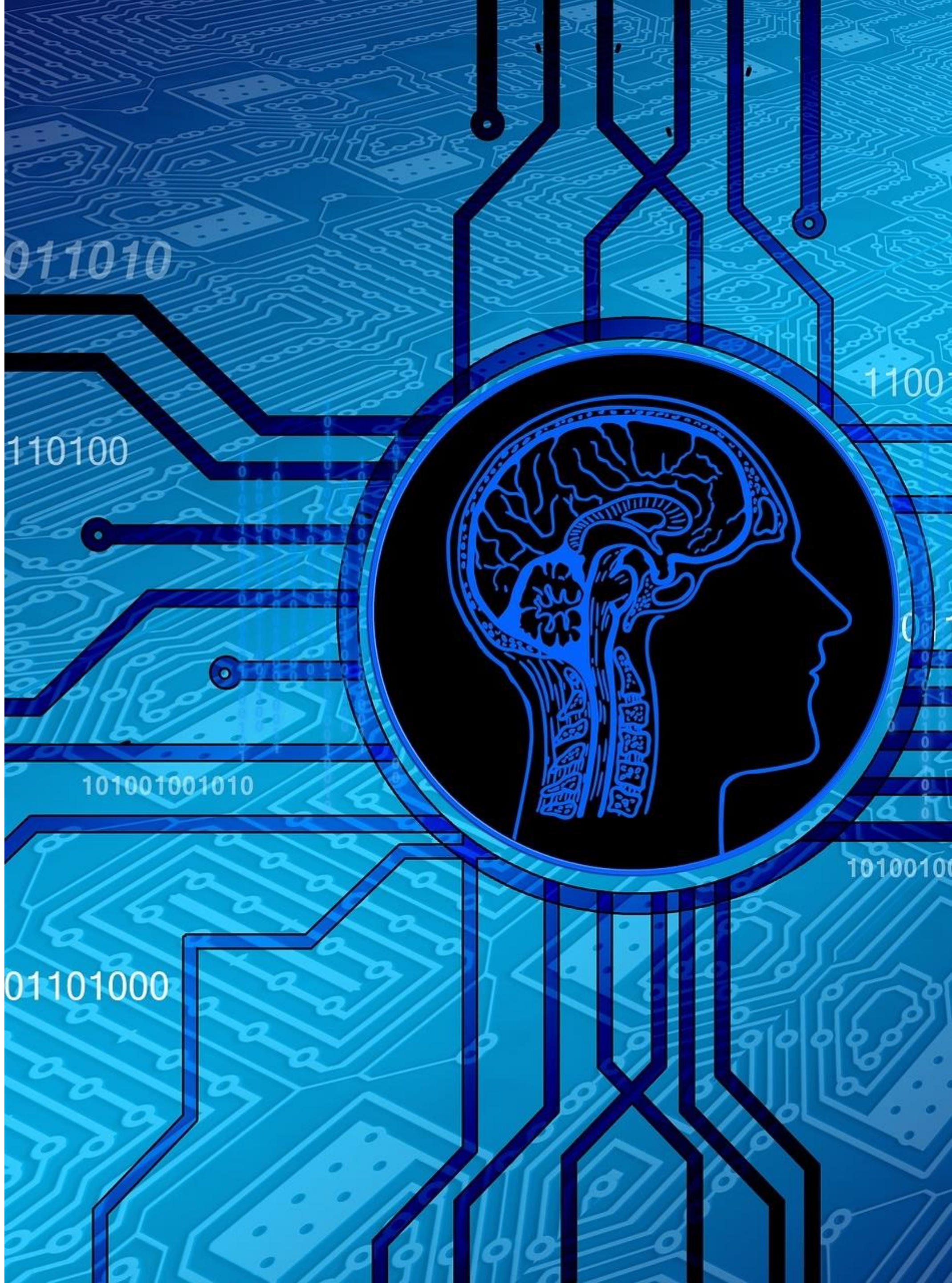
Problem Solving and Search

Informatics 2D: Reasoning and Agents

Lecture 2

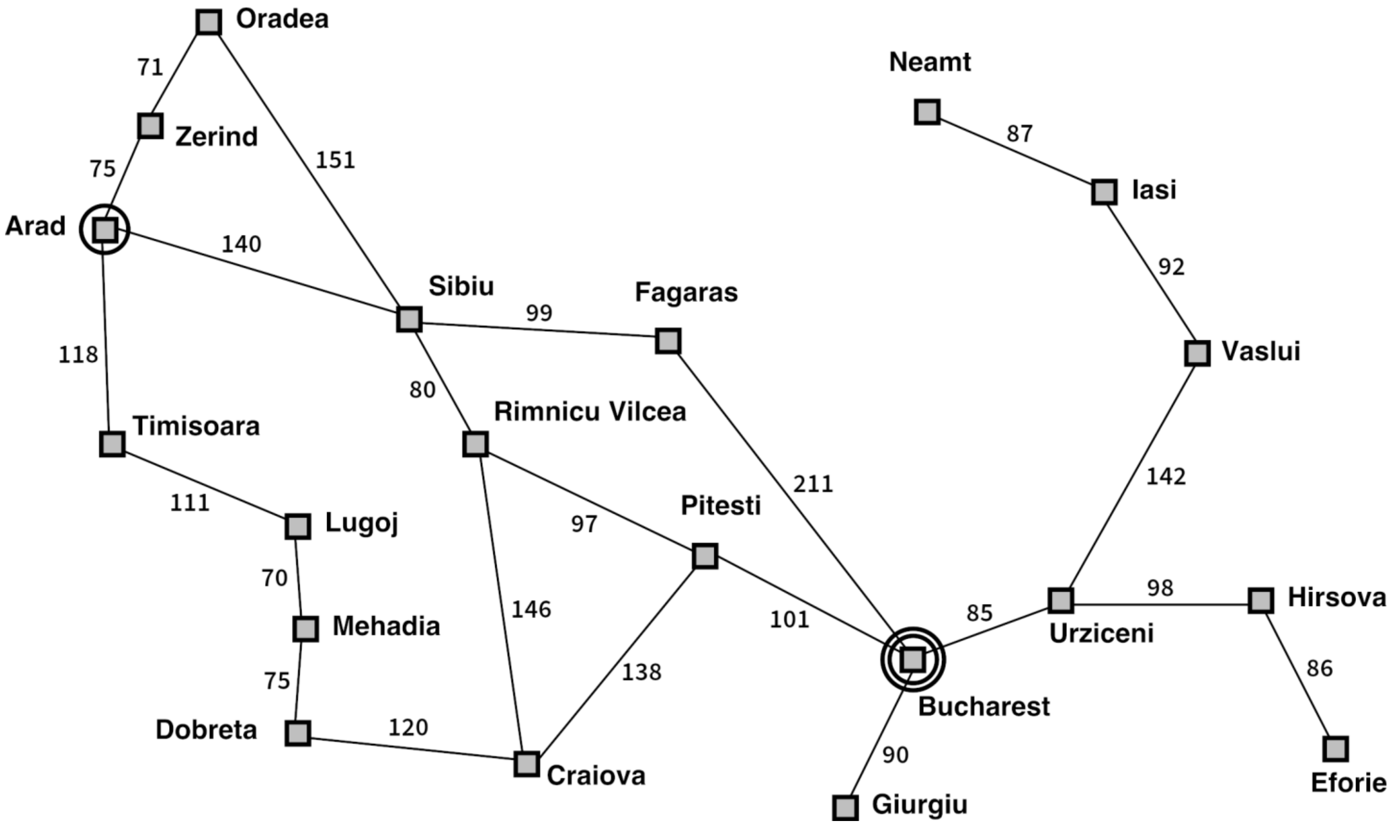


Problem-solving Agents

Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    persistent:   seq, an action sequence, initially empty
                  state, some description of the current world state
                  goal, a goal, initially null
                  problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Example: Romania



On holiday in Romania.

Currently in **Arad**.

Flight leaves tomorrow from **Bucharest**.

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

Problem types

Deterministic, fully observable → single-state problem

- Agent knows exactly which state it will be in; solution is a sequence

Non-observable → sensorless problem (conformant problem)

- Agent may have no idea where it is; solution is a sequence

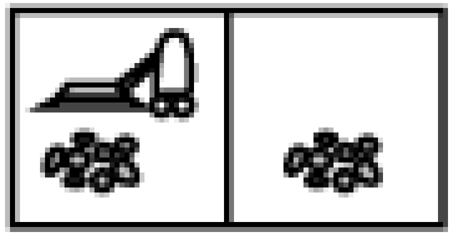
Nondeterministic and/or partially observable → contingency problem

- percepts provide new information about current state
- often interleave search, execution

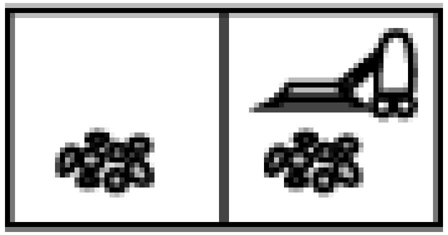
Unknown state space → exploration problem

Example: vacuum world

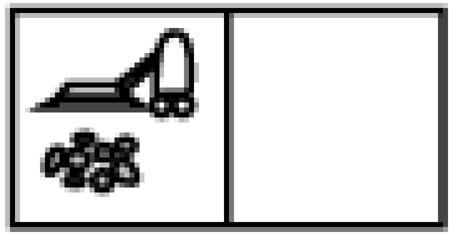
1



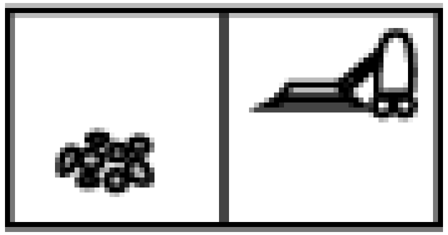
2



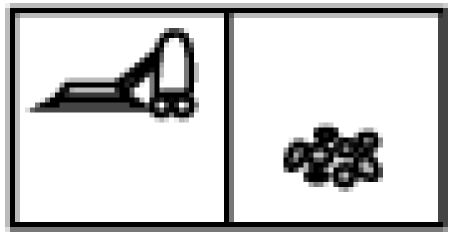
3



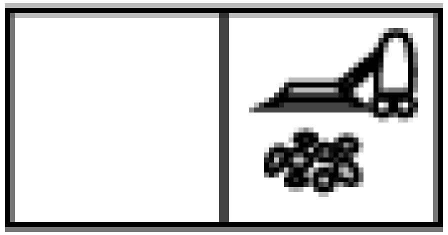
4



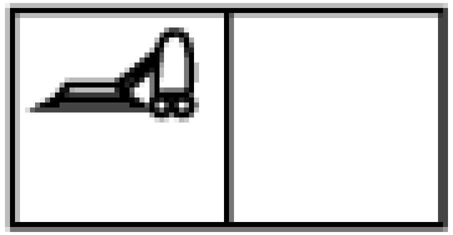
5



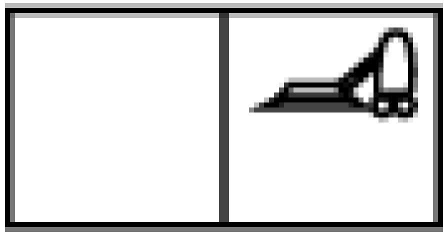
6



7



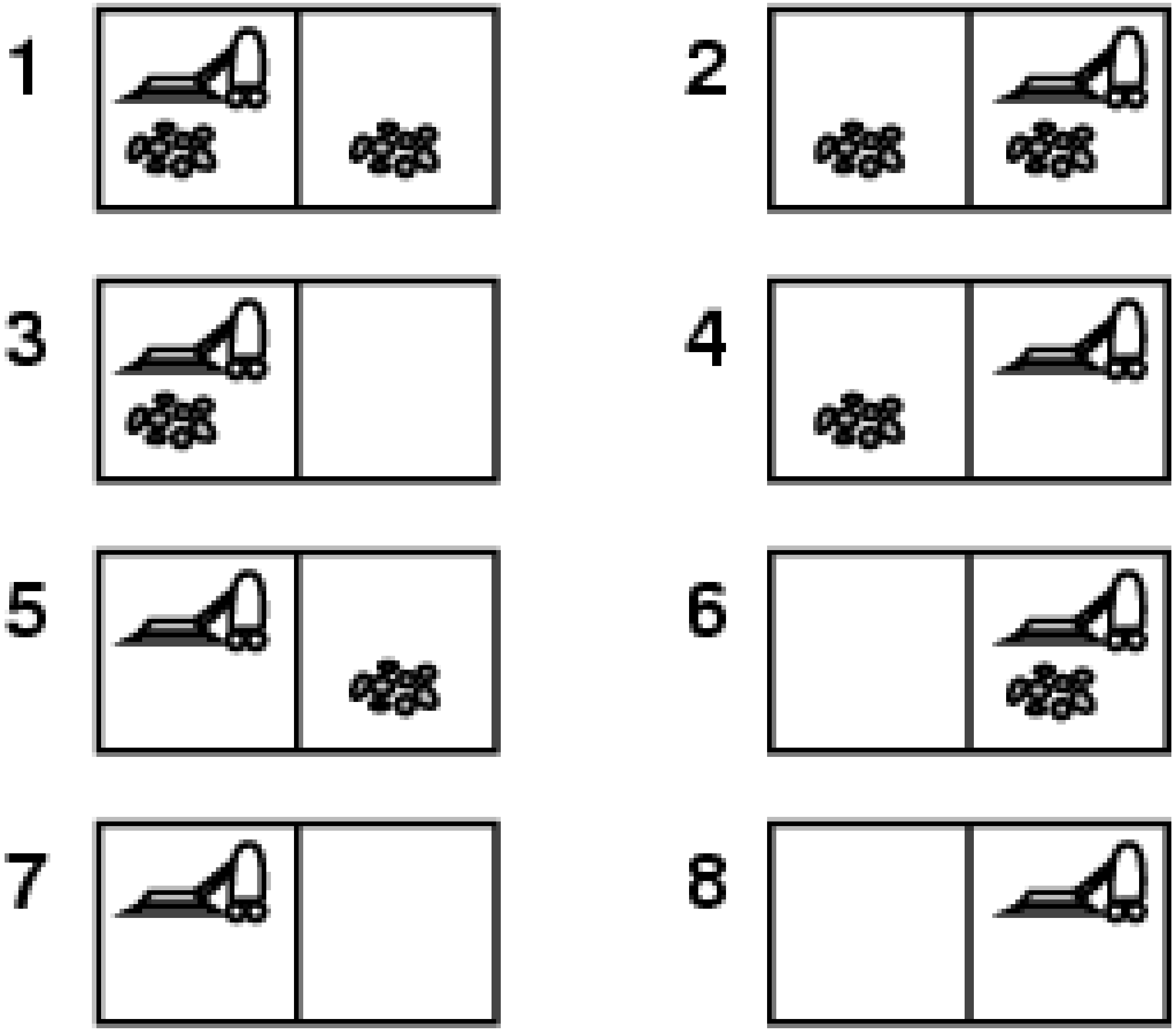
8



Single-state:

Start in 5

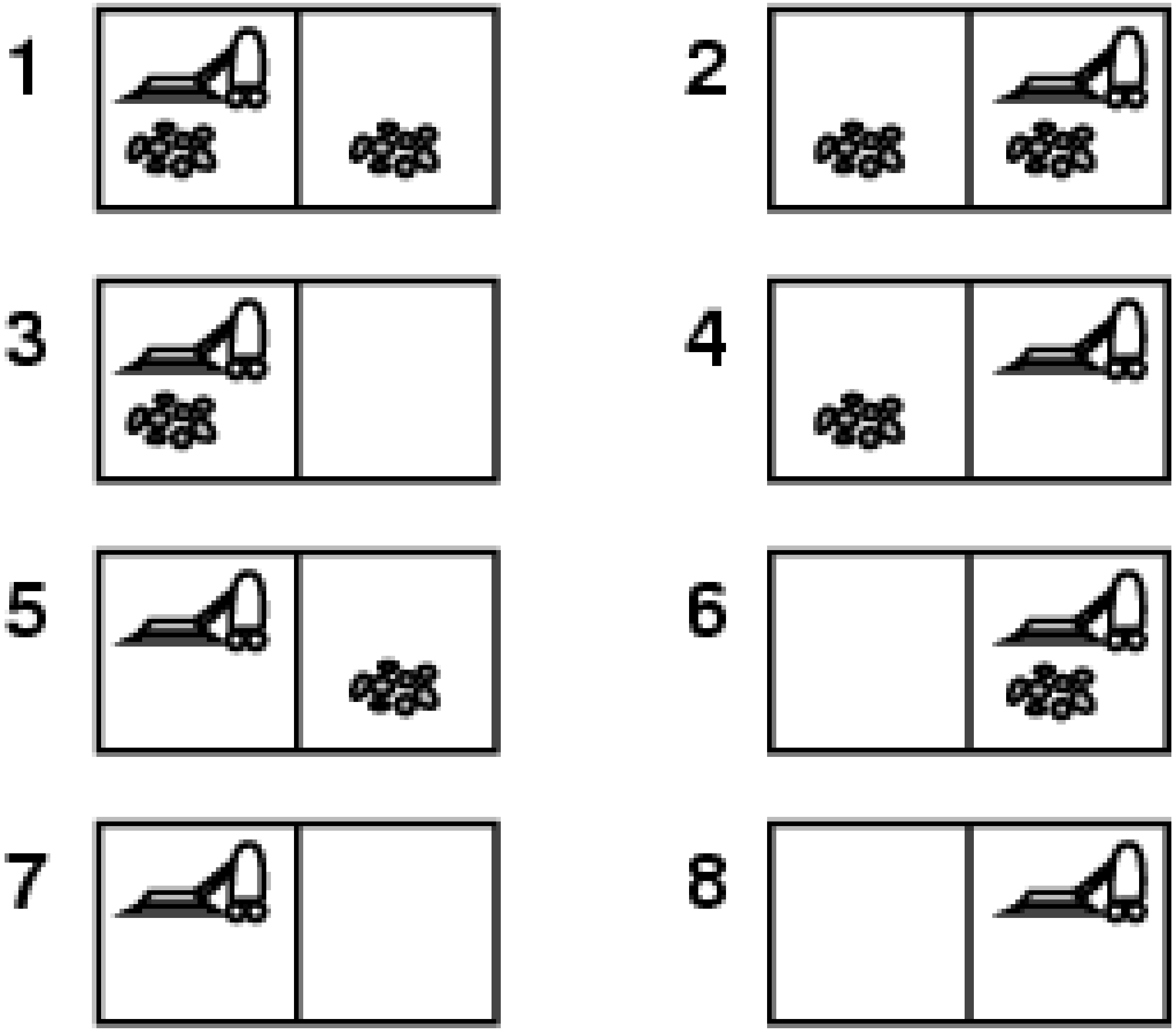
Solution?



Example: vacuum world

Single-state:
Start in 5
Solution?
[*Right*, *Suck*]

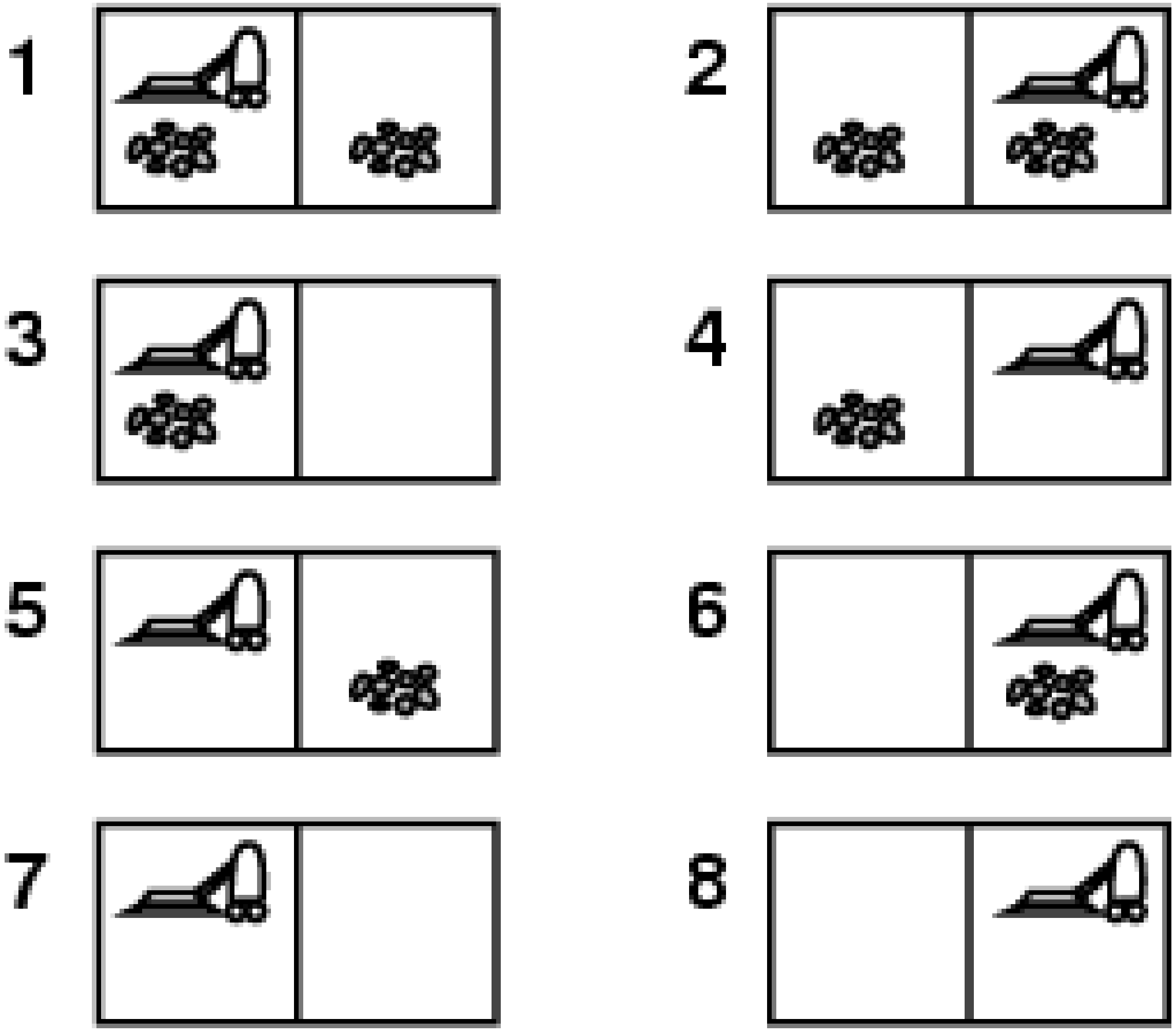
Sensorless:
Start in {*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*}
e.g. *Right* goes to {*2*,*4*,*6*,*8*}
Solution?



Example: vacuum world

Single-state:
Start in 5
Solution?
[*Right*, *Suck*]

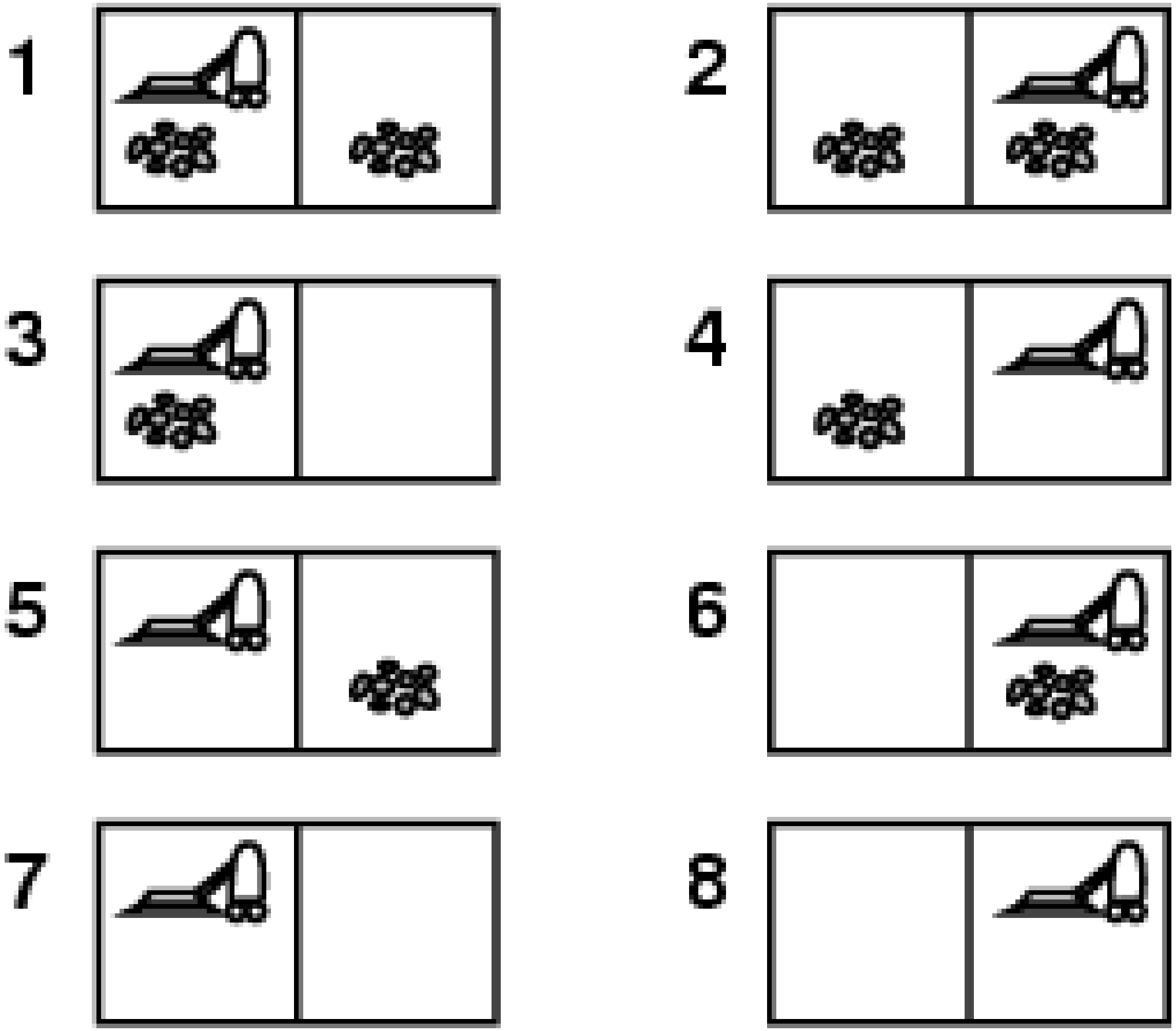
Sensorless:
Start in {*1,2,3,4,5,6,7,8*}
e.g. *Right* goes to {*2,4,6,8*}
Solution?
[*Right*, *Suck*, *Left*, *Suck*]



Example: vacuum world

Contingency:

- Nondeterministic: *Suck* may dirty a clean carpet
- Partially observable: can only see dirt at current location.
- Percept: [*Left*, *Clean*]
  i.e., start in 5 or 7
  Solution?



Example: vacuum world

Contingency:

- Nondeterministic: *Suck* may dirty a clean carpet
- Partially observable: can only see dirt at current location.
- Percept: [*Left, Clean*]
  i.e., start in 5 or 7
  Solution?
  [*Right, **if** dirt **then** Suck*]

Problem Formulation

Single-state problem formulation

Initial State

- e.g. “in Arad”

Actions or Successor function

- $S(x)$ = set of action-state pairs
- e.g. S(Arad) = {<Arad → Zerind, Zerind>, … }

Goal test

- explicit e.g. x = “in Bucharest"
- implicit e.g. *Checkmate(x)*

Path cost (additive)

- e.g. sum of distances, number of actions executed, etc.
- $c(x,a,y)$ is the step cost of taking action a in state x to reach state y, assumed to be ≥ 0

Single-state problem formulation

Initial State

- e.g. "in Arad"

Actions or Successor function

- *S*(*x*) = set of action-state pairs
- e.g. *S*(Arad) = {<Arad → Zerind, Zerind>, … }

> A ***solution*** is a sequence of actions leading from the initial state to a goal state, i.e. a state that succeeds the goal test.

Goal test

- explicit e.g. x = "in Bucharest"
- implicit e.g. *Checkmate(x)*

Path cost (additive)

- e.g. sum of distances, number of actions executed, etc.
- $c(x,a,y)$ is the step cost of taking action a in state x to reach state y, assumed to be ≥ 0

Selecting a state space

Real world is absurdly complex

→ state space must be abstracted for problem solving

(Abstract) state = set of real states

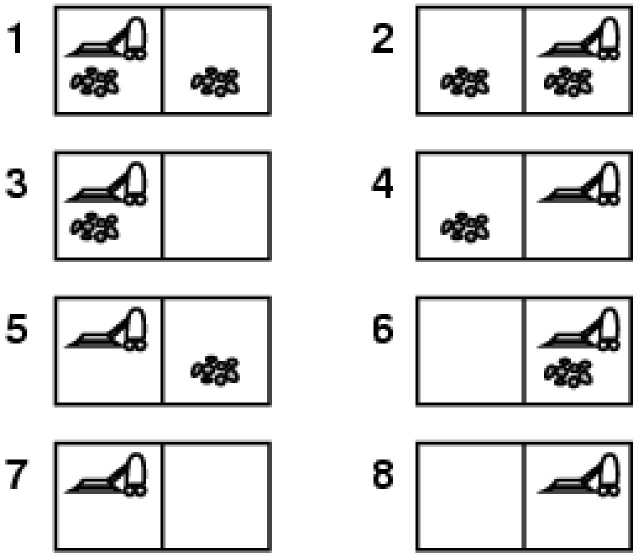
(Abstract) action = complex combination of real actions

- e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed 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



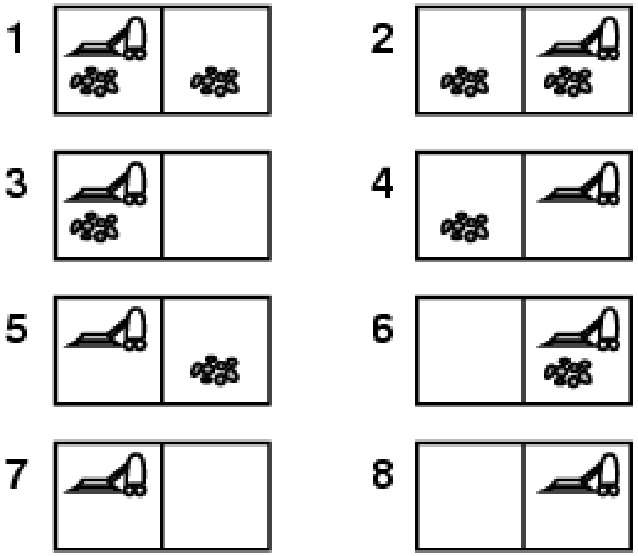
States

Actions

Goal test

Path cost (additive)

Example: Vacuum world



States

- Pair of dirt and robot locations

Actions

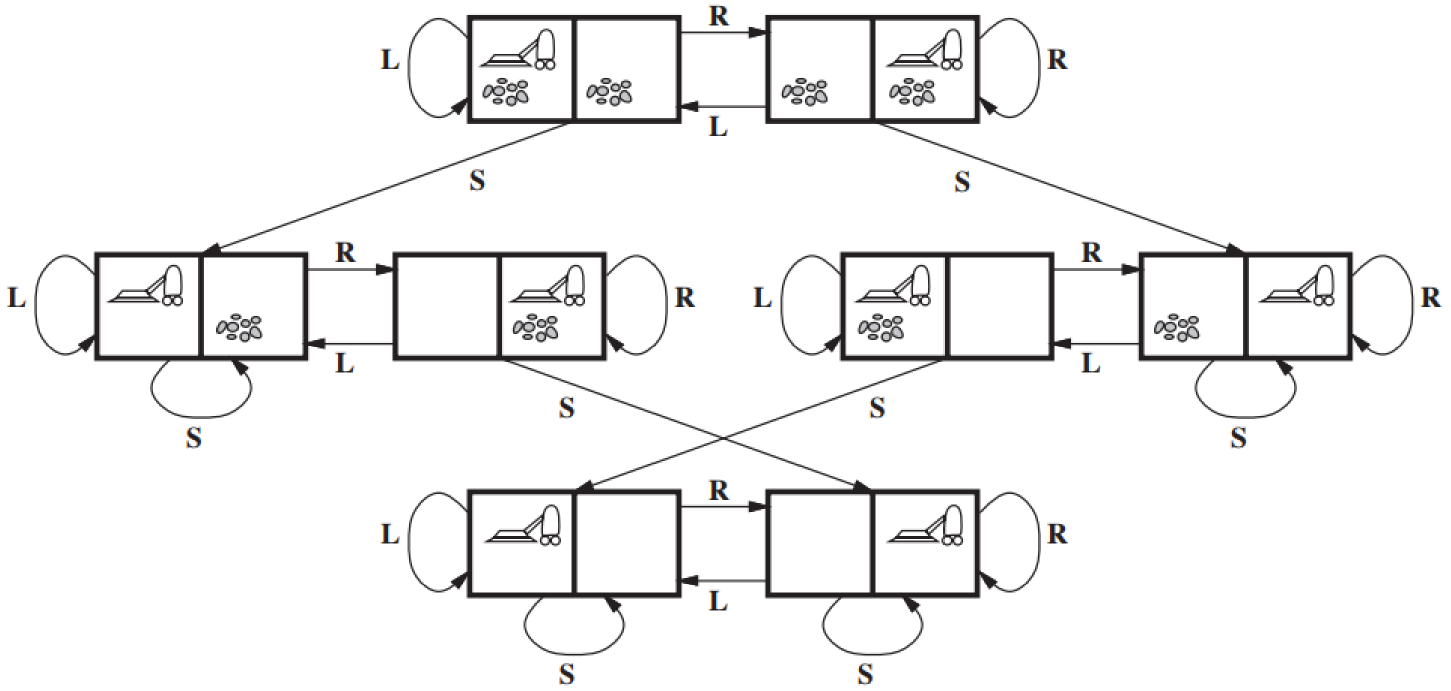
- *Left, Right, Suck*

Goal test

- No dirt at any location

Path cost (additive)

- 1 per action



Example: Vacuum world

States

- Pair of dirt and robot locations

Actions

- *Left, Right, Suck*

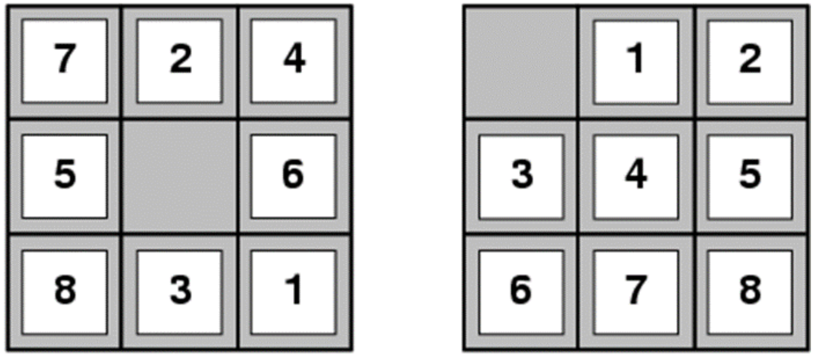
Goal test

- No dirt at any location

Path cost (additive)

- 1 per action

Example: 8-puzzle



7	2	4
5		6
8	3	1

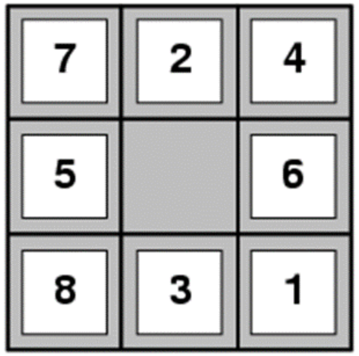
Start State

	1	2
3	4	5
6	7	8

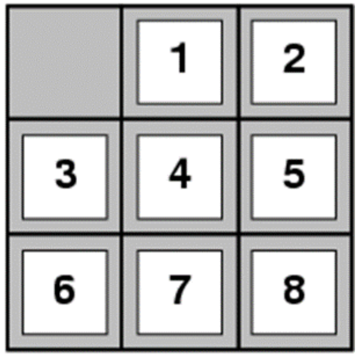
Goal State

- States
- Actions
- Goal test
- Path cost (additive)

Example: 8-puzzle



Start State



Goal State

States

- Integer location of tiles

Actions

- Move blank *left, right, up, down*

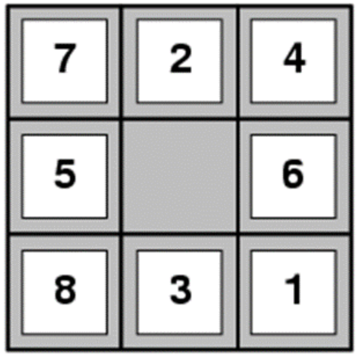
Goal test

- = Goal state (given)

Path cost (additive)

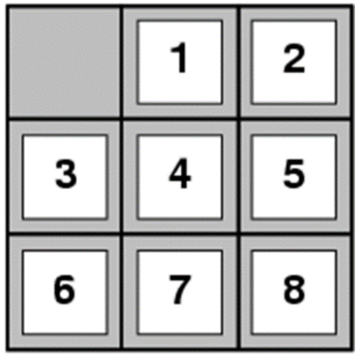
- 1 per move

Example: 8-puzzle



7	2	4
5		6
8	3	1

Start State



	1	2
3	4	5
6	7	8

Goal State

NP-
Hard

States

- Integer location of tiles

Actions

- Move blank *left, right, up, down*

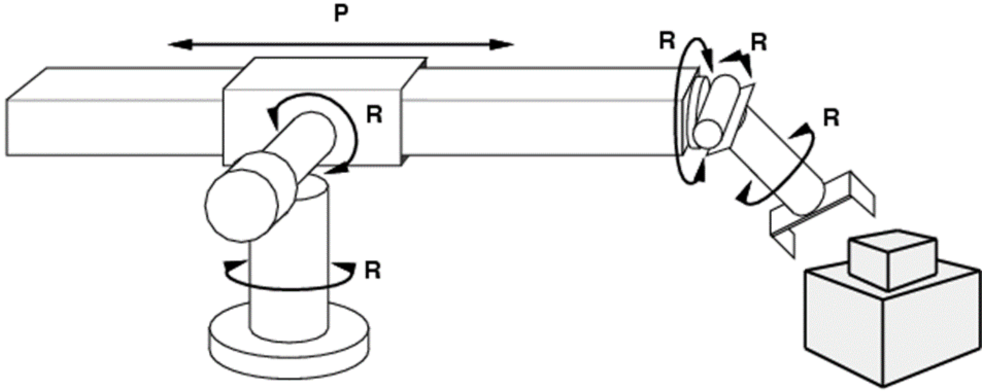
Goal test

- = Goal state (given)

Path cost (additive)

- 1 per move

Example: Robotic assembly



States

- Real-valued coordinates of robot joint angles
- Parts of the object to be assembled

Actions

- Continuous motions of robot joints

Goal test

- = complete assembly

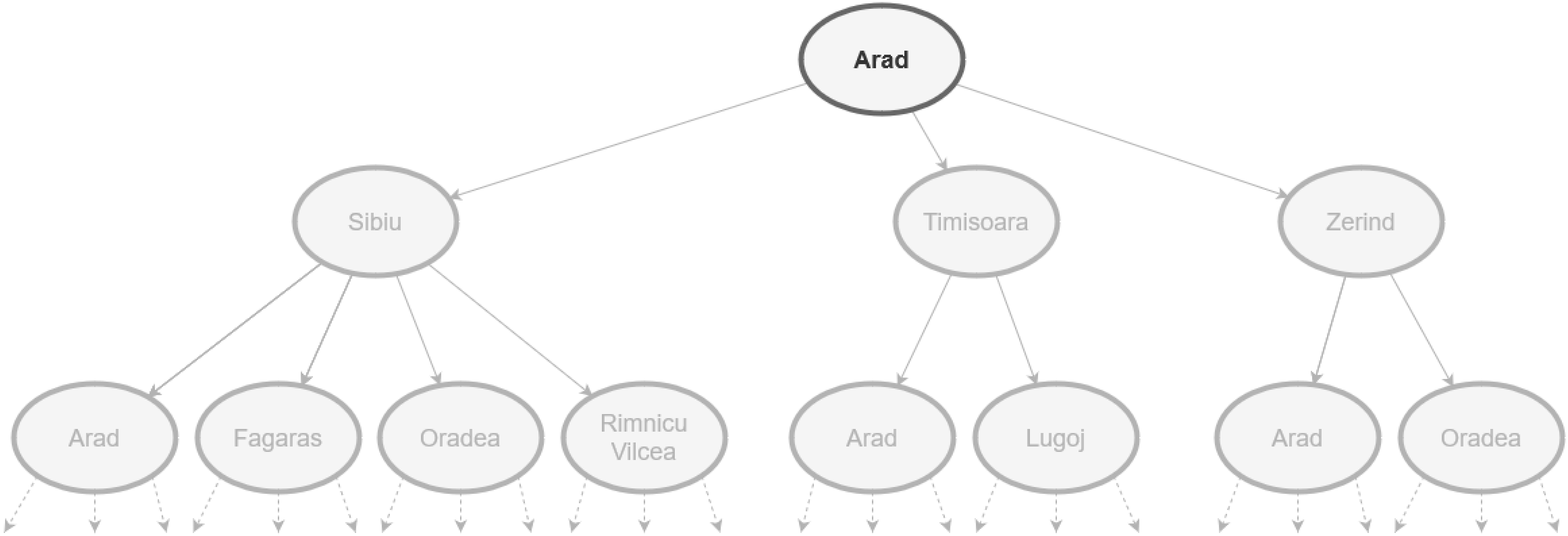
Path cost (additive)

- Time to execute

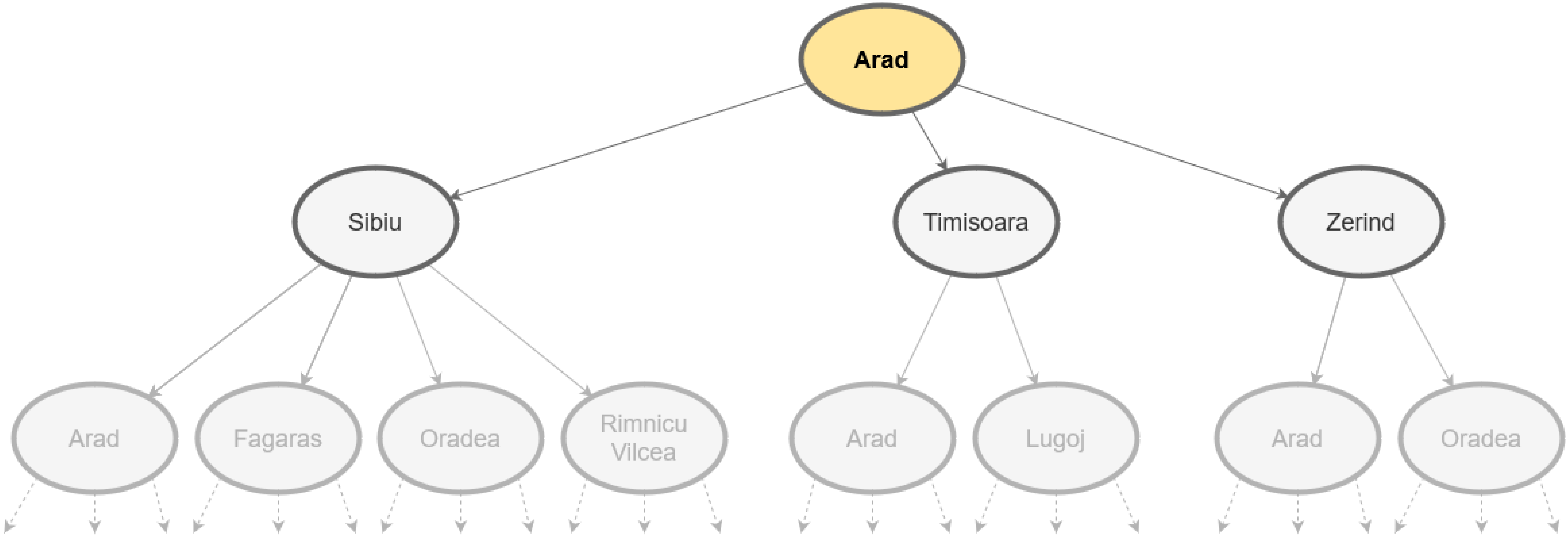
Searching for Solutions

Tree search algorithms

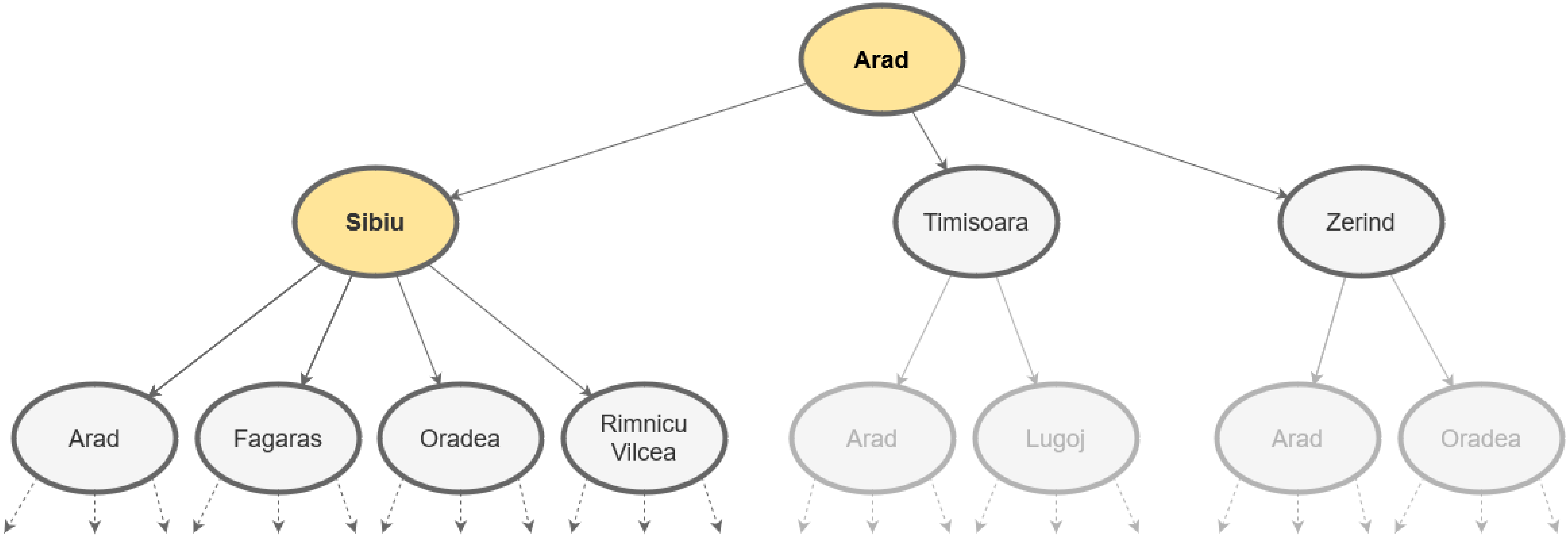
```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```



Tree search example

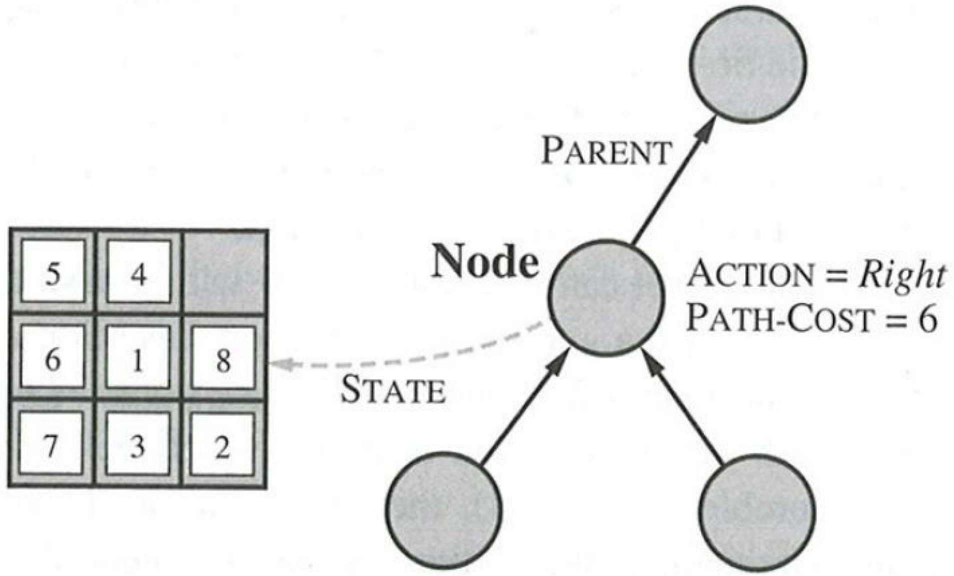


Tree search example



Tree search example

Implementation: states vs. nodes



A state is a (representation of) a physical configuration

A node is a book-keeping data structure constituting part of a **search tree**; includes *state, parent node, action, path cost*

Using these it is easy to compute the components for a child node.
(The CHILD–NODE function)

Implementation: general tree search

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

```
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT(parent.STATE, action),
        PARENT = parent, ACTION = action,
        PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.

Why?

- Formulating problems in a way that a computer can understand.
- Breaking down the problem and its parameters.
- Clarifying the possible actions and assumptions about them.
- Creating structures where we can methodically and systematically search for solutions.