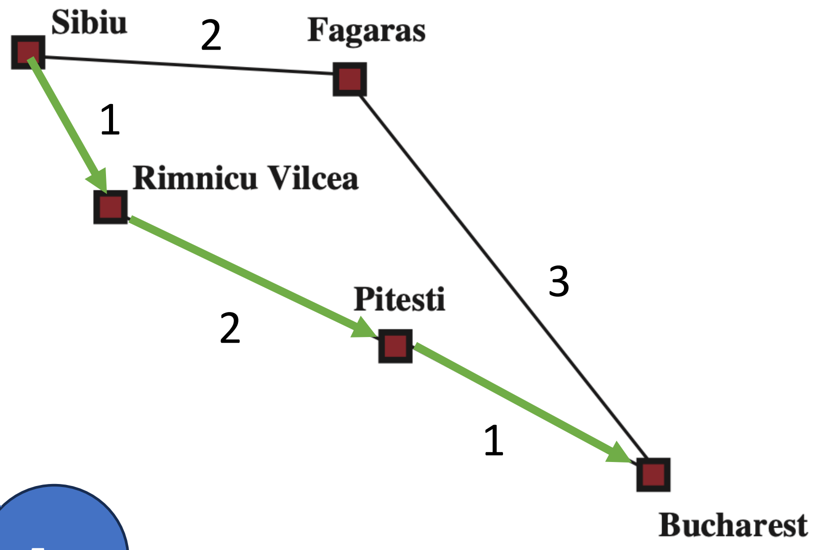




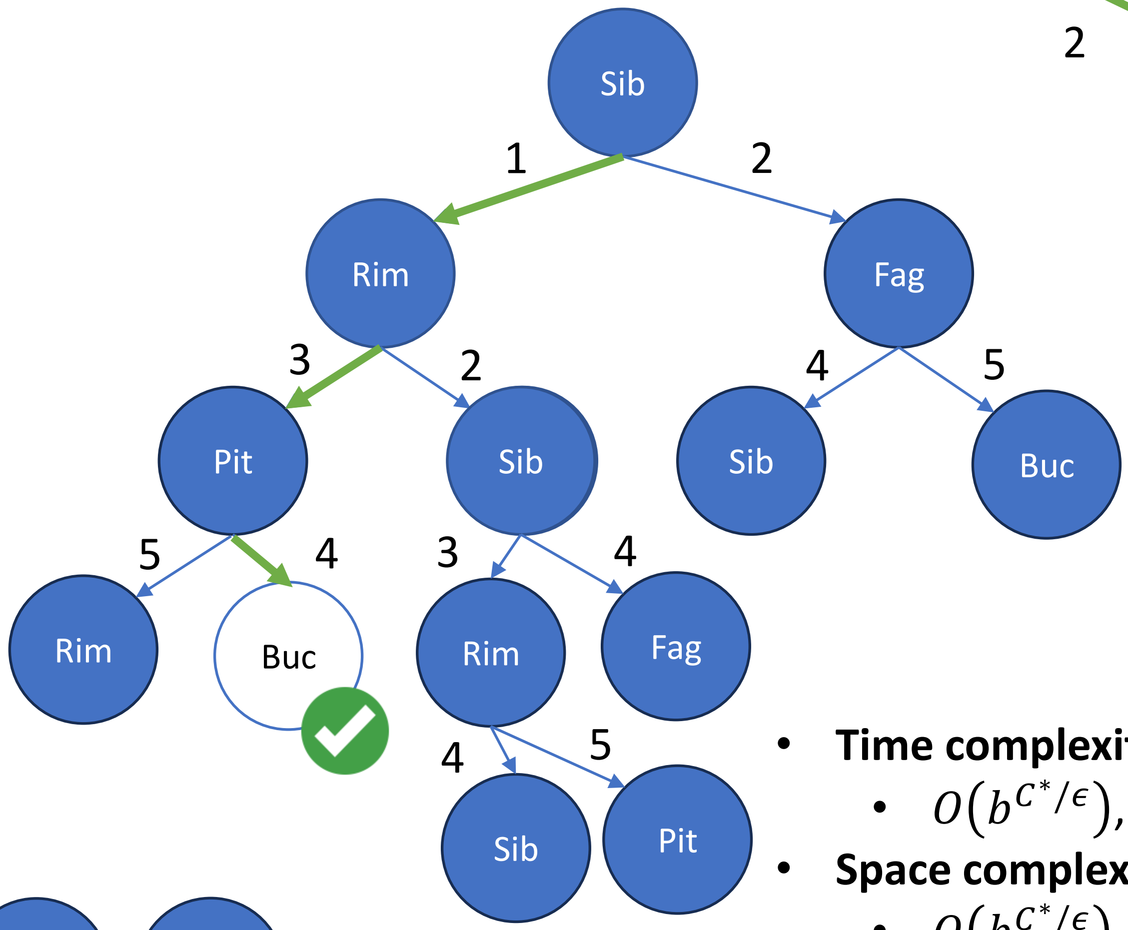
Priority Queue:



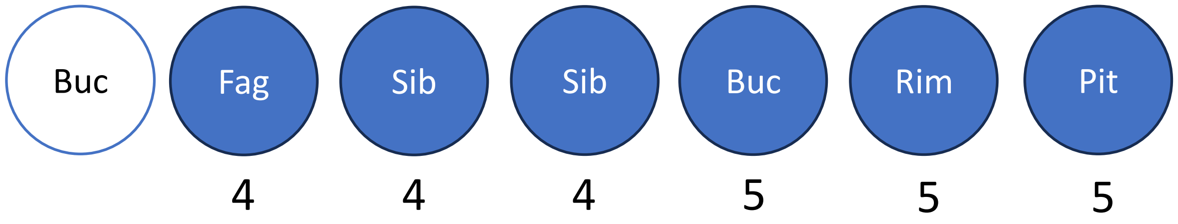
Uniform-cost Search




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

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```



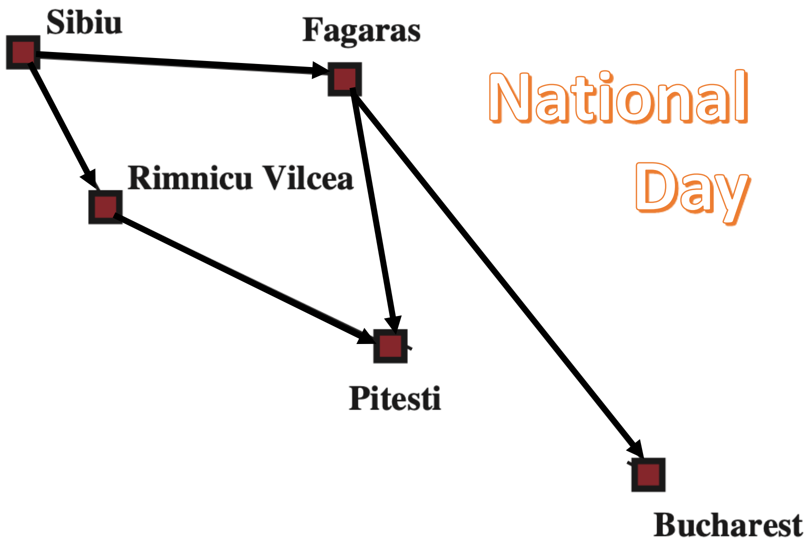
Priority Queue:



- **Time complexity (# nodes expanded)?**
  - $O\left(b^{C^*/\epsilon}\right)$, C^* cost of optimal solution
- **Space complexity?**
  - $O\left(b^{C^*/\epsilon}\right)$
- **Complete?** Yes, if step cost $\geq \epsilon$
- **Optimal?** Yes

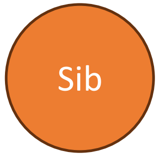
Depth-first Search

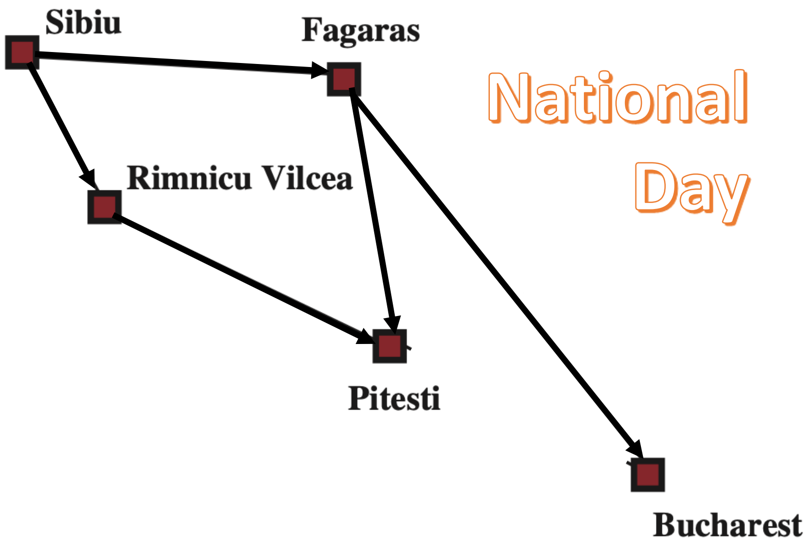
```
create frontier : stack

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```



Depth-first Search

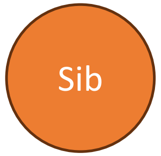
```
create frontier : stack

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```





National Day

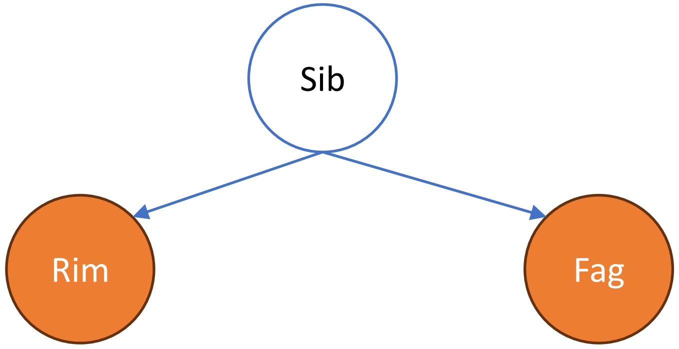
Stack:

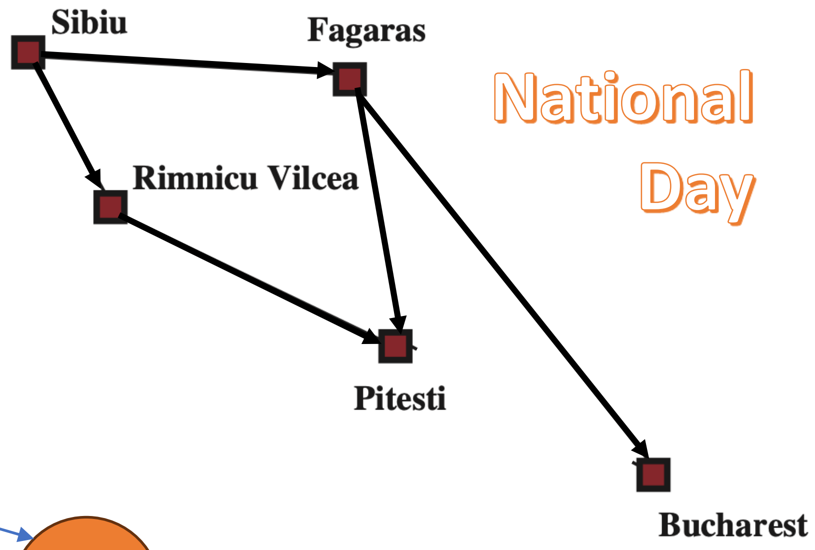


Depth-first Search

```
create frontier : stack

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```

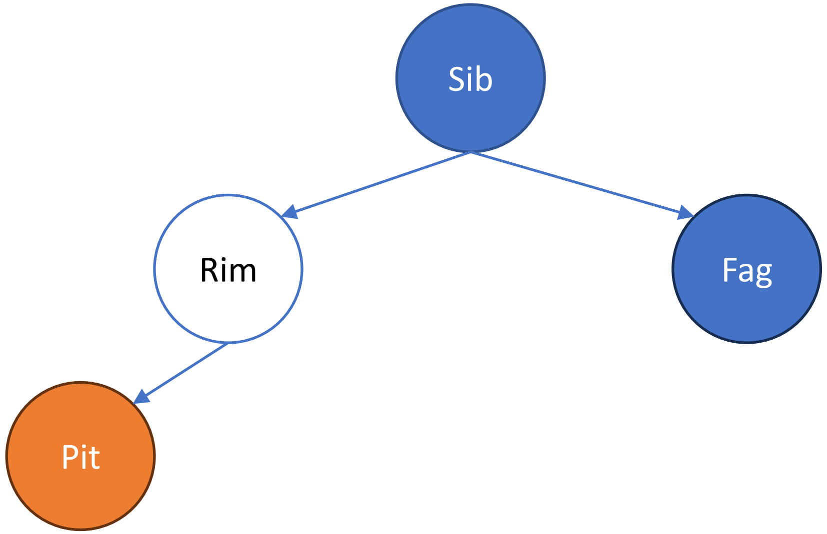
Stack:

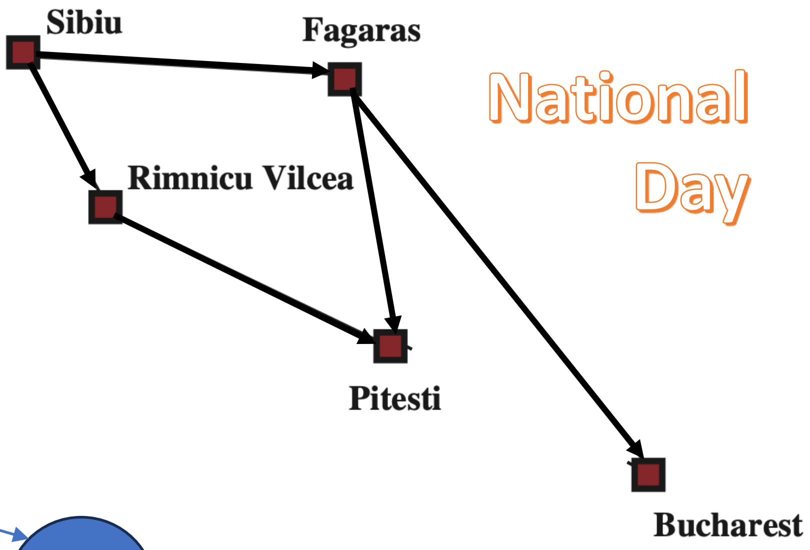




Depth-first Search

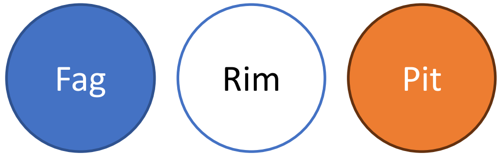
```
create frontier : stack

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```





National Day

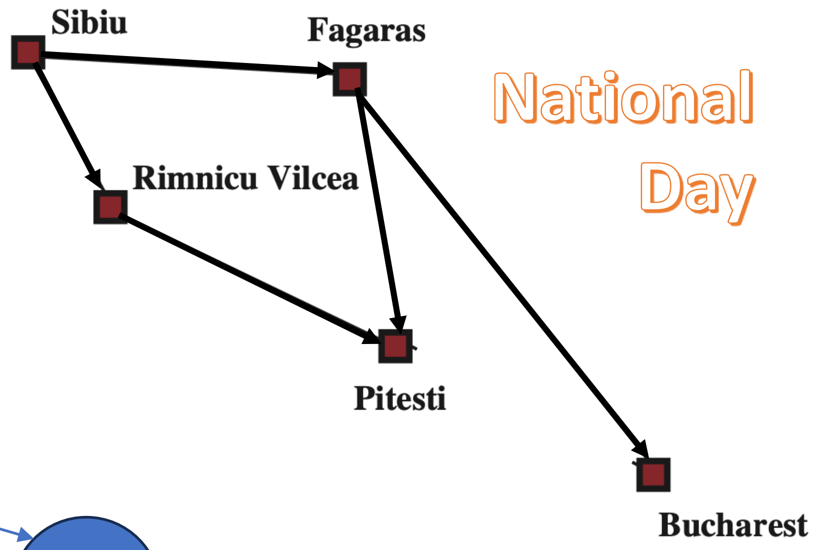
Stack:

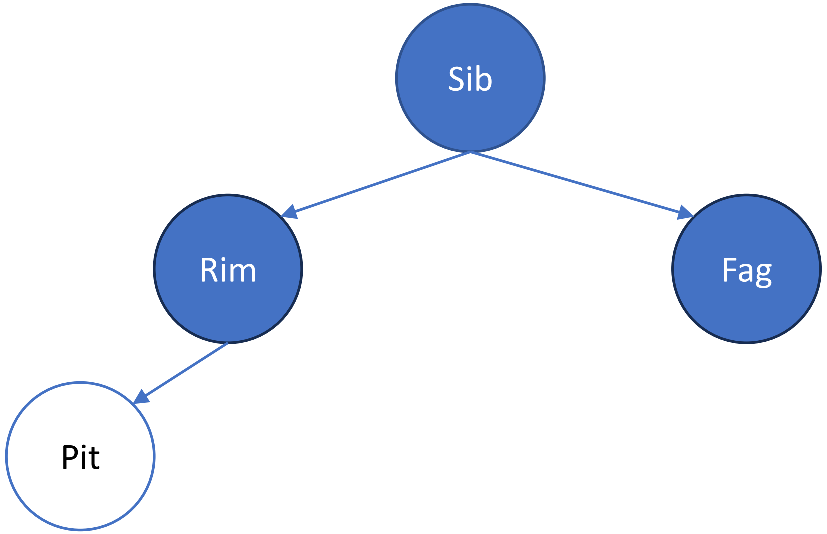


Depth-first Search

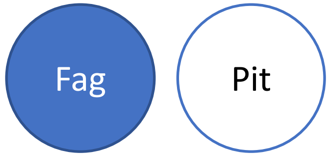
National Day

```
create frontier : stack

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```



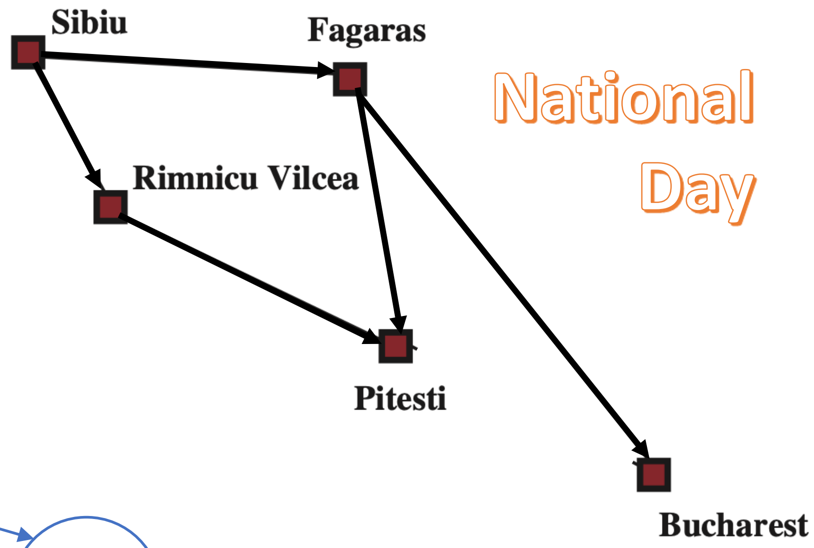


Stack:

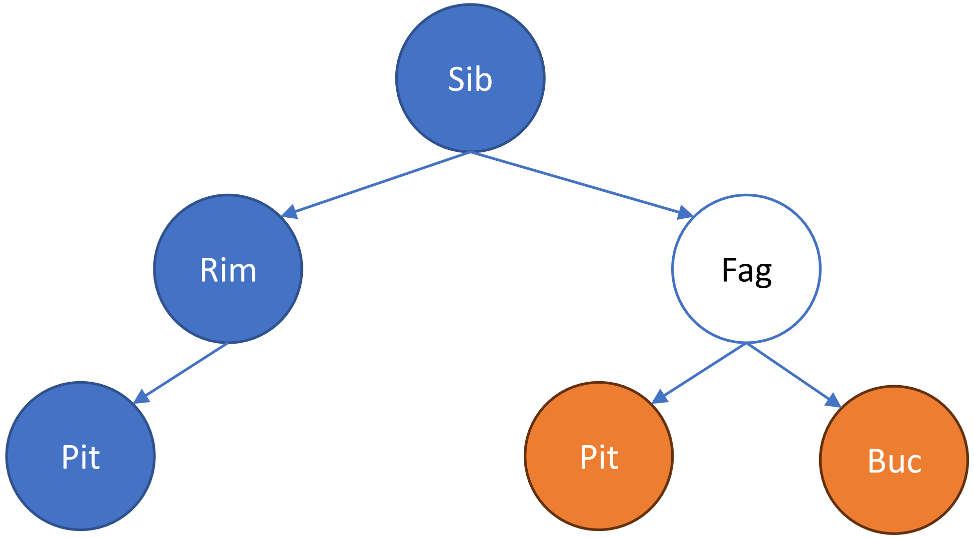


Depth-first Search

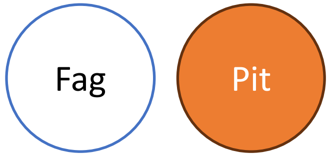
```
create frontier : stack

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```



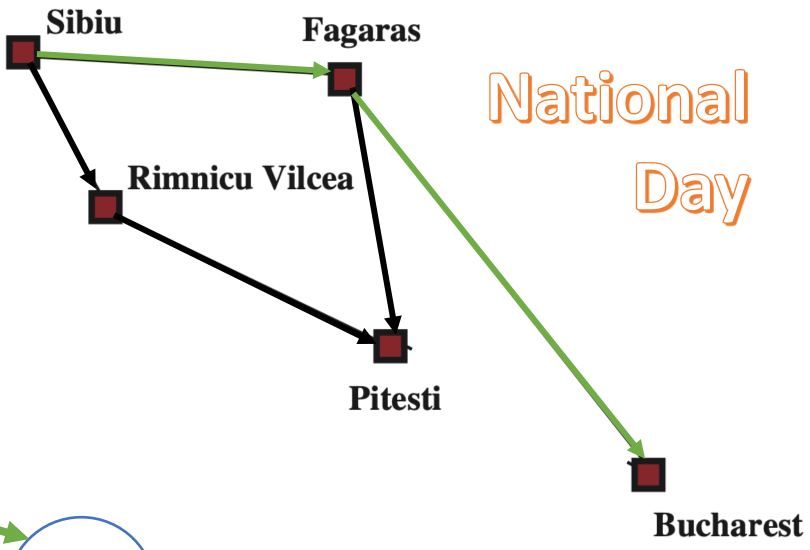
National
Day



Stack:

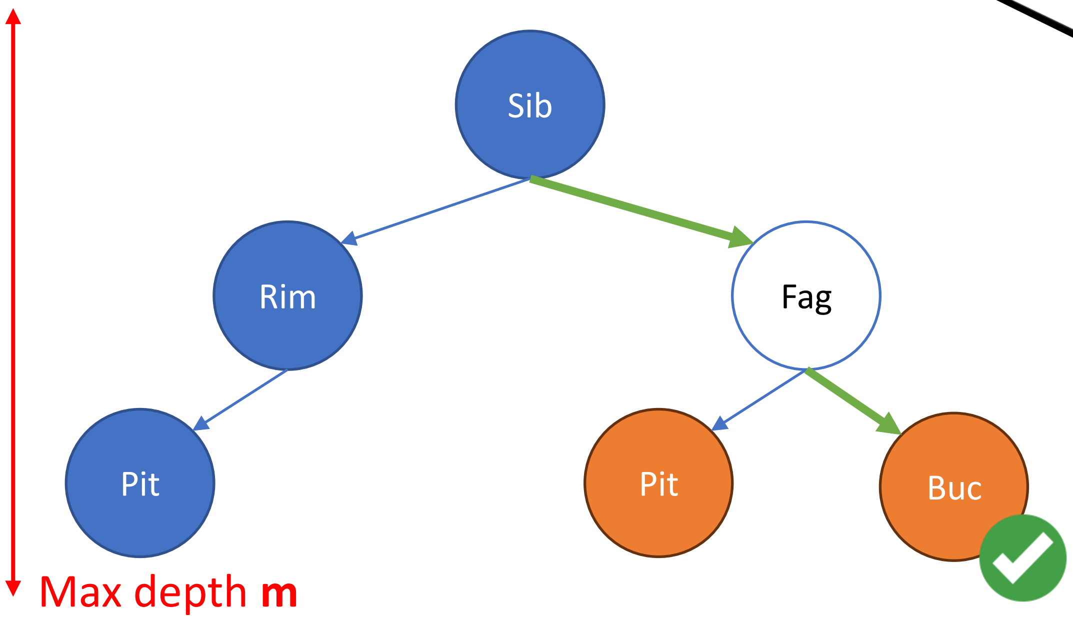


Depth-first Search



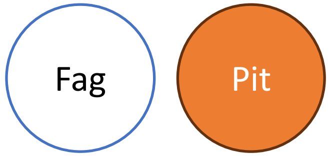
National Day

```
create frontier : stack

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```



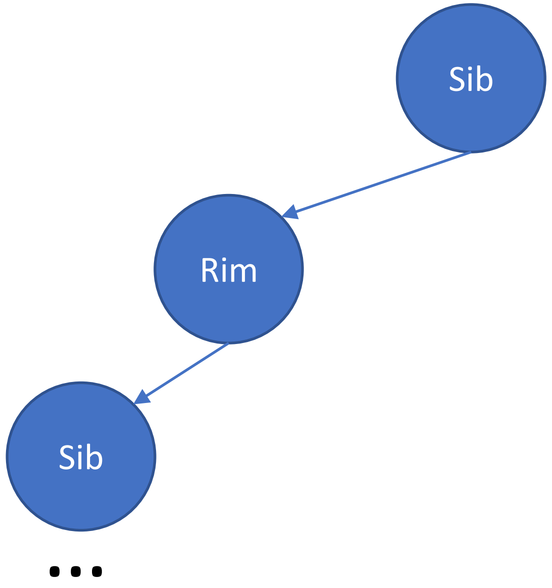
- **Time complexity (# nodes expanded)?**
  - $O(b^m)$
- **Space complexity?**
  - $O(bm)$
- **Complete?**

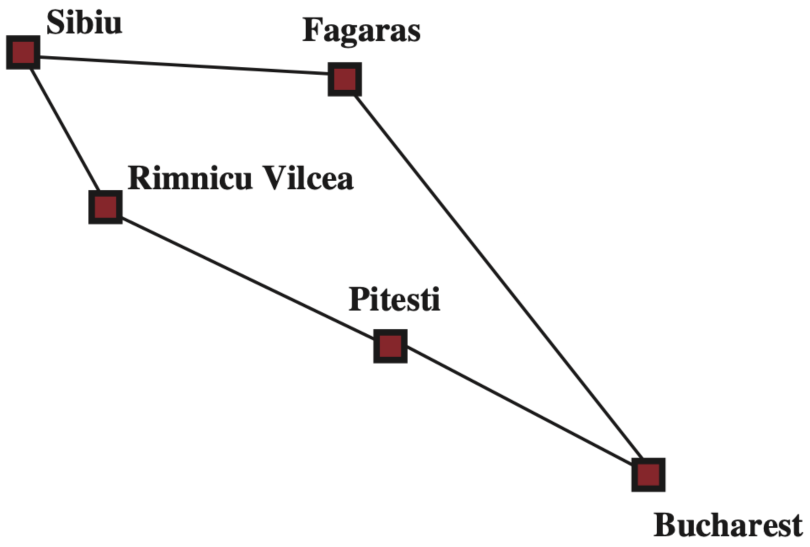
Stack:



Depth-first Search

```
create frontier : stack


insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)


      if next state is goal: return solution
      frontier.add(next state)
return failure
```

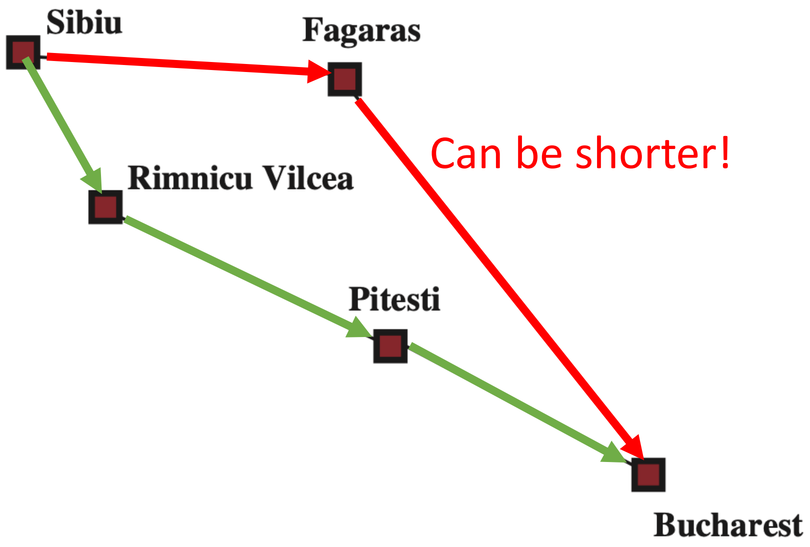


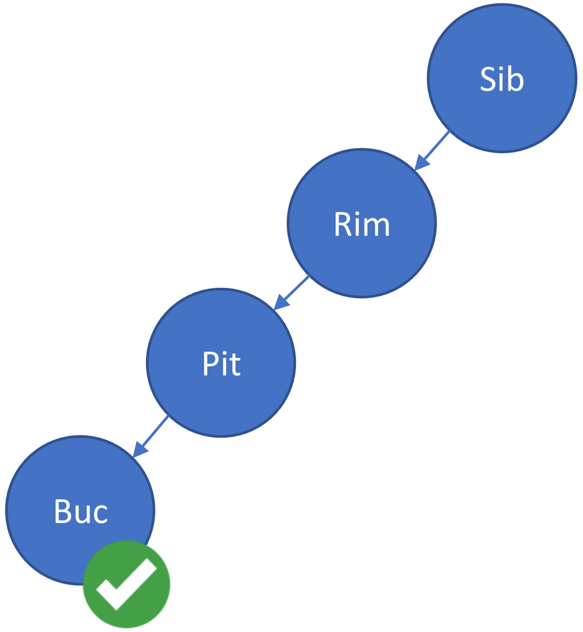


- **Time complexity (# nodes expanded)?**
  - $O(b^m)$
- **Space complexity?**
  - $O(bm)$
- **Complete?** No, when depth is infinite or can go back and forth (loops)
- **Optimal?**

Depth-first Search

```
create frontier : stack

insert initial state
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)

      if next state is goal: return solution
      frontier.add(next state)
return failure
```





- **Time complexity (# nodes expanded)?**
  - $O(b^m)$
- **Space complexity?**
  - $O(bm)$
- **Complete?** No, when depth is infinite or can go back and forth (loops)
- **Optimal?** No

How do we handle infinite depth?

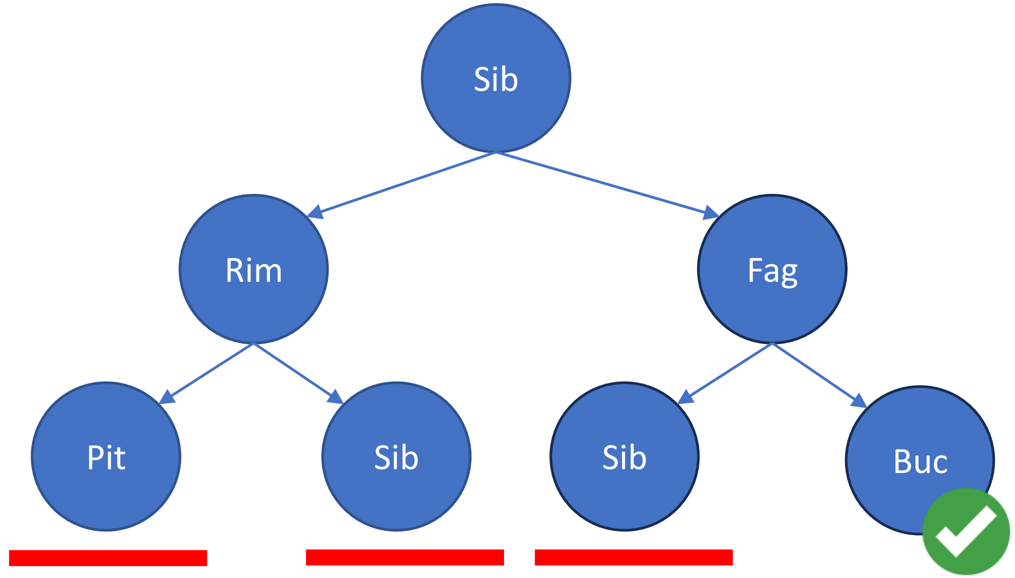
Outline

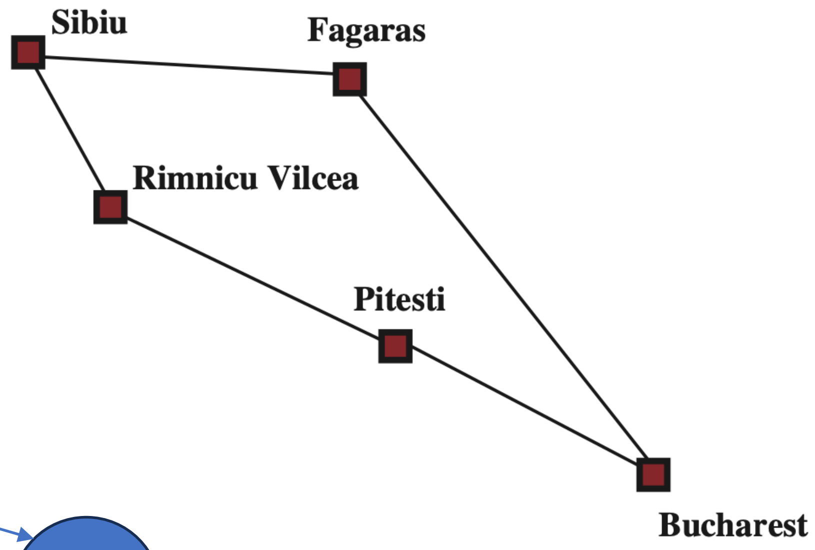
- Problem-solving agents
- Search algorithms
- Uninformed search algorithms
  - Breadth-first Search (BFS)
  - Uniform-cost search
  - Depth-first Search (DFS)
- **Variants of uninformed search algorithms**
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search
- Dealing with repeated states
- Informed search algorithms
  - Greedy best-first search
  - A* search
  - Heuristics
- Variants of A*

Depth-limited Search (DLS)

- Limit the search depth
- Backtrack once the depth limit is reached

Depth limit **l** = 2





How do we know the depth of the solution?

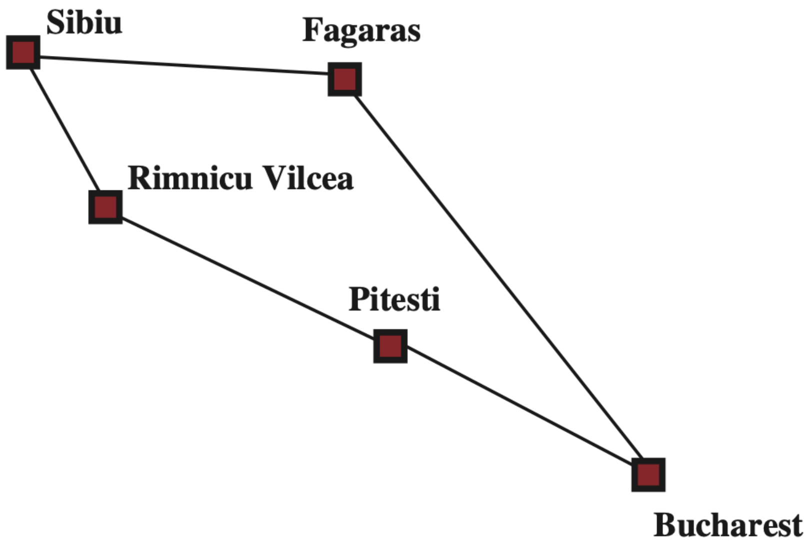
We don't know :(

- **Time complexity (# nodes expanded)?**
  - $b^0 + b^1 + \cdots + b^l = O(b^l)$
- **Space complexity?**
  - $O(bl)$
- **Complete?** No
- **Optimal?** No

Iterative Deepening Search (IDS)

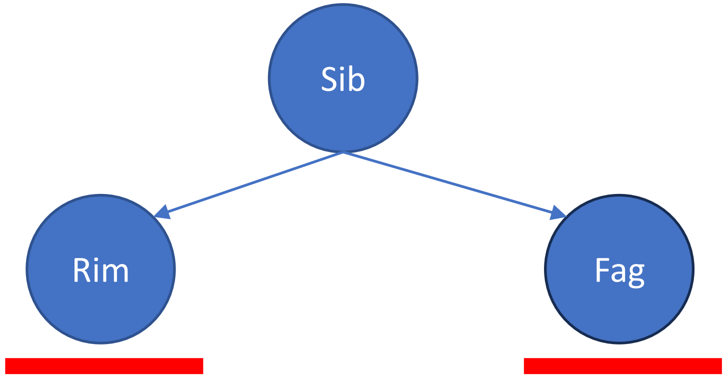
- Do depth-limited search with max depth 0 … N
- Return solution if found
- Increase the depth otherwise

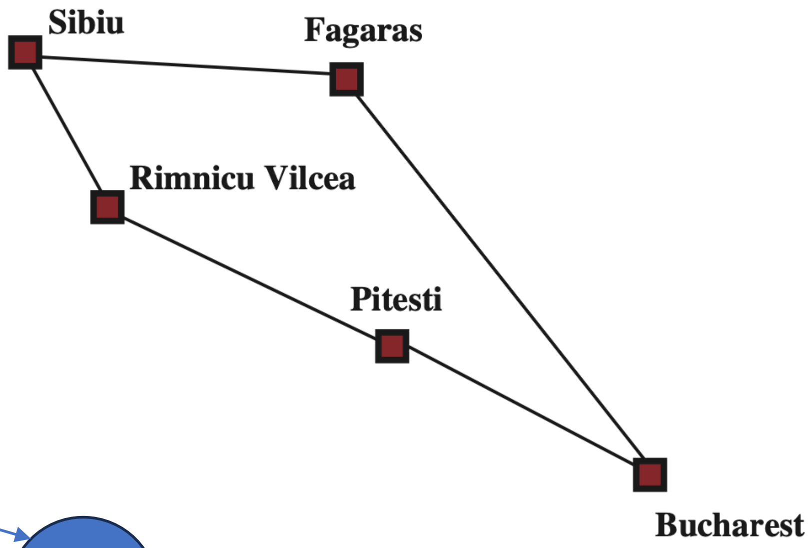
Depth limit **l** = 0



Iterative Deepening Search (IDS)

Depth limit **l** = 1

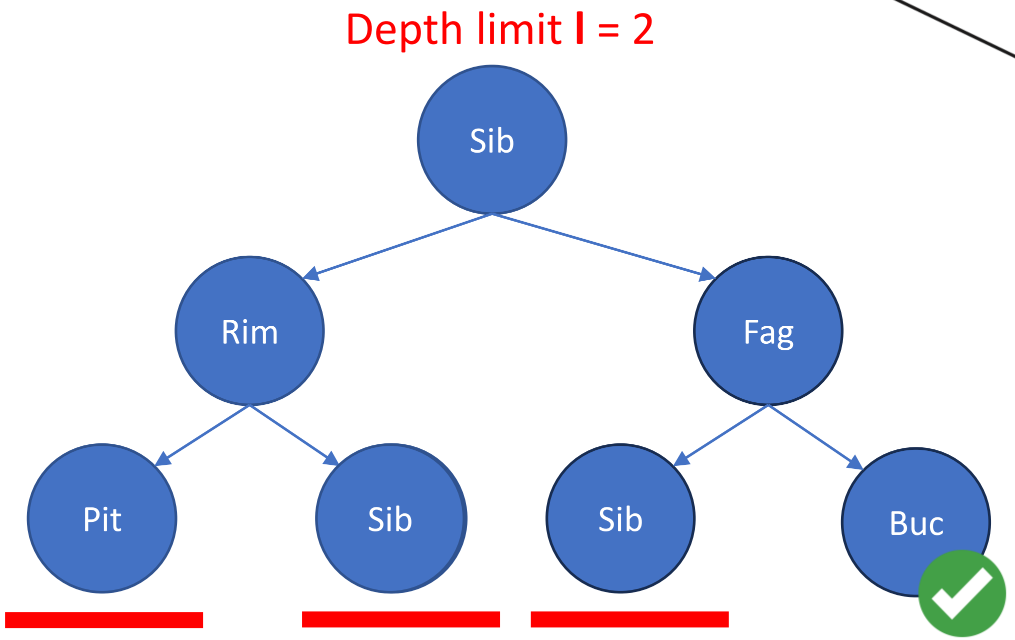


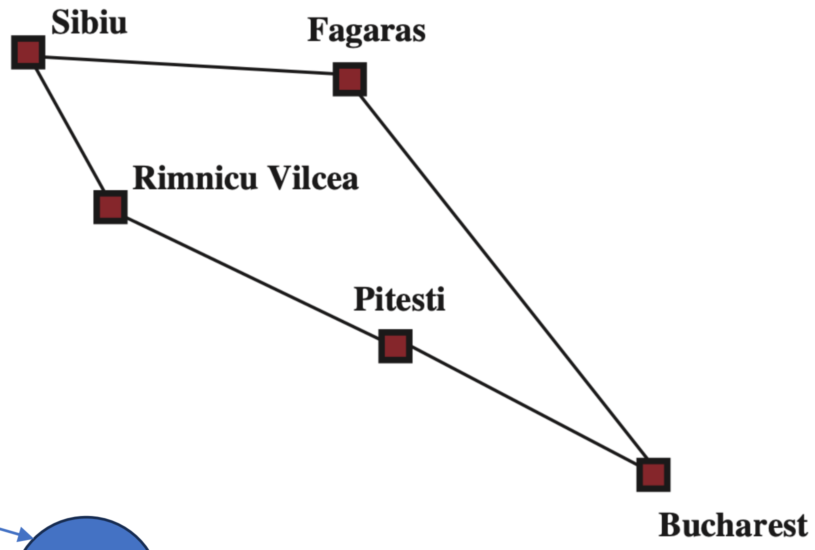


- Do depth-limited search with max depth 0 … N
- Return solution if found
- Increase the depth otherwise

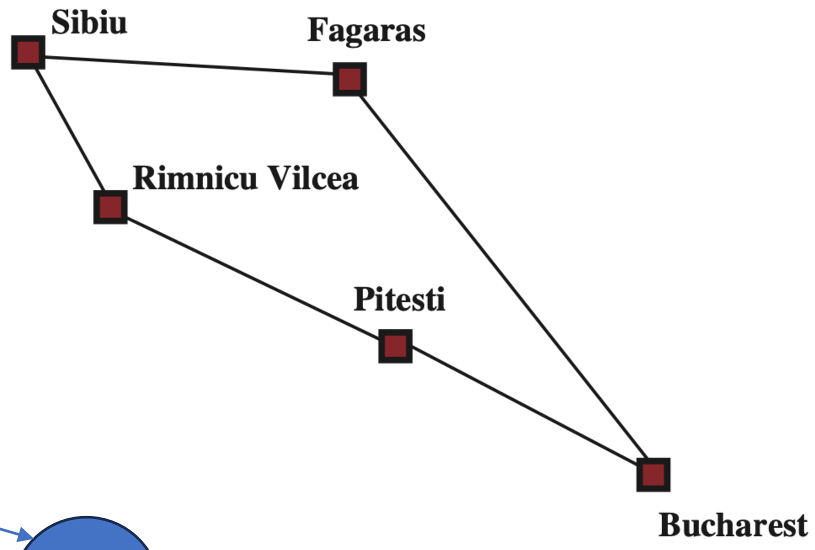
Iterative Deepening Search (IDS)

- Do depth-limited search with max depth 0 … N
- Return solution if found
- Increase the depth otherwise



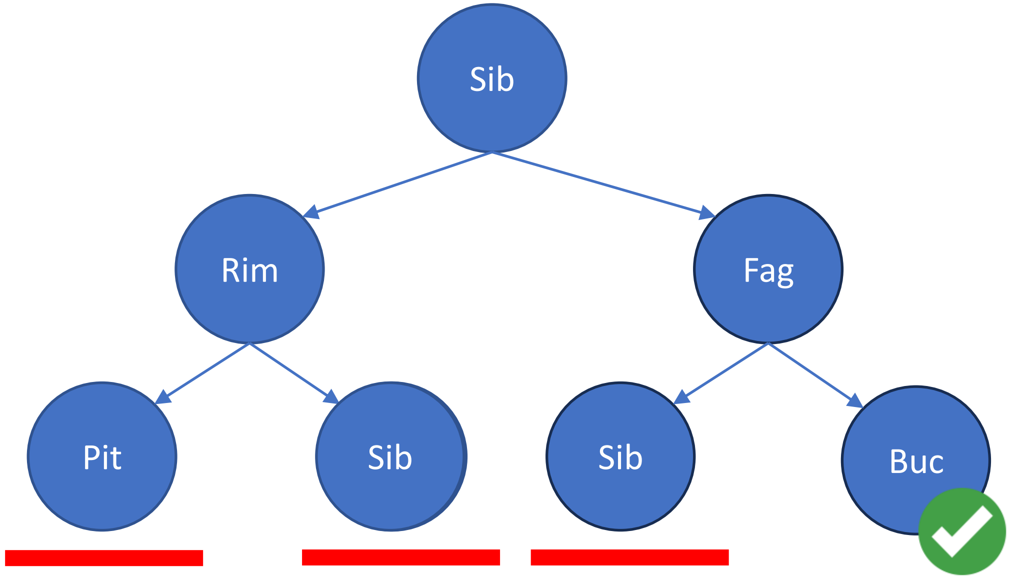


Iterative Deepening Search (IDS)



- Do depth-limited search with max depth 0 … N
- Return solution if found
- Increase the depth otherwise

Depth limit **l** = 2



- **Time complexity (# nodes expanded)?**
  - Overhead! $\boxed{b^0 + (b^0 + b^1) + \cdots +} (b^0 + \cdots + b^d)$
  - $= (d+1)b^0 + db^1 + (d-1)b^2 + \cdots + 2b^{d-1} + b^d = O(b^d)$
- **Space complexity?**
  - $O(bd)$
- **Complete?** Yes
- **Optimal?** Yes, if uniform step cost

Iterative Deepening Search (IDS)

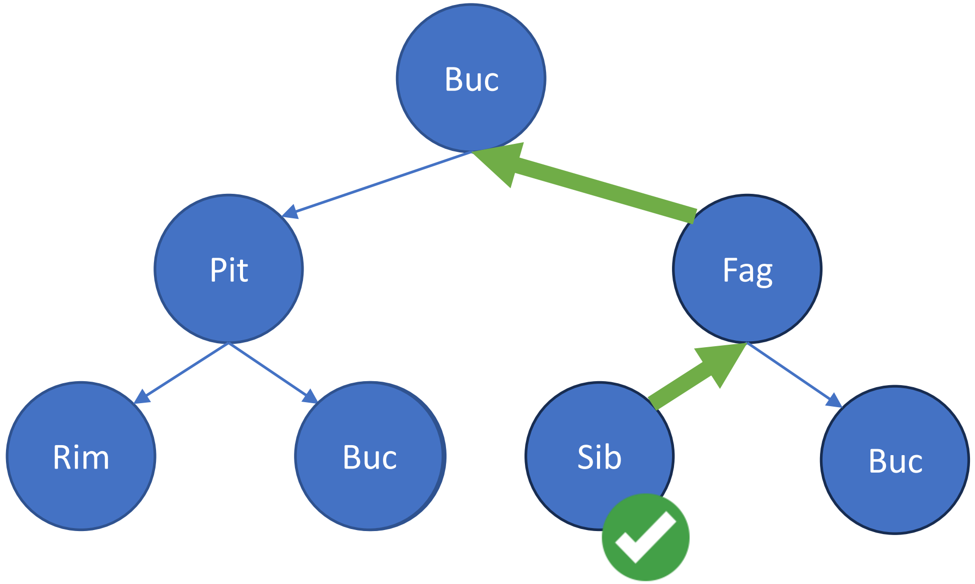
For b=10, d=5,

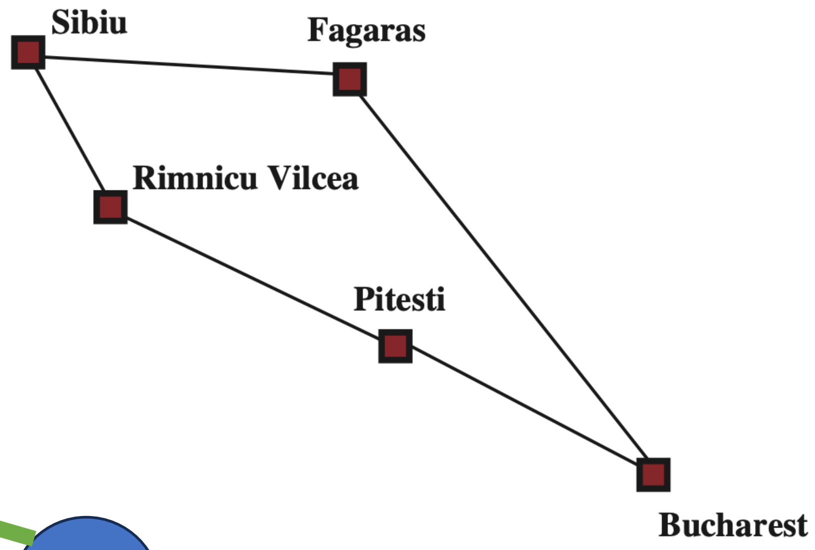
- N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
- N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

Overhead = (123,456 - 111,111)/111,111 = 11%

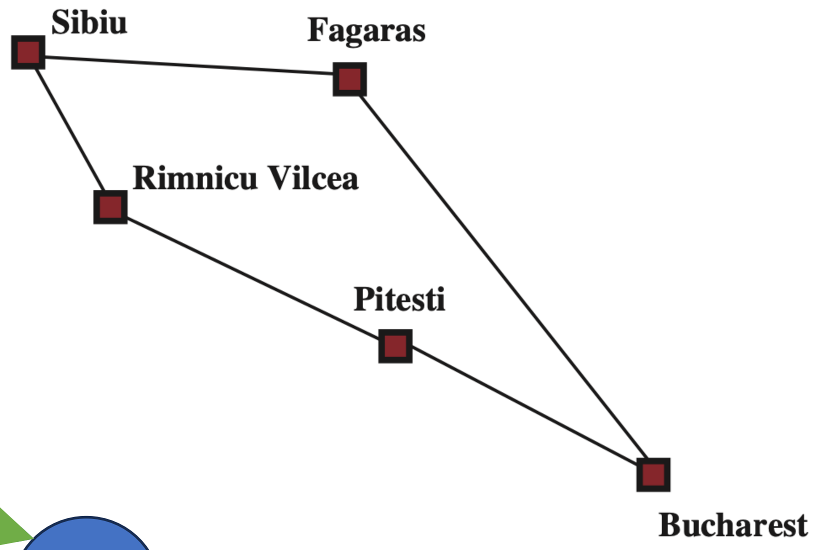
Backward Search

Search from the goal





Bidirectional Search



Combine search:

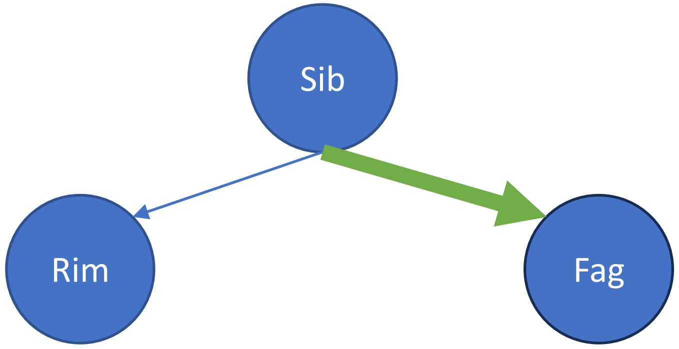
- Forward (from the start)
- Backward (from the goal)

Stop when two searches meet

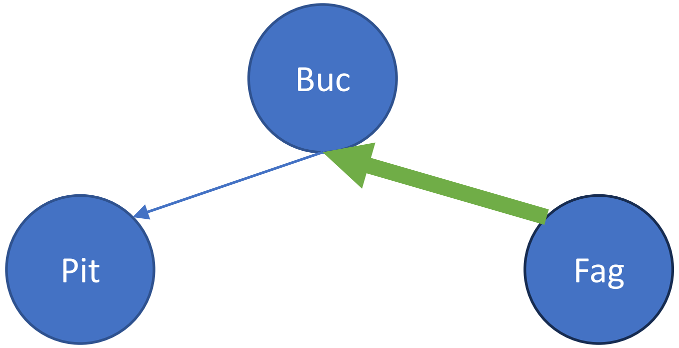
Intuition:

$$2 \times O\left(b^{d/2}\right) < O\left(b^{d}\right)$$

Forward:



Backward:



Issues?

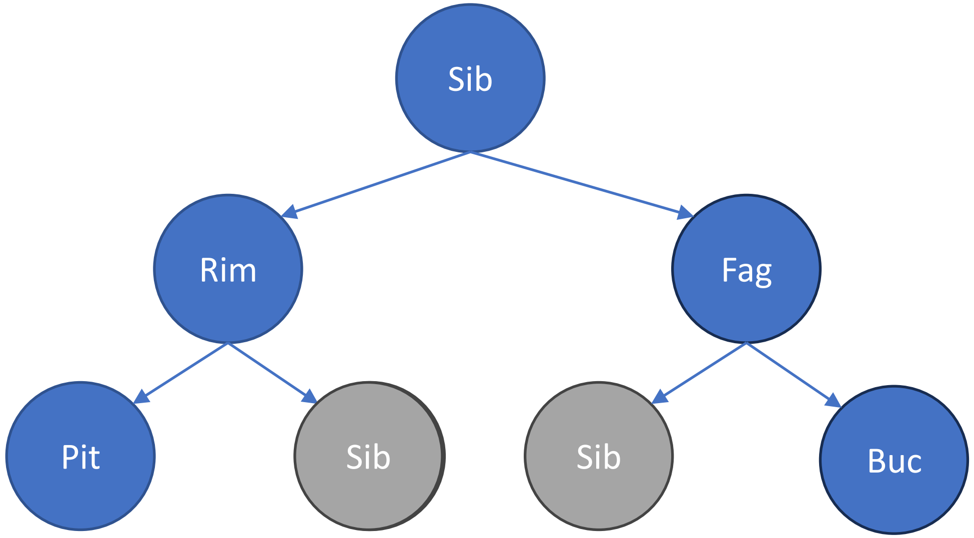
- Operators need to be reversible
- Many goal states
- How to efficiently check if a node appears in the other search tree?

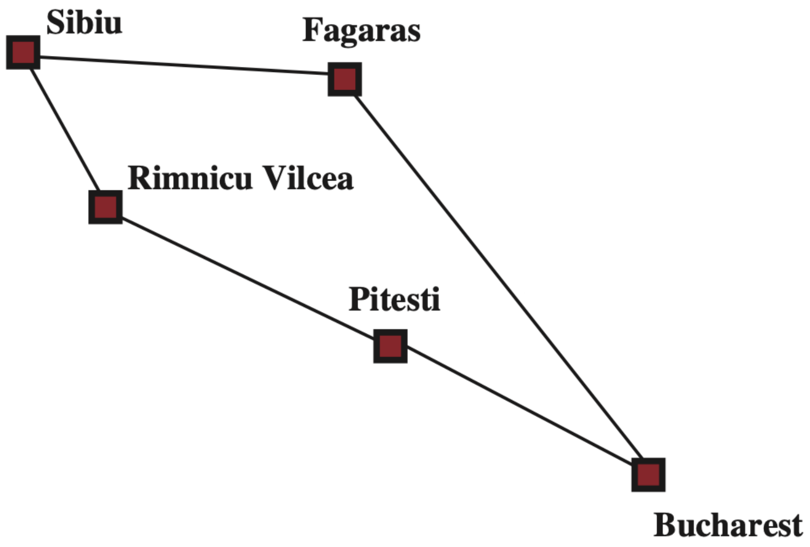
Outline

- Problem-solving agents
- Search algorithms
- Uninformed search algorithms
    - Breadth-first Search (BFS)
    - Uniform-cost search
    - Depth-first Search (DFS)
- Variants of uninformed search algorithms
    - Depth-limited search
    - Iterative deepening search
    - Bidirectional search
- **Dealing with repeated states**
- Informed search algorithms
    - Greedy best-first search
    - A* search
    - Heuristics
- Variants of A*

Dealing with Repeated States

- Remember states that are already visited
- Don't visit again





Tree Search

Keep track of visited nodes
(use hashtable/dictionary!)

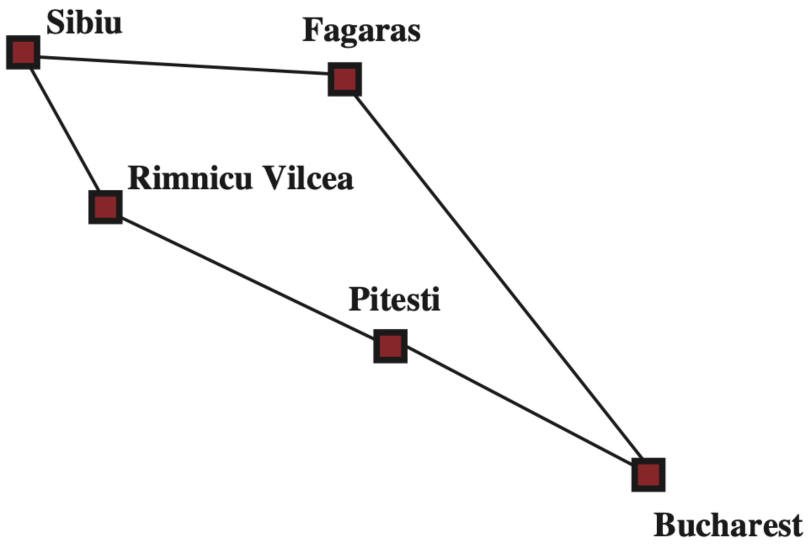
Graph Search

```
create frontier


insert initial state
while frontier is not empty:
    state = frontier.pop()
    for action in actions(state):
        next state = transition(state, action)

        if next state is goal: return solution
        frontier.add(next state)

return failure
```

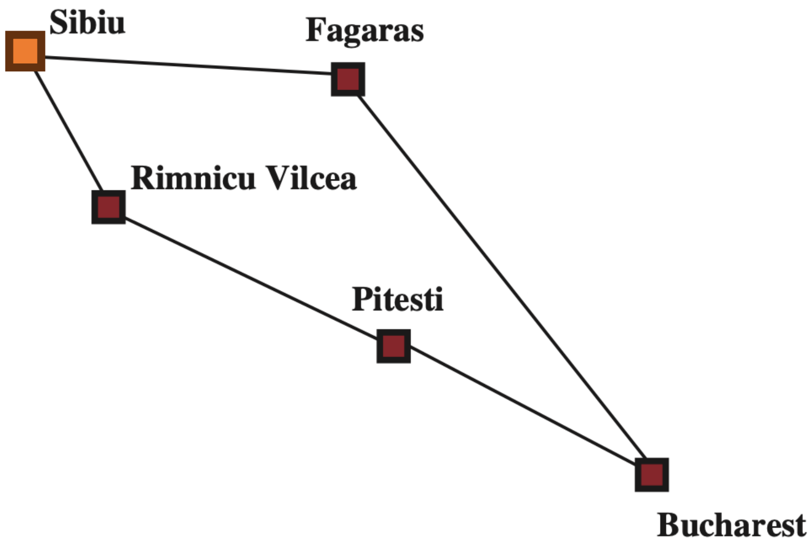
```
create frontier
create visited
insert initial state to queue and visited
while frontier is not empty:
    state = frontier.pop()
    for action in actions(state):
        next state = transition(state, action)
        if next state in visited: continue
        if next state is goal: return solution
        frontier.add(next state)
        visited.add(next state)
return failure
```

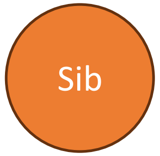
BFS with Graph Search

```
create frontier : queue
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
       if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```



BFS with Graph Search






```
create frontier : queue
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
       if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```

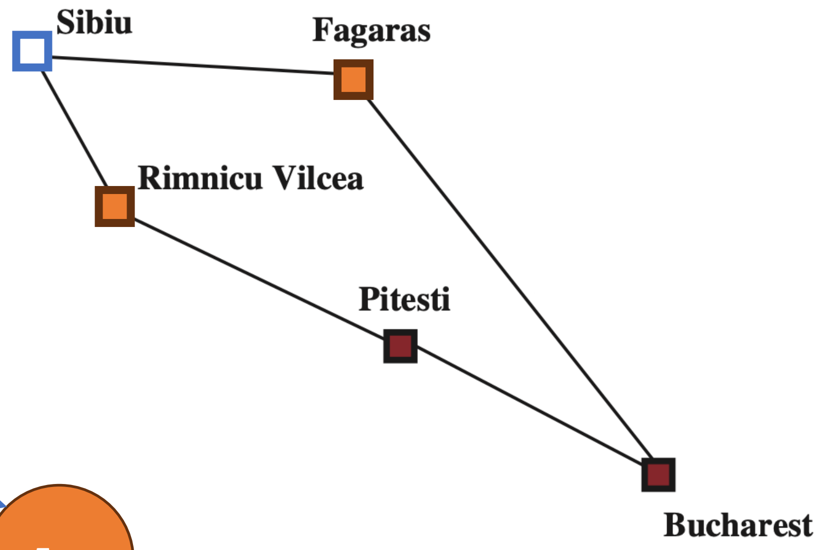
Queue:

Sib

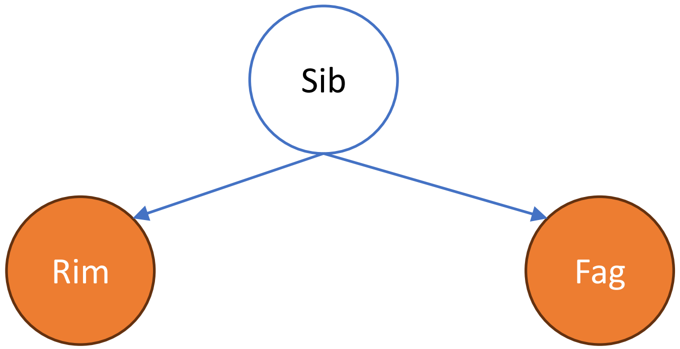
Visited:

Sib

BFS with Graph Search



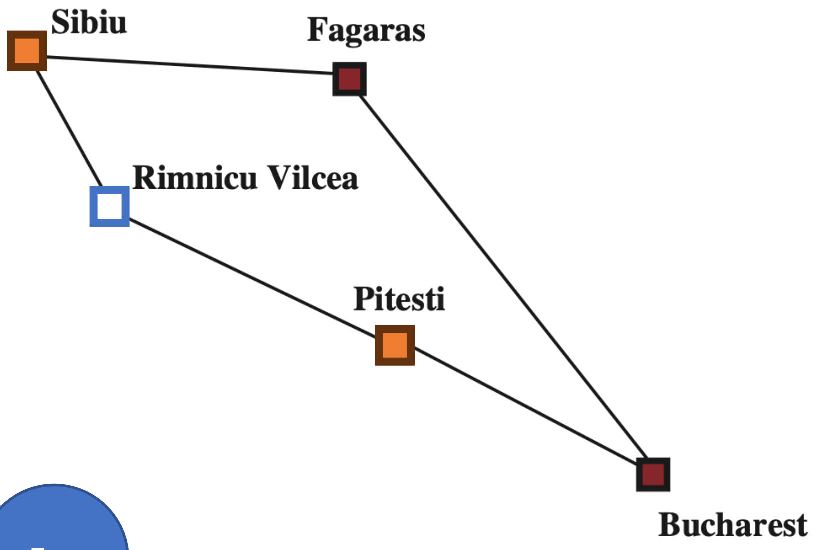

```
create frontier : queue
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
       if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```



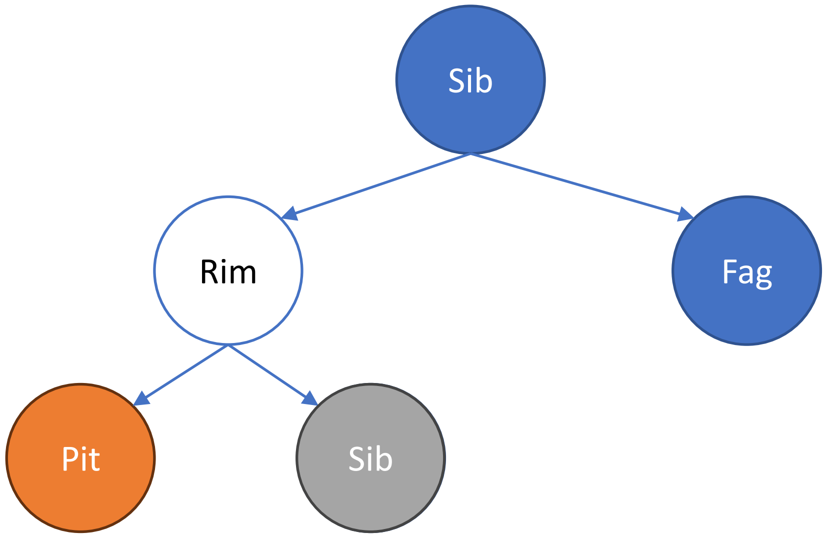
Queue:

Visited:

BFS with Graph Search



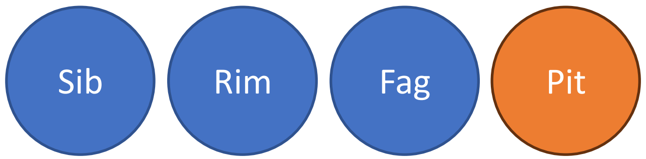

```
create frontier : queue
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
      if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```



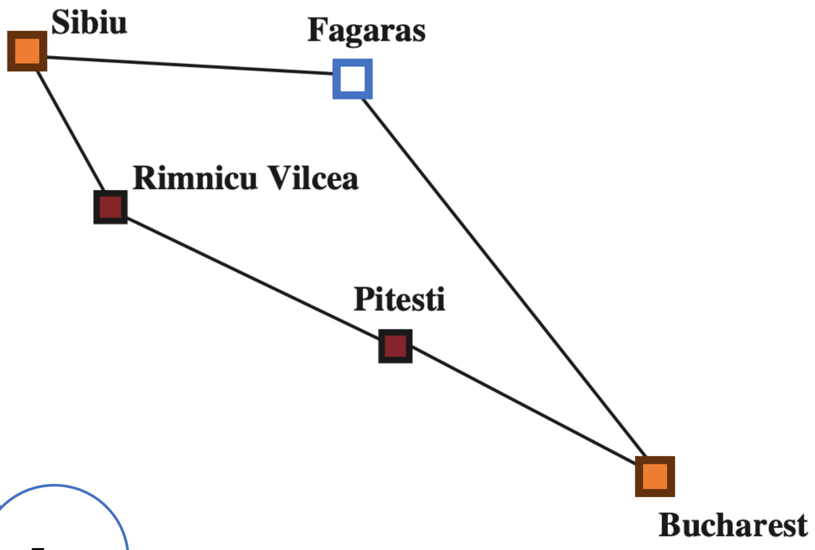
Visited!
(not added to the queue)

Queue:

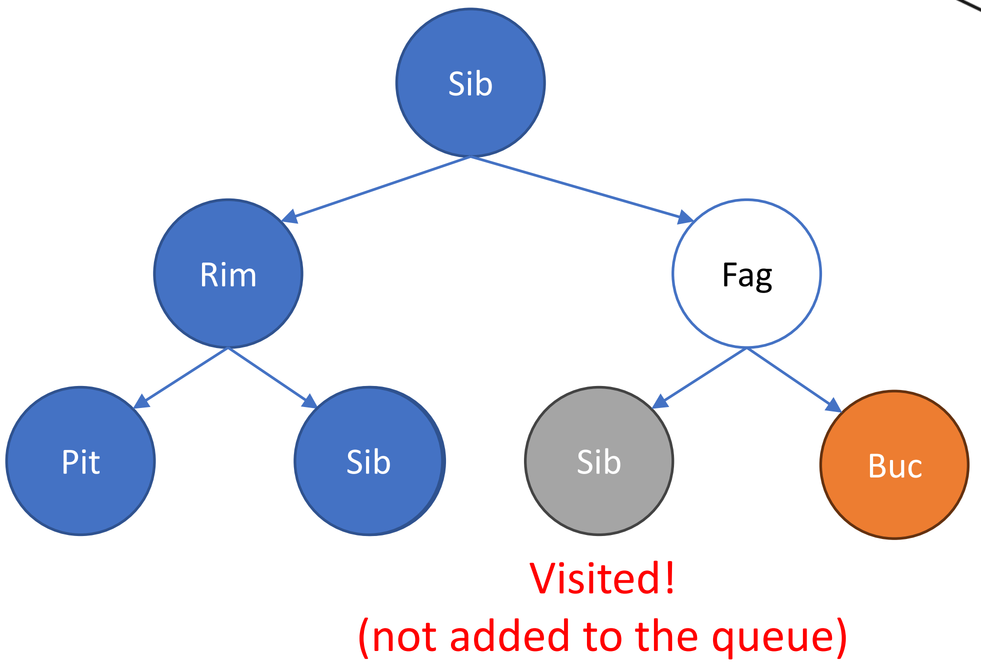
Visited:



BFS with Graph Search



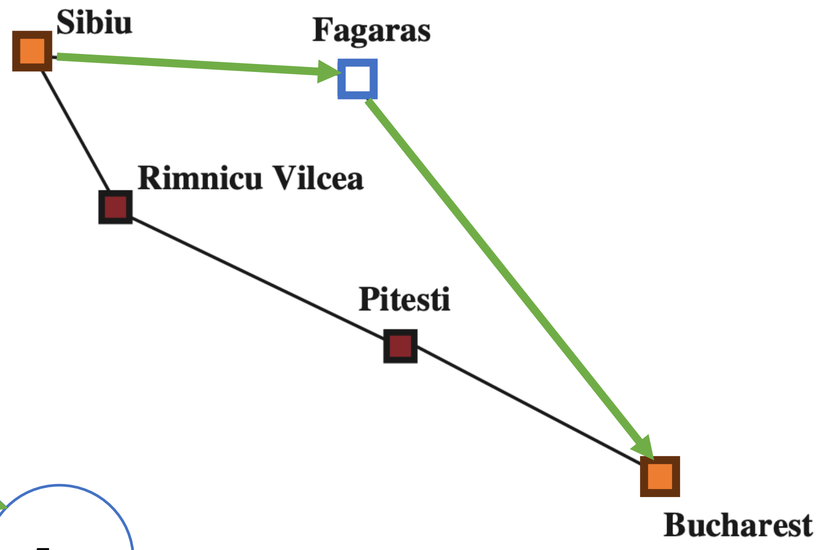

```
create frontier : queue
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
       if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```



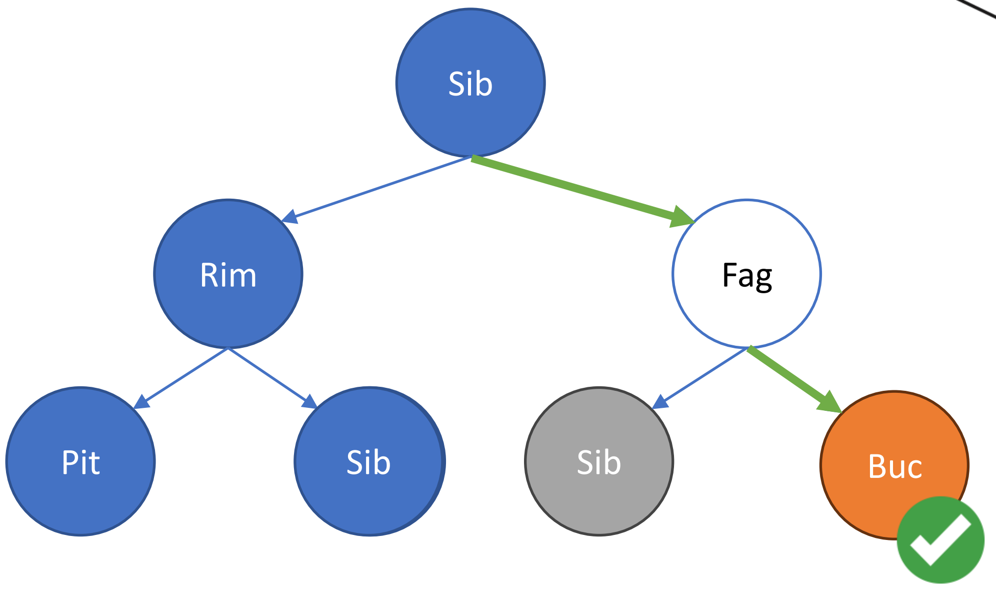
Queue:

Visited:

BFS with Graph Search




```
create frontier : queue
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
       if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```

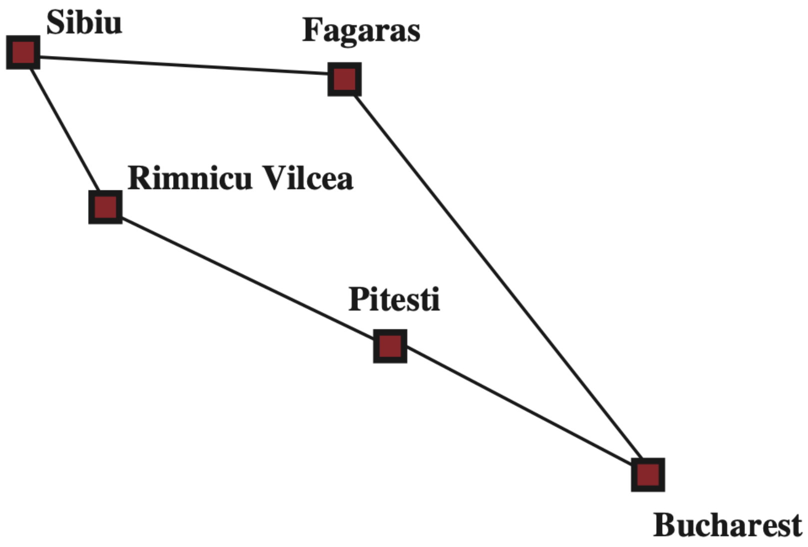


Queue:

Visited:

DFS with Graph Search

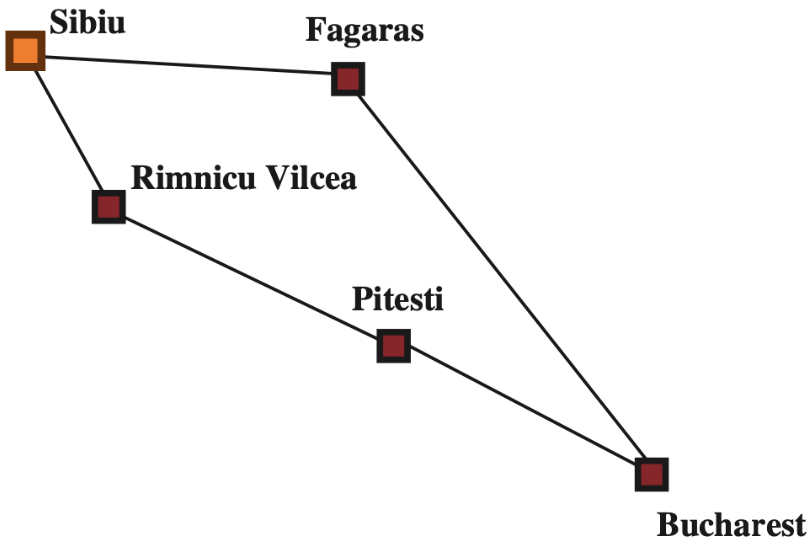
```
create frontier : stack
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
       if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```



DFS with Graph Search

```
create frontier : stack
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
       if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```

Sib



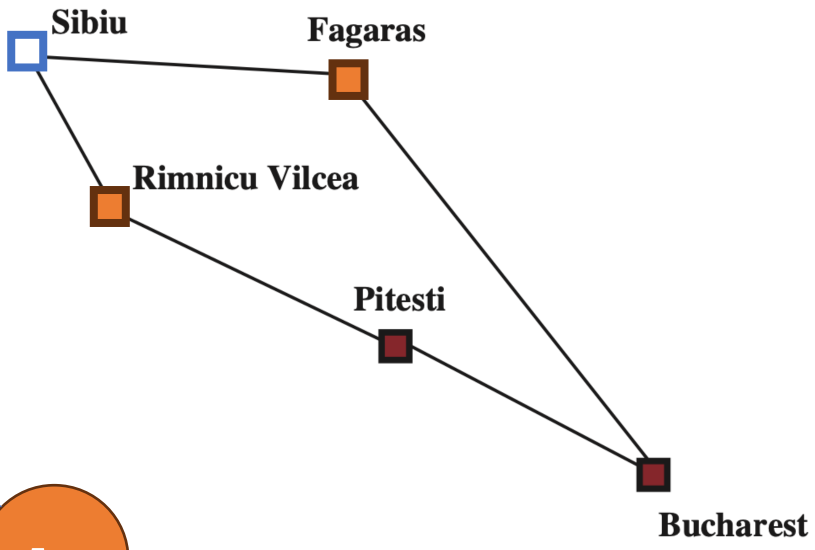
Stack:

Sib

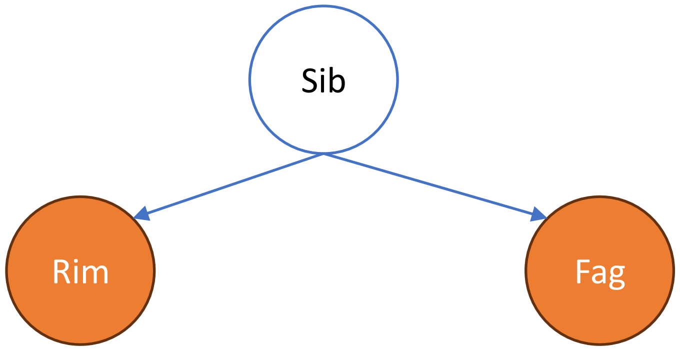
Visited:

Sib

DFS with Graph Search



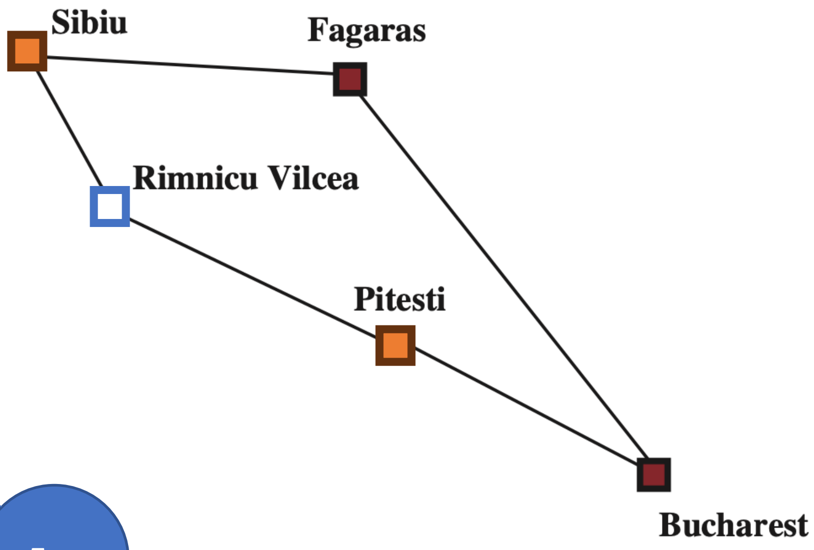

```
create frontier : stack
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
       if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```



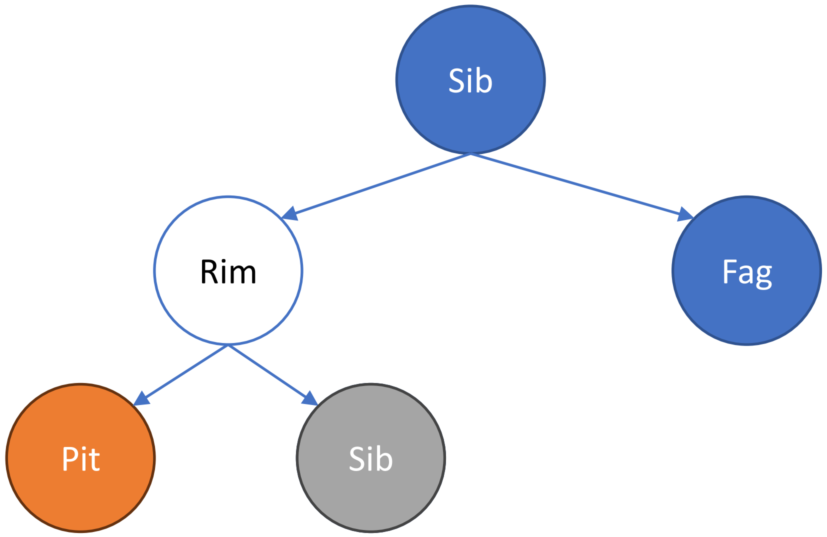
Stack:

Visited:

DFS with Graph Search

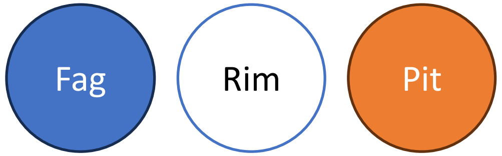



```
create frontier : stack
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
      if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```

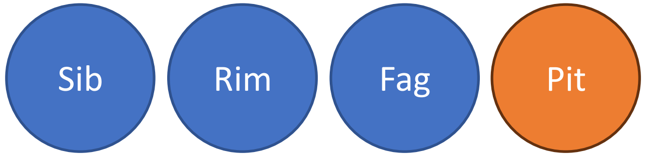


Visited!
(not added to the stack)

Stack:

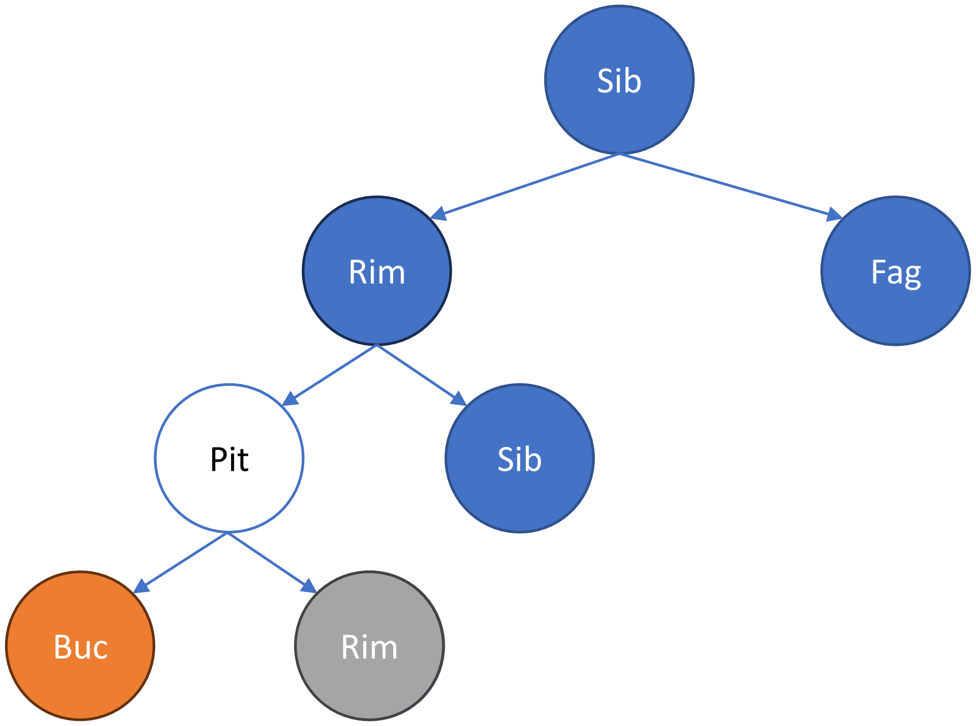


Visited:

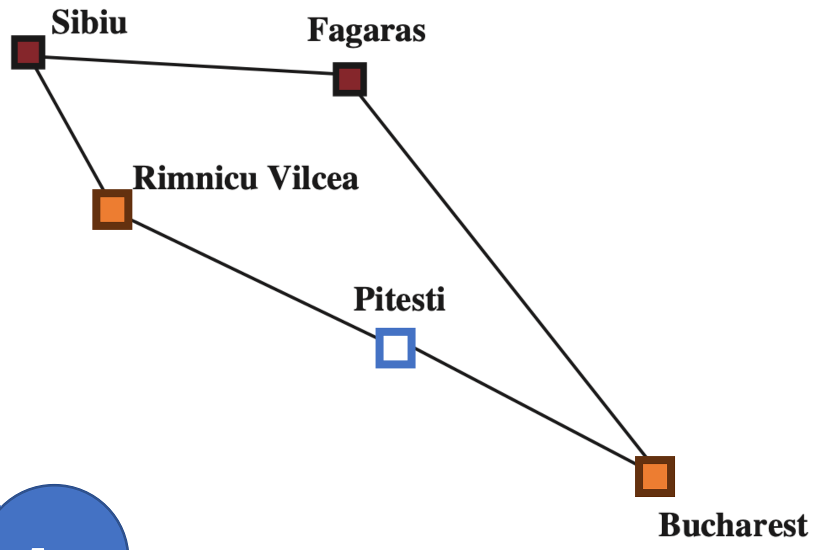


DFS with Graph Search

```
create frontier : stack
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
      if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```



Visited!
(not added to the stack)

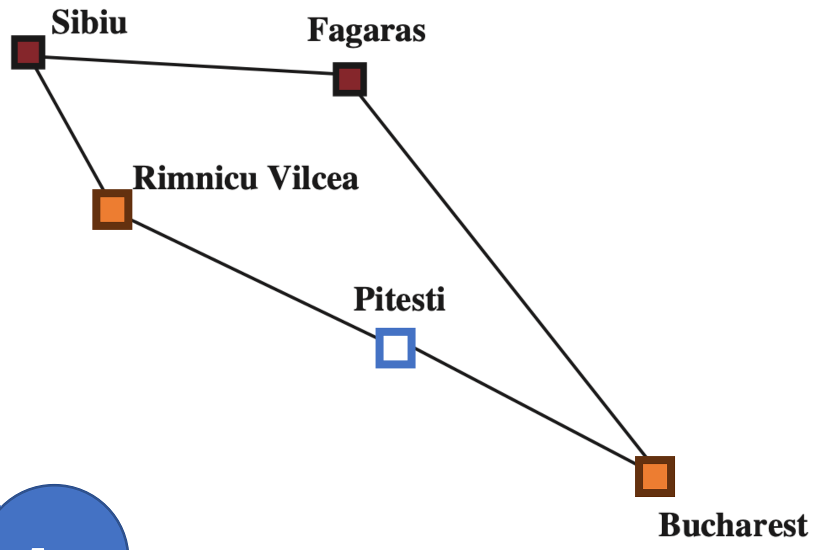


Stack:

Fag Pit Buc

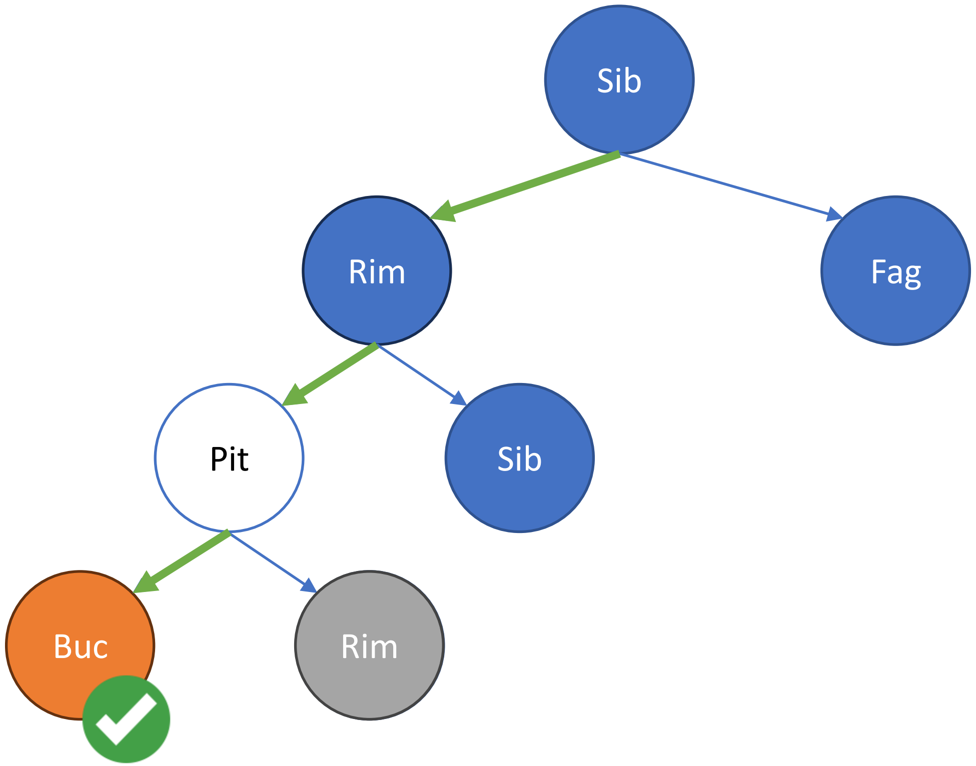
Visited:

Sib Rim Fag Pit Buc

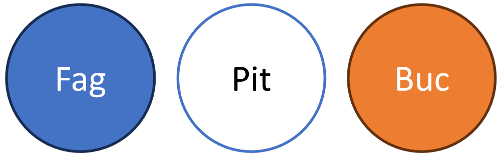


DFS with Graph Search

```
create frontier : stack
create visited
insert initial state to queue and visited
while frontier is not empty:
   state = frontier.pop()
   for action in actions(state):
      next state = transition(state, action)
       if next state in visited: continue
      if next state is goal: return solution
      frontier.add(next state)
      visited.add(next state)
return failure
```



Stack:



Visited:

Choosing a Search Strategy

Depends on the problem:

- Number of goal states
- Distribution of goal states in search tree
- Finite/infinite branching factor/depth
- Repeated states
- Need for optimality?
- Need to know if there is no solution?

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, …, N
  - Bidirectional search – search from the **start** and the **goal**, meet in the **middle**
- Dealing with repeated states
  - Graph search – don't visit nodes that are already **visited**

Outline

- Problem-solving agents
- Search algorithms
- Uninformed search algorithms
  - Breadth-first Search (BFS)
  - Uniform-cost search
  - Depth-first Search (DFS)
- Variants of uninformed search algorithms
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search
- Dealing with repeated states
- **Informed search algorithms**
  - Greedy best-first search
  - A* search
  - Heuristics
- Variants of A*

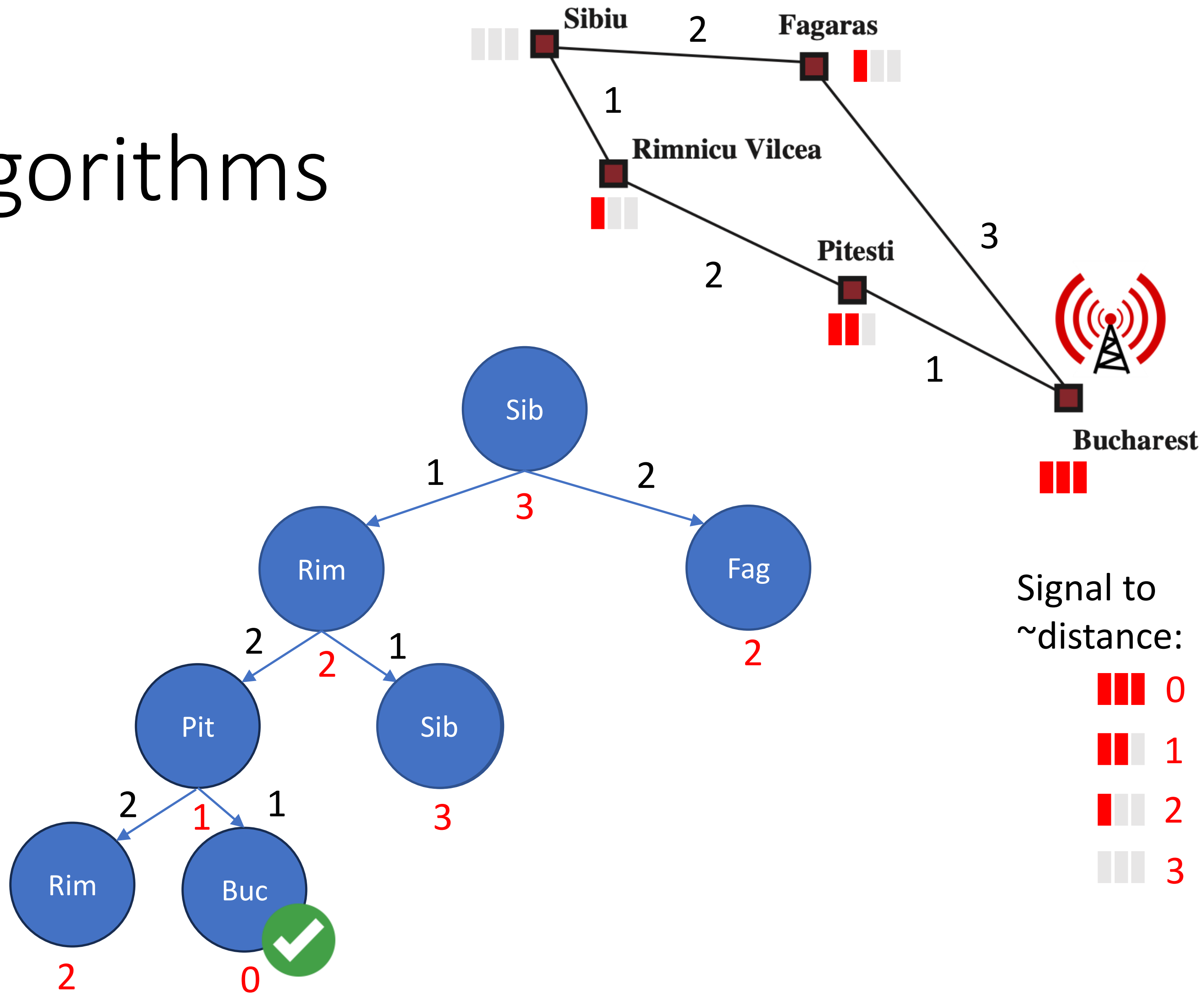
Informed Search Algorithms

Uninformed search:

Search blindly

Informed search:

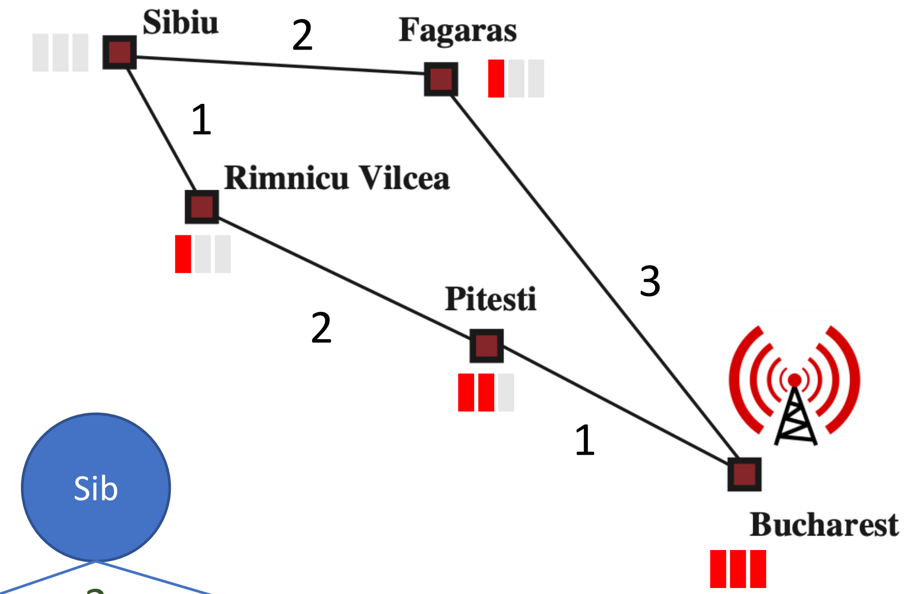
Use domain information to <u>guide</u> the search

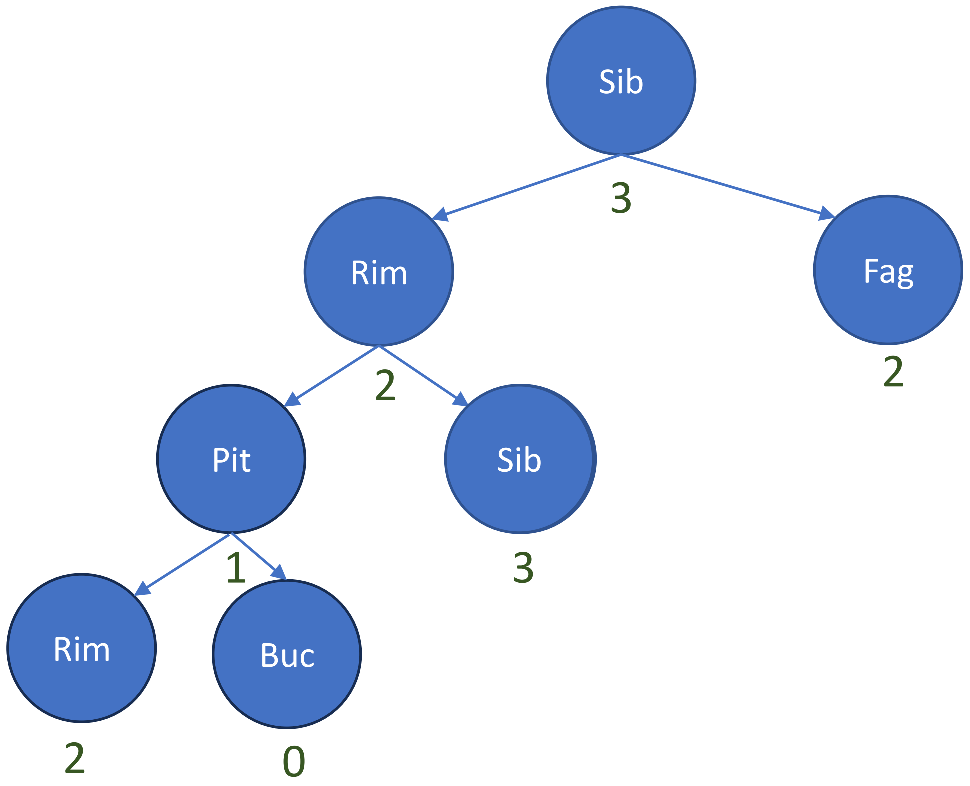


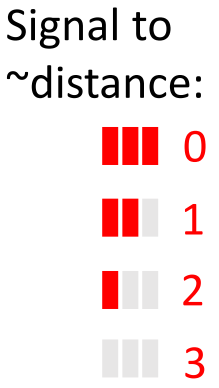
Best-first Search

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

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

Evaluation function f(n):
estimate the "goodness" of a state







Special cases:

- Greedy best-first search
- A* search

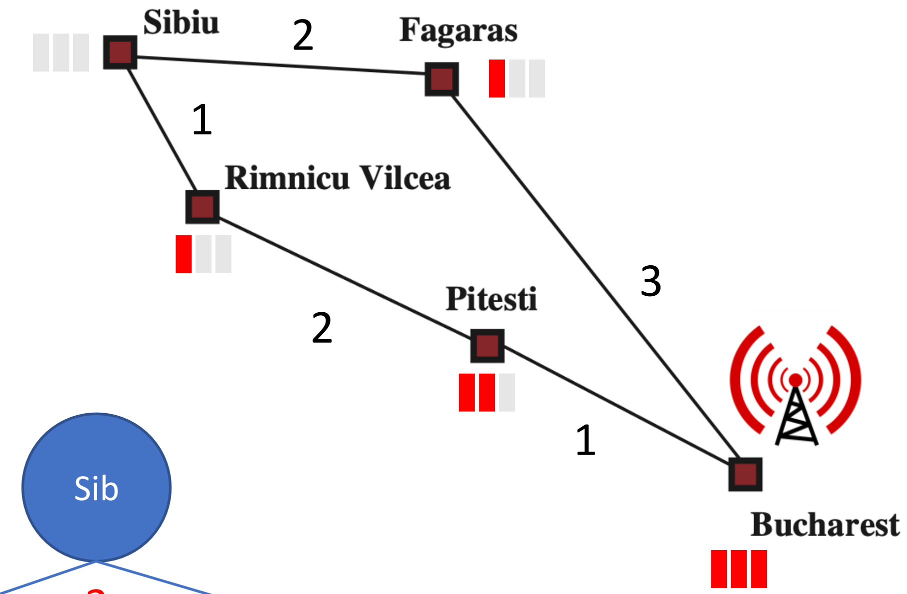
Greedy Best-first Search

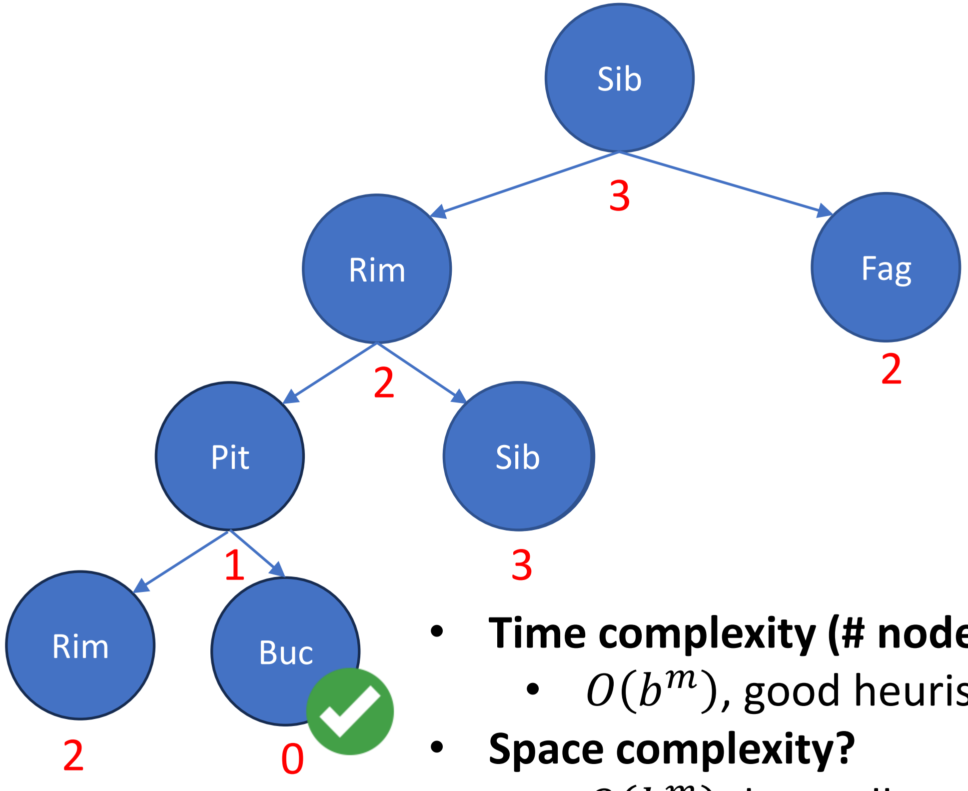
```
create frontier : priority queue f(n)

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

Evaluation function f(n):
estimate the "goodness" of a state

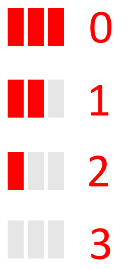
f(n) = h(n)

Heuristic: estimated cost from n to goal



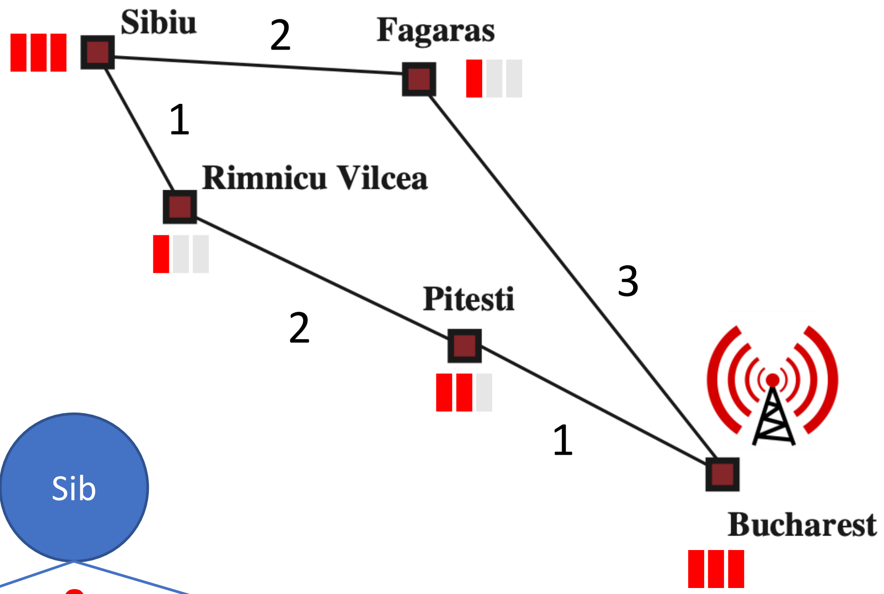


Signal to ~distance:



- **Time complexity (# nodes expanded)?**
  - $O(b^m)$, good heuristic gives improvement
- **Space complexity?**
  - $O(b^m)$, keep all nodes in memory
- **Complete?**

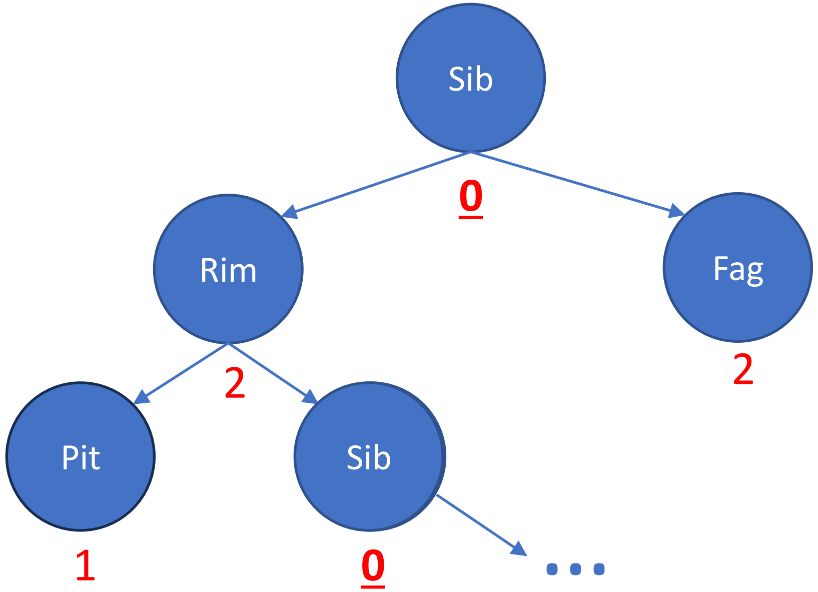
Greedy Best-first Search



Anomaly!

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

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

Evaluation function f(n): estimate the "goodness" of a state



Signal to ~distance:
0
1
2
3

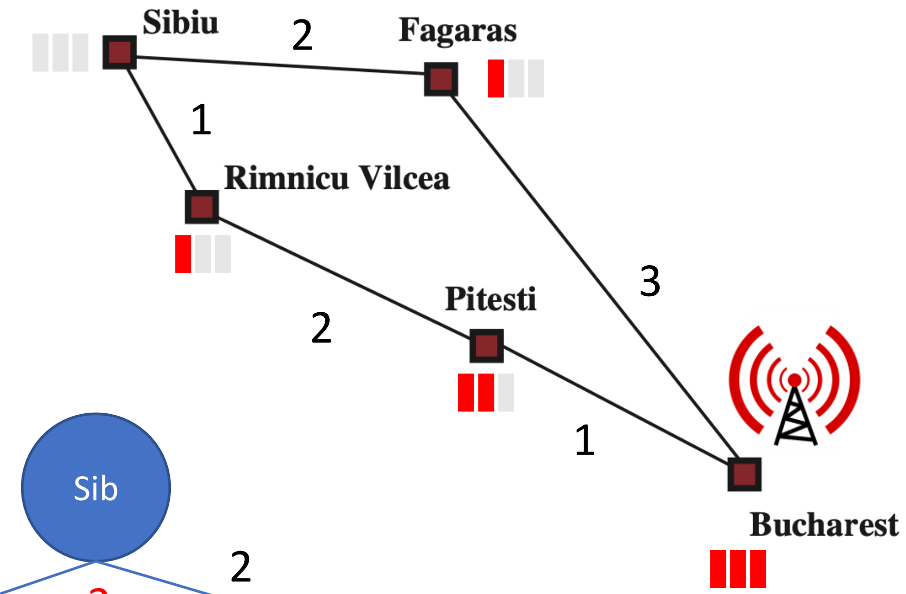
$f(n) = h(n)$

Heuristic: estimated cost from n to goal

Doesn't consider the cost so far!

- **Time complexity (# nodes expanded)?**
  - $O(b^m)$, good heuristic gives improvement
- **Space complexity?**
  - $O(b^m)$, keep all nodes in memory
- **Complete?** No
- **Optimal?** No

A* Search



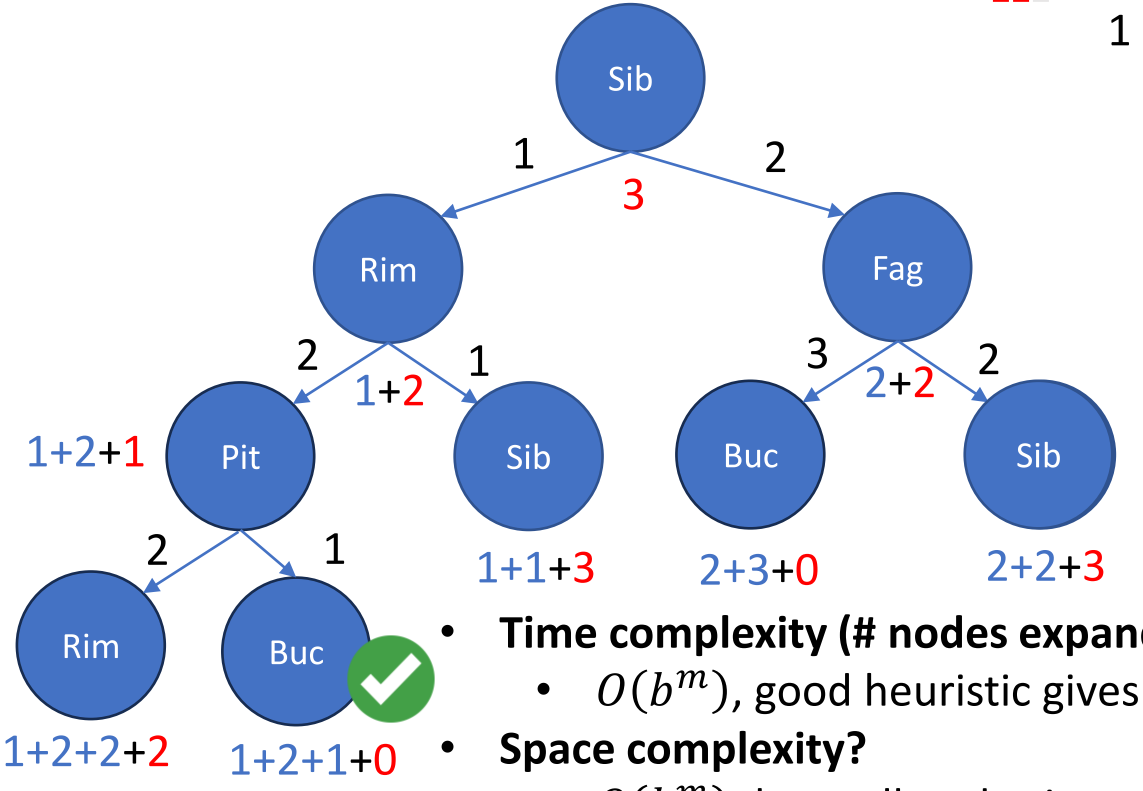

```
create frontier : priority queue f(n)

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

Evaluation function f(n): estimate the "goodness" of a state

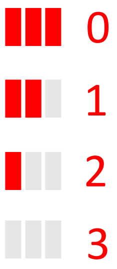
f(n) = g(n) + h(n)

Cost so far to reach n

Heuristic: estimated cost from n to goal



Signal to ~distance:



- **Time complexity (# nodes expanded)?**
  - $O(b^m)$, good heuristic gives improvement
- **Space complexity?**
  - $O(b^m)$, keep all nodes in memory
- **Complete?** Yes
- **Optimal?** Yes*

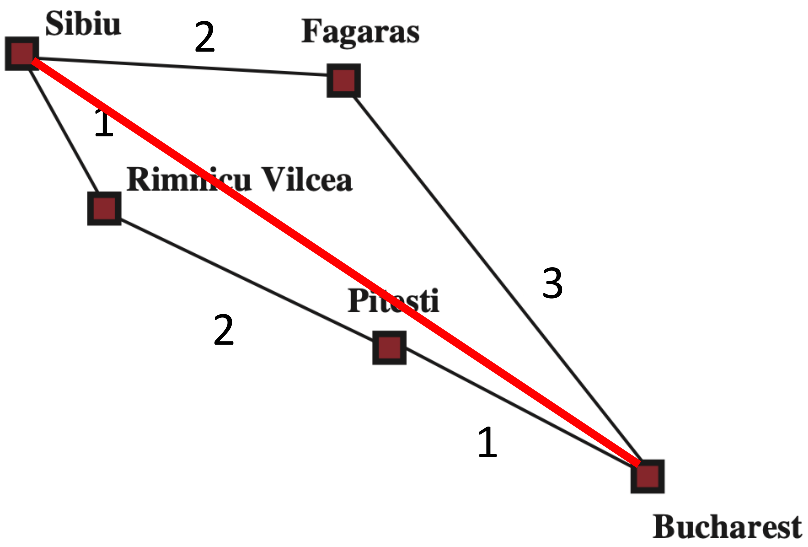
Admissible Heuristics

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

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

Theorem: if h(n) is admissible, A* using **tree search** is optimal

$h_{SLD}(Sibiu)$



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

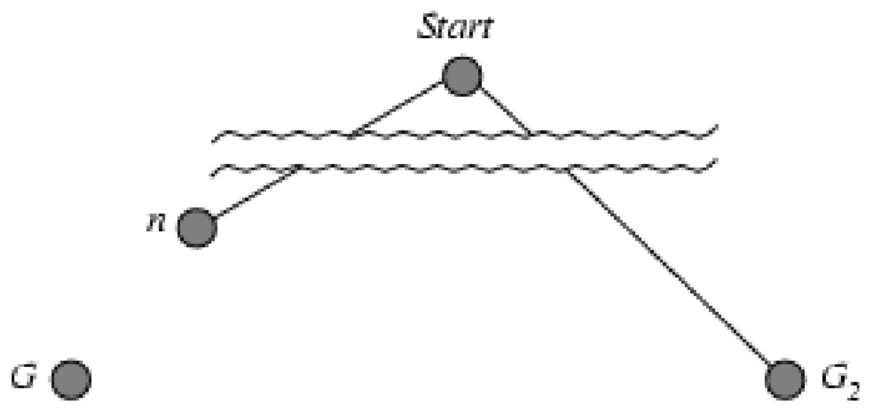
Admissible Heuristics: Optimality

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

$f(G) = g(G) + h(G) = g(G) + 0$
$f(G_2) = g(G_2) + h(G_2) = g(G_2) + 0$
$g(G_2) > g(G)$ since G_2 is suboptimal
$f(G_2) > f(G)$

$h(n) \leq h^*(n)$ since h is admissible
$f(n) = g(n) + h(n) \leq g(n) + h^*(n) = f(G)$

$f(n) \leq f(G) < f(G_2)$, G_2 will never be expanded



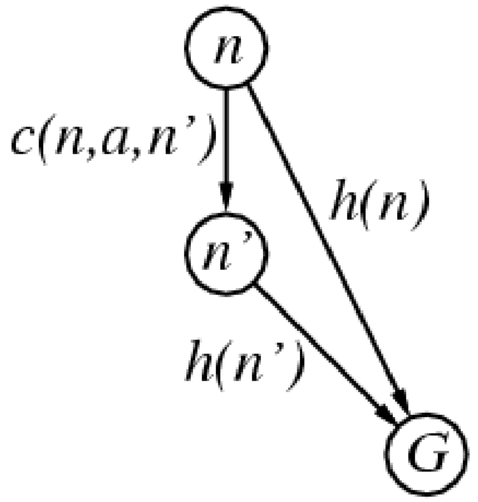
Consistent Heuristics

A heuristic $h(n)$ is **consistent** if for every node n, every successor n' of n generated by any action a, $h(n) \leq c(n,a,n') + h(n')$

If h is consistent, we have

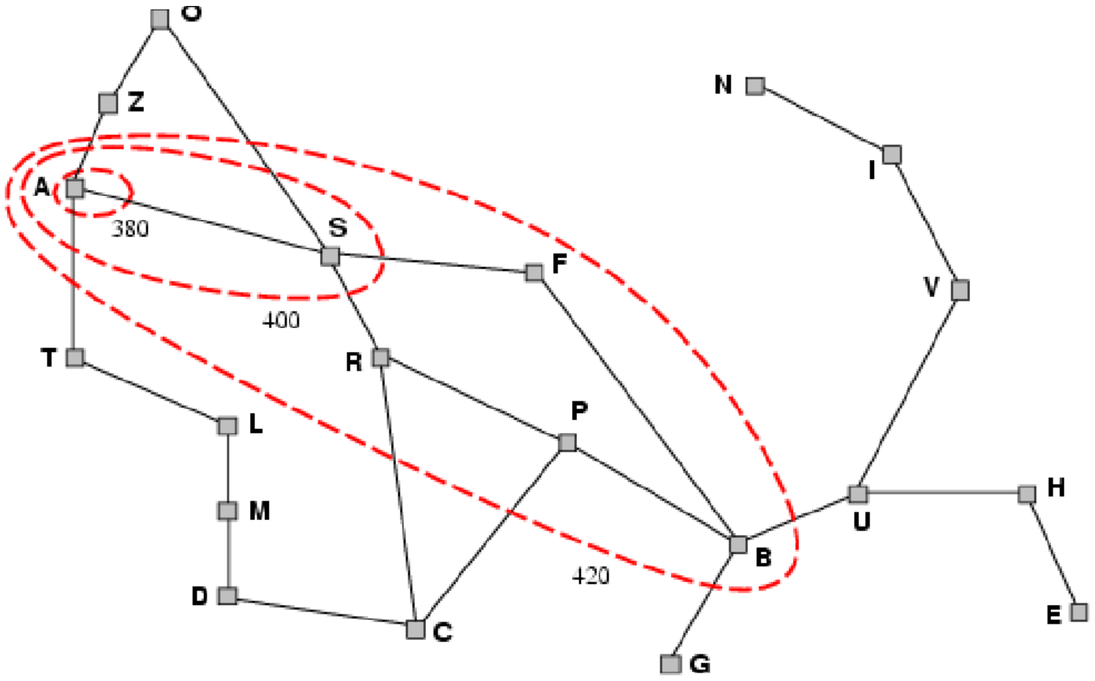
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) = f(n) \end{aligned}$$

i.e., f(n) is **non-decreasing** along any path



Theorem: If h(n) is consistent, A* using **graph search** is optimal

Consistent Heuristics: Optimality



A * expands nodes in order of increasing f-cost

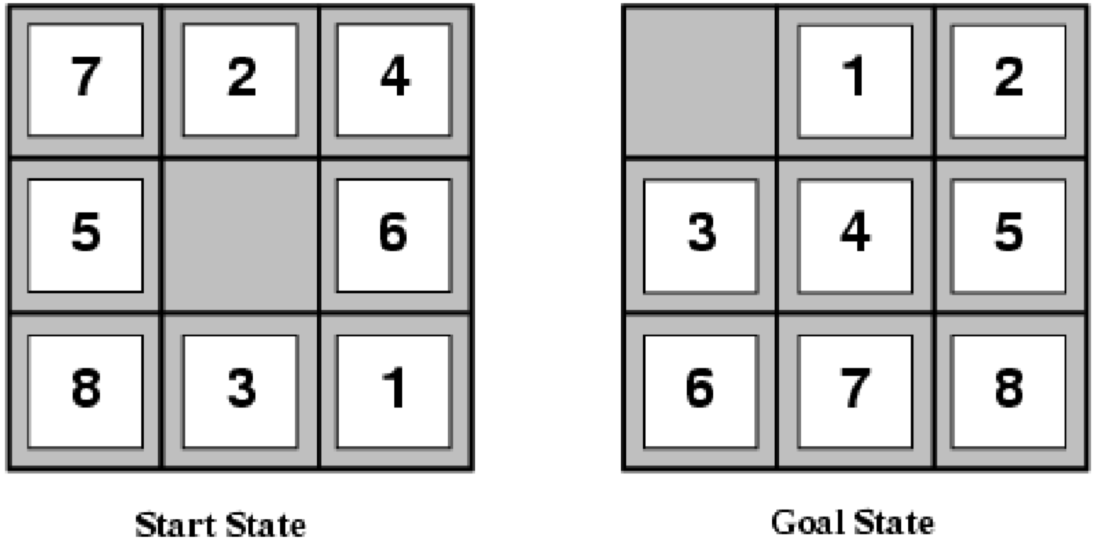
Gradually adds "f-contours" of nodes

Contour i has all nodes with $f = f^{(i)}$, where $f^{(i)} < f^{(i+1)}$

Dominance

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

h_2 is better for search.



Heuristics

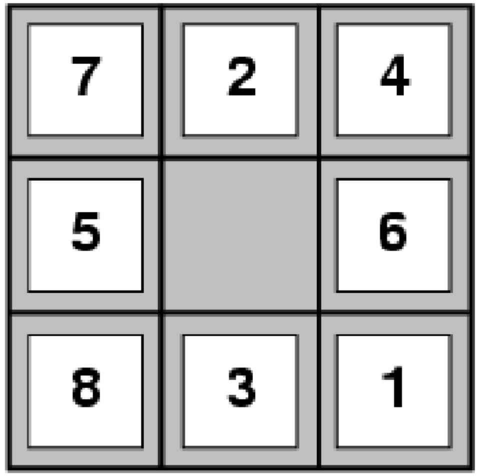
- h_1 number of misplaces tiles
- h_2 total Manhattan distance

h_2 dominates h_1

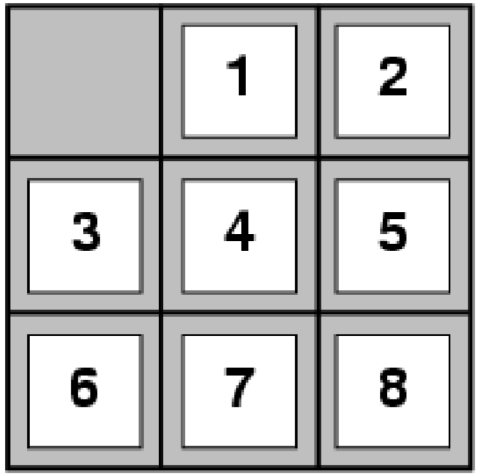
If each tile is at most one distance away from the goal, then h_2 = h_1, otherwise h_2 > h_1

“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.



Start State



Goal State

Original:
A tile can only move to adjacent blank

Relaxations:
- Each tile can move anywhere
    - h_1 number of misplaces tiles
- Each tile can move to any adjacent square
    - h_2 total Manhattan distance

Outline

- Problem-solving agents
- Search algorithms
- Uninformed search algorithms
    - Breadth-first Search (BFS)
    - Uniform-cost search
    - Depth-first Search (DFS)
- Variants of uninformed search algorithms
    - Depth-limited search
    - Iterative deepening search
    - Bidirectional search
- Dealing with repeated states
- Informed search algorithms
    - Greedy best-first search
    - A* search
    - Heuristics
- **Variants of A***

Variants of A*

- Iterative Deepening A* (IDA*)
  - Use iterative deepening search
  - Cutoff using f-cost [$f(n) = g(n) + h(n)$] instead of depth
- Simplified Memory-bounded A* (SMA*)
  - Drop the nodes with worst f-cost if memory is full

Summary: Informed Search Algorithms

- Informed search: <u>guide</u> search with domain information
- Best-first search
  - Greedy best-first search
    - f(n) = h(n), **heuristic** estimate of cost from n to goal
  - A* search
    - f(n) = g(n) + h(n), cost so far + heuristic
- Heuristics
  - Admissible: $h(n) \le h^*(n)$
  - Consistent: $h(n) \le c(n,a,n') + h(n')$
  - Dominant: if $h1(n) \le h2(n)$, h2 dominant
- Creating admissible heuristic: **true cost** of the **relaxed problem**
- A* variants: IDA*, SMA*
  - Idea: prune to save memory

Summary

- Problem-solving agents
- Uninformed search algorithms
  - Breadth-first Search (BFS) – layer by layer
  - Uniform-cost search – Djikstra
  - Depth-first Search (DFS) – go deep first
- Variants of uninformed search algorithms
  - Depth-limited search – set max depth
  - Iterative deepening search – try DLS with depth limit 0, …, N
  - Bidirectional search – combine forward and backward search
- Dealing with repeated states – visit state only once
- Informed search algorithms
  - Greedy best-first search – f(n) = h(n)
  - A* search – f(n) = g(n) + h(n)
  - Heuristics: admissibility, consistency, dominance
- Variants of A*: IDA*, SMA*

Coming Up Next Week

- Local search
  - Hill climbing
  - Simulated annealing
  - Beam search
  - Genetic algorithms
- Adversarial search
  - Games vs search problems
  - Minimax
  - Alpha-beta pruning

To Do

- **Lecture Training 2**
  - +100 Free EXP
  - +50 Early bird bonus
- **Problem Set 0**
  - Due Saturday, 26th August (Tomorrow)
- **Tutorial Swaps**
  - Due Sunday, 27th August