Problem Solving via Search

Outline

I. Problem formulation

II. Example problems

III. Search algorithms

I. Problem-Solving Agent

♣ A reflex agent cannot operate when the mapping from states to to actions becomes too large to store.

```
if current percepts then action
```

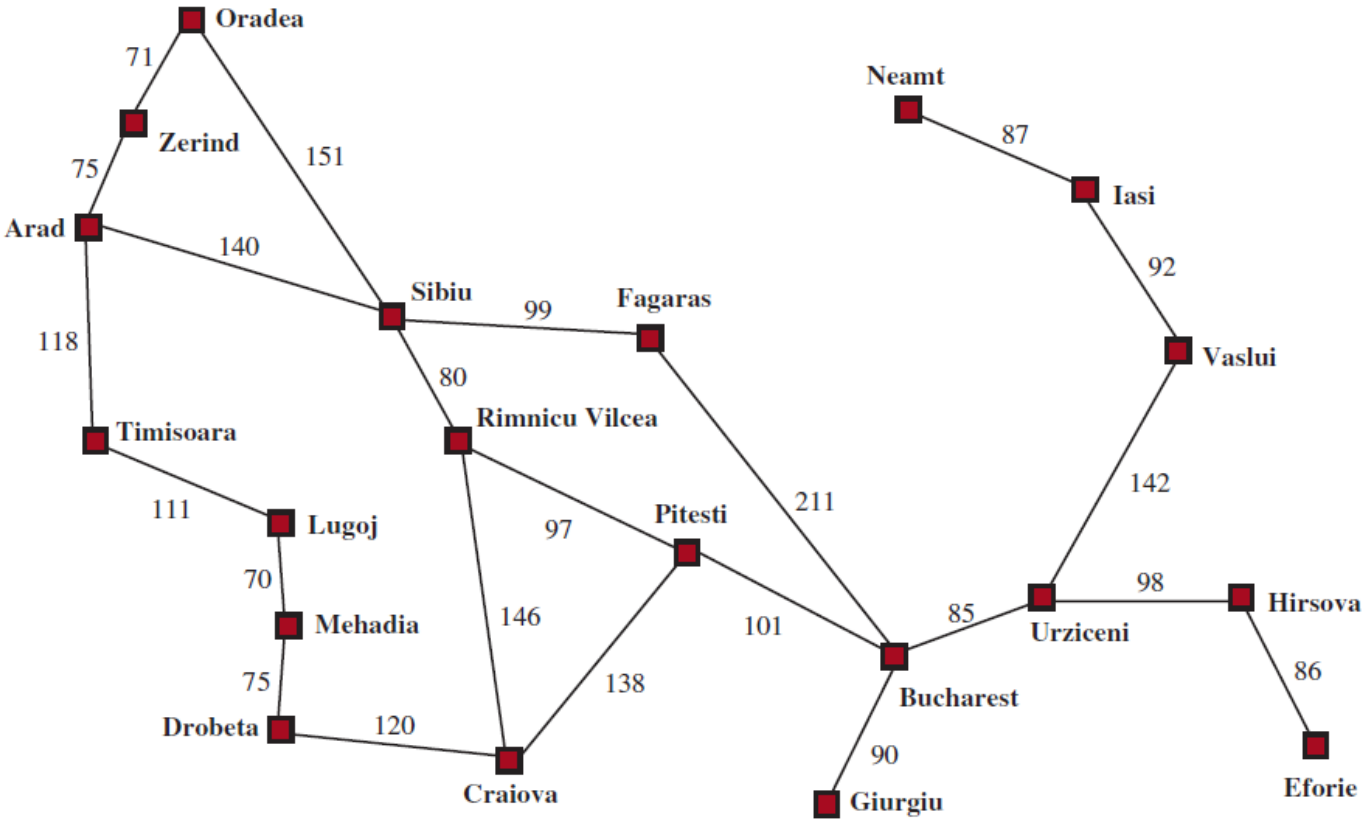
- A problem-solving agent is one kind of goal-based agent:
  - ♦ It considers states with no internal structure, i.e., in *atomic* representations.
- Problems are solved by general-purpose *search* algorithms.

1975 ACM Turing Award Lecture:

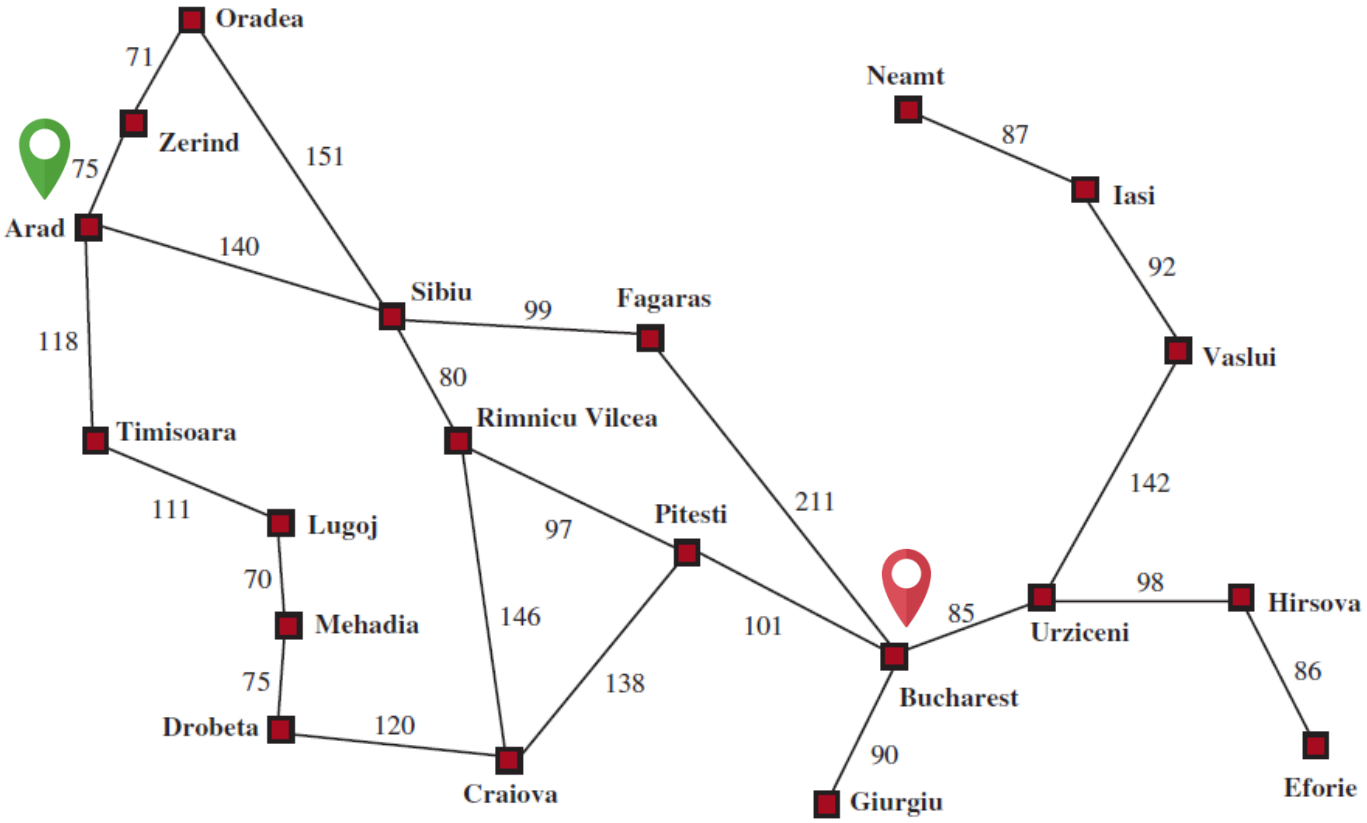
Computer science as empirical inquiry: symbols and search

Allen Newell and Herbert Simon

Sightseeing Trip in Romania

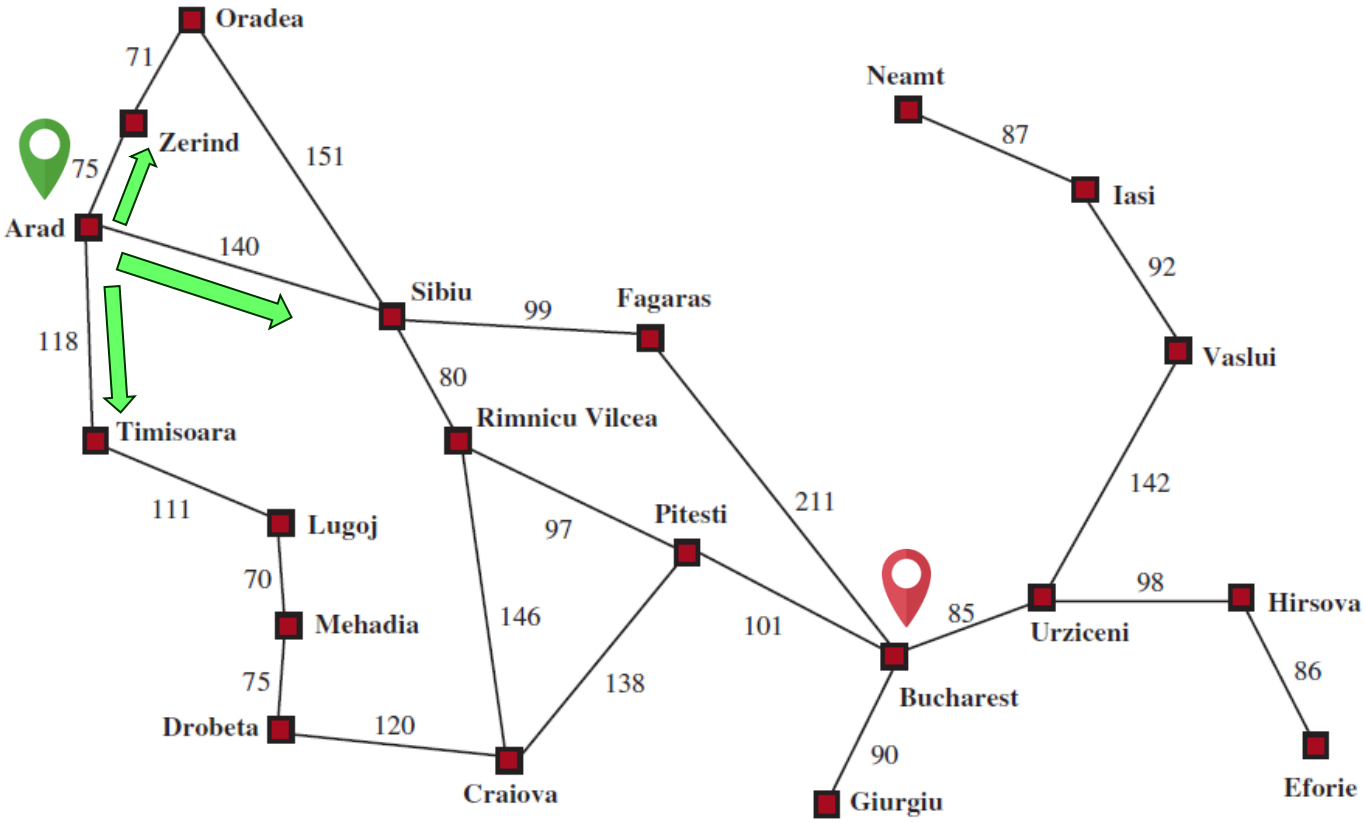


Sightseeing Trip in Romania



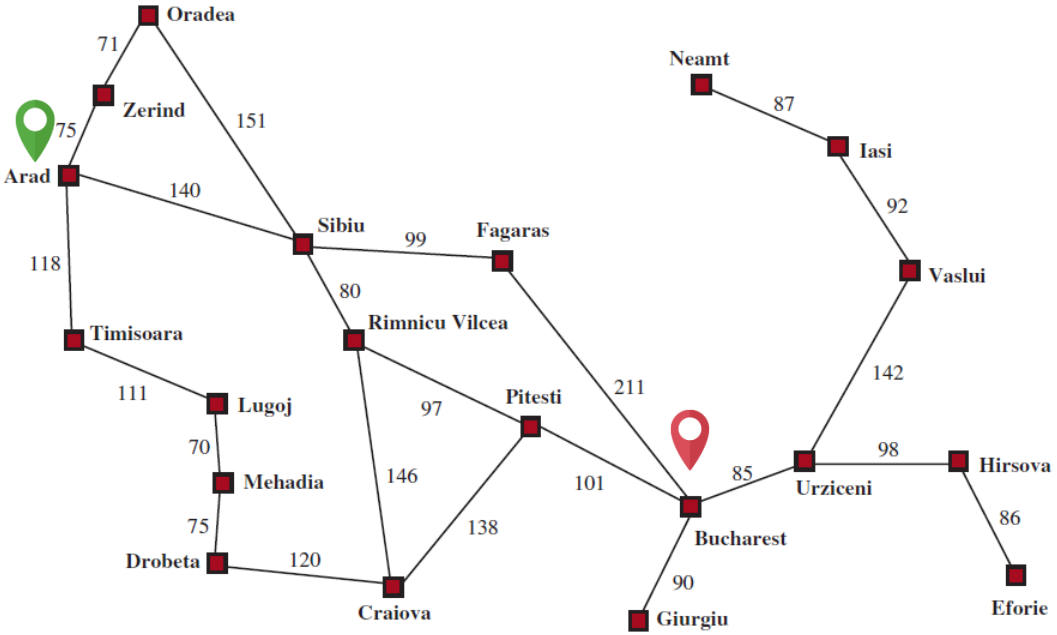
How to get to Bucharest from Arad?

Sightseeing Trip in Romania



How to get to Bucharest from Arad?

Four-Phase Problem Solving



- Goal formulation
- Problem formulation

  states: cities

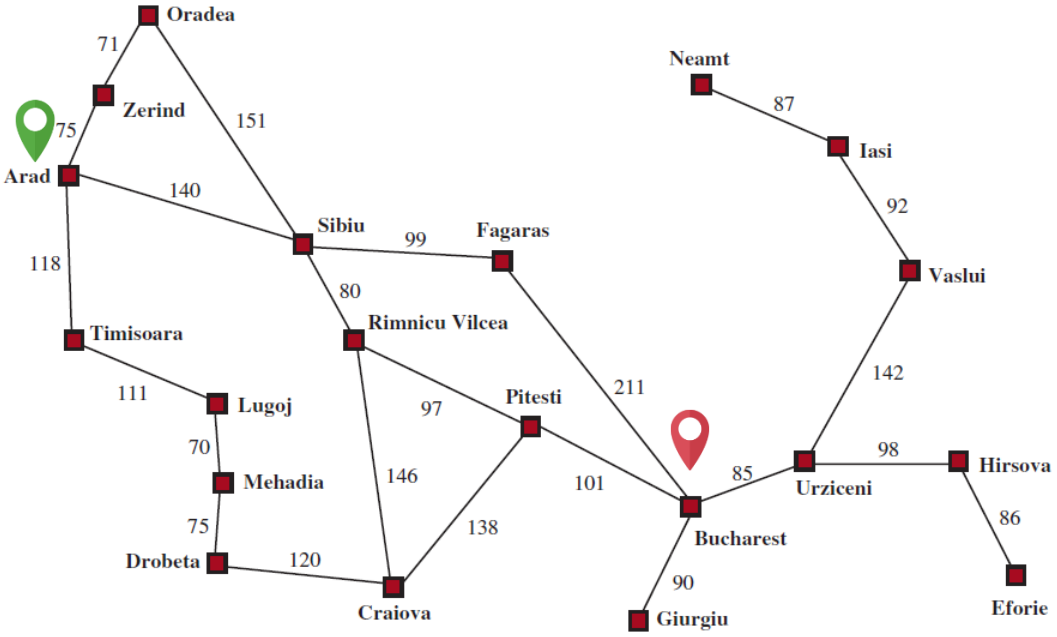
  action: travel from one city to an adjacent city
- Search

  Find a solution.

  ↑

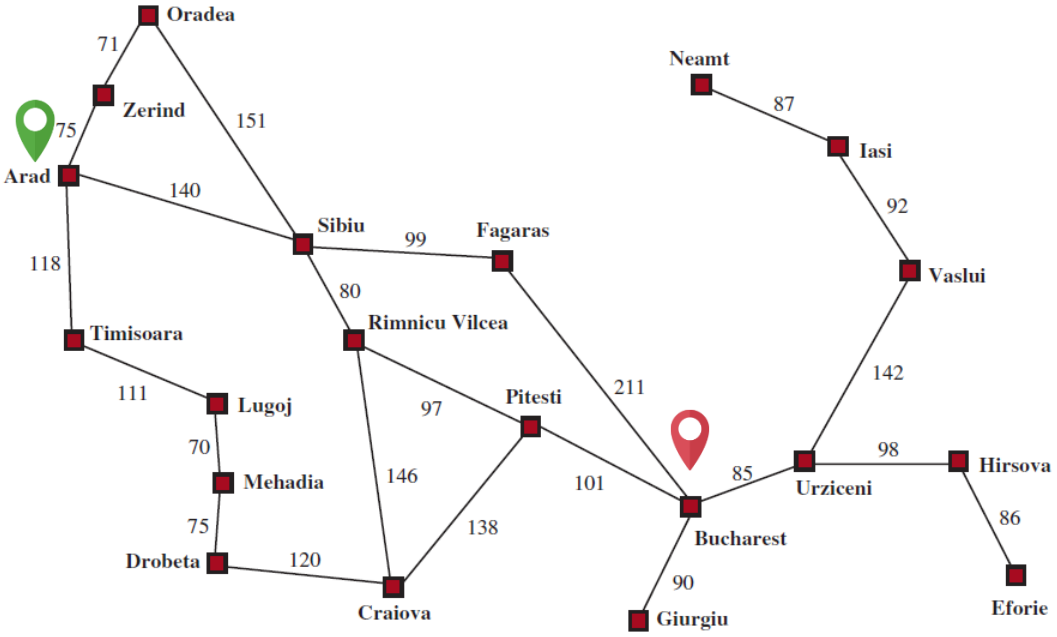
  A sequence of actions to reach the goal
- Execution

Search Problem



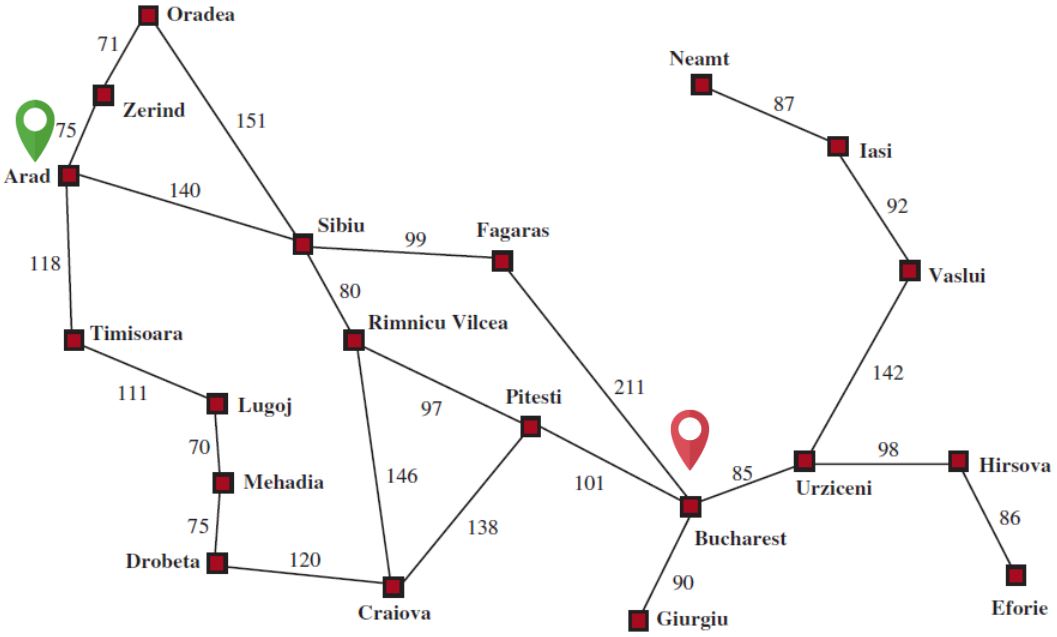
- State space (as a graph)

Search Problem



- State space (as a graph)
- Initial state (e.g., Arad)

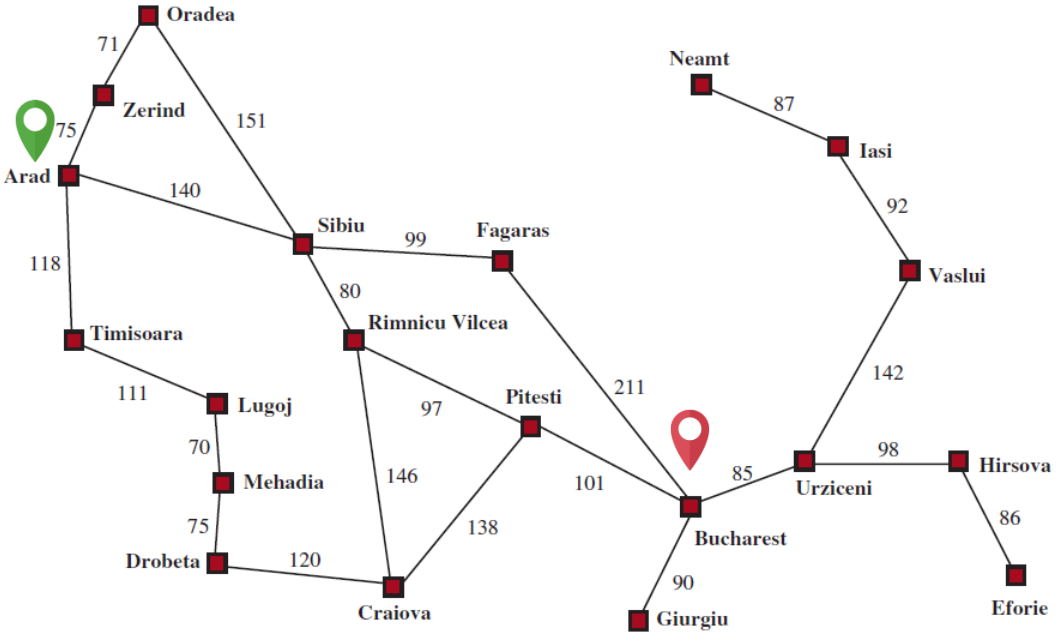
Search Problem



- State space (as a graph)
- Initial state (e.g., Arad)
- Goal state(s) (e.g., Bucharest)

```
IS-GOAL(Fagaras)
```

Search Problem

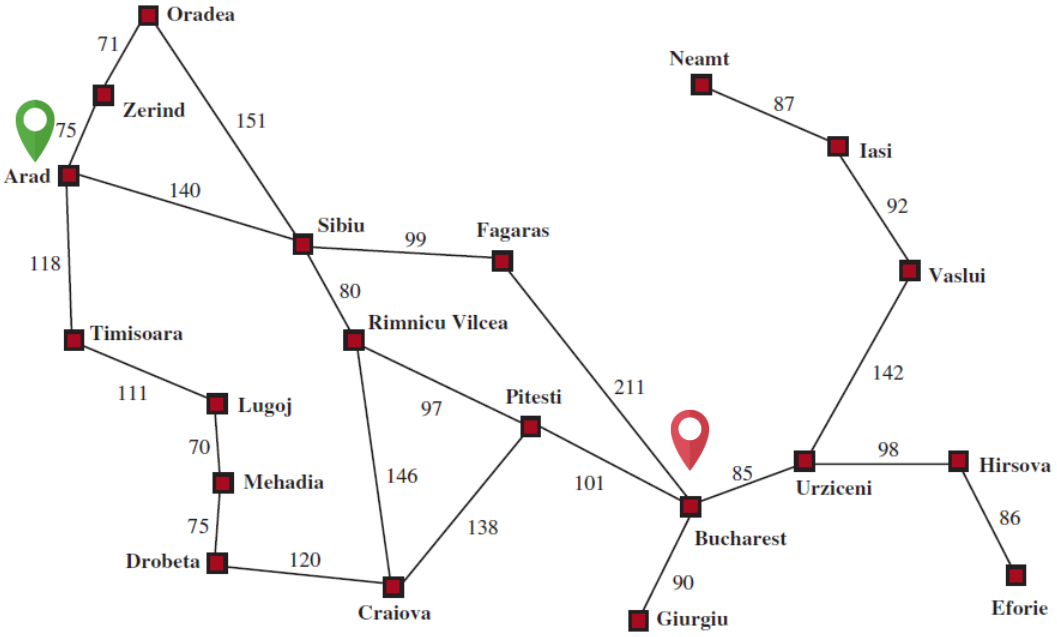


- State space (as a graph)
- Initial state (e.g., Arad)
- Goal state(s) (e.g., Bucharest)

```
IS-GOAL(Fagaras)
```

- Actions

Search Problem



- State space (as a graph)
- Initial state (e.g., Arad)
- Goal state(s) (e.g., Bucharest)

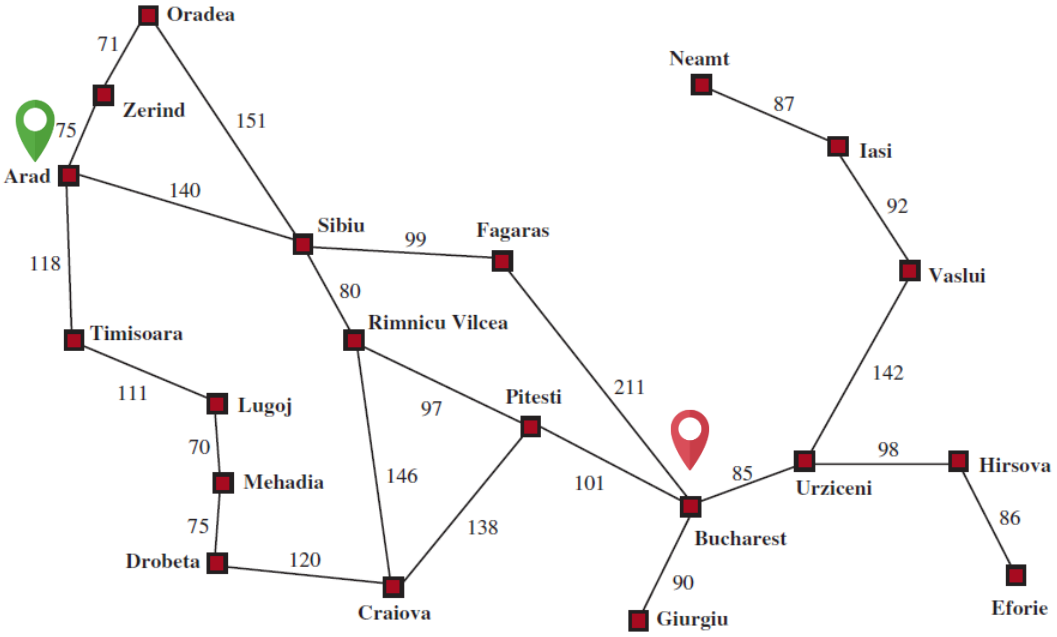
```
IS-GOAL(Fagaras)
```

- Actions

ACTIONS(s): a finite set of actions executable at state s.

ACTIONS($Arad$) = $\{ToSibiu, ToTimisoara, ToZerind\}$

Search Problem



- State space (as a graph)
- Initial state (e.g., Arad)
- Goal state(s) (e.g., Bucharest)

```
IS-GOAL(Fagaras)
```

- Actions

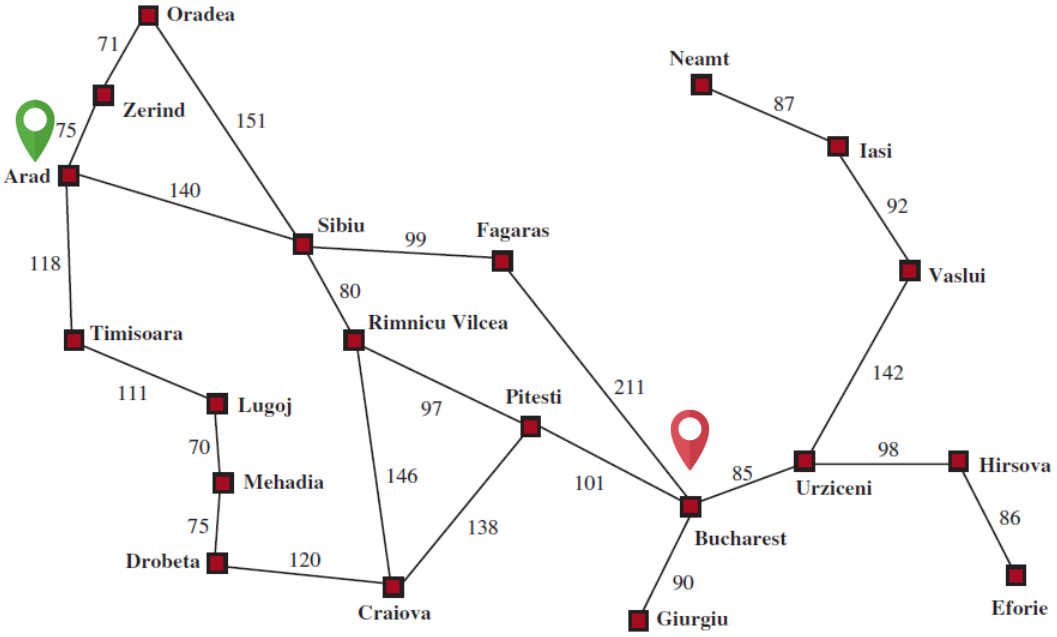
ACTIONS(s): a finite set of actions executable at state s.

ACTIONS($Arad$) = $\{ToSibiu, ToTimisoara, ToZerind\}$

- Transition model

RESULT($Arad, ToZerind$) = $Zerind$

Search Problem



- State space (as a graph)
- Initial state (e.g., Arad)
- Goal state(s) (e.g., Bucharest)

```
IS-GOAL(Fagaras)
```

- Actions

ACTIONS(s): a finite set of actions executable at state s.

ACTIONS($Arad$) = $\{ToSibiu, ToTimisoara, ToZerind\}$

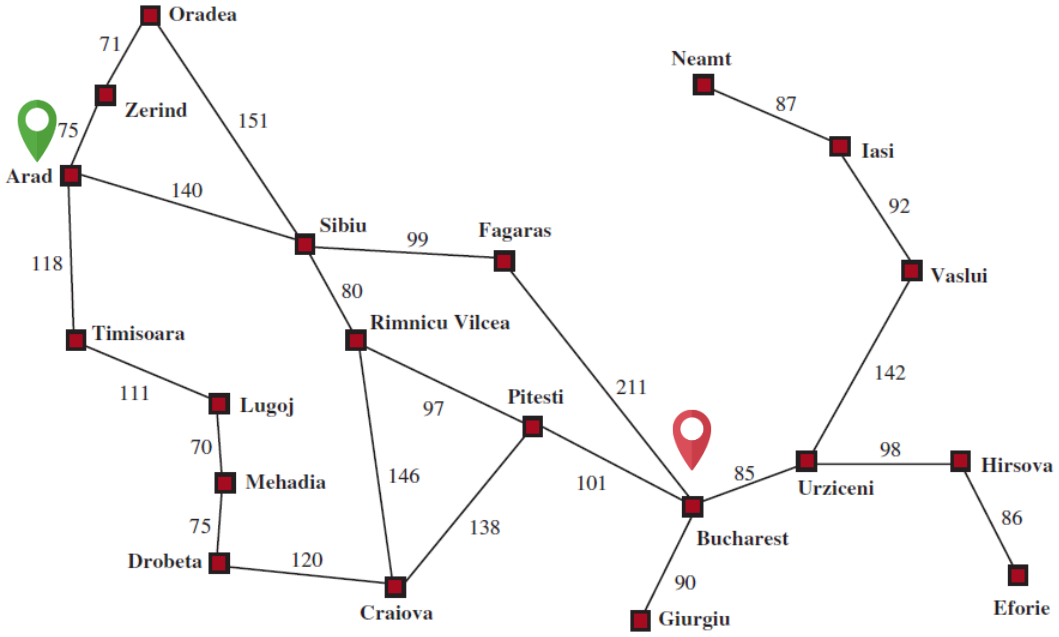
- Transition model

RESULT($Arad, ToZerind$) = $Zerind$

- Action cost function

$c(s, a, s')$: cost of applying action a in state s to reach state s'.

Search Problem



- State space (as a graph)
- Initial state (e.g., Arad)
- Goal state(s) (e.g., Bucharest)

```
IS-GOAL(Fagaras)
```

- Actions

ACTIONS(s): a finite set of actions executable at state s.

ACTIONS($Arad$) = $\{ToSibiu, ToTimisoara, ToZerind\}$

- Transition model

RESULT($Arad, ToZerind$) = $Zerind$

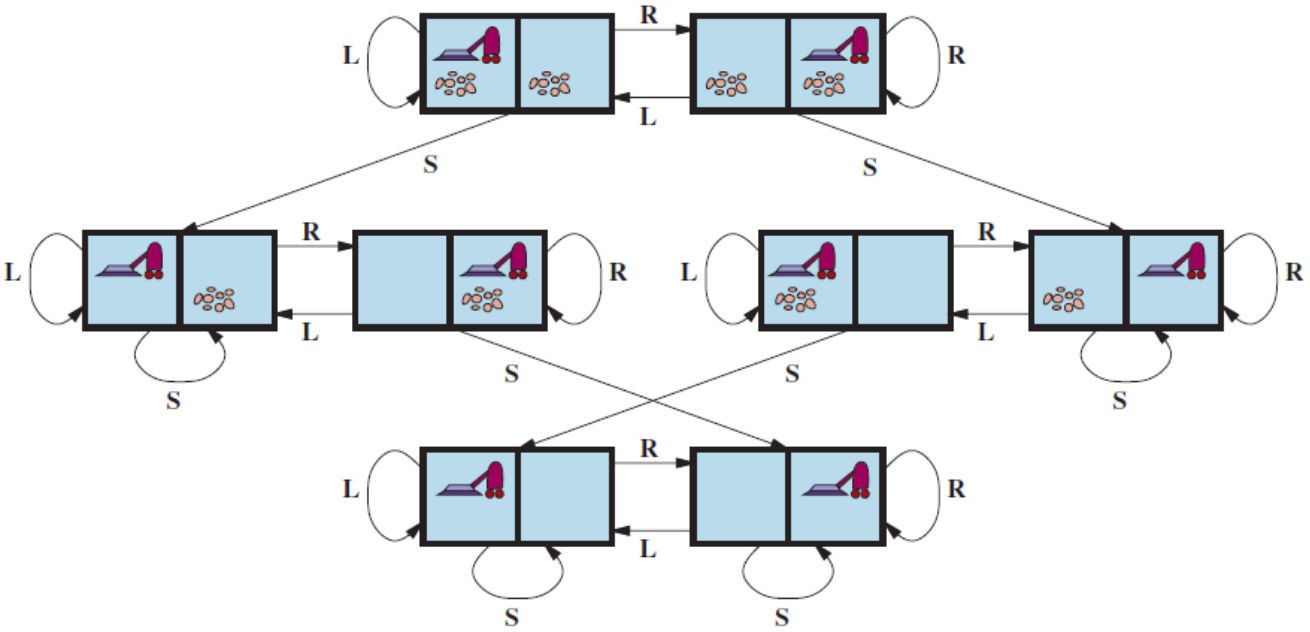
- Action cost function

$c(s, a, s')$: cost of applying action a in state s to reach state s'.

- Solution: initial state ⇝ goal state (e.g., Arad – Sibiu – Fagaras – Bucharest)

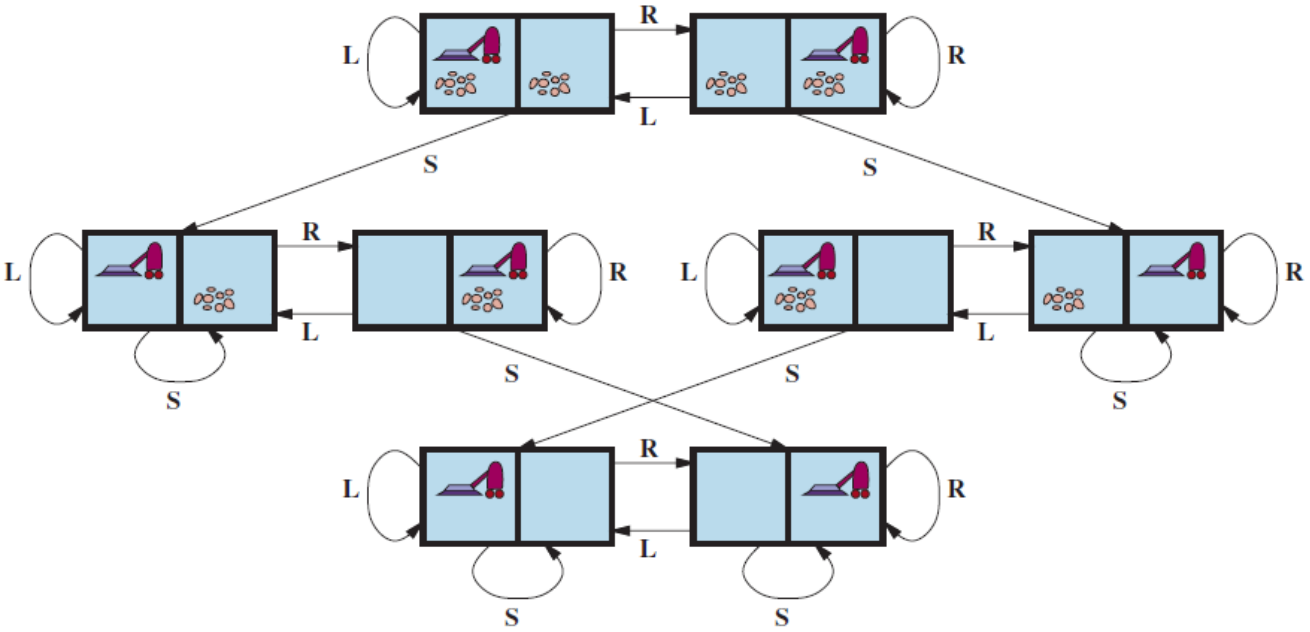
II. Example 1: Vacuum World

State-space graph:



II. Example 1: Vacuum World

State-space graph:



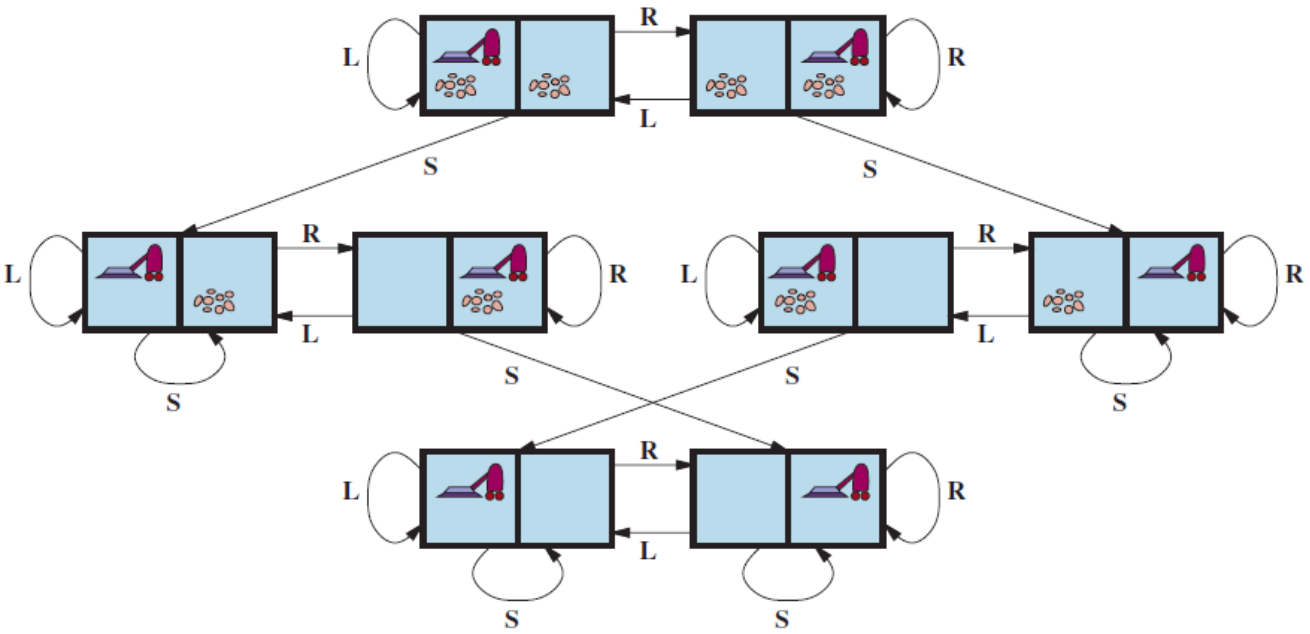
Agent in either cell

↓

2

II. Example 1: Vacuum World

State-space graph:



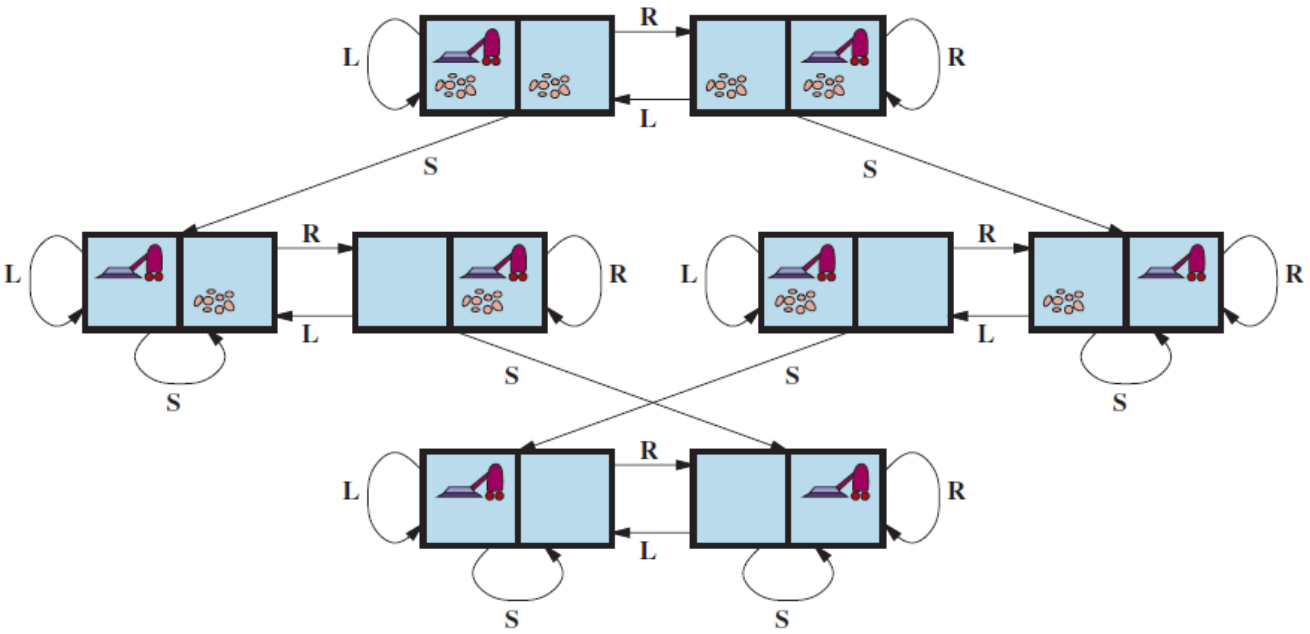
Agent in either cell

$2 \cdot 2$

Left cell has dirt or not

II. Example 1: Vacuum World

State-space graph:



Agent in either cell

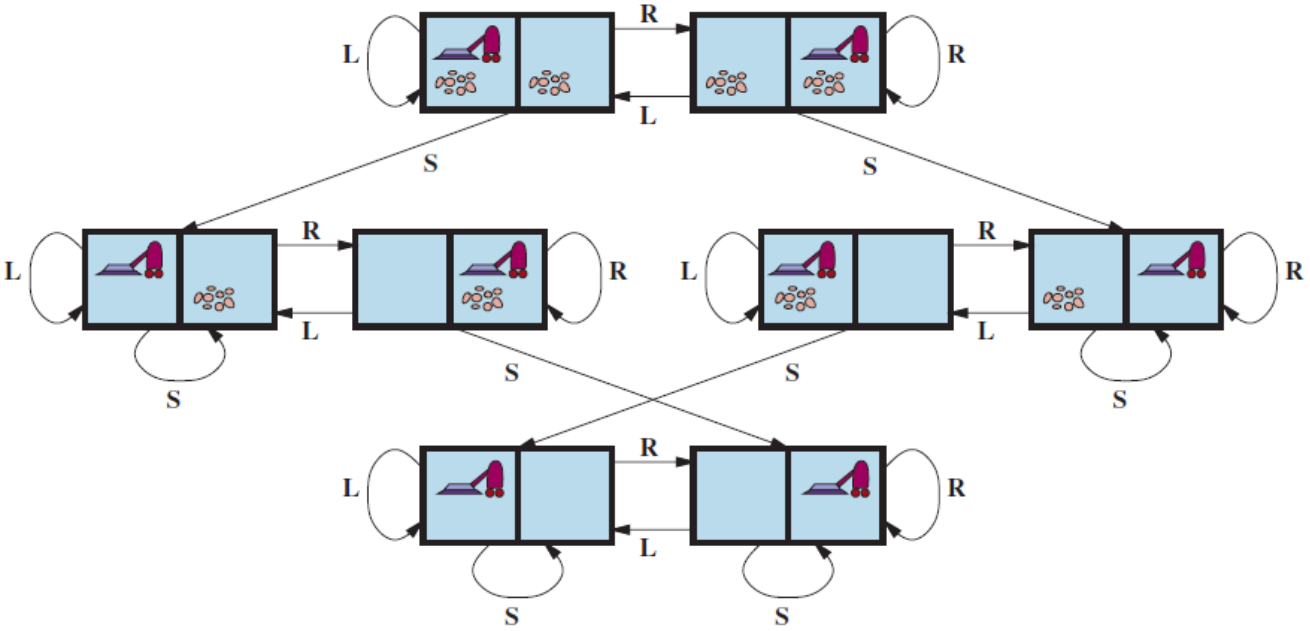
$$2 \cdot 2 \cdot 2 = 8 \text{ states}$$

Left cell has dirt or not

Right cell has dirt or not

II. Example 1: Vacuum World

State-space graph:



- Actions: *Suck*, *Left*, *Right*

Agent in either cell

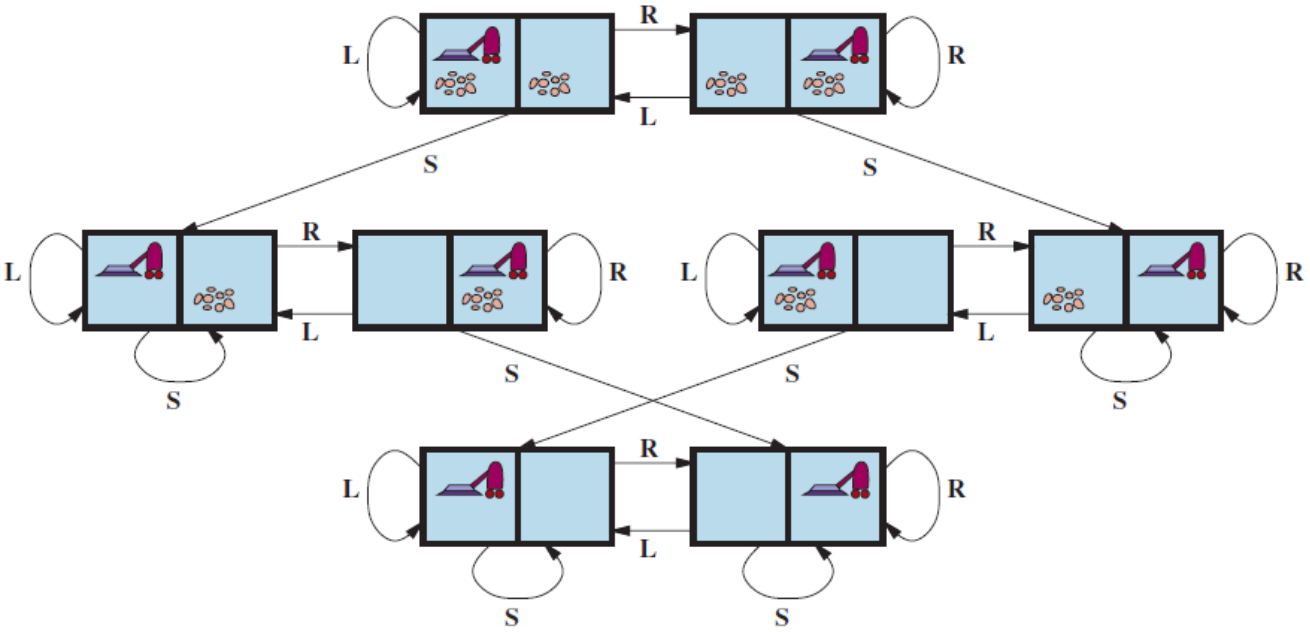
$2 \cdot 2 \cdot 2 = 8$ states

Left cell has dirt or not

Right cell has dirt or not

II. Example 1: Vacuum World

State-space graph:



- Actions: *Suck*, *Left*, *Right*
- Goal: every cell is clean.

Agent in either cell

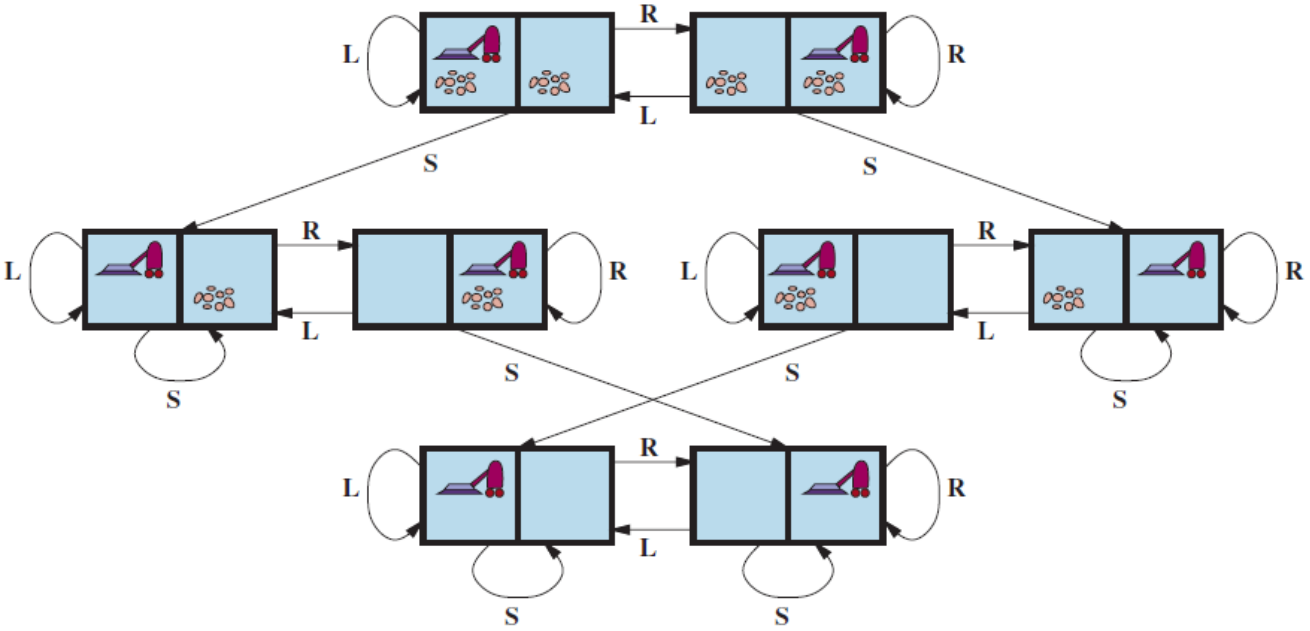
$$2 \cdot 2 \cdot 2 = 8 \text{ states}$$

Left cell has dirt or not

Right cell has dirt or not

II. Example 1: Vacuum World

State-space graph:



- Actions: *Suck*, *Left*, *Right*
- Goal: every cell is clean.
- Cost: 1 for each action

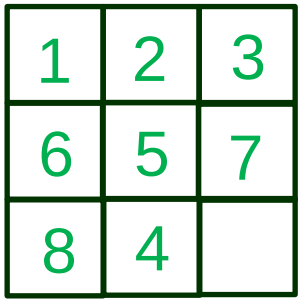
Agent in either cell

$$2 \cdot 2 \cdot 2 = 8 \text{ states}$$

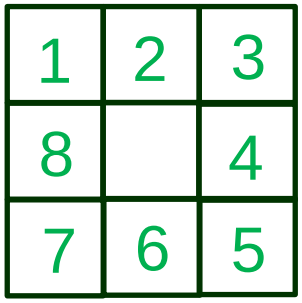
Left cell has dirt or not

Right cell has dirt or not

Example 2: 8-Puzzle



Initial state



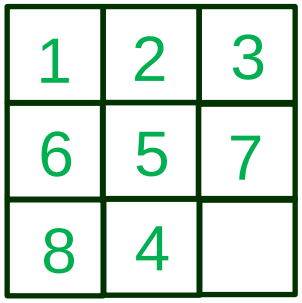
Goal state

- Actions: ways of sliding a tile (adjacent to the blank space).

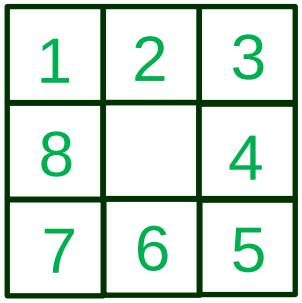
Left, Right, Up, Down

Applied to the
right neighbor
of the blank space

Example 2: 8-Puzzle



Initial state



Goal state

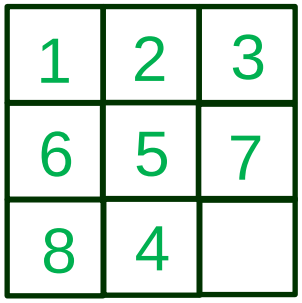
- Actions: ways of sliding a tile (adjacent to the blank space).

Left, Right, Up, Down

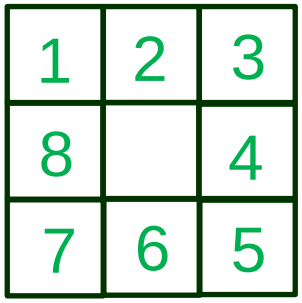
Applied to the right neighbor of the blank space

Only two possible actions at the initial state: *Right* and *Down.*

Example 2: 8-Puzzle



Initial state



Goal state

- Actions: ways of sliding a tile (adjacent to the blank space).

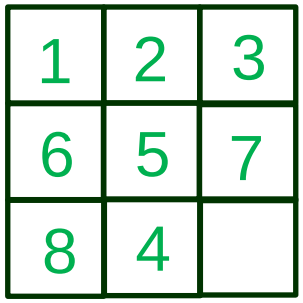
Left, Right, Up, Down

Applied to the right neighbor of the blank space

Only two possible actions at the initial state: *Right* and *Down.*

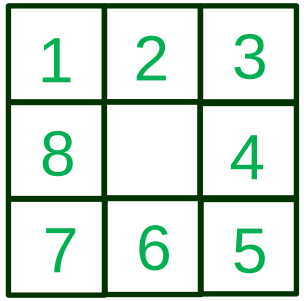
Cost 1 for each action.

Solution to an 8-Puzzle

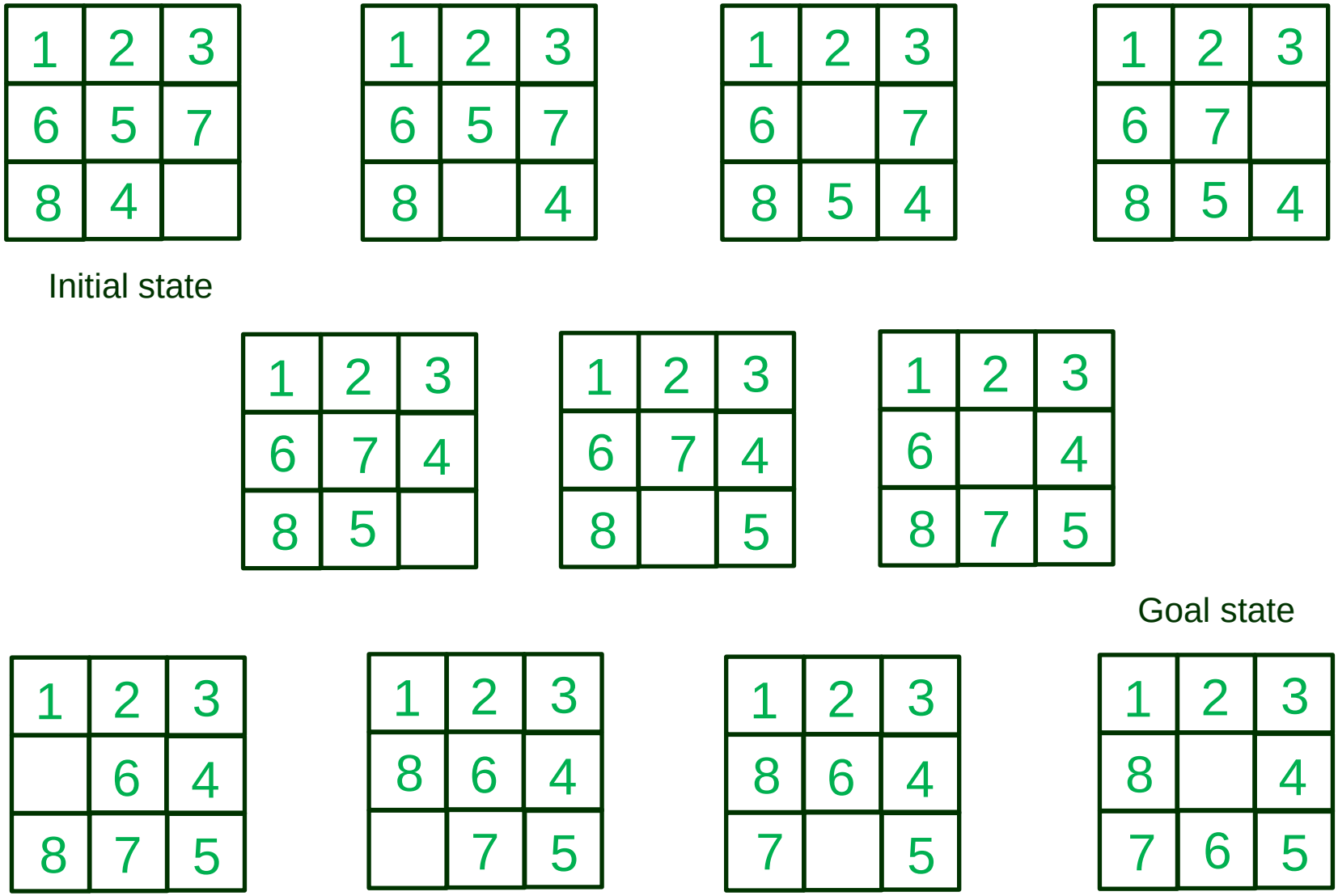


Initial state

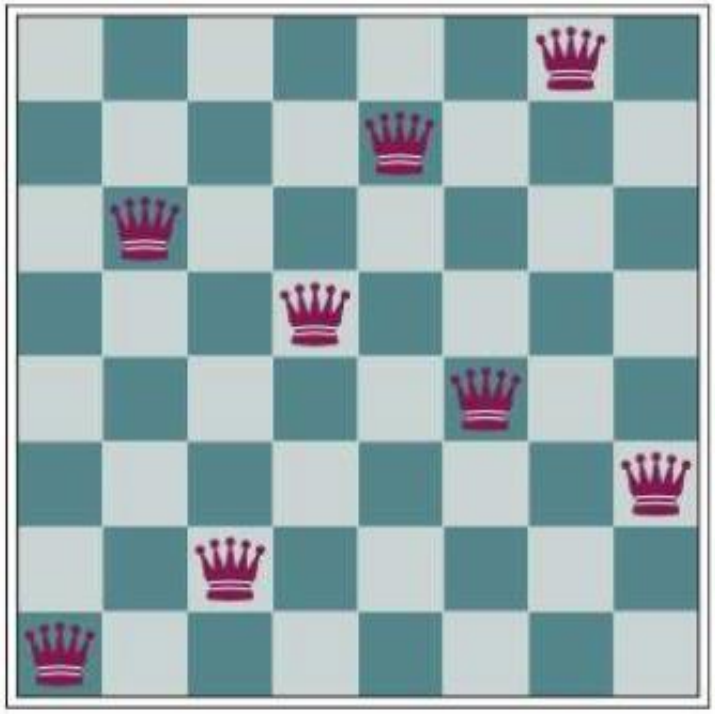
Goal state



Solution to an 8-Puzzle



8-Queens Problem



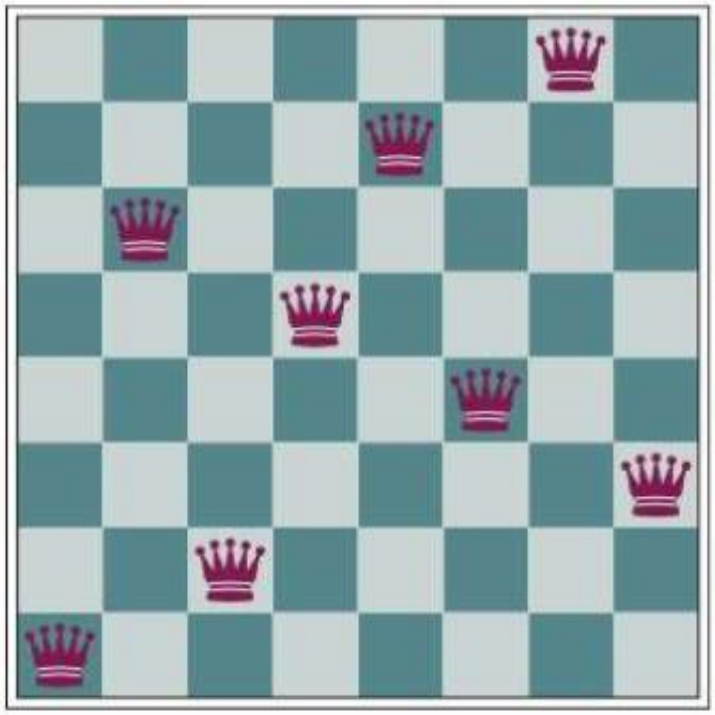
- Goal: A placement of 8 queens on the chess board in which no queen attacks another.

  ↓

  same row, column, or diagonal

- Initial state: no queen on the board.

8-Queens Problem



Close to a solution

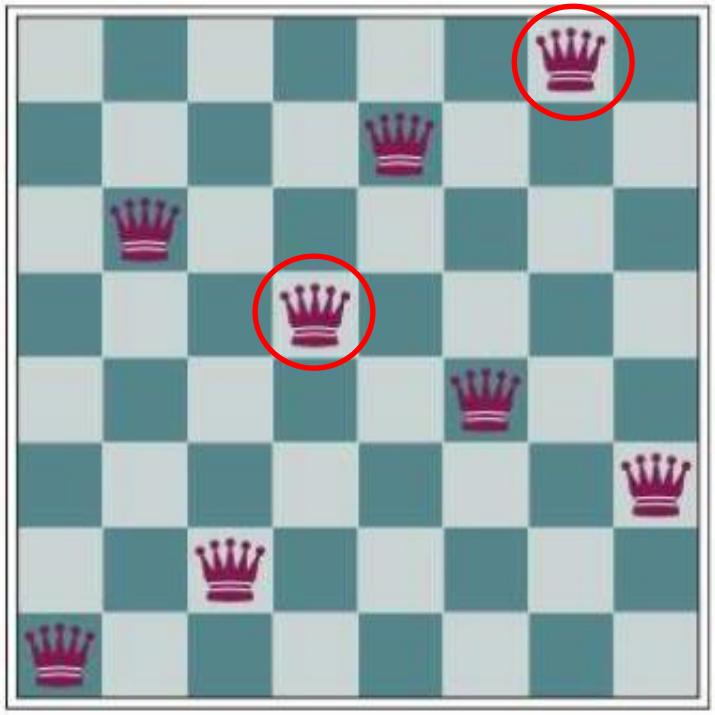
- Goal: A placement of 8 queens on the chess board in which no queen attacks another.

  ↓

  same row, column, or diagonal

- Initial state: no queen on the board.

8-Queens Problem



Close to a solution

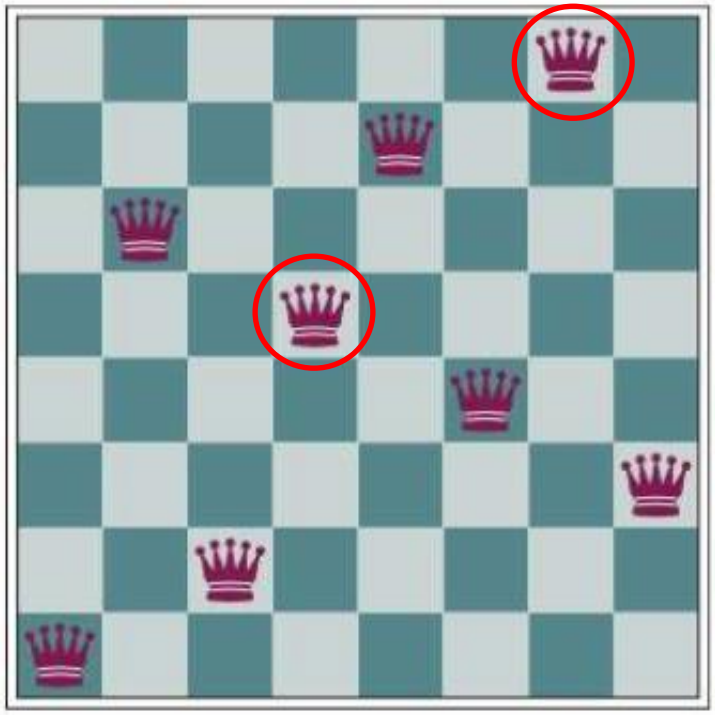
- Goal: A placement of 8 queens on the chess board in which no queen attacks another.

  ↓

  same row, column, or diagonal

- Initial state: no queen on the board.

8-Queens Problem



Close to a solution

- Goal: A placement of 8 queens on the chess board in which no queen attacks another.

  ↓

  same row, column, or diagonal

- Initial state: no queen on the board.

Constraint satisfaction!

Knuth's Conjecture (1964)

Donald Knuth (Stanford)

"father of the analysis of algorithms"
ACM Turing Award (1974)
National Medal of Science (1979)

Any integer > 4 can be reached from 4 via a sequence of square root, floor, and factorial operations.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}} \right\rfloor = 5$$

- States: positive real numbers.
- Actions: square root, floor, or factorial operation.
- Action cost: 1.

* Photo from https://amturing.acm.org/byyear.cfm.

Knuth's Conjecture (1964)

Donald Knuth (Stanford)

"father of the analysis of algorithms"
ACM Turing Award (1974)
National Medal of Science (1979)

Any integer > 4 can be reached from 4 via a sequence of square root, floor, and factorial operations.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}} \right\rfloor = 5$$

- States: positive real numbers.
- Actions: square root, floor, or factorial operation.
- Action cost: 1.

* Photo from https://amturing.acm.org/byyear.cfm.

Real-World Problems

- Route finding (e.g., from Arad to Bucharest)
- Traveling salesman problem

Round trip visiting 13,509 US cities
with population size > 500 exactly once*

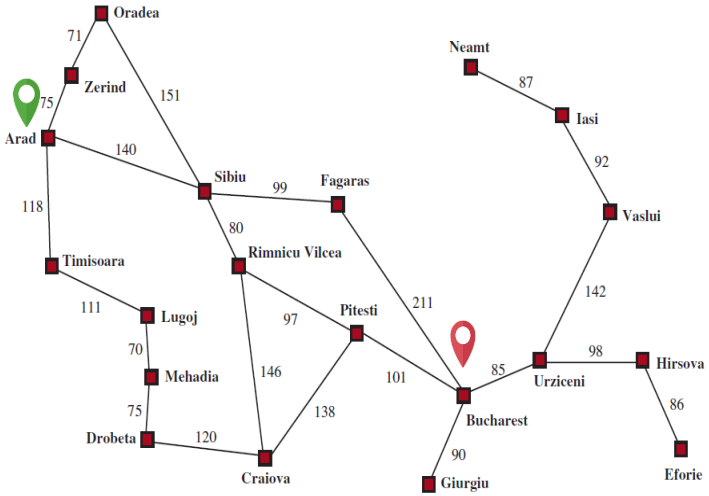
- VLSI layout

Positioning of millions of components and connections on a chip.

- Robot navigation
- Autonomous assembly sequencing (e.g., protein design)

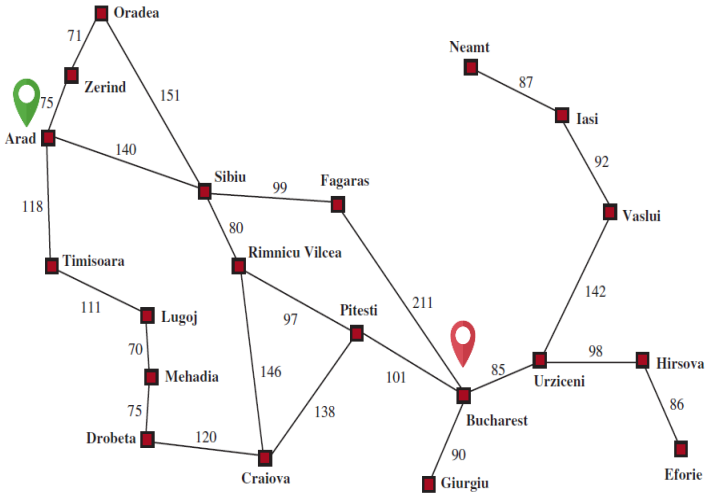
* Figure from http://www.crpc.rice.edu/CRPC/newsletters/sum98/news_tsp.html.

III. Tree Search

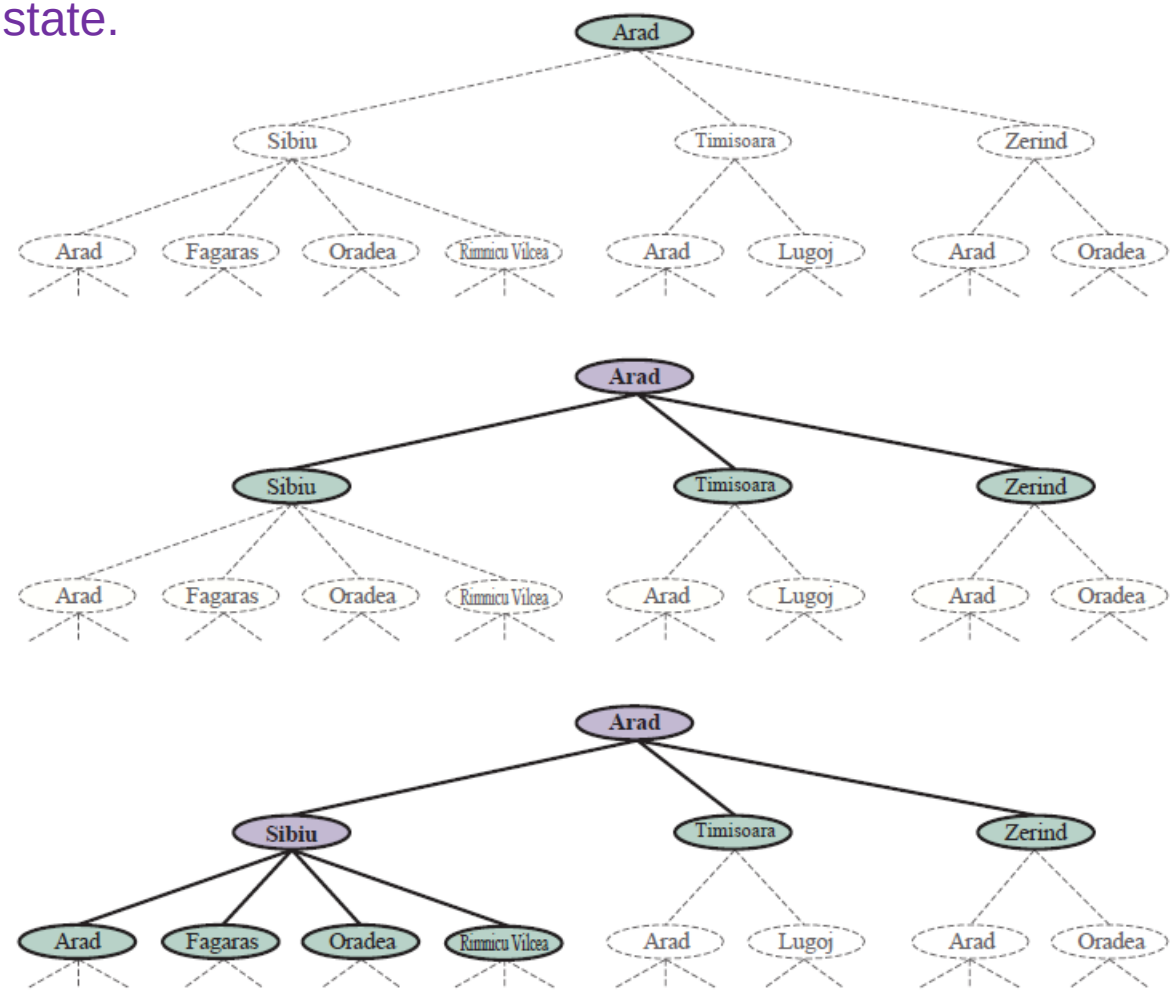


Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.

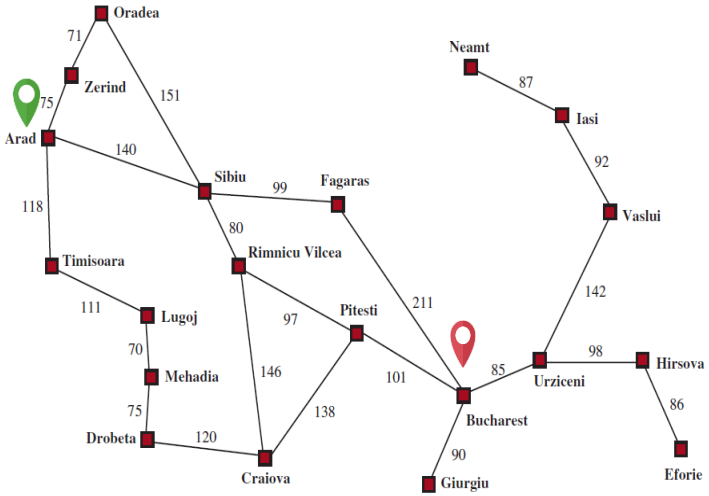
III. Tree Search



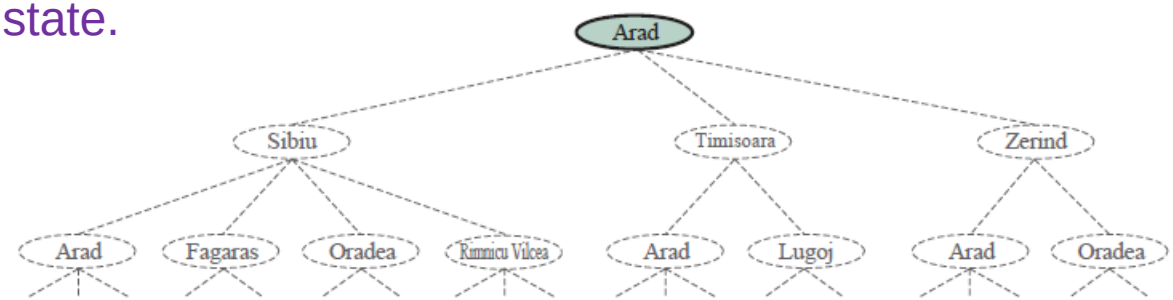
Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.



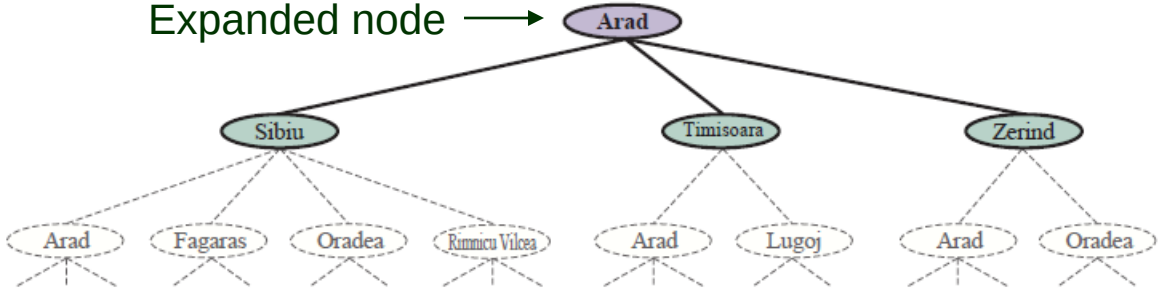
III. Tree Search

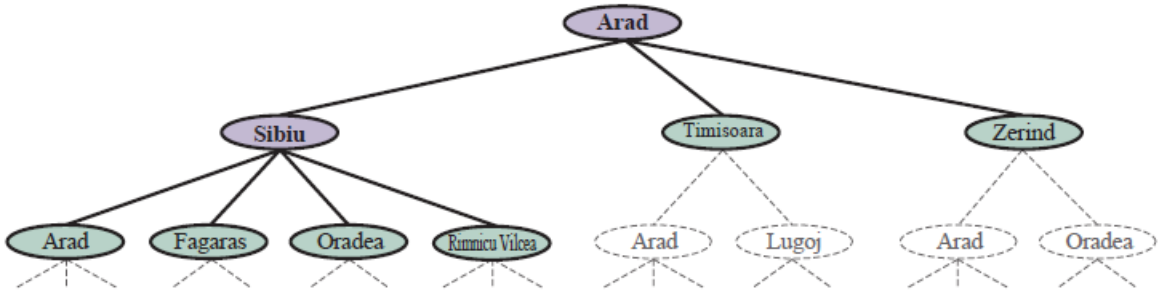


Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.

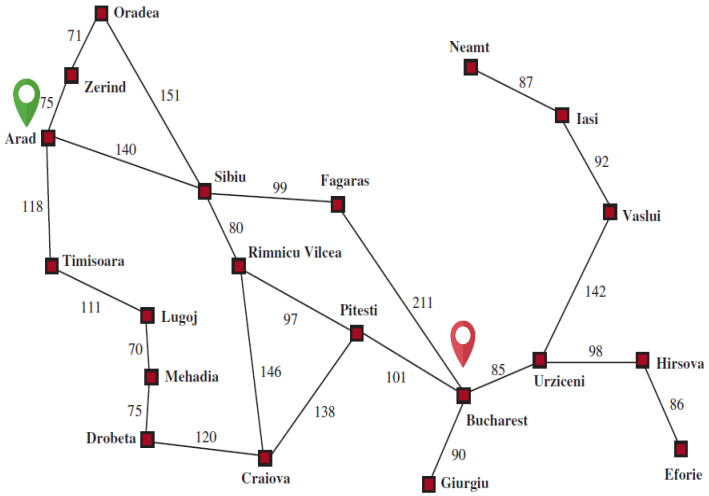


Expanded node →

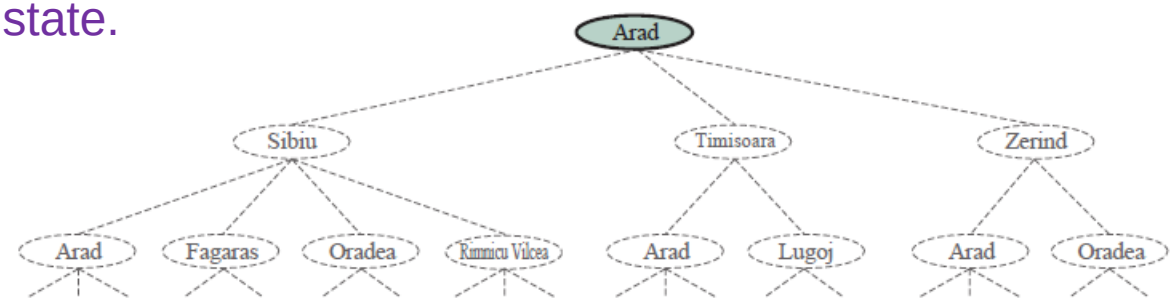




III. Tree Search

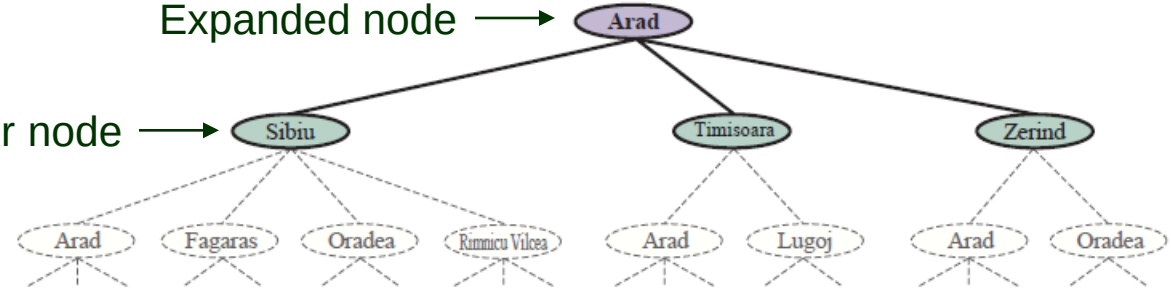


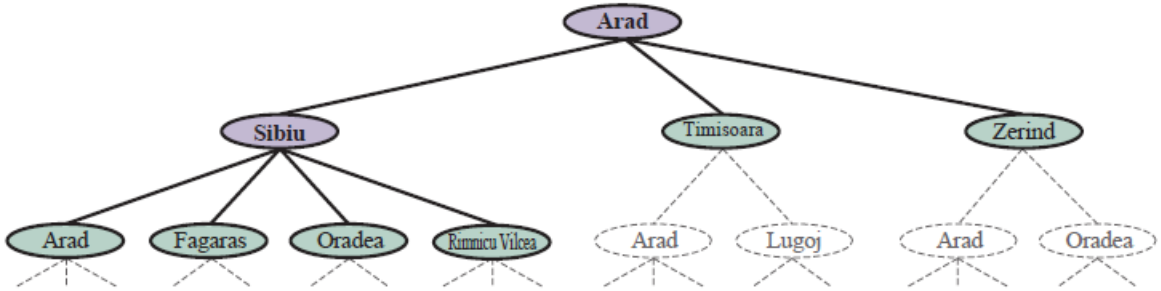
Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.



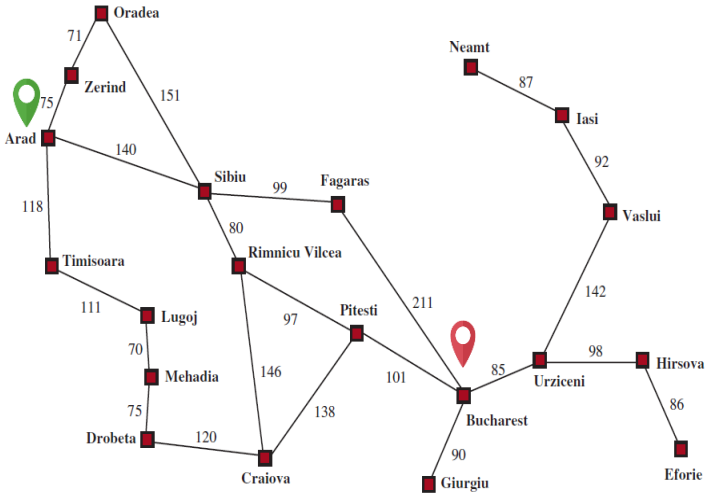
Expanded node →

Frontier node →

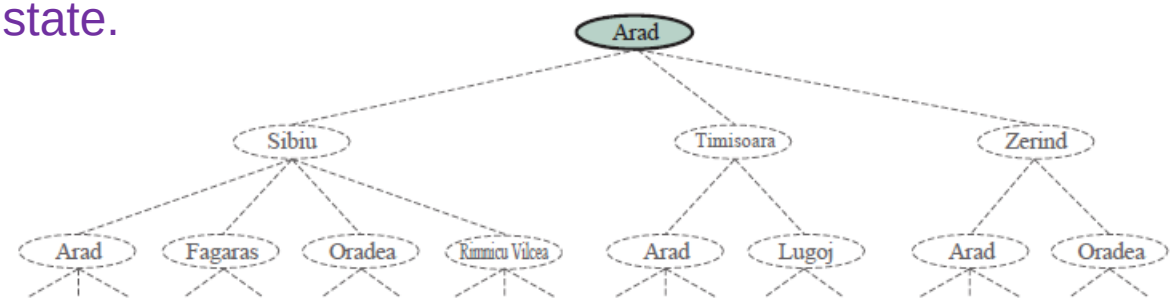




III. Tree Search

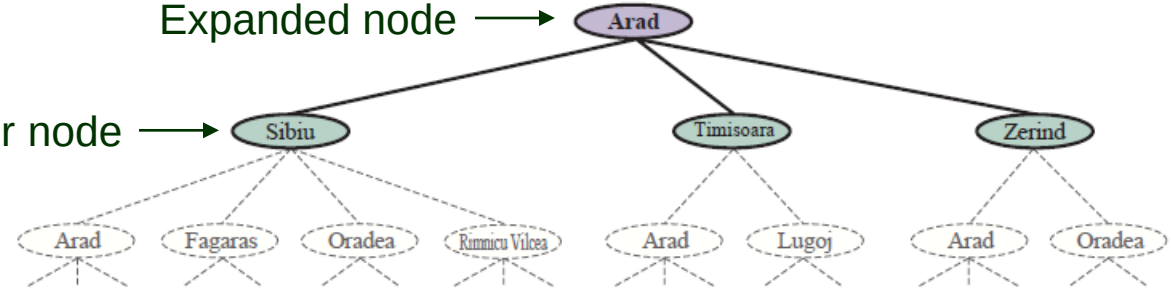


Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.

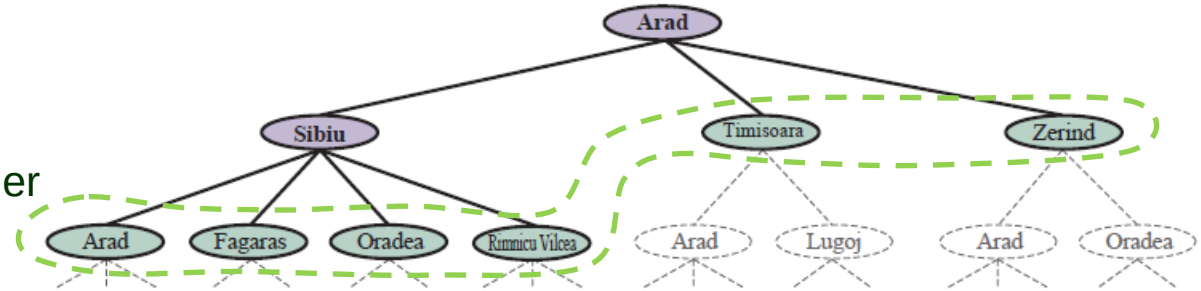


Expanded node →

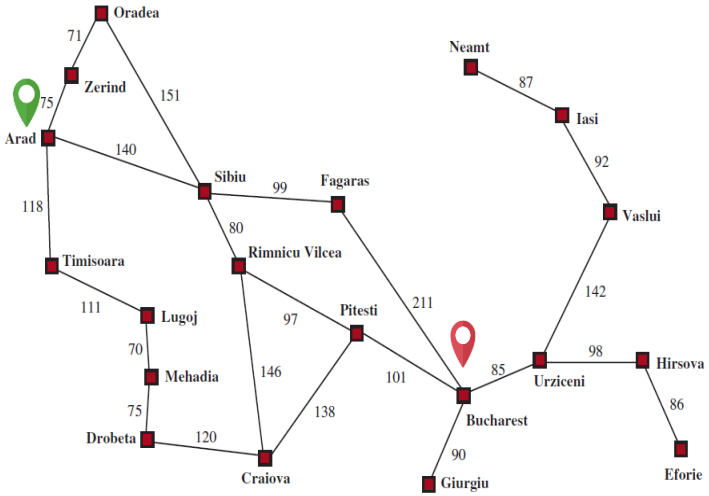
Frontier node →



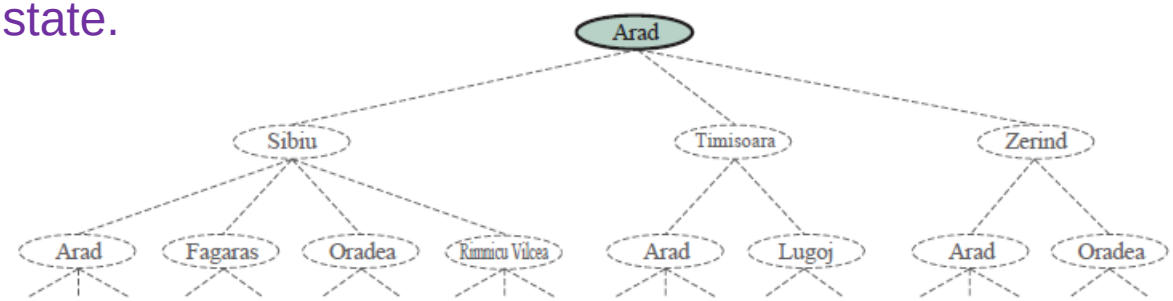
Frontier



III. Tree Search



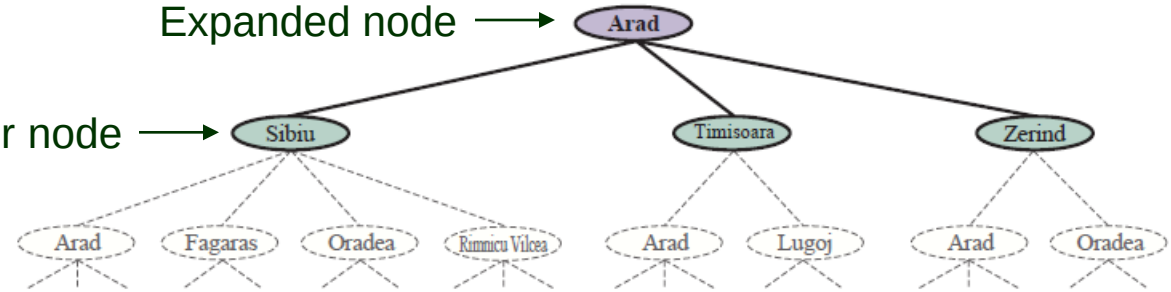
Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.



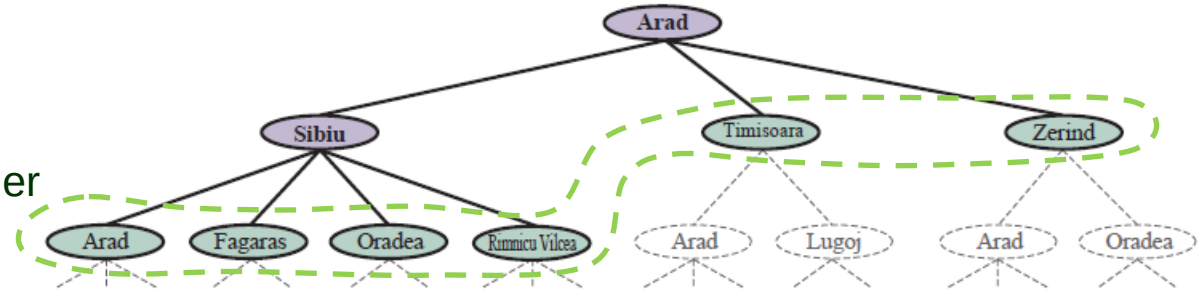
Expanded node →

Frontier node →

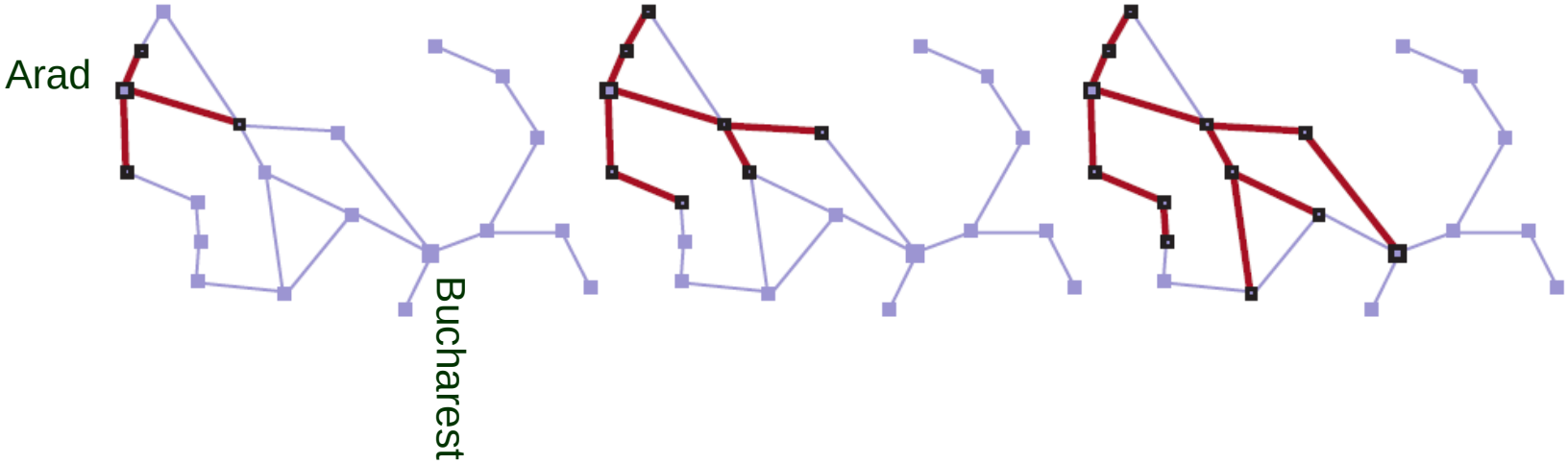
Node to be generated next →



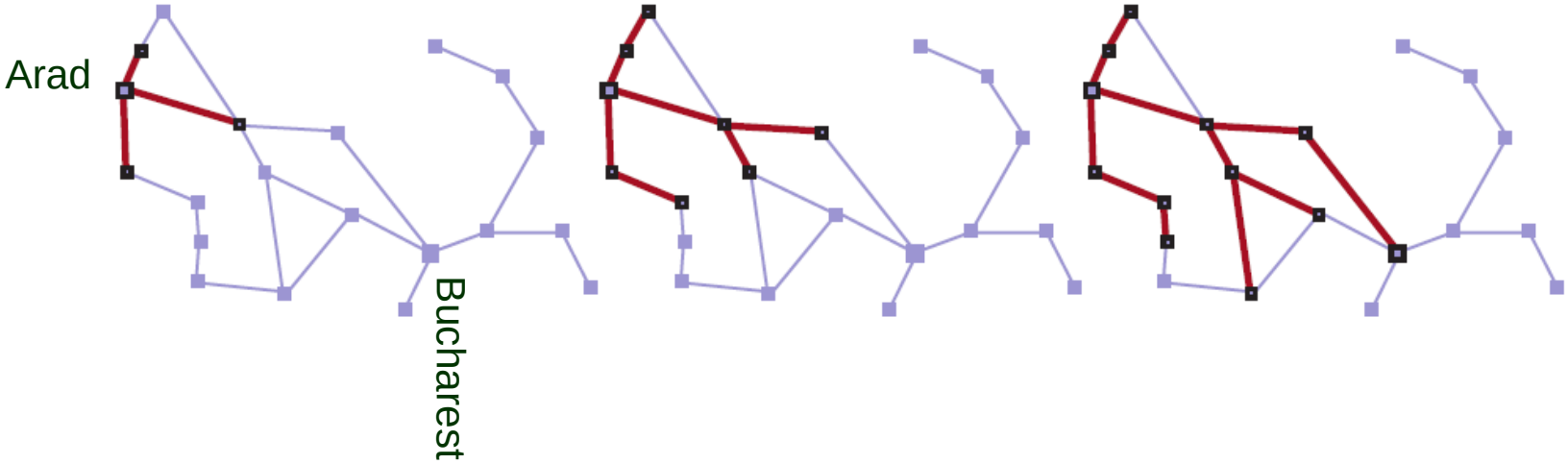
Frontier



Generated Search Trees

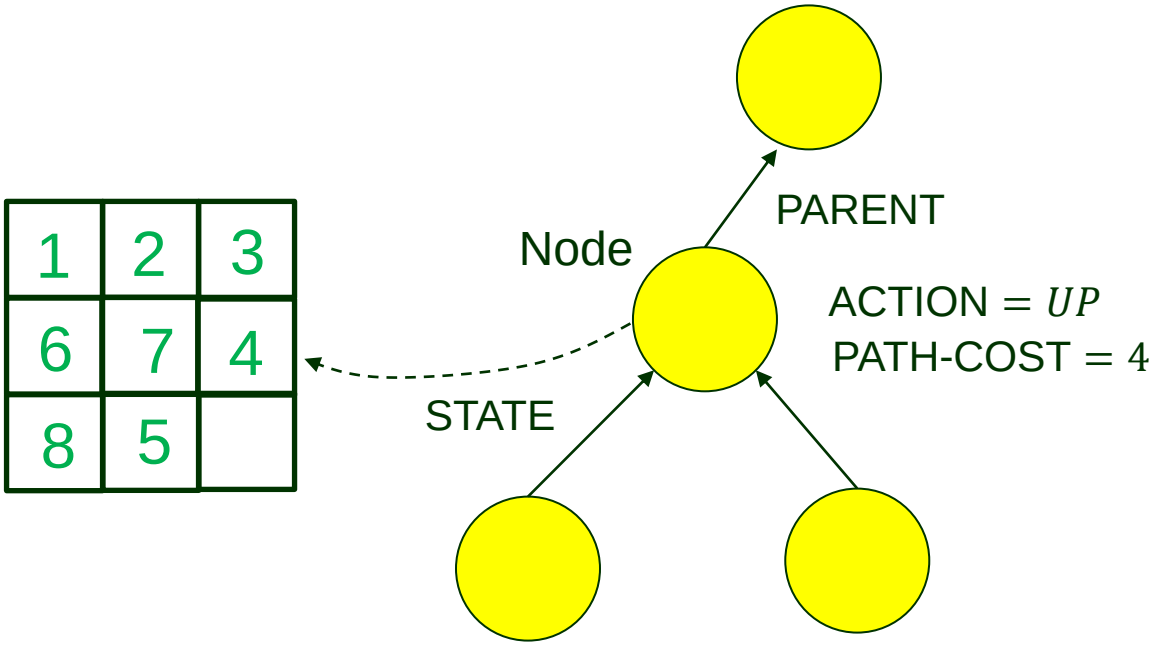


Generated Search Trees



Which node on the frontier to expand next?

State vs. Node



Best-First Search

Choose a node n which minimum value $f(n)$.

evaluation function

Best-First Search

Choose a node n which minimum value $f(n)$.

evaluation function

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element // states on the frontier
  reached ← a lookup table, with one entry with key problem.INITIAL and value node // states
  while not IS-EMPTY(frontier) do                                   // that have been reached
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Best-First Search

Choose a node n which minimum value $f(n)$.

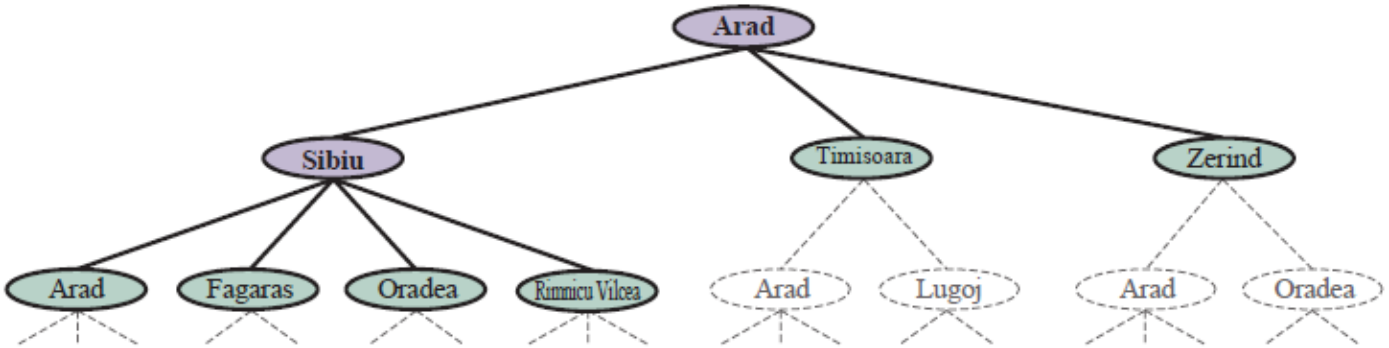
evaluation function

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element // states on the frontier
  reached ← a lookup table, with one entry with key problem.INITIAL and value node // states
  while not IS-EMPTY(frontier) do                                    // that have been reached
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

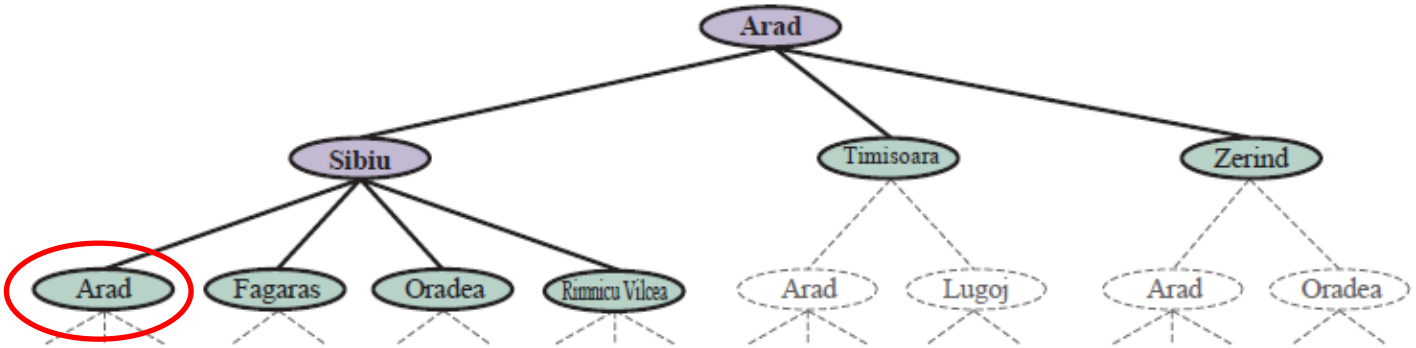
Can implement BFS and DFS.

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

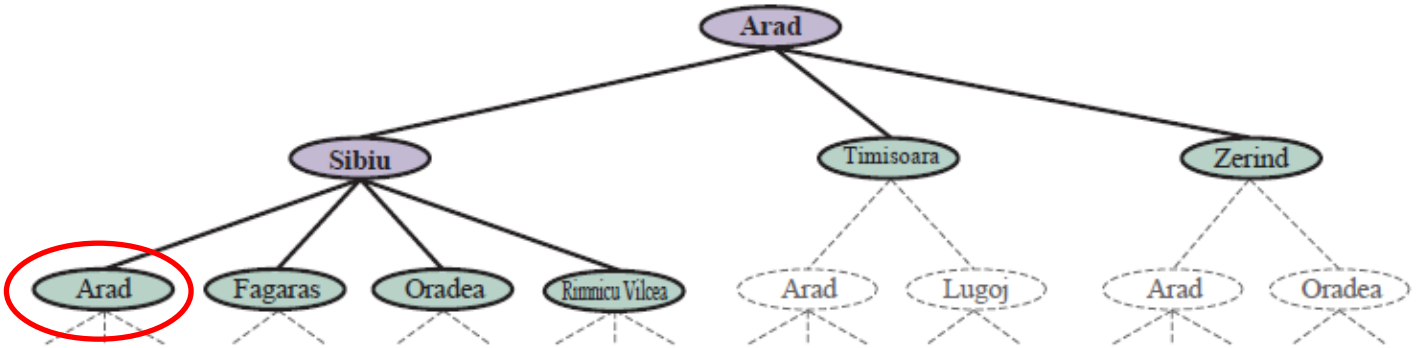
Repeated State

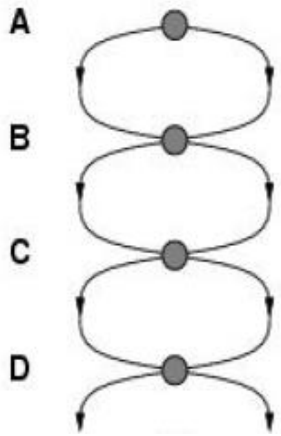


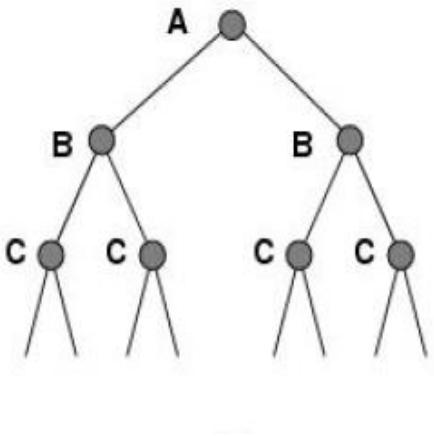
Repeated State



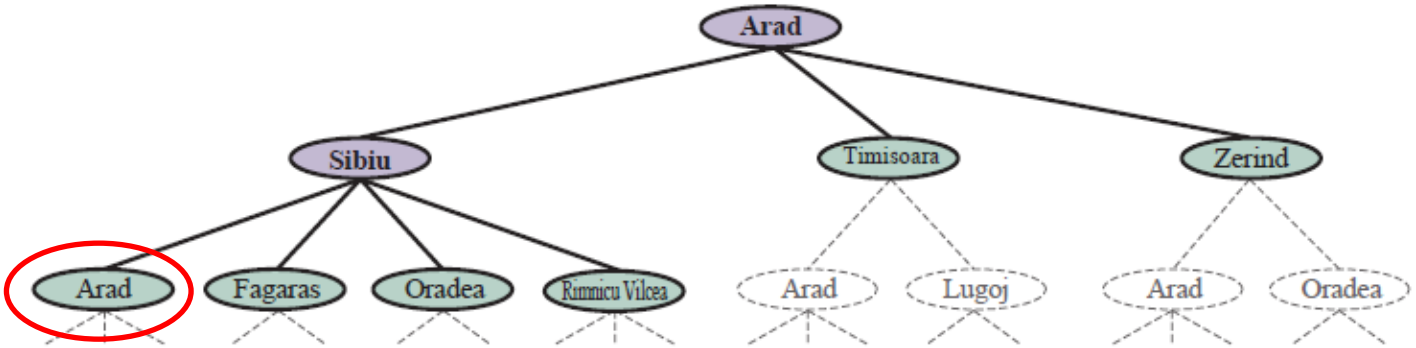
Repeated State

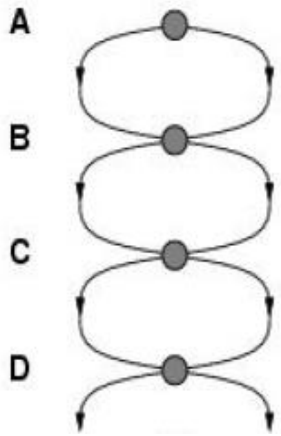


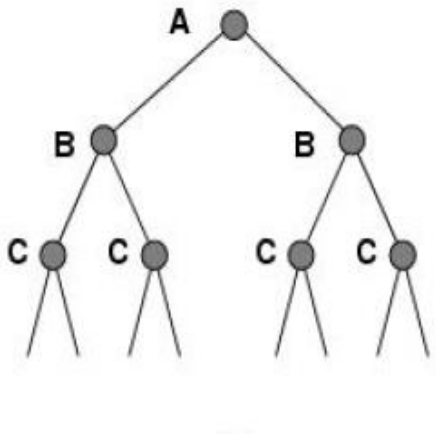




Repeated State

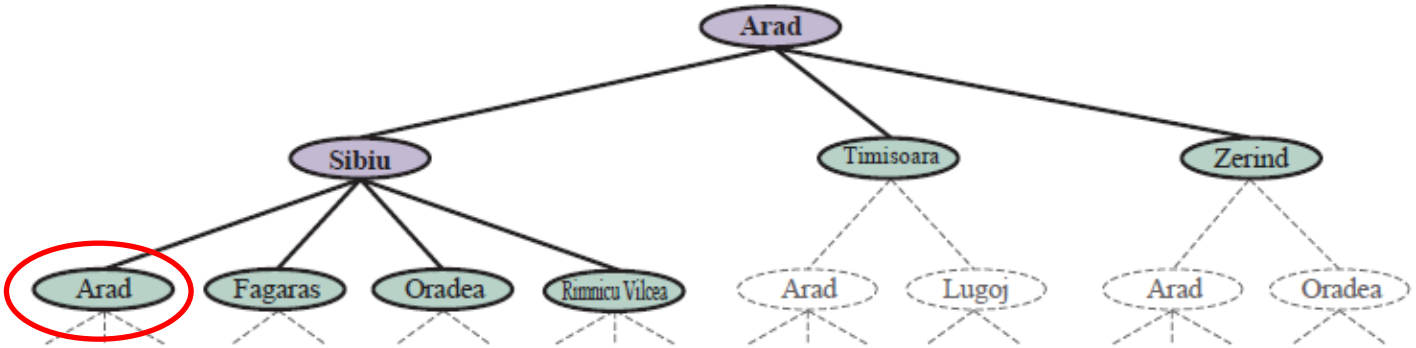


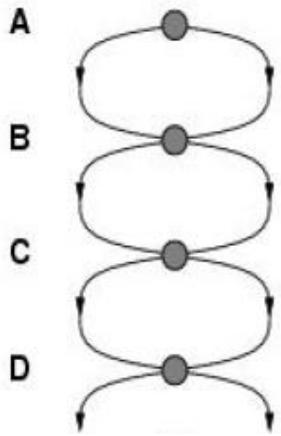


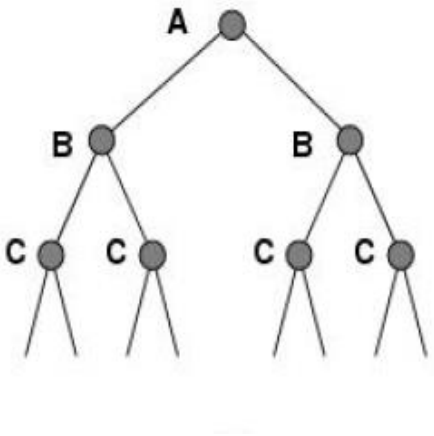


- Failure to detect repeated states can turn a solvable problem into an unsolvable one.

Repeated State







- ♠ Failure to detect repeated states can turn a solvable problem into an unsolvable one.
- ♦ Keep only the best path to each state.

Performance Measures

- Completeness: Is the algorithm guaranteed to find a solution whenever one exists, and to report failure otherwise?

The state space may be infinite!

- Cost optimality: Does it find a solution with the lowest path cost of all solutions?

- Time complexity: Physical time or the number of states and actions.

- Space complexity: Memory needed for the search.

$$|V| + |E|?$$

Depth and Branching Factor

- The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

Depth and Branching Factor

- ♣ The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

- But, often *implicit* graph representation of a state space in AI:

Depth and Branching Factor

- ♣ The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

- But, often *implicit* graph representation of a state space in AI:
  - ♦ the initial state
  - ♦ actions
  - ♦ a transition model

Depth and Branching Factor

- The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

- But, often *implicit* graph representation of a state space in AI:
  - the initial state
  - actions
  - a transition model

- Accordingly, complexity is measured in terms of
  - d: *depth* (number of actions in the optimal solution)

Depth and Branching Factor

- The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

- But, often *implicit* graph representation of a state space in AI:
  - the initial state
  - actions
  - a transition model

- Accordingly, complexity is measured in terms of
  - d: *depth* (number of actions in the optimal solution)
  - m: *maximum number of actions* in any path

Depth and Branching Factor

- The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

- But, often *implicit* graph representation of a state space in AI:
  - the initial state
  - actions
  - a transition model

- Accordingly, complexity is measured in terms of
  - d: *depth* (number of actions in the optimal solution)
  - m: *maximum number of actions* in any path
  - b: *branching factor* (number of successors of a node).