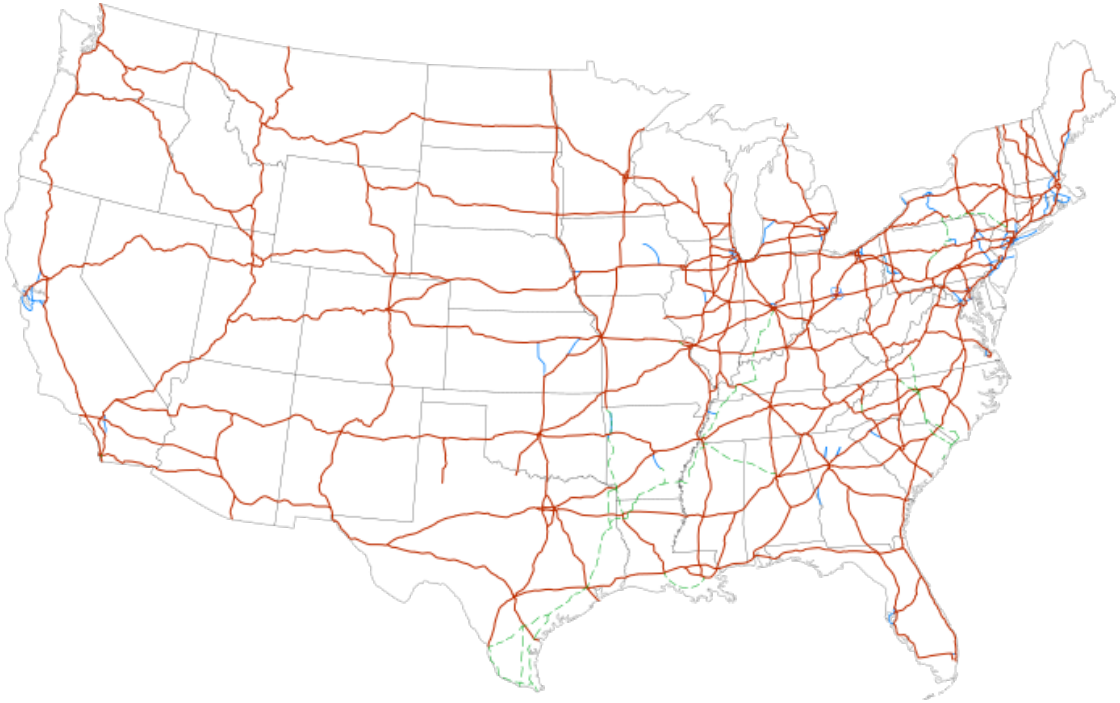
CS440/ECE 448 Lecture 2: Breadth-First Search

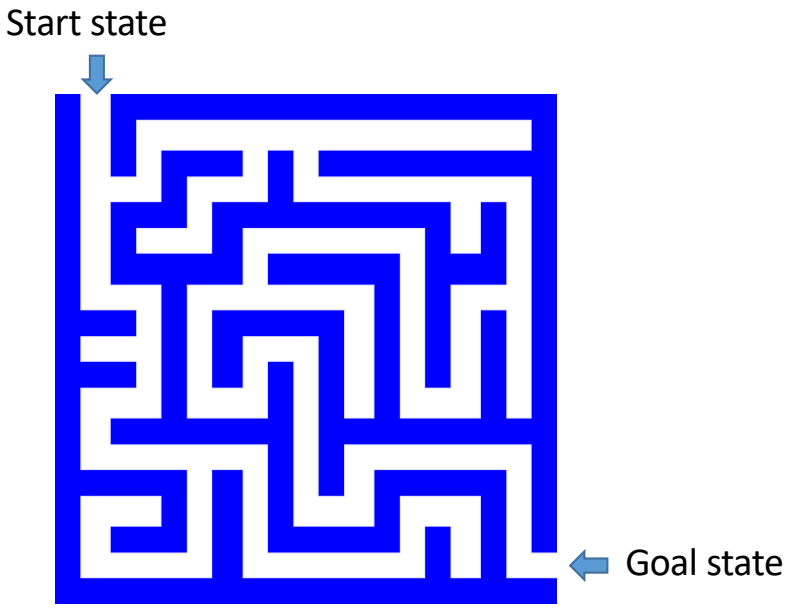


Outline of today's lecture

1. Initial state, goal state, transition model
2. General algorithm for solving search problems
    1. First data structure: a frontier queue
    2. Second data structure: a search tree
    3. Third data structure: a "visited states" dict
3. Breadth-first search (BFS) and Depth-first search (DFS)
4. Minimum spanning tree (MST)

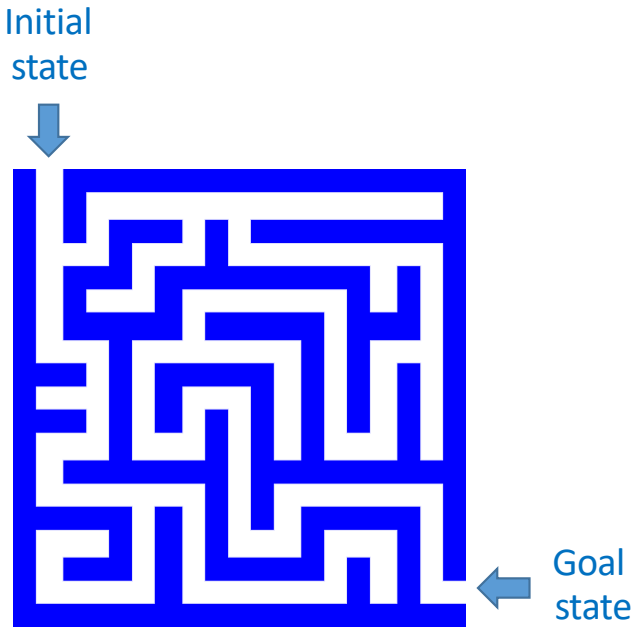
Search

- We will consider the problem of designing **goal-based agents** in **fully observable**, **deterministic**, **discrete**, **static**, **known** environments
- Environment is **sequential**: agent's action changes its state
- Agent must plan the best sequence of actions to achieve a goal



Search problem components

- **Initial state**
- **Actions**
- **Transition model**
  - What state results from performing a given action in a given state?
- **Goal state**
- **Path cost**
  - Assume that it is a sum of nonnegative *step costs*



- The **optimal solution** is the sequence of actions that gives the *lowest* path cost for reaching the goal

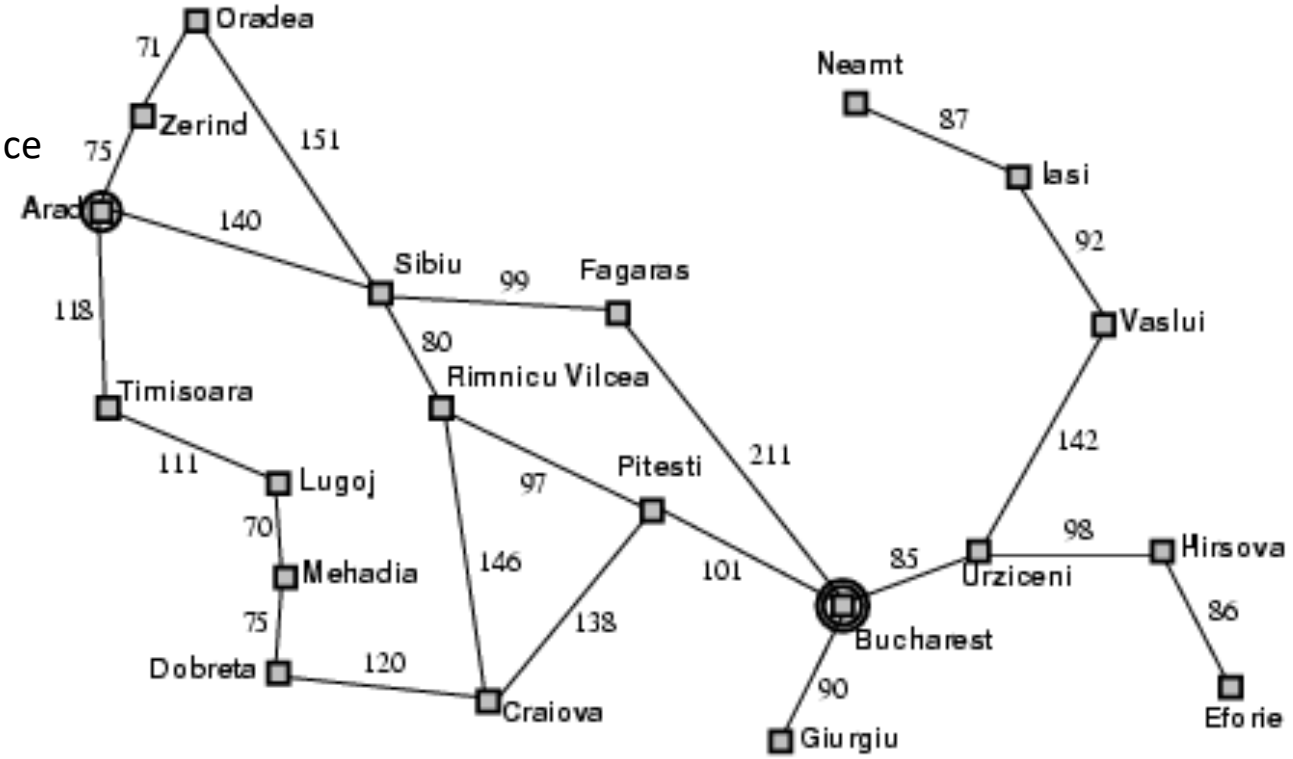
Knowledge Representation: State

- State = description of the world
  - Must have enough detail to decide whether or not you're currently in the **initial state**
  - Must have enough detail to decide whether or not you've reached the **goal state**
  - Often but not always: "defining the state" and "defining the transition model" are the same thing

Example of state definition: Romania

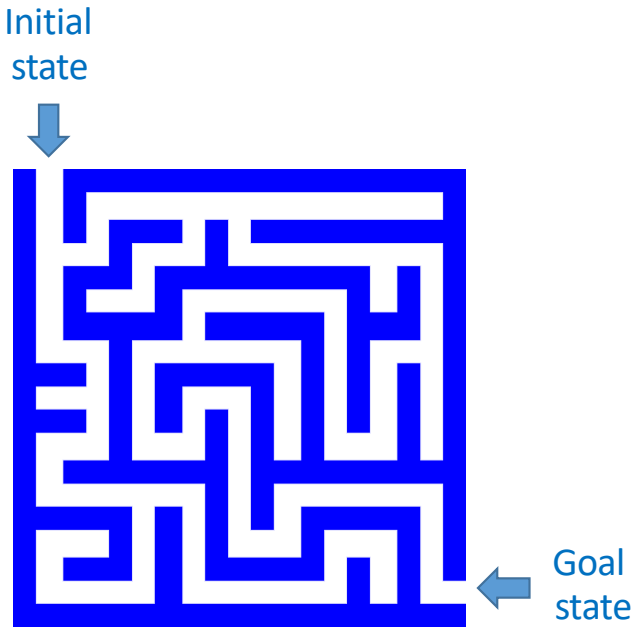
- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

- **state** = name of the city
- **Path cost**
  - Sum of edge costs (total distance traveled)



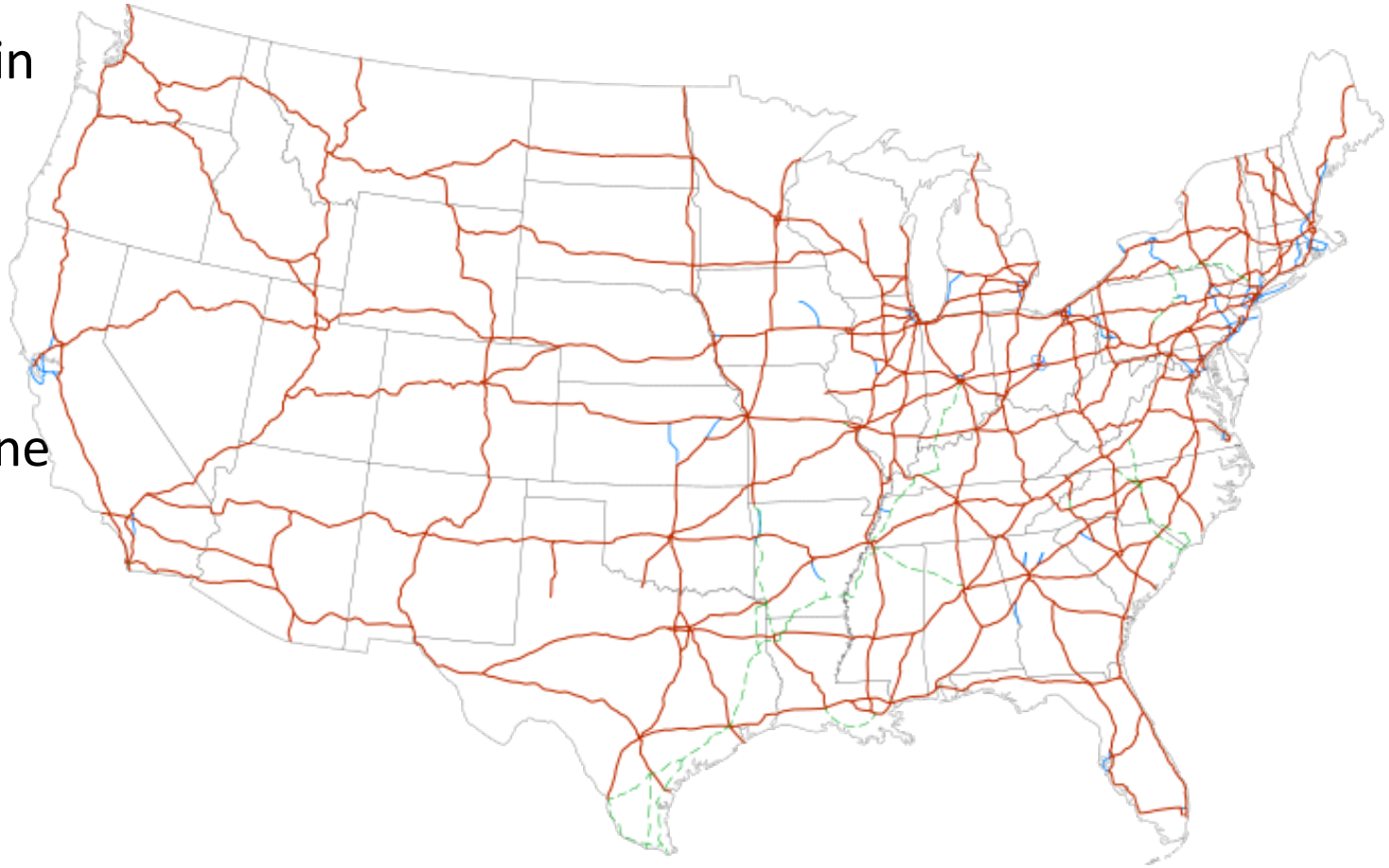
Example of state definition: Maze solving

- **State = (x,y)**, current position of the agent



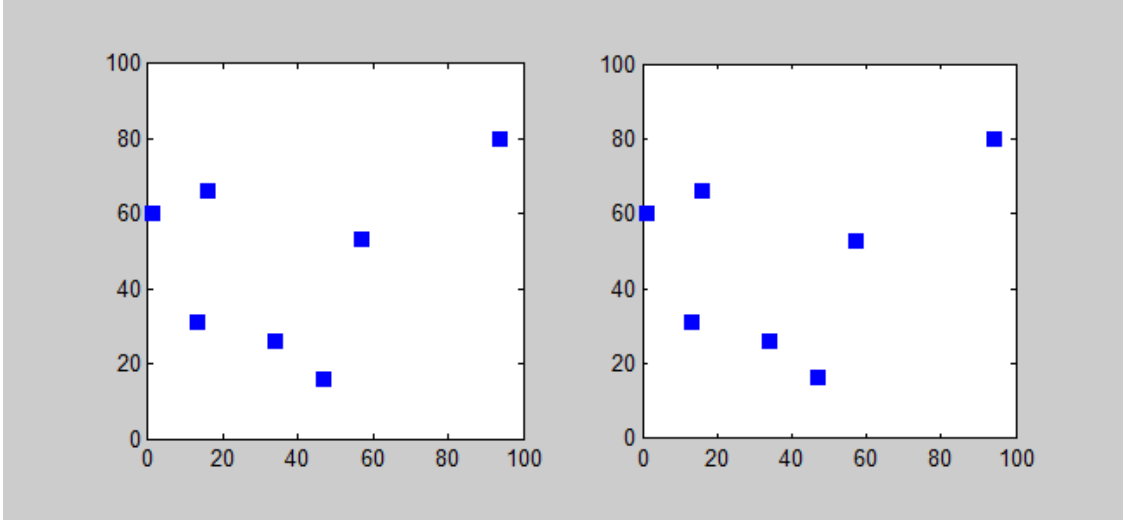
Example of state definition: Traveling salesman problem

- Goal: visit every city in the United States
- Path cost: total miles traveled
- Initial state: Champaign, IL
- Action: travel from one city to another
- Transition model: when you visit a city, mark it as “visited.”



Example of state definition: Traveling salesman problem

- **state = (agent, goals)**
  - **agent = (agent_x,agent_y)** is current position of the agent
  - **goals = [goal[0], goal[1], …]** lists the goals that have not yet been reached
    - **goal[i] = (goal_x, goal_y)** tells the location of the i'th remaining goal



Solving TSP using a branch-and-bound algorithm. CC-BY-SA 3.0, Saurabh Harsh, 2012, https://commons.wikimedia.org/wiki/File:Branchbound.gif

Outline of today's lecture

1. Initial state, goal state, transition model
2. General algorithm for solving search problems
    1. First data structure: a frontier queue
    2. Second data structure: a search tree
    3. Third data structure: a "visited states" dict
3. Breadth-first search (BFS) and Depth-first search (DFS)
4. Minimum spanning tree (MST)

How does this problem differ from every problem you've ever seen before?

- Search differs from most Computer Science problems in that the state space might be infinite. We don't assume, in advance, that we can enumerate every possible configuration of the world.
- Traditional definition of Dijkstra's algorithm:
  - First, list all of the possible states in the "not explored" list
  - Then, move them to the "explored" list after we visit them
- Modifying Dijkstra's algorithm for the infinite-world assumption:
  - Instead of a list of all possible states, we have a method **(next_state,cost)=Transition_Model(current_state, action)**
  - Instead of an infinite "not explored" list, we have a finite "frontier."

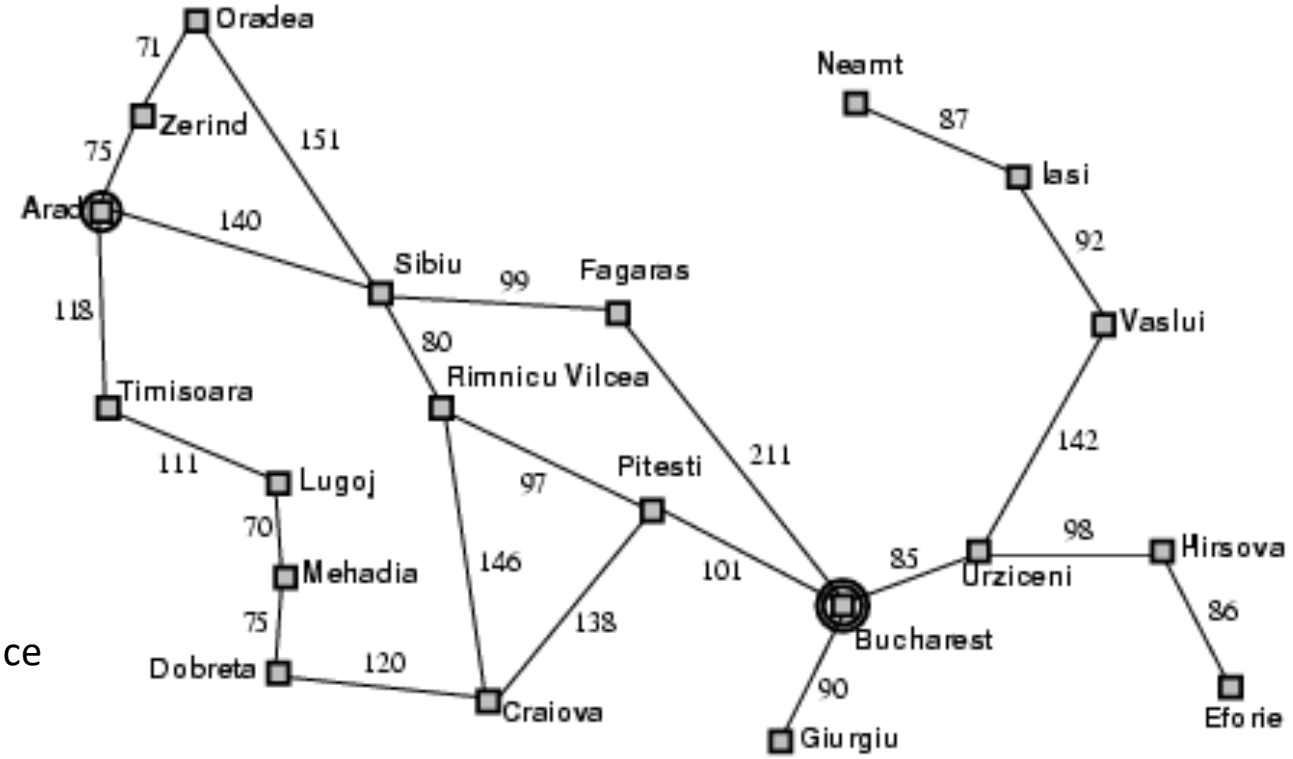
First data structure: Frontier

- Frontier = set of nodes that you know how to reach, but you haven't yet tested to see what comes next after those states
- **node = (state, parent_node, path_cost)**
- Initialize: **frontier = { (initial_state, None, 0) }**
- Iterate, until goal is reached:
  - Set **current_state** to some node from the frontier, remove it from the frontier.
  - Expand **current_state**:
    - If it's the goal, then you're done! Return the corresponding path.
    - If not, then find its children, and transition costs, using **(next_state,cost)=Transition_Model(current_state, action)**, and add them to the frontier.

Example: Romania

- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

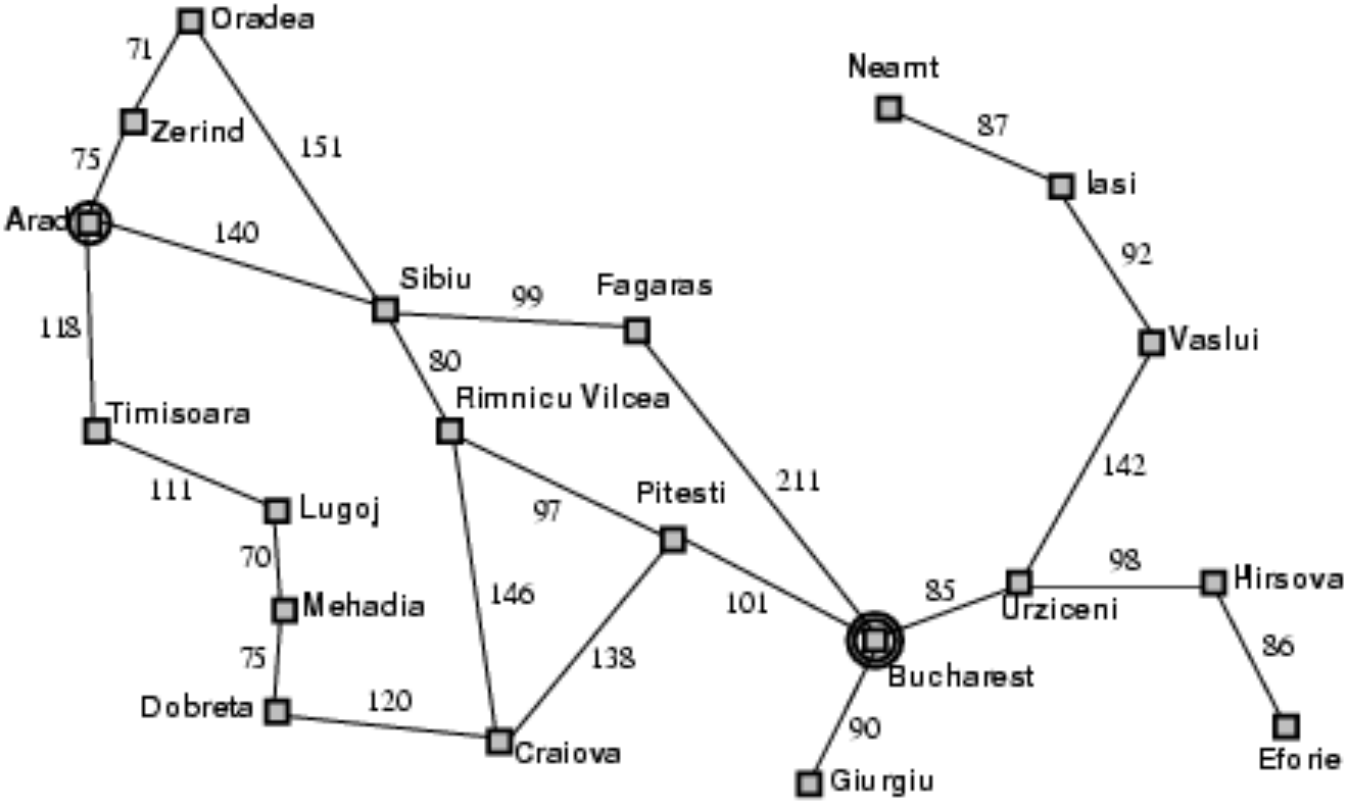
- **Initial state**
  - Arad
- **Actions**
  - Go from one city to another
- **Transition model**
  - If you go from city A to city B, you end up in city B
- **Goal state**
  - Bucharest
- **Path cost**
  - Sum of edge costs (total distance traveled)



Search step 0

Frontier: { Arad }

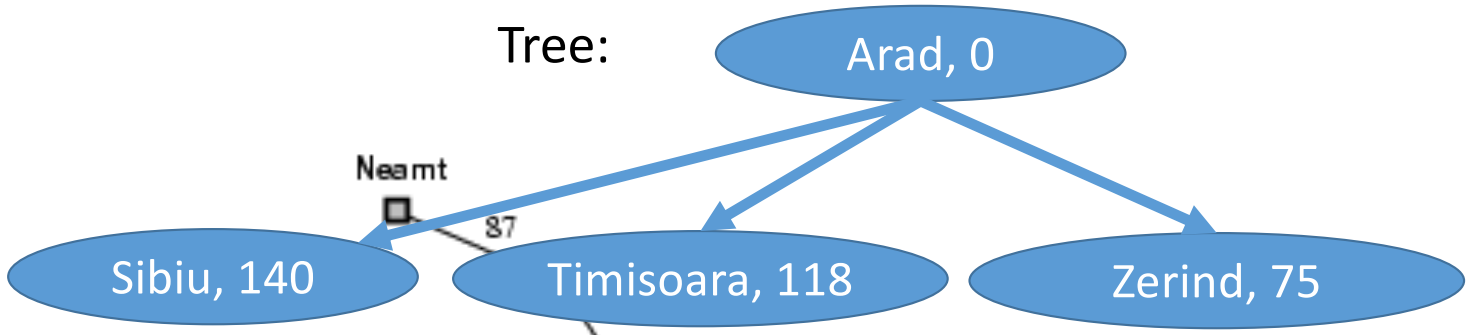
Tree: Arad, 0

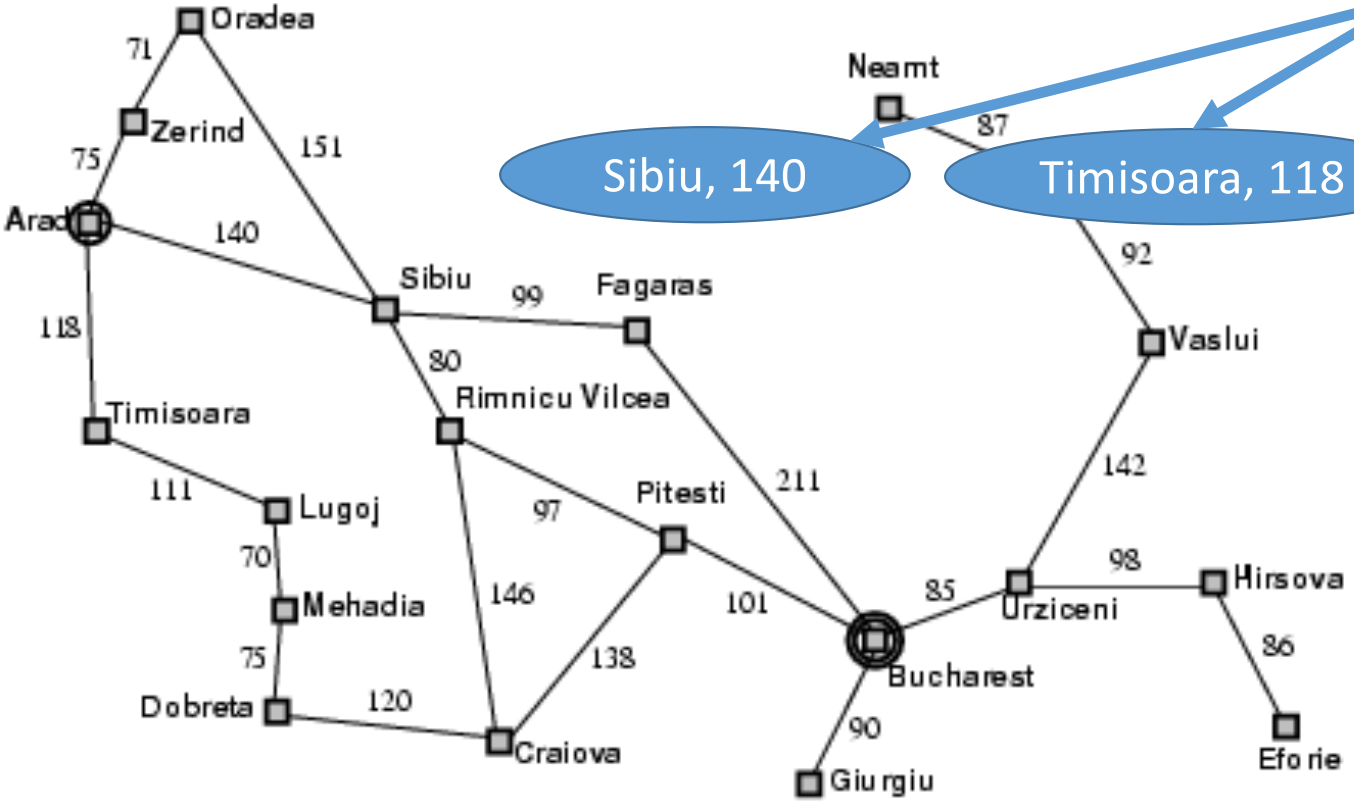


Search step 1

Frontier: { Sibiu, Zerind, Timisoara }

Tree:



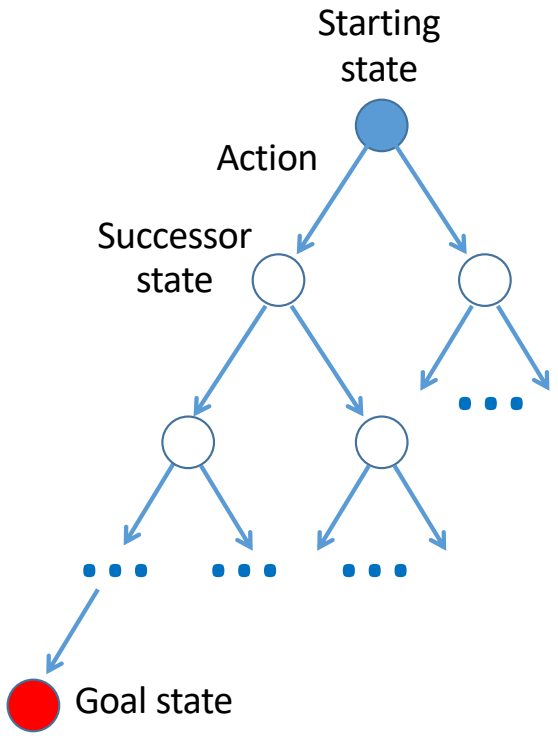


Tree Search: Basic idea

1. SEARCH for an optimal solution
   - Maintain a **frontier** of unexpanded states
   - At each step, pick a state from the frontier to **expand:**
     - Check to see whether or not this state is the goal state. If so, DONE!
     - If not, then list all of the states you can reach from this state, add them to the frontier, and add them to the tree
2. BACK-TRACE: go back up the tree; list, in reverse order, all of the actions you need to perform in order to reach the goal state.
3. ACT: the agent reads off the sequence of necessary actions, in order, and does them.

Nodes vs. States

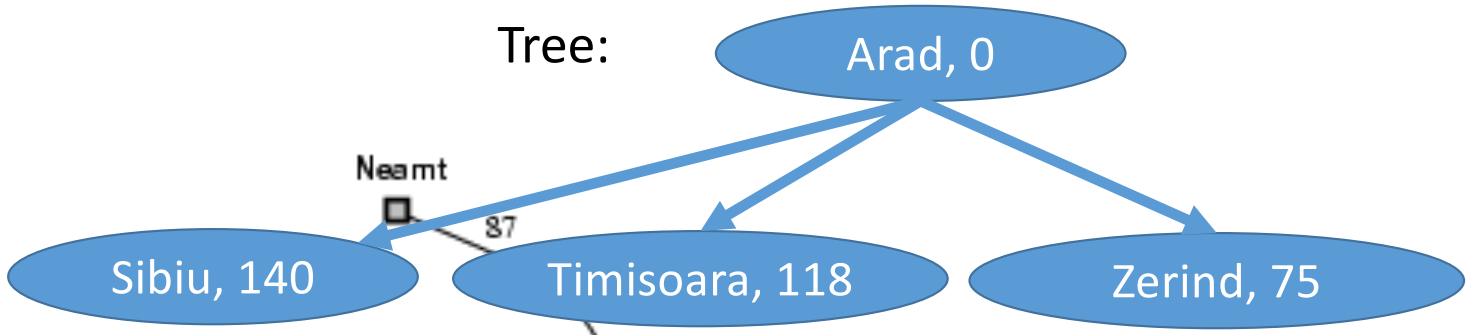
- State = description of the world
  - Must have enough detail to decide whether or not you're currently in the initial state
  - Must have enough detail to decide whether or not you've reached the goal state
  - Often but not always: "defining the state" and "defining the transition model" are the same thing
- Node = a point in the search tree
  - Knows the ID of its STATE
  - Knows the ID of its PARENT NODE
  - Knows the COST of the path

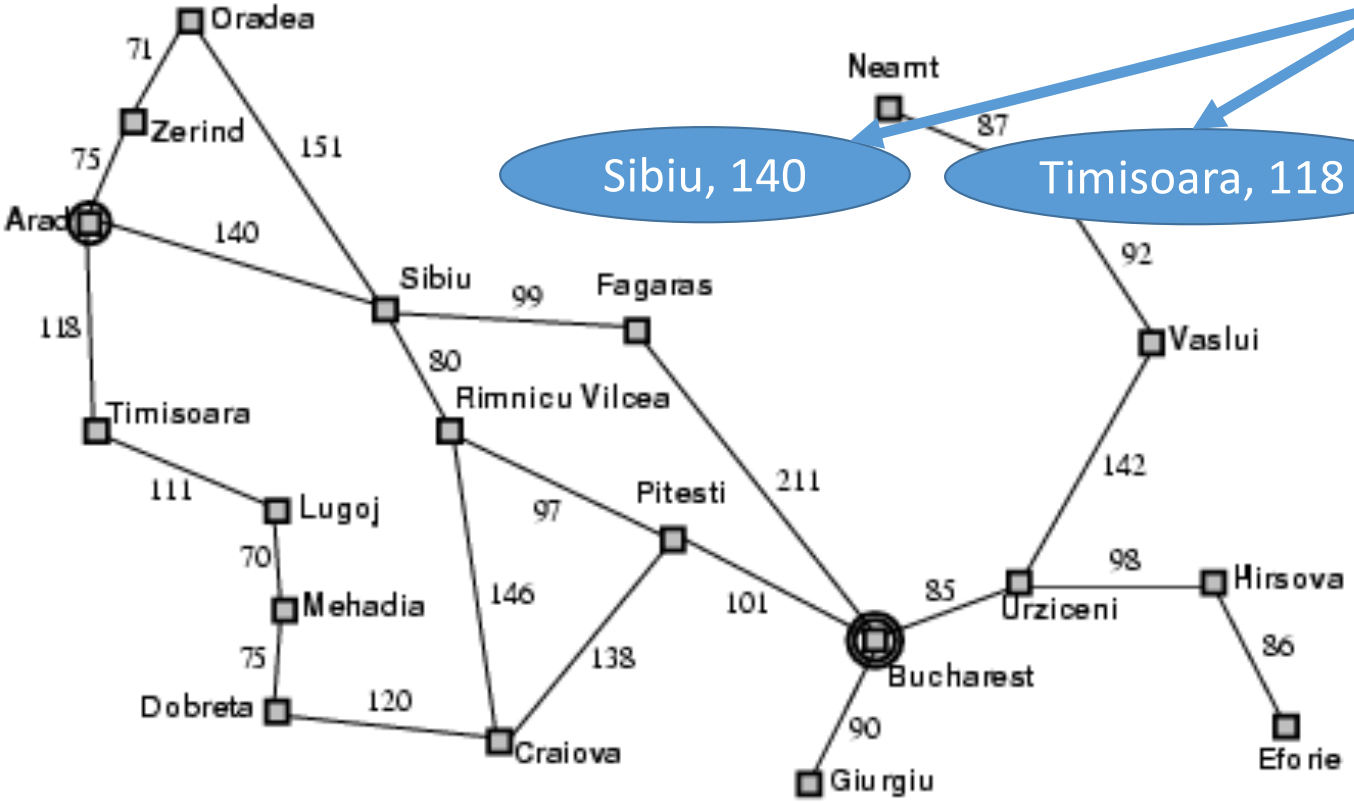


Search step 1

Frontier: { Sibiu, Zerind, Timisoara }

Tree:

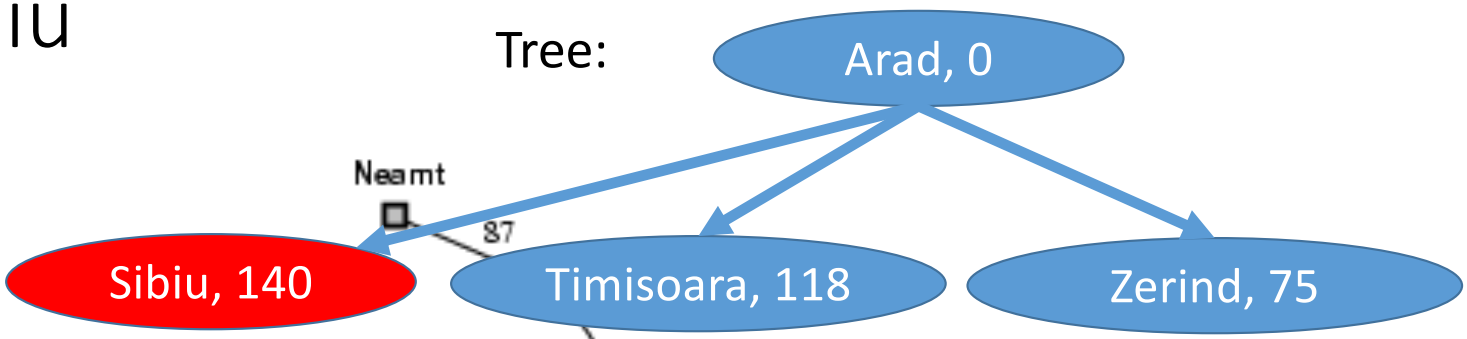


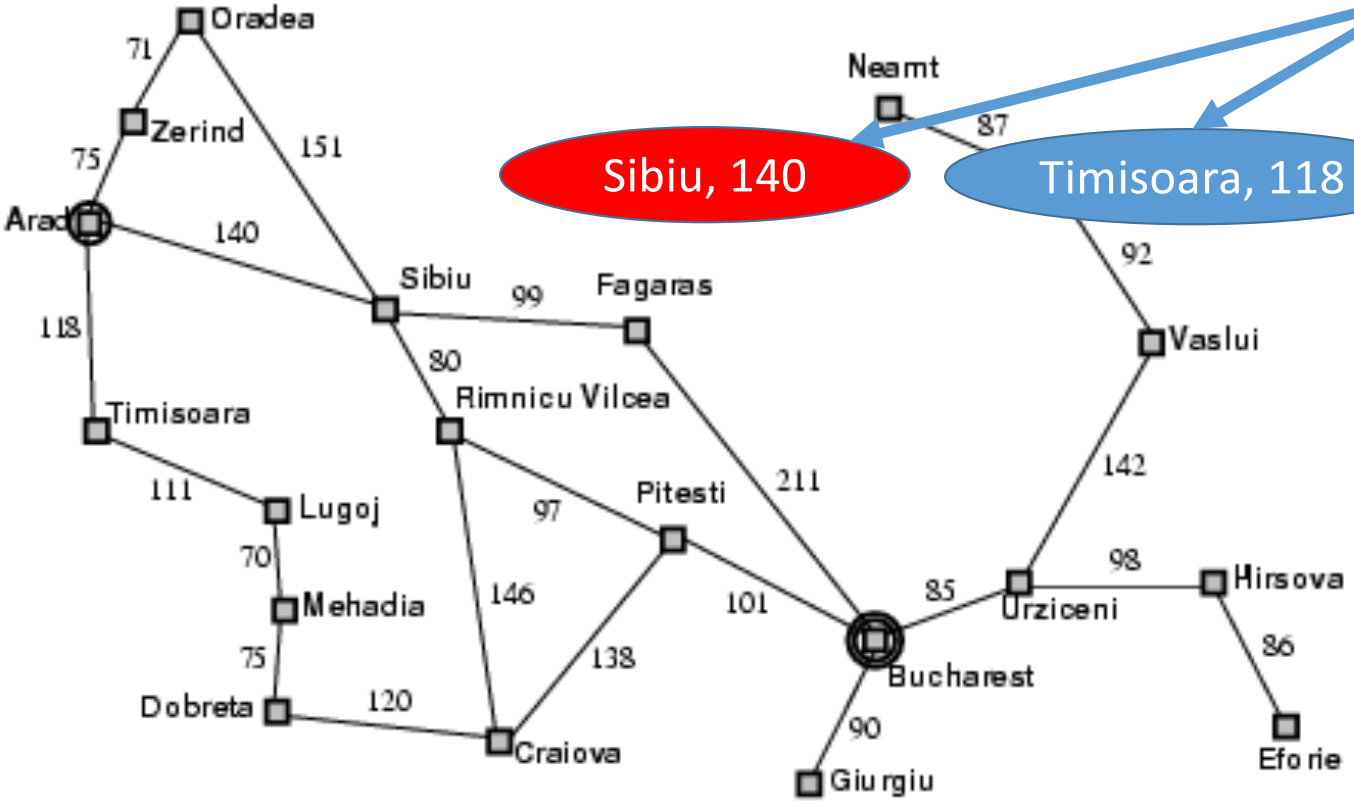


Search step 2
Expand Sibiu

Frontier: { Sibiu, Zerind, Timisoara }

Tree:

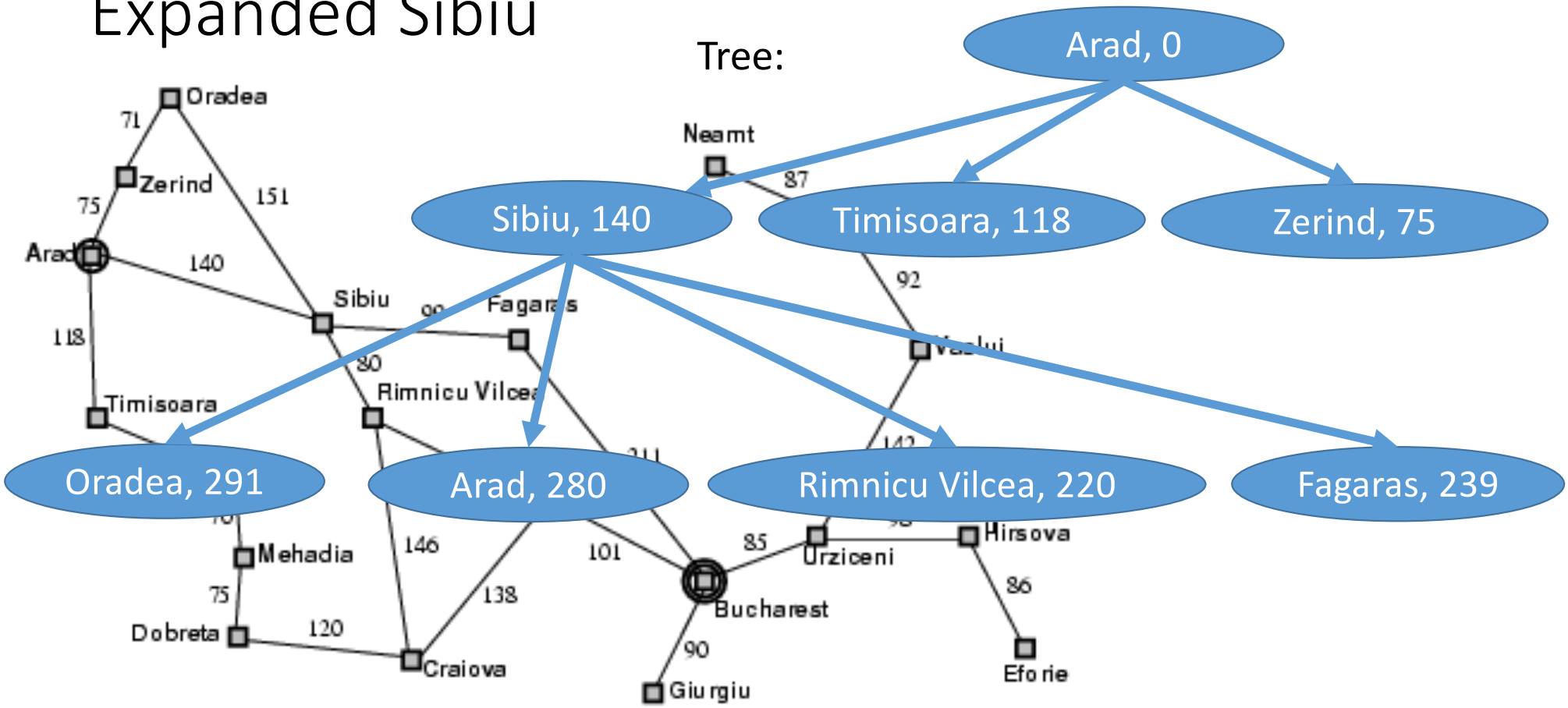




Search step 2
Expanded Sibiu

Frontier: { Zerind, Timisoara, Oradea, Arad, Rimnicu Vilcea, Fagaras }

Tree:



Tree Search: Computational Complexity

Without an EXPLORED set

- b = “branching factor” = max # states you can reach from any given state
- d = “depth” = # layers in the tree (# moves that you have made)
- Without an explored set: complexity = O{b^d}

Solution: keep track of the states you have explored

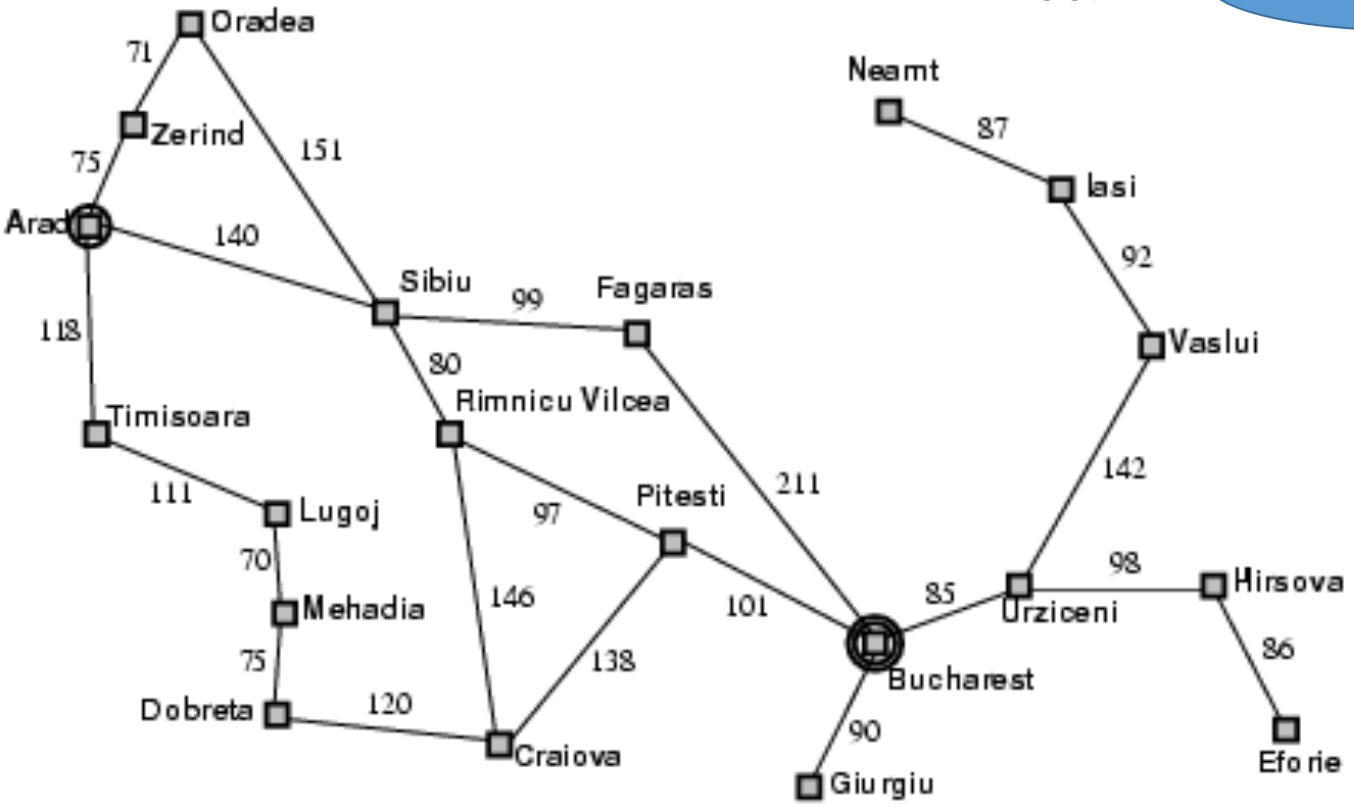
- When you expand a state, you get the list of its possible child states
- ONLY IF a child state is not already explored, put it on the frontier, and put it on the explored set.
- Result: complexity = min(O{b^d}, O{# possible world states})

Search step 0

Frontier: { Arad }

Explored: { Arad }

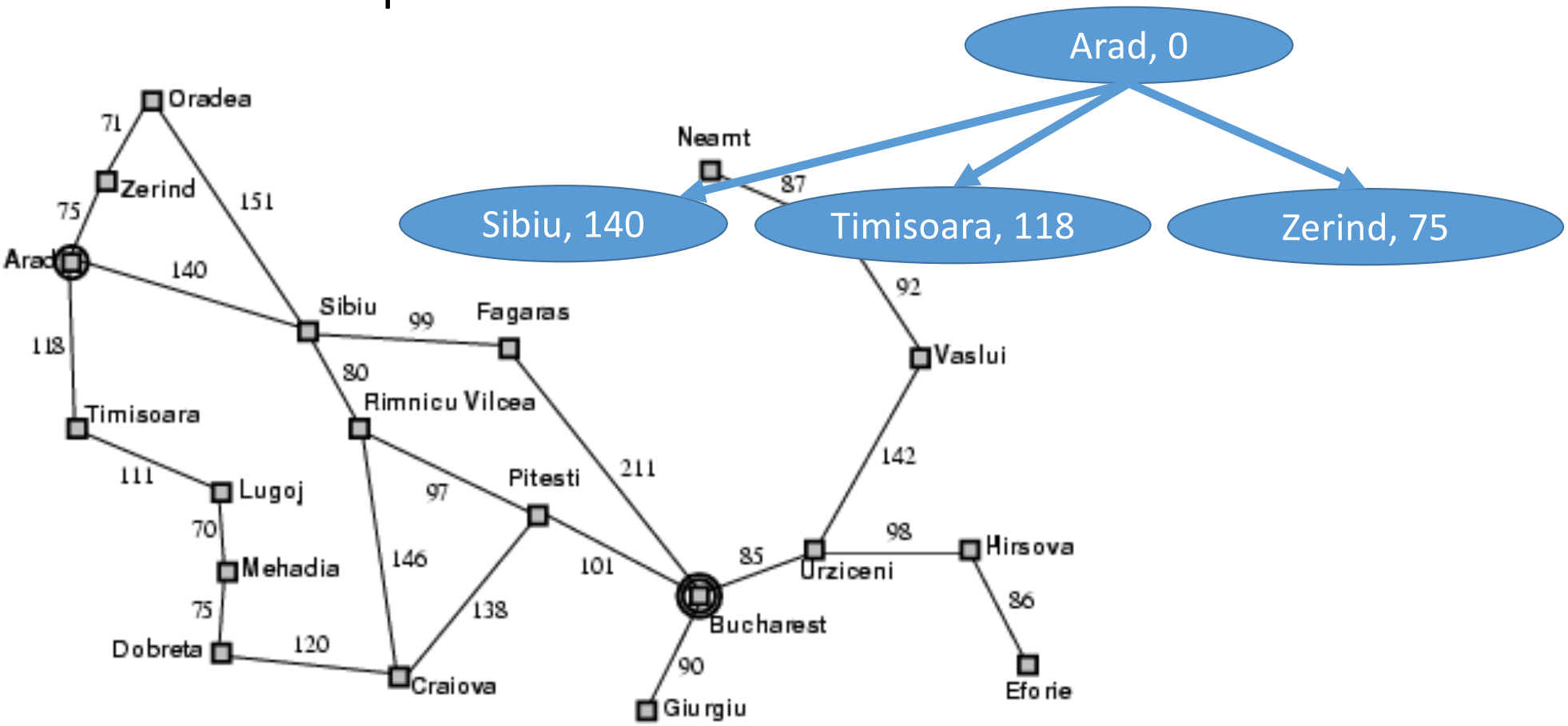
Tree: Arad, 0



Search step 1

Frontier: { Sibiu, Zerind, Timisoara }

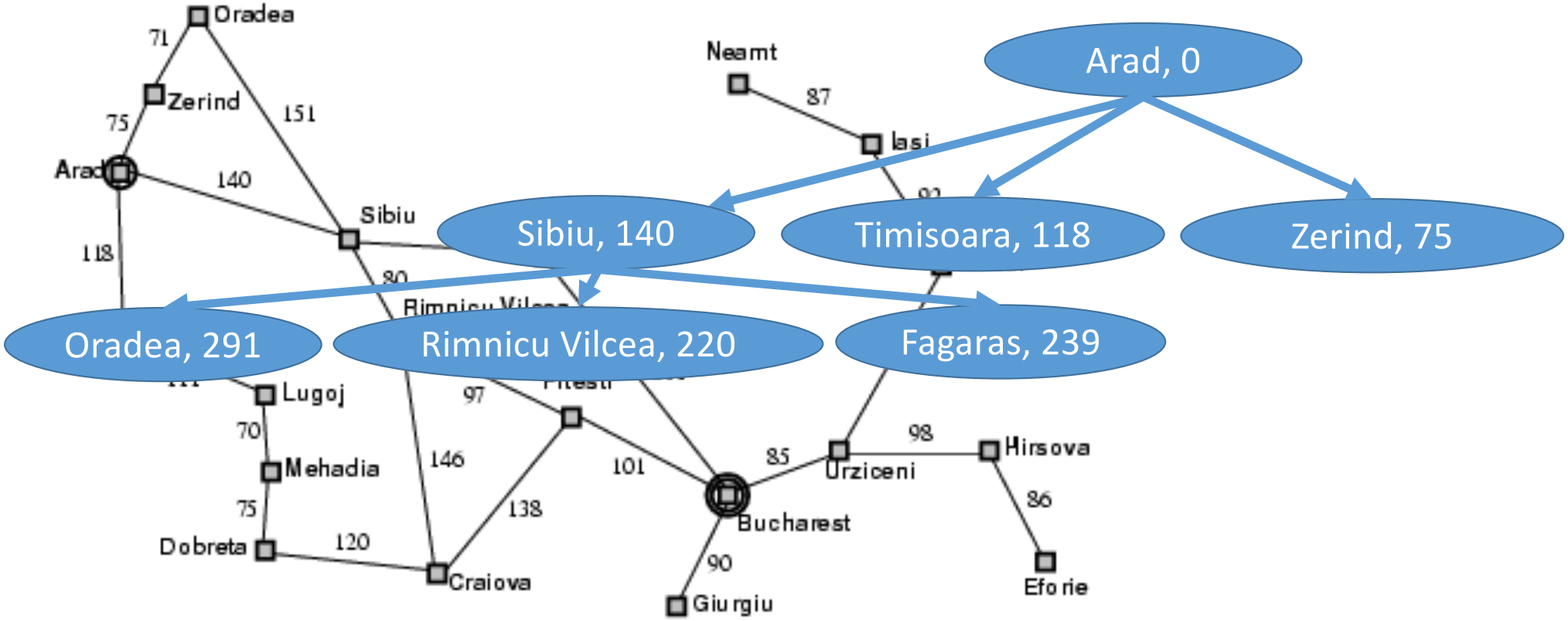
Explored: { Arad, Sibiu, Zerind, Timisoara }



Search step 2

Frontier: { Zerind, Timisoara, Oradea, Rimnicu Vilcea, Fagaras }

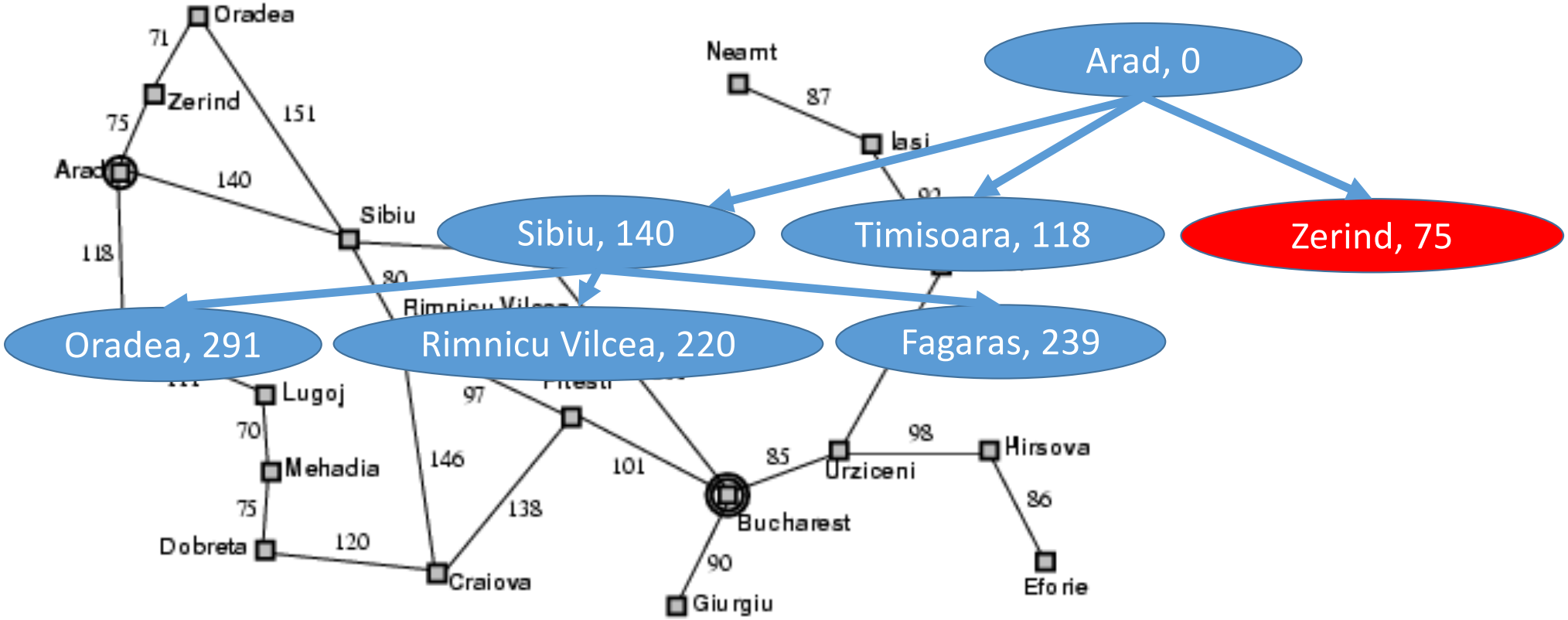
Explored: { Arad, Sibiu, Zerind, Timisoara, Oradea, Rimnicu Vilcea, Fagaras }



Search step 3: expand Zerind

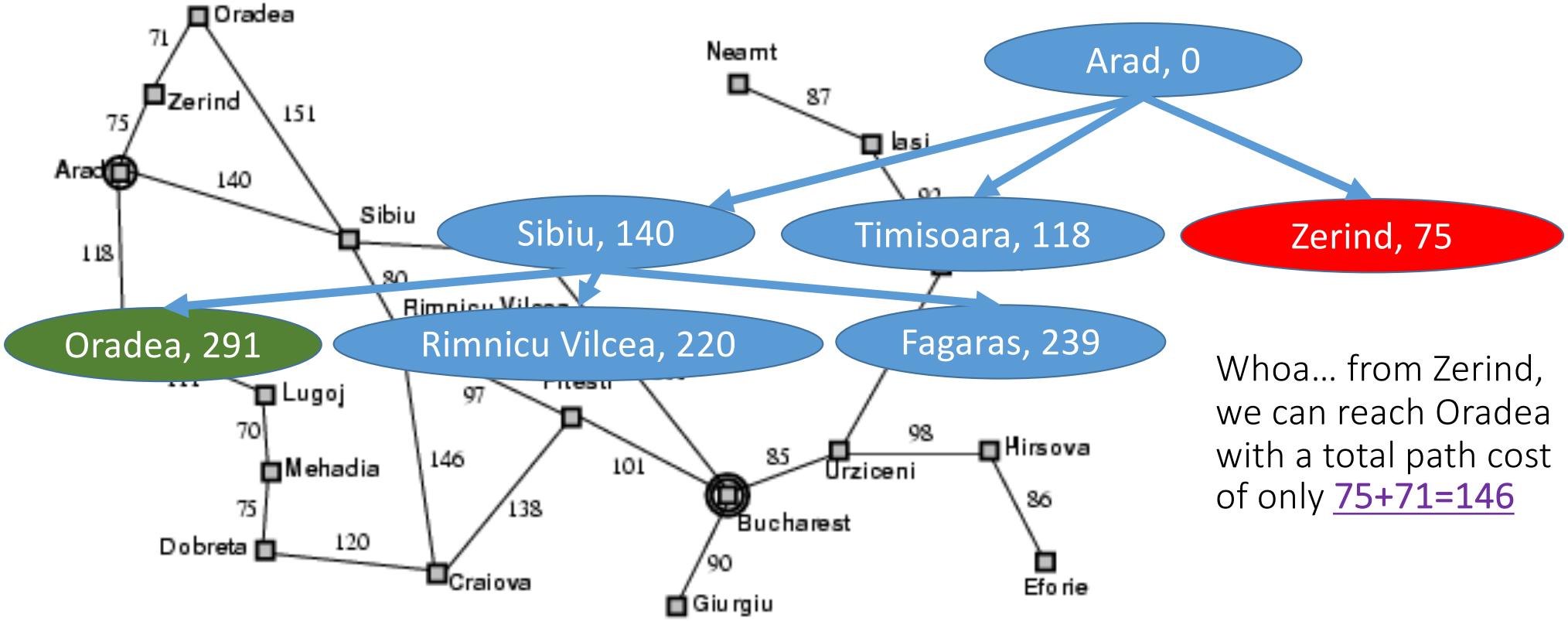
Frontier: { **Zerind**, Timisoara, Oradea, Rimnicu Vilcea, Fagaras }

Explored: { Arad, Sibiu, Zerind, Timisoara, Oradea, Rimnicu Vilcea, Fagaras }



Frontier: { **Zerind**, Timisoara, Oradea, Rimnicu Vilcea, Fagaras }

Explored: { Arad, Sibiu, Zerind, Timisoara, Oradea, Rimnicu Vilcea, Fagaras }



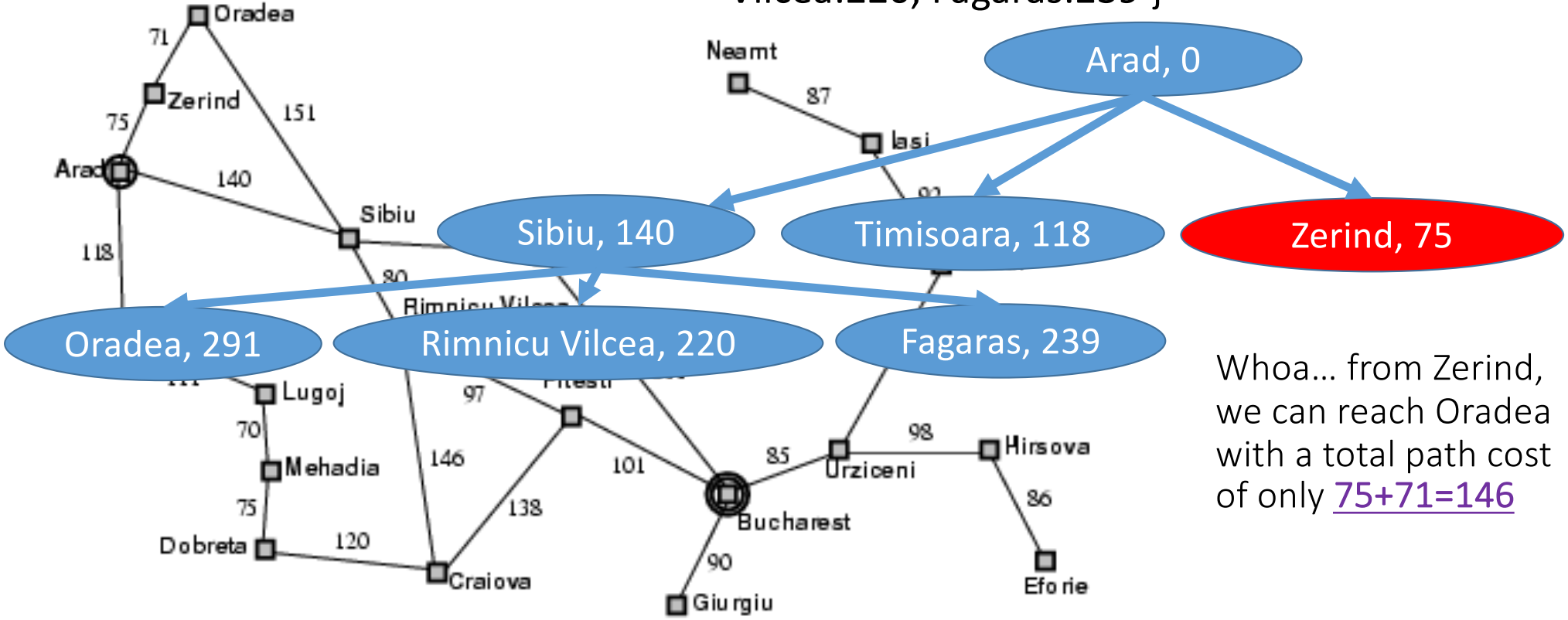
Whoa… from Zerind, we can reach Oradea with a total path cost of only 75+71=146

Third data structure: Explored Dictionary

- Explored = dictionary mapping from state ID to path cost
- If we find a new path to the same state, with HIGHER COST, then we ignore it
- If we find a new path to the same state, with LOWER COST, then we expand the new path

Frontier: { **Zerind:75**, Timisoara:118, Oradea:291, Rimnicu Vilcea:220, Fagaras:239 }

Explored: { Arad:0, Sibiu:140, Zerind:75, Timisoara:118, Oradea:291, Rimnicu Vilcea:220, Fagaras:239 }

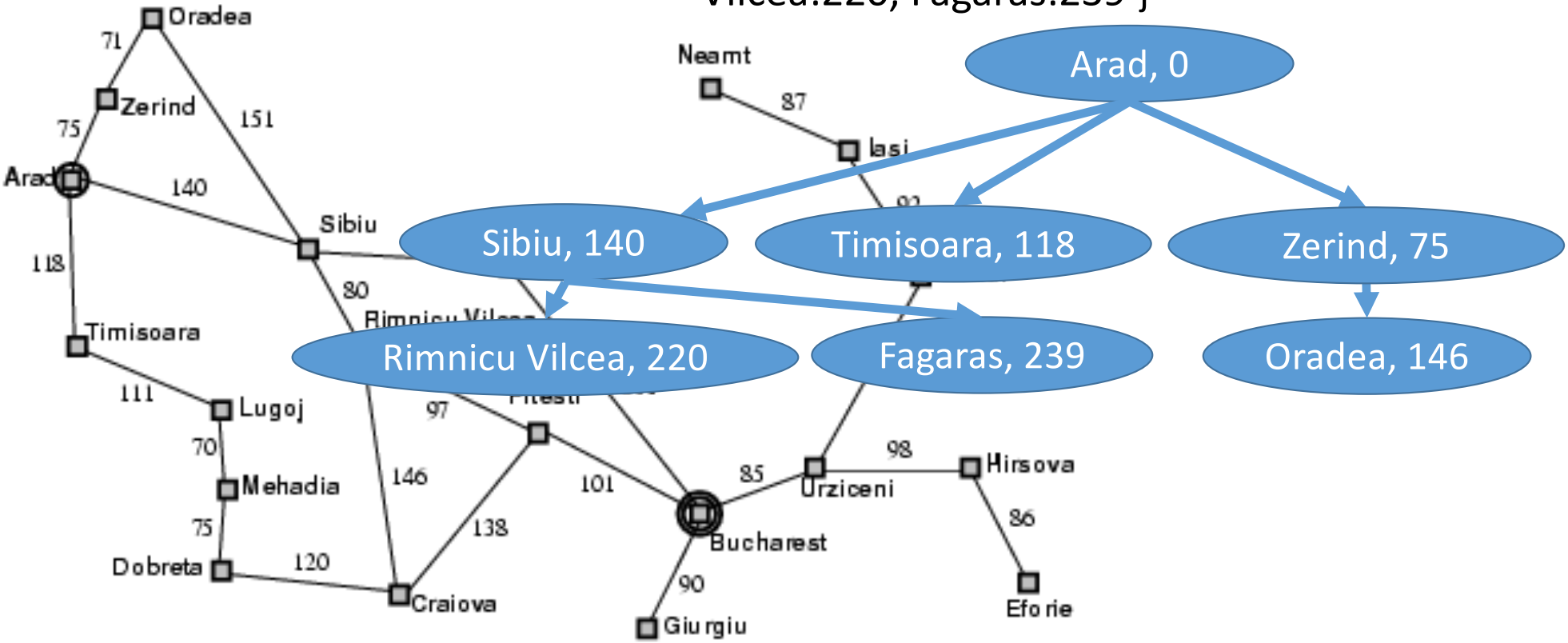


Whoa… from Zerind, we can reach Oradea with a total path cost of only 75+71=146

Search step 3: expanded Zerind

Frontier: { Timisoara:118, Oradea:**146**, Rimnicu Vilcea:220, Fagaras:239 }

Explored: { Arad:0, Sibiu:140, Zerind:75, Timisoara:118, Oradea:**146**, Rimnicu Vilcea:220, Fagaras:239 }



Outline of today's lecture

1. Initial state, goal state, transition model
2. General algorithm for solving search problems
    1. First data structure: a frontier queue
    2. Second data structure: a search tree
    3. Third data structure: a "visited states" dict
3. Breadth-first search (BFS) and Depth-first search (DFS)
4. Minimum spanning tree (MST)

In which order should you pick nodes from the frontier?

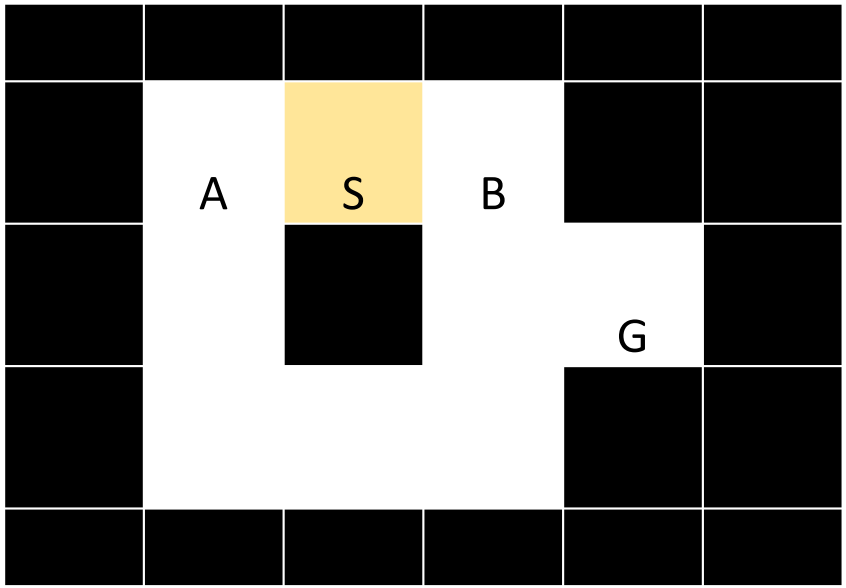
- FIFO (first-in, first-out):
  - the next node you expand will always be the one **least** recently added to the frontier.
  - corresponds to a breadth-first search (BFS), i.e., the same order that Dijkstra's algorithm would use
- LIFO (last-in, first-out):
  - the next node you expand will always be the one **most** recently added to the frontier.
  - corresponds to a depth-first search (DFS)

BFS: Frontier = FIFO

Example:

explored = { S }

frontier = { A, B }

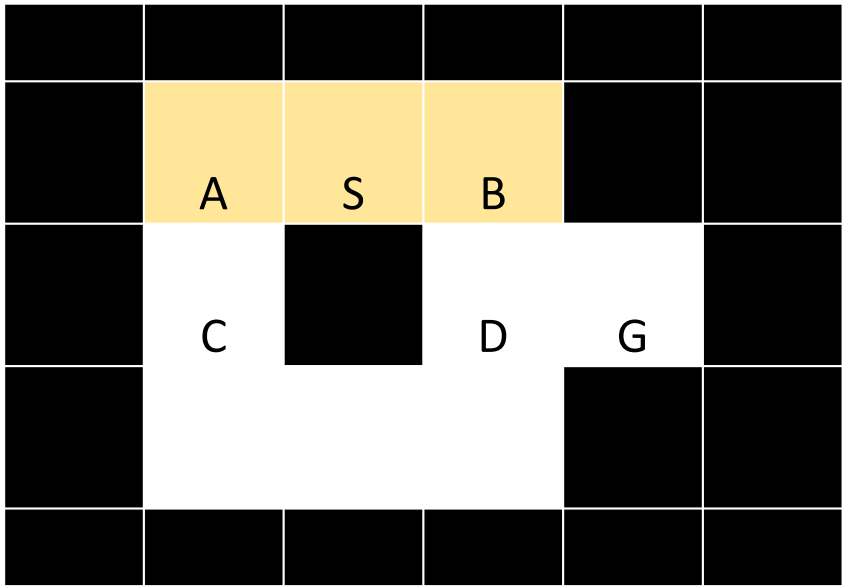


BFS: Frontier = FIFO

Example:

explored = { S, A, B }

frontier = { C, E }

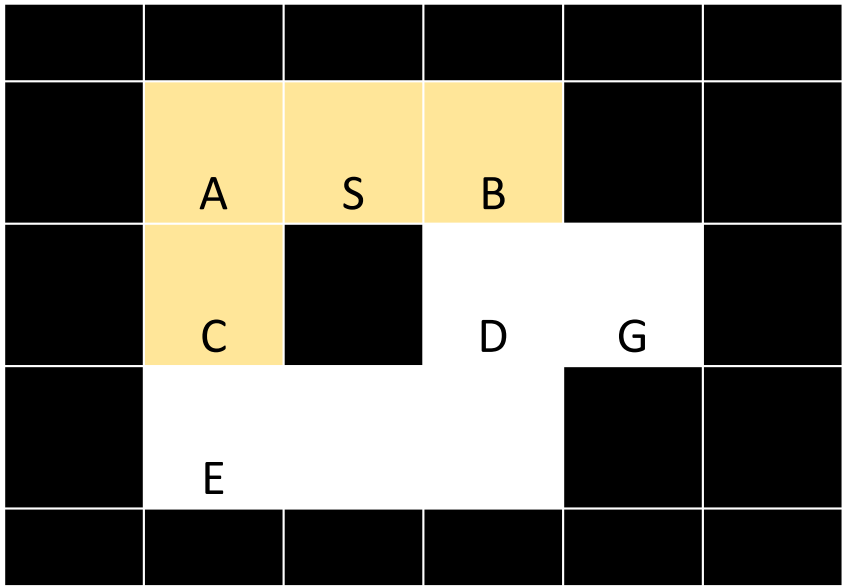


BFS: Frontier = FIFO

Example:

explored = { S, A, B, C }

frontier = { D, E }

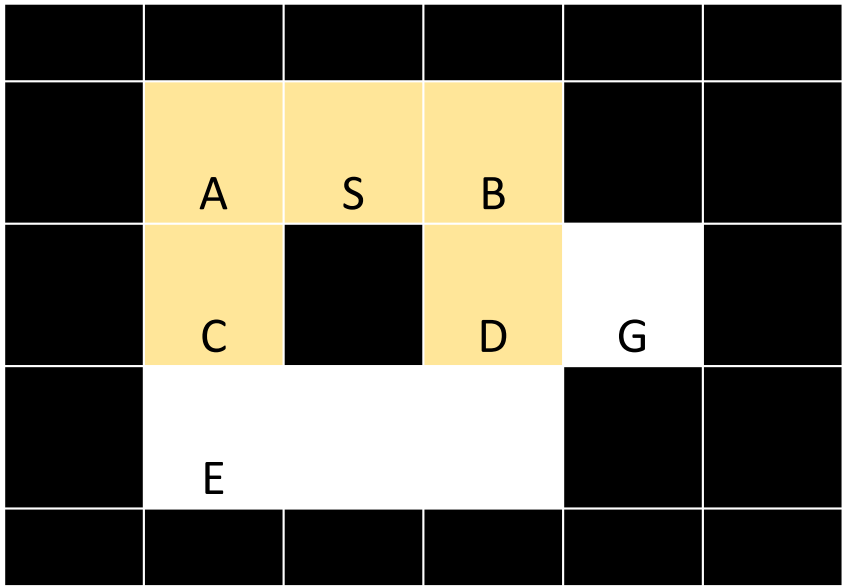


BFS: Frontier = FIFO

Example:

explored = { S, A, B, C, D }

frontier = { E, G }

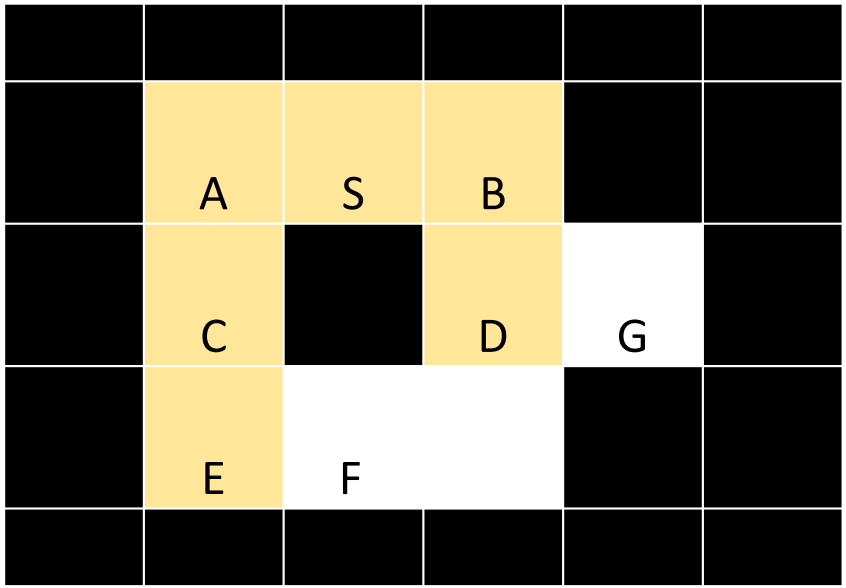


BFS: Frontier = FIFO

Example:

explored = { S, A, B, C, D, E }

frontier = { G, F }



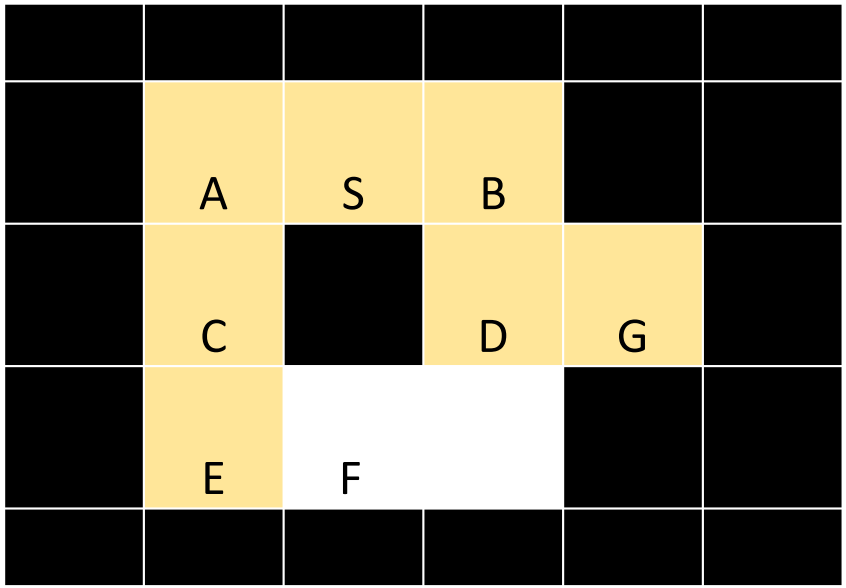
BFS: Frontier = FIFO

Example:

explored = { S, A, B, C, D, E, G }

frontier = { F }

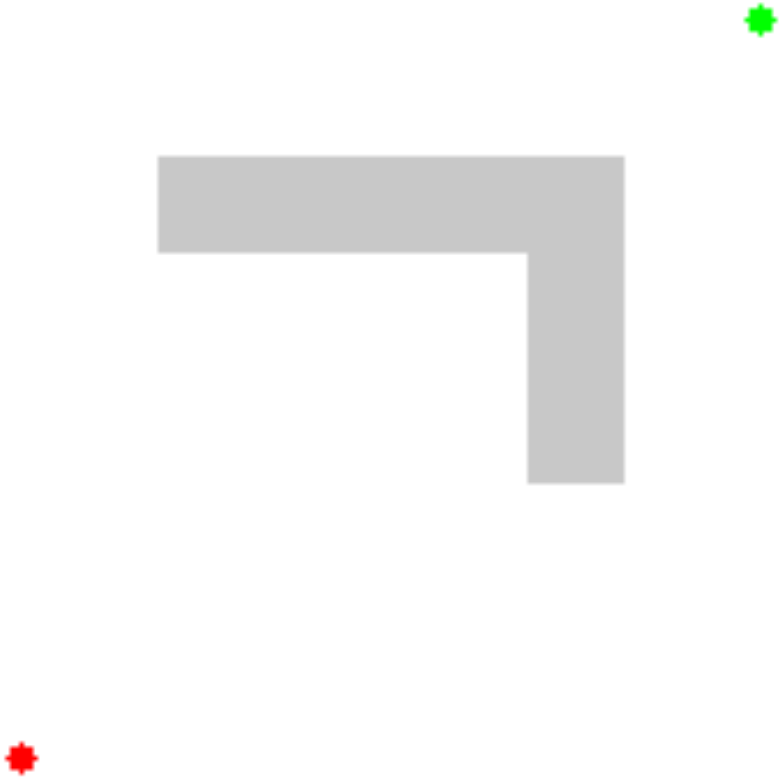
Goal reached!



BFS: How to do it

- Notice that BFS searches in exactly the same order as Dijkstra's algorithm.

- BFS is the normal way you would implement Dijkstra's algorithm for a possibly-infinite search space.



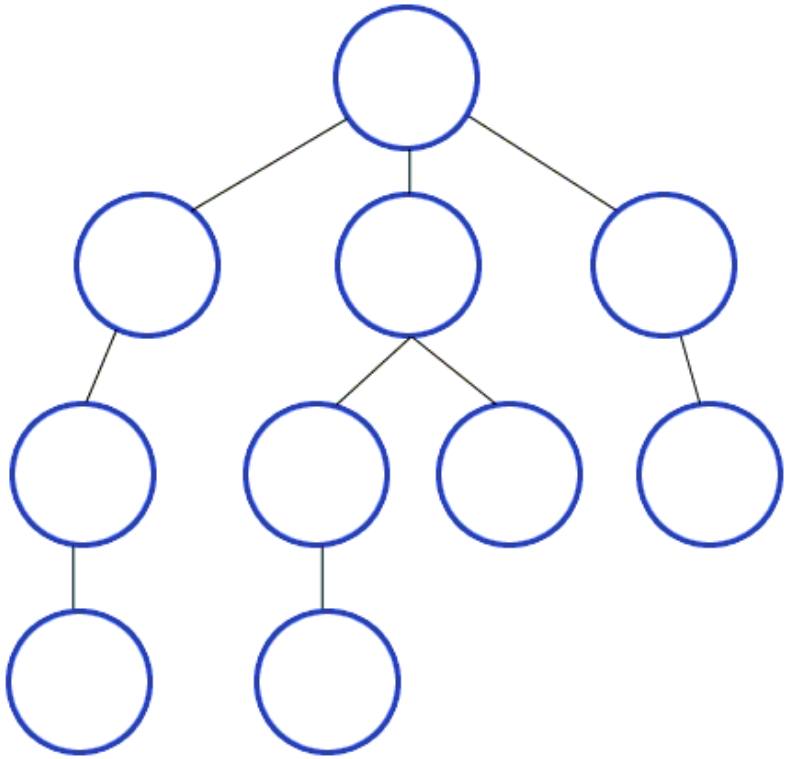
Dijkstra's progress, CC-BY 3.0, Subh83, 2011
https://commons.wikimedia.org/wiki/File:Dijkstras_progress_animation.gif

Depth-first search (DFS)

Exactly like BFS, but instead of expanding the frontier in FIFO order, expand it in LIFO order (last in, first out).

Result: most recently discovered path is pursued, all the way to the end.

Very efficient if there are many solutions, very inefficient if there are few solutions.



Depth-first-search. CC-BY-SA 3.0, Mre, 2009
https://commons.wikimedia.org/wiki/File:Depth-First-Search.gif

Analysis of search strategies

- Strategies are evaluated along the following criteria:
  - **Completeness:** does it always find a solution if one exists?
  - **Optimality:** does it always find a least-cost solution?
  - **Time complexity:** number of nodes generated
  - **Space complexity:** maximum number of nodes in memory
- Time and space complexity are measured in terms of
  - ***b****:* maximum branching factor of the search tree
  - ***d****:* depth of the optimal solution
  - ***m***: maximum length of any path in the state space (may be infinite)

Properties of breadth-first search

- **Complete?**

  Yes (if branching factor b is finite).

  Even w/o explored-set checking, it still works!

- **Optimal?**

  Yes

- **Time?**

  Number of nodes in a b-ary tree of depth d: $O(b^d)$

  (d is the depth of the optimal solution)

- **Space?**

  $O(b^d)$

Properties of depth-first search

- **Complete? (always finds a solution if one exists?)**
  - Fails in infinite-depth spaces
  - Fails if there are loops (unless you keep an "Explored Set")
- **Optimal? (always finds an optimal solution?)**
  - No – returns the first solution it finds
- **Time? (how long does it take, in terms of b, d, m?)**
  - $O(b^m)$ (remember BFS was $O(b^d)$)
  - Terrible if m is much larger than d
- **Space? (how much storage space?)**
  - $O(bm)$, i.e., linear space!
  - The frontier doesn't need to keep track of failed paths, only the currently active path

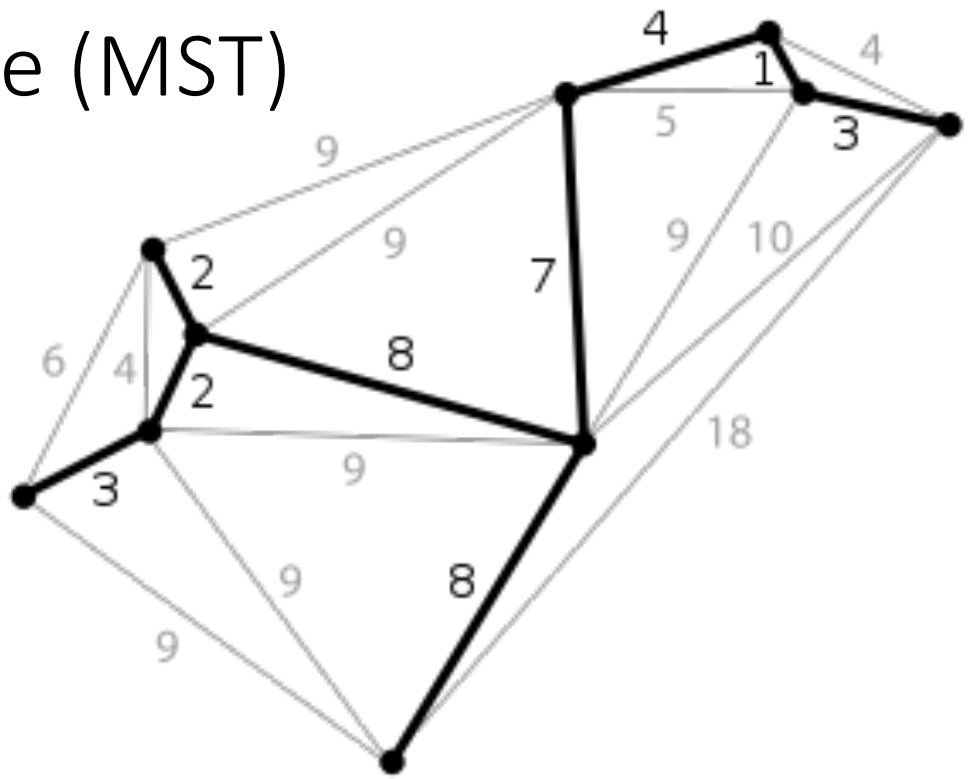
Outline of today's lecture

1. Initial state, goal state, transition model
2. General algorithm for solving search problems
    1. First data structure: a frontier queue
    2. Second data structure: a search tree
    3. Third data structure: a "visited states" set
3. Breadth-first search (BFS) and Depth-first search (DFS)
4. Minimum spanning tree (MST)

Minimum Spanning Tree (MST)

Digression: something totally different…

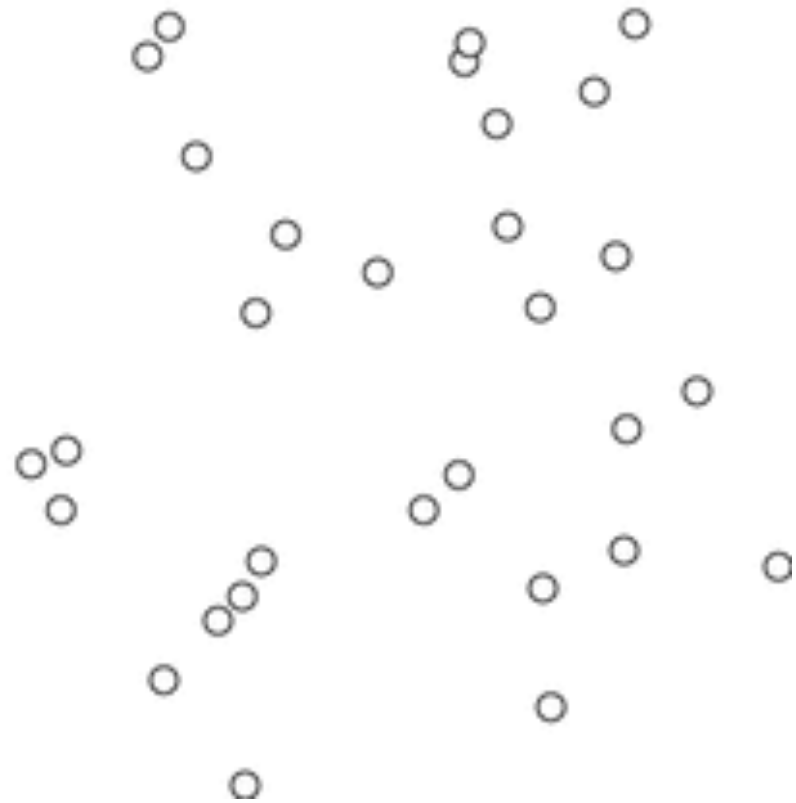
An MST is a **tree** (no cycles) that **spans** a graph (touches every vertex) with the **minimum** possible total distance of the selected edges.



In the example at right, total cost of the MST is the cost of all of its selected eges, i.e., 3+2+2+8+8+7+4+1+3=38

Kruskal's Algorithm

1. Sort the edges in increasing order of cost
2. Start state: every vertex is an independent tree, disconnected from every other
3. While there is more than one tree:
   1. Expand the next-least-cost edge
   2. If it connects two different trees, merge them into the same tree
   3. If not, discard it



Kruskal's algorithm. CC-BY-SA 4.0, Shiyu Ji, 2016
https://commons.wikimedia.org/wiki/File:KruskalDemo.gif

Outline of today's lecture

1. Initial state, goal state, transition model
2. General algorithm for solving search problems
    1. First data structure: a frontier queue
    2. Second data structure: a search tree
    3. Third data structure: a "visited states" dict
3. Breadth-first search (BFS) and Depth-first search (DFS)
4. Minimum spanning tree (MST)