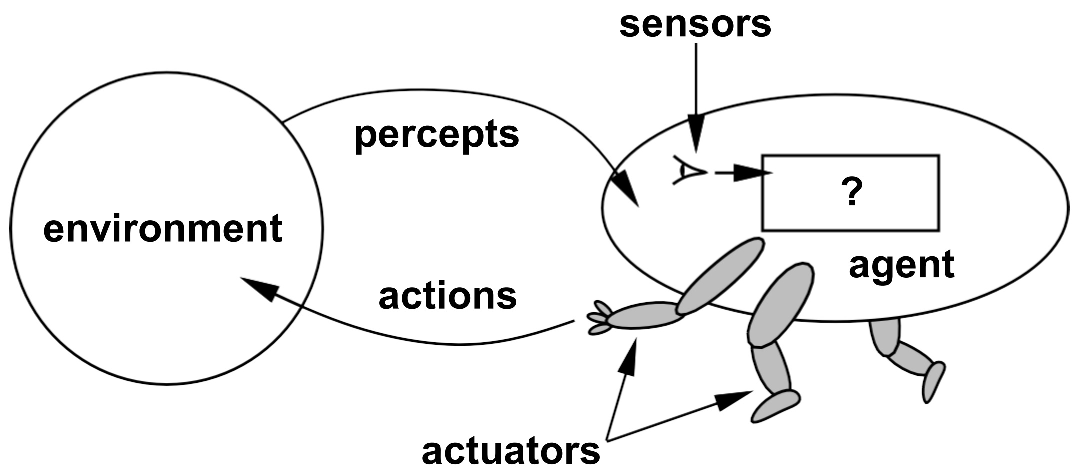
MSAI3105-01-Introduction to Artificial Intelligence

Ahmed El-Sayed, Ph.D.
Lecturer of AI and Deep Learning at the SPS, Clark University
Associate Professor of Electrical and Computer Engineering
University of Bridgeport
Director of the Laboratory for Advanced Control, Autonomous Systems, and Automation
(LACASA)

Agents and environments



Agents include humans, robots, softbots, thermostats, etc.

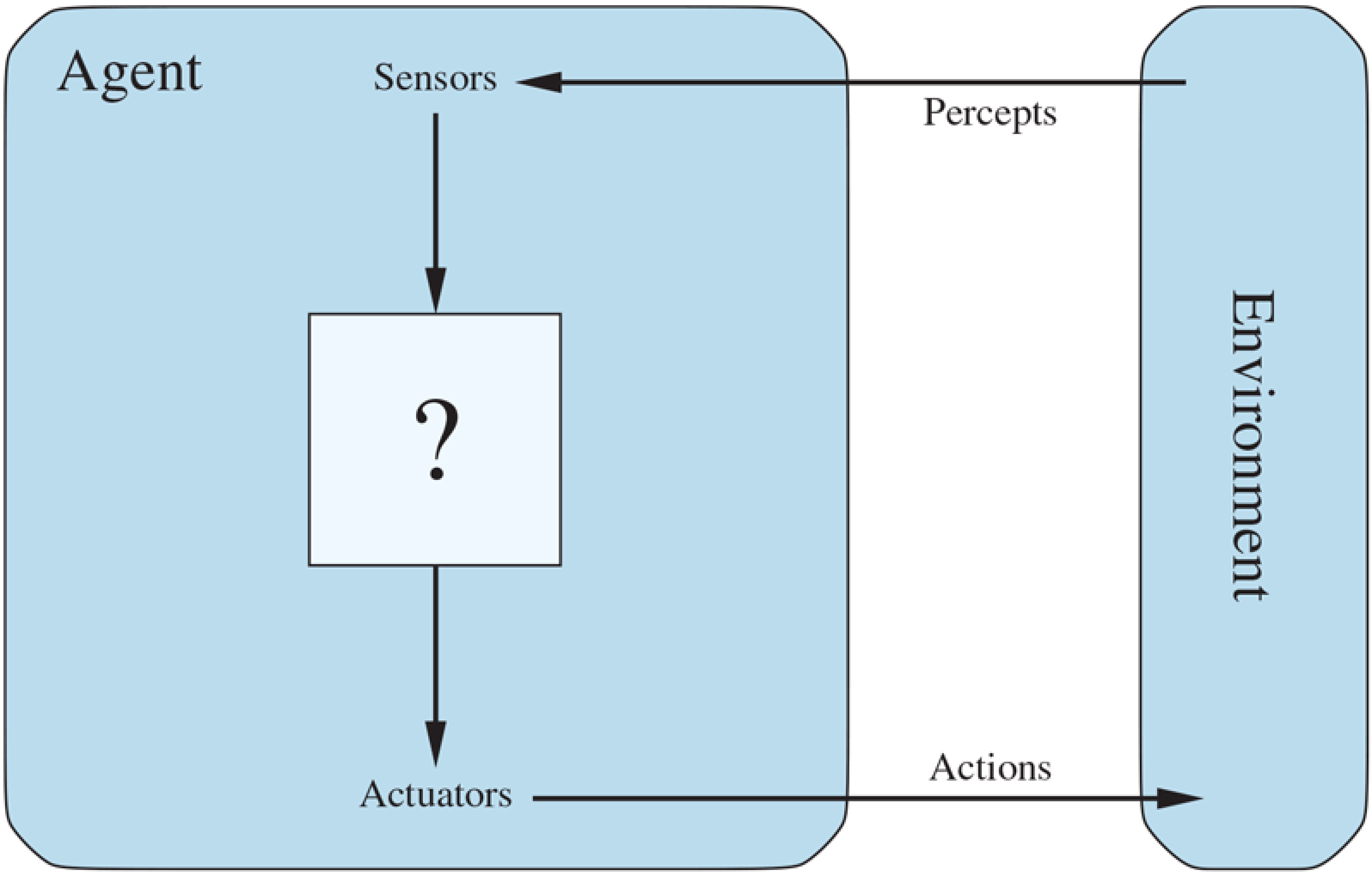
An agent can be anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators

The agent function maps from percept histories to actions:

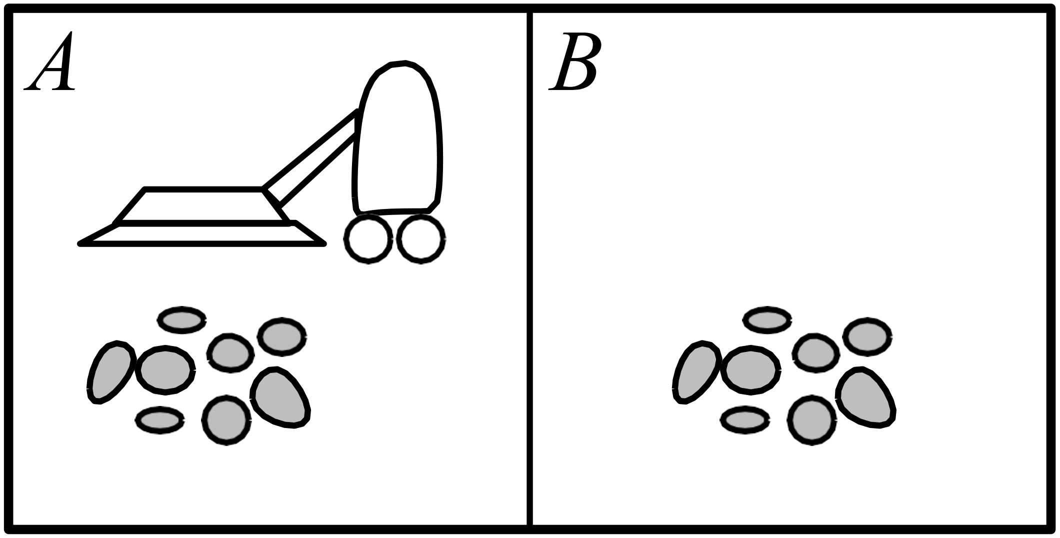
$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

The agent program runs on the physical architecture to produce f

General Agent



Vacuum-cleaner world



Percepts: location and contents, e.g., [A, $Dirty$]

Actions: $Left$, $Right$, $Suck$, $NoOp$

A vacuum-cleaner agent

Percept sequence	Action
[*A*, *Clean*]	*Right*
[*A*, *Dirty*]	*Suck*
[*B*, *Clean*]	*Left*
[*B*, *Dirty*]	*Suck*
.	.

```
function Reflex-Vacuum-Agent( [location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

What is the right function?
Can it be implemented in a small agent program?

Rationality

Fixed performance measure evaluates the environment sequence (metric)

- one point per square cleaned up in time T?
- one point per clean square per time step, minus one per move?
- penalize for $> k$ dirty squares?

A rational agent chooses whichever action maximizes the expected value of the performance measure given the percept sequence to date and built in knowledge.

Rational $\neq$ omniscient

– percepts may not supply all relevant information

Rational $\neq$ clairvoyant

– action outcomes may not be as expected

Hence, rational $\neq$ successful

Rational $\Rightarrow$ exploration, learning, autonomy

PEAS

To design a rational agent, we must specify the task environment

Consider, e.g., the task of designing an automated taxi:

Performance measure??

Environment??

Actuators??

Sensors??

PEAS

To design a rational agent, we must specify the task environment

Consider, e.g., the task of designing an automated taxi:

Performance measure?? safety, destination, profits, legality, comfort, . . .

Environment?? US streets/freeways, traffic, pedestrians, weather, . . .

Actuators?? steering, accelerator, brake, horn, speaker/display, . . .

Sensors?? video, accelerometers, gauges, engine sensors, keyboard, GPS, . . .

Internet shopping agent

Performance measure??

Environment??

Actuators??

Sensors??

Internet shopping agent

Performance measure?? price, quality, appropriateness, efficiency

Environment?? current and future WWW sites, vendors, shippers

Actuators?? display to user, follow URL, fill in form

Sensors?? HTML pages (text, graphics, scripts)

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
Observable??				
Deterministic??				
Episodic??				
Static??				
Discrete??				
Single-agent??				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
Observable??	Yes	Yes	No	No
Deterministic??				
Episodic??				
Static??				
Discrete??				
Single-agent??				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
Observable??	Yes	Yes	No	No
Deterministic??	Yes	No	Partially	No
Episodic??				
Static??				
Discrete??				
Single-agent??				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
Observable??	Yes	Yes	No	No
Deterministic??	Yes	No	Partially	No
Episodic??	No	No	No	No
Static??				
Discrete??				
Single-agent??				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
Observable??	Yes	Yes	No	No
Deterministic??	Yes	No	Partially	No
Episodic??	No	No	No	No
Static??	Yes	Semi D	Semi D	No
Discrete??				
Single-agent??				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
Observable??	Yes	Yes	No	No
Deterministic??	Yes	No	Partially	No
Episodic??	No	No	No	No
Static??	Yes	Semi	Semi	No
Discrete??	Yes	Yes	Yes	No
Single-agent??				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
Observable??	Yes	Yes	No	No
Deterministic??	Yes	No	Partially	No
Episodic??	No	No	No	No
Static??	Yes	Semi	Semi	No
Discrete??	Yes	Yes	Yes	No
Single-agent??	Yes	No	Yes (except auctions)	No

The environment type largely determines the agent design

The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent

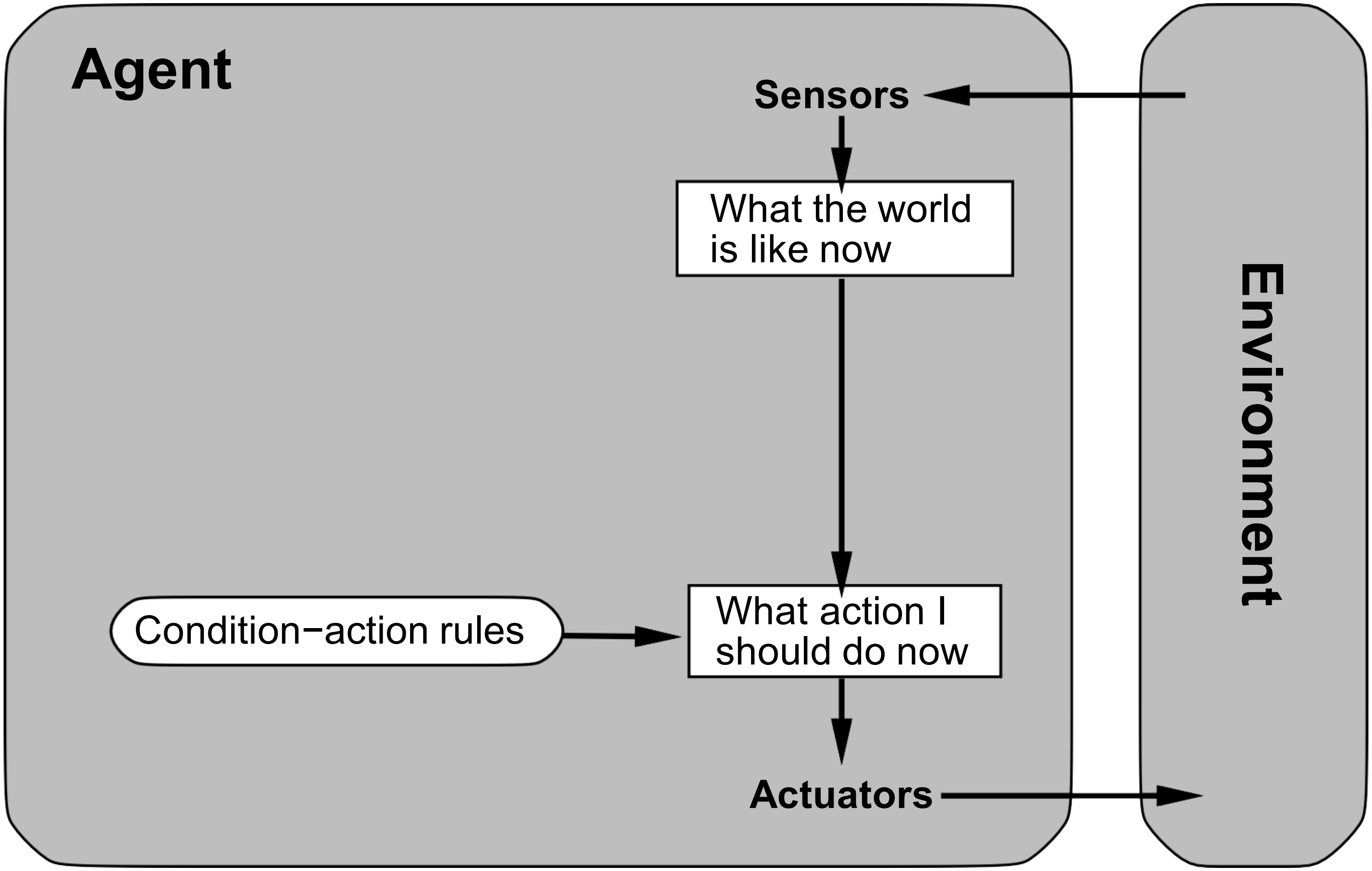
Agent types

Four basic types in order of increasing generality:

- simple reflex agents
- goal-based agents
- utility-based agents
- learning agents

All these can be turned into learning agents

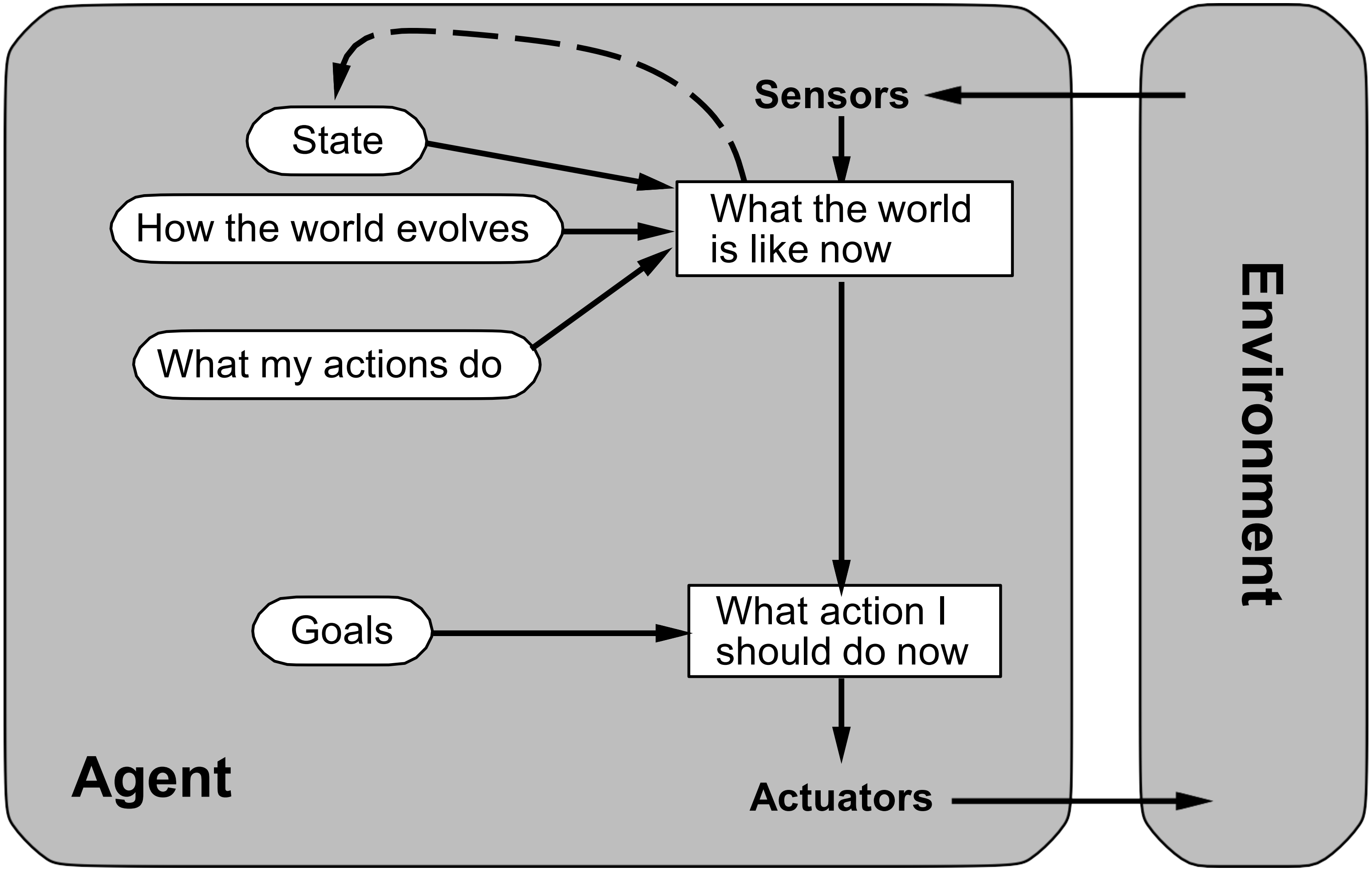
Simple reflex agents



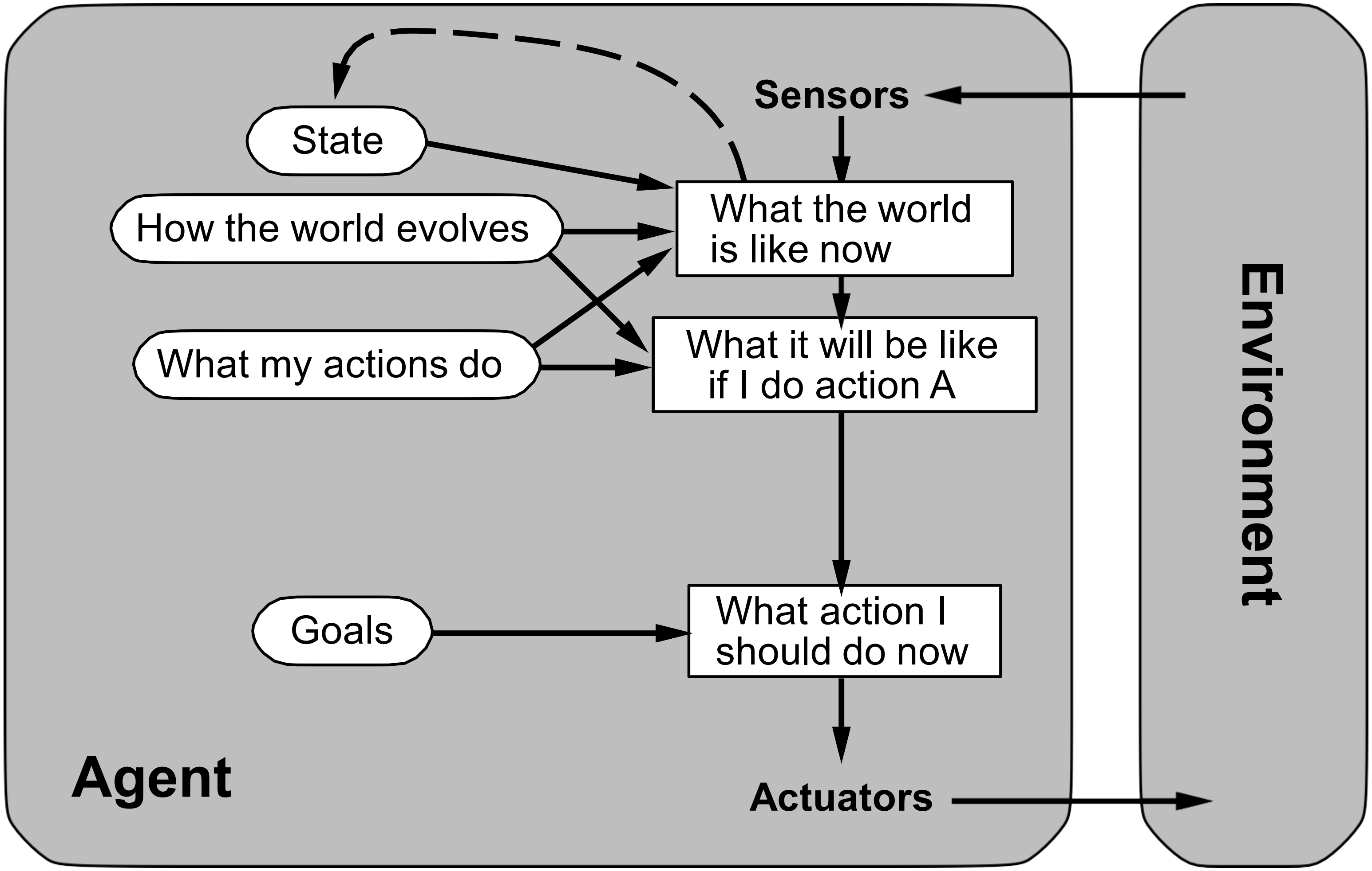
Example

```
function Reflex-Vacuum-Agent( [location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

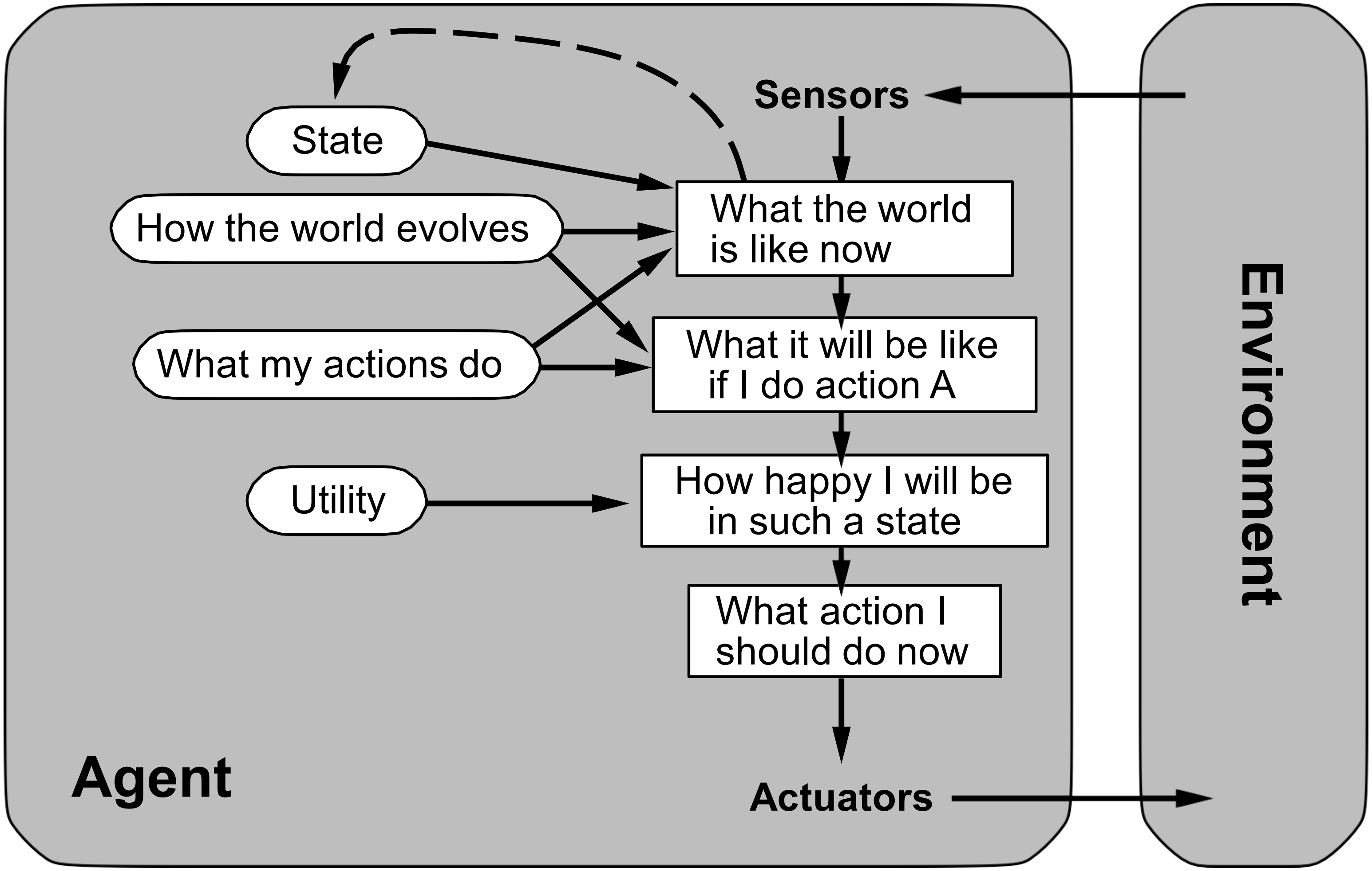
Model-based agents



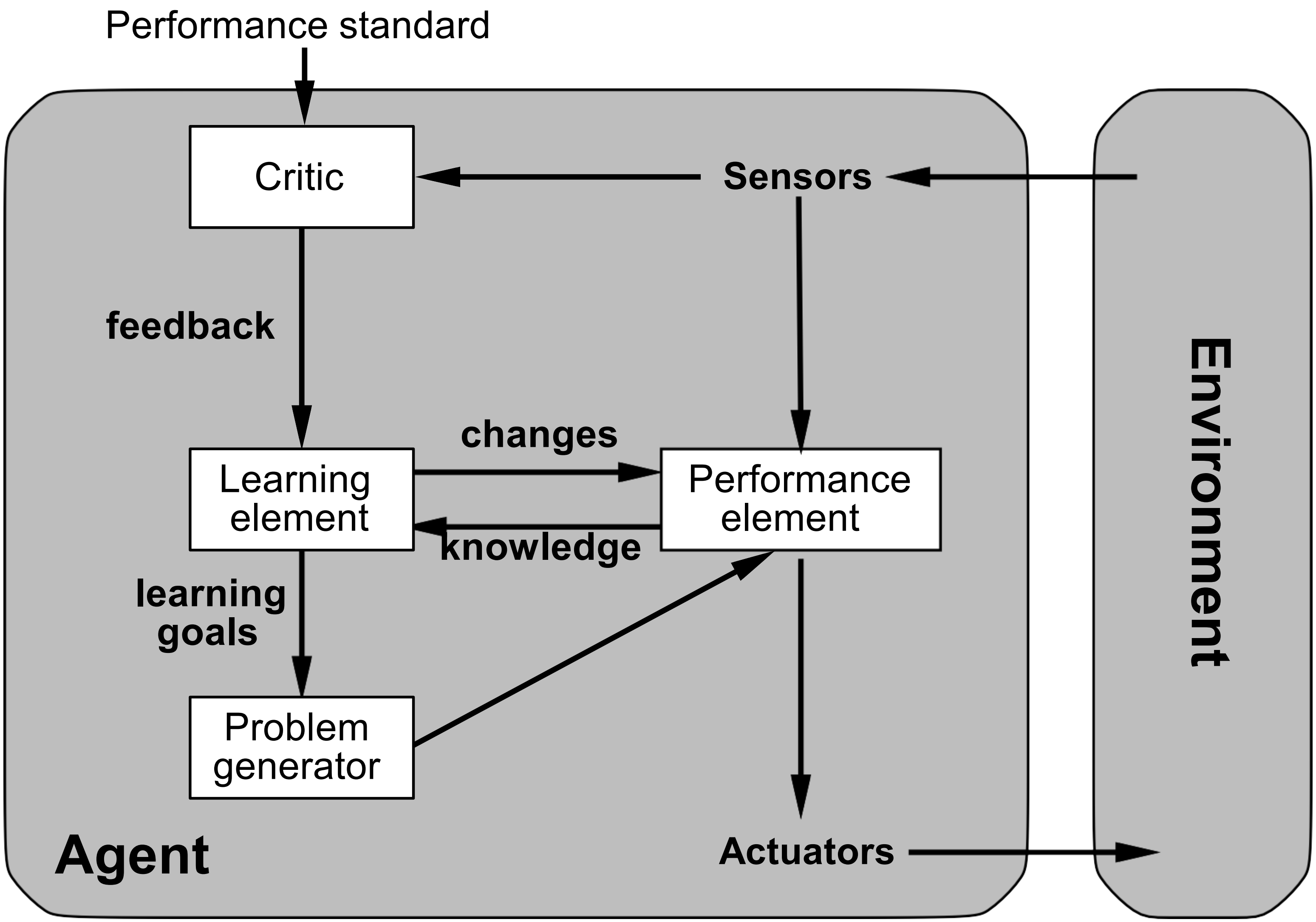
Goal-based agents



Utility-based agents

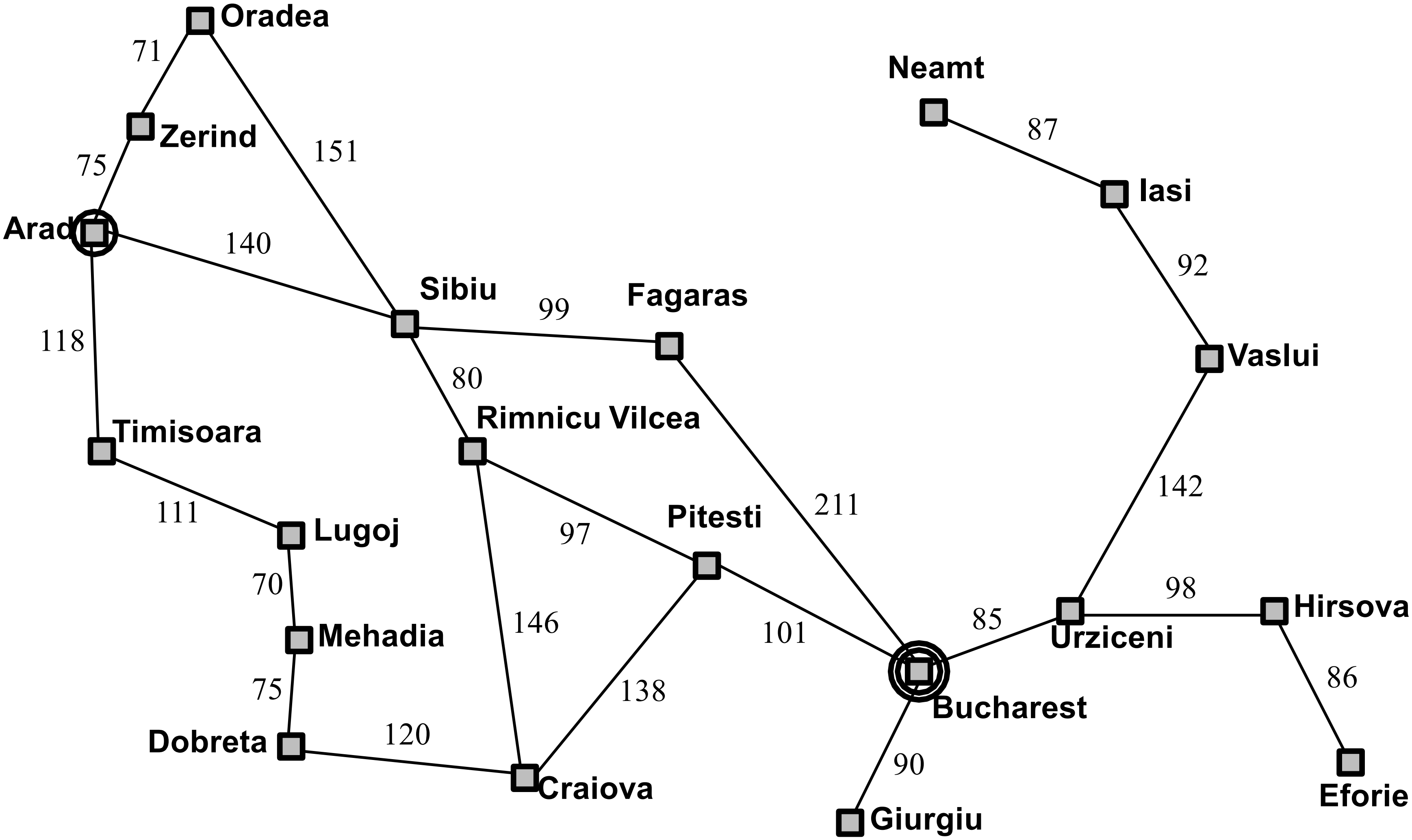


Learning agents



1. Thinking rationally: search and planning
2. Acting rationally: probability
3. Thinking like a human: neural nets
4. Acting like a human: reinforcement learning, games, and vector semantics

Example: Romania



Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:
  be in Bucharest

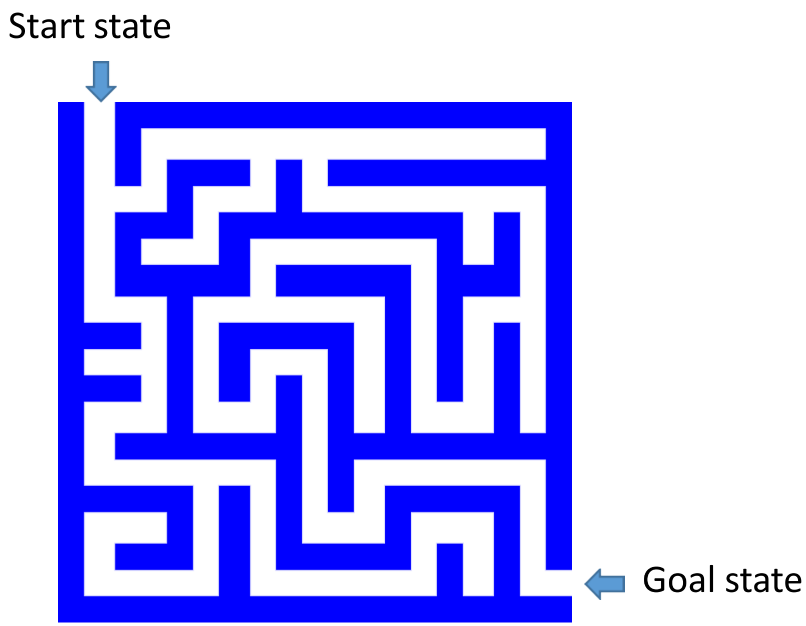
Formulate problem:
  states: various cities
  actions: drive between cities

Find solution:
  sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

1. How to turn ANY problem into a SEARCH problem:
   1. Initial state, goal state, transition model
   2. Actions, path cost
2. General algorithm for solving search problems
   1. First data structure: a frontier list
   2. Second data structure: a search tree
   3. Third data structure: a “visited states” list

Search

- We will consider the problem of designing **goal-based agents** in **fully observable, deterministic, discrete, static, known** environments



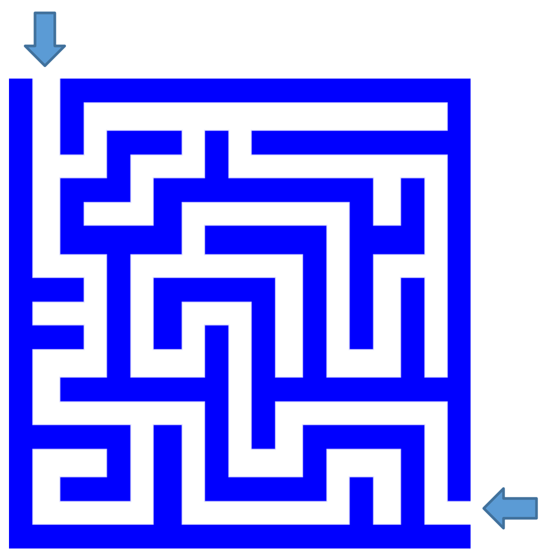
Search

We will consider the problem of designing **goal-based agents** in **fully observable, deterministic, discrete, known** environments

- The agent must find a *sequence of actions* that reaches the goal
- The **performance measure** is defined by (a) reaching the goal and (b) how "expensive" the path to the goal is
- We are focused on the process of finding the solution; while executing the solution, we assume that the agent can safely ignore its percepts (**static environment, open-loop system**)

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

Pearson

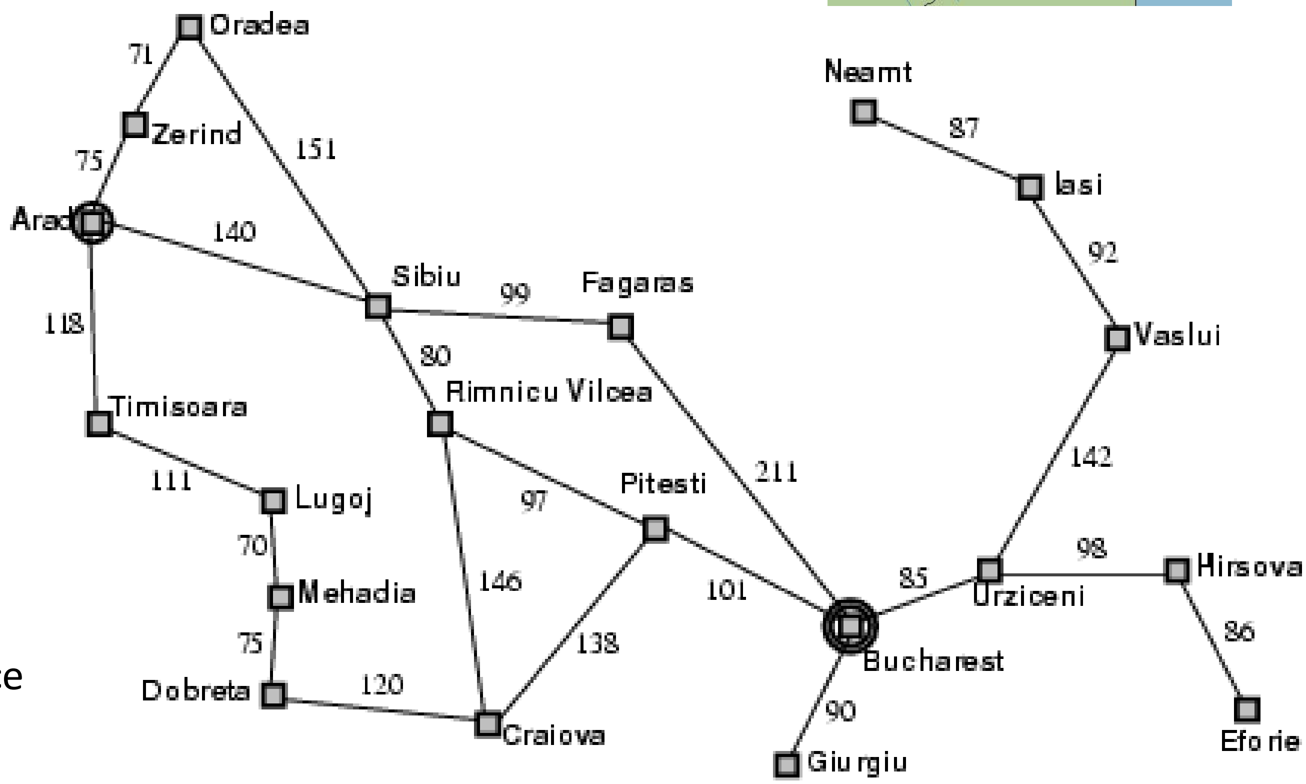
Knowledge Representation: State

- 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: Romania

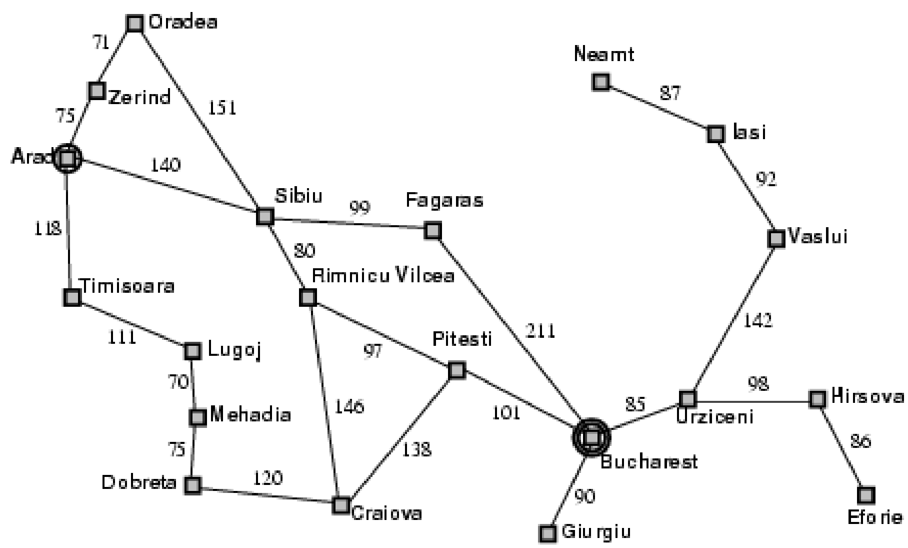
- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

- **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)



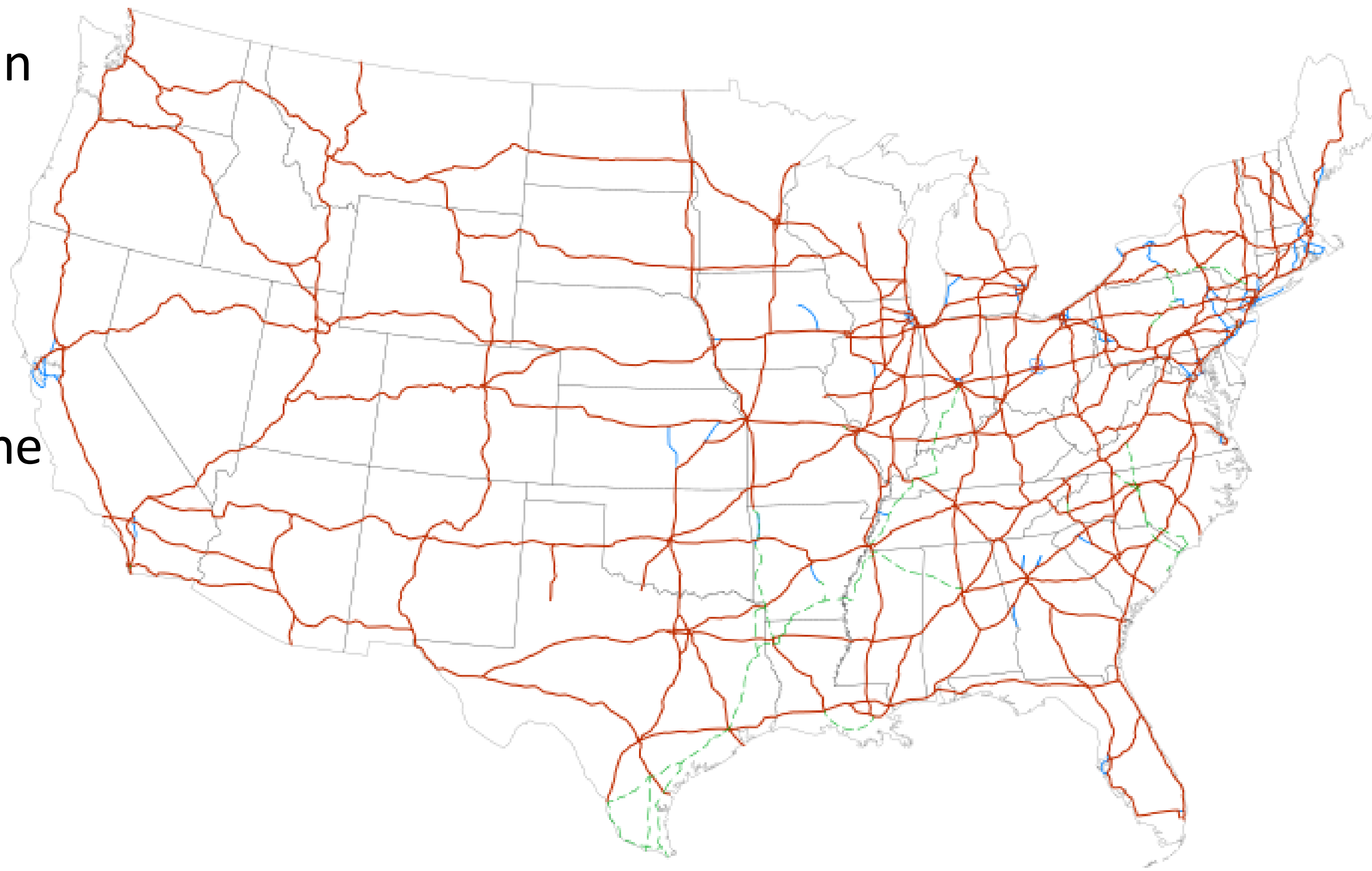
State space

- The initial state, actions, and transition model define the **state space** of the problem
  - The set of all states reachable from initial state by any sequence of actions
  - Can be represented as a **directed graph** where the nodes are states and links between nodes are actions
- What is the state space for the Romania problem?



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.”



Complexity of the State Space

- State Space of Romania problem: size = # cities
  - State space is linear in the size of the world
  - A search algorithm that examines every possible state is reasonable
- State Space of Traveling Salesman problem: size = 2^(#cities)
  - State space is exponential in the size of the world
  - A search algorithm that examines every possible state is unreasonable

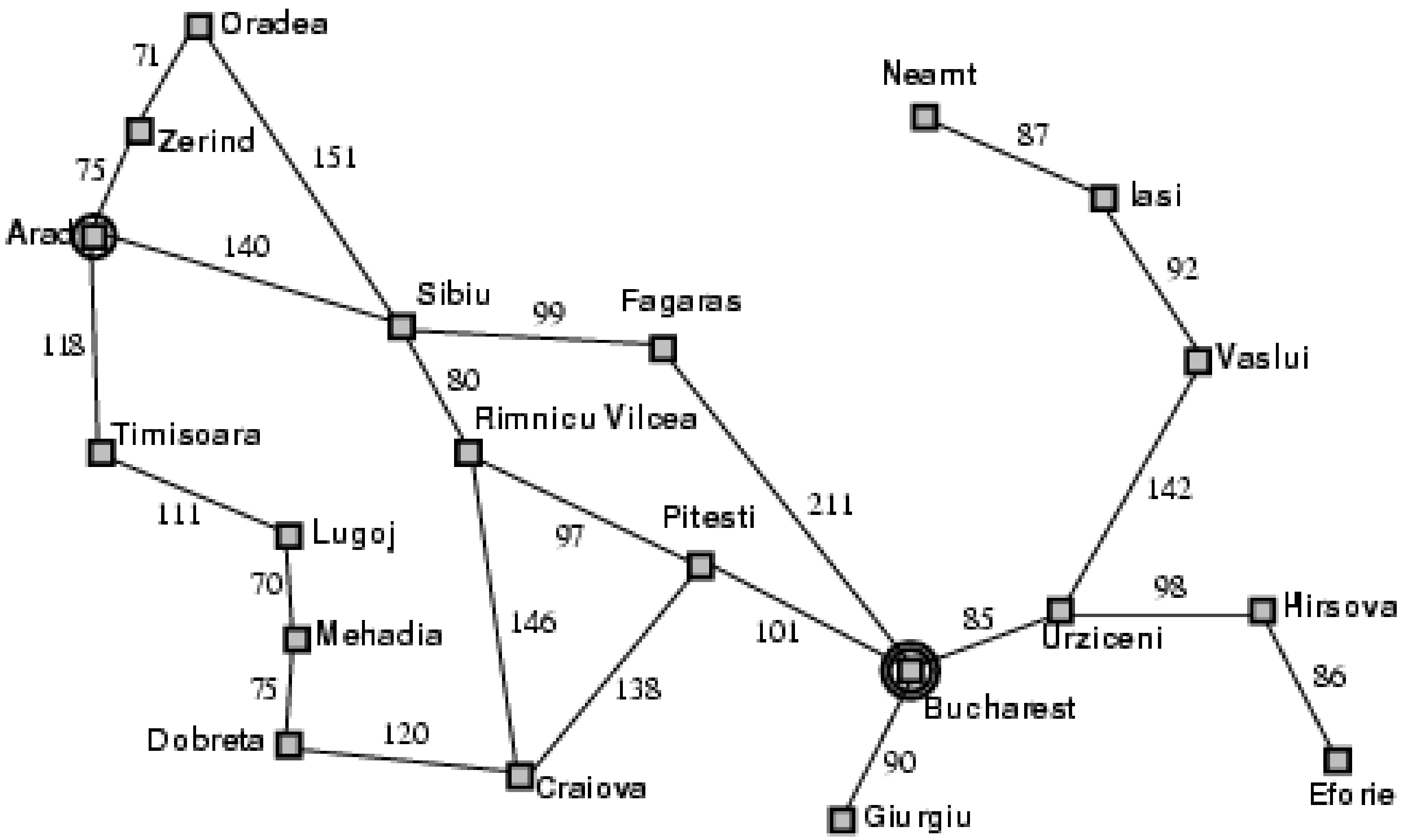
1. How to turn ANY problem into a SEARCH problem:
    1. Initial state, goal state, transition model
    2. Actions, path cost
2. General algorithm for solving search problems
    1. First data structure: frontier (a set)
    2. Second data structure: a search tree (a directed graph)
    3. Third data structure: explored (a dictionary)

First data structure: Frontier Set (Open Set)

- Frontier set = set of states that you know how to reach, but you haven't yet tested to see what comes next after those states
- Initially: FRONTIER = { initial_state }
- First step in the search: figure out which states you can reach from the initial_state, add them to the FRONTIER

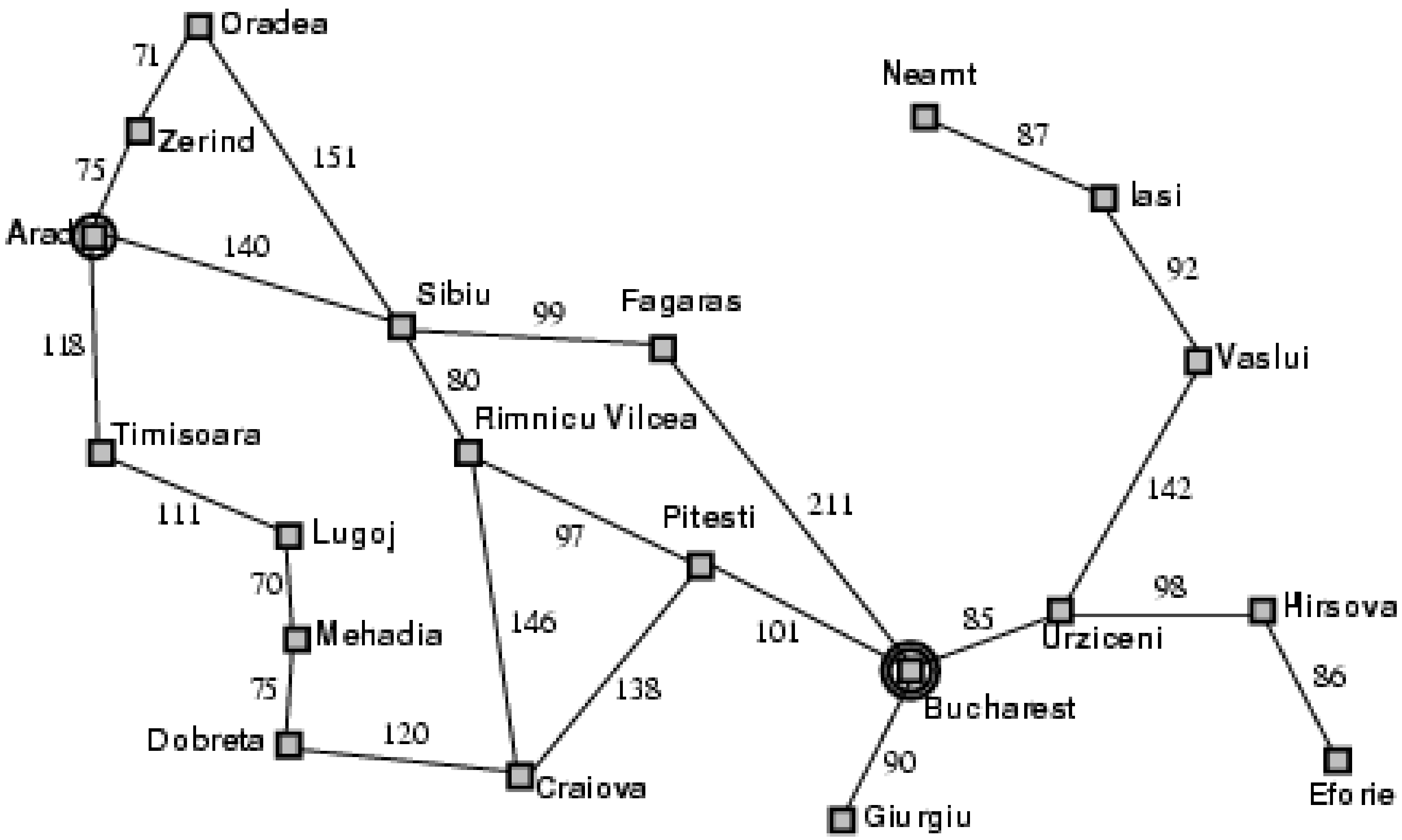
Search step 0

Frontier = { Arad }



Search step 1

Frontier = { Sibiu, Timisoara, Zerind }



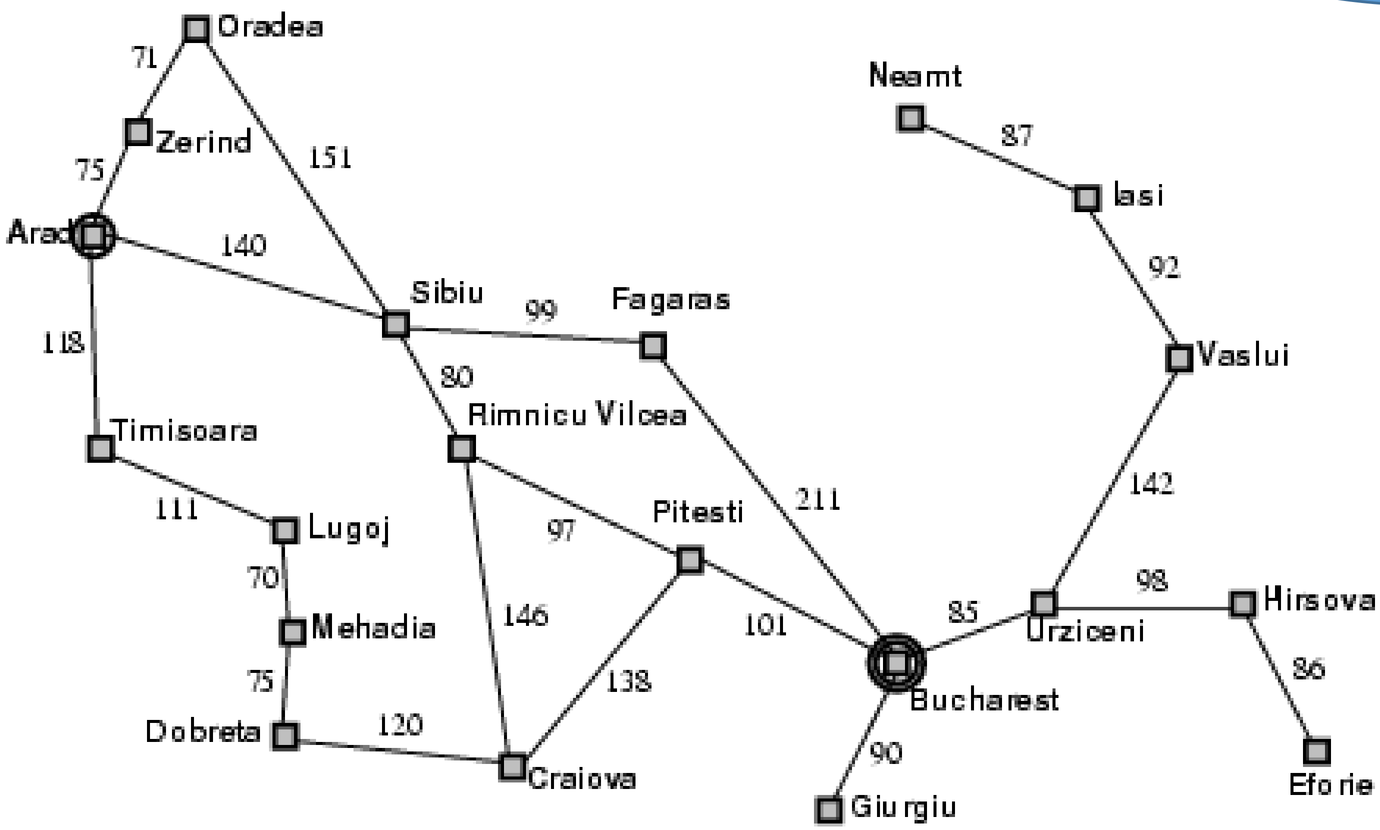
Second data structure: Search Tree

- Tree = directed graph of nodes
- Node = (world_state, parent_node, path_cost)

Search step 0

Frontier: { Arad }

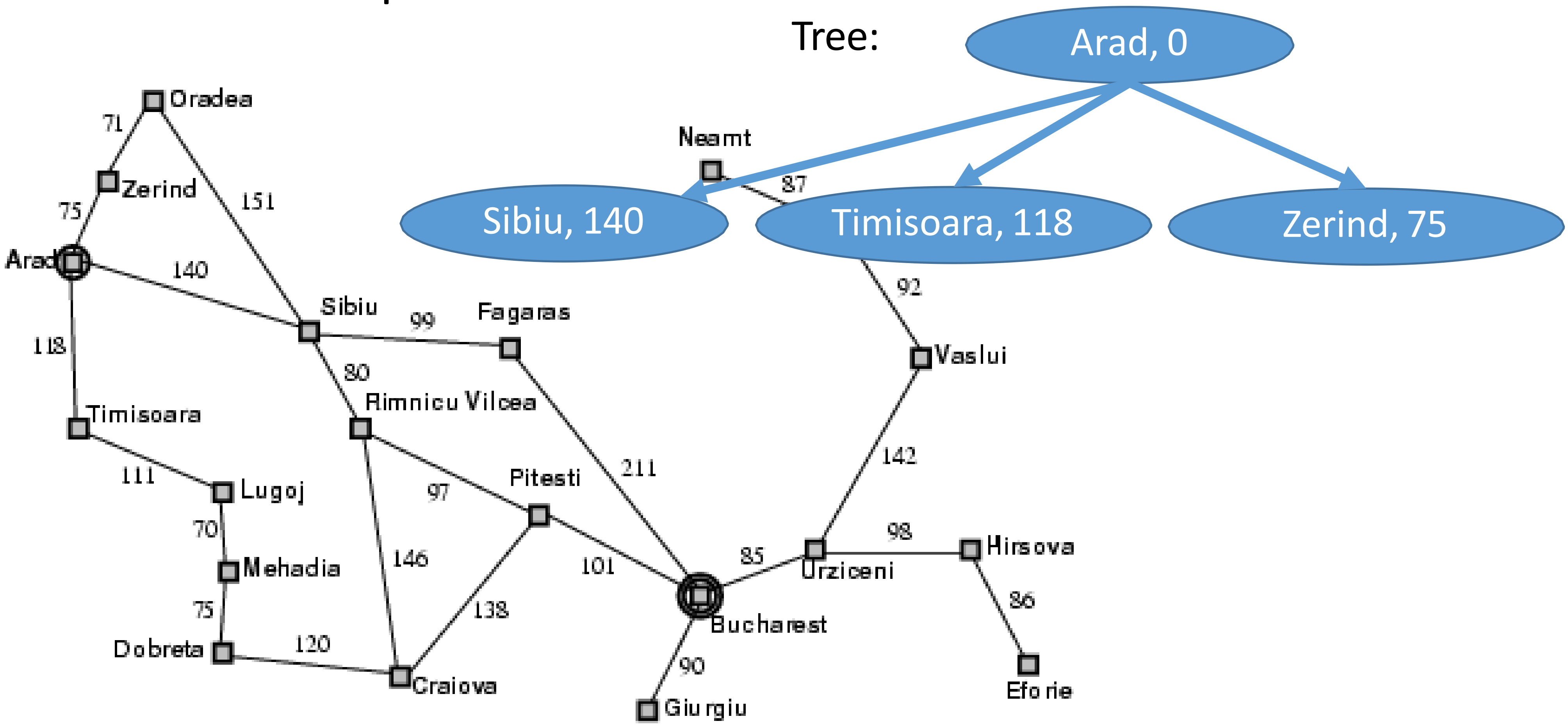
Tree: Arad, 0



Search step 1

Frontier: { Sibiu, Zerind, Timisoara }

Tree:

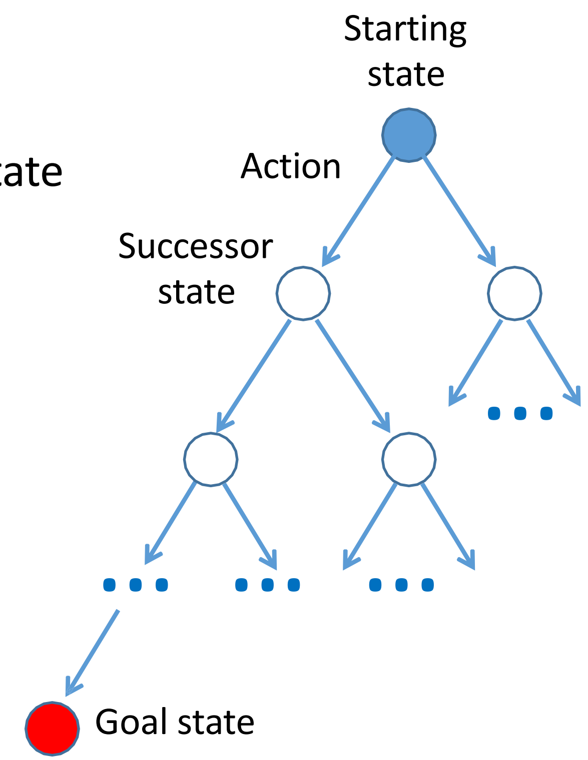


Tree Search: Basic idea

1. SEARCH for an optimal solution
   - Maintain a **frontier** of unexpanded states, and a **tree** showing all known paths
   - 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.

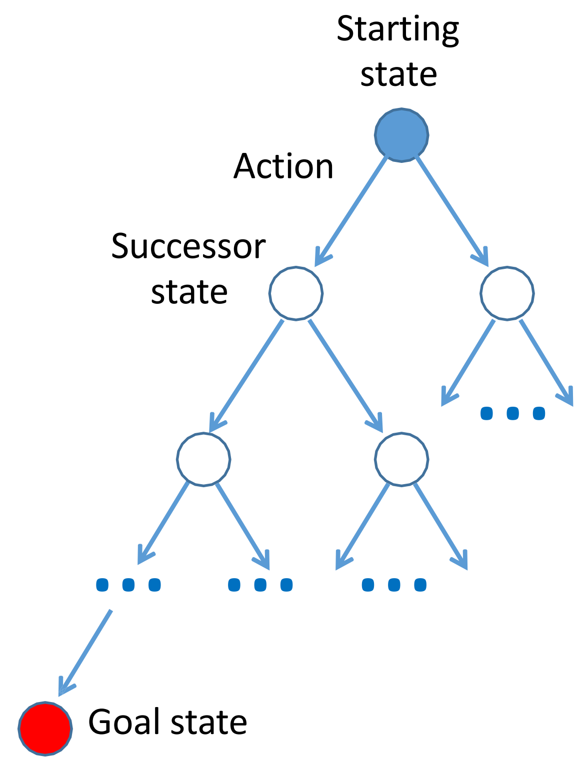
Search Tree

- “What if” tree of sequences of actions and outcomes
- The root node corresponds to the starting state
- The children of a node correspond to the **successor states** of that node’s state
- A path through the tree corresponds to a sequence of actions
  - A solution is a path ending in the goal state
- Nodes vs. states
  - A state is a representation of the world, while a node is a data structure that is part of the search tree
    - Node has to keep pointer to parent, path cost, possibly other info



Nodes vs. States

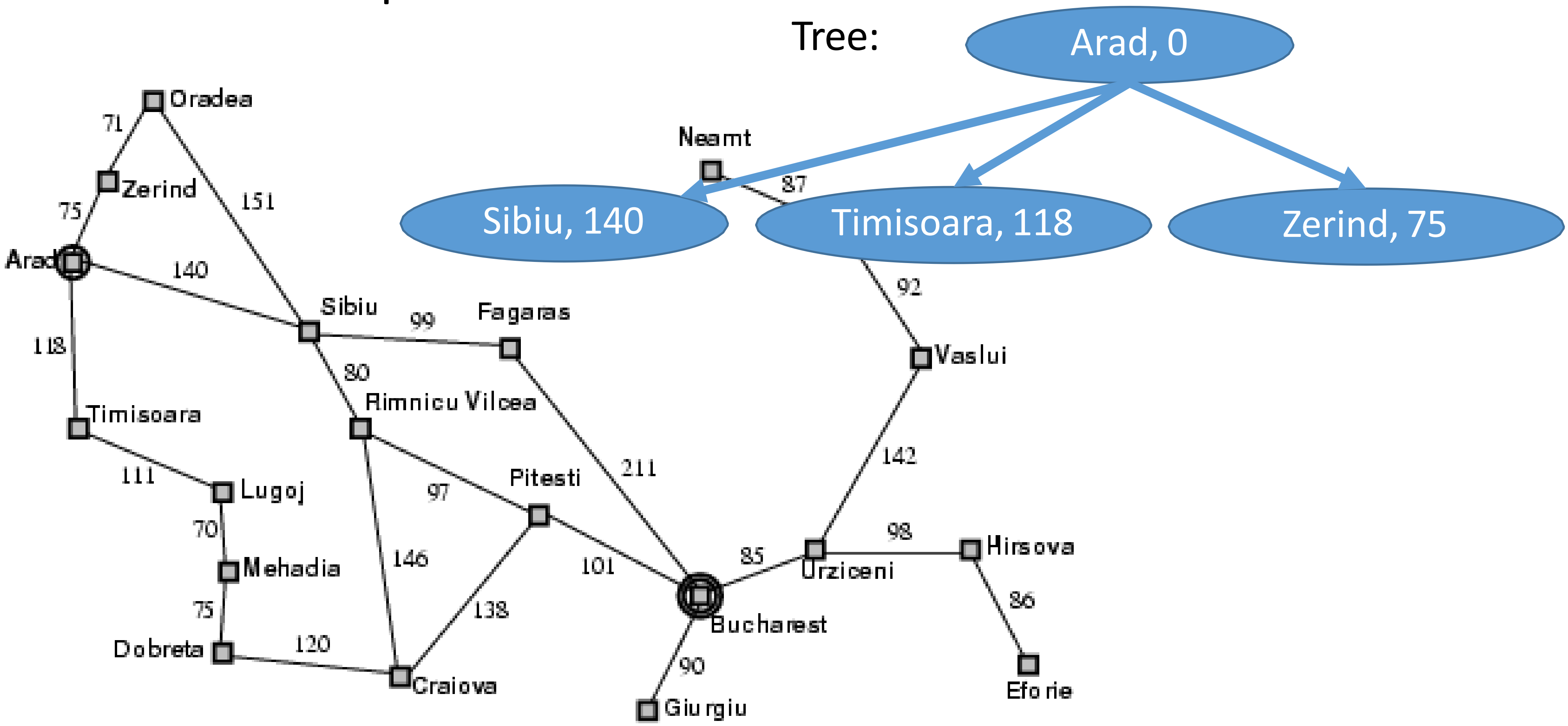
- State = description of the world
  - Must have enough detail to decide whether or not you're currently in the initial state
  - Must have enough detail to decide whether or not you've reached the goal state
  - Often but not always: "defining the state" and "defining the transition model" are the same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

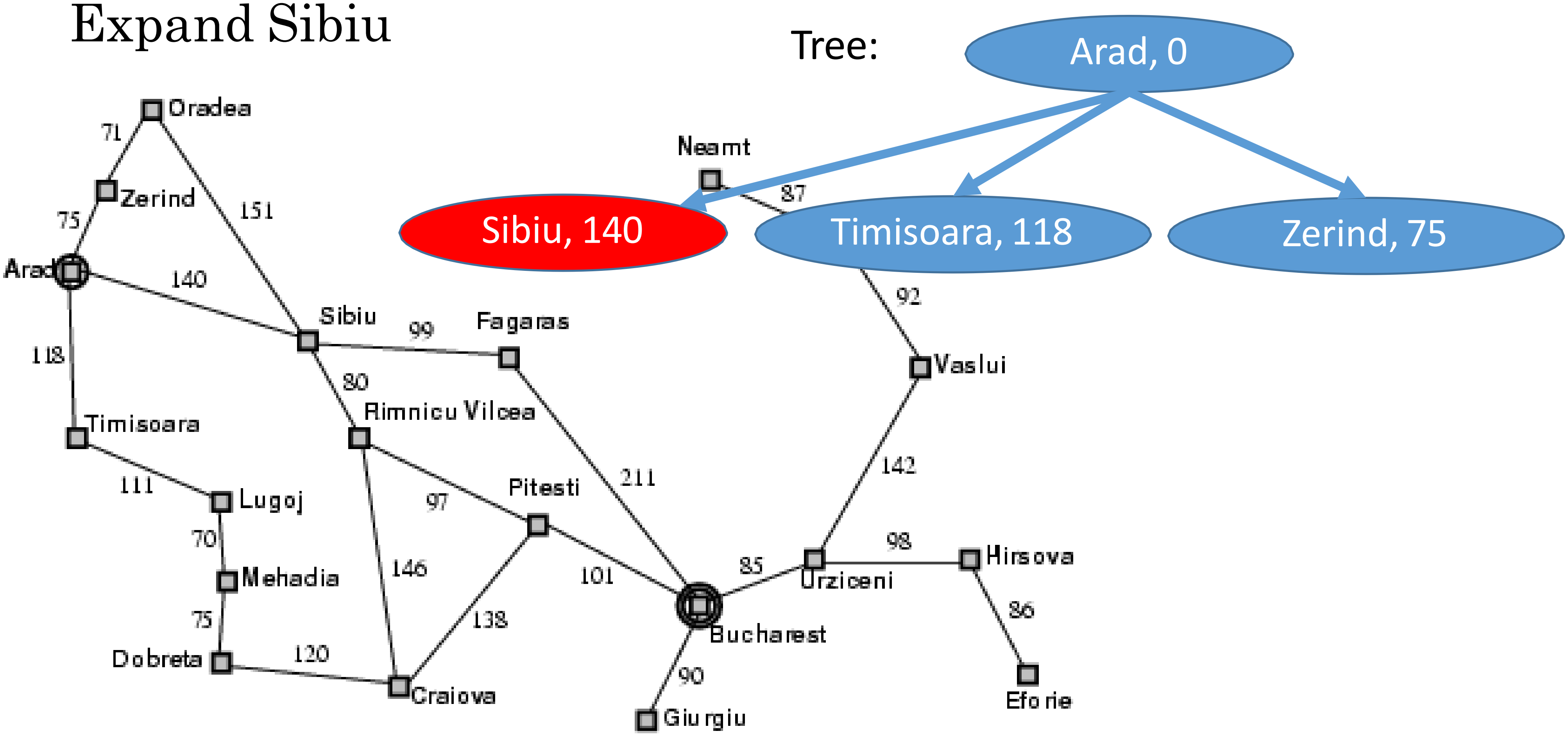


Search step 2

Expand Sibiu

Frontier: { Sibiu, Zerind, Timisoara }

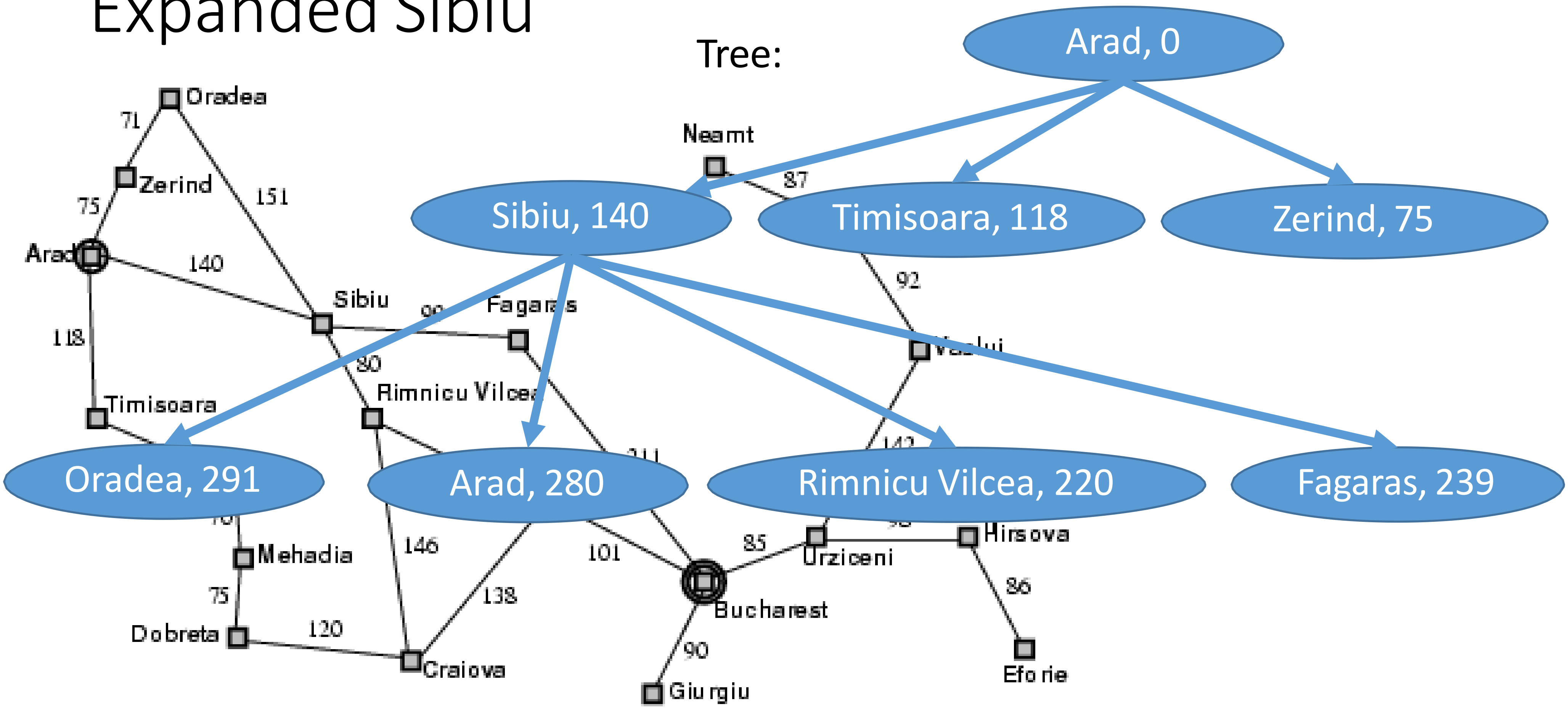
Tree:



Search step 2
Expanded Sibiu

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

Tree:



Tree Search: Computational Complexity

Without an EXPLORED set (Closed 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

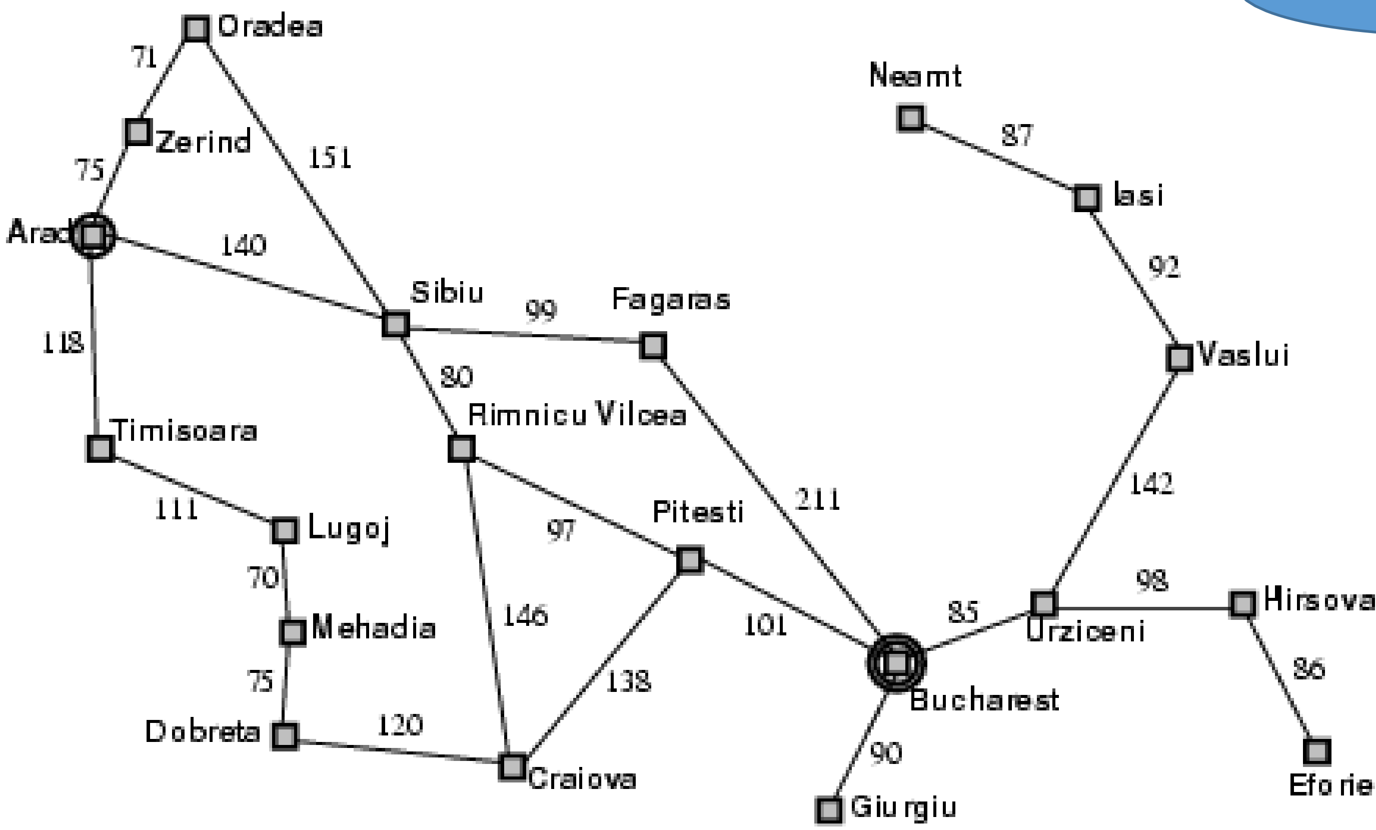
- When you expand a state, you get the list of its possible child states
- ONLY IF a child state is not already explored, put it on the frontier, and put it on the explored set.
- Result: complexity = min(O{b^d}, O{# possible world states})

Search step 0

Frontier: { Arad }

Explored: { Arad }

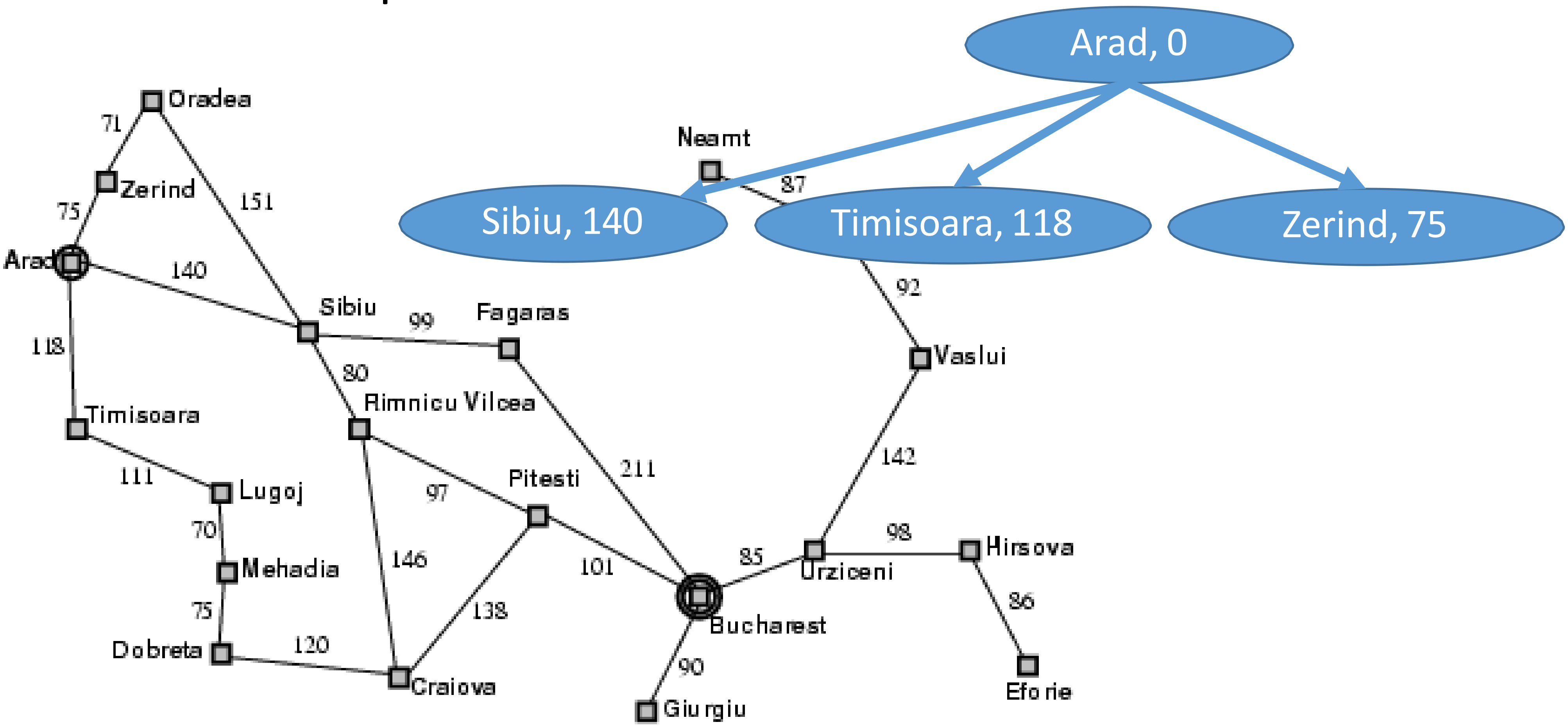
Tree:



Frontier: { Sibiu, Zerind, Timisoara }

Explored: { Arad, Sibiu, Zerind, Timisoara }

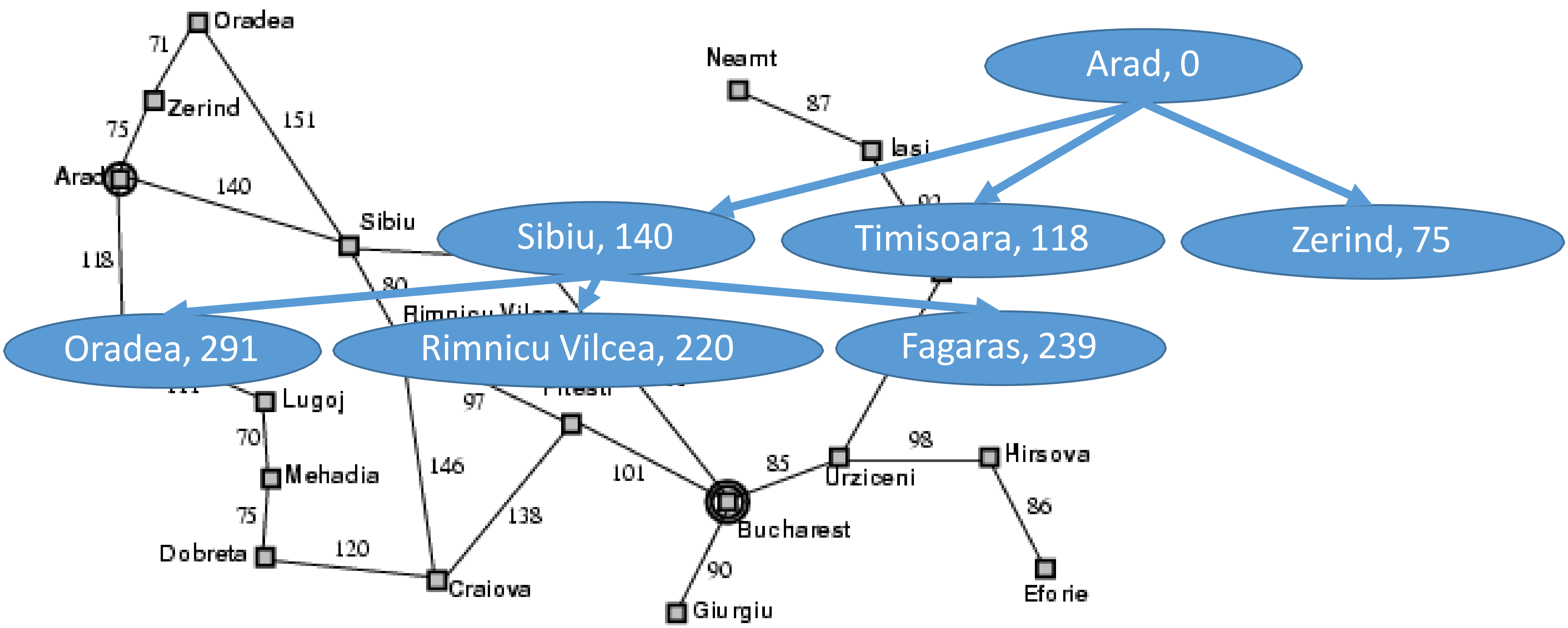
Search step 1



Search step 2

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

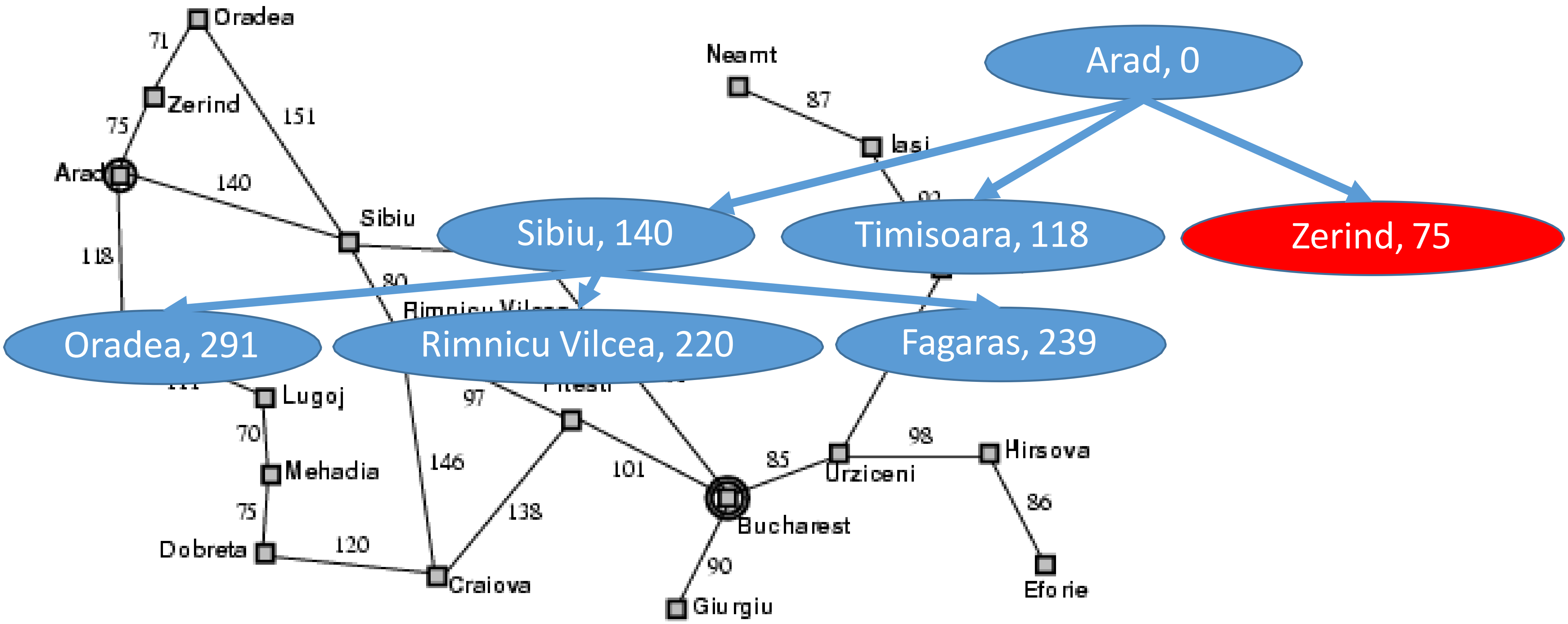
Explored: { Arad, Sibiu, Zerind, Timisoara, Oradea, Rimnicu Vilcea, Fagaras }



Search step 3: expand Zerind

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

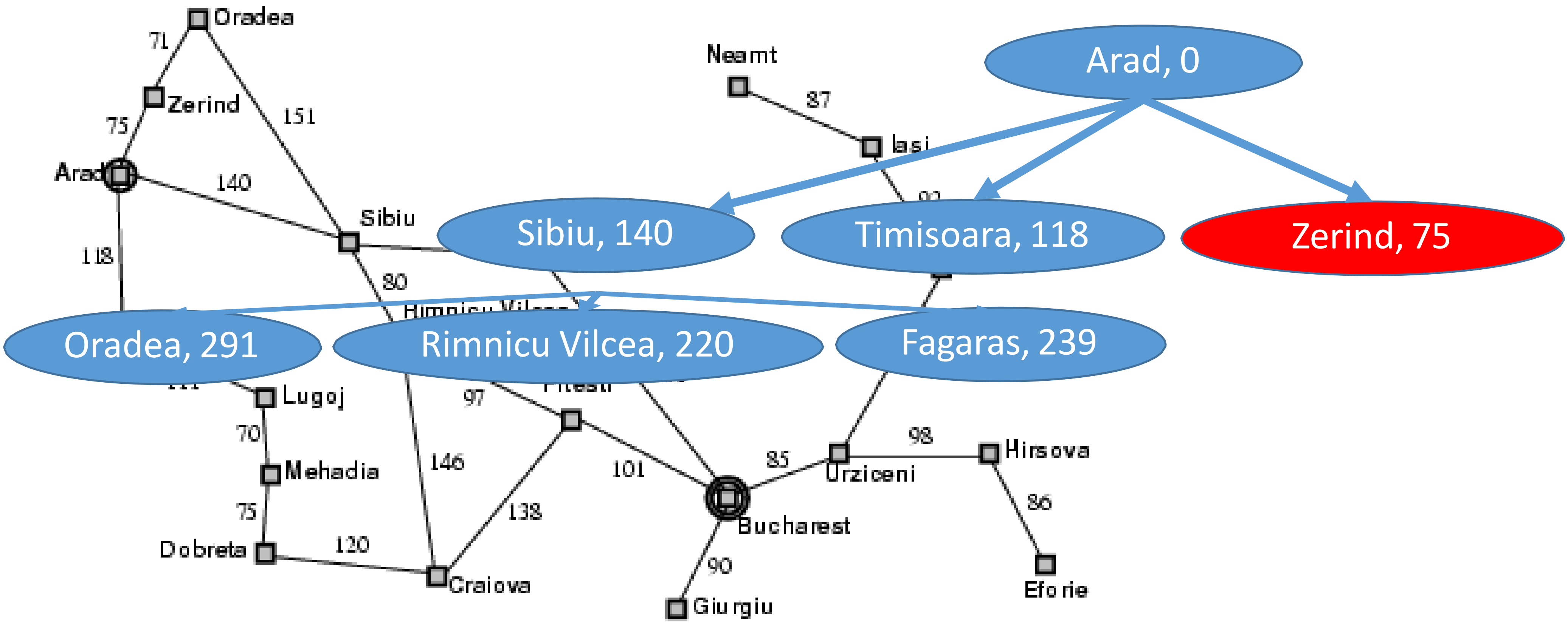
Explored: { Arad, Sibiu, Zerind, Timisoara, Oradea, Rimnicu Vilcea, Fagaras }



Search step 3:
we can reach Oradea with a total path cost of only 75+71=146

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

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



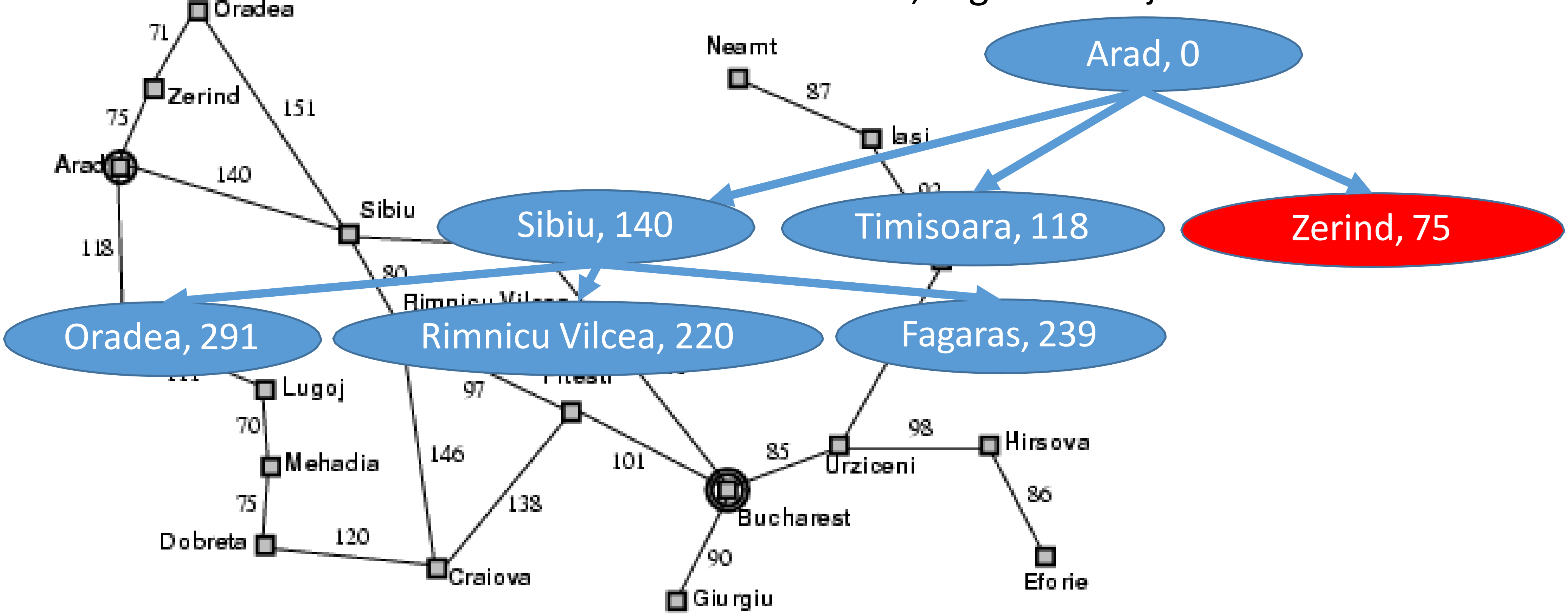
Third data structure: Explored Dictionary

- 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

Search step 3:
we can reach Oradea with a total path cost of only 75+71=146

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

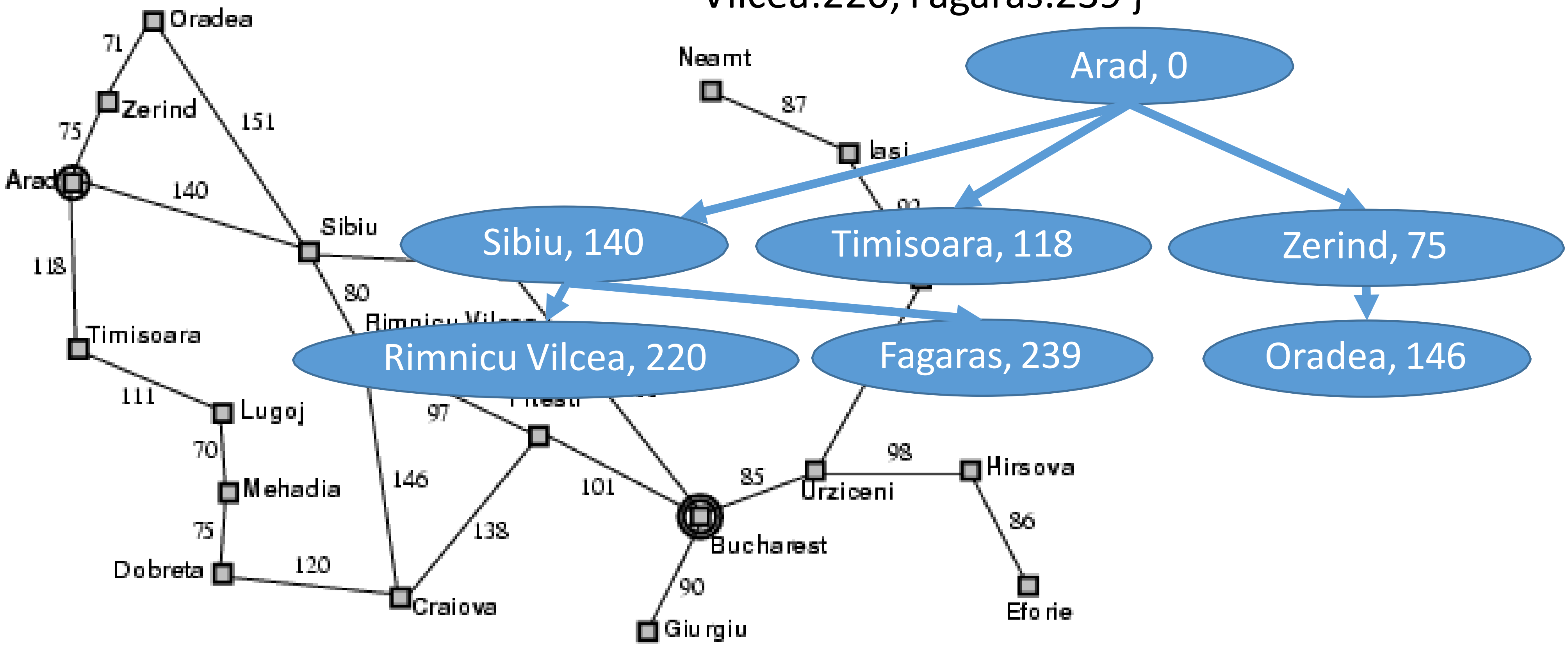
Explored: { Arad:0, Sibiu:140, Zerind:75, Timisoara:118, Oradea:291, Rimnicu Vilcea:220, Fagaras:239 }



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 }



Tree Search: Basic idea

At each step, pick a state from the frontier to **expand:**

1. Check to see whether or not this state is the goal state. If so, DONE! If not, then for each child:
2. Check to see whether this child is already in the explored set with a LOWER COST. If so, ignore it. If not:
3. Add it to the frontier, to the tree, and to the explored dict.

Complexity = min(O{b^d}, O{# possible world states}).

Next time: how can we limit d?

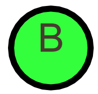
Basics of Forward Search

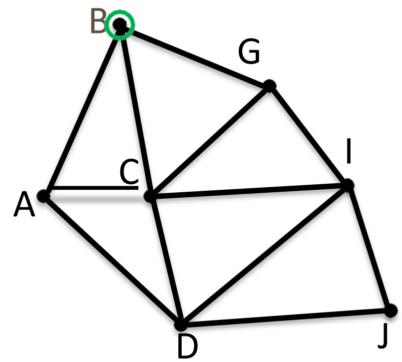
- Generally, start from the start, grow tree until you find a solution (path to goal)
- Expanding a node refers to adding children to the tree, pushing them onto the open set
- Try to expand as few tree nodes as possible

- **Open** set maintains a list of frontier (unexpanded) plans
  - Keeps track of what nodes to expand next
  - Often stored as a priority queue
  - For each node in the open list, we know of at least one path to it from the start

- **Closed** set keeps track of nodes that have been expanded
  - For each node in the closed list, we've already found the lowest-cost path to it from the start

Breadth-First Search Example





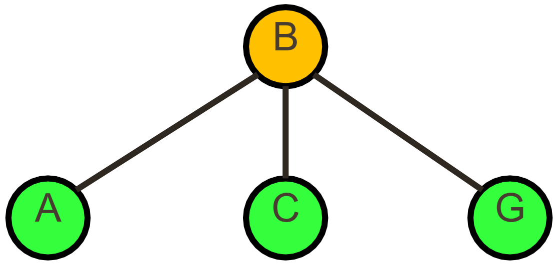

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

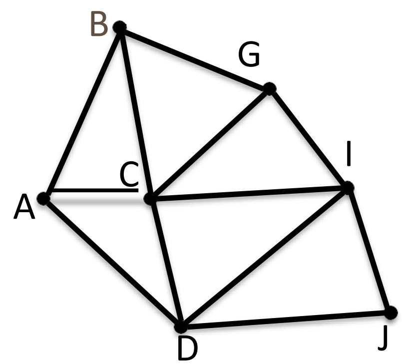
Open (Q):
{B}

Visited:
{}

Our (BFS) queue will be FIFO:
- push (*Q.Insert*) onto the end
- pop (*Q.GetFirst*) from the front

Breadth-First Search Example



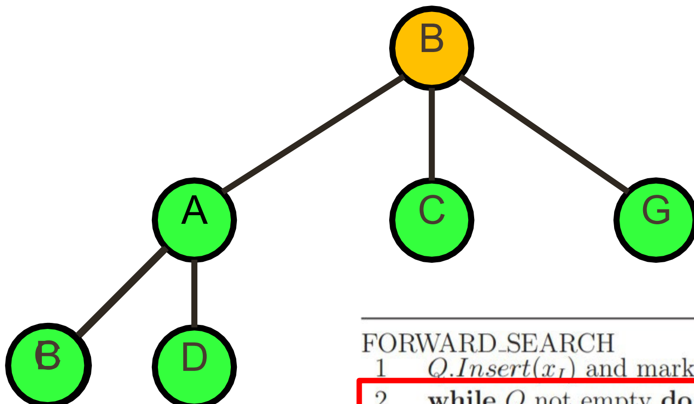



```
FORWARD_SEARCH
  Q.Insert(x_I) and mark x_I as visited
  while Q not empty do
      x ← Q.GetFirst()
      if x ∈ X_G
          return SUCCESS
      forall u ∈ U(x)
          x' ← f(x, u)
          if x' not visited
              Mark x' as visited
              Q.Insert(x')
          else
              Resolve duplicate x'
  return FAILURE
```

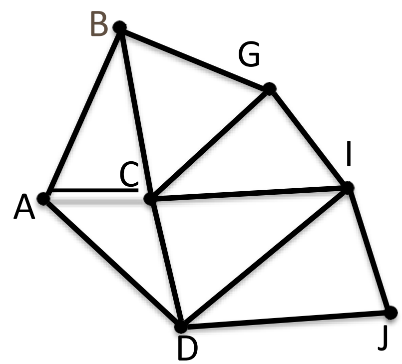
Open (Q):
{A,C,G}

Visited:
{B,A,C,G}

Breadth-First Search Example



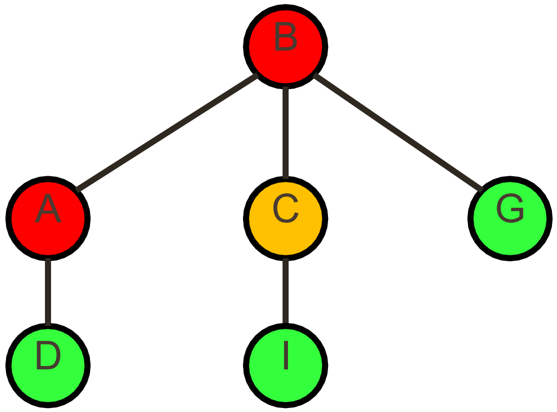

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

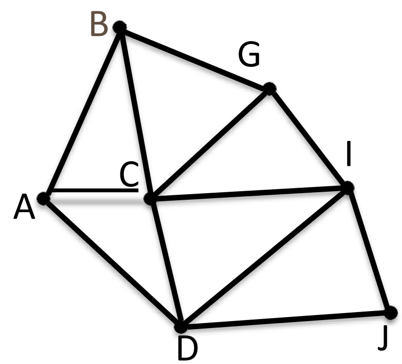


Open (Q): {C,G,D}

Visited: {B,A,C,G,D}

Breadth-First Search Example

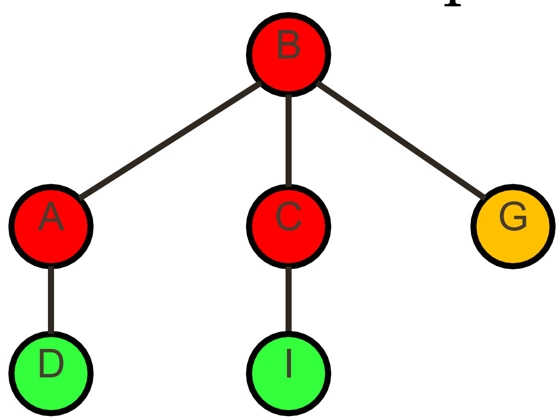


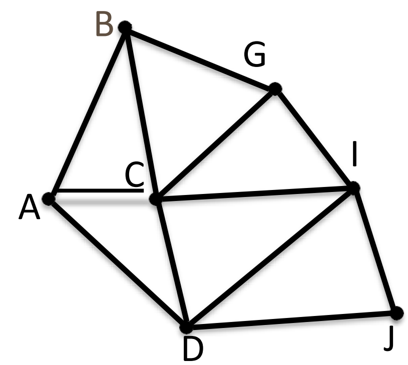


Open (Q):
{G,D,I}

Visited:
{B,A,C,G,D,I}

Breadth-First Search Example

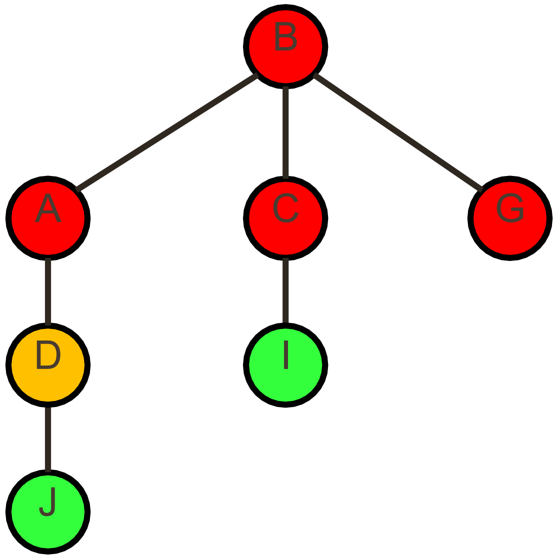


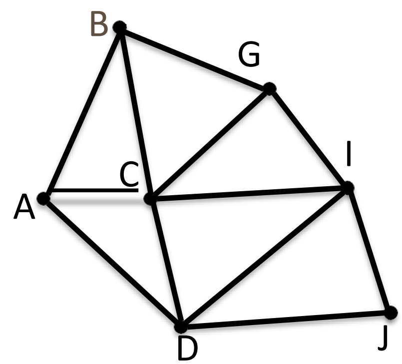


Open (Q):
{D,I}

Visited:
{B,A,C,G,D,I}

Breadth-First Search Example

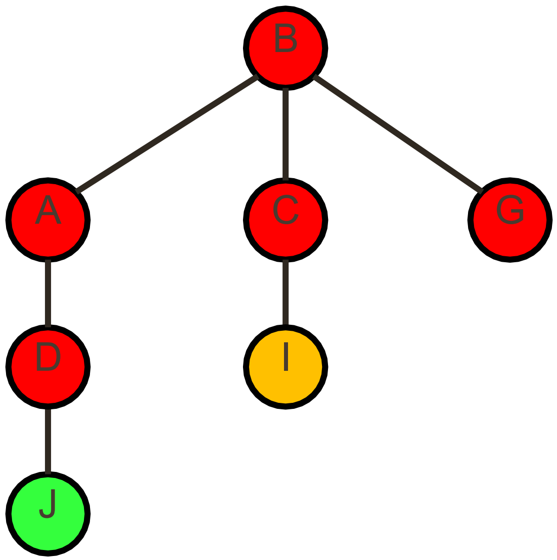


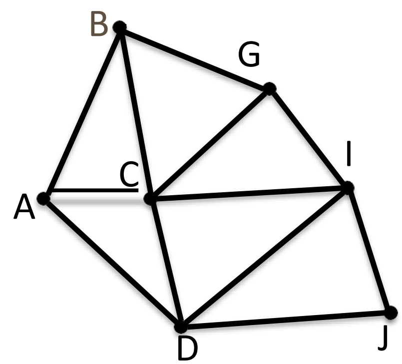


Open (Q):
{I,J}

Visited:
{B,A,C,G,D,I,J}

Breadth-First Search Example

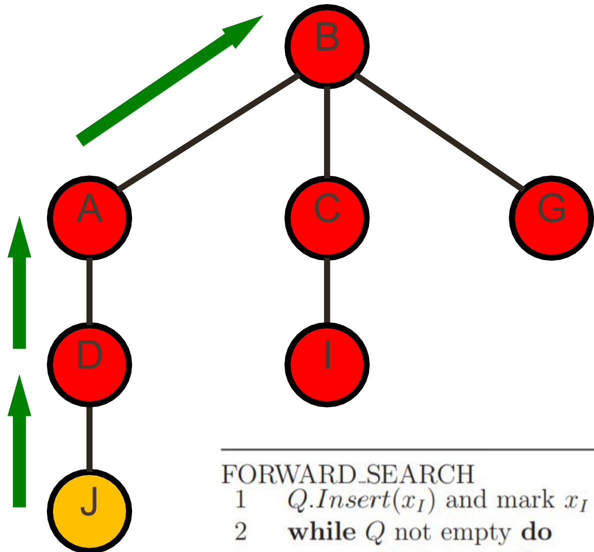


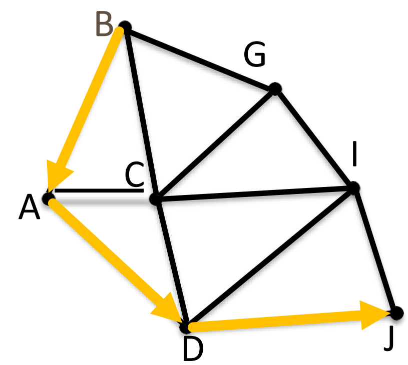


Open (Q):
{J}

Visited:
{B,A,C,G,D,I,J}

Breadth-First Search Example





Open (Q): {}

Visited: {B,A,C,G,D,I,J}

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

Final path solution: B→A→D→J

Other solutions may exist but have the same number or more transitions

Breadth-First Search

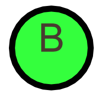
- Complete (will find the solution if it exists)
- Guaranteed to find the shortest (number of edges) path
  - First solution found is the optimal path
- What about non-uniform edge weights? (… Dijkstra)
- Time complexity O(|V|+|E|)
- Consider another approach: Depth-first search

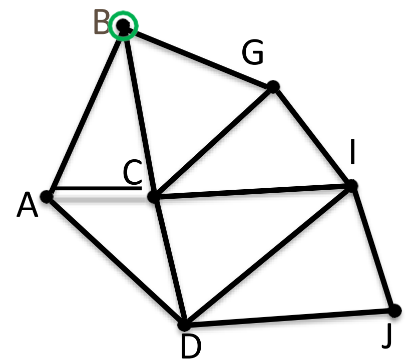
Depth-first search

- Instead of searching across levels of the tree, DFS starts at the root node and explores as far as possible along each branch before backtracking

- Similar implementation to BFS, but with a stack (last-in first-out) queue

Depth-First Search Example





FORWARD_SEARCH

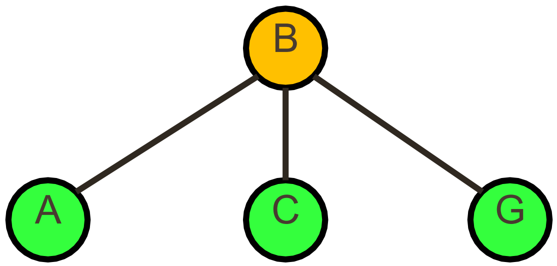
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
    $x \leftarrow Q.GetFirst()$
    if $x \in X_G$
        return SUCCESS
    forall $u \in U(x)$
        $x' \leftarrow f(x, u)$
        if x' not visited
            Mark x' as visited
            $Q.Insert(x')$
        else
            Resolve duplicate x'
return FAILURE

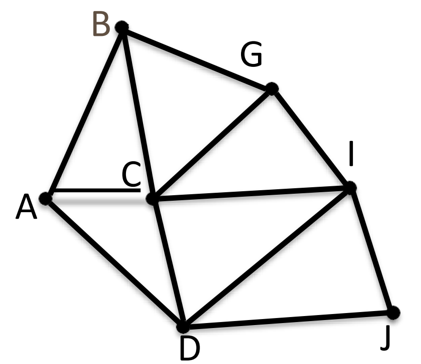
Open (Q): {B}

Visited: {}

Our (DFS) queue will be LIFO:
- push ($Q.Insert$) onto the front
- pop ($Q.GetFirst$) from the front

Depth-First Search Example





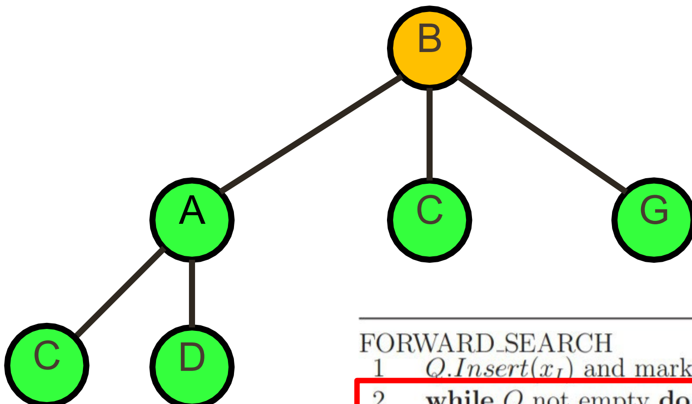
FORWARD_SEARCH

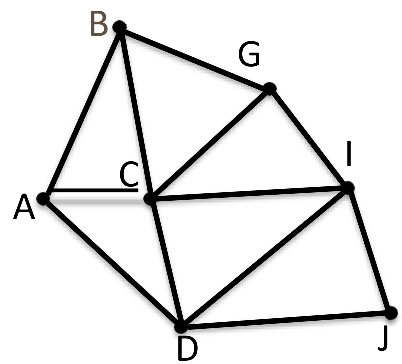
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
  $x \leftarrow Q.GetFirst()$
  if $x \in X_G$
    return SUCCESS
  forall $u \in U(x)$
    $x' \leftarrow f(x, u)$
    if x' not visited
      Mark x' as visited
      $Q.Insert(x')$
    else
      Resolve duplicate x'
return FAILURE

Open (Q):
{A,C,G}

Visited:
{B,A,C,G}

Depth-First Search Example



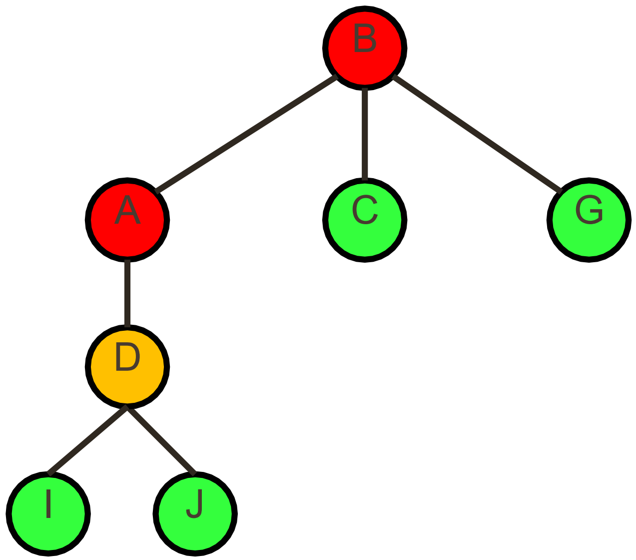


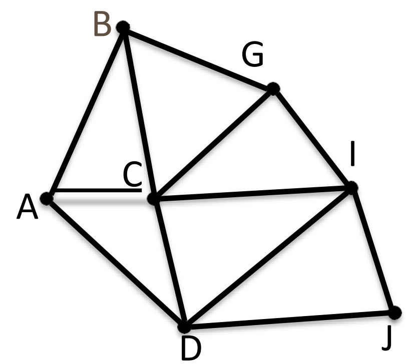

```
FORWARD_SEARCH
    Q.Insert(x_I) and mark x_I as visited
    while Q not empty do
        x ← Q.GetFirst()
        if x ∈ X_G
            return SUCCESS
        forall u ∈ U(x)
            x' ← f(x, u)
            if x' not visited
                Mark x' as visited
                Q.Insert(x')
            else
                Resolve duplicate x'
    return FAILURE
```

Open (Q):
{D,C,G}

Visited:
{B,A,C,G,D}

Depth-First Search Example

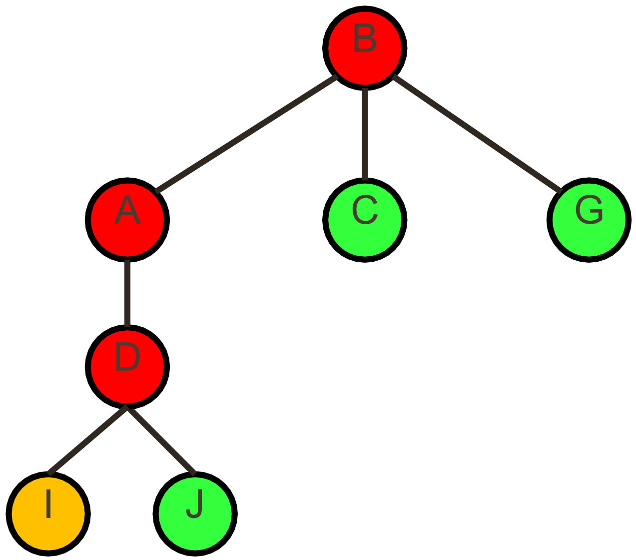


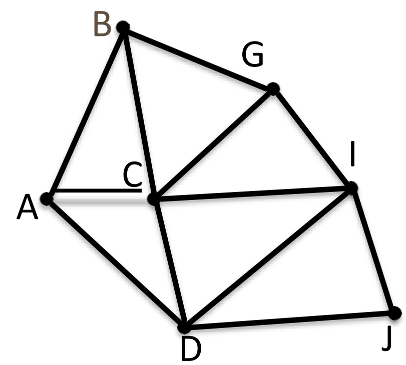


Open (Q):
{I,J,C,G}

Visited:
{B,A,C,G,D,I,J}

Depth-First Search Example

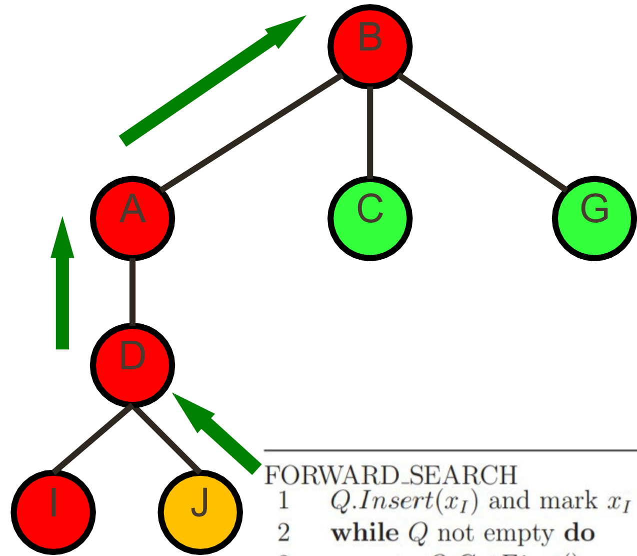




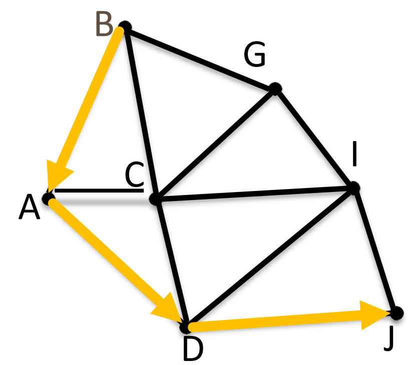
Open (Q):
{J,C,G}

Visited:
{B,A,C,G,D,I,J}

Depth-First Search Example




```
FORWARD_SEARCH
    Q.Insert(x_I) and mark x_I as visited
    while Q not empty do
        x ← Q.GetFirst()
        if x ∈ X_G
            return SUCCESS
        forall u ∈ U(x)
            x' ← f(x, u)
            if x' not visited
                Mark x' as visited
                Q.Insert(x')
            else
                Resolve duplicate x'
    return FAILURE
```

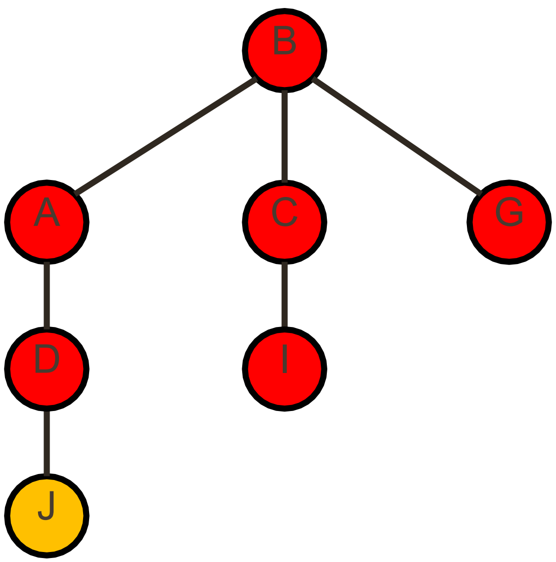


Open (Q):
{}

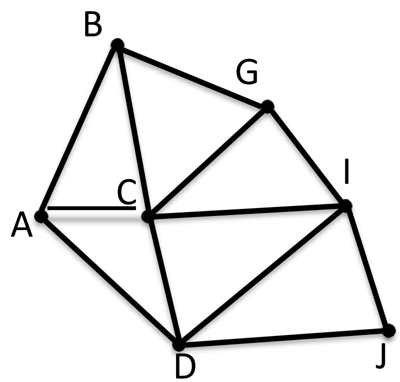
Visited:
{B,A,C,G,D,I,J}

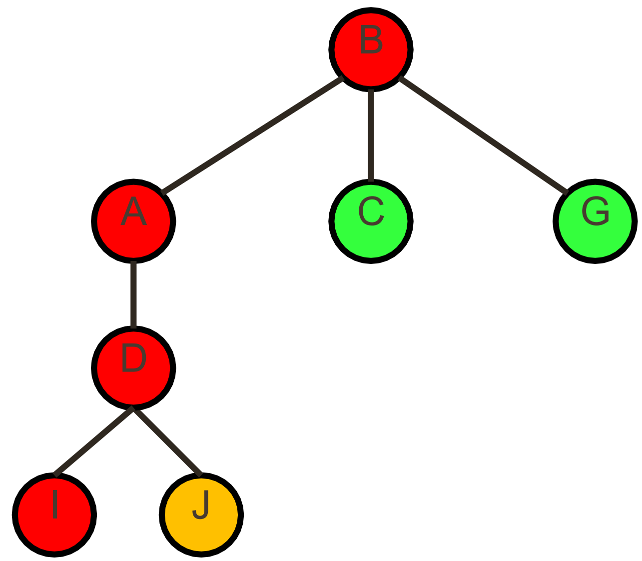
Final path solution: B→A→D→J

Search tree comparison



BFS





DFS