Izmir Institute of Technology

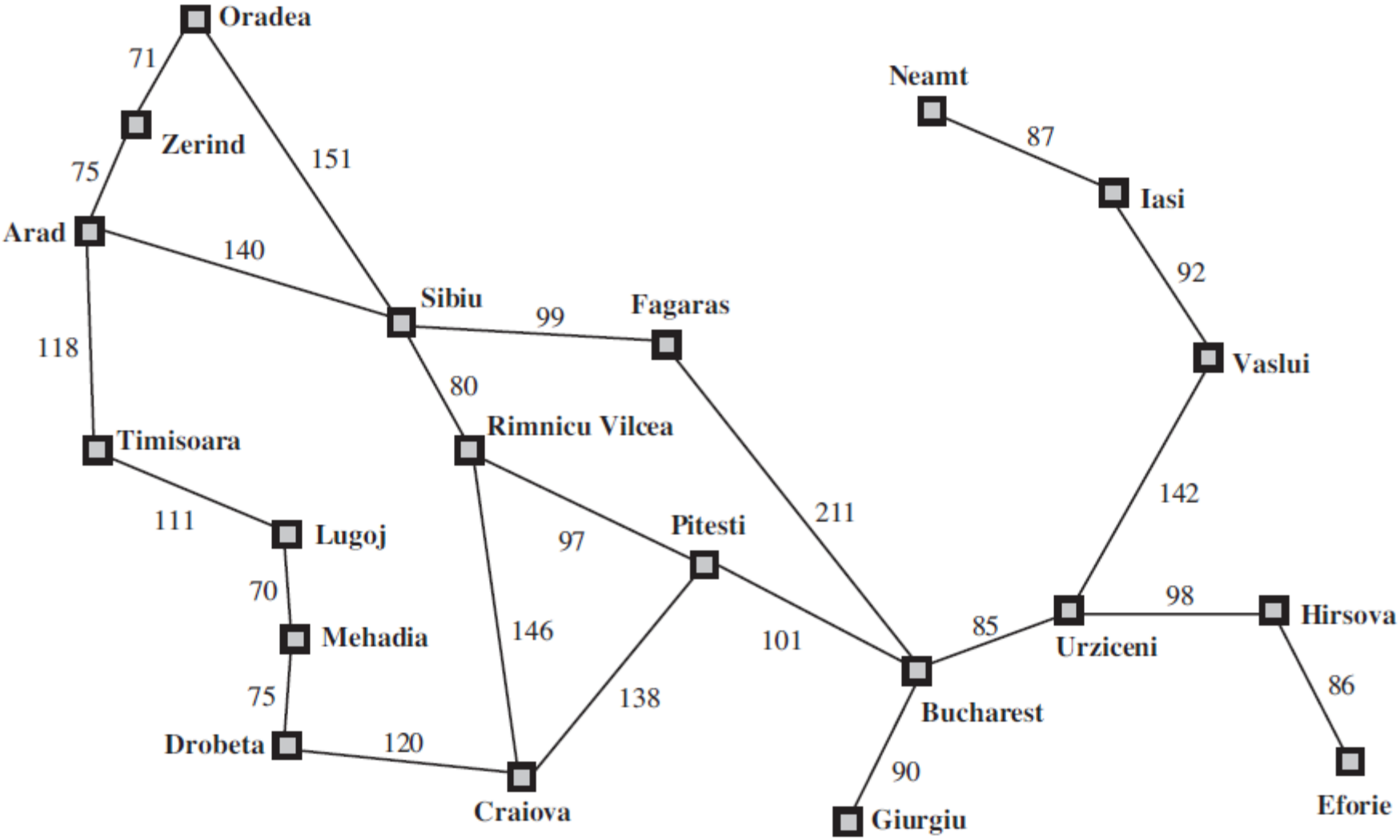
# CENG 461 – Artificial Intelligence

Solving Problems by Searching

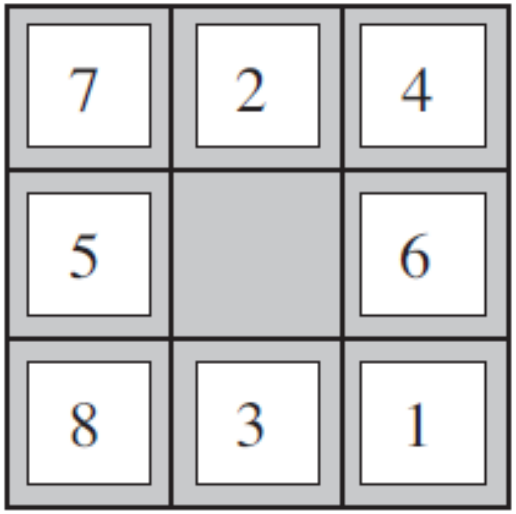
# Introduction

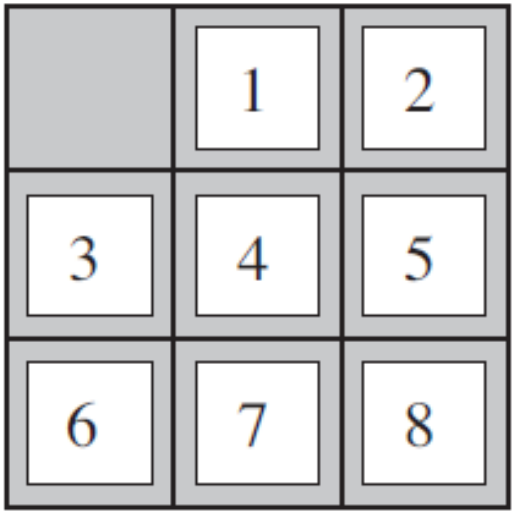
- We will consider problems that can not be solved by a single action.
- The solution will require a sequence of actions.
- At each step we will have multiple possible actions with possibly different but known costs.
- The difficulty lies in the large set of possible states that we need to consider to find the optimal sequence of actions to reach our goal.

# Route Finding



# 8 Puzzle

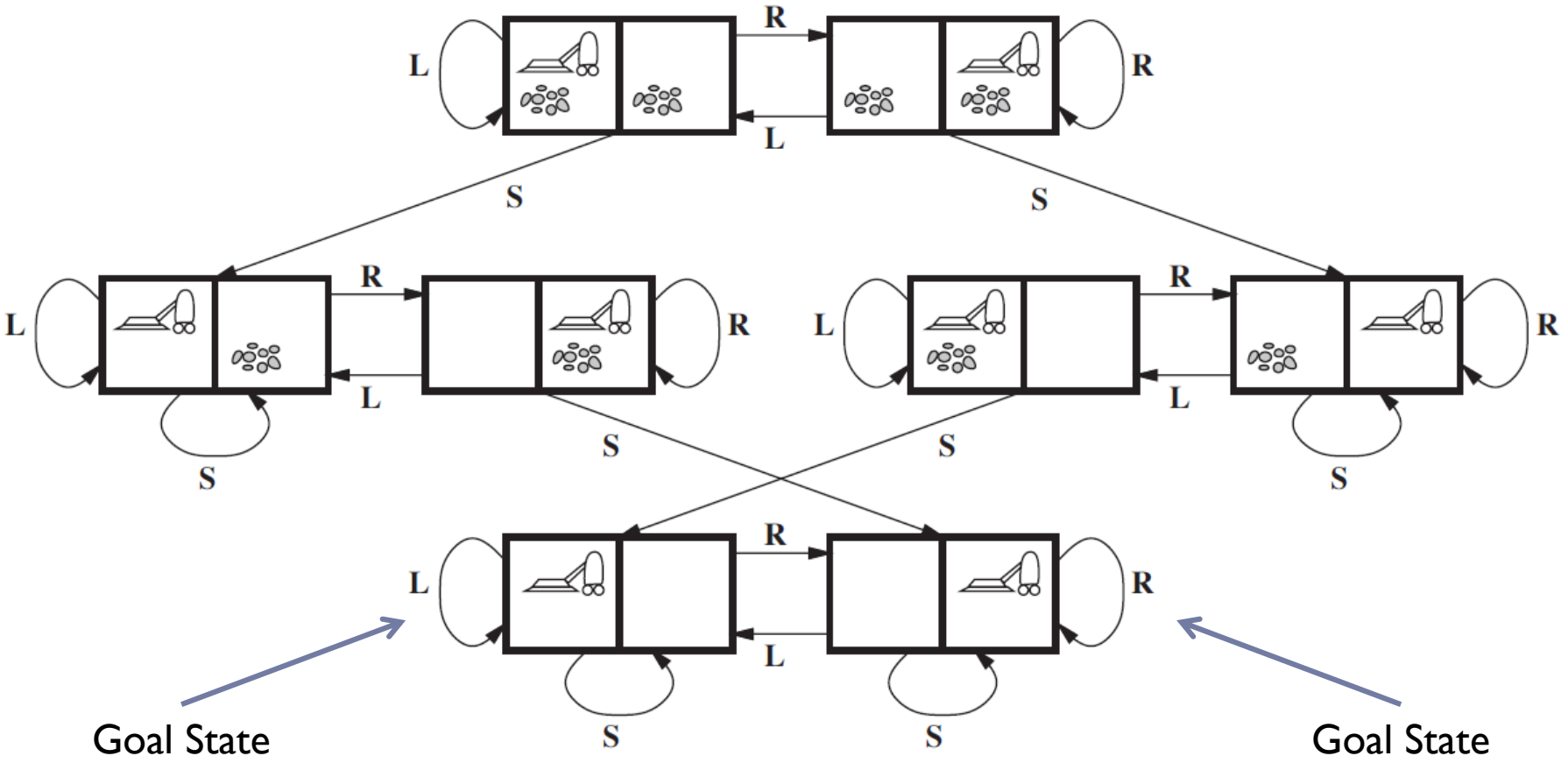


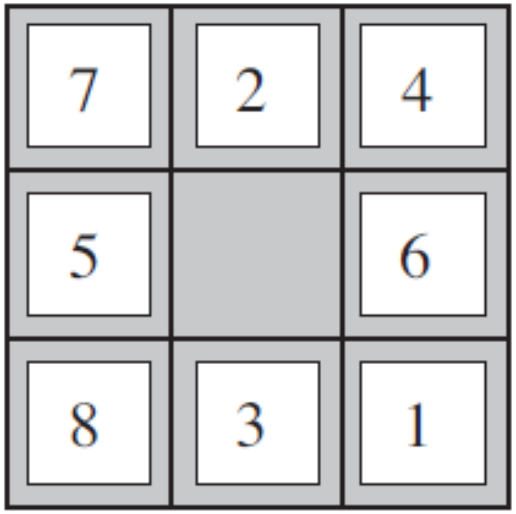

# Problem Definition

- The initial state
  - $s_0$
- Actions for each state
  - Actions(s) → { $a_1, a_2, a_3, \ldots, a_N$ }
- Result state for each (state, action) pair
  - Result(s, a) → s'
- Goal predicate
  - Checks whether we have reached a state or not, GoalTest(s) →T|F
- Path cost for a sequence of actions from a starting state to an end state
  - If the path cost is the sum of step costs along the path, then we need to know the step cost for each (state, action) pair: StepCost(s,a,s')
- Solution/Optimal Solution

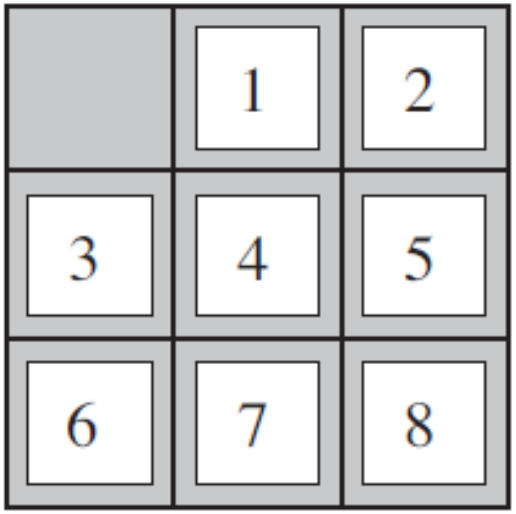
# Vacuum-world



# 8-Puzzle



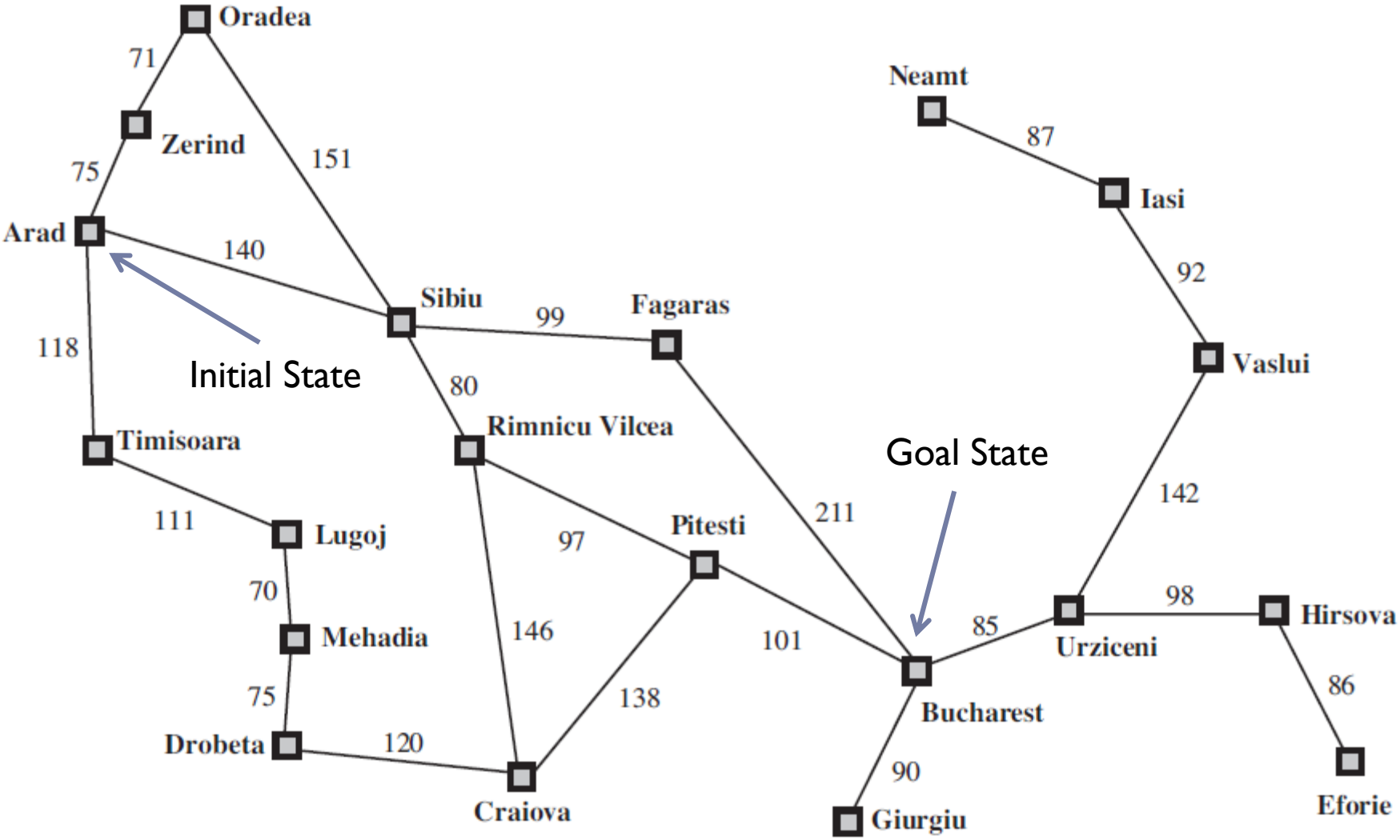

Initial State




Goal State

States, Actions, Path cost, Optimal solution ?

# Route Finding




- States, Actions, Path cost, Optimal solution ?

# Searching

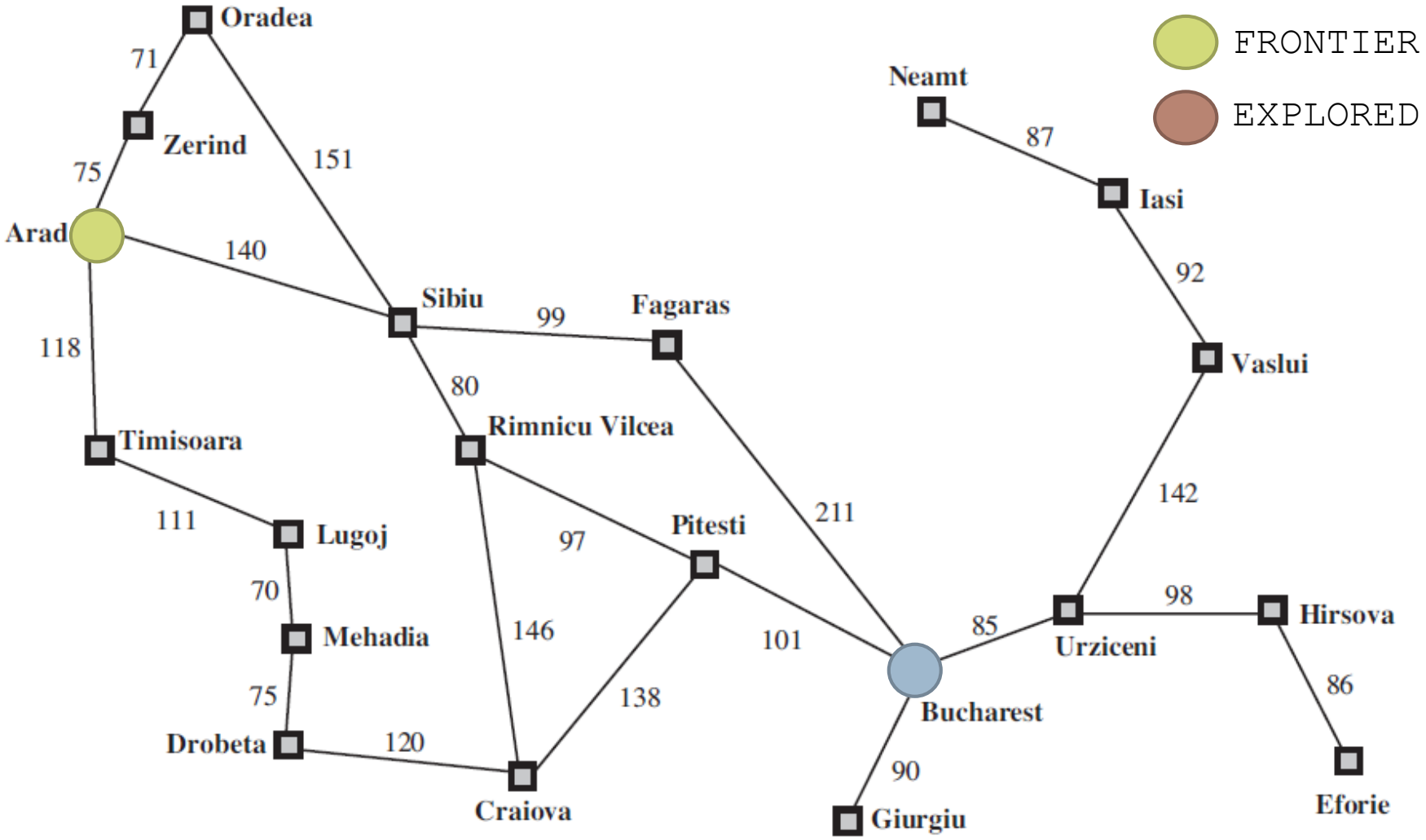
- A search tree is used for the search.
  - The root node is the initial state.
  - Each branch corresponds to an action taken at the parent node.
  - For each node, we need to check if we are in a goal state, and if not expand the tree by applying all possible actions for that node leading to child nodes.
  - A leaf node has no children. The set of all leaf nodes available for expansion at any given point is called the **frontier.**
- For a search problem we have
  - Frontier States
  - Explored States
  - Unexplored States

# Tree Search

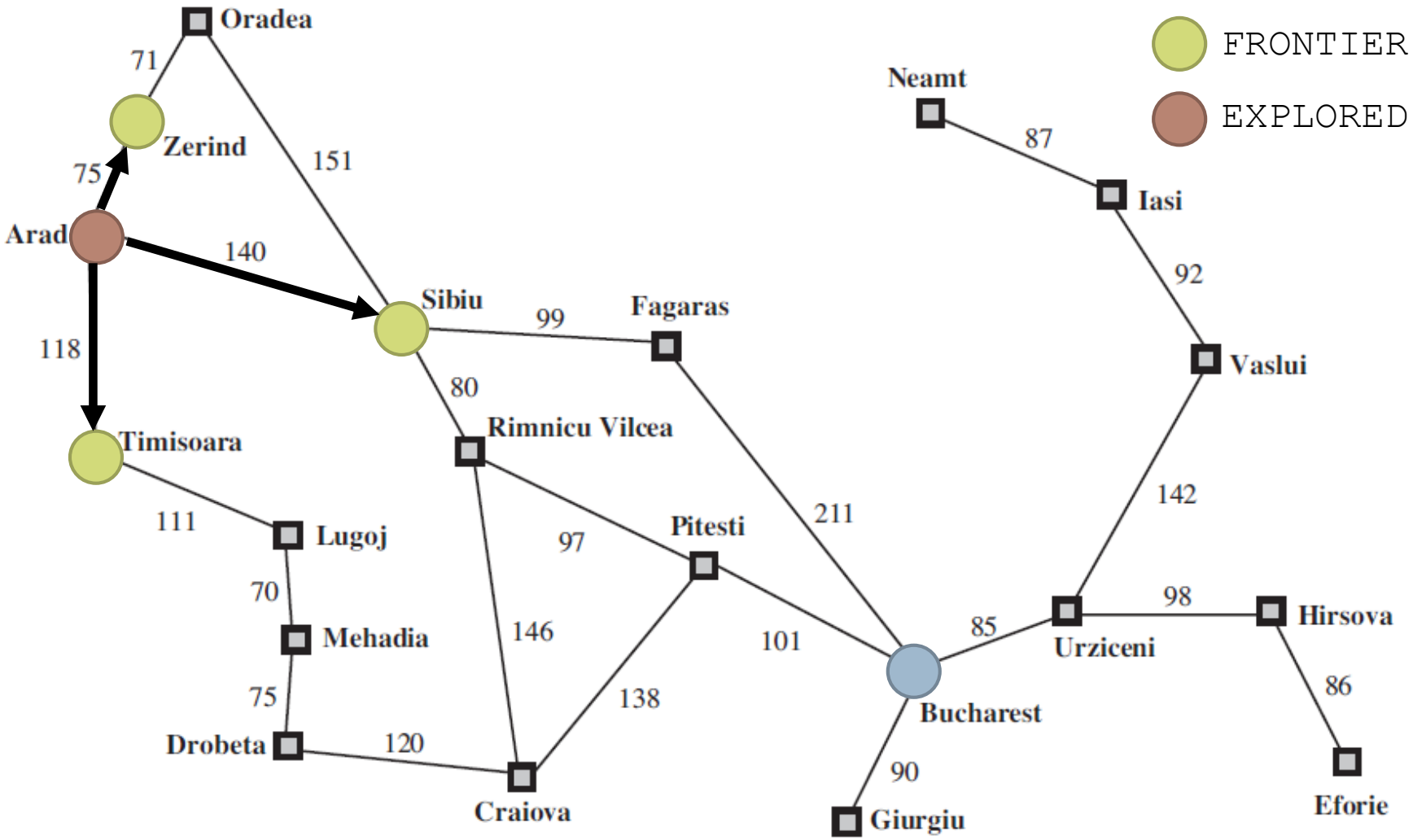
- At each step we need to choose which node to expand first (indicated by the blue line in the pseudo-code below).
- There are multiple ways to decide and this choice leads to different strategies.

```
function TREE-SEARCH(problem) %returns a solution, or failure
  frontier = initial state  %initialize the frontier
  loop do
    if frontier is empty then
       return failure
    choose a leaf node (s) and remove it from frontier
    if node s contains a goal state then
       return the corresponding solution
    expand node s by applying possible actions (result(s,a))
    add the resulting nodes to the frontier
```

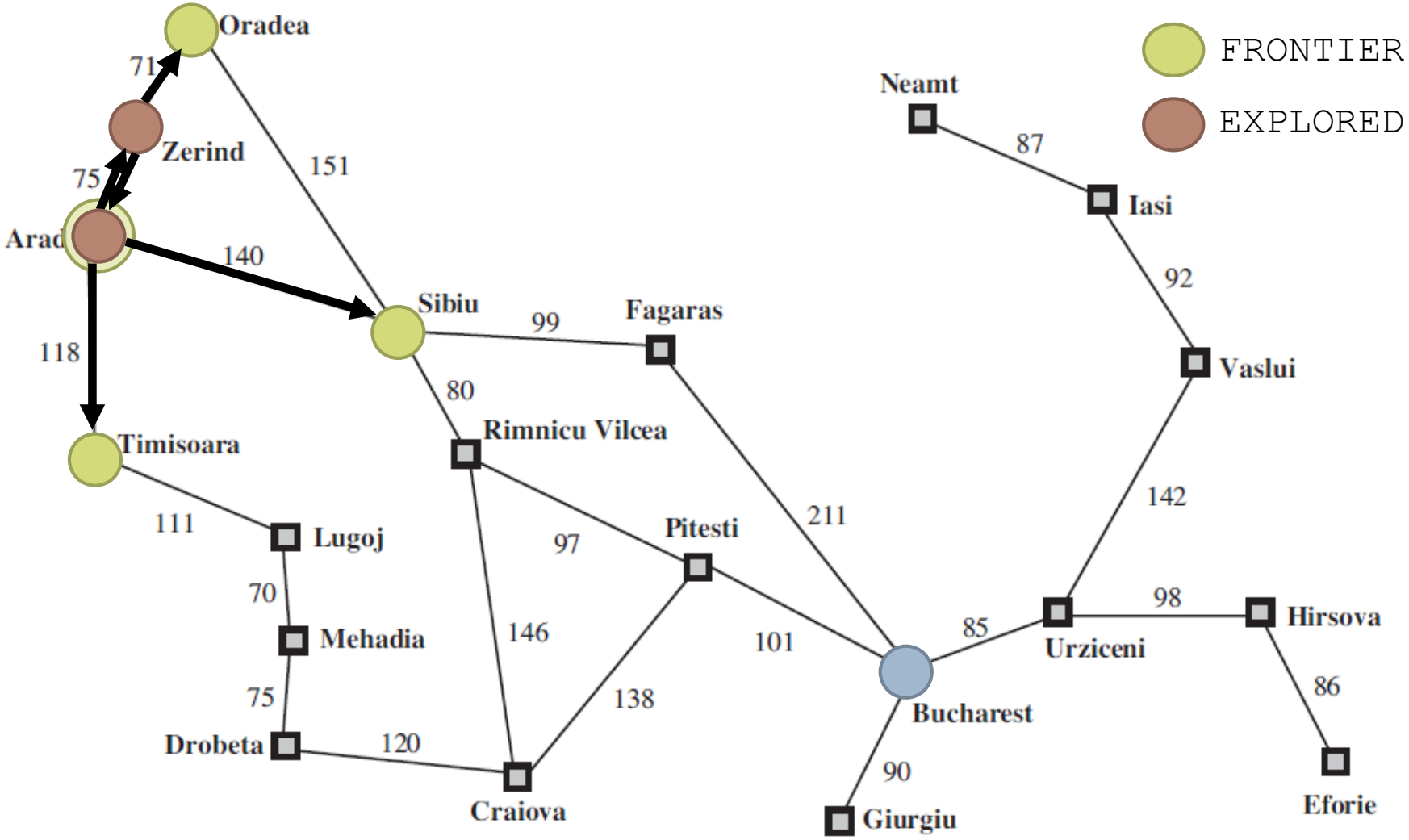
# Tree Search for Route Finding



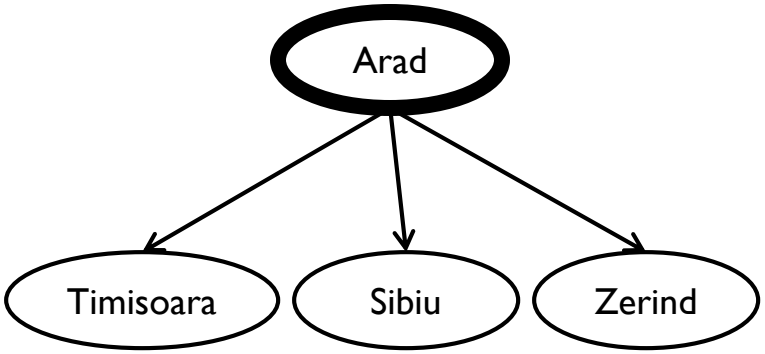
# Tree Search for Route Finding



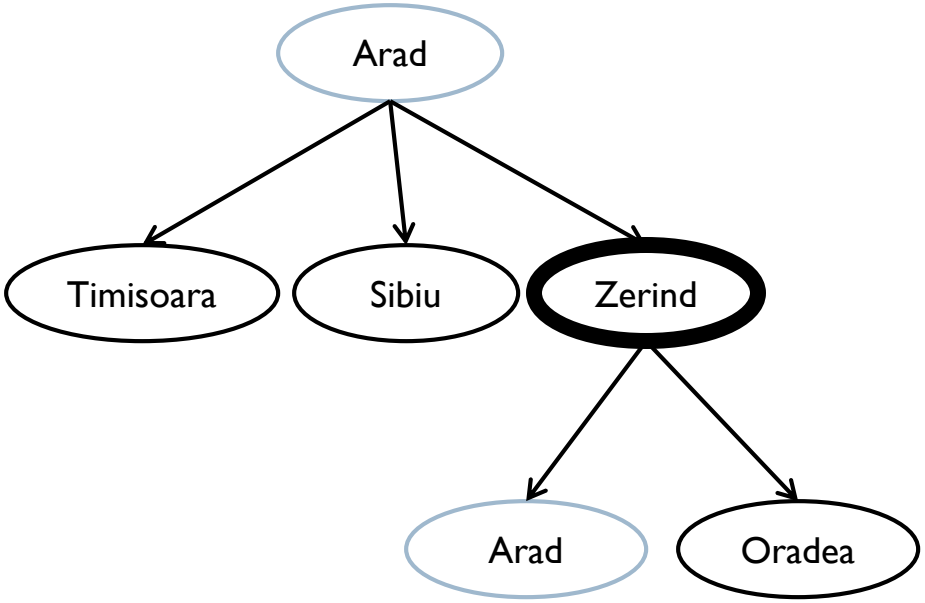
# Tree Search for Route Finding



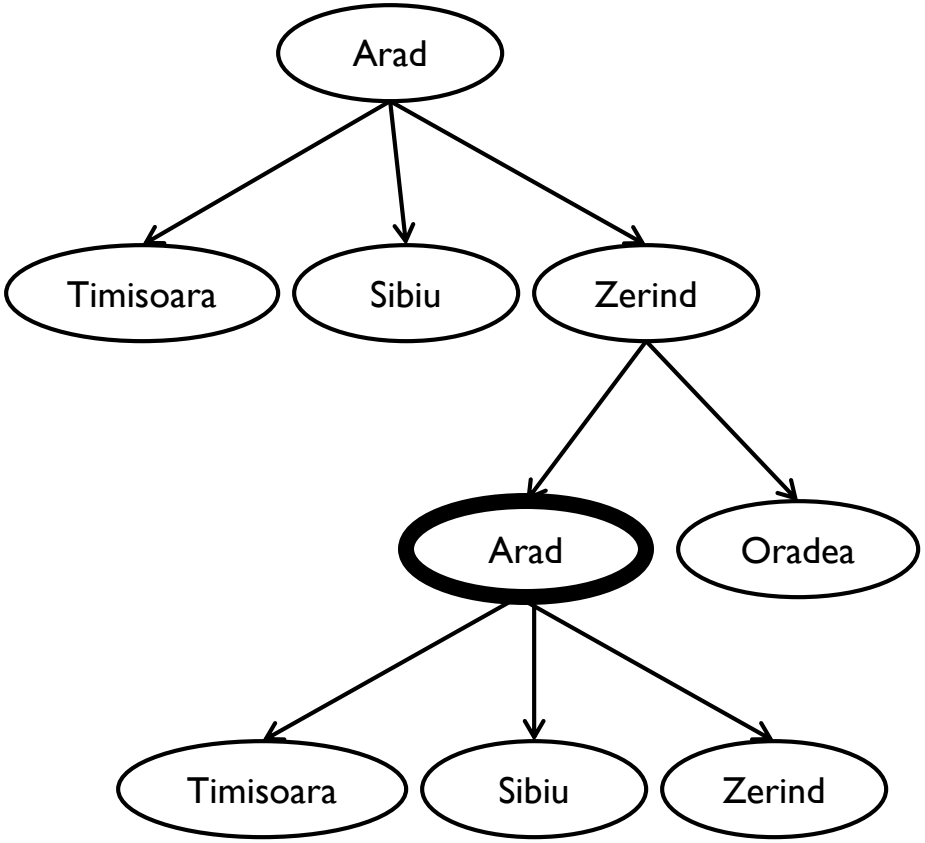
# Tree Search for Route Finding




# Tree Search for Route Finding




# Tree Search for Route Finding



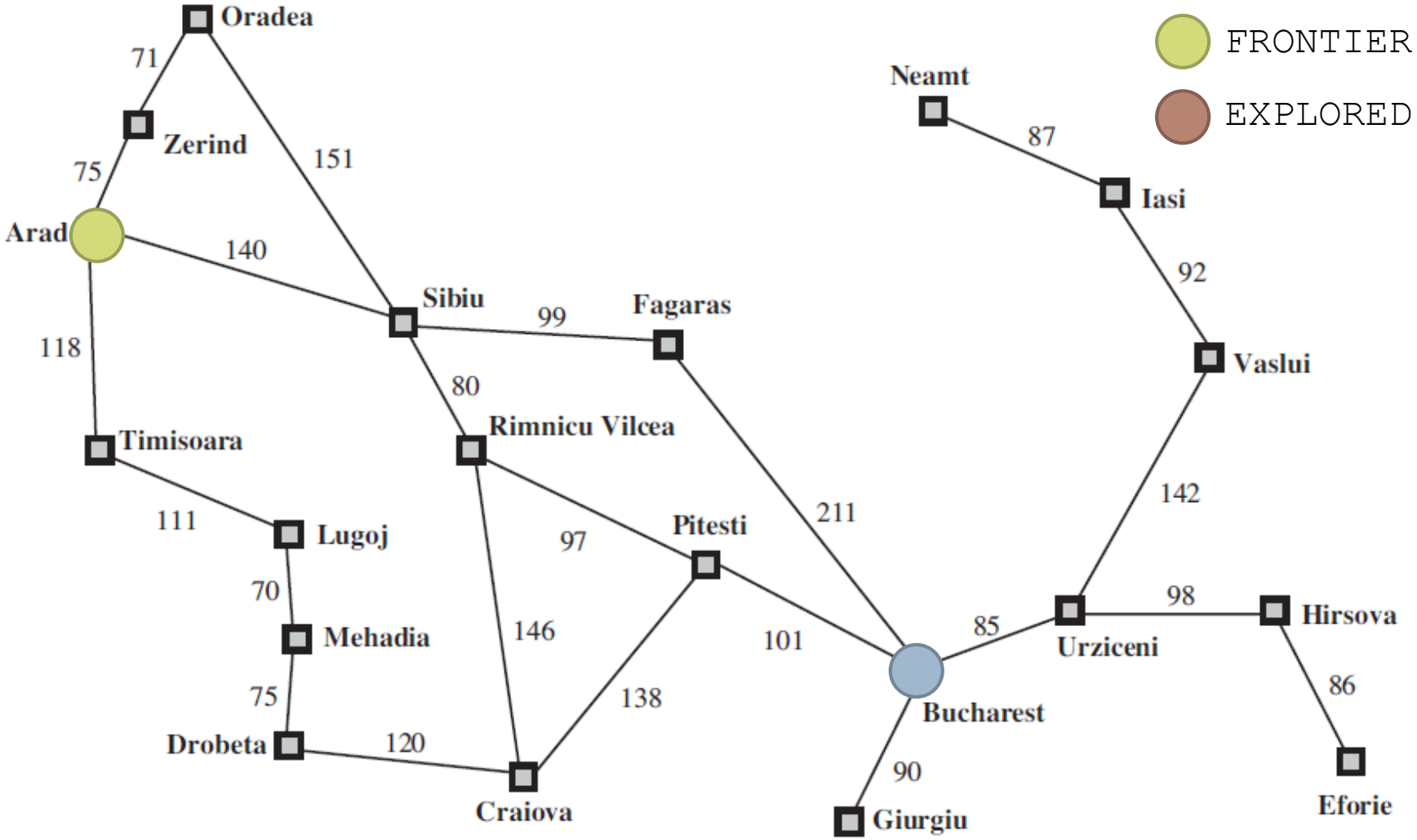

# Avoiding Loops with Graph Search

- When it is possible to get back to a previously visited states, the tree-search algorithm explores it again.
- Such loops can be avoided by keeping a history of explored states.

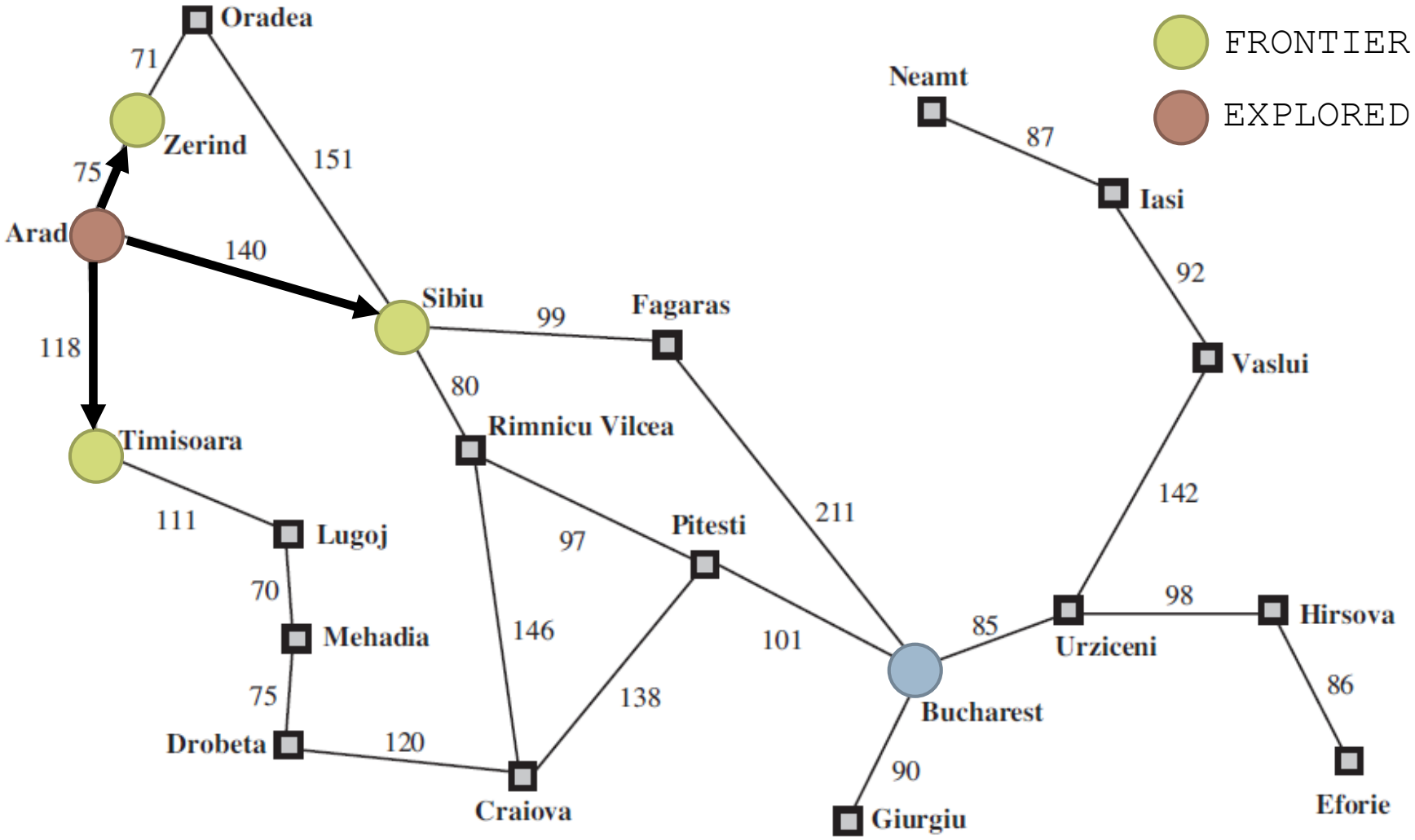
```
function GRAPH-SEARCH(problem) %returns a solution, or failure
    frontier = initial state  %initialize the frontier
    explored = {} %initialize the explored set to be empty
    loop do
        if the frontier is empty then
            return failure
        choose a leaf node (s) and remove it from frontier
        add node s to the explored
        if node s contains a goal state then
            return the corresponding solution
        expand node s by applying all possible actions (result(s,a))
        if the resulting node is not in frontier or explored then
            add the resulting nodes to frontier
```

* Blue lines indiate the difference between graph-search and tree-search

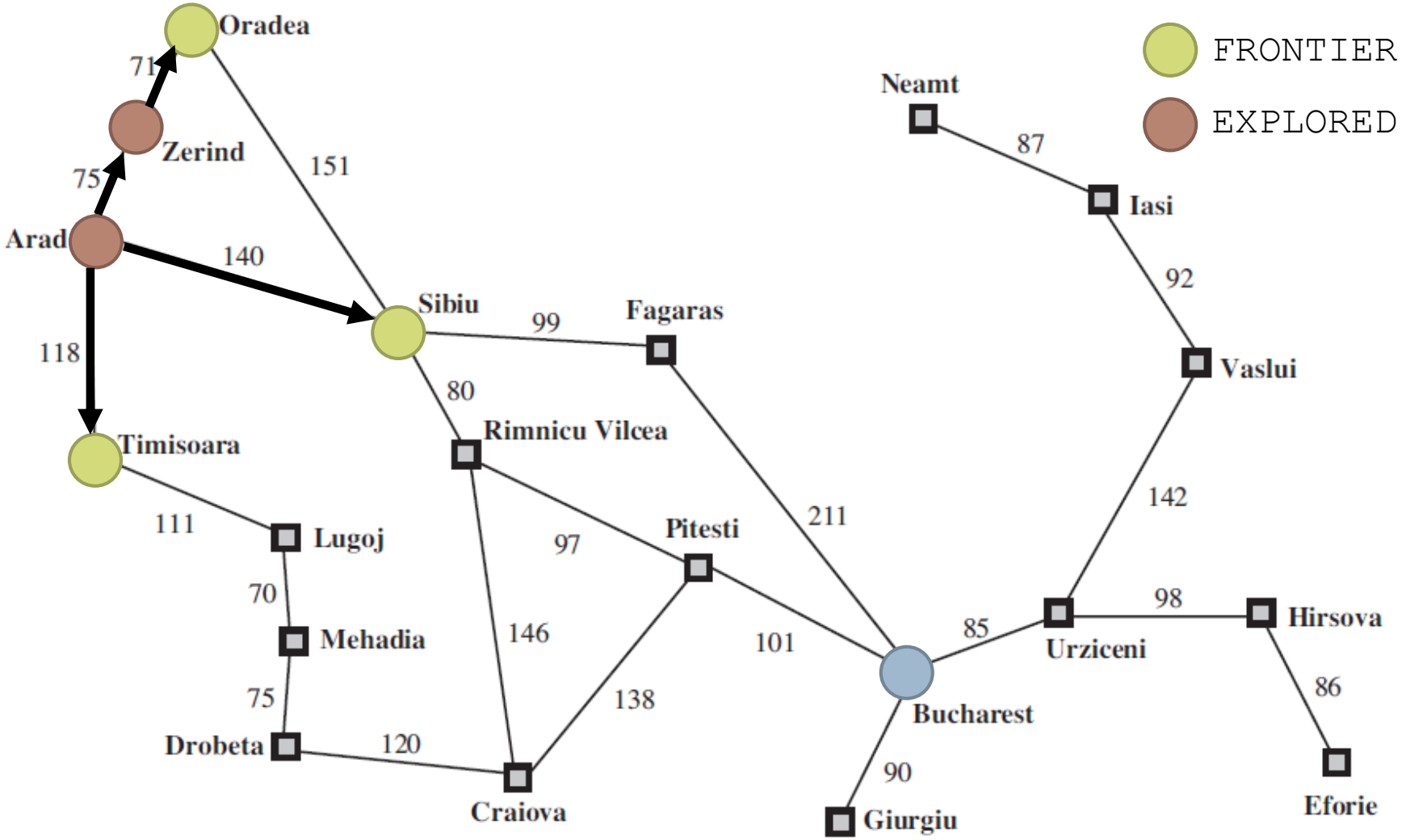
# Graph Search for Route Finding



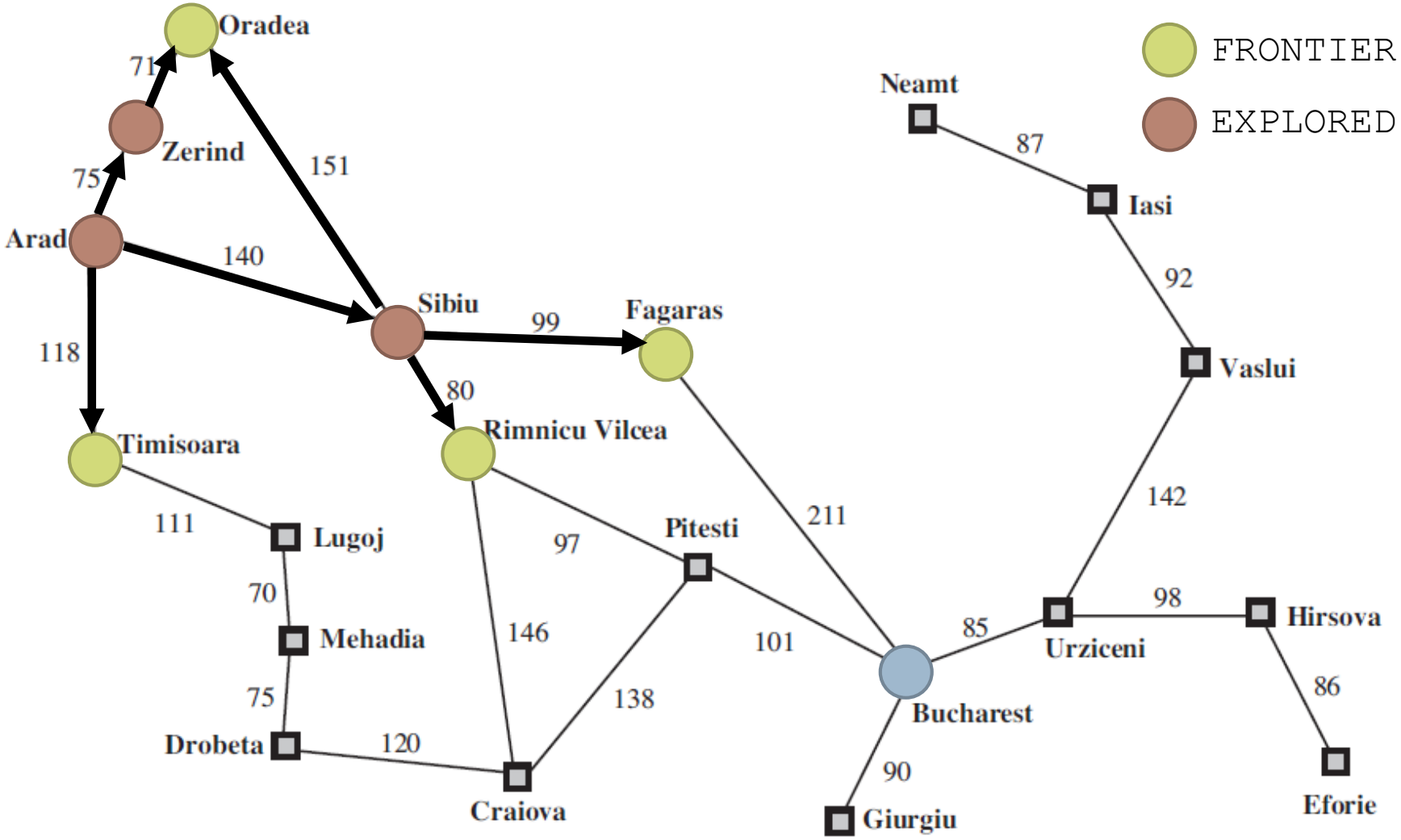
# Graph Search for Route Finding



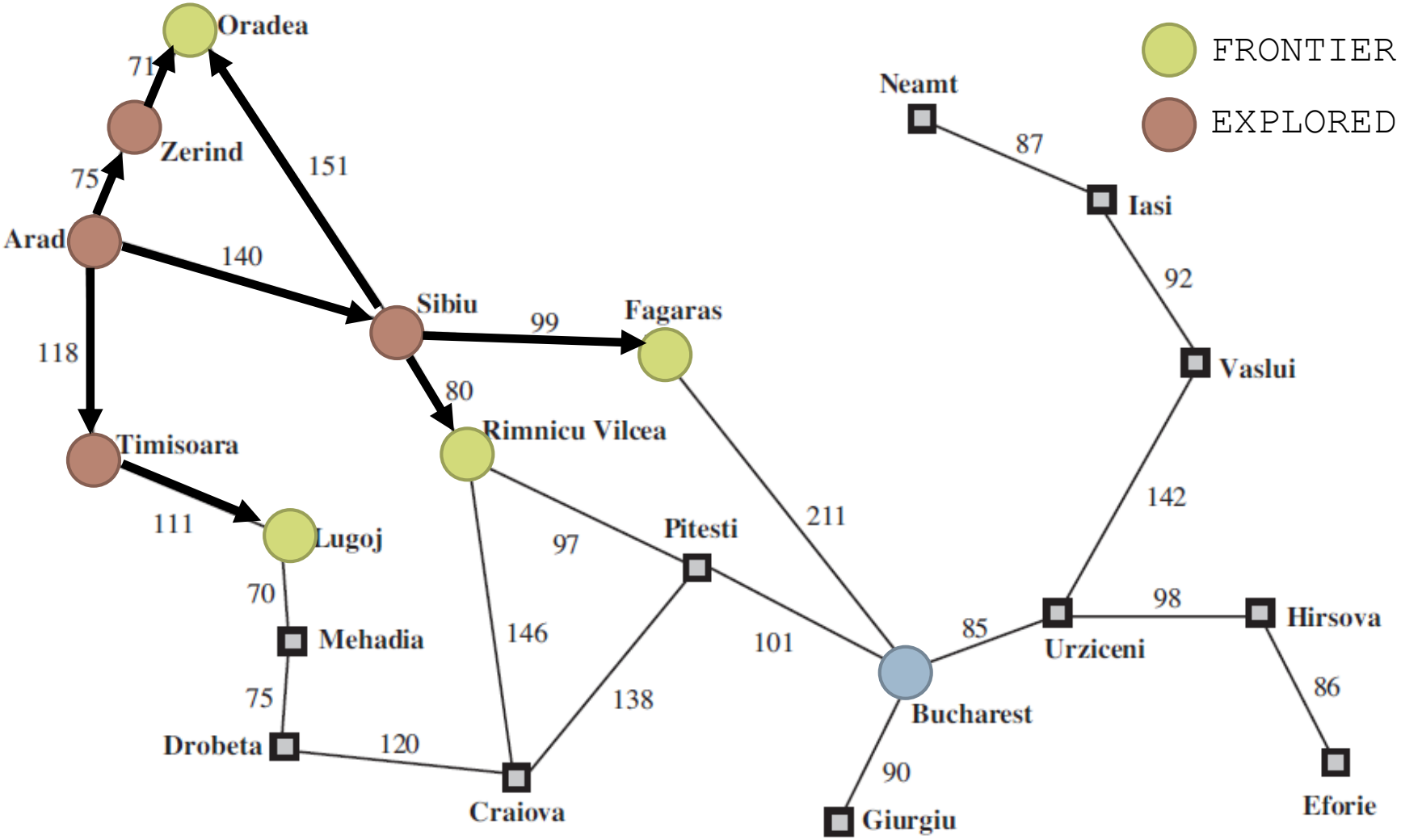
# Graph Search for Route Finding



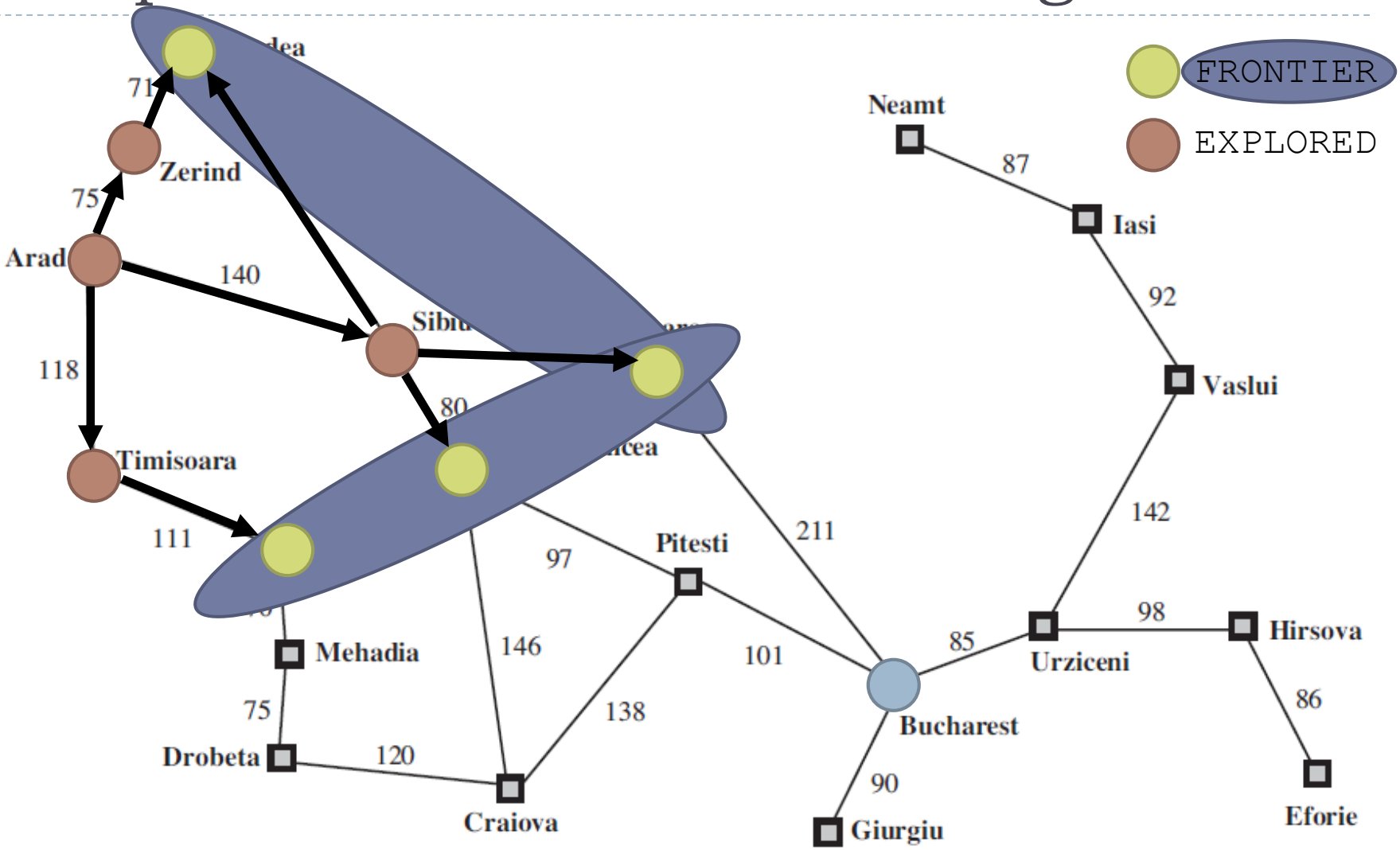
# Graph Search for Route Finding



# Graph Search for Route Finding

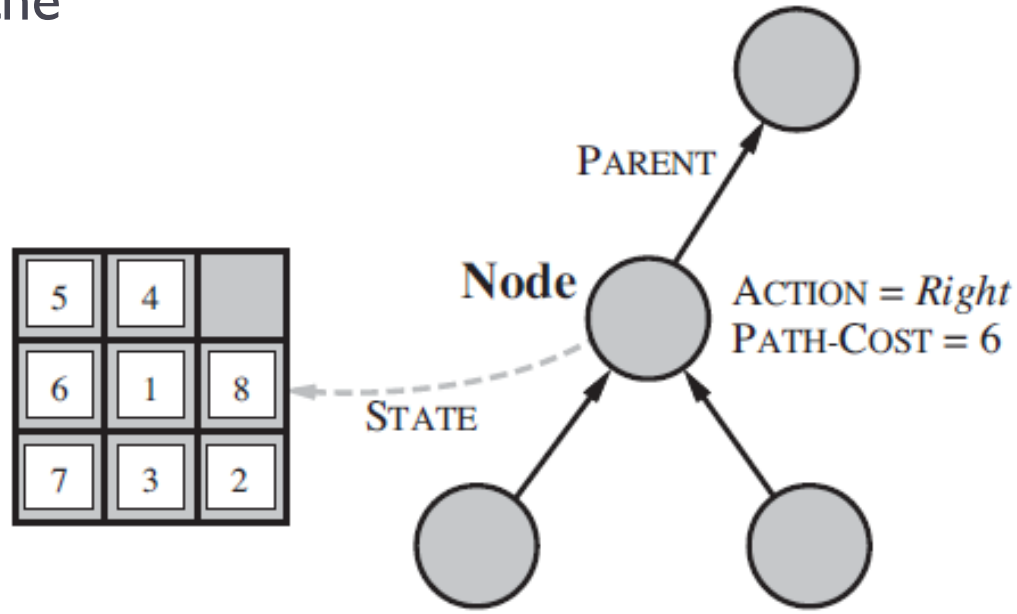


# Graph Search for Route Finding



# Implementing Search Algorithms

- For each node, we need
  - **state**: the state that the node corresponds to
  - **parent**: the node that generated this one, we need this to backtrack from a goal state.
  - **action**: the action at the parent node.
  - **path_cost**: the cost from initial state to this node.




# Implementing Search Algorithms

- Make sure to distinguish between states and nodes

  Nodes lie on a path whereas a state corresponds to a configuration of the world. Same state can occur at different nodes. E.g. You can reach a city using different paths.

- The frontier is usually stored in a queue (the algorithm can easily choose the next node to expand according to its preferred strategy):
  - It could be a regular FIFO queue.
  - It could be a stack.
  - It could be a priority queue.
- We also need to store the explored set

# Measuring Performance

- Completeness
  - Is it guaranteed to find a solution?
- Optimality
  - Is it guaranteed to find the optimal solution?
- Time Complexity
  - How does the running time and problem size related?
- Space Complexity
  - How does the required memory and problem size related?

# Uninformed Search Strategies

- When we do not know anything about relative suitability of expanding non-goal states, we perform an uninformed (blind) search.
- The strategies below are distinguished by the order in which nodes are expanded.
  - Breadth-First Search
  - Uniform-Cost Search
  - Depth-First Search
  - Depth-Limited Search
  - Iterative Deepening Depth-First Search
  - Bidirectional Search

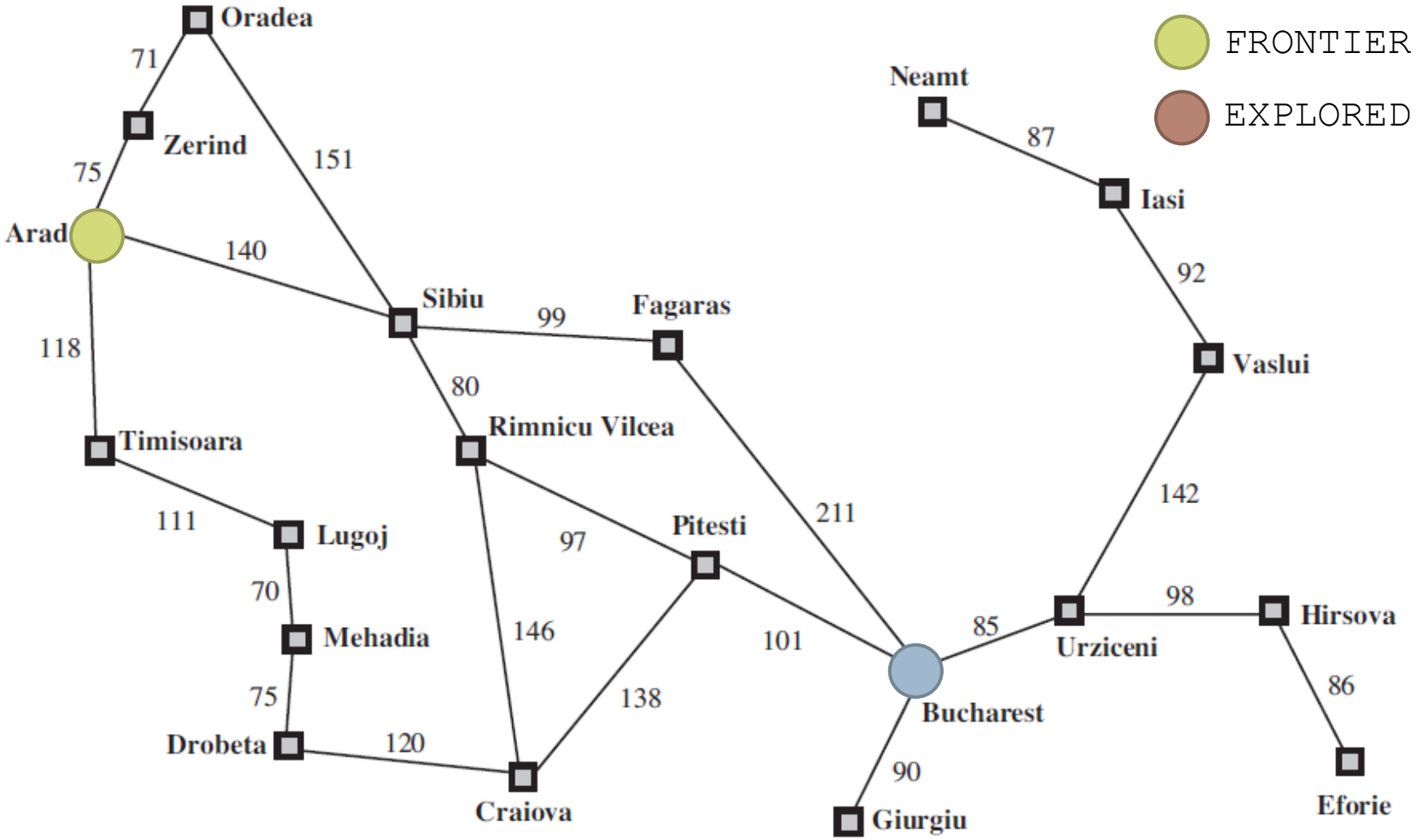
# Breadth-First Search

- Expand the root node first, then its successors, then all the successors of the root node are expanded next, then their successors, and so on.
- All the nodes are expanded at a given depth (n) in the search tree before any nodes at the next level (n+1) are expanded.
- The frontier can be implemented as a FIFO queue.

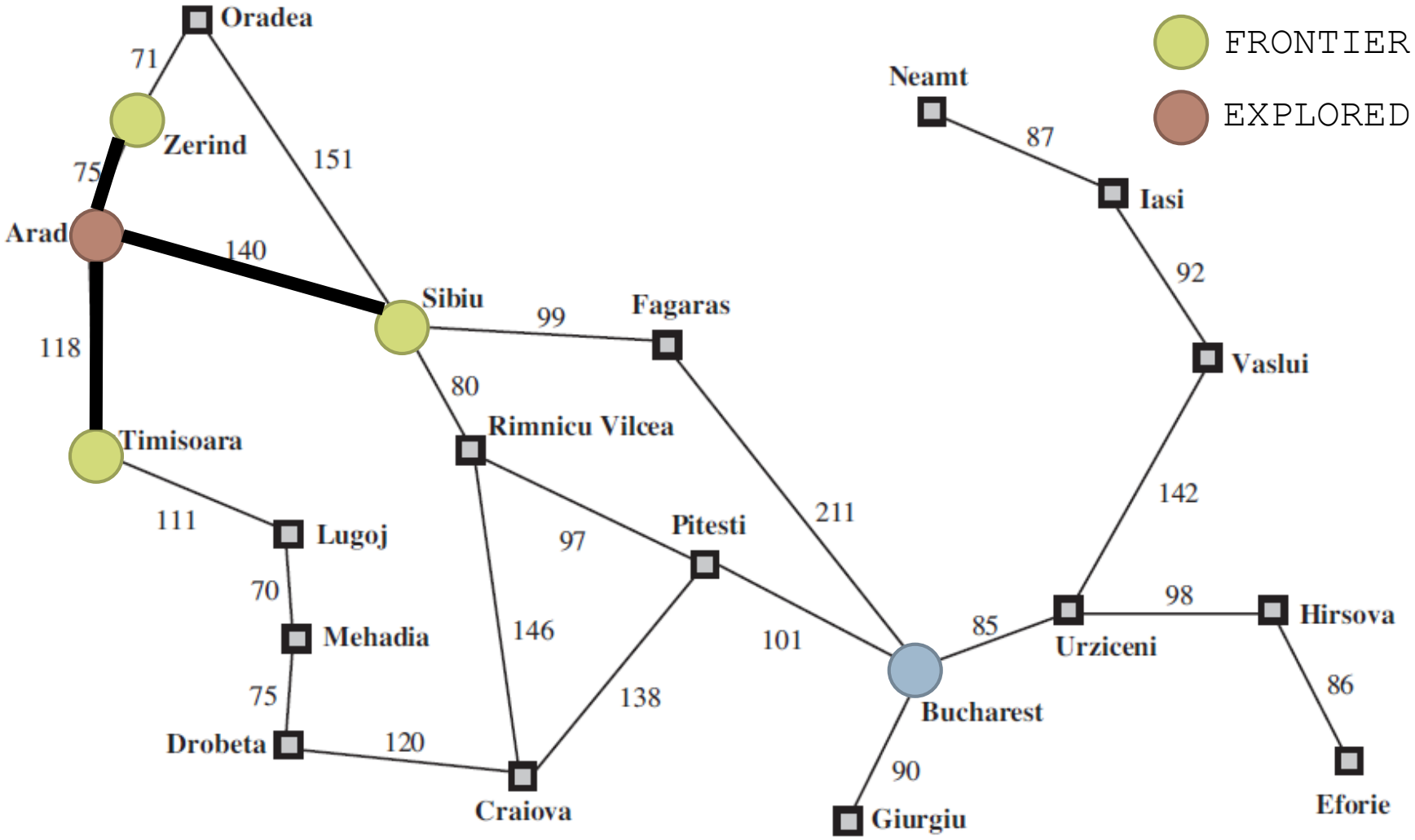
# Breadth-First Search (for reference)

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

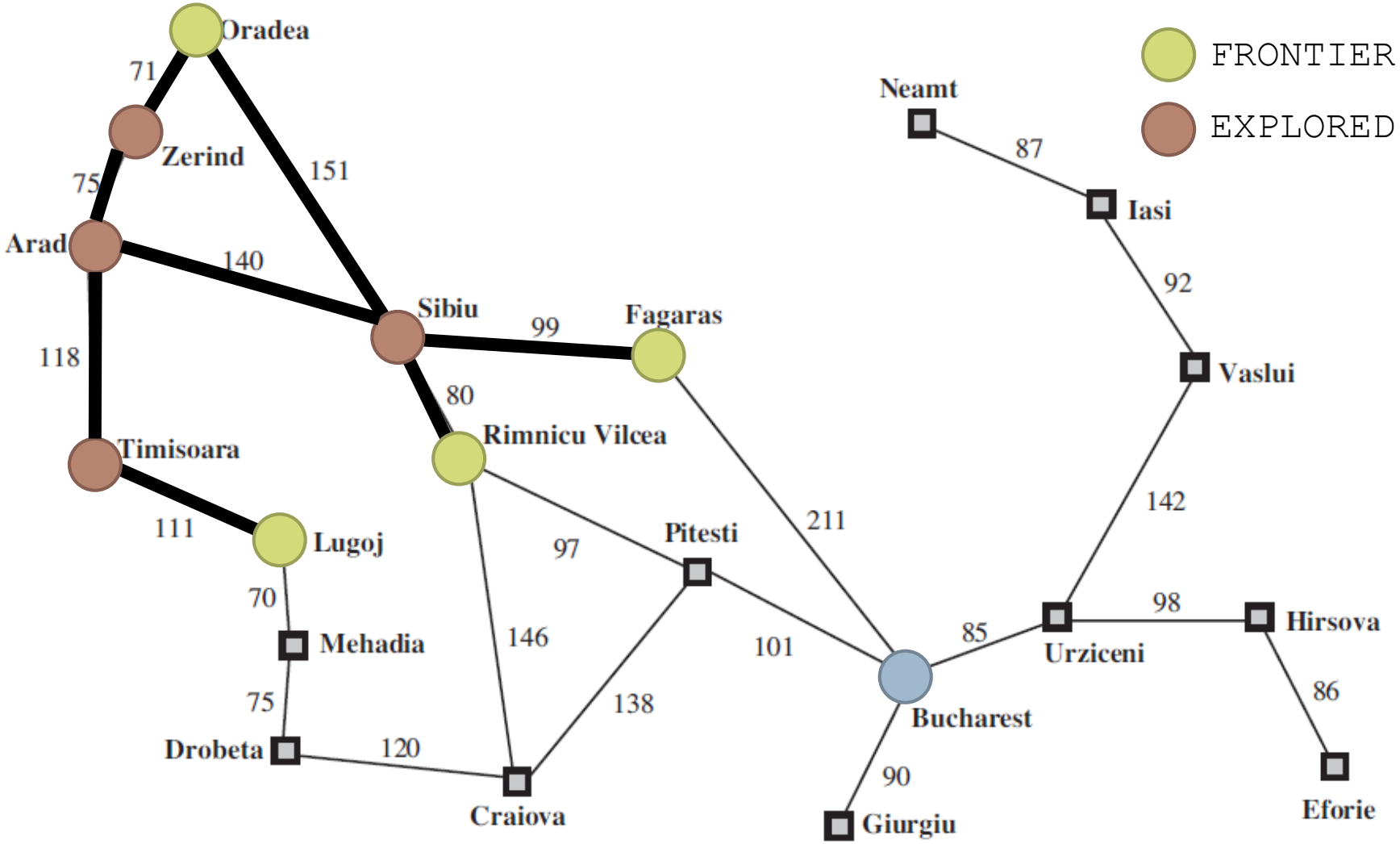
# Breadth-First Search for Route Finding



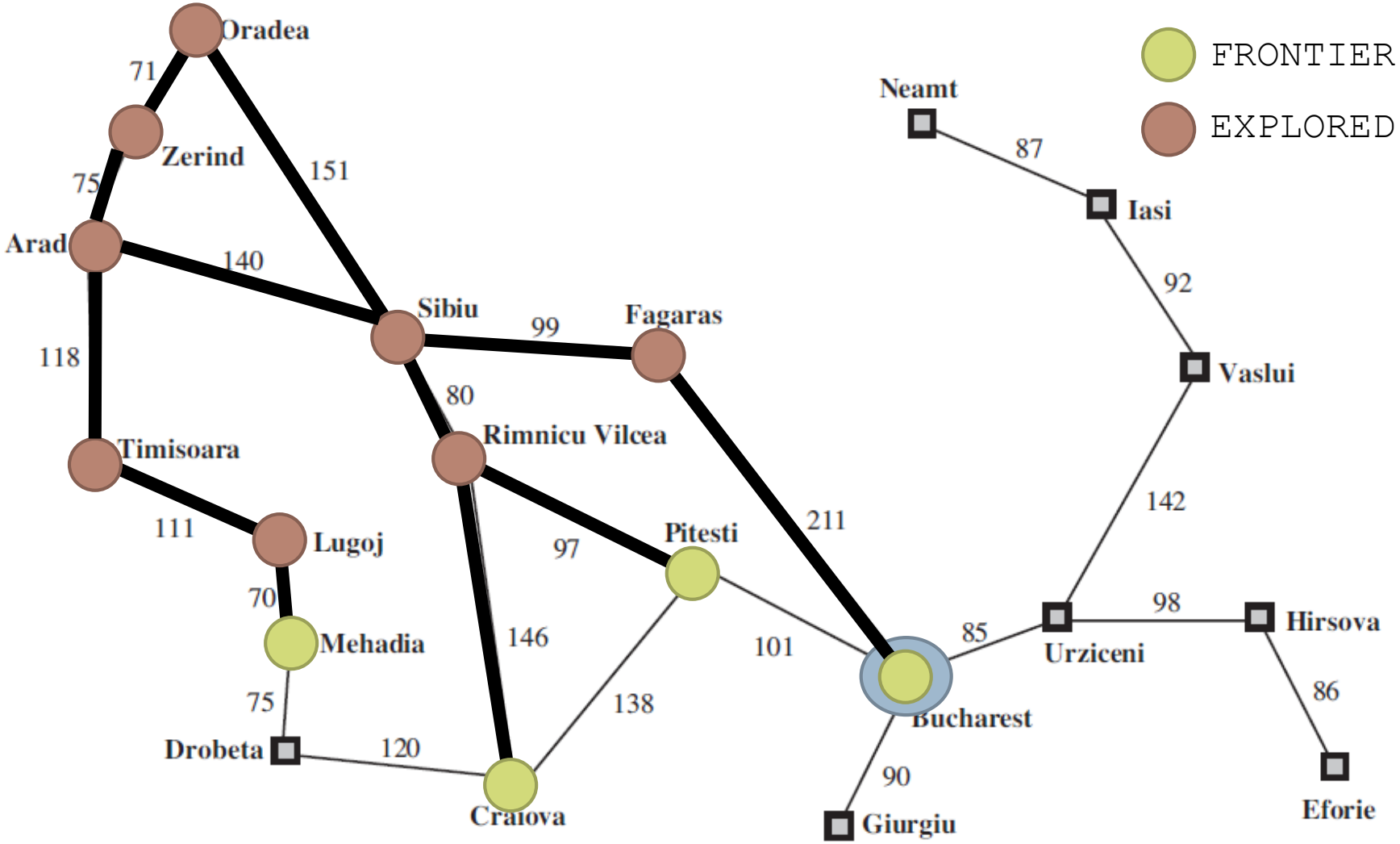
# Breadth-First Search for Route Finding



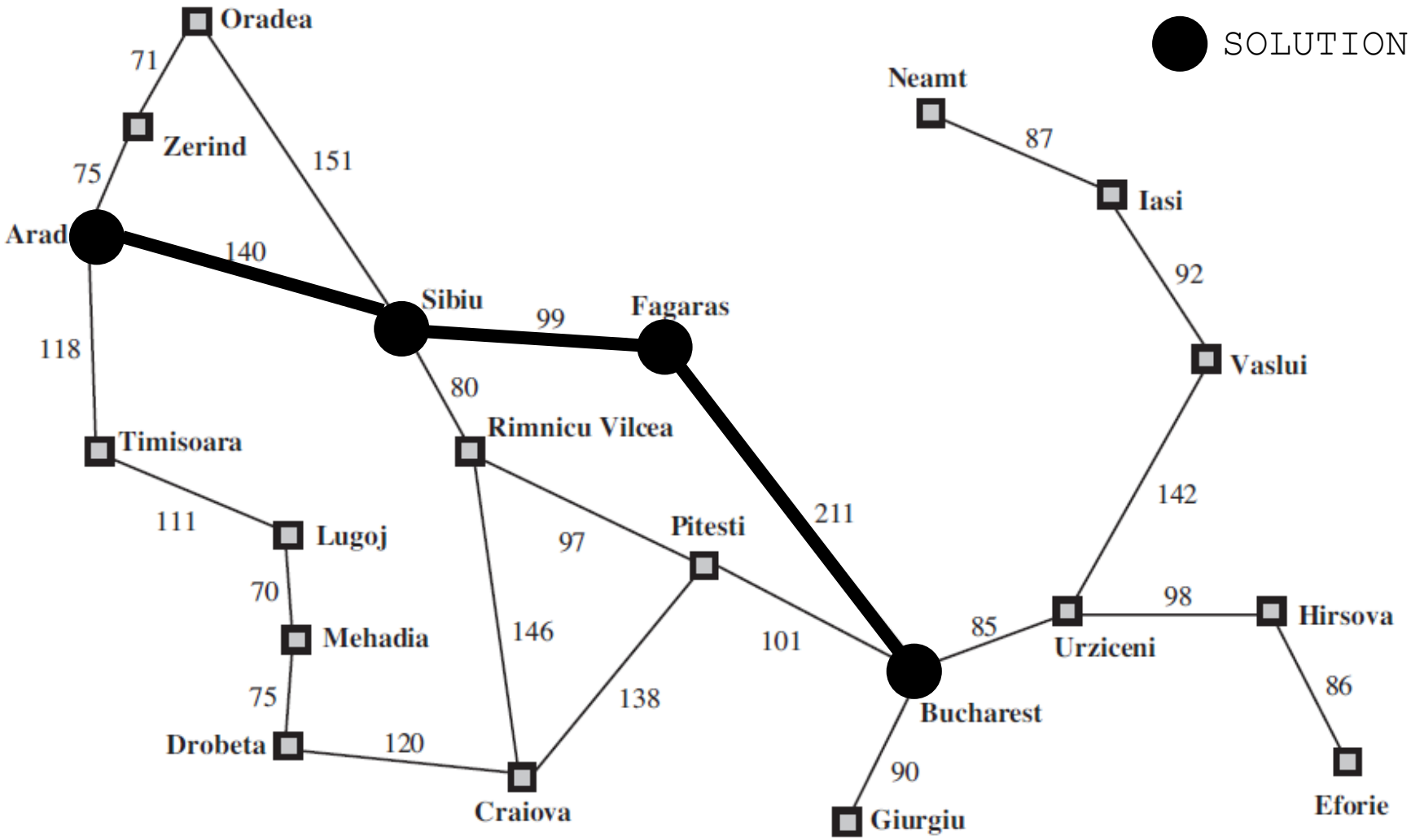
# Breadth-First Search for Route Finding



# Breadth-First Search for Route Finding



# Breadth-First Search for Route Finding

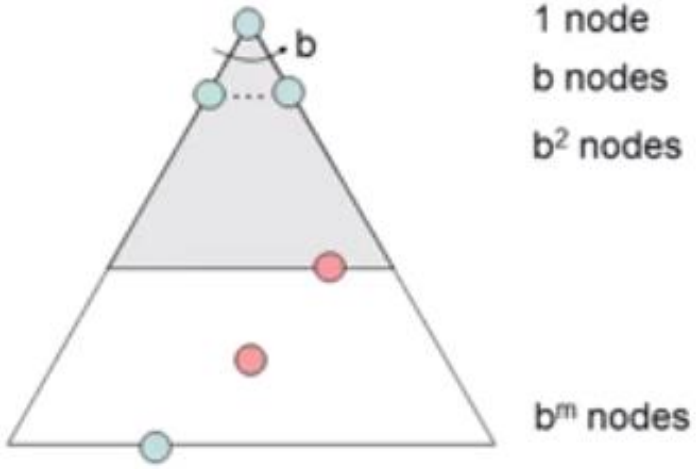



- Is it the minimum cost path?

# Breadth-First Search

- It is complete since if the goal state is at depth n, it is guaranteed to be explored after n-1.
- It is optimal only if the path cost is a non-decreasing function of the depth of the node, i.e. if a better solution can not exist deeper (e.g. all step costs are equal).
- Its time complexity is $O(b^d)$, where b is the branching factor and d is the solution depth.

  If goal-state is checked after explored → $O(b^{d+1})$




- Its space complexity is $O(b^d)$, i.e. the number of nodes in the frontier.

- If goal-state is checked after explored → $O(b^{d+1})$

# Breadth-First Search

... is not a time and memory friendly choice.

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 110 | .11 ms | 107 kB |
| 4 | 11110 | 11 ms | 10.6 MB |
| 6 | $10^6$ | 1.1 s | 1 GB |
| 8 | $10^8$ | 2 min | 103 GB |
| 10 | $10^{10}$ | 3 h | 10 TB |
| 12 | $10^{12}$ | 13 days | 1 PB |
| 14 | $10^{14}$ | 3.5 years | 99 PB |
| 16 | $10^{16}$ | 350 years | 10 EB |

b = 10; 1 million nodes/second; 1000 bytes/node

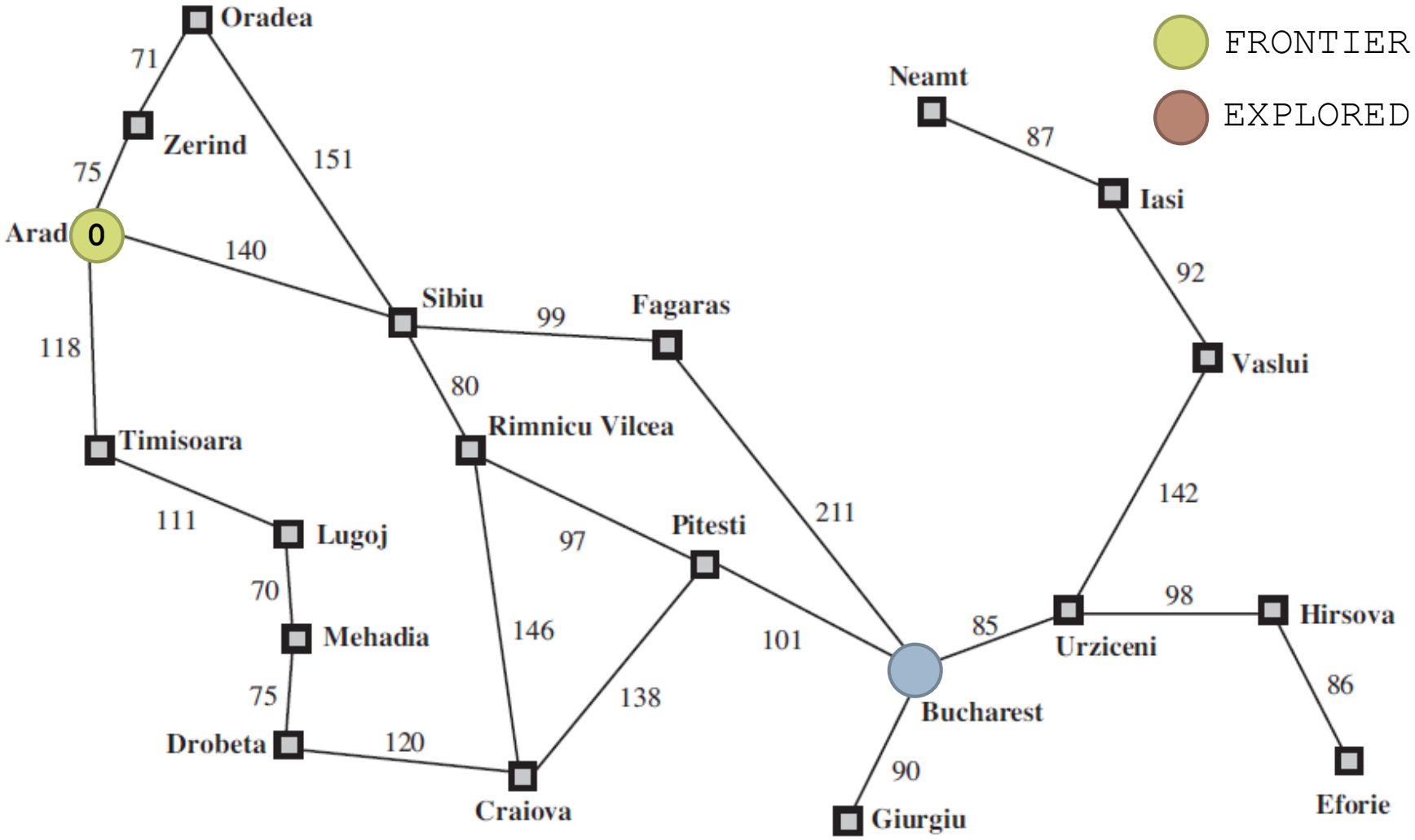
# Uniform-Cost Search

- We need an optimal algorithm for any step cost function, i.e. when the step costs are not equal.
- Expand the root node, then expand the node with the lowest path cost, …
- Implement the frontier as a *priority queue* ordered by path cost function.
- Add a test to see if a better path is found in a node in the frontier to update priority queue order.

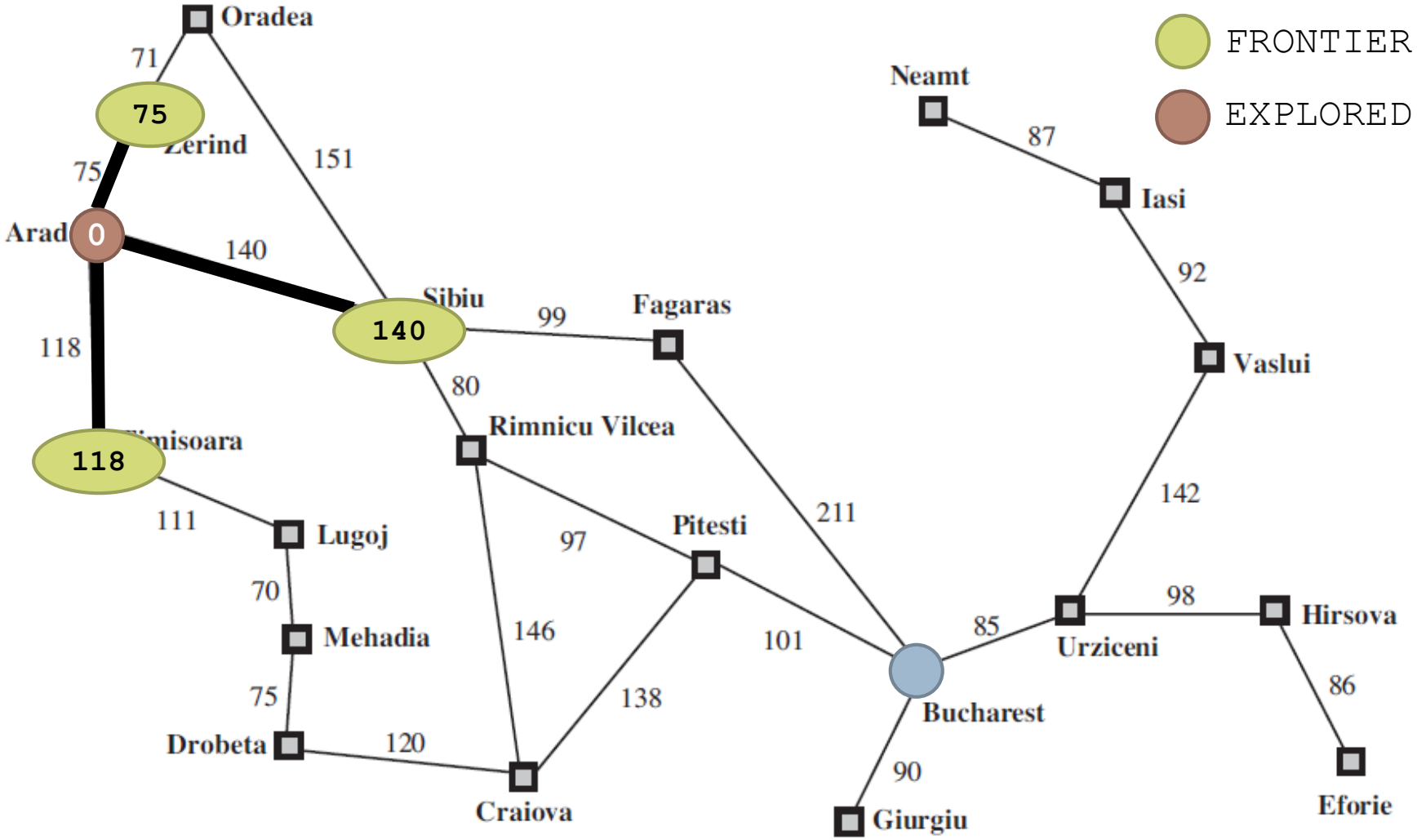
# Uniform-Cost Search (for reference)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, containing only node
    explored ← an empty set
    loop do
        if EMPTY?( frontier) then return failure
        node ← POP( frontier ) /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

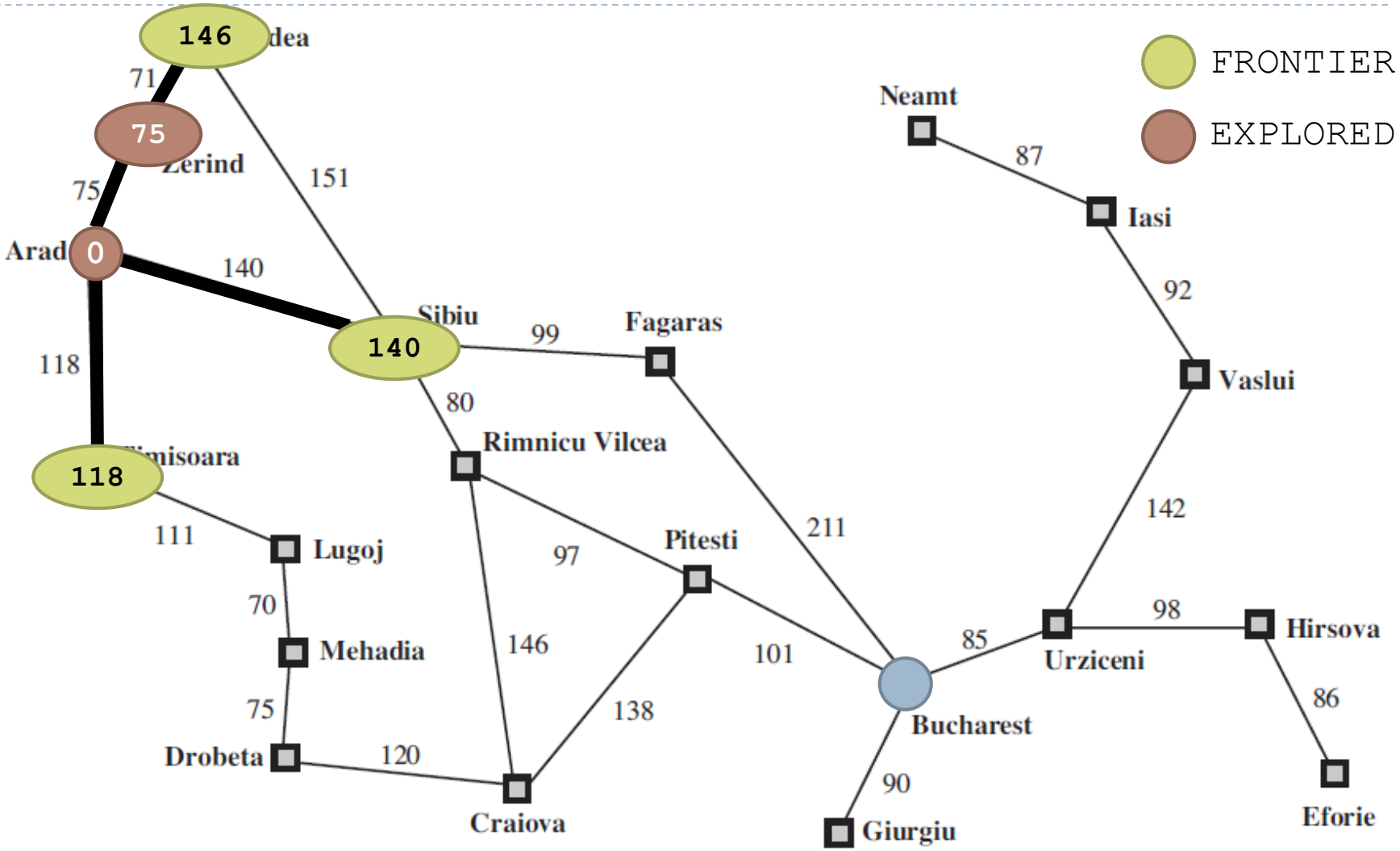
# Uniform-Cost Search for Route Finding



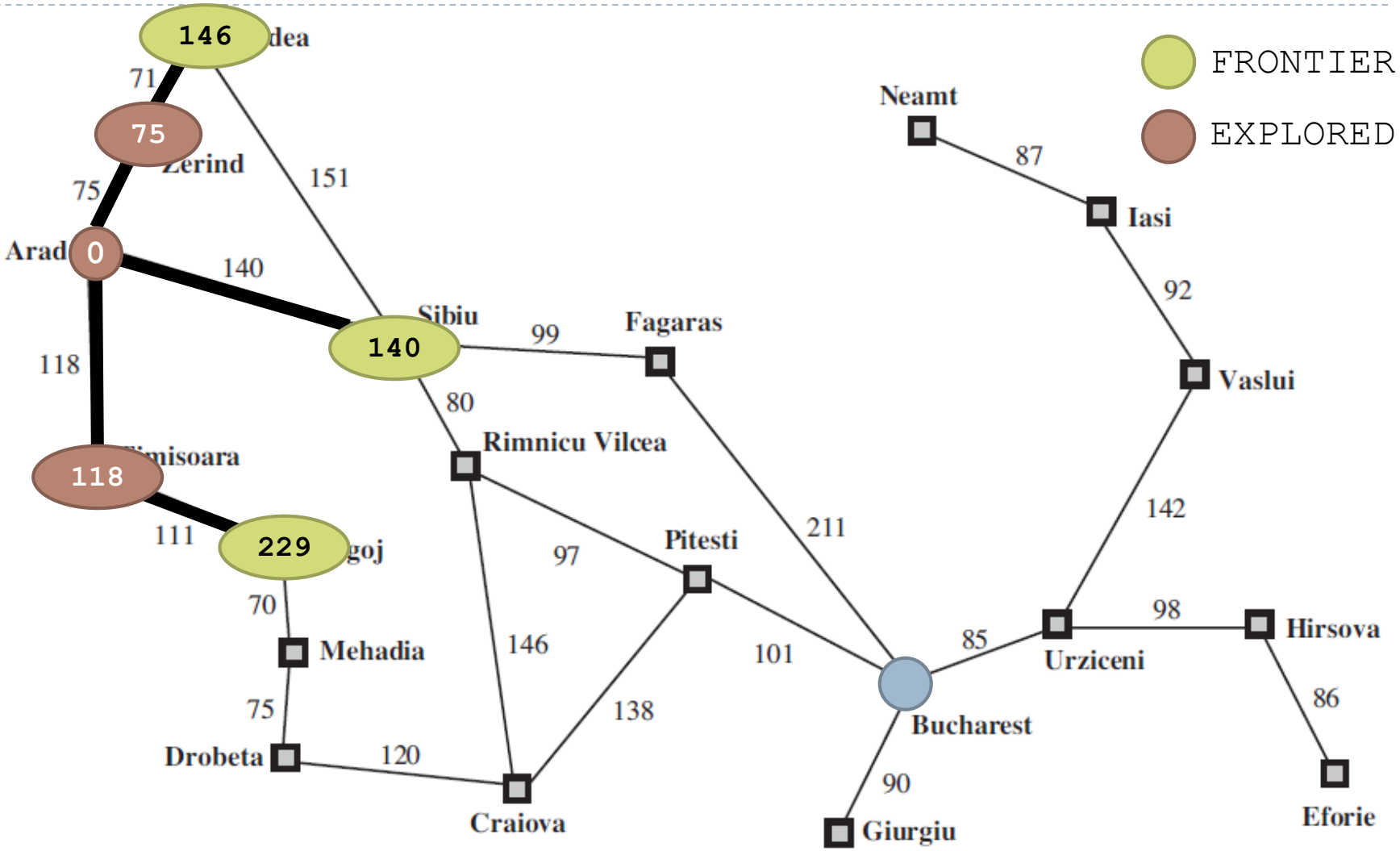
# Uniform-Cost Search for Route Finding



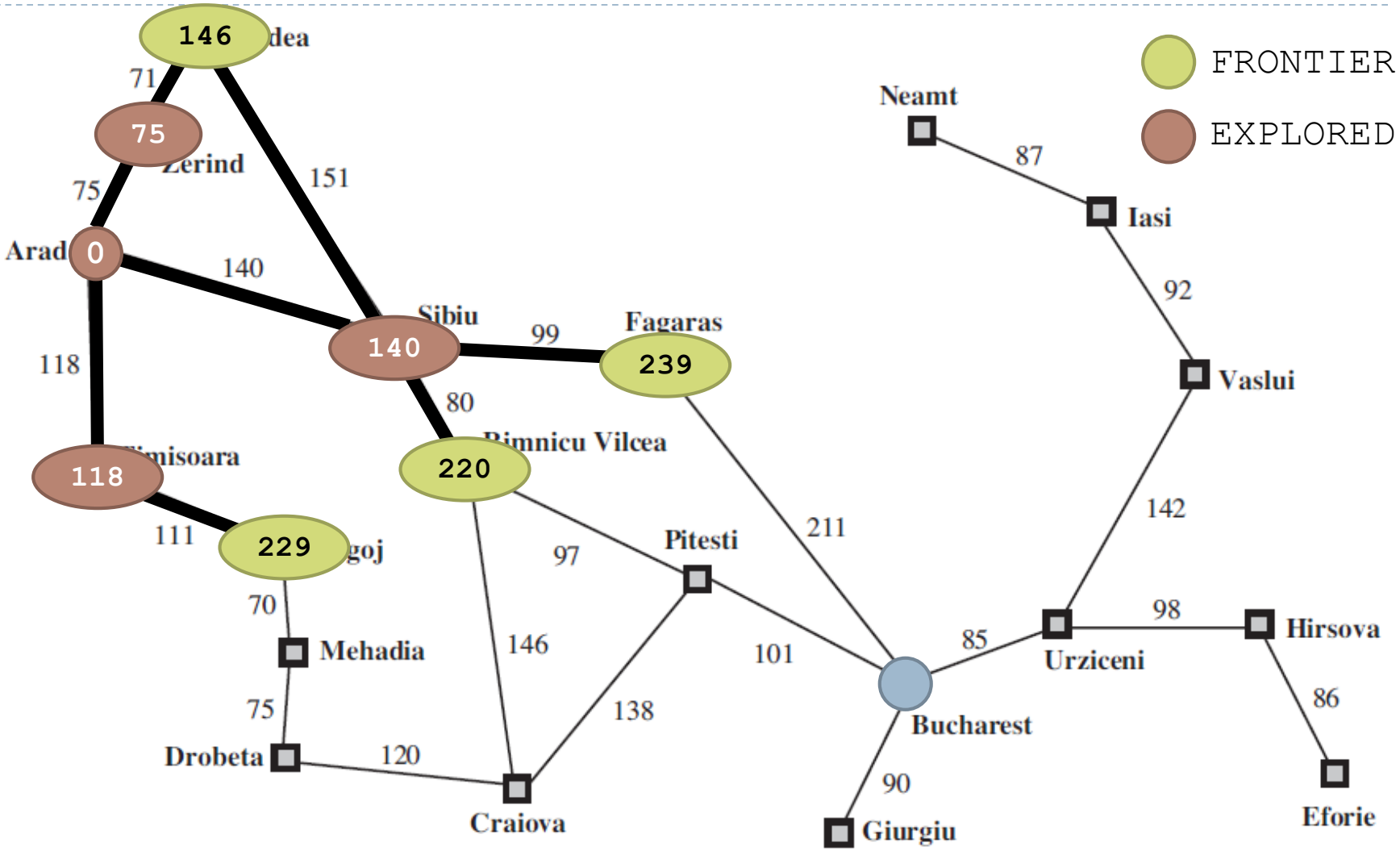
# Uniform-Cost Search for Route Finding



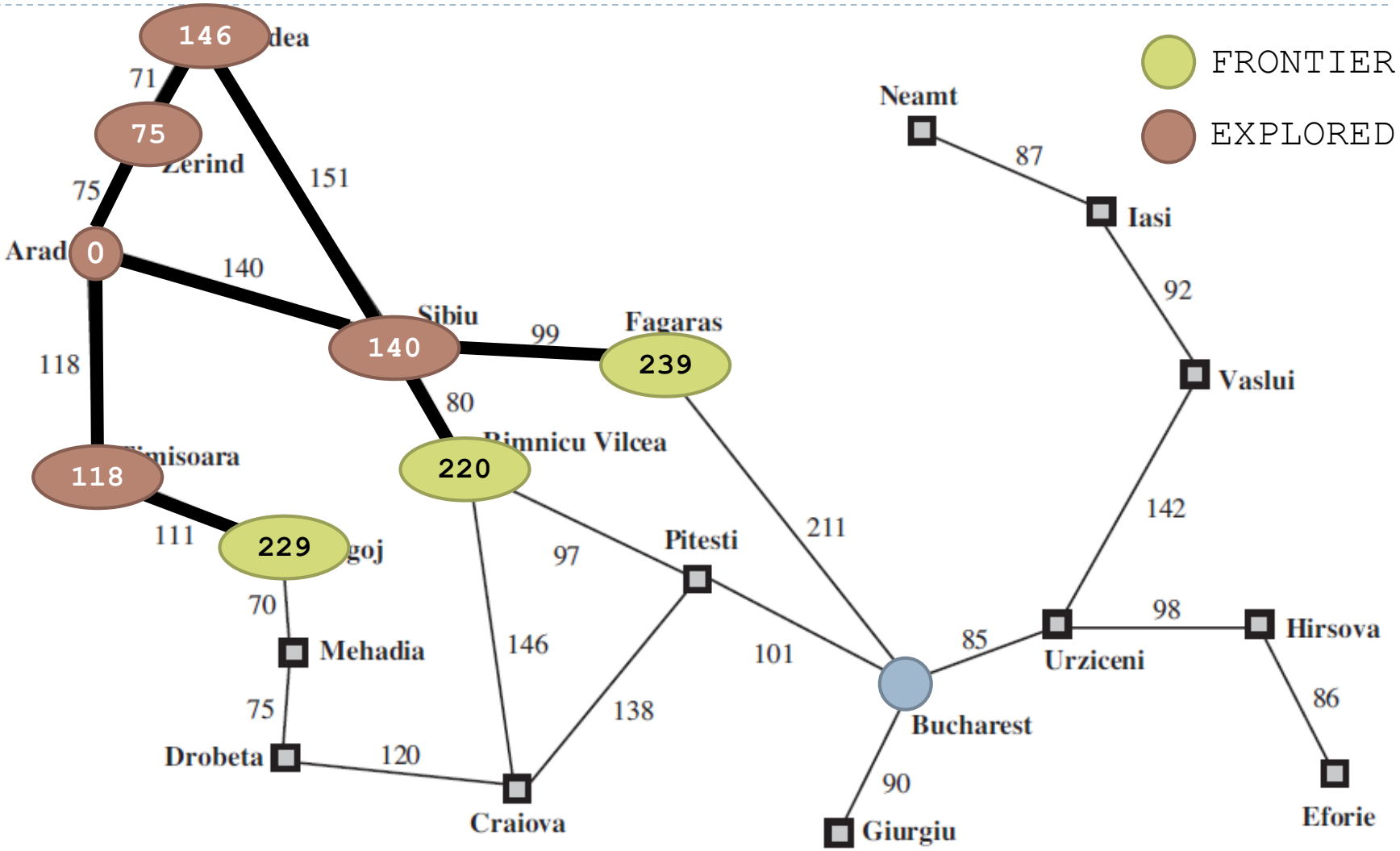
# Uniform-Cost Search for Route Finding



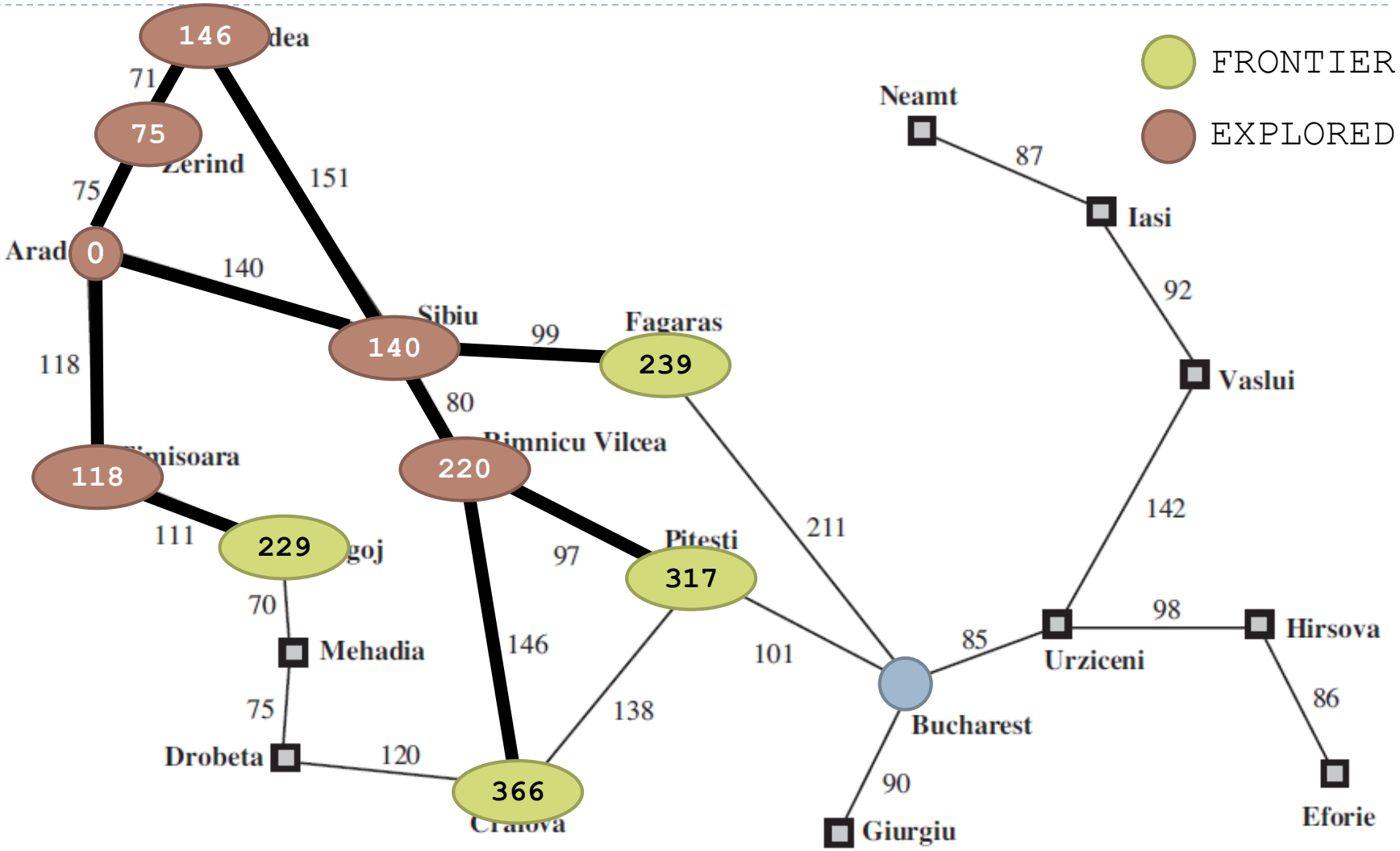
# Uniform-Cost Search for Route Finding



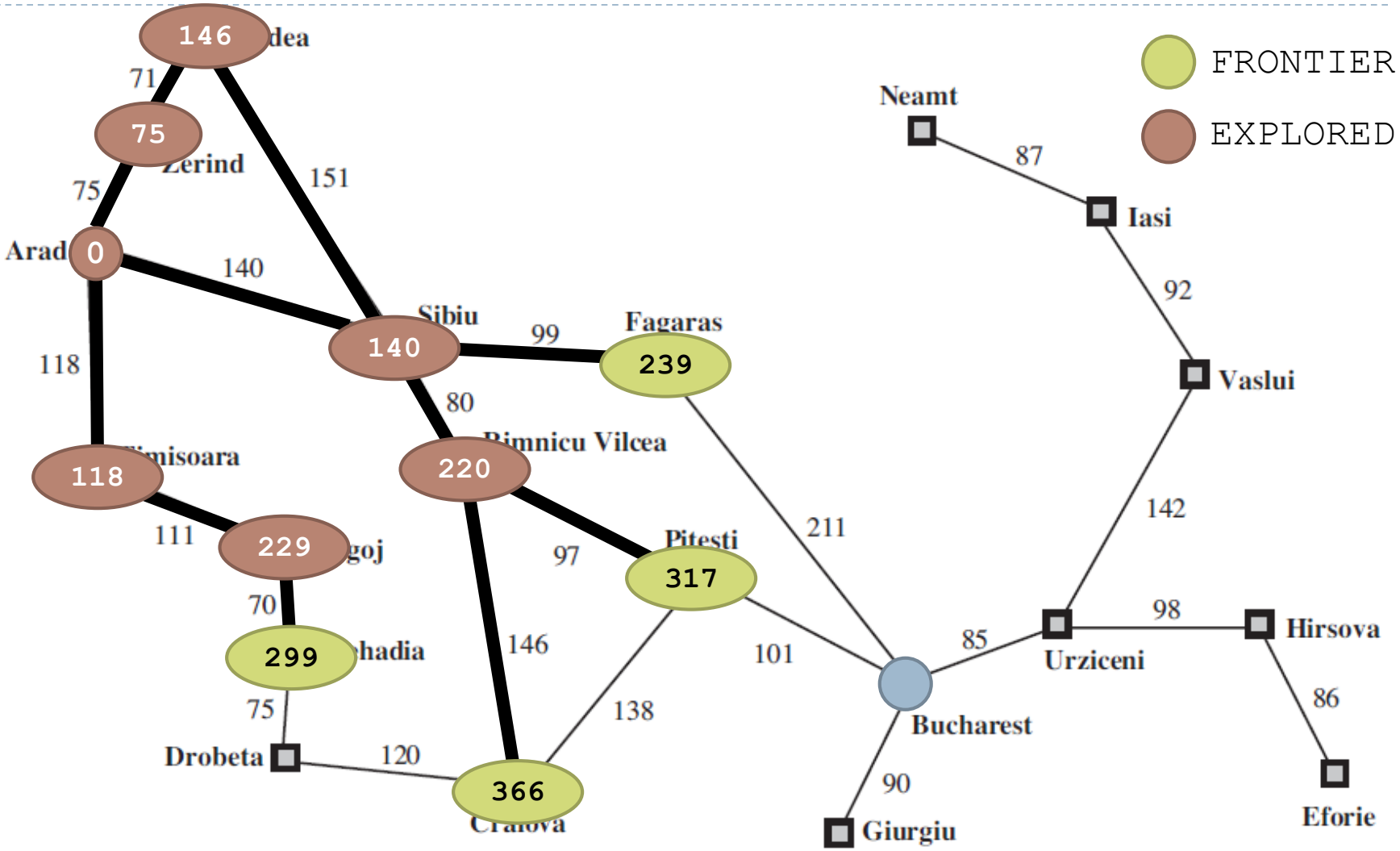
# Uniform-Cost Search for Route Finding



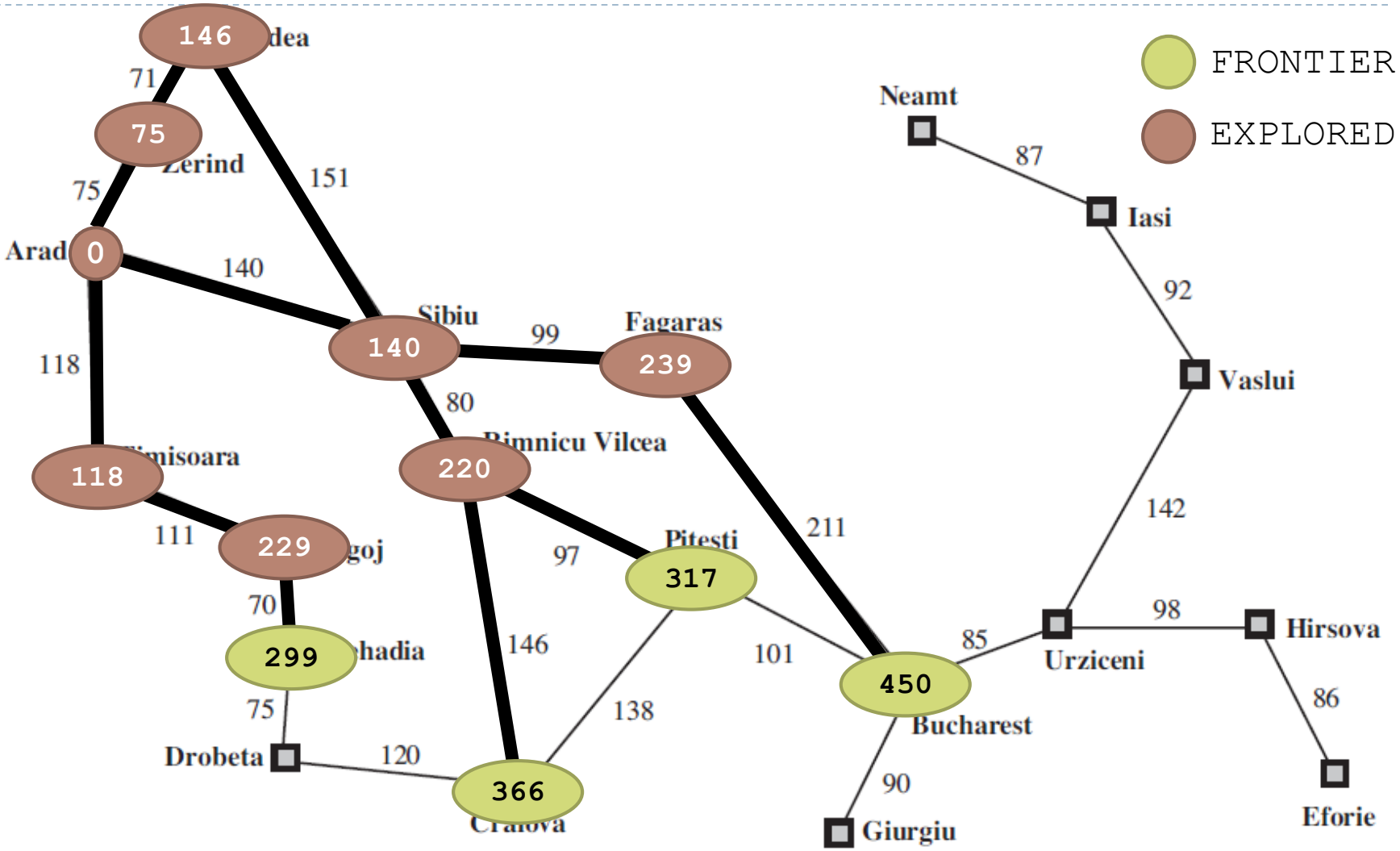
# Uniform-Cost Search for Route Finding



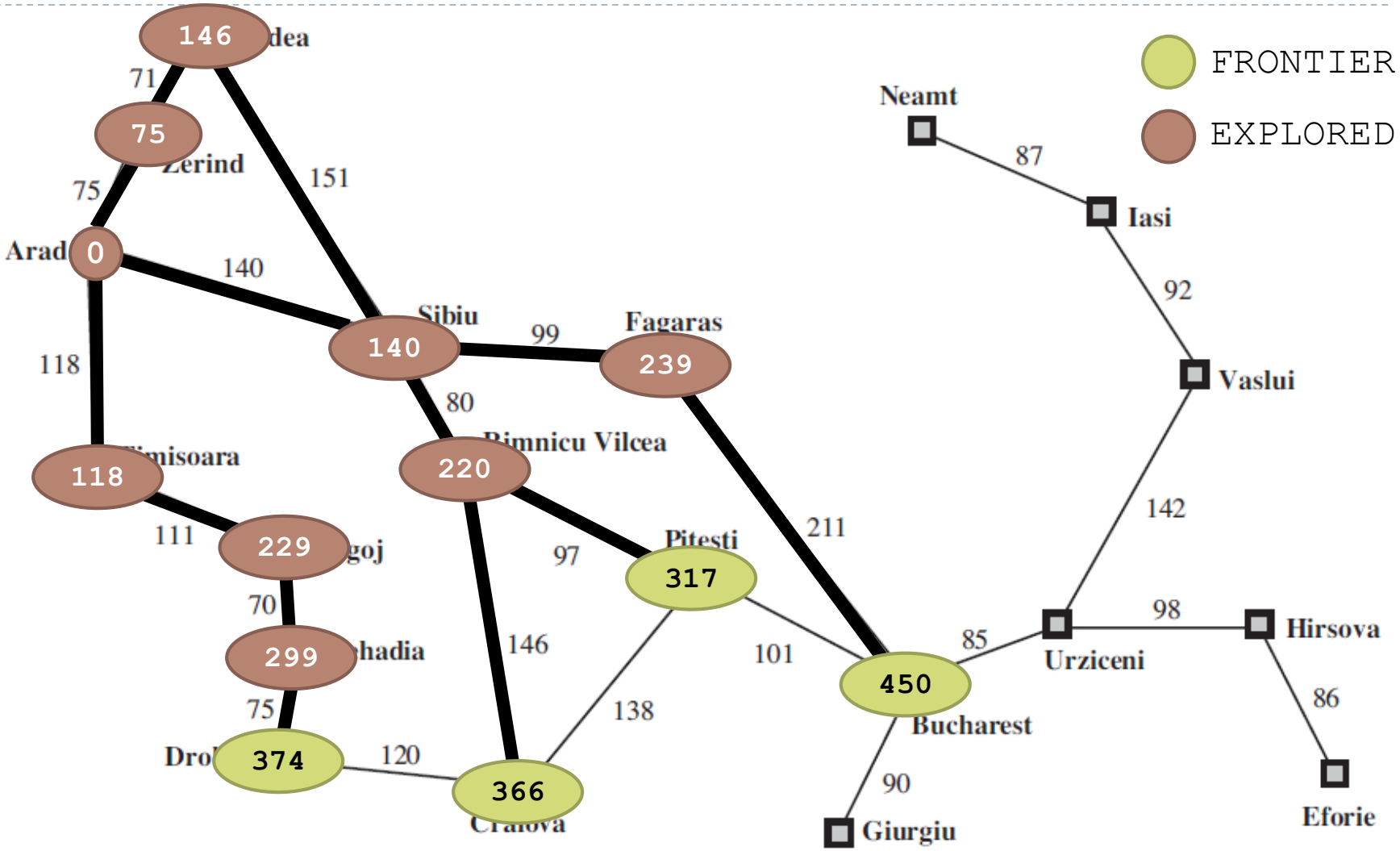
# Uniform-Cost Search for Route Finding



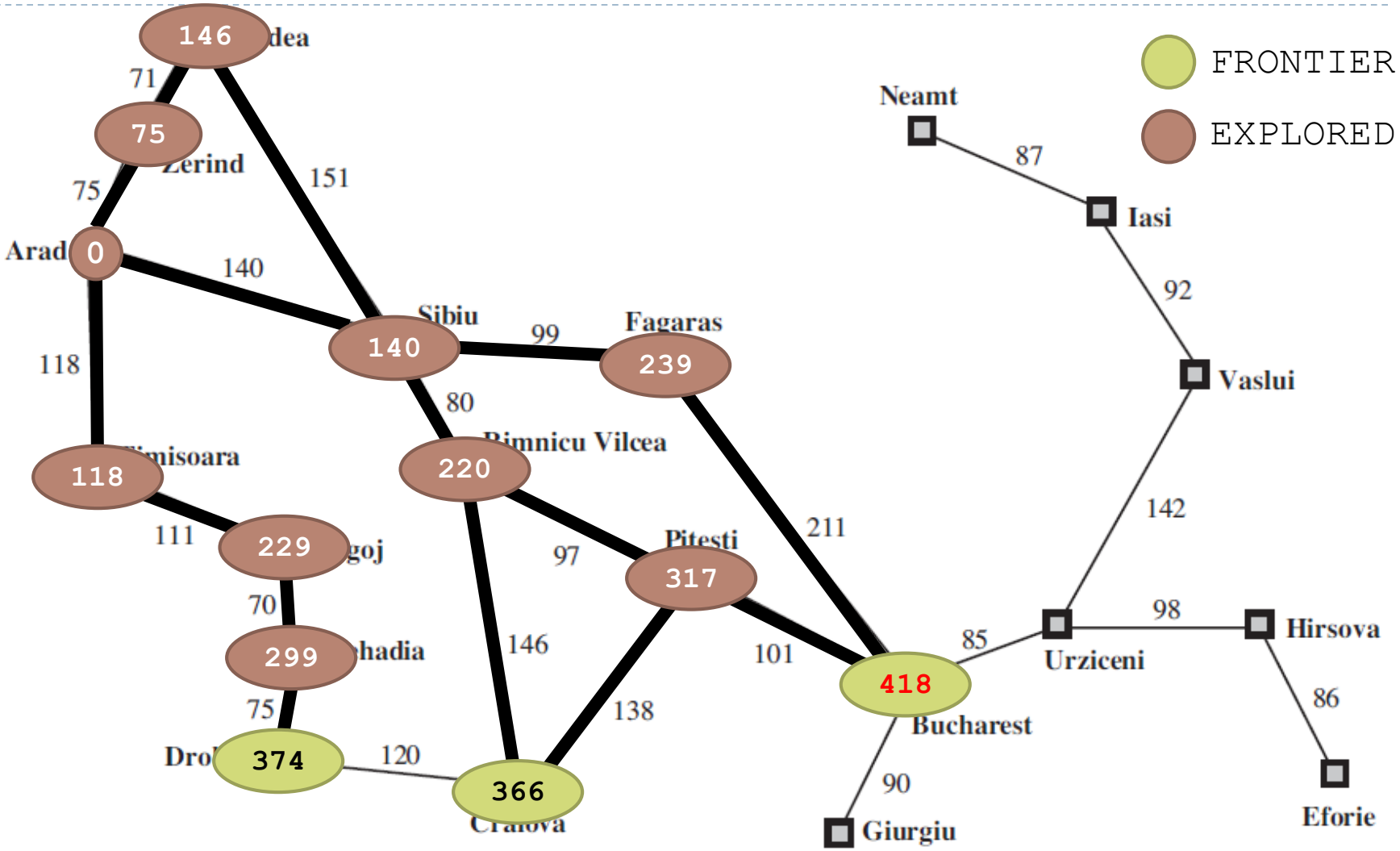
# Uniform-Cost Search for Route Finding



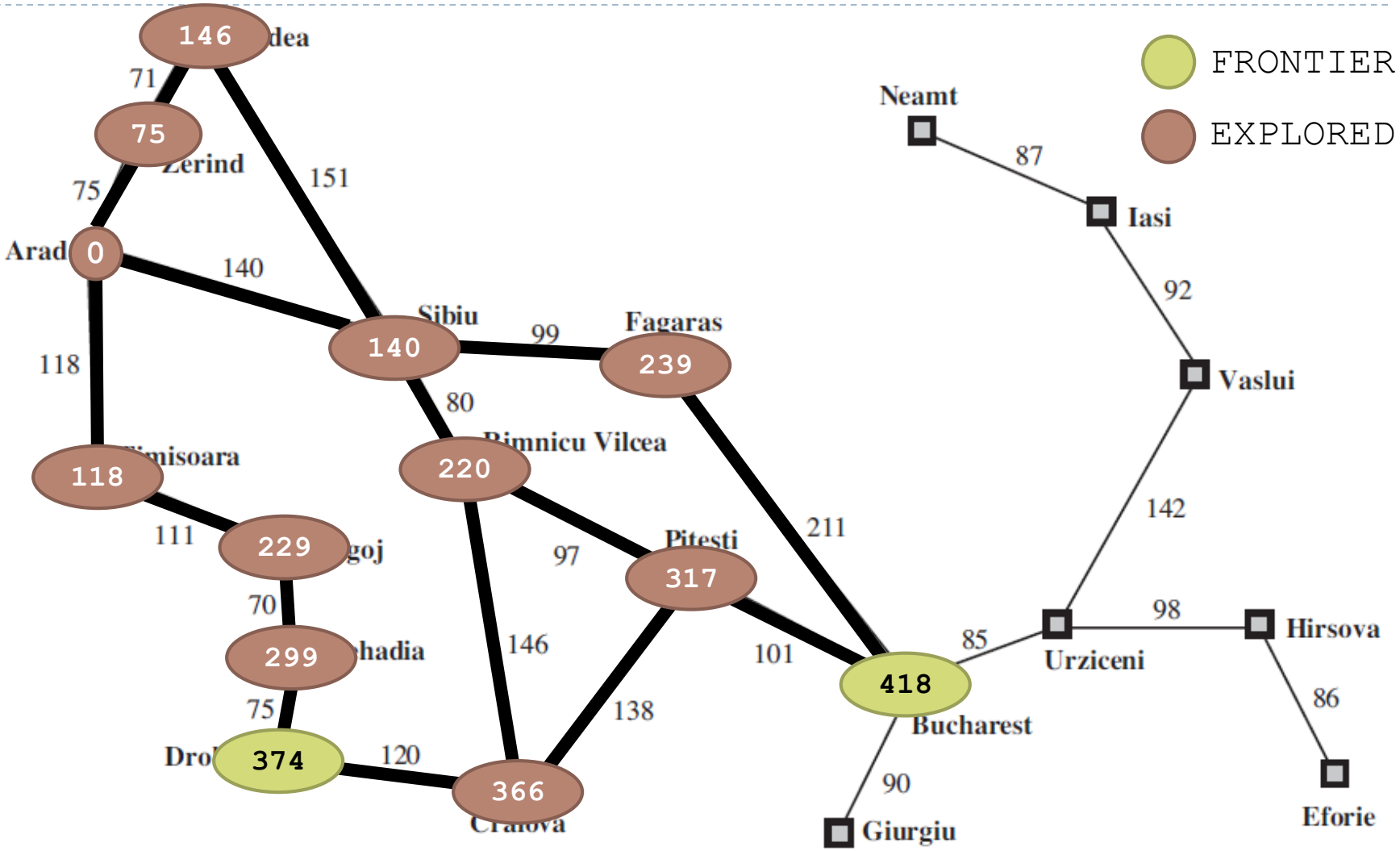
# Uniform-Cost Search for Route Finding



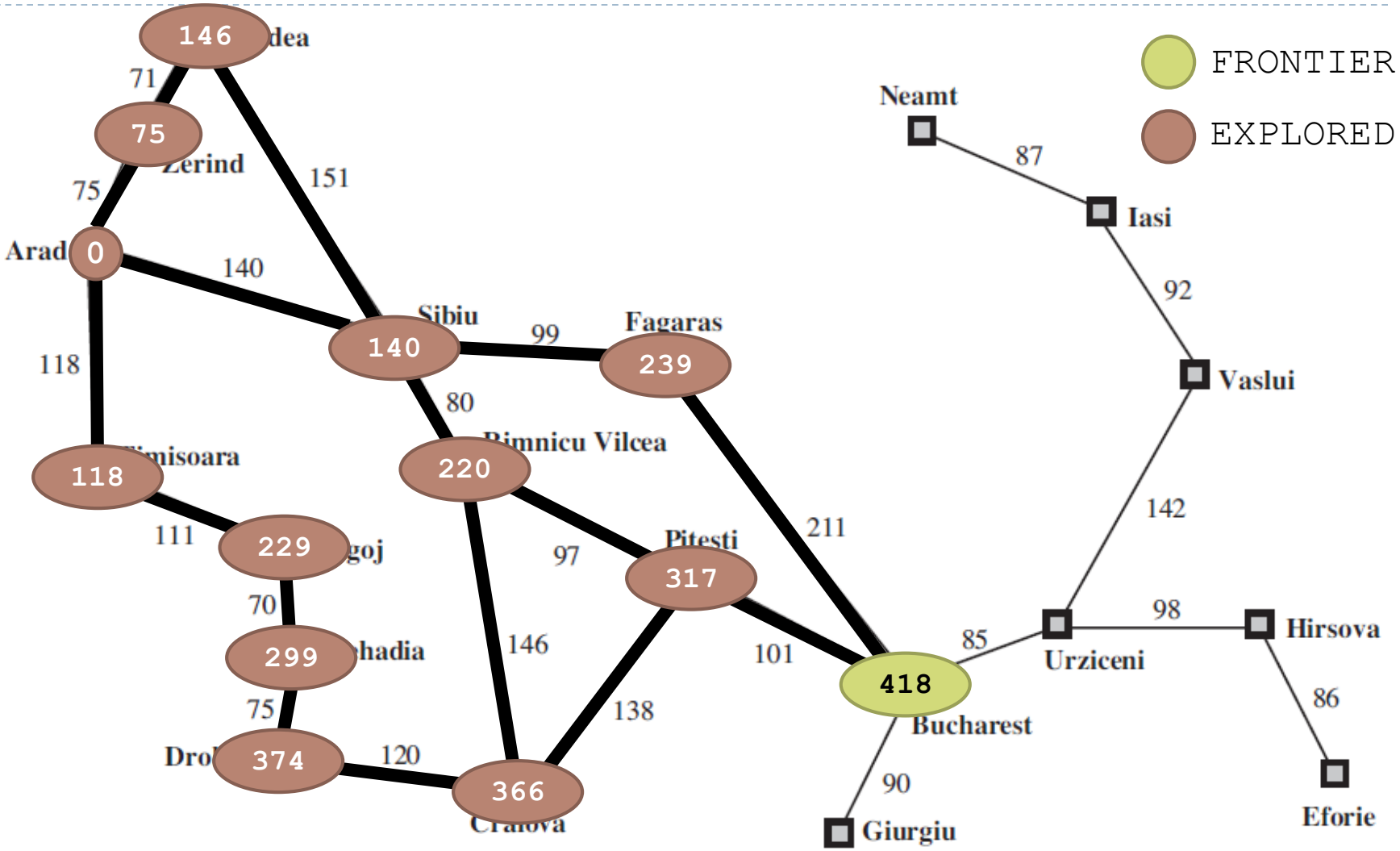
# Uniform-Cost Search for Route Finding



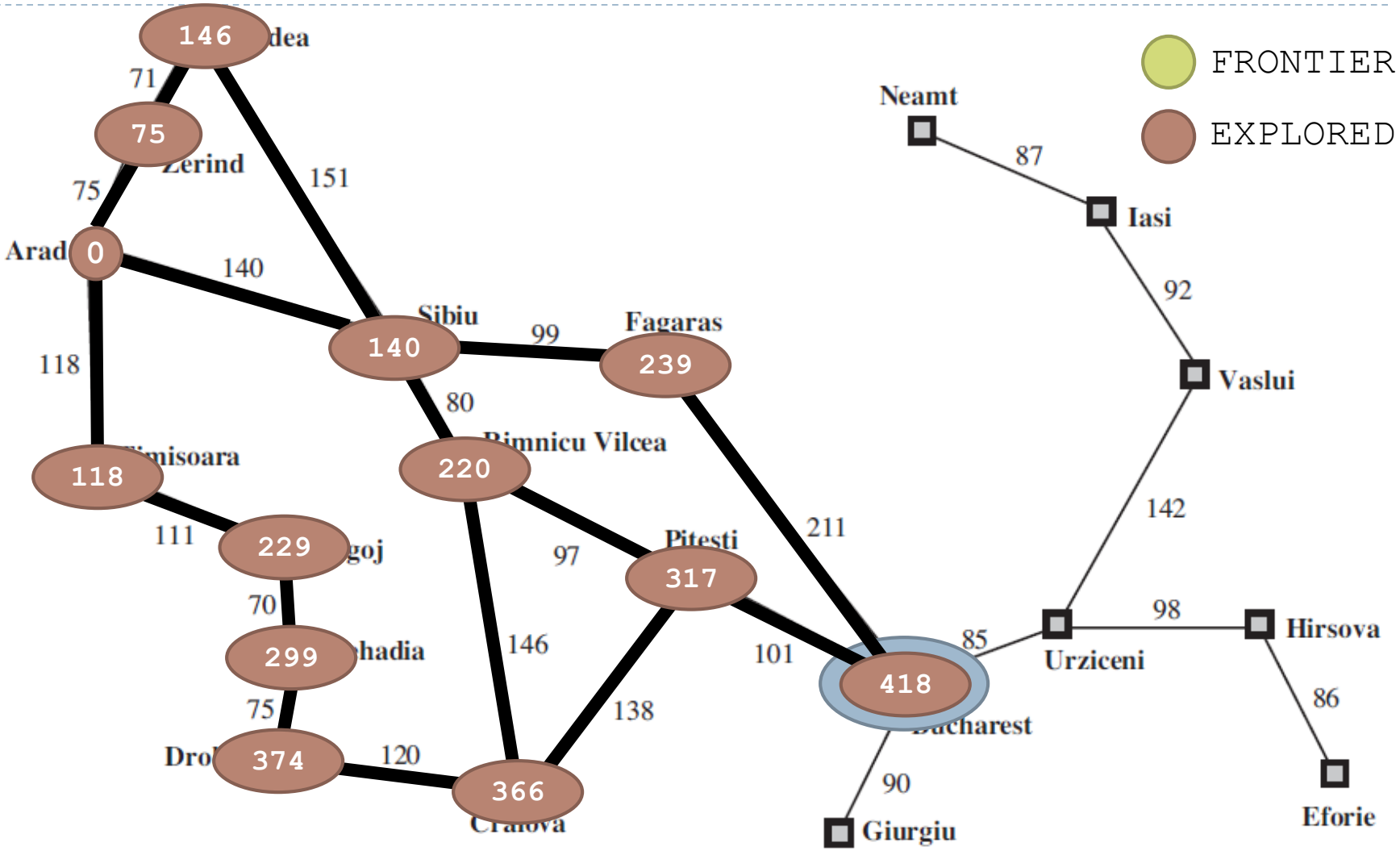
# Uniform-Cost Search for Route Finding



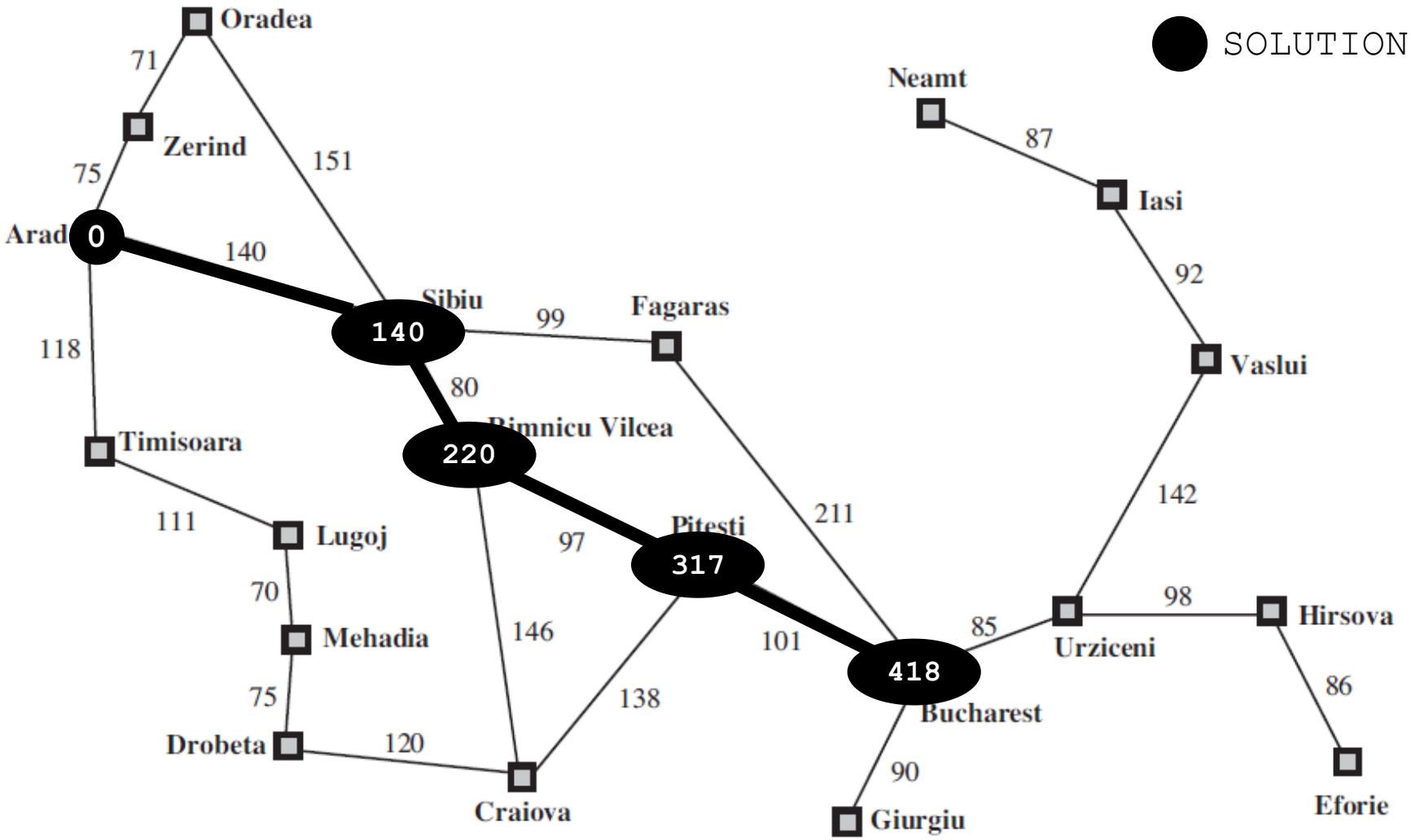
# Uniform-Cost Search for Route Finding



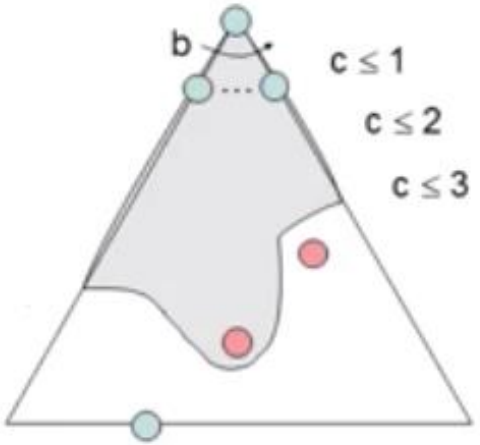
# Uniform-Cost Search for Route Finding



# Uniform-Cost Search for Route Finding



# Uniform-Cost Search




- It is optimal, i.e. the first solution we find is the minimum cost one.
- It is complete if every step cost exceeds some small positive constant, ε

  There should not be zero or negative cost loops
- Its space and time complexity is

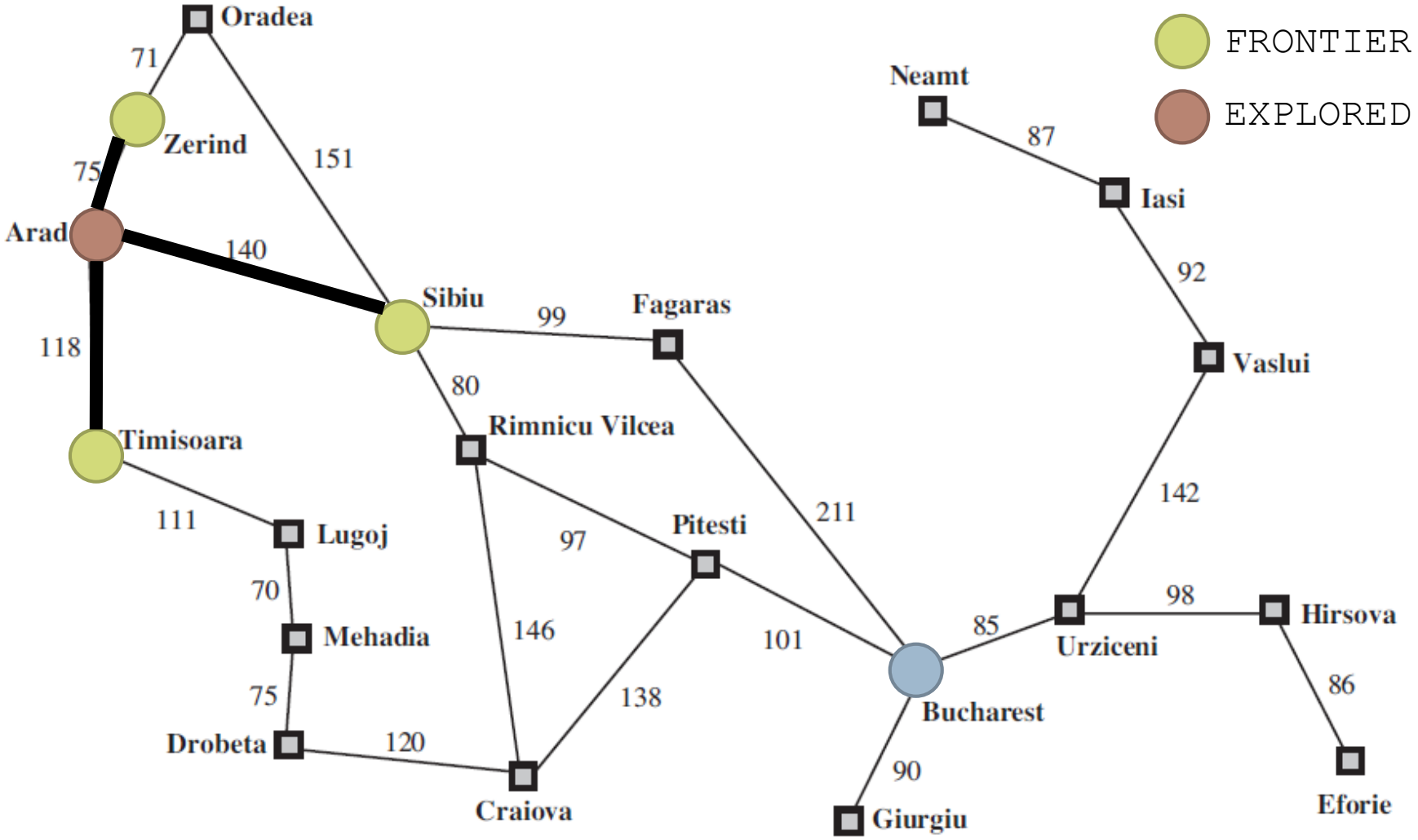
  $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, where C* is the optimal path cost and ε is the minimal action cost.

  In case the step costs are all equal, this is equivalent to $O(b^{1+d})$

# Depth-First Search

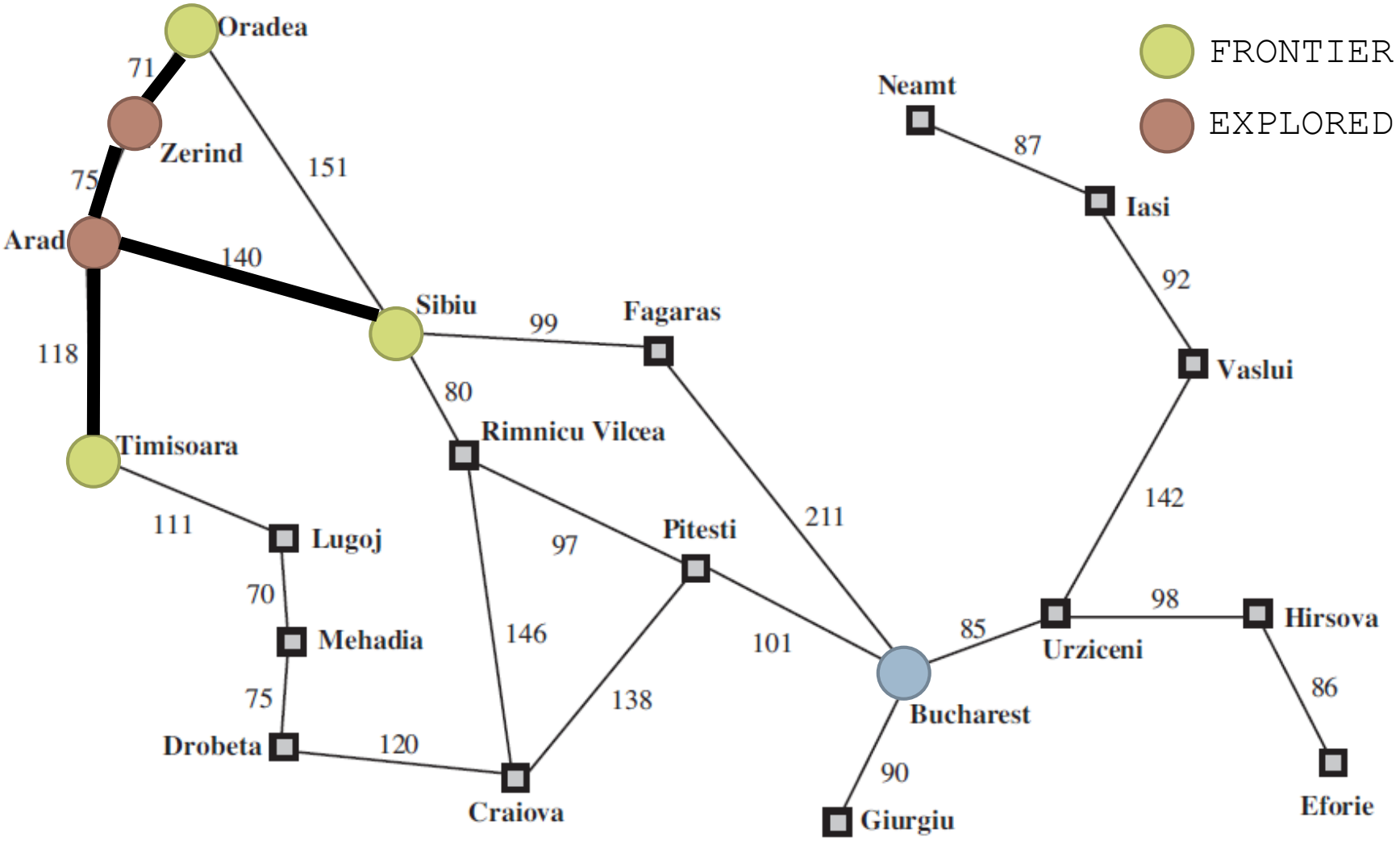
- Expand the root node, then keep expanding the deepest unexplored node.
- Implement the frontier as a LIFO queue (a stack).

# Depth-First Search for Route Finding



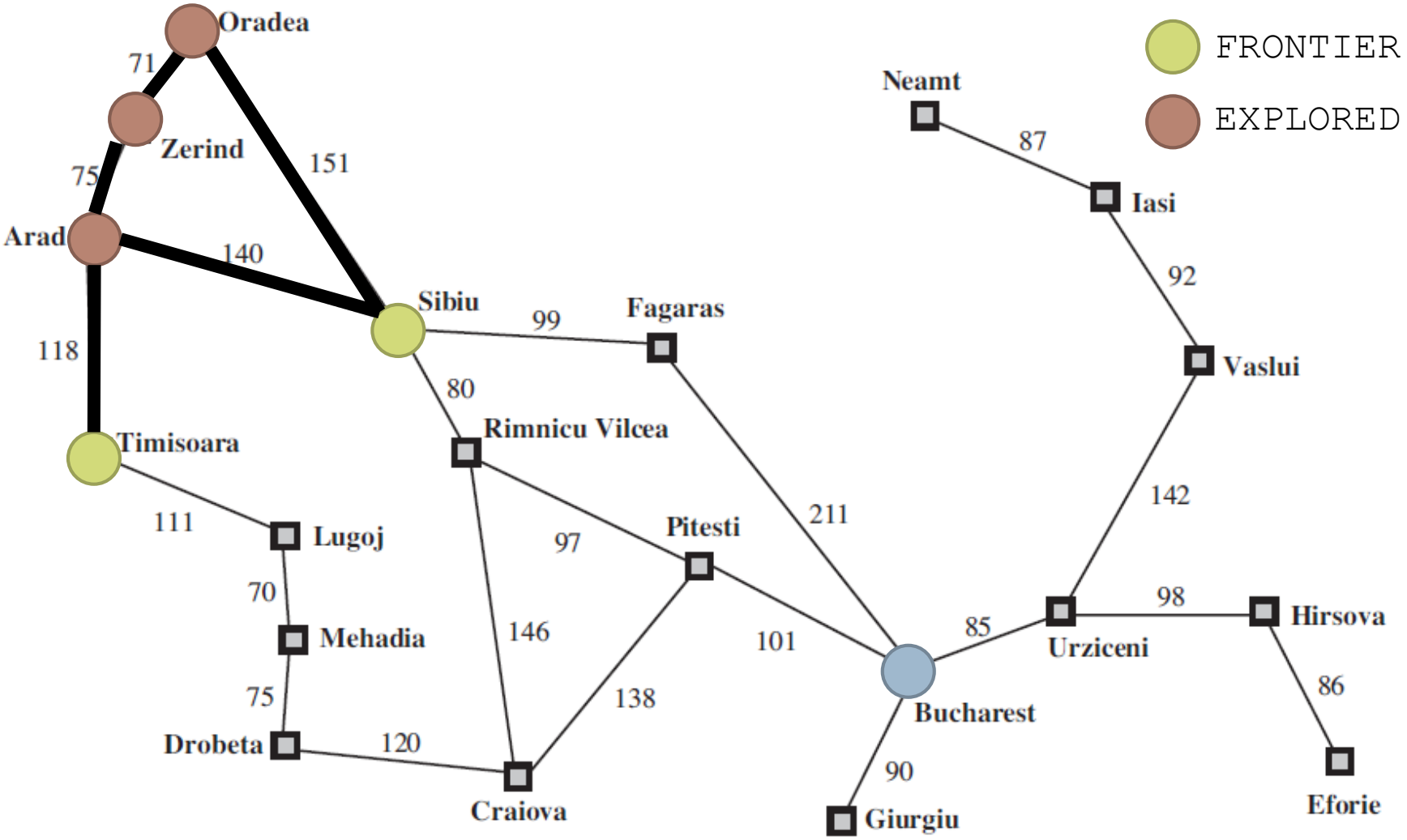

- Break the ties by expanding the node starting with alphabetically last character.

# Depth-First Search for Route Finding



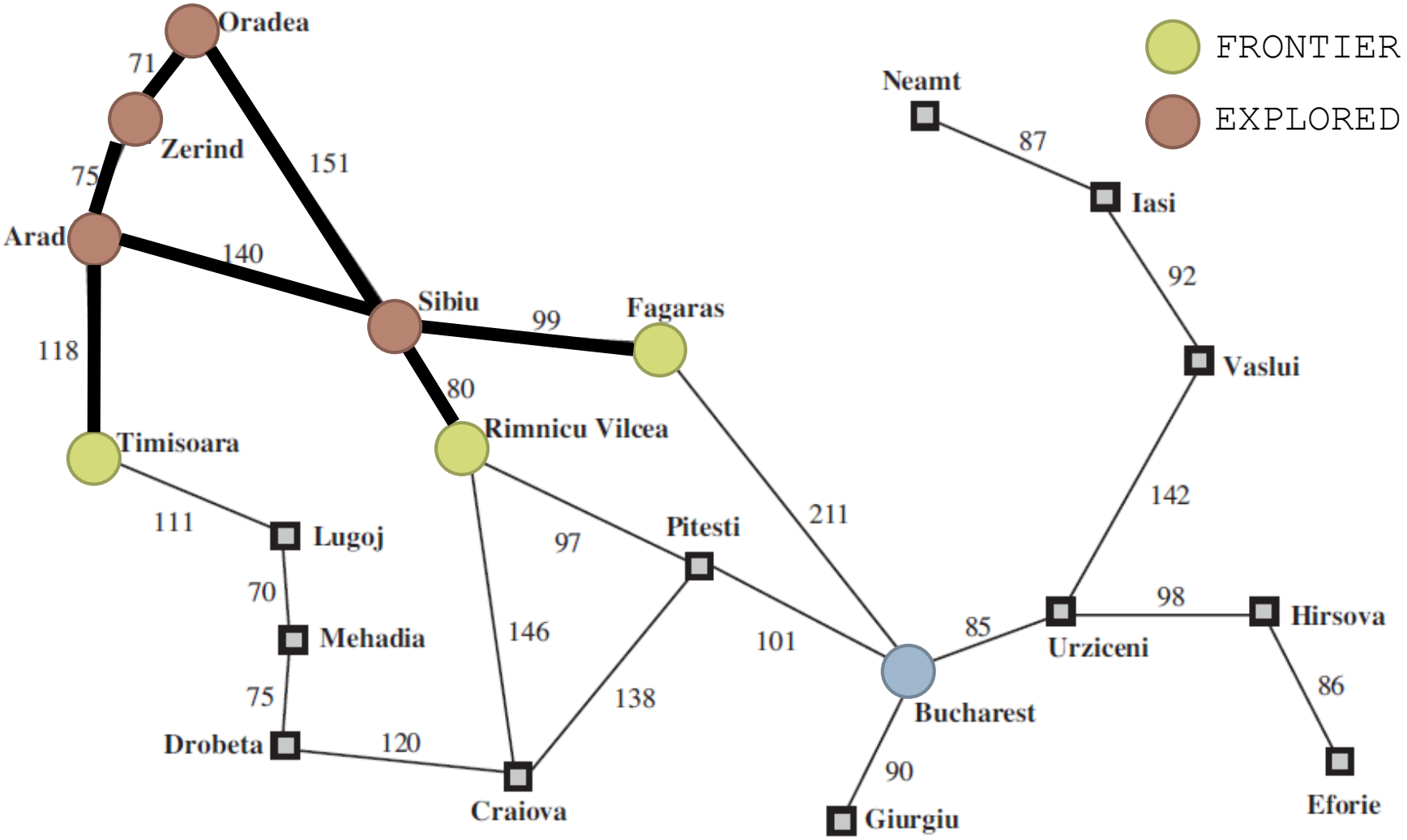

- Break the ties by expanding the node starting with alphabetically last character.

# Depth-First Search for Route Finding



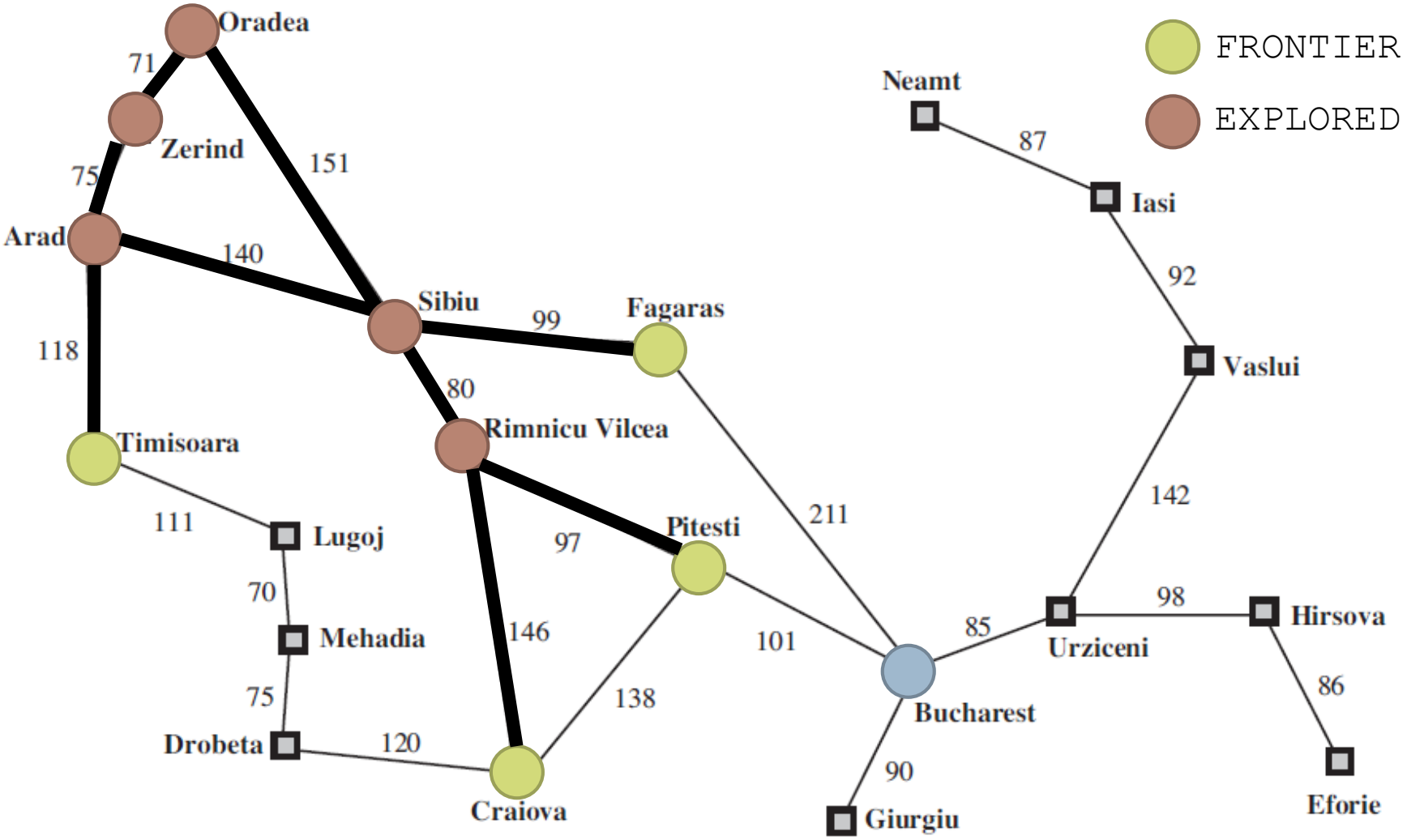

- Break the ties by expanding the node starting with alphabetically last character.

# Depth-First Search for Route Finding



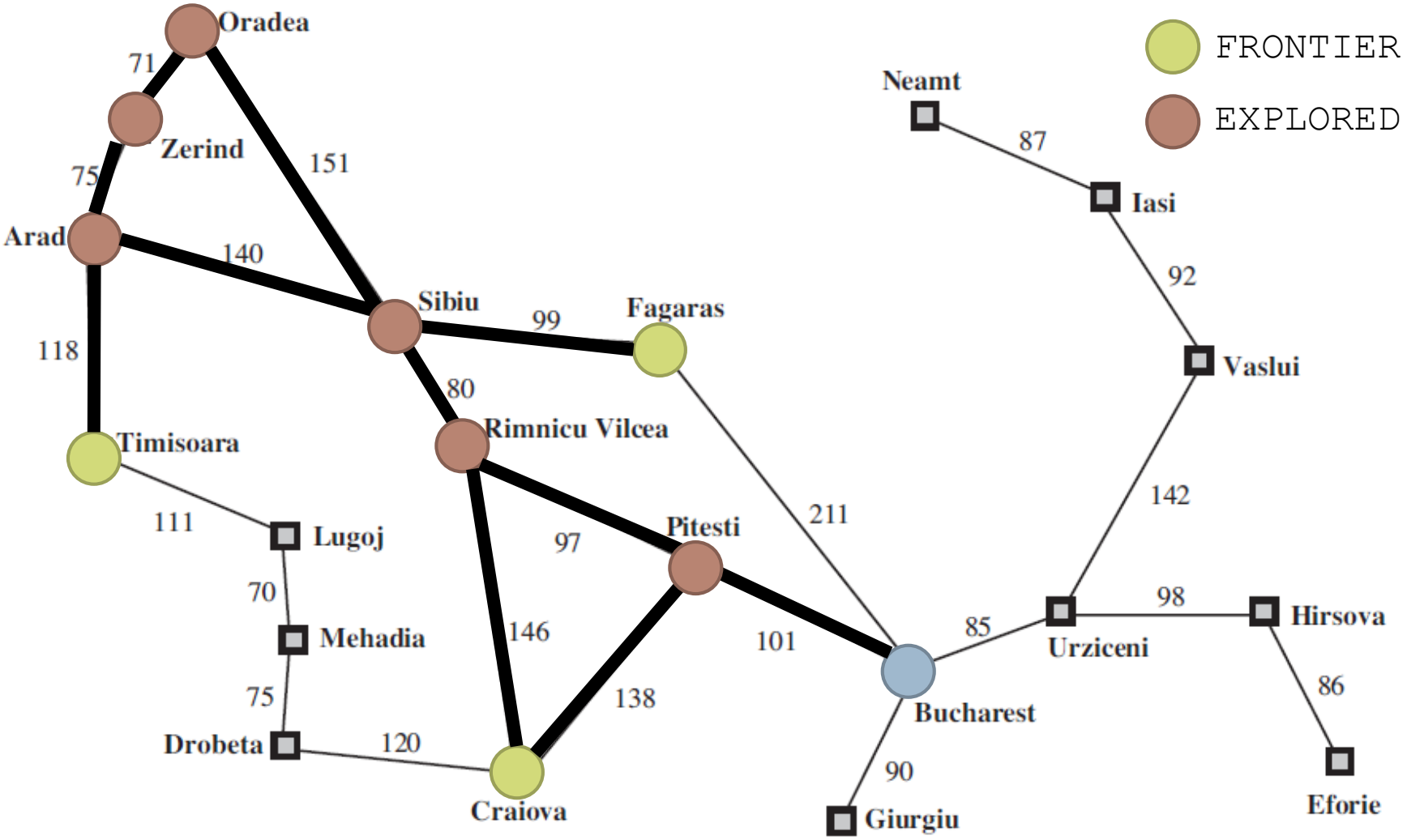

- Break the ties by expanding the node starting with alphabetically last character.

# Depth-First Search for Route Finding



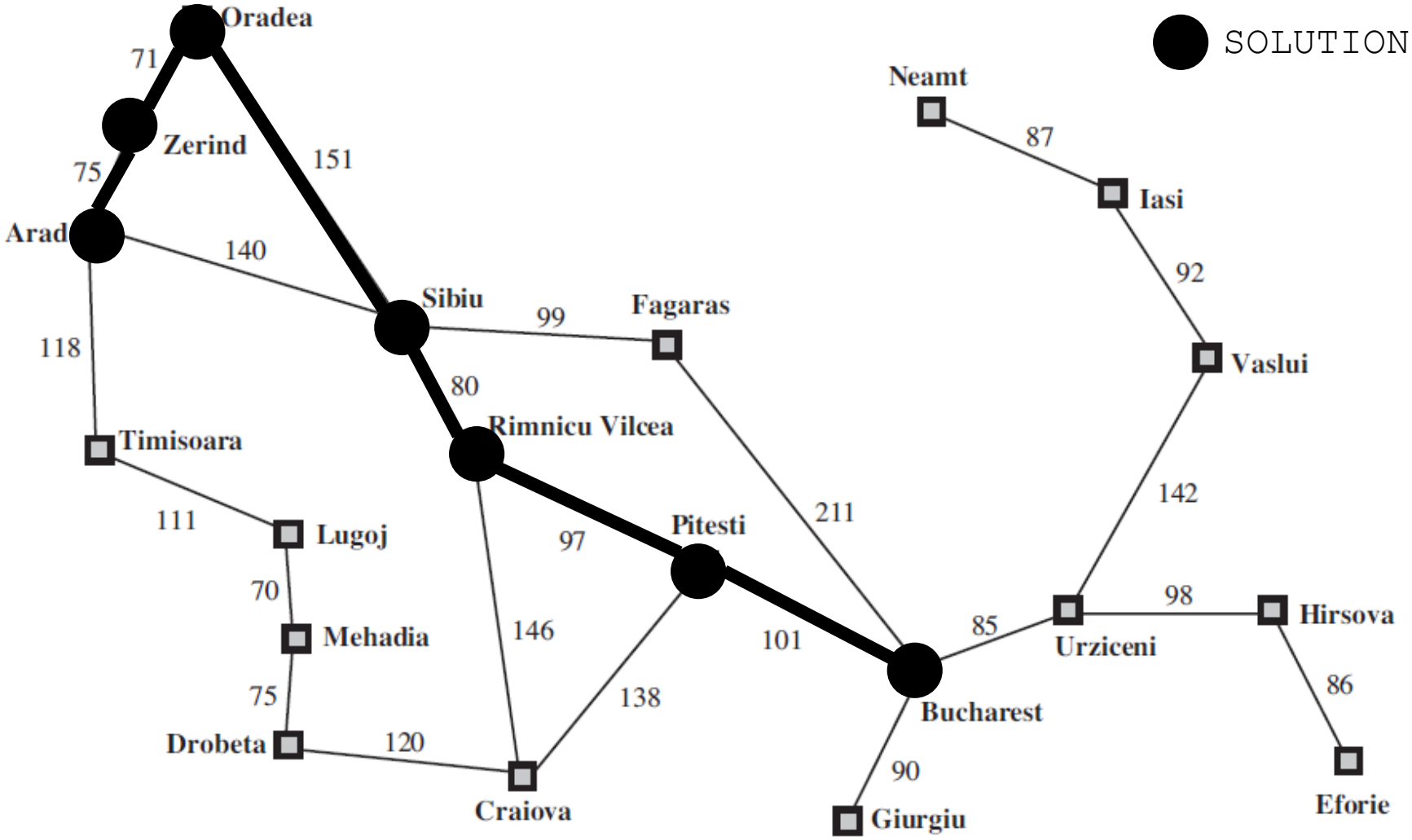

- Break the ties by expanding the node starting with alphabetically last character.

# Depth-First Search for Route Finding




- Break the ties by expanding the node starting with alphabetically last character.

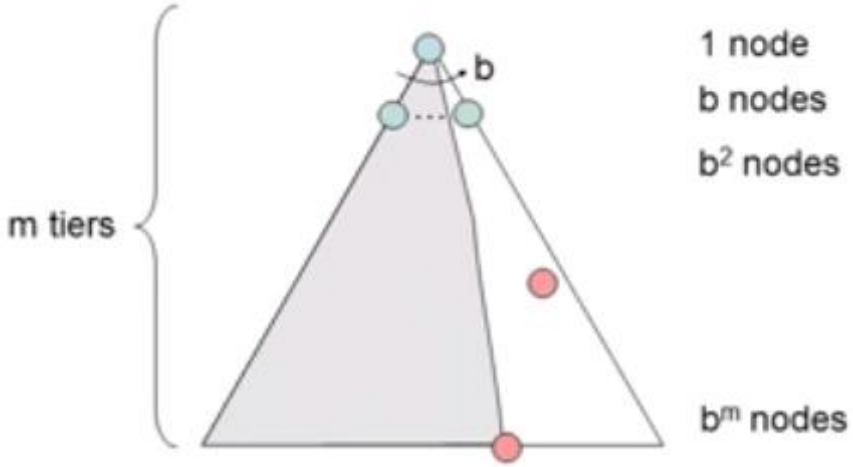
# Depth-First Search for Route Finding




- Is it the minimum cost path?

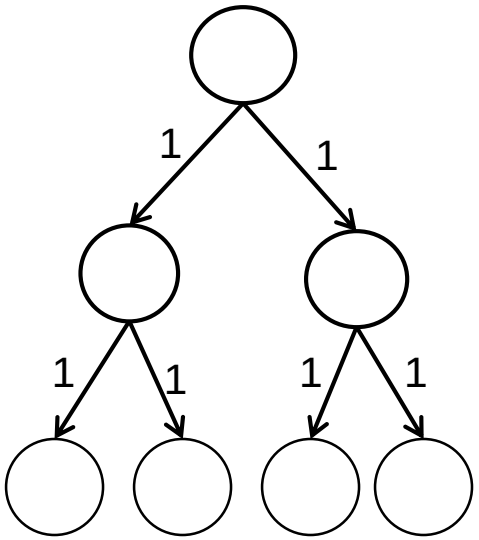
# Depth-First Search

- It is not optimal.
- It is not complete for an infinite search tree. It never finds the solution.
- Its time complexity is $O(b^m)$,
  b is the branching factor, m is the maximum depth of the tree.
- Its space complexity is O(bm),
  and only m nodes in the frontier.
- Its low space complexity makes it a good candidate for actual implementations.

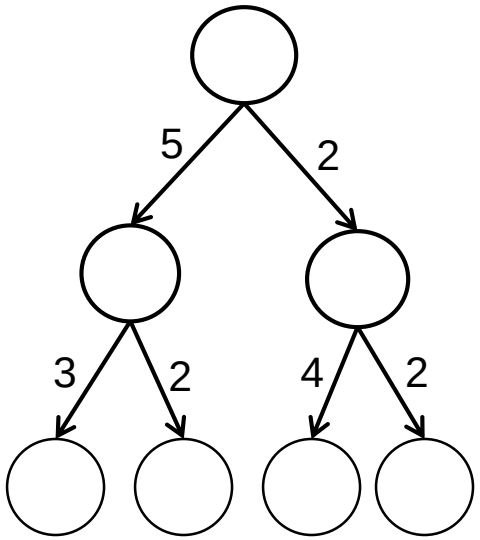



# Search comparison

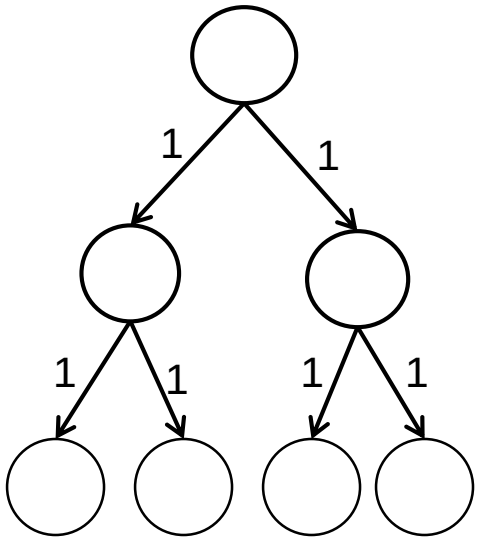
Breadth-first search




Uniform-cost search




Depth-first search



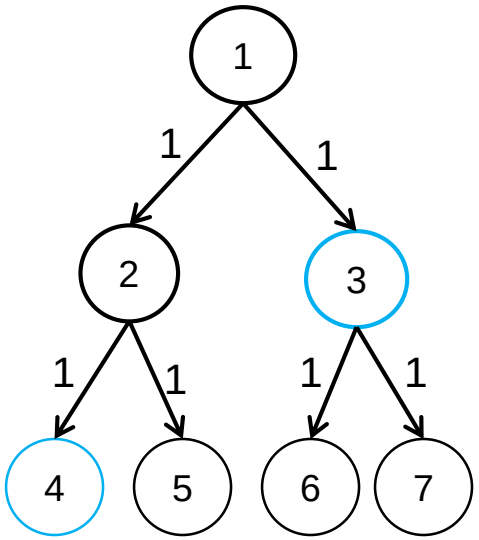

For each search method, tell in what order the tree is expanded?

For each search method, tell if it is optimal?

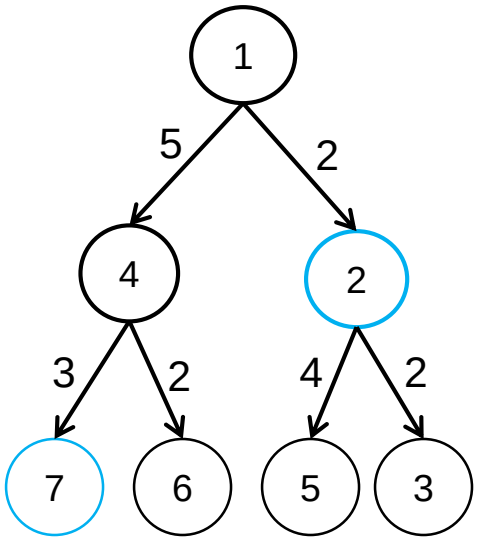
(solution can be at any node)

# Search comparison

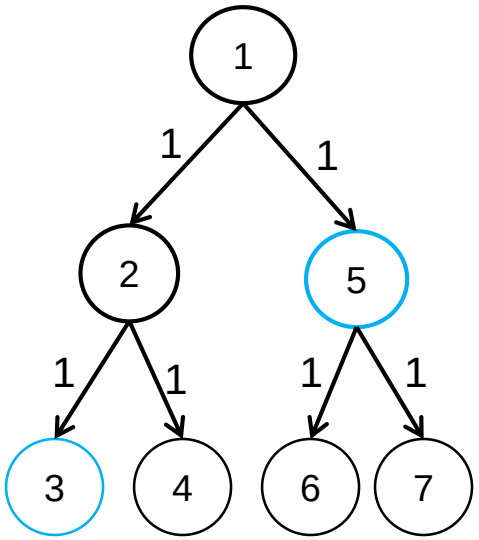
Breadth-first search




Uniform-cost search




Depth-first search



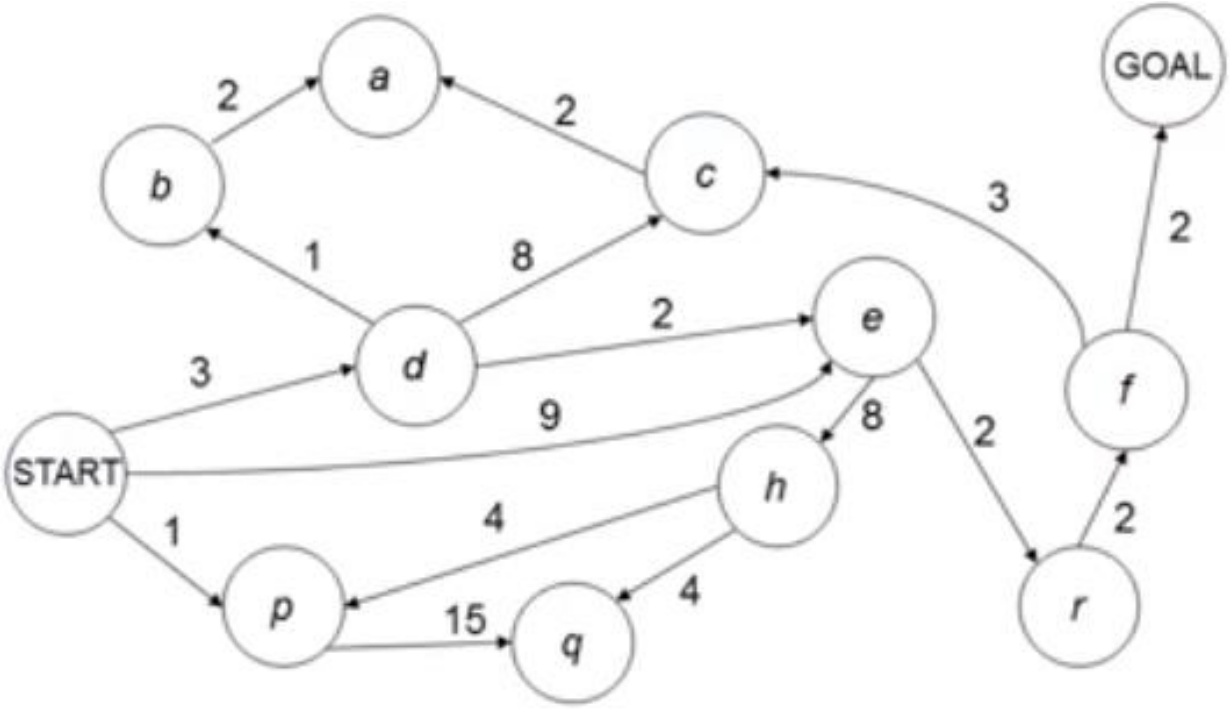

For each search method, tell in what order the tree is expanded? Lets assume blue circles are solutions, then depth-first search finds a higher cost solution before a minimum cost one.

# Variants of the Above

- Depth-Limited Search
  - Limit the depth to a certain level to avoid the failure of depth-first search in infinite state spaces.
- Iterative Deepening Depth-First Search
  - Iteratively increase the depth level limit.
- Bidirectional Search
  - Start from the initial and goal states and expand both, check for intersection of frontiers.

# Home-Exercise

- Apply Breadth-first, Depth-first, Uniform-cost.




- Break the ties by expanding the node starting with alphabetically first character.