

Artificial Intelligence Foundation – JC3001

Lecture 4: Search I: Uninformed Search I

Prof. Aladdin Ayesh (aladdin.ayesh@abdn.ac.uk)

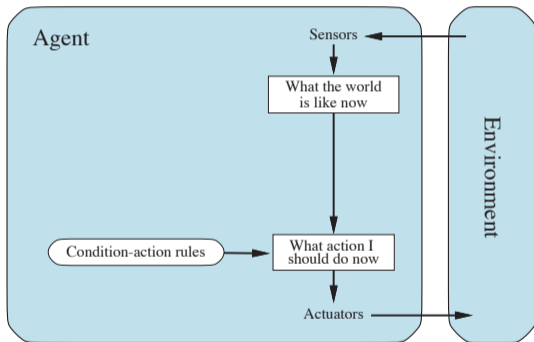
Dr. Binod Bhattarai (binod.bhattarai@abdn.ac.uk)

Dr. Gideon Ogunniye, (g.ogunniye@abdn.ac.uk)

September 2025

Material adapted from:
Russell and Norvig (AIMA Book): Chapter 3 (3.1–3.5)
Malte Helmert (University of Basel)

- Part 1: Introduction
 - ① Introduction to AI ✓
 - ② Agents ✓
- Part 2: Problem-solving
 - ① **Search 1: Uninformed Search**
 - ② Search 2: Heuristic Search
 - ③ Search 3: Local Search
 - ④ Search 4: Adversarial Search
- Part 3: Reasoning and Uncertainty
 - ① Reasoning 1: Constraint Satisfaction
 - ② Reasoning 2: Logic and Inference
 - ③ Probabilistic Reasoning 1: BNs
 - ④ Probabilistic Reasoning 2: HMMs
- Part 4: Planning
 - ① Planning 1: Intro and Formalism
 - ② Planning 2: Algos and Heuristics
 - ③ Planning 3: Hierarchical Planning
 - ④ Planning 4: Stochastic Planning
- Part 5: Learning
 - ① Learning 1: Intro to ML
 - ② Learning 2: Regression
 - ③ Learning 3: Neural Networks
 - ④ Learning 4: Reinforcement Learning
- Part 6: Conclusion
 - ① Ethical Issues in AI
 - ② Conclusions and Discussion



function Simple-Reflex-Agent(*percept*)

returns an action

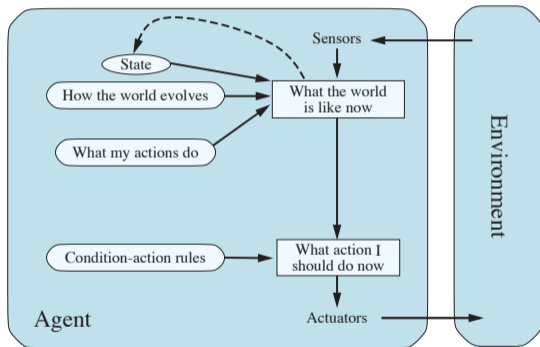
persistent: *rules* ▷ a set of condition-action rules

state \leftarrow interpretInput(*percept*)

rule \leftarrow ruleMatch(*state*)

action \leftarrow *rule*.Action

return *action*



function Model-Based-Reflex-Agent(*percept*)

returns an action

persistent: *state*

t_m

s_m

rules

action

state \leftarrow `updateState`(*state*, *action*, *percept*, *t_m*, *s_m*)

rule \leftarrow `ruleMatch`(*state*, *rules*)

action \leftarrow *rule*.Action

return *action*

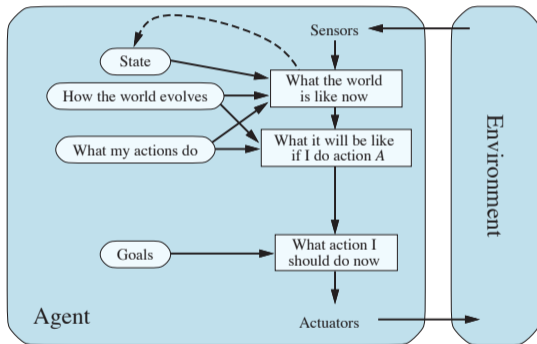
▷ conception of the world

▷ env. dynamics model

▷ sensor dynamics model

▷ a set of condition-action rules

▷ the most recent action



- (Complex) Goals drive behaviour
- Together with a model allows long-term reasoning
- Today, we see how to build such agents using search

- Problem-solving agents
- Search Algorithms (Tree/Graph)
- Uninformed Search Strategies



Outline

1 Problem-solving agents

► Problem-solving agents

► Problem Formulation

► Solving Problems

```
1: function Simple-Problem-Solving-Agent(percept, state)
2:   static  $seq \leftarrow []$ 
3:    $state \leftarrow \text{updateState}(state, percept)$ 
4:   if  $seq$  is empty then
5:      $goal \leftarrow \text{formulateGoal}(state)$ 
6:      $problem \leftarrow \text{formulateProblem}(state, goal)$ 
7:      $seq \leftarrow \text{search}(problem)$ 
8:    $action \leftarrow \text{first}(seq)$ 
9:    $seq \leftarrow \text{rest}(seq)$ 
10:  return  $action$ 
```

▷ *an action sequence*

```
1: function Simple-Problem-Solving-Agent(percept, state)
2:   static seq  $\leftarrow$  []
3:   state  $\leftarrow$  updateState(state, percept)
4:   if seq is empty then
5:     goal  $\leftarrow$  formulateGoal(state)
6:     problem  $\leftarrow$  formulateProblem(state, goal)
7:     seq  $\leftarrow$  search(problem)
8:     action  $\leftarrow$  first(seq)
9:     seq  $\leftarrow$  rest(seq)
10:  return action
```

▷ *an action sequence*

Formulating a goal (via `formulateGoal`) is the first step in problem solving;
what are we trying to achieve?

```
1: function Simple-Problem-Solving-Agent(percept, state)
2:   static  $seq \leftarrow []$ 
3:    $state \leftarrow \text{updateState}(state, percept)$ 
4:   if  $seq$  is empty then
5:      $goal \leftarrow \text{formulateGoal}(state)$ 
6:      $problem \leftarrow \text{formulateProblem}(state, goal)$ 
7:      $seq \leftarrow \text{search}(problem)$ 
8:      $action \leftarrow \text{first}(seq)$ 
9:      $seq \leftarrow \text{rest}(seq)$ 
10:  return  $action$ 
```

▷ *an action sequence*

Problem formulation involves identifying the actions and states that must be considered.

```
1: function Simple-Problem-Solving-Agent(percept, state)
2:   static  $seq \leftarrow []$ 
3:    $state \leftarrow \text{updateState}(state, percept)$ 
4:   if  $seq$  is empty then
5:      $goal \leftarrow \text{formulateGoal}(state)$ 
6:      $problem \leftarrow \text{formulateProblem}(state, goal)$ 
7:      $seq \leftarrow \text{search}(problem)$ 
8:    $action \leftarrow \text{first}(seq)$ 
9:    $seq \leftarrow \text{rest}(seq)$ 
10:  return  $action$ 
```

▷ *an action sequence*

Search involves examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence.

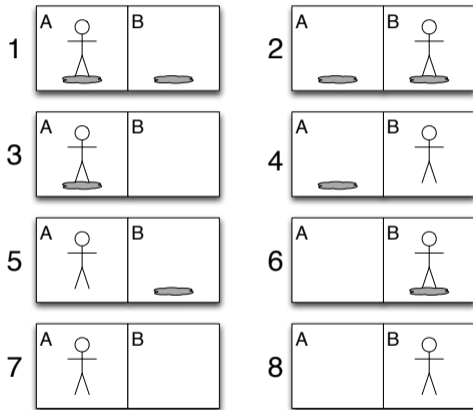
```
1: function Simple-Problem-Solving-Agent(percept, state)
2:   static  $seq \leftarrow []$ 
3:    $state \leftarrow \text{updateState}(state, percept)$ 
4:   if  $seq$  is empty then
5:      $goal \leftarrow \text{formulateGoal}(state)$ 
6:      $problem \leftarrow \text{formulateProblem}(state, goal)$ 
7:      $seq \leftarrow \text{search}(problem)$ 
8:      $action \leftarrow \text{first}(seq)$ 
9:      $seq \leftarrow \text{rest}(seq)$ 
10:  return  $action$ 
```

▷ *an action sequence*

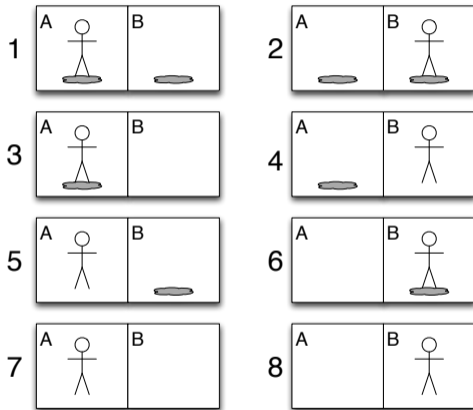
A search algorithm takes a **problem** as an input,
and returns a **solution** in the form of an action sequence.

- **Deterministic, fully observable** \Rightarrow single-state problem
Agent knows exactly which state it will be in; solution is a sequence
- **Non-observable** \Rightarrow conformant problem
Agent may have no idea where it is; solution (if any) is a sequence
- **Nondeterministic** and/or **partially observable** \Rightarrow contingency problem
percepts provide **new** information about current state
solution is a **contingent plan** or a **policy**
often interleave search, execution
- **Unknown state space** \Rightarrow exploration problem (“online”)

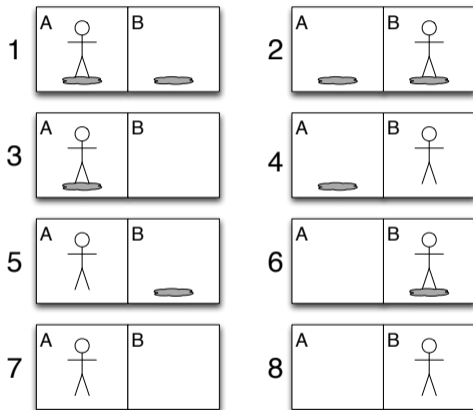
- Single-state, start in #5. *Solution?*



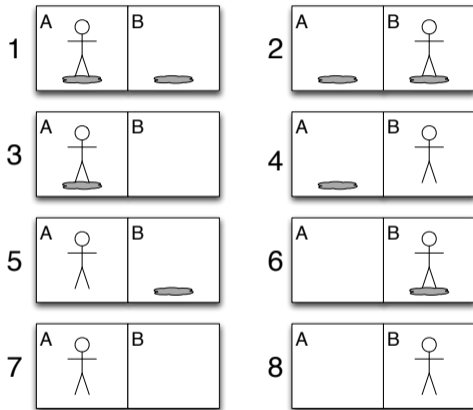
- Single-state, start in #5. *Solution?*
[Right, Suck]



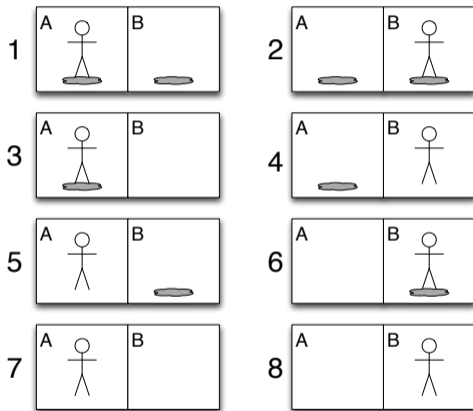
- Single-state, start in #5. *Solution?*
[Right, Suck]
- Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g. **Right** goes to $\{2, 4, 6, 8\}$. *Solution?*



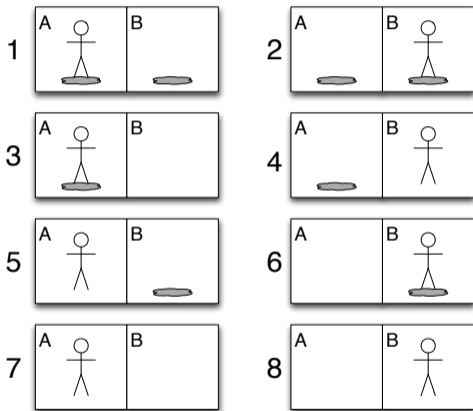
- Single-state, start in #5. *Solution?*
[Right, Suck]
- Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}
e.g. **Right** goes to {2, 4, 6, 8}. *Solution?*
[Right, Suck, Left, Suck]



- Single-state, start in #5. *Solution?*
[Right, Suck]
- Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}
e.g. **Right** goes to {2, 4, 6, 8}. *Solution?*
[Right, Suck, Left, Suck]
- Contingency, start in #5
Murphy's Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution?



- Single-state, start in #5. *Solution?*
[Right, Suck]
- Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}
e.g. **Right** goes to {2, 4, 6, 8}. *Solution?*
[Right, Suck, Left, Suck]
- Contingency, start in #5
Murphy's Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution?
[Right, if dirt then Suck]





Outline

2 Problem Formulation

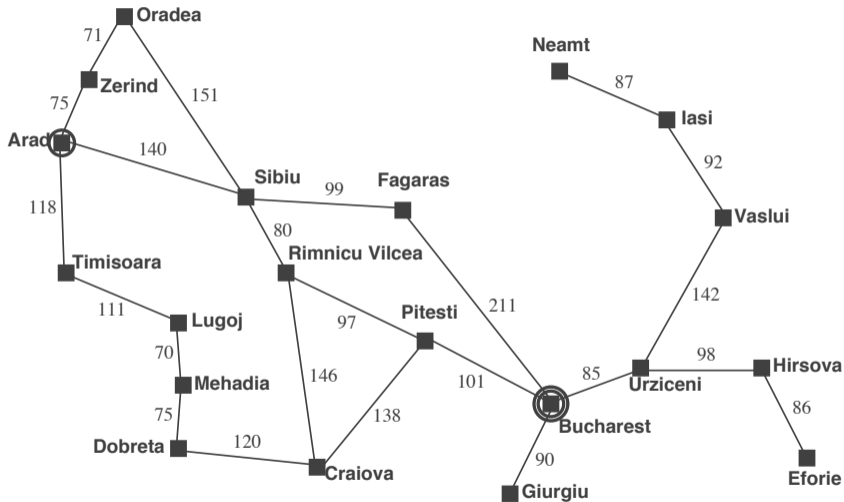
► Problem-solving agents

► Problem Formulation

► Solving Problems

Example: Romania

2 Problem Formulation



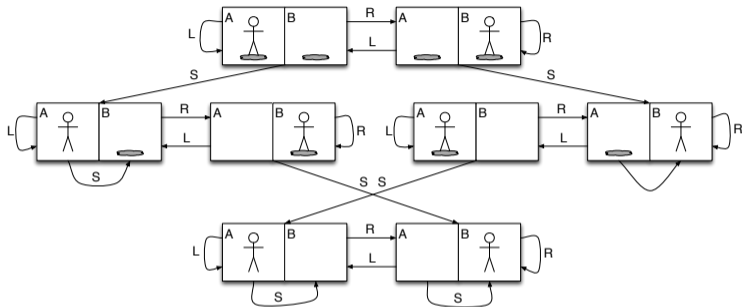
- On holiday in Romania:
 - currently in Arad
 - flight leaves tomorrow from Bucharest
- **Formulate goal:** be in Bucharest
- **Formulate problem:**
 - states:** various cities
 - actions:** drive between cities
- **Find solution:**
 - sequence of cities, e.g. Arad, Sibiu, Fagaras, Bucharest

- A problem is defined by five items:
 - **initial state** e.g., “at Arad”
 - **actions** e.g. drive the route Arad \rightarrow Zerind
 - **successor function** $S(x)$ = set of action-state pairs
e.g., $S(\text{Arad}) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots\}$
 - **goal test**, can be
explicit, e.g., $x = \text{“at Bucharest”}$
implicit, e.g., $\text{NoDirt}(x)$
 - **path cost** (additive)
e.g., sum of distances, number of actions executed, etc.
 $c(x, a, y)$ is the **step cost**, assumed to be ≥ 0
- A **solution** is a sequence of actions leading from the initial state to a goal state

- Real world is absurdly **complex**
⇒ state space must be abstracted for problem solving
- (Abstract) **state** = set of real states
- (Abstract) **action** = complex combination of real actions
e.g., “Arad → Zerind” represents a set of possible routes, detours, stops, etc.
- For realizability, **any** real state “in Arad” must get to some real state “in Zerind”
- (Abstract) **solution** = set of real paths that are solutions in the real world
- Each abstract action should be “easier” than the original problem!

Example: Vacuum world state space graph

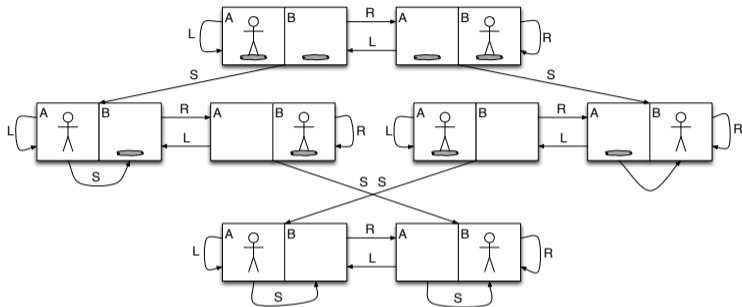
2 Problem Formulation



- What are the?
 - States:
 - Actions:
 - Goal test:
 - Path cost:

Example: Vacuum world state space graph

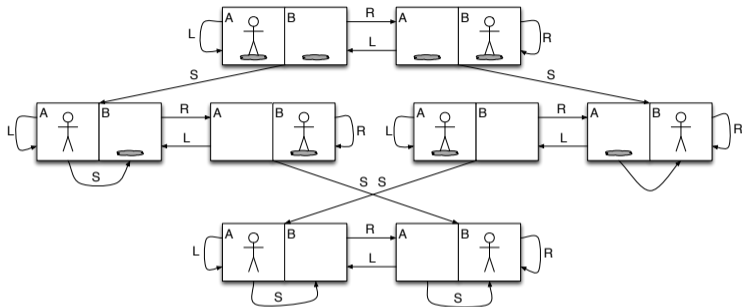
2 Problem Formulation



- What are the?
 - States: integer dirt and robot locations (ignore dirt amounts etc.)
 - Actions:
 - Goal test:
 - Path cost:

Example: Vacuum world state space graph

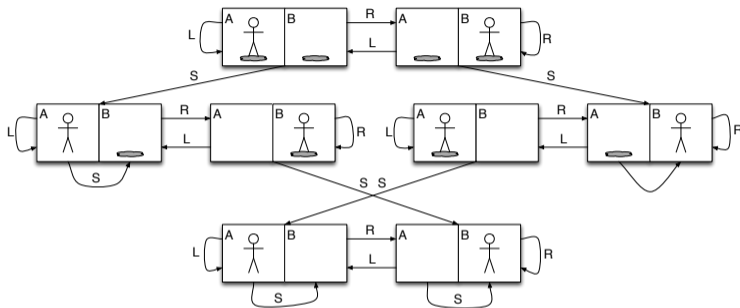
2 Problem Formulation



- What are the?
 - States: integer dirt and robot locations (ignore dirt amounts etc.)
 - Actions: **Left, Right, Suck, NoOp**
 - Goal test:
 - Path cost:

Example: Vacuum world state space graph

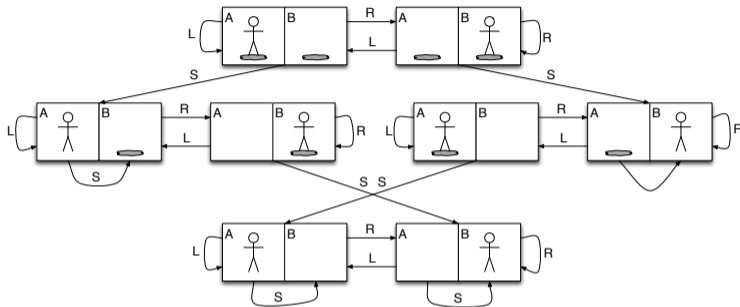
2 Problem Formulation



- What are the?
 - States: integer dirt and robot locations (ignore dirt amounts etc.)
 - Actions: **Left, Right, Suck, NoOp**
 - Goal test: no dirt
 - Path cost:

Example: Vacuum world state space graph

2 Problem Formulation



- What are the?
 - States: integer dirt and robot locations (ignore dirt amounts etc.)
 - Actions: **Left, Right, Suck, NoOp**
 - Goal test: no dirt
 - Path cost: 1 per action (0 for **NoOp**)

Example: The 8-puzzle

2 Problem Formulation

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

- What are the?
 - States:
 - Actions:
 - Goal test:
 - Path cost:

Example: The 8-puzzle

2 Problem Formulation

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

- What are the?
 - States: integer locations of tiles (ignore intermediate positions)
 - Actions:
 - Goal test:
 - Path cost:

Example: The 8-puzzle

2 Problem Formulation

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

- What are the?
 - States: integer locations of tiles (ignore intermediate positions)
 - Actions: move blank left, right, up, down (ignore unjamming etc.)
 - Goal test:
 - Path cost:

Example: The 8-puzzle

2 Problem Formulation

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

- What are the?
 - States: integer locations of tiles (ignore intermediate positions)
 - Actions: move blank left, right, up, down (ignore unjamming etc.)
 - Goal test: = goal state (given)
 - Path cost:

Example: The 8-puzzle

2 Problem Formulation

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

- What are the?
 - States: integer locations of tiles (ignore intermediate positions)
 - Actions: move blank left, right, up, down (ignore unjamming etc.)
 - Goal test: = goal state (given)
 - Path cost: 1 per action

Example: The 8-puzzle

2 Problem Formulation

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

- What are the?
 - States: integer locations of tiles (ignore intermediate positions)
 - Actions: move blank left, right, up, down (ignore unjamming etc.)
 - Goal test: = goal state (given)
 - Path cost: 1 per action

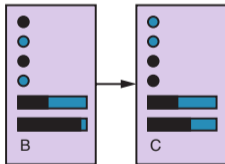
[Note: optimal solution of n-Puzzle family is NP-hard]

How do we define a state-space?

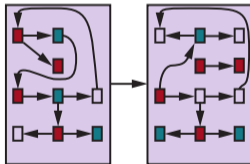
- For finite state spaces, we can enumerate
Vacuum World – 3 binary vars: position (A/B), dirtA, dirtB
8-puzzle – $9!$ possible states ($9!/2$ reachable states)
- For infinite state spaces, we need to define a function that generates the next state



(a) Atomic



(b) Factored



(c) Structured



Outline

3 Solving Problems

► Problem-solving agents

► Problem Formulation

► Solving Problems

Having defined a problem, we now need to solve it

We do this by searching through the state space

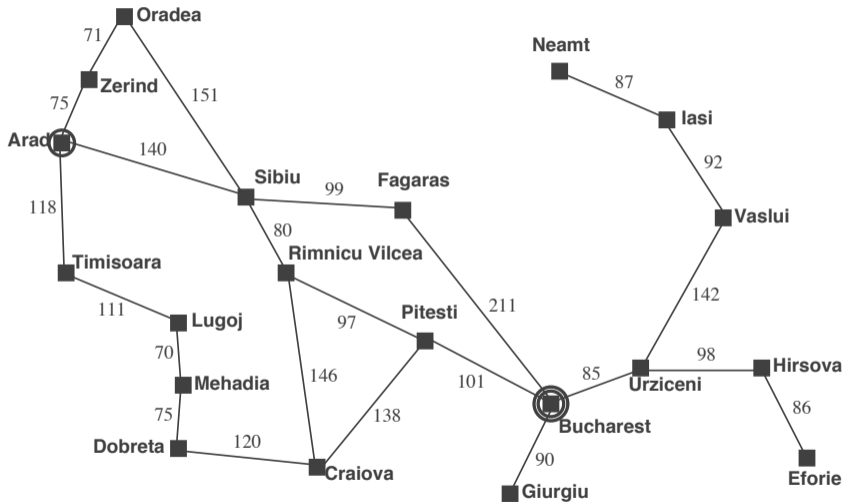
```
1: function treeSearch(problem  $p$ , strategy  $s$ )
2:    $frontier.add(p.initial)$ 
3:   loop
4:     if  $frontier$  is empty then
5:       return  $fail$ 
6:      $n \leftarrow s.removeChoice(frontier)$ 
7:     if  $p.goalTest(n.state)$  then
8:       return  $getPath(n)$ 
9:     for all  $a \in p.actions(n)$  do
10:       $n' \leftarrow a.result(n.state)$ 
11:       $frontier.add(n')$ 
```

What is a node here?

3 Solving Problems

- $n.STATE$: the corresponding state in the state-space
- $n.PARENT$: the node that generated n
- $n.ACTION$: the action applied to $n.PARENT$ which resulted in n
- $n.PATH-COST$: the cost of the path from the initial state to n , as indicated by the $PARENT$ pointers

- Expansion takes place by applying successor functions to the selected node, providing new states.
- Nodes that have been generated, but not expanded are referred to as the **frontier** or **fringe**.
- Different search **strategies** behave very differently.
- The search tree is not the same as the state space. In the route finding example, there are 20 nodes in the state space, but the search tree has an infinite number of nodes.



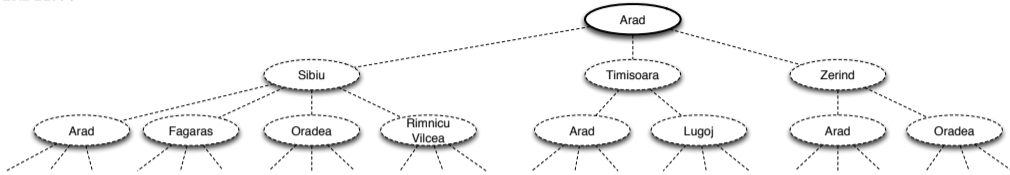
Basic idea:

offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

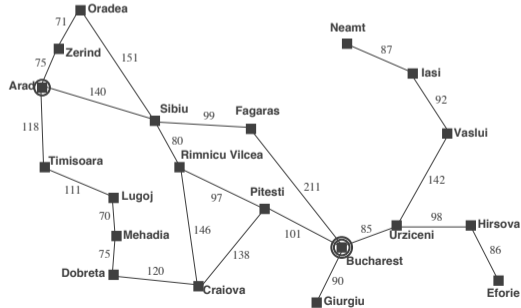
```
1: function treeSearch(problem  $p$ , strategy  $s$ )
2:    $frontier.add(p.initial)$ 
3:   loop
4:     if  $frontier$  is empty then
5:       return fail
6:      $n \leftarrow s.removeChoice(frontier)$ 
7:     if  $p.goalTest(n.state)$  then
8:       return getPath( $n$ )
9:     for all  $a \in p.actions(n)$  do
10:       $n' \leftarrow a.result(n.state)$ 
11:       $frontier.add(n')$ 
```

Tree search example

3 Solving Problems

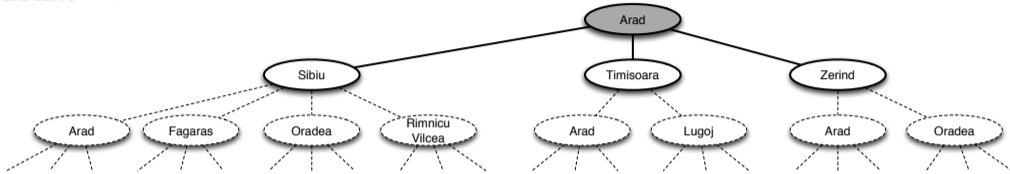


Initial State

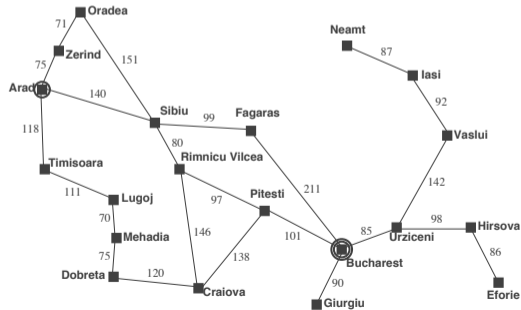


Tree search example

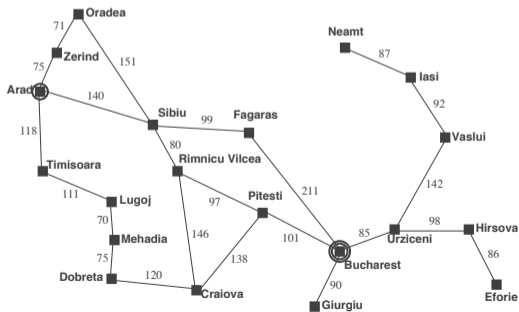
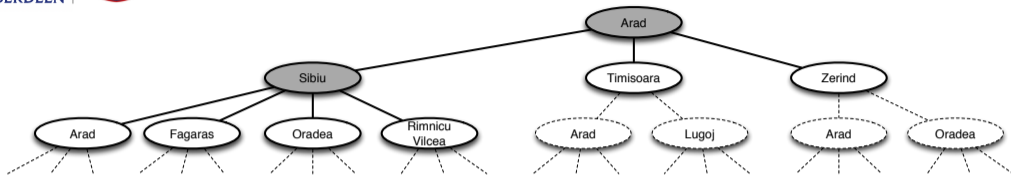
3 Solving Problems



After expanding Arad



3 Solving Problems



After expanding Sibiu

To continue in the next session.