香港浸會大學
HONG KONG BAPTIST UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE
計算機科學系

COMP7015 Artificial Intelligence

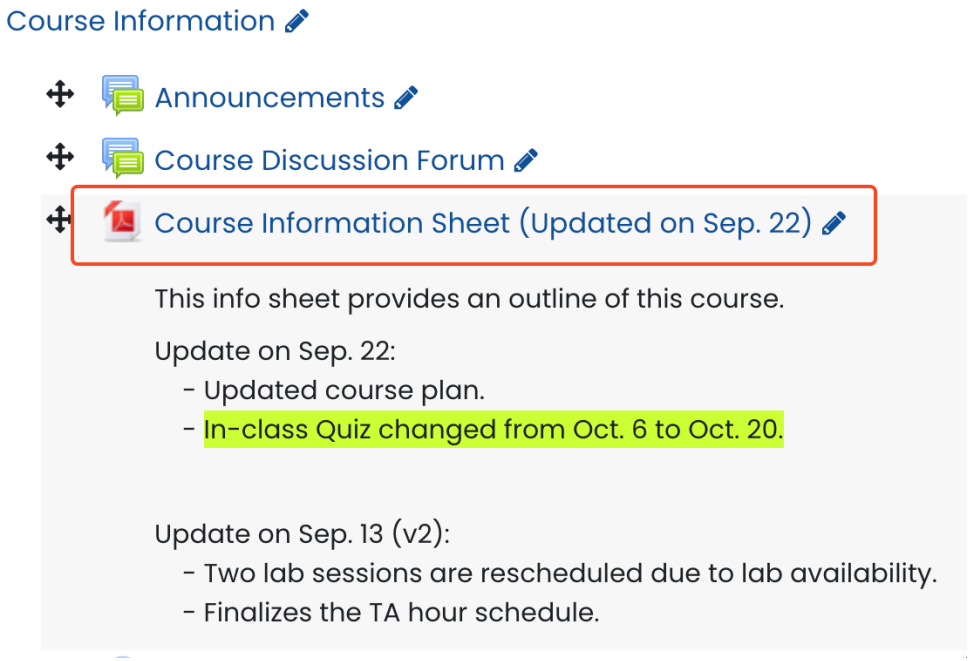
Lecture 3: Search II

Instructor: Dr. Kejing Yin

September 22, 2022

Logistics

- Major update of info sheet: **New Quiz Time**



- LAB SESSIONS (Due to lab availability, only Saturday sessions can be scheduled):
  - Lab: FSC 8/F (Fong Shu Chuen Library, Ho Sin Hang Campus)
  - Five lab sessions will be delivered on Sep. 17, Sep. 24, Oct. 15, ~~Oct. 22~~, ~~Oct. 29~~, Nov. 5, and Nov. 12.
  - Lab sessions will be divided to two identical sections due to limited PC availability.
    - Section 1: 2:00 pm – 3:45 pm (Capacity: 80 PCs);
    - Section 2: 4:00 pm – 5:45 pm (Capacity: 80 PCs).
- TA HOURS (Q&A for programming):
  - Mr. TANG Zhenheng: 2:00 pm – 4:00 pm on Sep. 20, Sep. 27, Oct. 18, Nov. 8, and Nov. 15.
  - Mr. LI Ruiqi: 2:00 pm – 4:00 pm on Oct. 11, Oct. 25, Nov. 1, and Nov. 22.
  - Zoom link will be sent out before the TA hours start.

Week 7	**Lecture 7: Generalization and Model Selection (Oct. 20)** [4, Chap. 5] In-class quiz (1.5h; Oct. 20) Release of programming assignment 1 *(Due: 23:59 pm, Nov. 4)*

Logistics

- Lab 1: Solving problems using search on Sep. 24 (Sat.) at FSC 8/F
  Lab materials will be available on Sep. 23 (Fri.) in Moodle

- Next Office Hour: Sep. 26 (Mon.)
- Next TA Hour: Sep. 27 (Tue.)

- **In-class Quiz on Oct. 20 (mark your calendar)**

Logistics

- Assignments: more than just assessments.

- Written Assignment 1 will be out at 9:30 pm (Due: 23:59 pm, Oct. 5)
- Late policy: -10% for each 12 hours late.

- Discussions on course materials are encouraged. However, you must write the solutions to the assignments and projects on your own and understand them fully.
- Lending and borrowing assignment are both subject to heavy penalty, including receiving zero scores, failing the course, and other disciplinary actions.

Recap: How do we formulate a search problem?

- possible states (*s*) that the environment can be in: *the state space.*
- An initial state and a set of goal states (or one goal state)

- The actions: ACTION(*s*) *a finite set of actions given a state s*
- A transition function: RESULT(s, *a*) *result of taking action a at state s*
- An action cost function:
  ACTION-COST(*s, a, s'*) *cost of applying action a at state s to reach state s'*
- A solution: a path from initial to goal state. *It is optimal if it has minimal cost.*

Recap: How do we formulate a search problem?

- **State**: agent loc . x dirt loc. = 2 x 2^2 = 8 reachable states
- **Initial state**: any state can be initial.
- **Actions**: {Left, Right, Suck}
- **Goal test**: both location A and B are clean
- **Action cost**: each step costs 1 (i.e., path cost = #actions in the path)
- **Transition function**

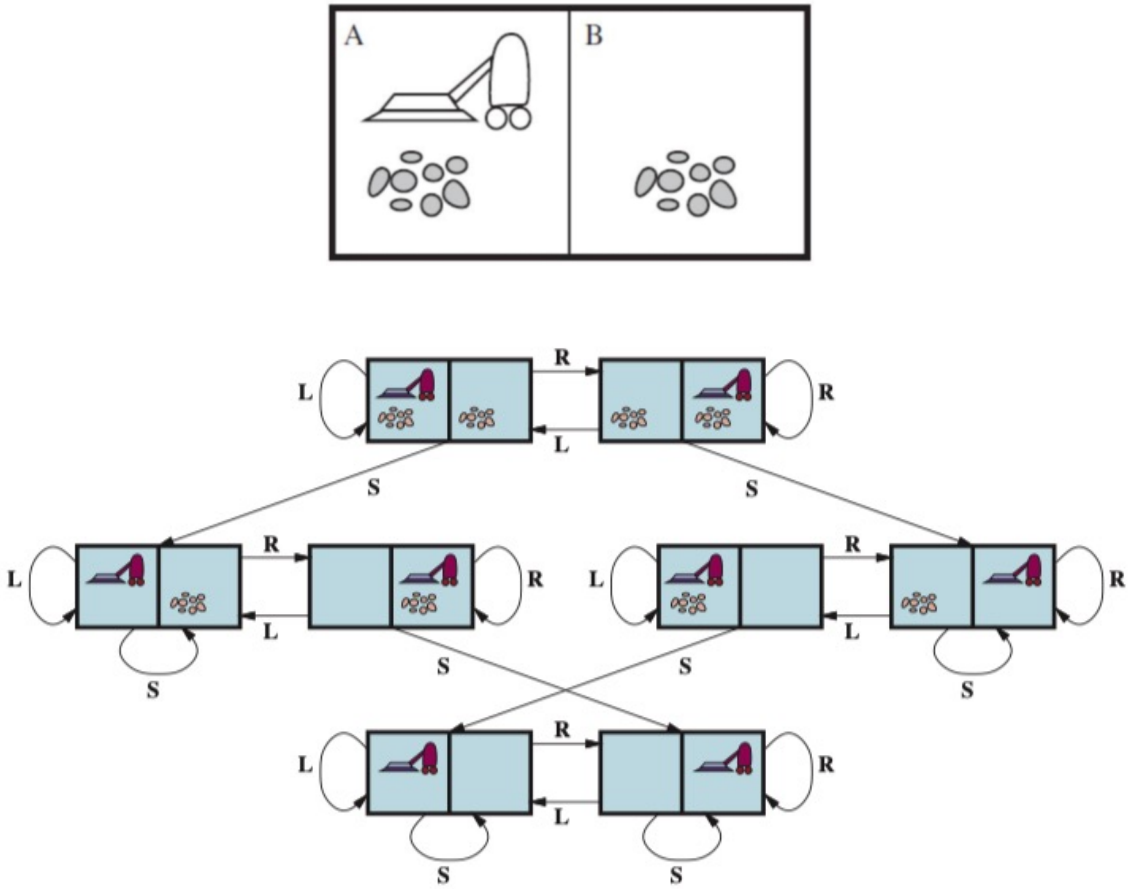


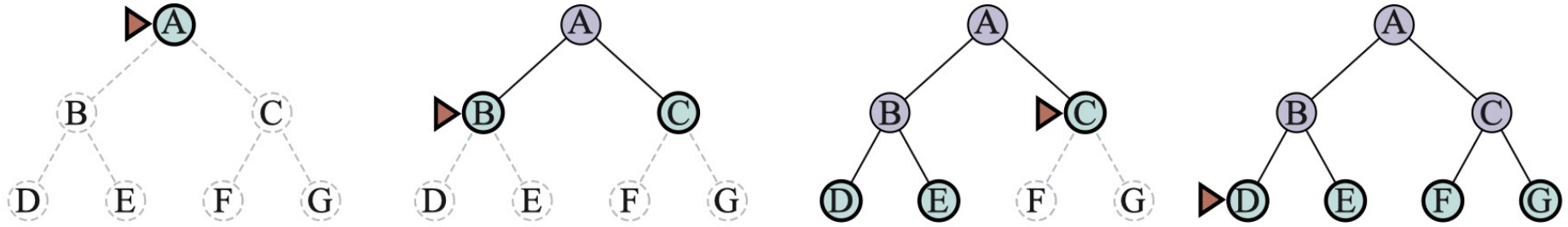
Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Road Map of Search Algorithms

- Uniformed search (blind search; no *clue about how close a state is to the goal state*)
  - ✓ Breadth-first Search (BFS)
  - Uniform-cost Search
  - Depth-first Search (DFS)
    - ✓ Graph search implementation
    - Tree-like search implementation
  - Depth-limited Search and Iterative deepening Search
- Informed search (heuristic search; we have some hints about the location of goals)
  - Greedy Search
  - A* Search
- Constraint satisfaction problems (CSPs)

Recap: Breadth-first Search (BFS)

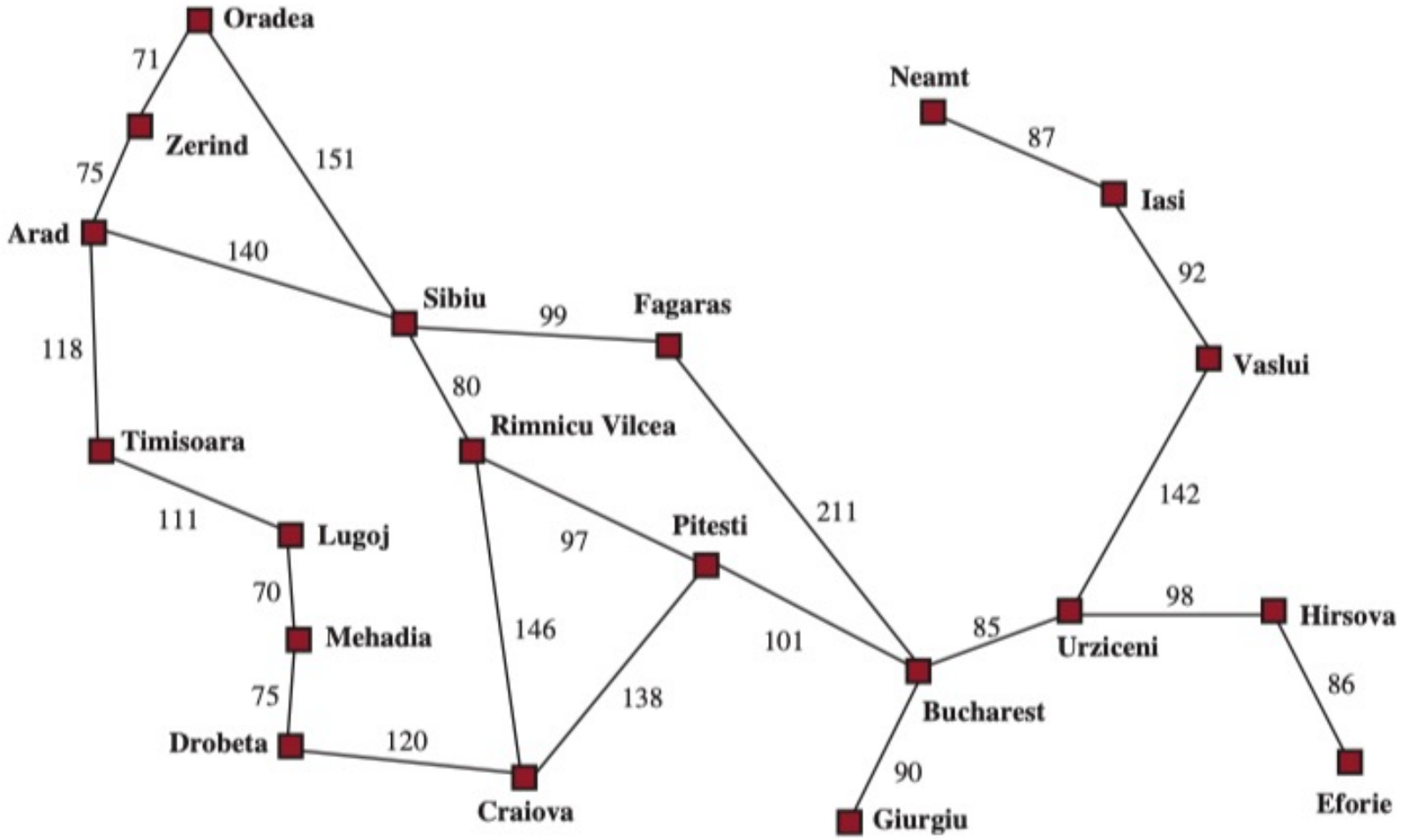
- Always expand the nodes at depth *d-1* before expanding nodes at depth *d*.
- Store the visited nodes (in purple) and frontiers (nodes to visit; in cyan).



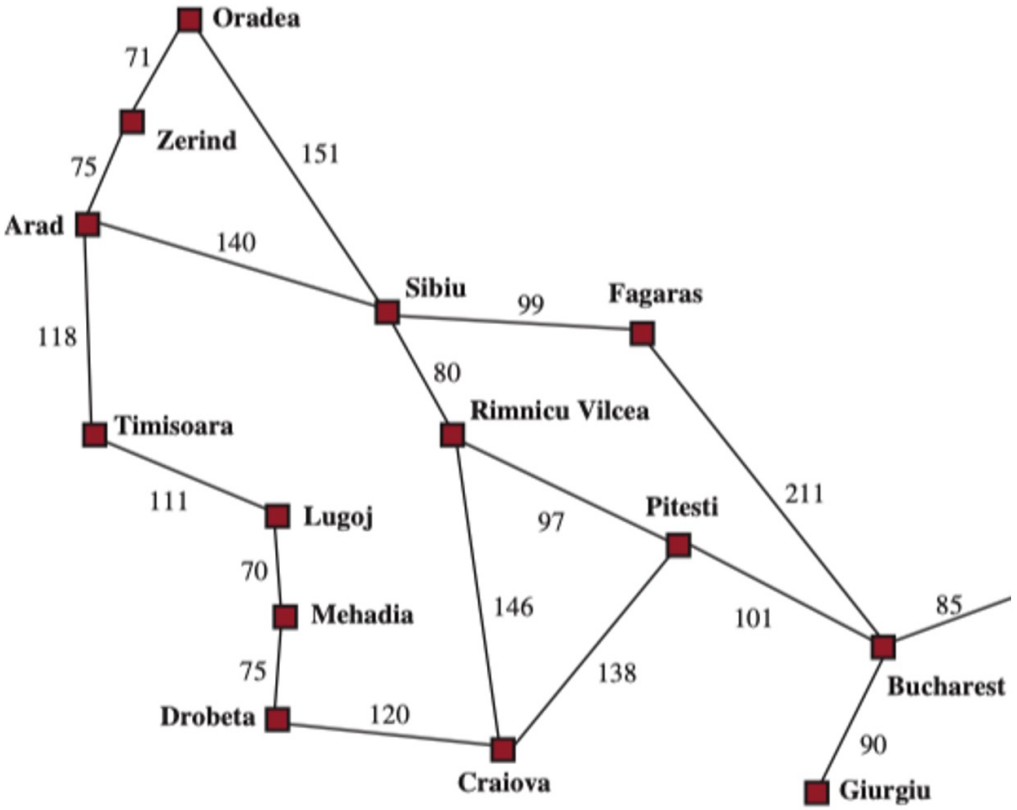
- Guaranteed to find the solution, if it exists.
- It always finds the shallowest goal state first (minimal number of actions)
  → *cost-optimal if all actions have the same cost.*

- $O(b^d)$ time and memory complexity.
  b: avg. number of successors of each node; *d*: depth of the goal state.

Exercise: Perform BFS and DFS to reach Bucharest from Arad



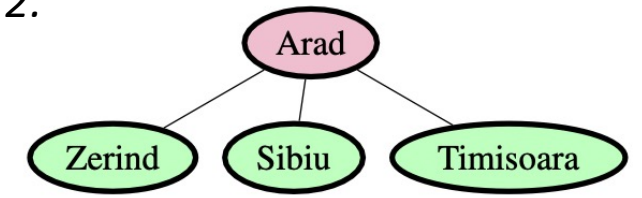
Sample Solution using BFS



Step 1:

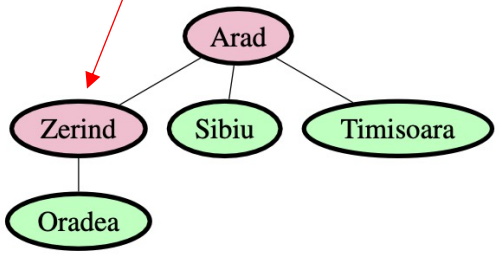

```
frontiers = [Arad]
visited = []
```

Step 2:



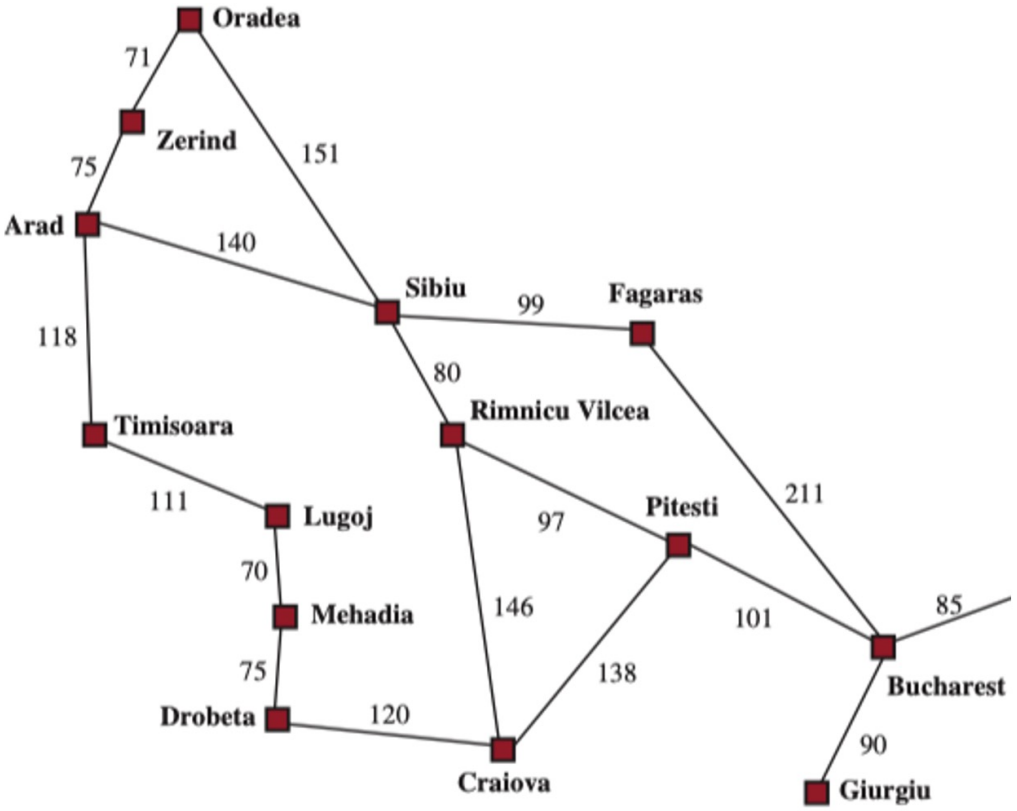

```
frontiers = [Zerind, Sibiu, Timisoara]
visited = [Arad]
```

Step 3:

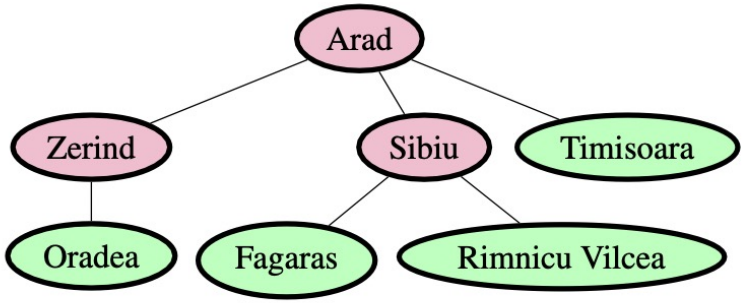



```
frontiers = [Sibiu, Timisoara, Oradea]
visited = [Arad, Zerind]
```

Sample Solution using BFS

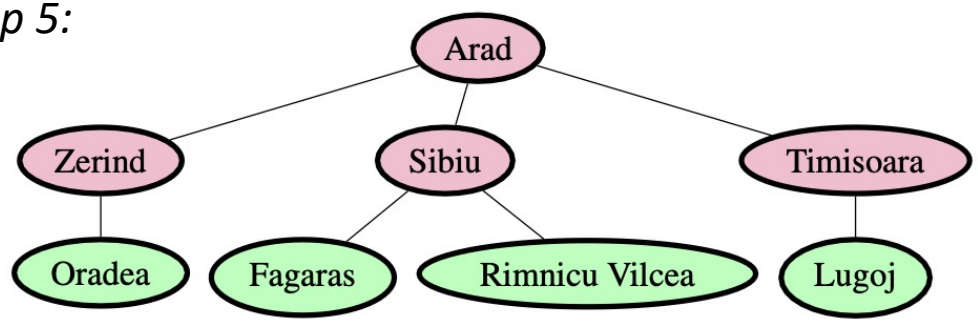


Step 4:



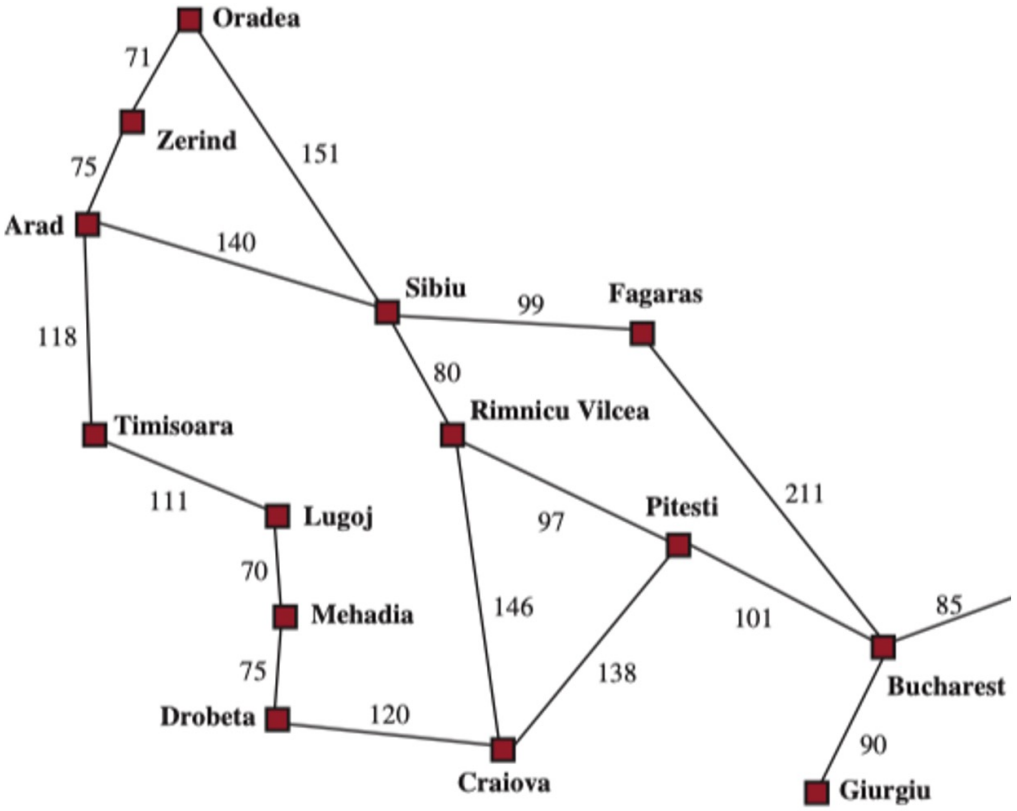

```
frontiers = [Timisoara, Oradea, Fagaras, Rimnicu Vilcea]
visited = [Arad, Zerind, Sibiu]
```

Step 5:

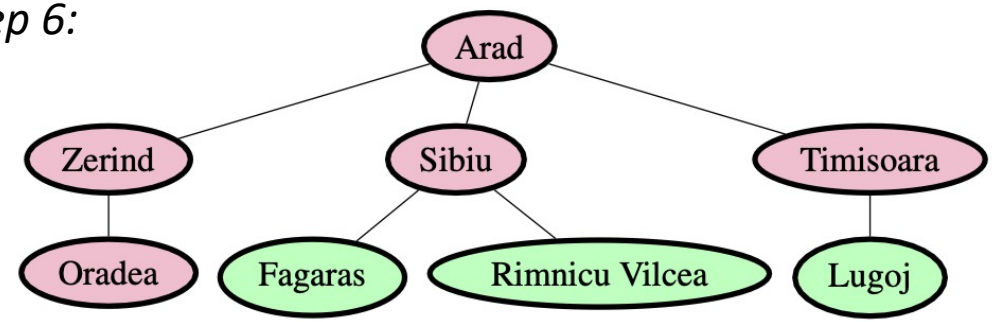



```
frontiers = [Oradea, Fagaras, Rimnicu Vilcea, Lugoj]
visited = [Arad, Zerind, Sibiu, Timisoara]
```

Sample Solution using BFS

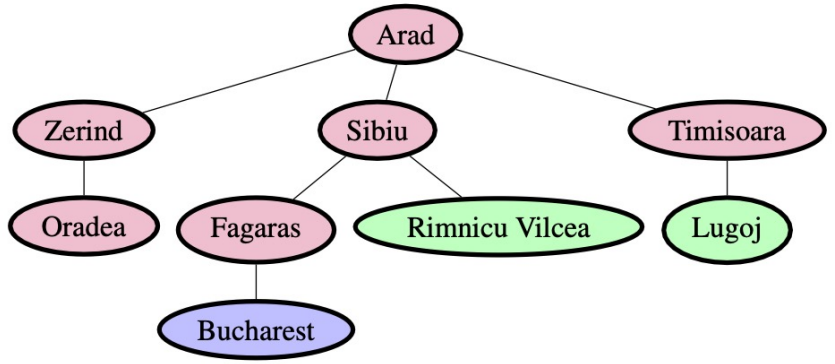


Step 6:




```
frontiers = [Fagaras, Rimnicu Vilcea, Lugoj]
visited = [Arad, Zerind, Sibiu, Timisoara, Oradea]
```

Step 7:

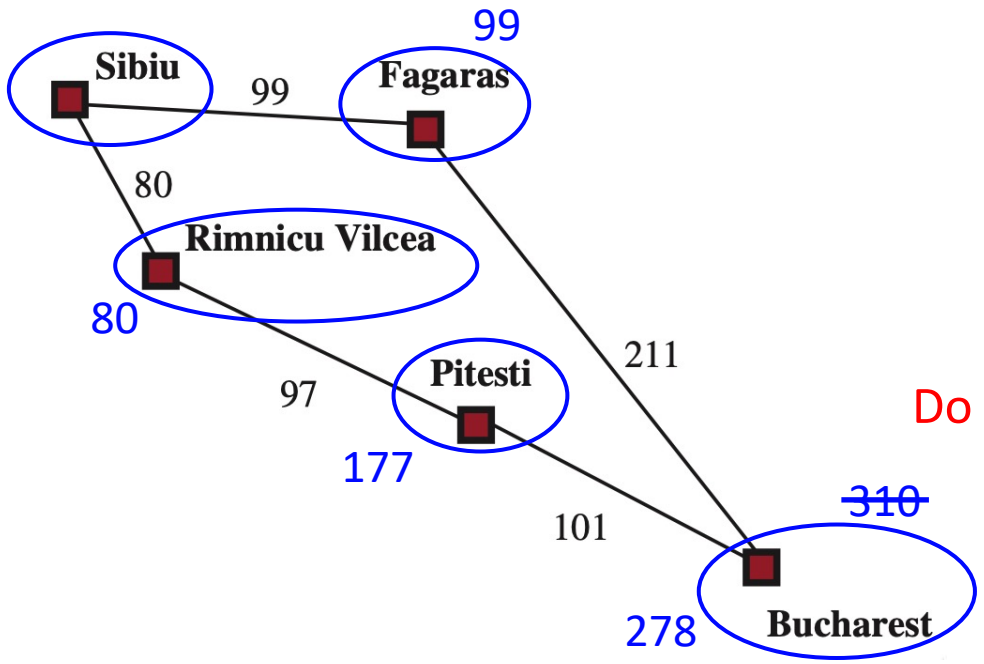



```
frontiers = [Rimnicu Vilcea, Lugoj]
visited = [Arad, Zerind, Sibiu, Timisoara, Fagaras]
```

solution: Arad → Sibiu → Fagaras → Bucharest

Uniform-cost Search

Consider the cost



Route-finding from Sibiu to Bucharest
Numbers are distances between cities

Expand the node with smallest cost from path to the node.

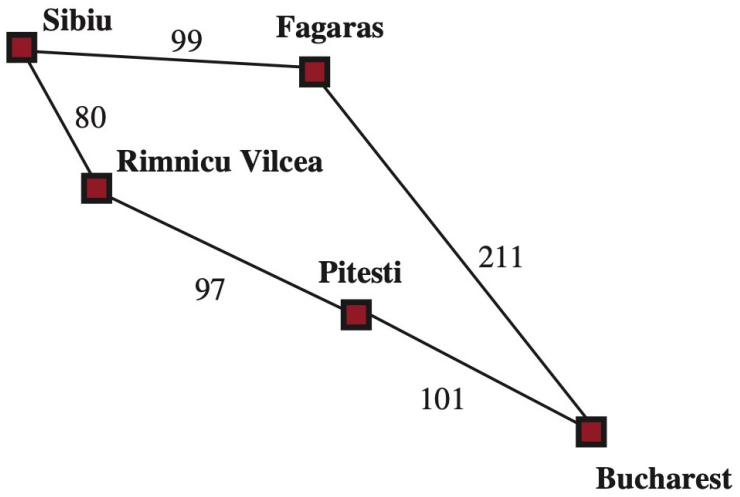
1. Expand Sibiu
   frontiers = [Fagaras(99), RV(80)]
   visited = [Sibiu]
2. Expand Fagaras or Rimnicu Vilcea?
   RV is the least-cost node.
   frontiers = [Fagaras(99), Pitesti(177)]
   visited = [Sibiu, RV]
3. Expand Fagaras or Pitesti? Fagaras is now the least-cost node.
   frontiers = [Pitesti(99), Bucharest(310)]
   visited = [Sibiu, RV, Fagaras]

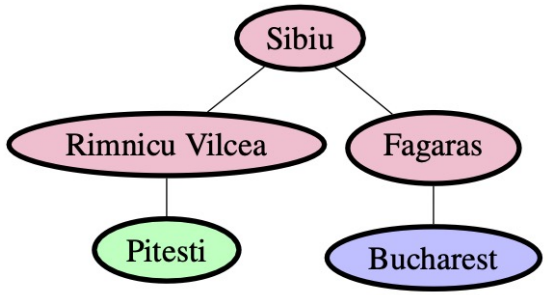
Do we check goal state now? No, we check goal state when expanding it
(other lower-cost paths may exist)

4. Expand Pitesti or Bucharest?
   Pitesti is now the least-cost node.
   frontiers = [Bucharest(~~310~~ 278)]
   visited = [Sibiu, RV, Fagaras, Pitesti]
5. Expand Bucharest: *goal state reached*

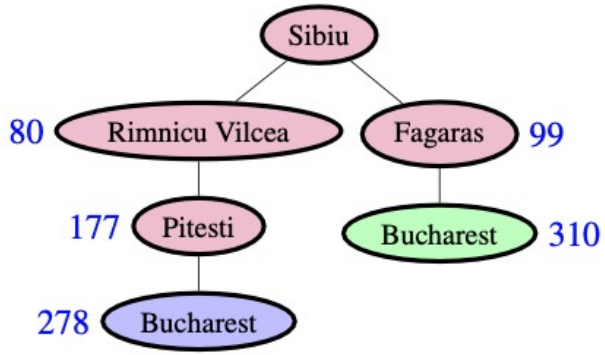
Solution: Sibiu → RV → Pitesti → Bucharest (cost=278)

Breadth-first Search vs Uniform-cost Search





BFS

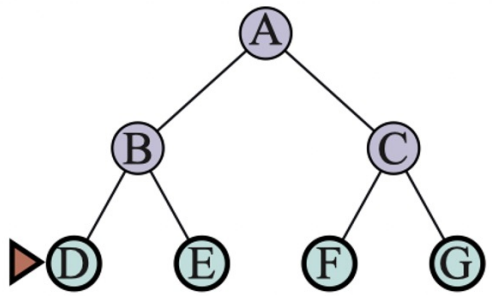


Uniform-cost Search

	BFS	Uniform-cost Search
When to check goal state?	When adding to the frontiers *(does not care about cost)*	When expanding the node *(lower-cost path may exist)*
Cost-optimal?	If all actions have the same cost	Yes, cost-optimal
Which is faster?	$O(b^d)$ ← *It depends* → $O(b^{1+\lfloor C^*/\epsilon \rfloor})$	

Could be much greater than b^d

Complexity Explained *(Out of the scope a little bit for this course)*



BFS

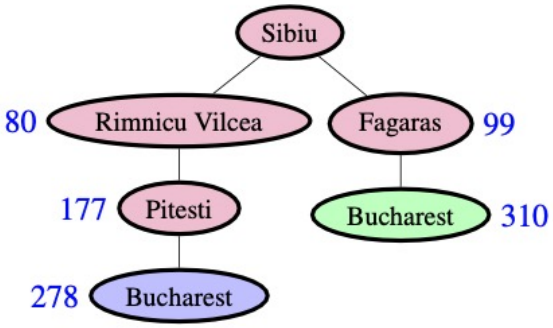
$O(b^d)$

b: avg. number of descendants of each node

d: depth of the goal

In worst case, number of nodes need to expand:

$$1 + b + b^2 + \cdots + \boxed{b^d}$$



Uniform-cost Search

$O(b^{1+\lfloor C^*/\epsilon \rfloor})$

C^*: cost of the optimal solution

ϵ: lower bound on the cost of each action

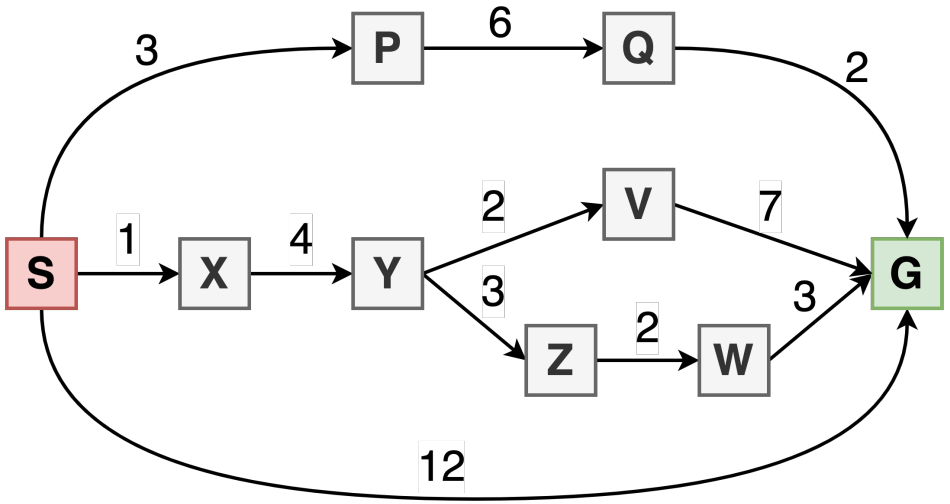
The depth of goal in worst case: $1 + \lfloor C^*/\epsilon \rfloor$

In worst case, number of nodes need to expand:

$$1 + b + b^2 + \cdots + \boxed{b^{1+\lfloor C^*/\epsilon \rfloor}}$$

Don't worry if you do not follow the O notion. Just be aware that the worst-case time and space complexity of uniform-cost search could be much greater than BFS.

Example and Exercise:



1. Use BFS to find a path from S to G. What is the solution?

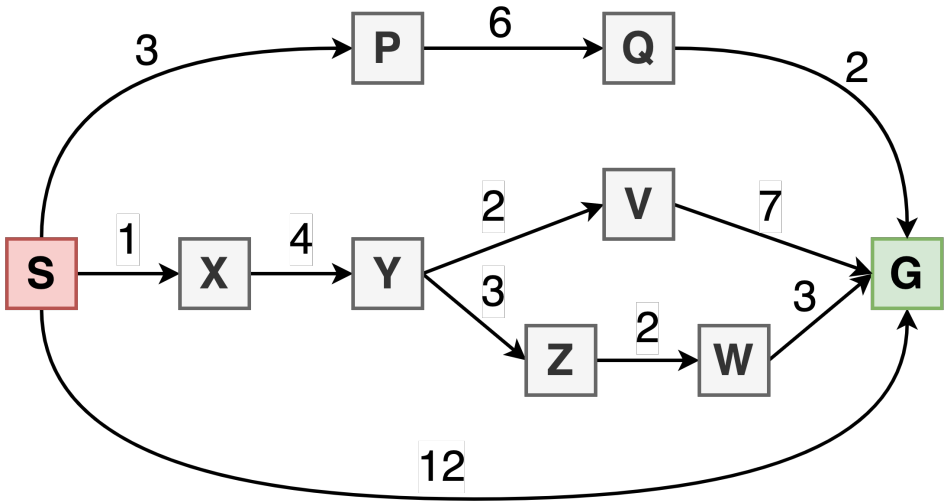
   A) S → X → Y → V → G *cost = 14*
   B) S → X → Y → Z → W → G *cost = 13*
   ✓ C) S → G *cost = 12*
   D) S → P → Q → G *cost = 11*

2. How many nodes did we expand in BFS?

   ✓ A) 1 *Expand S*
   B) 2
   C) 3
   D) 4

3. Is BFS cost-optimal in this example? **No**

Example and Exercise:



BFS solution: S → G (*cost = 12*) expanded one node only

4. What is the solution using uniform-cost search?

A) S → X → Y → V → G *cost = 14*
B) S → X → Y → Z → W → G *cost = 13*
C) S → G *cost = 12*
✓ D) S → P → Q → G *cost = 11*

5. How many nodes did we expand?

A) 1
B) 5
C) 7
✓ D) 9

6. Is Uniform-cost search cost-optimal? **Yes**

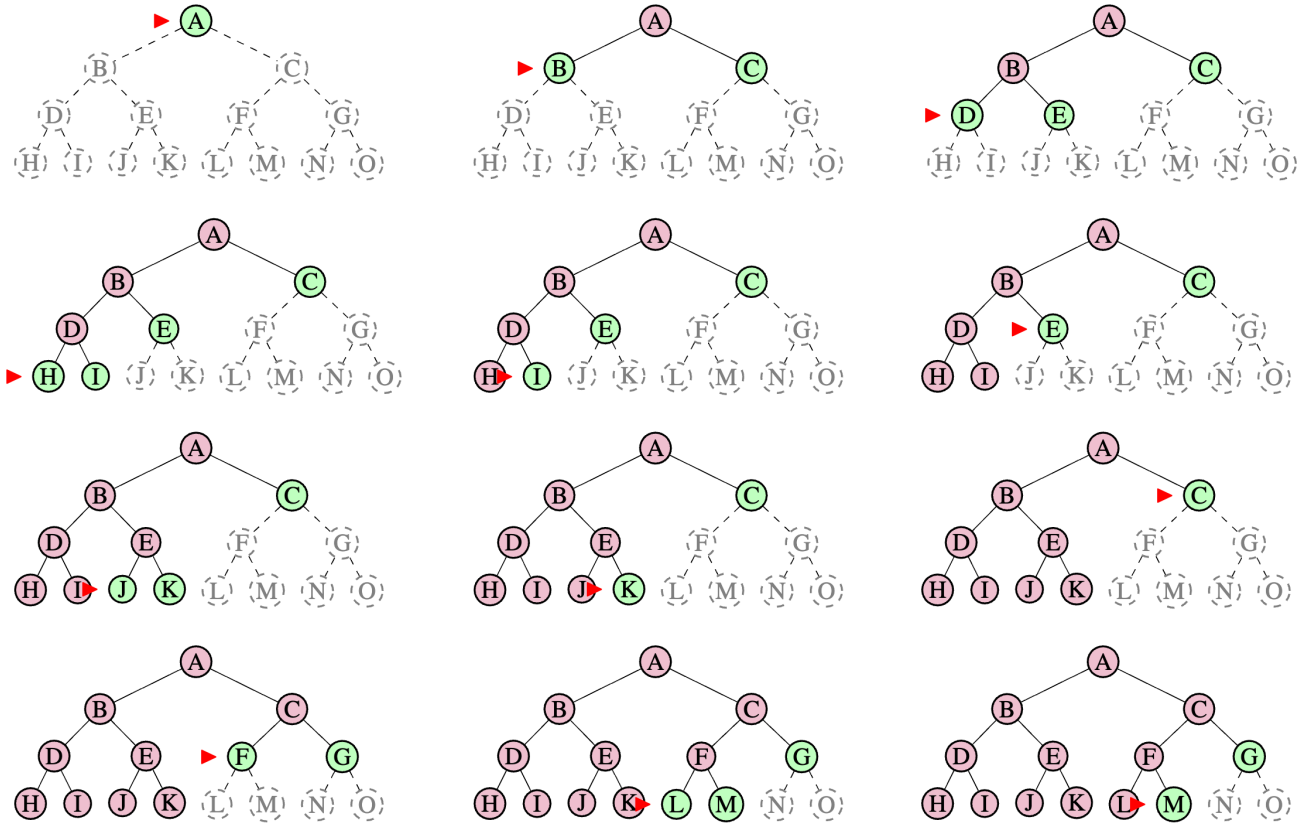
Road Map of Search Algorithms

- Uniformed search (blind search; n*o clue about how close a state is to the goal state*)
  - ✓ Breadth-first Search (BFS)
  - ✓ Uniform-cost Search
  - Depth-first Search (DFS)
    - ✓ Graph search implementation
    - Tree-like search implementation
  - Depth-limited Search and Iterative deepening Search
- Informed search (heuristic search; we have some hints about the location of goals)
  - Greedy Search
  - A* Search
- Constraint satisfaction problems (CSPs)

Depth-first Search (DFS; graph search)

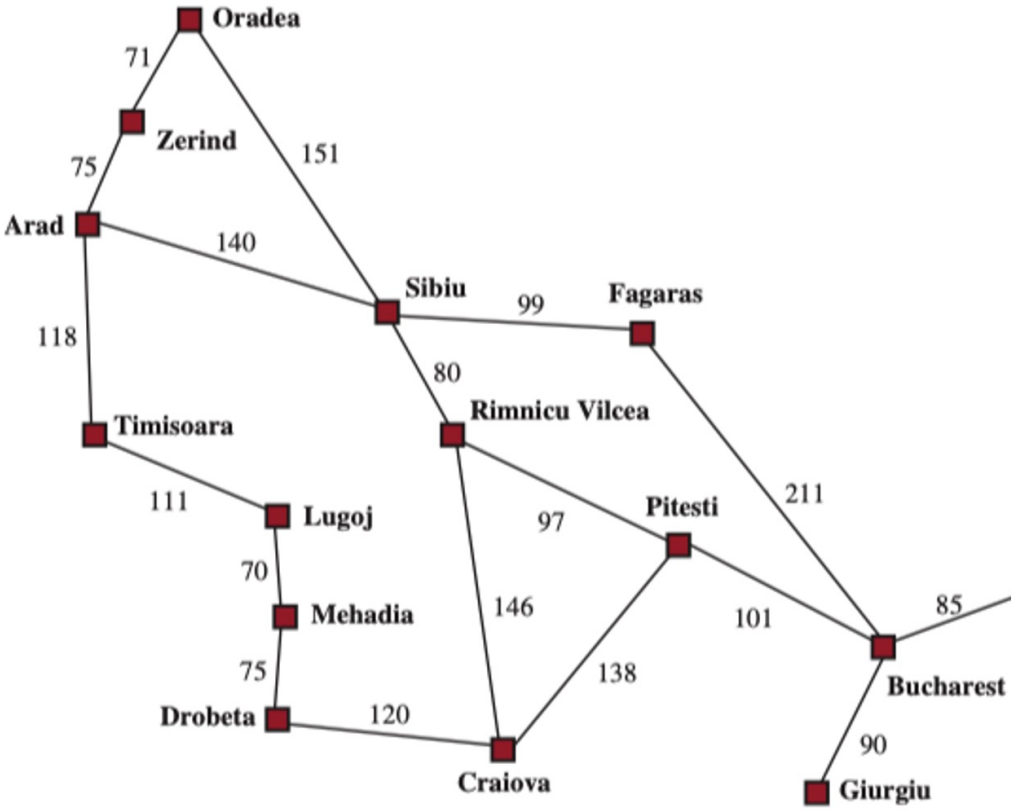
- Always expand one of the nodes at the deepest level of the tree.
- Store the frontiers (nodes to visit; in cyan)
- **If implemented in graph search, store the visited nodes (in purple).**

- If implemented in tree-like search, discard visited nodes that have no descendants. *(memory-efficient, but could visit the same node multiple times)*



Graph search implementation of DFS
All visited nodes are stored, one node only visited once

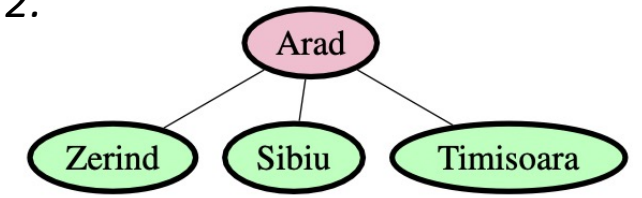
Sample Solution using DFS (graph search)



Step 1:

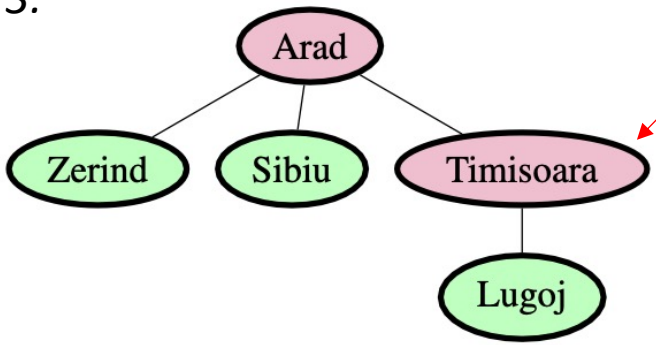

```
frontiers = [Arad]
visited = []
```

Step 2:



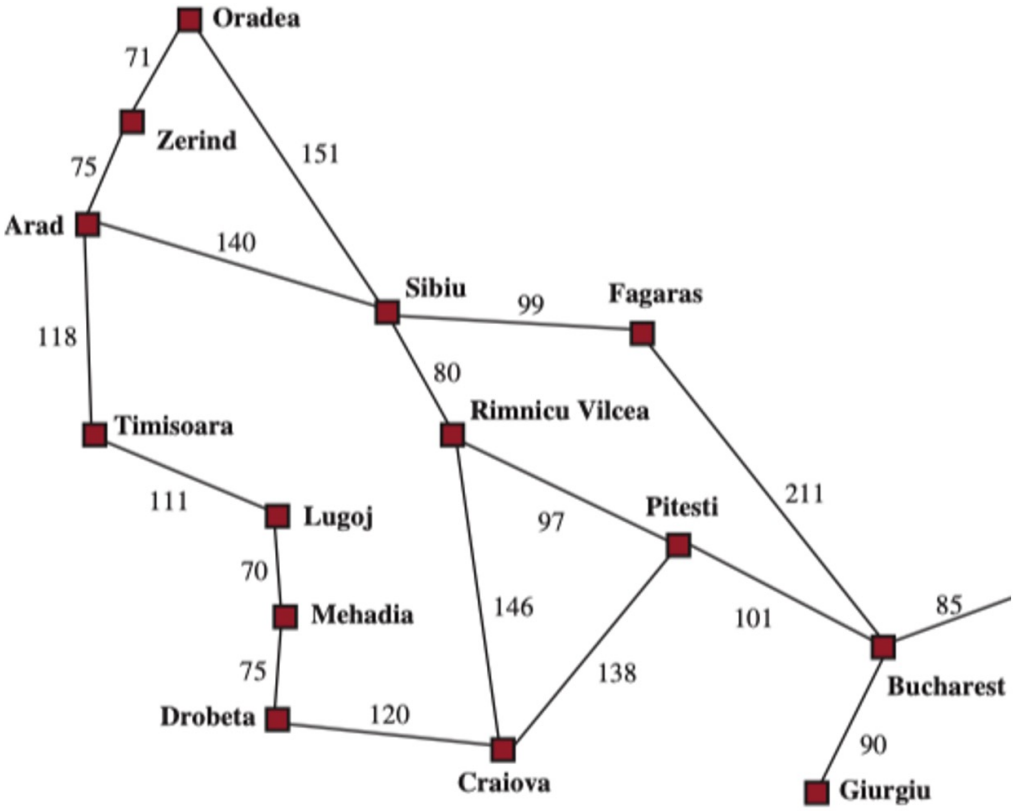

```
frontiers = [Zerind, Sibiu, Timisoara]
visited = [Arad]
```

Step 3:

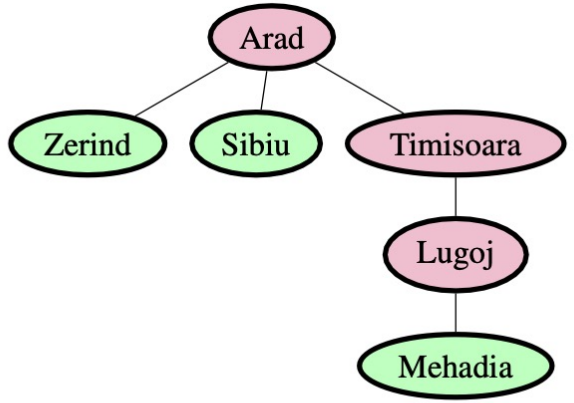



```
frontiers = [Zerind, Sibiu, Lugoj]
visited = [Arad, Timisoara]
```

Sample Solution using DFS (graph search)

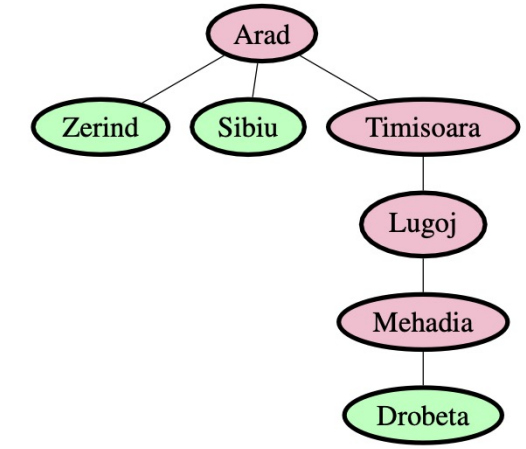


Step 4:



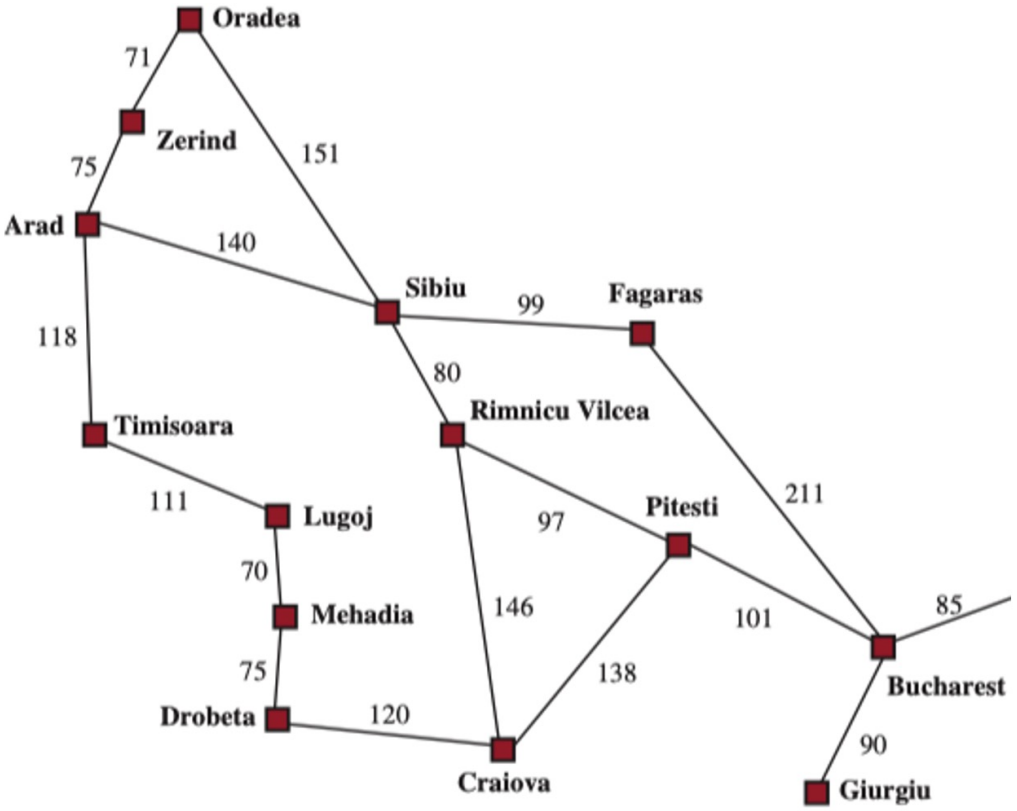

```
frontiers = [Zerind, Sibiu, Mehadia]
visited = [Arad, Timisoara, Lugoj]
```

Step 5:

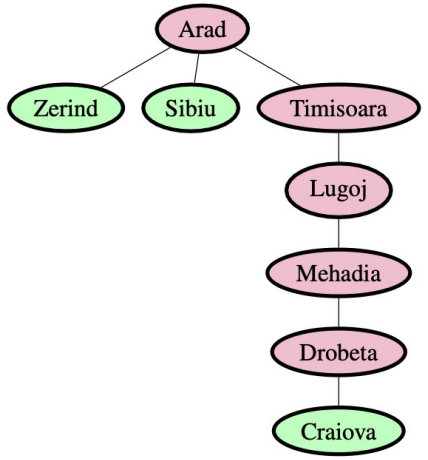



```
frontiers = [Zerind, Sibiu, Drobeta]
visited = [Arad, Timisoara, Lugoj, Mehadia]
```

Sample Solution using DFS (graph search)

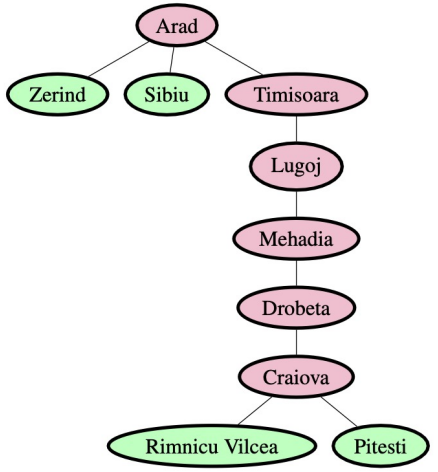


Step 6:



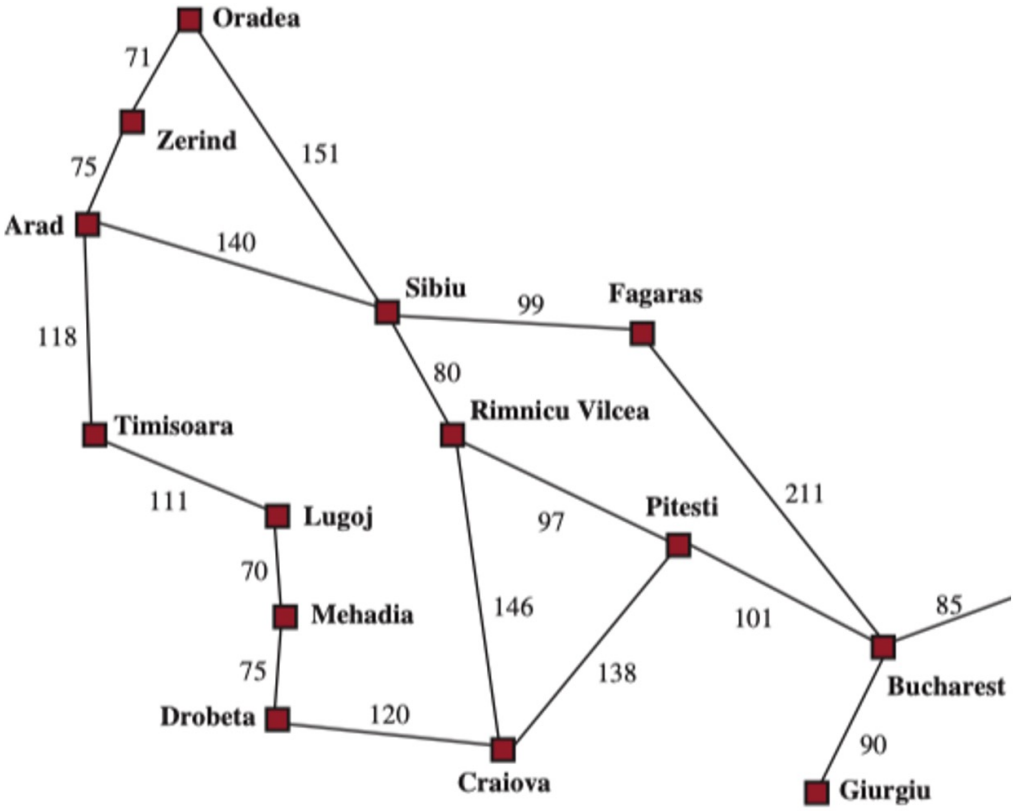

```
frontiers = [Zerind, Sibiu, Craiova]
visited = [Arad, Timisoara, Lugoj, Mehadia, Drobeta]
```

Step 7:

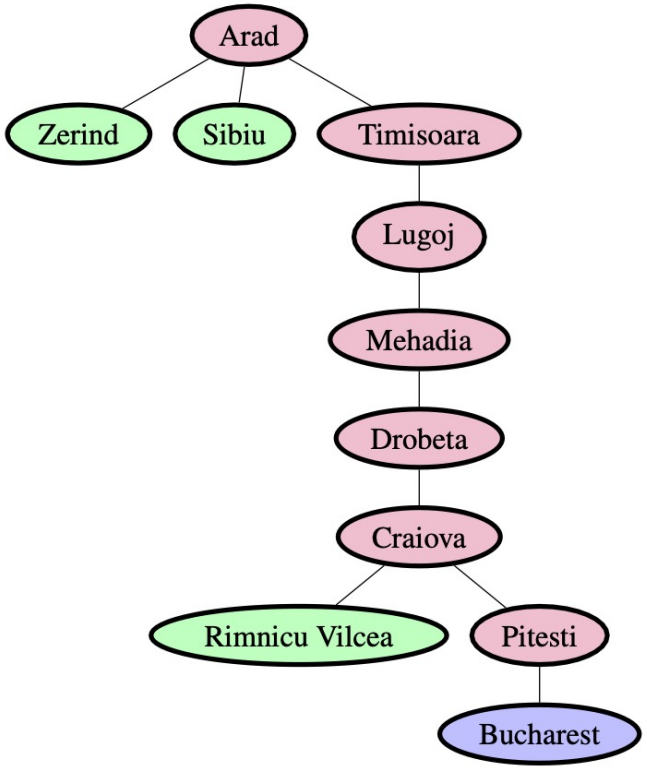



```
frontiers = [Zerind, Sibiu, Rimnicu Vilcea, Pitesti]
visited = [Arad, Timisoara, Lugoj, Mehadia, Drobeta, Craiova]
```

Sample Solution using DFS (graph search)



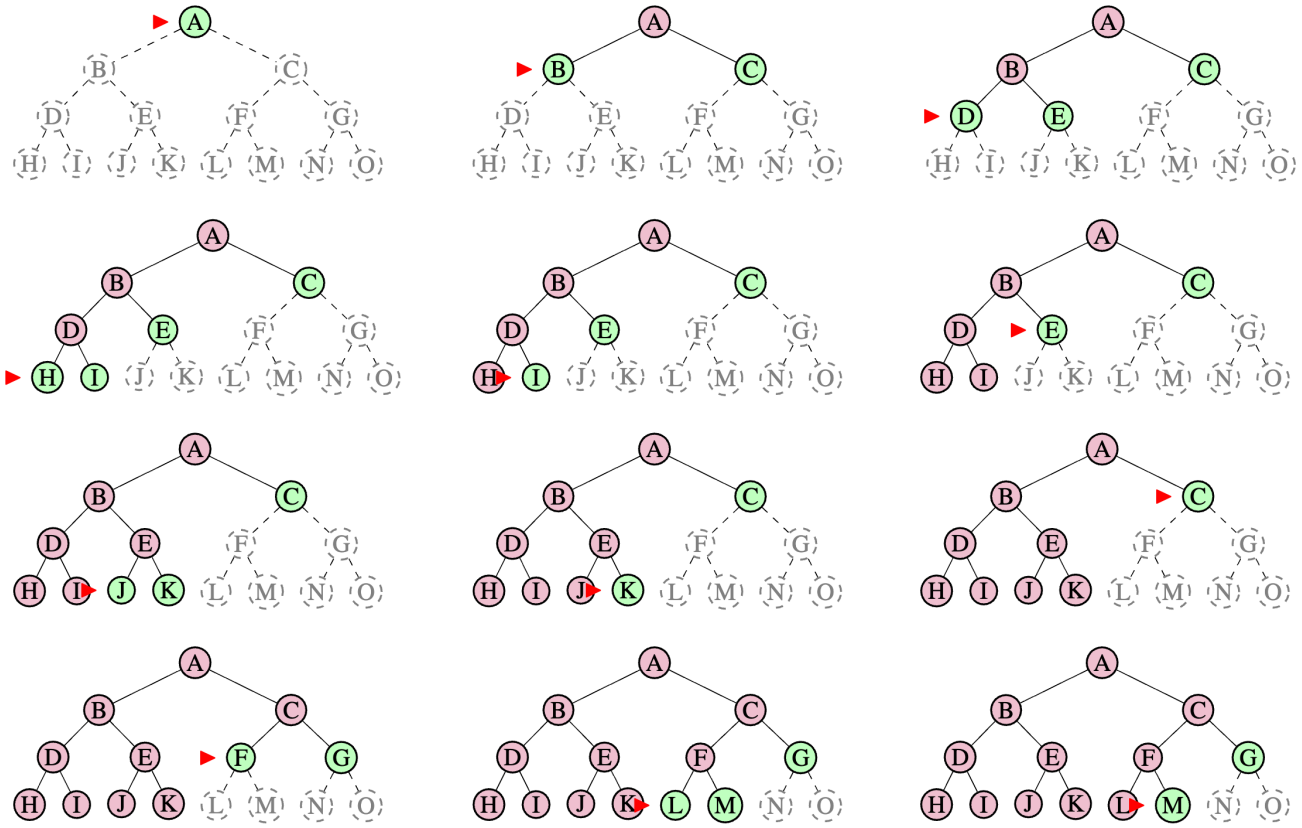
Step 8:




```
frontiers = [Zerind, Sibiu, Rimnicu Vilcea]
visited = [Arad, Timisoara, Lugoj, Mehadia, Drobeta, Craiova, Pitesti]
```

Solution: Arad → Timisoara → Lugoj → Mehadia → Drobeta → Craiova → Pitesti → Bucharest

Depth-first Search (DFS; graph search)



All visited nodes are stored in memory

Can we save some memory?

Discard visited nodes?

→ tree-like search

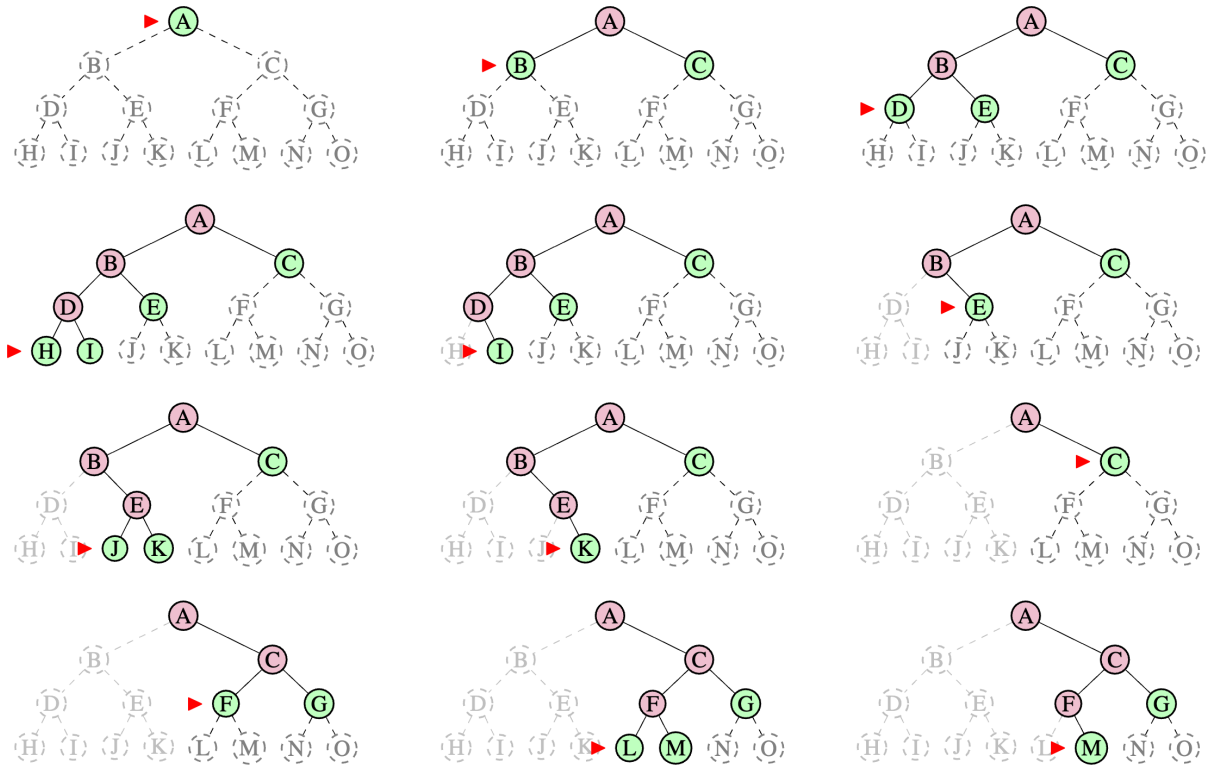
(actually more commonly used for DFS)

Depth-first Search (DFS; tree-like search)

- Always expand one of the nodes at the deepest level of the tree.
- Store the frontiers (nodes to visit; in cyan)
- If implemented in graph search, store the visited nodes (in purple).

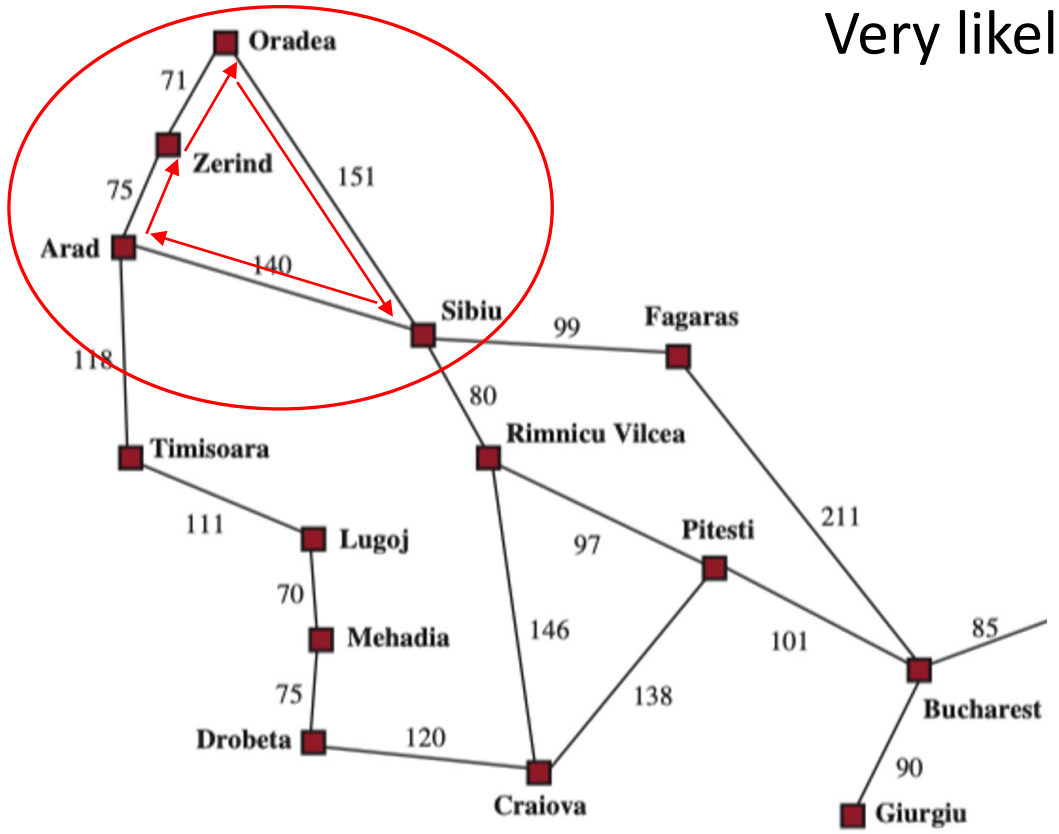
- **If implemented in tree-like search, discard visited nodes that have no descendants.** ***(memory-efficient, but could visit the same node multiple times)***

- Could get stuck going down the wrong path.

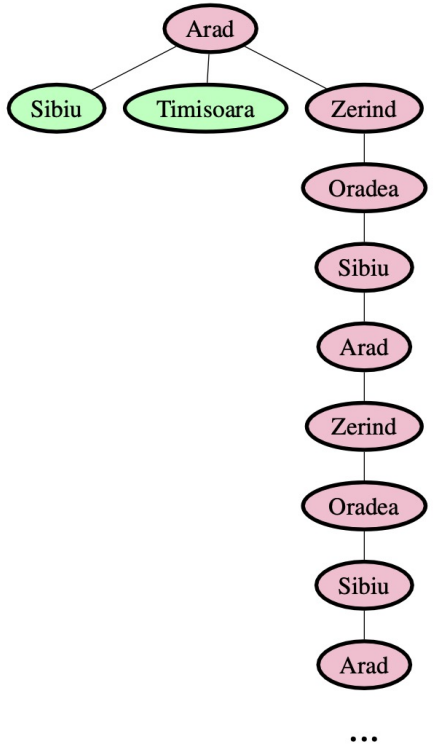


Tree-like search implementation of DFS
Visited nodes with no descendants are discarded.

What happens if we use tree-like search for path-finding?



Very likely, we will get stuck in infinite loops:

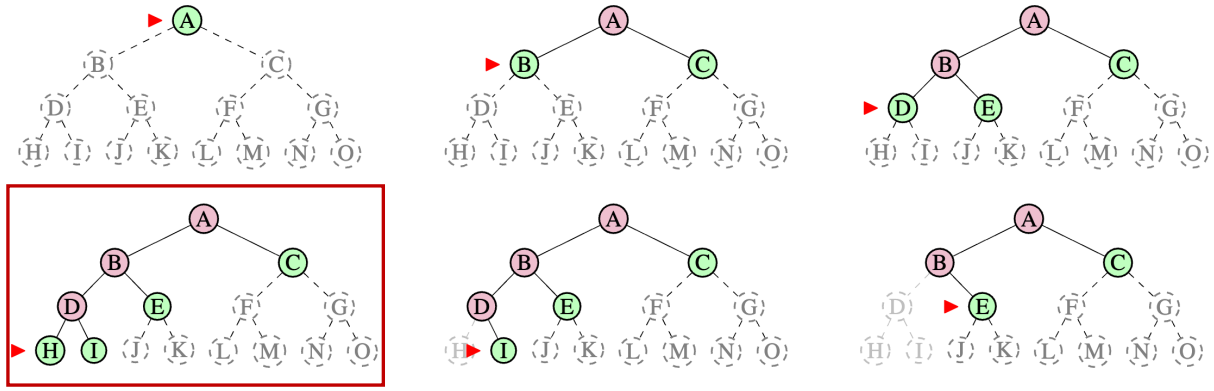


> ***Algorithms that cannot remember the past are doomed to repeat it.***

Pros and Cons of Tree-like DFS

Pros: Save memory

b: number of descendants of each node
m: maximum depth of the search tree

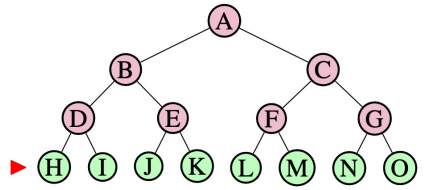


How many nodes in *frontiers* in worst case?

$$(b-1)*(m-1)+b=O(bm)$$

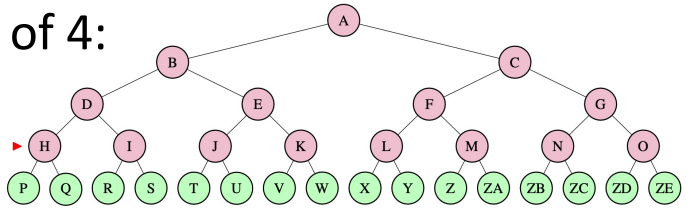
For BFS, it is $O(b^m)$

BFS with depth of 3:



Cons: visit a node multiple times, and get stuck in wrong path or infinite loops

BFS with depth of 4:

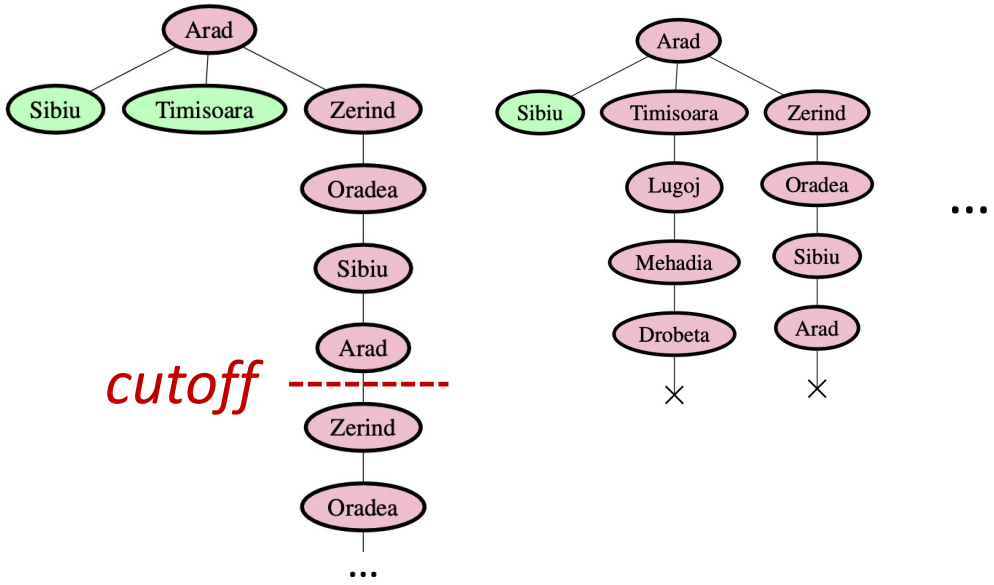


Depth-limited Search

Can we address the issue of getting stuck in wrong path or infinite loops when using DFS?

A simple idea: set a depth limit ℓ and ignore all nodes exceeding this depth.

E.g., we set $\ell = 4$:



...

If the goal is beyond depth ℓ, we cannot find it.

How to set a good ℓ ?
Need **knowledge** on the problem.

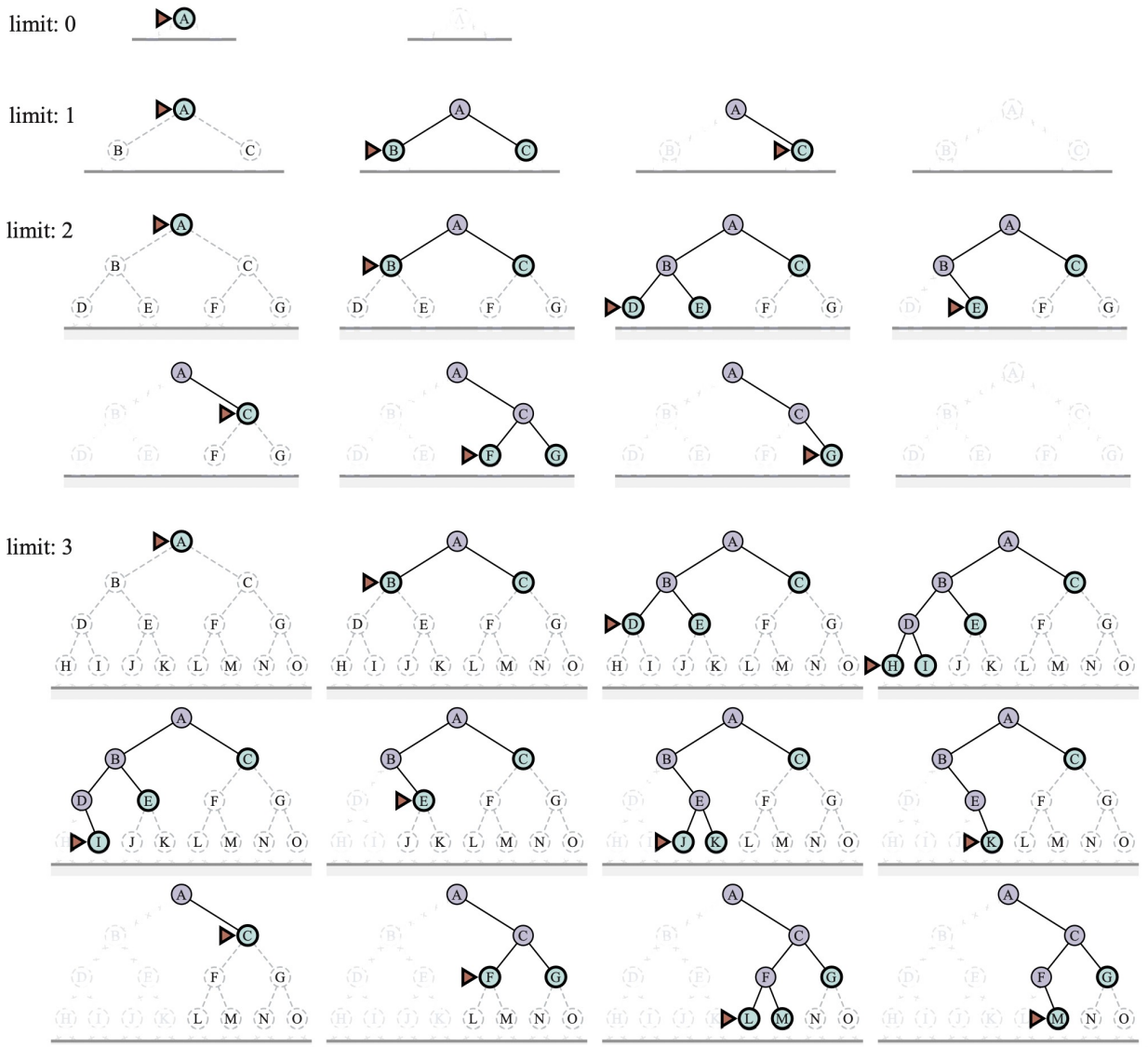
Examples:

- In the map of Romania, there are only 20 cities. ℓ=19?
- A closer look at the map: from any city, we can reach any other city in at most 9 actions. ℓ=9?

Iterative Deepening Depth-first Search

However, for most problems
we do not know a good ℓ …

Find ℓ by trying different values
from 0 up to infinity



Save memory at the cost of more time

- 😎 Tree-like DFS: save memory
- ☹ Repeat previous levels: need more time
- 😉 Most of nodes are in bottom levels,
  does not matter much to repeat upper levels

Comparing Uninformed Search Algorithms

Is it guaranteed to find a solution, if there exists one? When there is not, can it correctly report failure?

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]
Optimal cost?	Sometimes[3]	Yes	No	No	Sometimes[3]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$

No need to memorize the table; understand their pros and cons.

b: avg. number of descendants of each node;
m: maximum depth;
d: depth of the shallowest solution (if there is no solution, $d = m$);
ℓ: depth limit;
C^*: cost of the optimal solution
ϵ: lower bound on the cost of each action

[3]: when all actions have the same cost

How to Choose the Proper Algorithm?

Is it guaranteed to find a solution, if there exists one? When there is not, can it correctly report failure?

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]
Optimal cost?	Sometimes[3]	Yes	No	No	Sometimes[3]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$

If we know all actions cost the same: BFS? Uniform-Cost Search (= BFS in this case)?

If we need to find the optimal-cost solution: Uniform-Cost Search

If we have limited memory: Iterative Deepening DFS

If we know that the state space is acyclic or m is small: DFS

Analyze the problem and resources case by case.

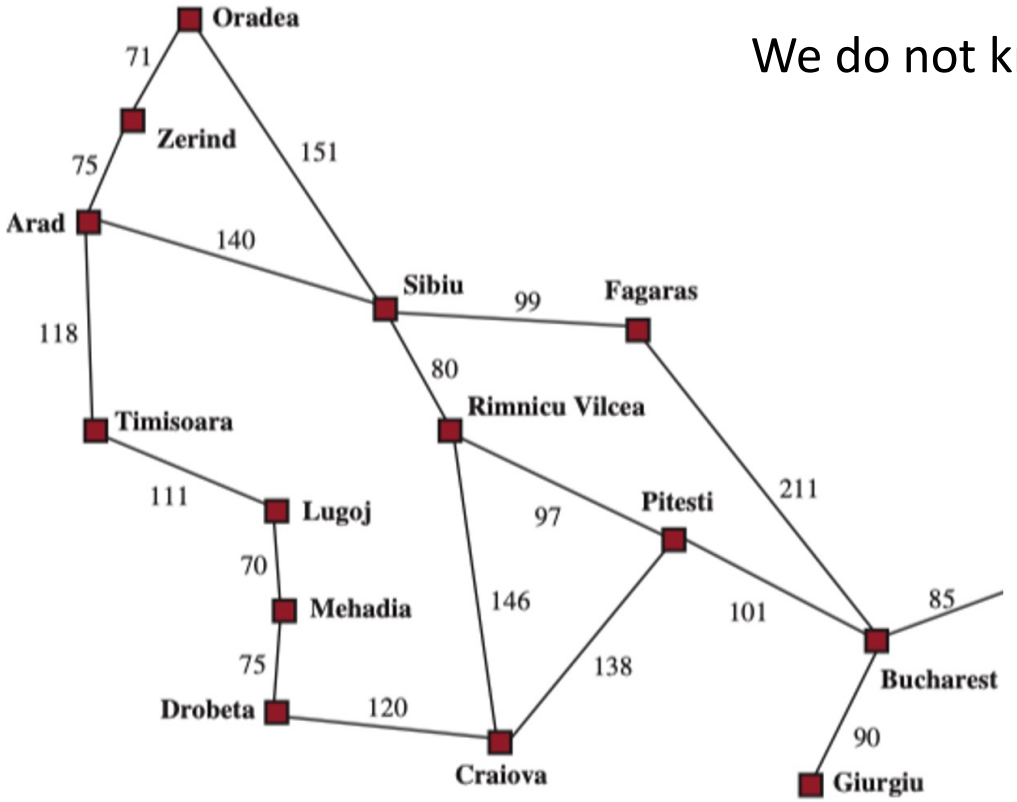
Road Map of Search Algorithms

- Uniformed search (blind search; n*o clue about how close a state is to the goal state*)
  - ✓ Breadth-first Search (BFS)
  - ✓ Uniform-cost Search
  - ✓ Depth-first Search (DFS)
    - ✓ Graph search implementation
    - ✓ Tree-like search implementation
  - ✓ Depth-limited Search and Iterative deepening Search
- Informed search (heuristic search; we have some hints about the location of goals)
  - Greedy Search
  - A* Search
- Constraint satisfaction problems (CSPs)

Informed (Heuristic) Search

Sometimes, we have some domain-specific hints about the location of the goal.

For example: routh-finding from Arad to Bucharest



We do not know the path, but we know the straight-line distance on the map

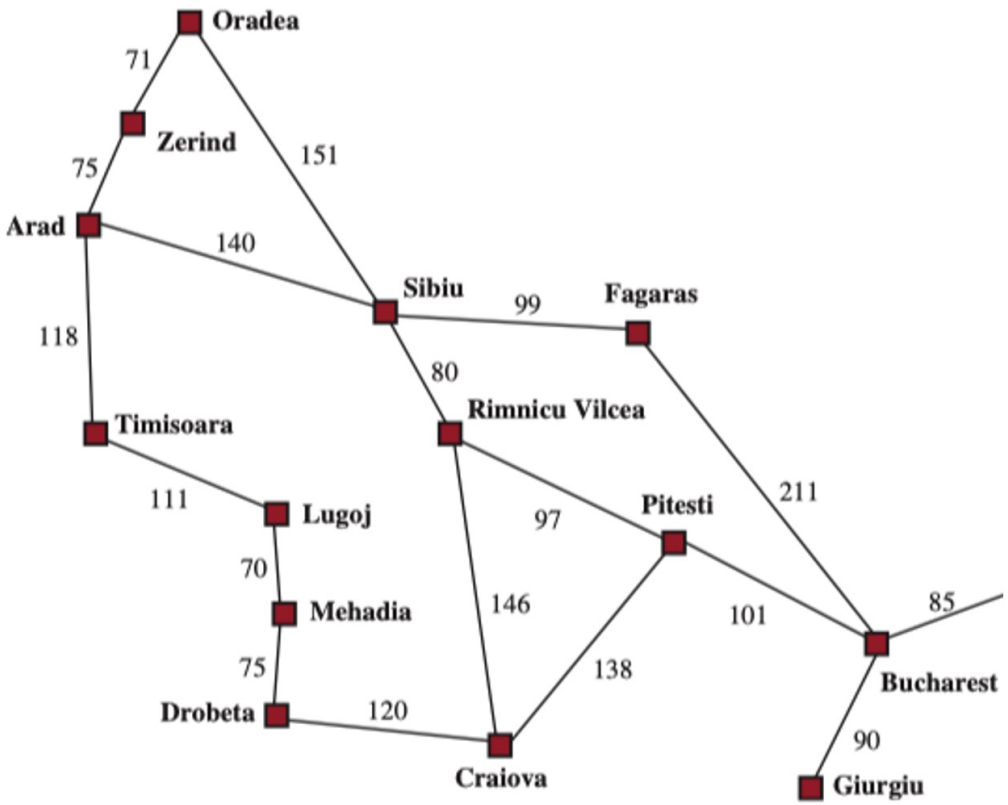
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

e.g., straight-line distance from each city to Bucharest

Informed (Heuristic) Search

How to use the domain-specific hints?

Define a **heuristic function** $h(n)$ = estimated cost of the *cheapest* path from the *state at node n* to *a goal state*



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

e.g., straight-line distance from each city to Bucharest

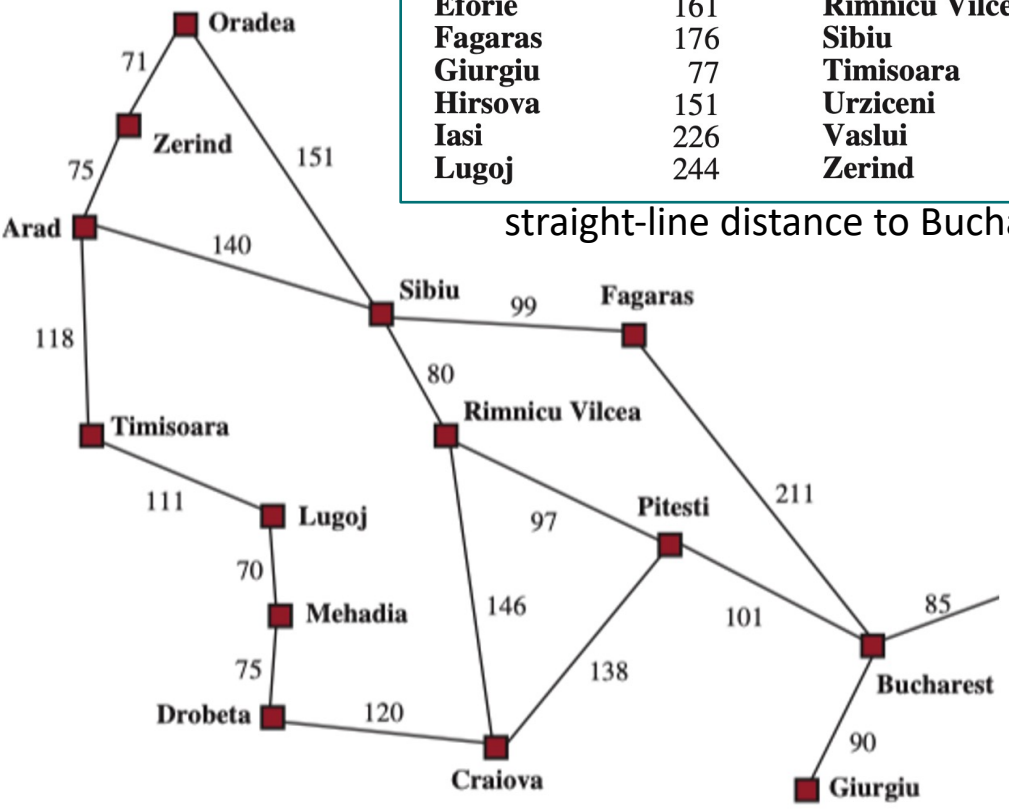
$$h_{SLD}(\text{Arad}) = 366$$
$$h_{SLD}(\text{Sibiu}) = 253$$
$$h_{SLD}(\text{Oradea}) = 380$$

Greedy Search

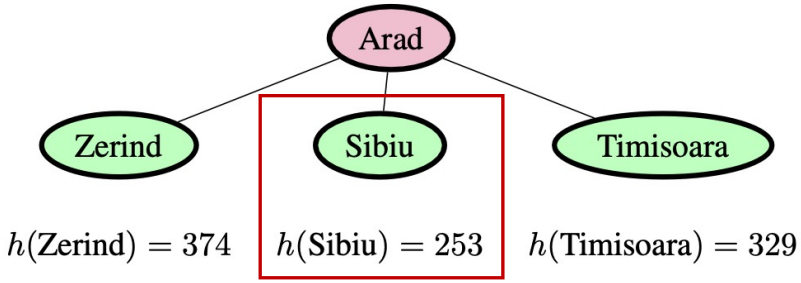
We expand the node which has the lowest $h(n)$.

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

straight-line distance to Bucharest



1. Expand <u>Arad</u>



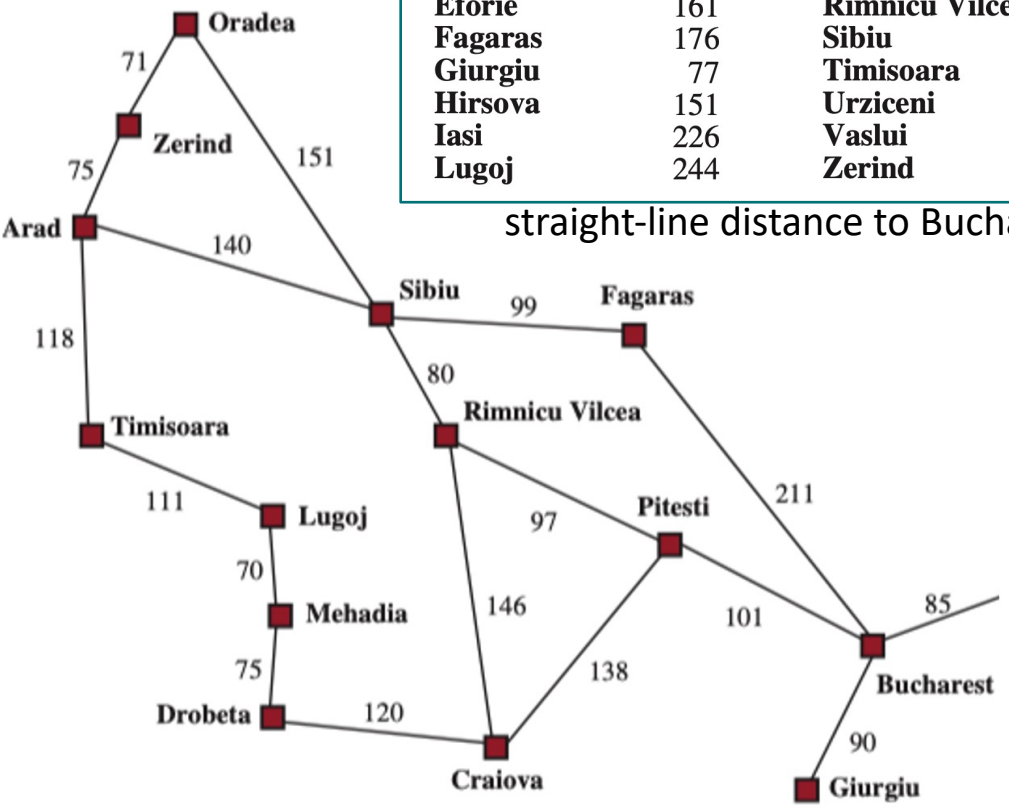
frontiers = [Zerind(374), Sibiu(253), Timisoara(329)]

Greedy Search

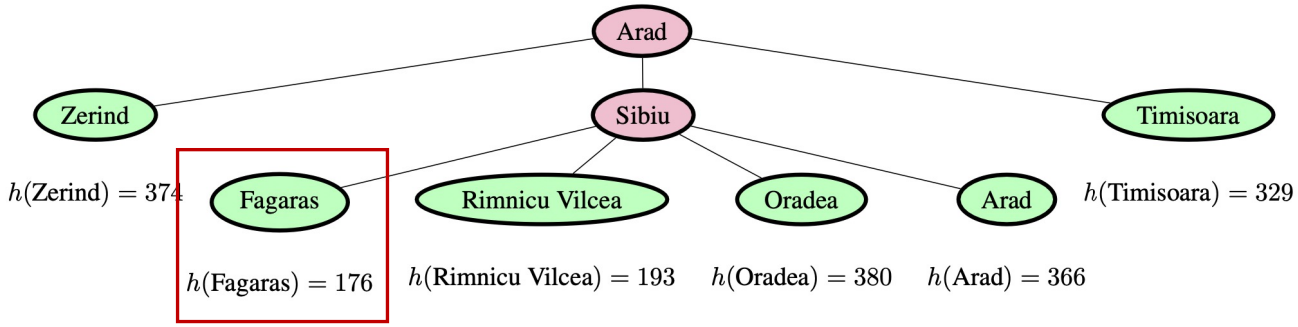
We expand the node which has the lowest $h(n)$.

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

straight-line distance to Bucharest



2. Expand Sibiu



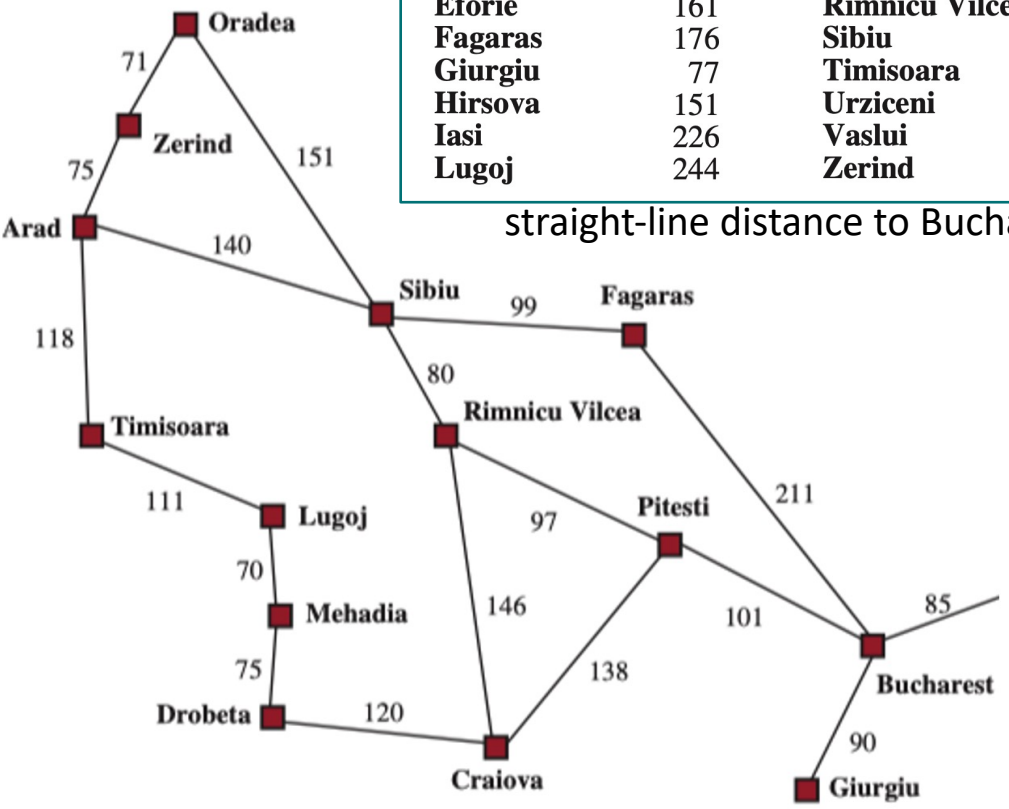
frontiers = [Zerind(374), Timisoara(329), Fagaras(176), RV(193), Oradea(380), Arad(366), Timisoara(329)]

Greedy Search

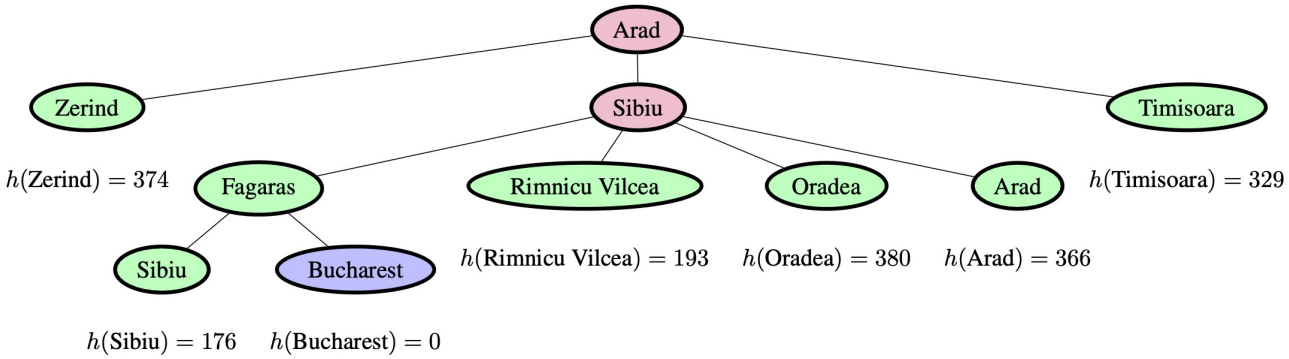
We expand the node which has the lowest $h(n)$.

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

straight-line distance to Bucharest



3. Expand Fagaras



Solution: Arad → Sibiu → Fagaras → Bucharest

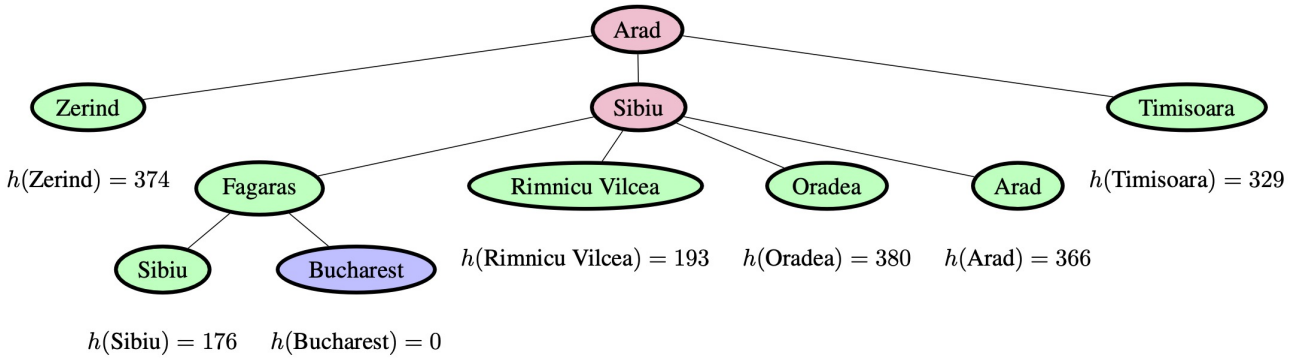
Cost = 140 + 99 + 211 = 450

Is it cost-optimal? **No**

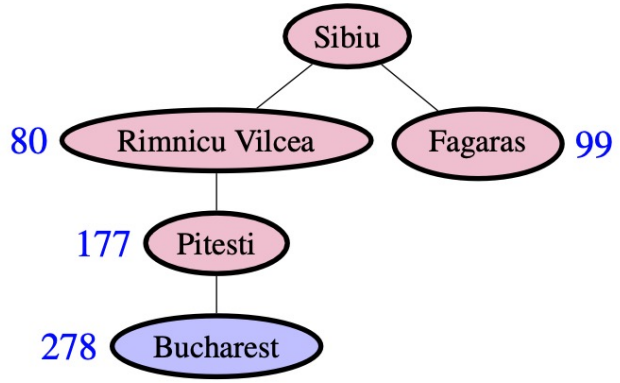
Cost-optimal Solution: Arad → Sibiu → RV → Pitesti → Bucharest

Cost = 140 + 80 + 97 + 101 = 418

Greedy Search vs Uniform-Cost Search



Greedy Search



Uniform-cost Search

	Greedy Search	Uniform-cost Search
Is informed?	Informed search *Make use of the heuristic function $h(n)$*	Informed search *Use the cost of the path so far* denote it by $g(n)$
Cost-optimal?	Not necessarily	Yes, cost-optimal

A* Search

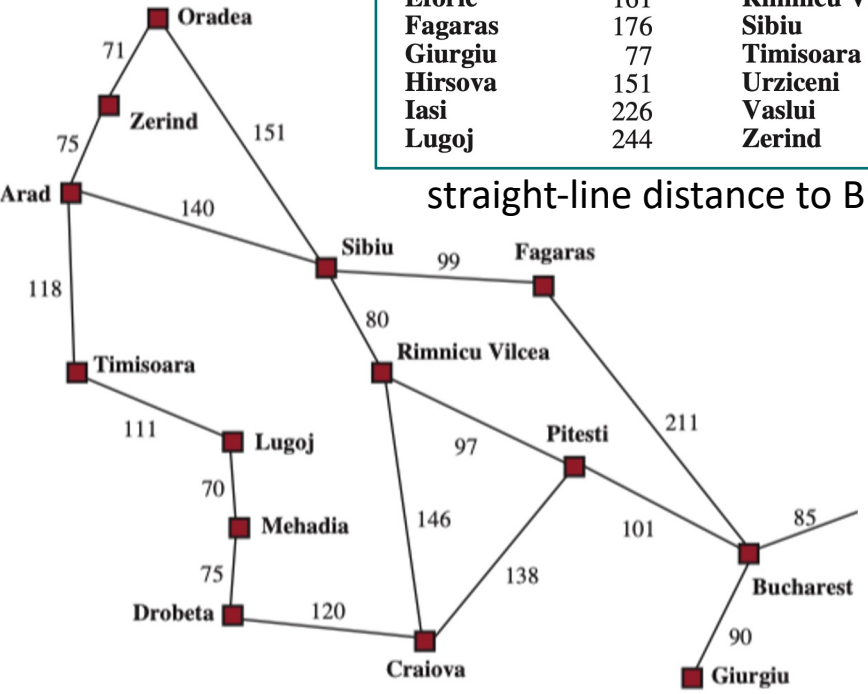
Combining greedy search and uniform-cost search:

We expand the node which has the lowest $h(n) + g(n)$.

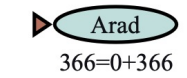
estimated cost of this path

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

straight-line distance to Bucharest



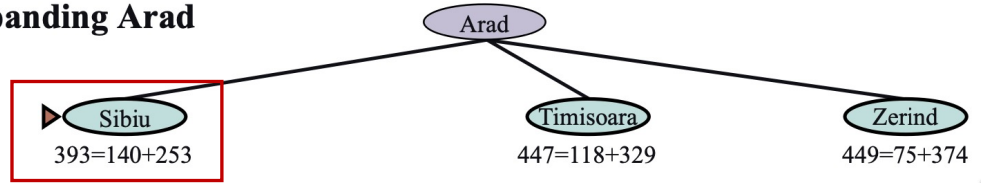
(a) The initial state



$g(\text{Arad}) = 0$

$h(\text{Arad}) = 366$

(b) After expanding Arad



$g(\text{Sibiu}) = 140$

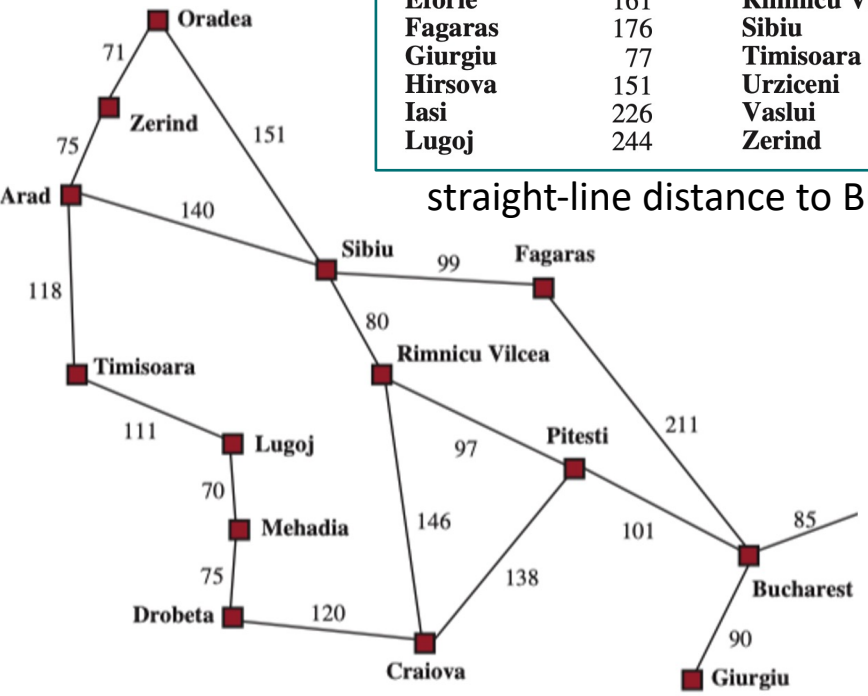
$h(\text{Sibiu}) = 253$

A* Search

Combining greedy search and uniform-cost search:

We expand the node which has the lowest $h(n) + g(n)$.

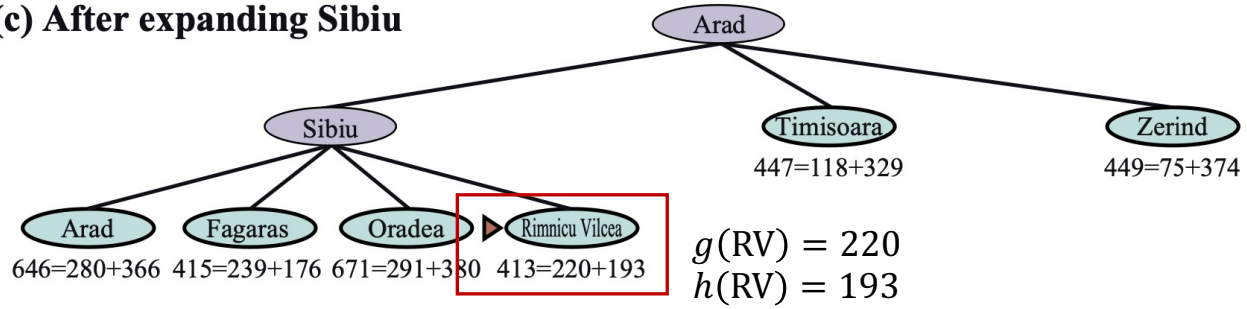
estimated cost of this path



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

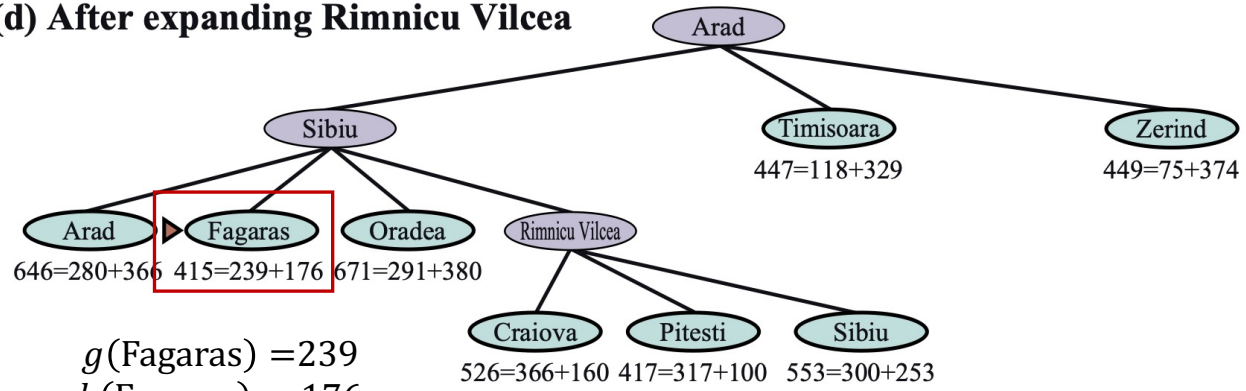
straight-line distance to Bucharest

(c) After expanding Sibiu



$g(\text{RV}) = 220$
$h(\text{RV}) = 193$

(d) After expanding Rimnicu Vilcea



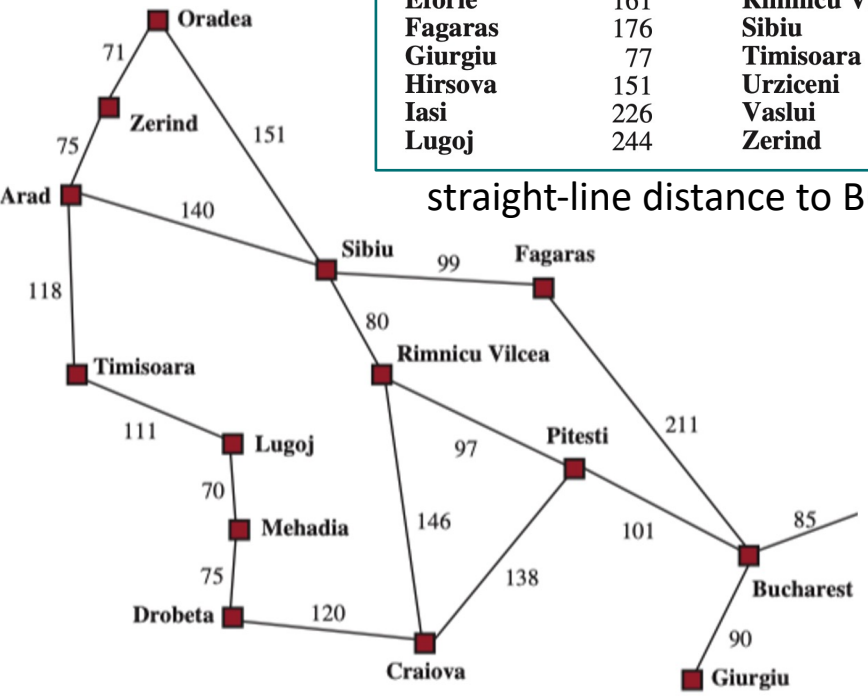
$g(\text{Fagaras}) = 239$
$h(\text{Fagaras}) = 176$

A* Search

Combining greedy search and uniform-cost search:

We expand the node which has the lowest $h(n) + g(n)$.

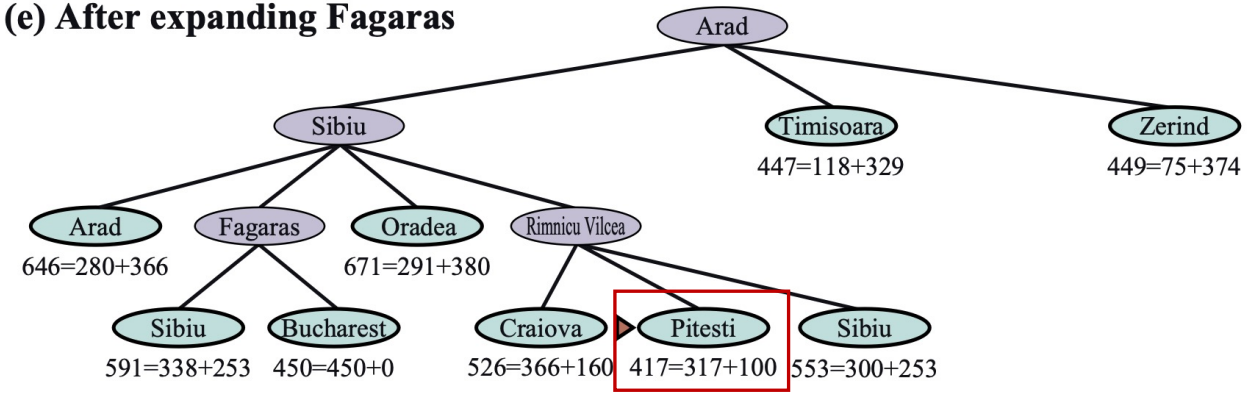
estimated cost of this path



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

straight-line distance to Bucharest

(e) After expanding Fagaras



$g(\text{Pitesti}) = 317$

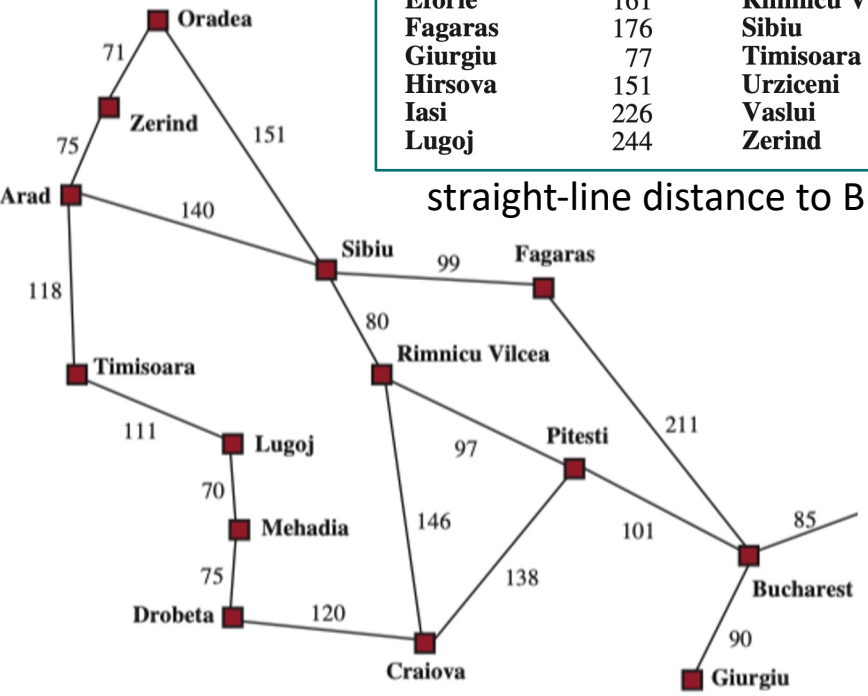
$h(\text{Pitesti}) = 100$

A* Search

Combining greedy search and uniform-cost search:

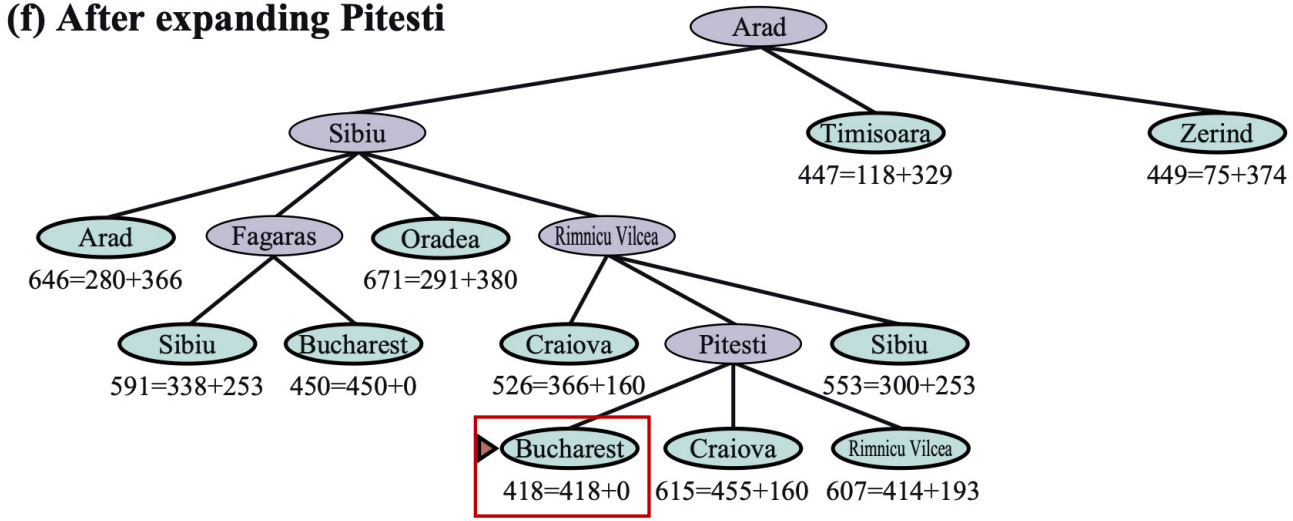
We expand the node which has the lowest $h(n) + g(n)$.

estimated cost of this path



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

straight-line distance to Bucharest



$g(\text{Bucharest}) = 418$

$h(\text{Bucharest}) = 0$

Solution: Arad → Sibiu → RV → Pitesti → Bucharest

Is it cost-optimal? **Yes, in this example**

A* Search

Is A* Search always cost-optimal?

Depends on certain properties of the heuristic, e.g., *admissibility*.

Admissible heuristic: the one that *never overestimates* the cost to reach a goal.

With an admissible heuristic, A* is cost-optimal.

Read the proof if you wonder why:

(optional)

Suppose the optimal cost is C*, but the algorithm returns C > C*
⇒ There must be some node n on the optimal path but is unexpanded (otherwise, the algorithm will return the optimal solution).

Contradiction

$$
\begin{aligned}
f(n) &> C^* && \text{(otherwise } n \text{ would have been expanded)} \\
f(n) &= g(n)+h(n) && \text{(by definition)} \\
f(n) &= g^*(n)+h(n) && \text{(because } n \text{ is on an optimal path)} \\
f(n) &\le g^*(n)+h^*(n) && \text{(because of admissibility, } h(n) \le h^*(n)) \\
f(n) &\le C^* && \text{(by definition, } C^* = g^*(n)+h^*(n))
\end{aligned}
$$

A* Search

Is A* Search always cost-optimal?

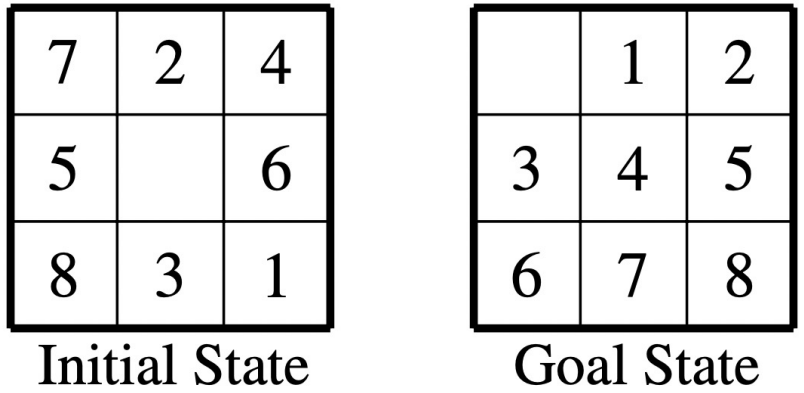
Depends on certain properties of the heuristic, e.g., *admissibility*.

Admissible heuristic: the one that *never overestimates* the cost to reach a goal.

With an admissible heuristic, A* is cost-optimal.

- The closer your heuristic function is closer to reality, the faster is the algorithm.
- Never overestimate the cost: in an extreme case, $h(n) = 0$ for all nodes, then it degenerates to uniform-cost search.

How much does heuristic function help?



8-puzzle problem: move the blank tile *left*, *up*, *right*, and *down* to make it match with the goal state. Number of reachable states: 9!/2 = 181,400.

How about 15-puzzle? Number of reachable states: 16! /2 states — over 10 trillion.

Road Map of Search Algorithms

- Uniformed search (blind search; n*o clue about how close a state is to the goal state*)
  - ✓ Breadth-first Search (BFS)
  - ✓ Uniform-cost Search
  - ✓ Depth-first Search (DFS)
    - ✓ Graph search implementation
    - ✓ Tree-like search implementation
  - ✓ Depth-limited Search and Iterative deepening Search
- Informed search (heuristic search; we have some hints about the location of goals)
  - ✓ Greedy Search
  - ✓ A* Search
- Constraint satisfaction problems (CSPs)

Summary of Search Algorithms

Search: repeatedly select a node from frontiers and expand it until a solution is found.

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

Tree search does not store the visited nodes.

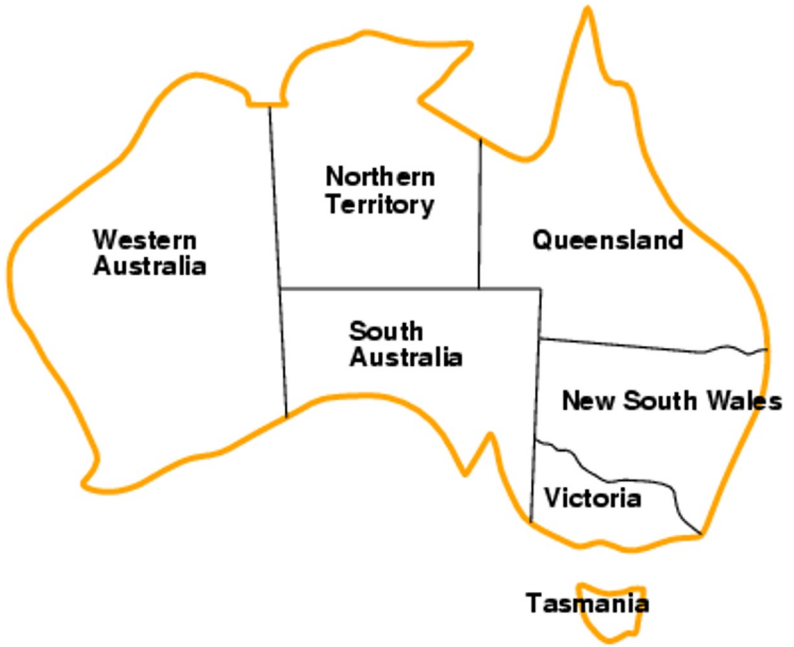
```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```

Search algorithms differ in how they choose a node to expand:

- **BFS**: choose the shallowest node
- **DFS**: choose the deepest node (depth-limited: the deepest one that does not exceed the limit)
- **Uniform-cost Search**: the node with smallest $g(n)$
- **Greedy Search**: the node with smallest $h(n)$
- **A* Search**: the node with smallest $h(n) + g(n)$

Constraint Satisfaction Problems (CSPs)

Consider the Australian map



There are 7 provinces:

WA, NT, Q, NSW, V, SA, T

Color each province with either

red, green, or blue

Rule: Adjacent regions must have different colors.

Example: color(WA) ≠ color(NT)

Constraint Satisfaction Problems (CSPs)

- A set of variables: $X_1, X_2, X_3, \ldots, X_n$
- Each variable has a non-empty domain of possible vales, e.g., $X_1 \in \{1, 2, 3, 4\}$
- A set of constraints, e.g., $X_1 \neq 3, \quad X_2 > X_3$
- Solution: assign a value to each variable such that none of the constraints are violated.
- Simple algorithm: **backtracking**; systematically go through all possible values, check all constraints when making variable assignment. When constraints are violated, back up to the previous level.

Constraint Satisfaction Problems (CSPs)

Example problem:

- $A \in \{1, 2, 3\}$
- $B \in \{1, 2, 3\}$
- $C \in \{1, 2, 3\}$

- Constraints:
  - $A > B$
  - $B \neq C$
  - $A \neq C$

A assignment	B assignment		C assignment	
A = 1	B = 1	✗		
	B = 2	✗		
	B = 3	✗		
A = 2	B = 1		C = 1	✗
	B = 1		C = 2	✗
	B = 1		C = 3	✓

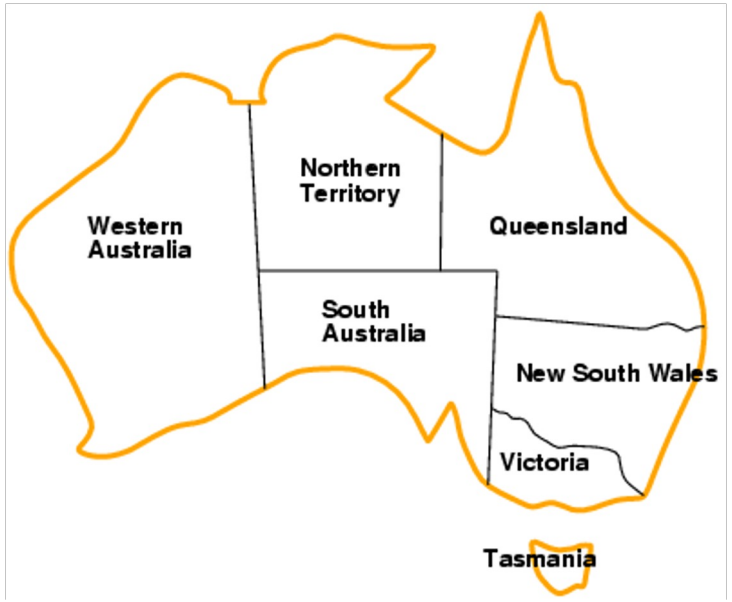
Solution: A = 2, B = 1, C = 3

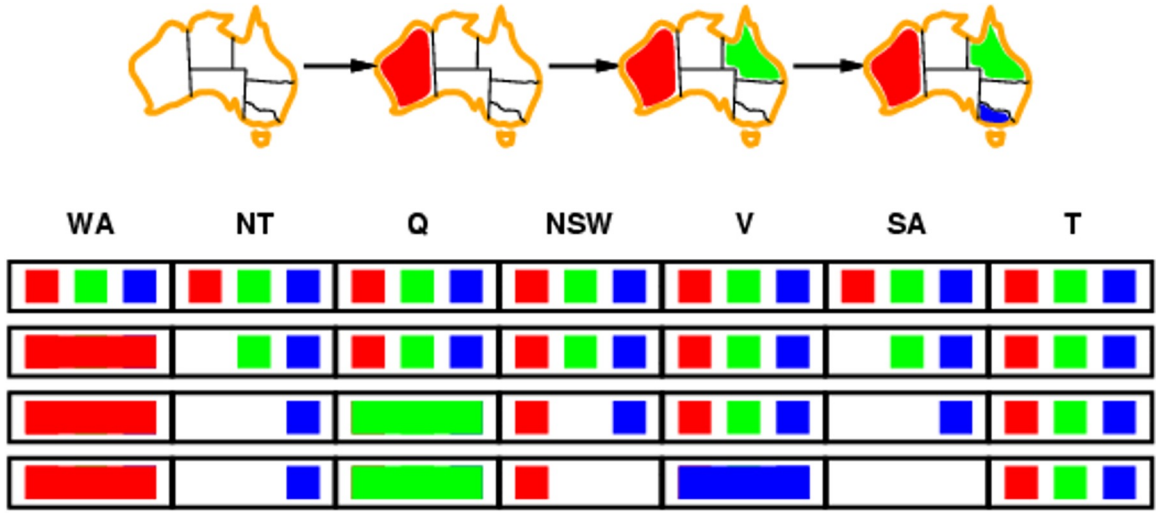
Can we improve the efficiency?

An observation: A>B, so if A=1, then B cannot be 2 or 3.

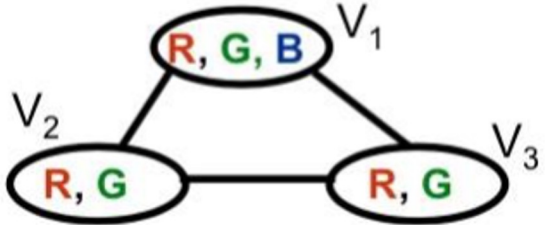
Backtracking with Forward Checking

- After selecting each assignment, remove any values of neighbouring domains that are inconsistent with the new assignment.
- Terminate search when any variable has no legal values.

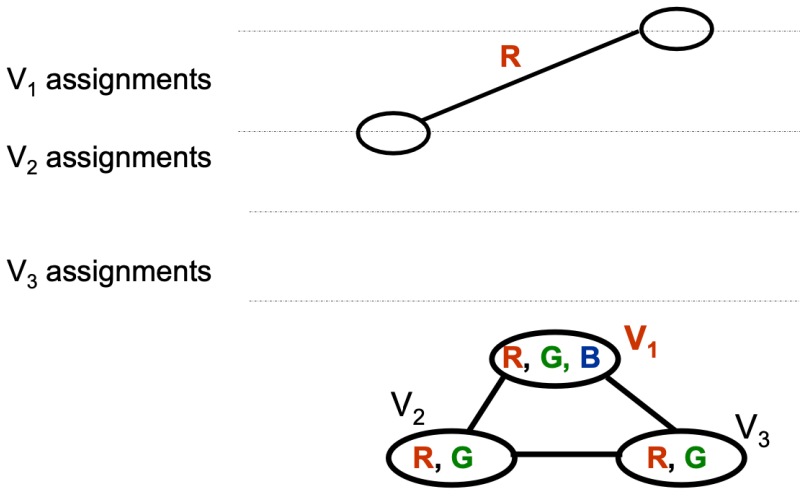




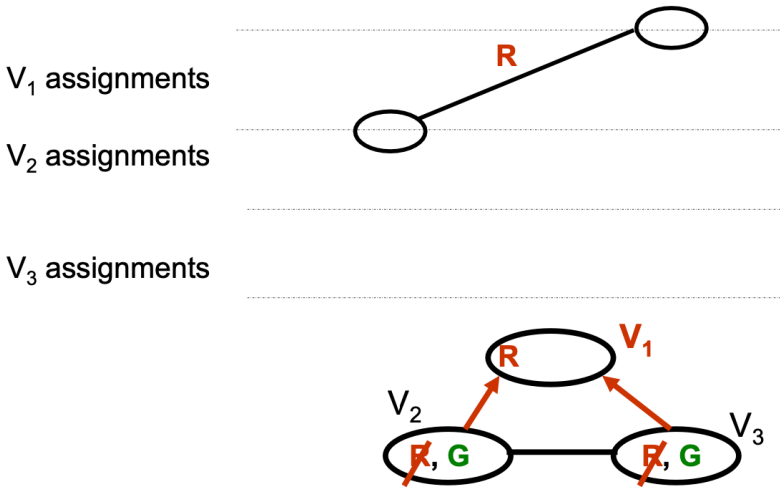
Backtracking with Forward Checking Example



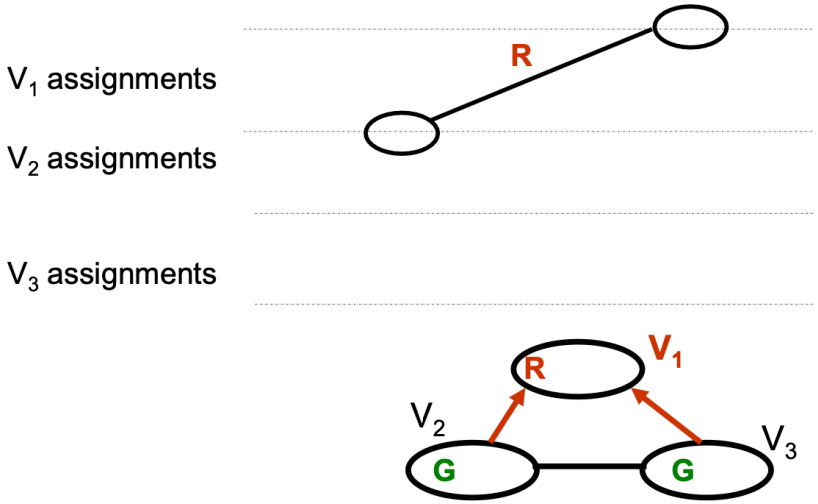
- Variables *V1, V2, V3*
- Domains D_1 = {R,G,B}; D_2 = {R,G}; D_3 = {R,G}
- Constraints: adjacent variables must have different colors



(1)

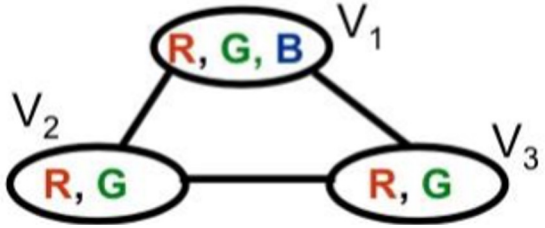


(2)

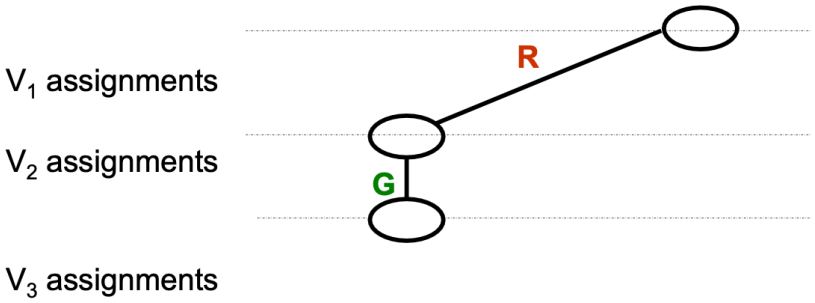


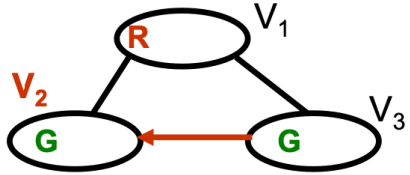
(3)

Backtracking with Forward Checking Example

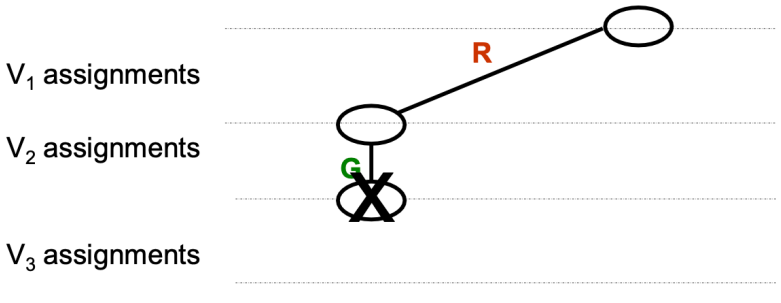


- Variables *V1, V2, V3*
- Domains D_1 = {R,G,B}; D_2 = {R,G}; D_3 = {R,G}
- Constraints: adjacent variables must have different colors



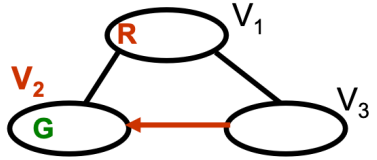


(4)

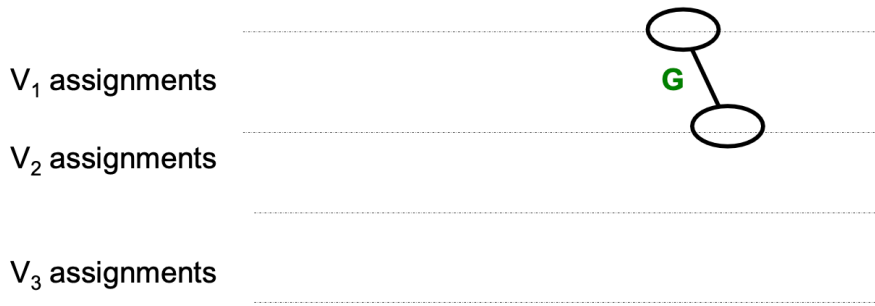


We have a conflict whenever a domain becomes empty.

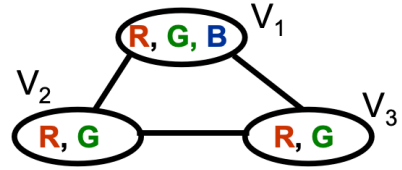
Back track



(5)

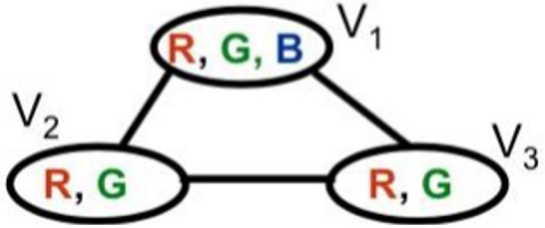


When backing up, restore the domain values.

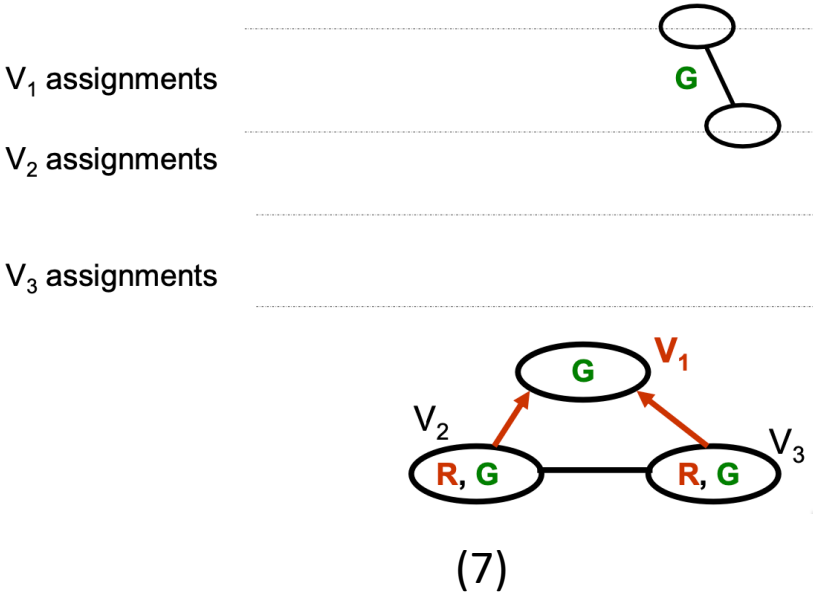


(6)

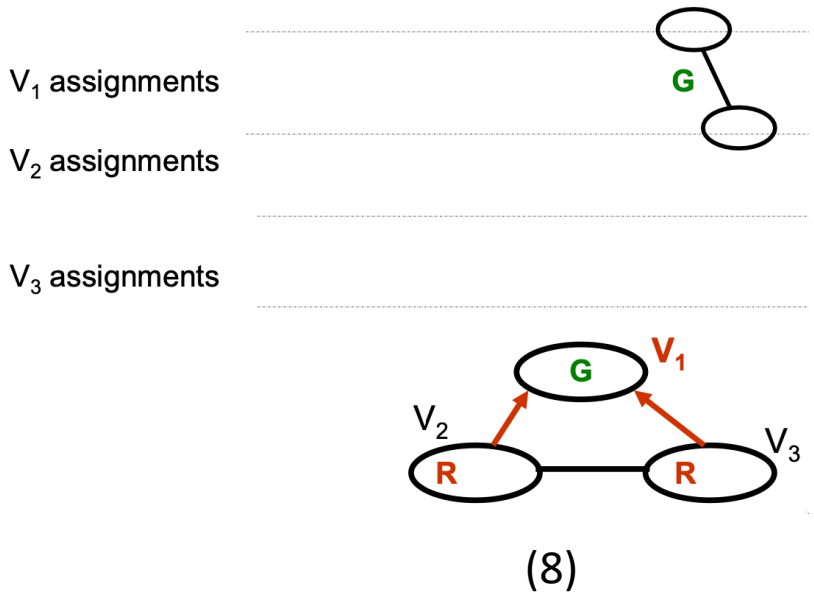
Backtracking with Forward Checking Example



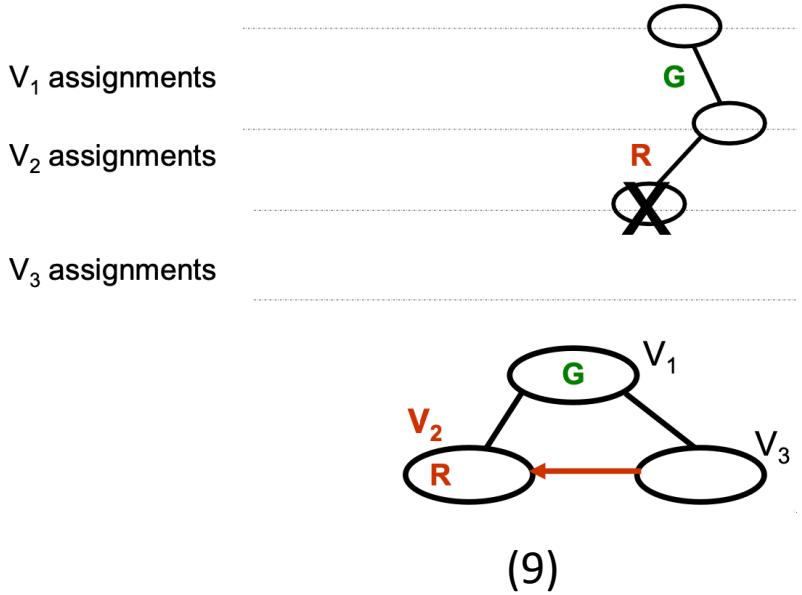
- Variables *V1, V2, V3*
- Domains D_1 = {R,G,B}; D_2 = {R,G}; D_3 = {R,G}
- Constraints: adjacent variables must have different colors



(7)

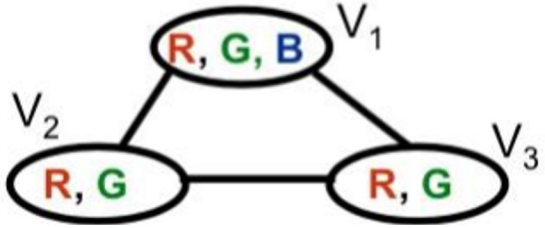


(8)

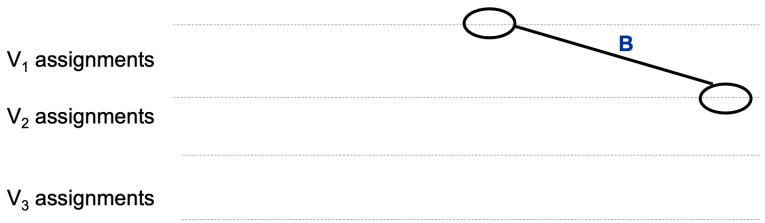


(9)

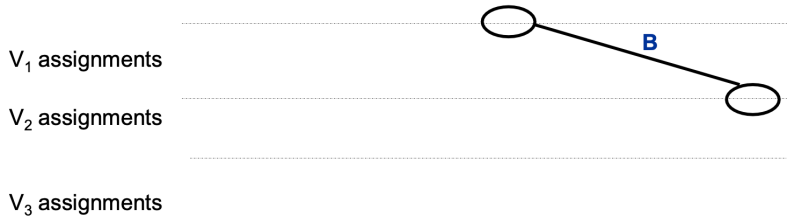
Backtracking with Forward Checking Example



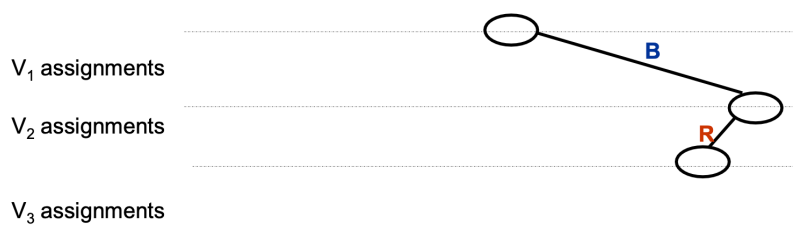
- Variables *V1, V2, V3*
- Domains D_1 = {R,G,B}; D_2 = {R,G}; D_3 = {R,G}
- Constraints: adjacent variables must have different colors



(10)

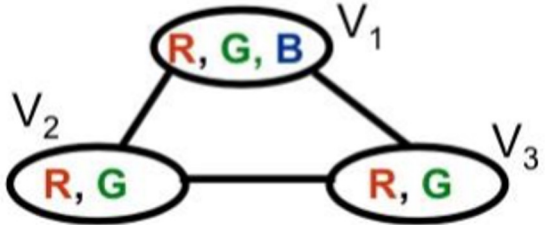


(11)

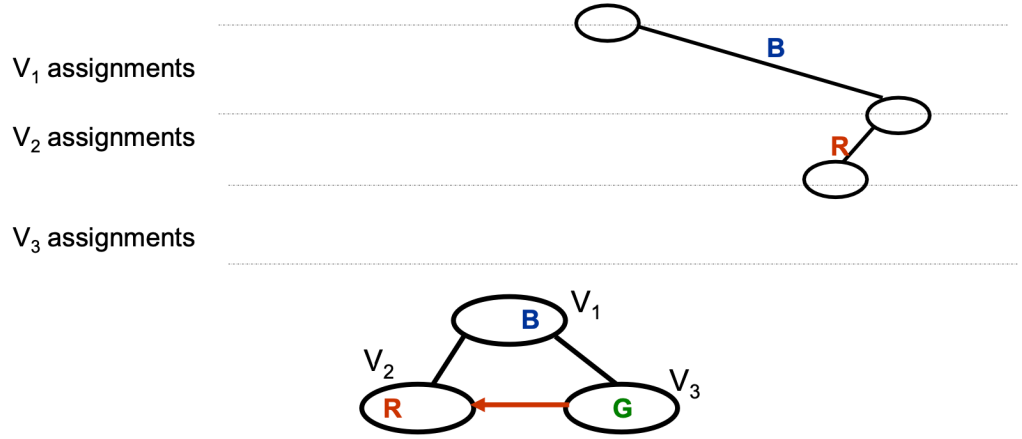


(12)

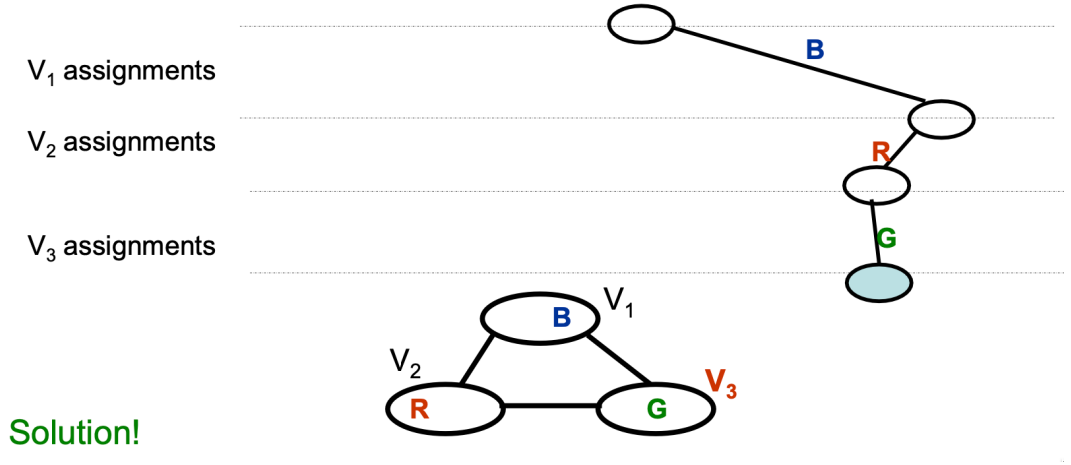
Backtracking with Forward Checking Example



- Variables *V1, V2, V3*
- Domains D_1 = {R,G,B}; D_2 = {R,G}; D_3 = {R,G}
- Constraints: adjacent variables must have different colors



(13)



(14)

Backtracking with Forward Checking Exercise

Solve this problem using backtracking with forward checking:

- $A \in \{1, 2, 3\}$
- $B \in \{1, 2, 3\}$
- $C \in \{1, 2, 3\}$

- Constraints:
  - $A > B$
  - $B \neq C$
  - $A \neq C$