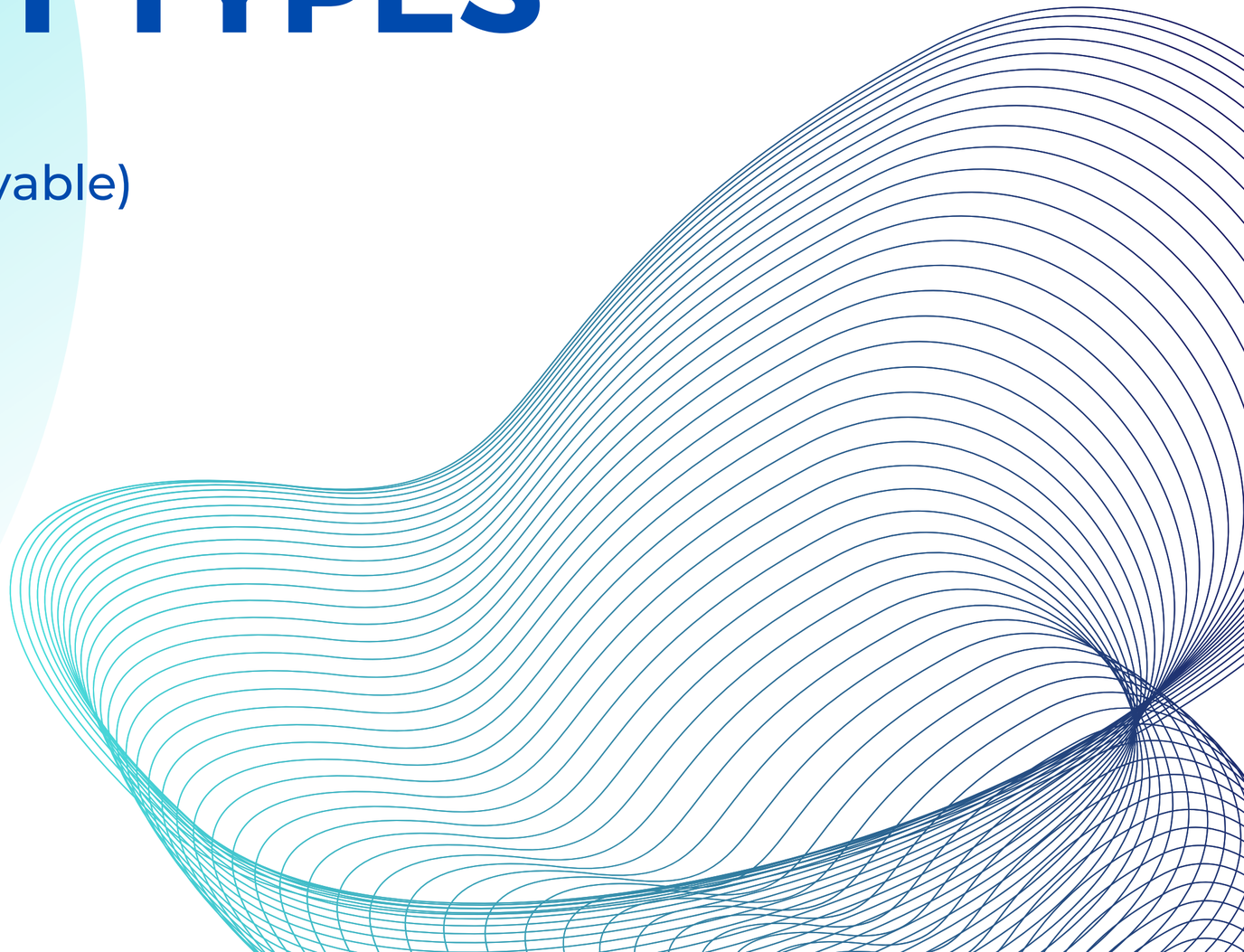
INTRODUCTION TO ARTIFICIAL INTELLIGENCE

PGS. TS. Nguyễn Thanh Bình

ENVIRONMENT TYPES

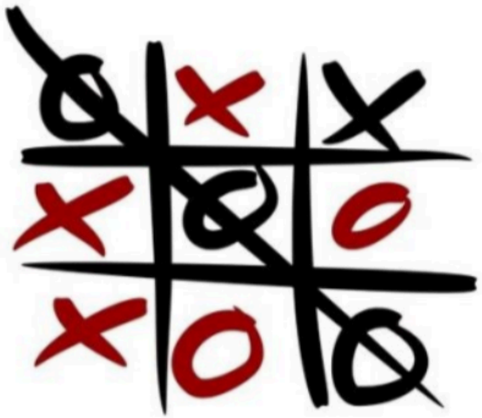
- Fully observable (vs. partially observable)
- Deterministic (vs. stochastic)
- Episodic (vs. sequential)
- Static (vs. dynamic)
- Discrete (vs. continuous)
- Single agent (vs. multi-agent)



ENVIRONMENT TYPES

- **Fully observable (**vs. **partially observable):** an agent's sensors give it access to the complete state of the environment at each point in time.

Fully Observable Task Environment
Tic Tac Toe/Noughts & Crosses/X's and O's



Partially Observable Task Environment
Cards Game /Hidden Hand Poker

ENVIRONMENT TYPES

Deterministic (vs. **stochastic)**: the next state of the environment is completely determined by the current state and the action executed by the agent. *(If the environment is deterministic except for the actions of other agents, the environment is strategic)*. The stochastic environment is random in nature which is not unique and cannot be completely determined by the agent.

Examples:

- Chess : there would be only a few possible moves for a coin at the current state and these moves can be determined.
- Self-Driving Cars: the actions of a self-driving car are not unique, it varies time to time.

ENVIRONMENT TYPES

Episodic (vs. **sequential):**

- In an episodic task environment, the agent's experience is divided into atomic episodes. Each episode consists of the agent perceiving and then performing a single action. The choice of action in each episode depends only the episode itself.
- In sequential environment, the current decision could affect all future decisions, such as *chess* and *taxi driving*.

ENVIRONMENT TYPES

Static (vs. **dynamic**): the environment is unchanged while the agent is deliberating.

- Taxi driving is dynamic: the other cars and taxi itself keep moving while the driving algorithm decides what to do next.
- Crossword puzzles are static.

The environment is semi-dynamic if the environment itself does not change with the passage of time by the agent's performance score does.

For example: chess when played with a clock is semi-dynamic.

ENVIRONMENT TYPES

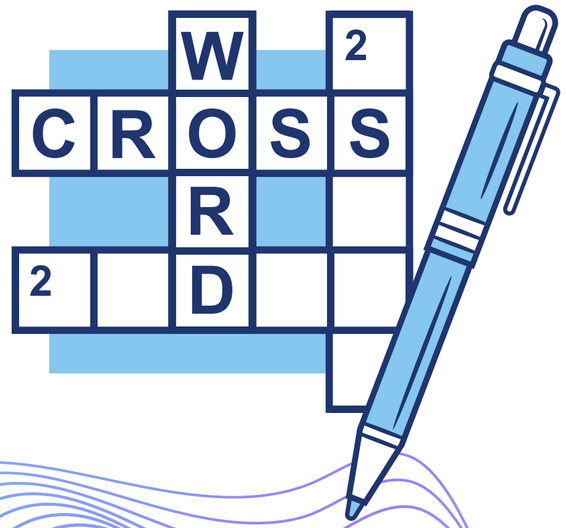
Discrete (vs. **continuous**): A limited number of distinct, clearly defined percepts and actions.

Example:

- Chess has a discrete set of percepts and actions.
- Taxing driving is a continuous state and continuous-time problem.

ENVIRONMENT TYPES

Single-agent (vs. **multi-agent**): an agent operating by itself in an environment.
Example: an agent solving a *crossword puzzle* by itself is clearly in a single-agent environment, whereas an agent *playing chess* is a two-agent environment.



AGENT FUNCTIONS AND PROGRAMS

- An agent is **completely specified by** the **agent function mapping percept sequences to actions.**
- The job of AI is to design the agent program that implements the agent function mapping percept to actions.
- One agent function is rational.
- Aim: find a way to implement the rational agent function concisely.

AGENT FUNCTIONS

- Take the current percept as input from the sensors (since nothing moreis available from the environment) and return an action to the actuators.
- Different from the agent function that takes the entire percept history.
- If the agent’s actions depend on the entire percept sequence, the agenthas to remember the percepts.

TABLE-DRIVEN AGENT

- To keep track of the percept sequence and then use it to index into a table of actions to decide what to do next.
- To build a rational agent in this way, one must construct a table that contains the appropriate action for every possible percept sequence

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

TABLE-DRIVEN AGENT

Cons:

- Huge table
- Take a long time to build the table
- No autonomy
- Need a long time to learn the table entries

VACUUM-CLEANER AGENT

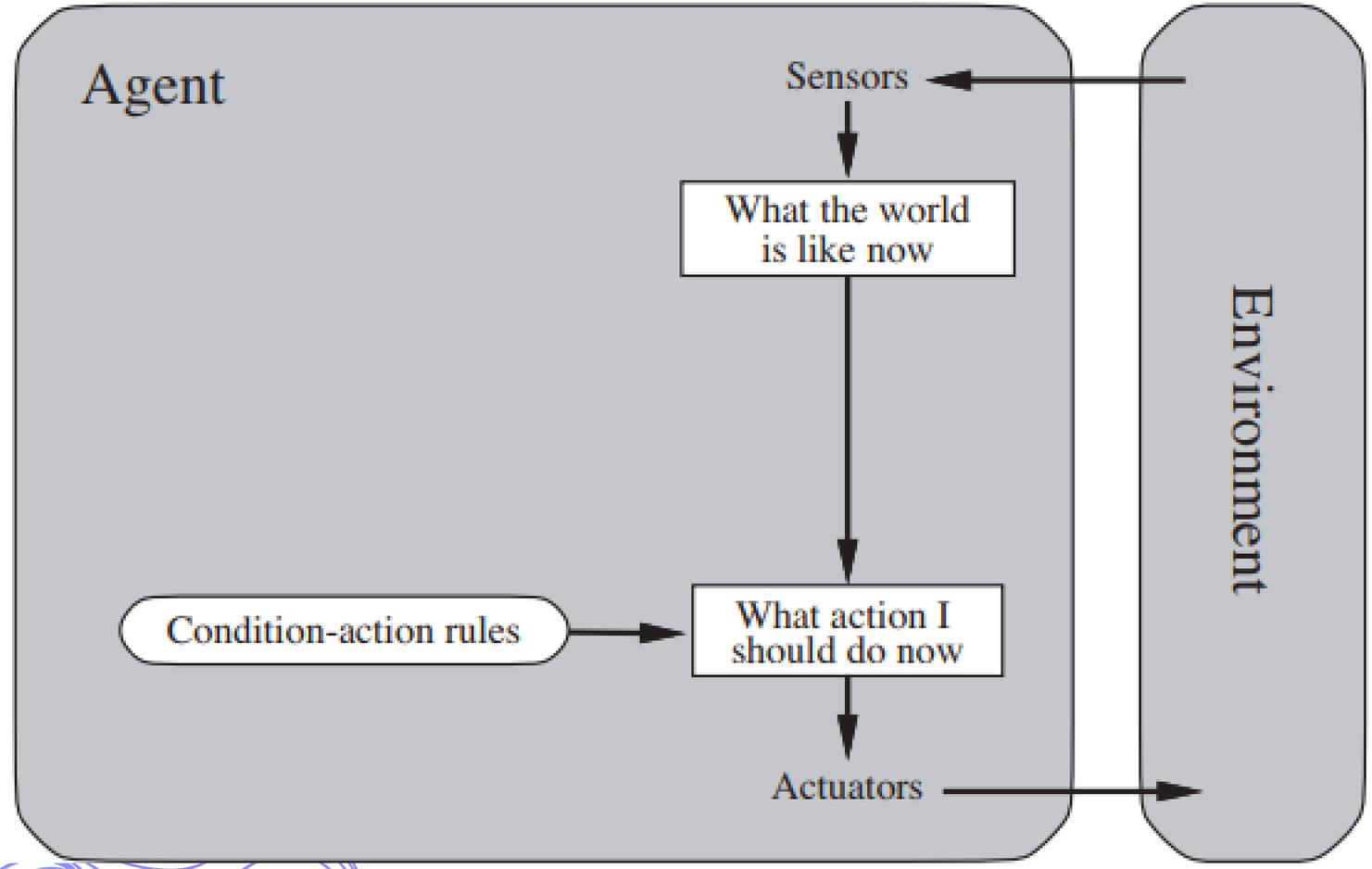
function REFLEX-VACUUM-AGENT([*location*,*status*]) **returns** an action

if *status* = *Dirty* **then return** *Suck*
else if *location* = *A* **then return** *Right*
else if *location* = *B* **then return** *Left*

Four basic kinds of agent programs:

- Simple reflex agents
- Model-based reflex agents
- Goal-based agents
- Utility-based agents

SIMPLE REFLEX AGENT

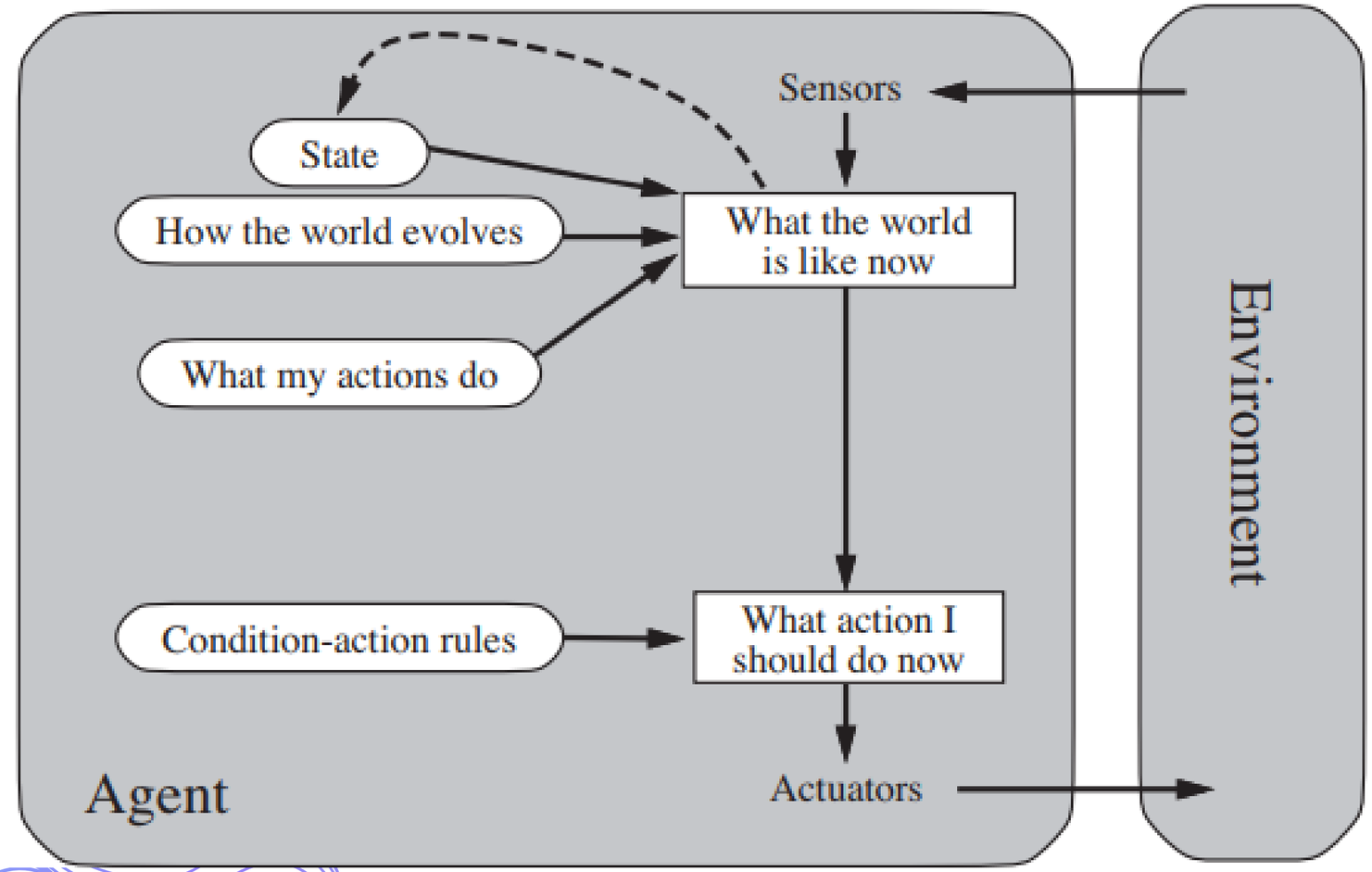


SIMPLE REFLEX AGENT

function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *rules*, a set of condition–action rules

  state ← INTERPRET-INPUT(*percept*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← *rule*.ACTION
  return *action*

MODEL-BASED REFLEX AGENTS



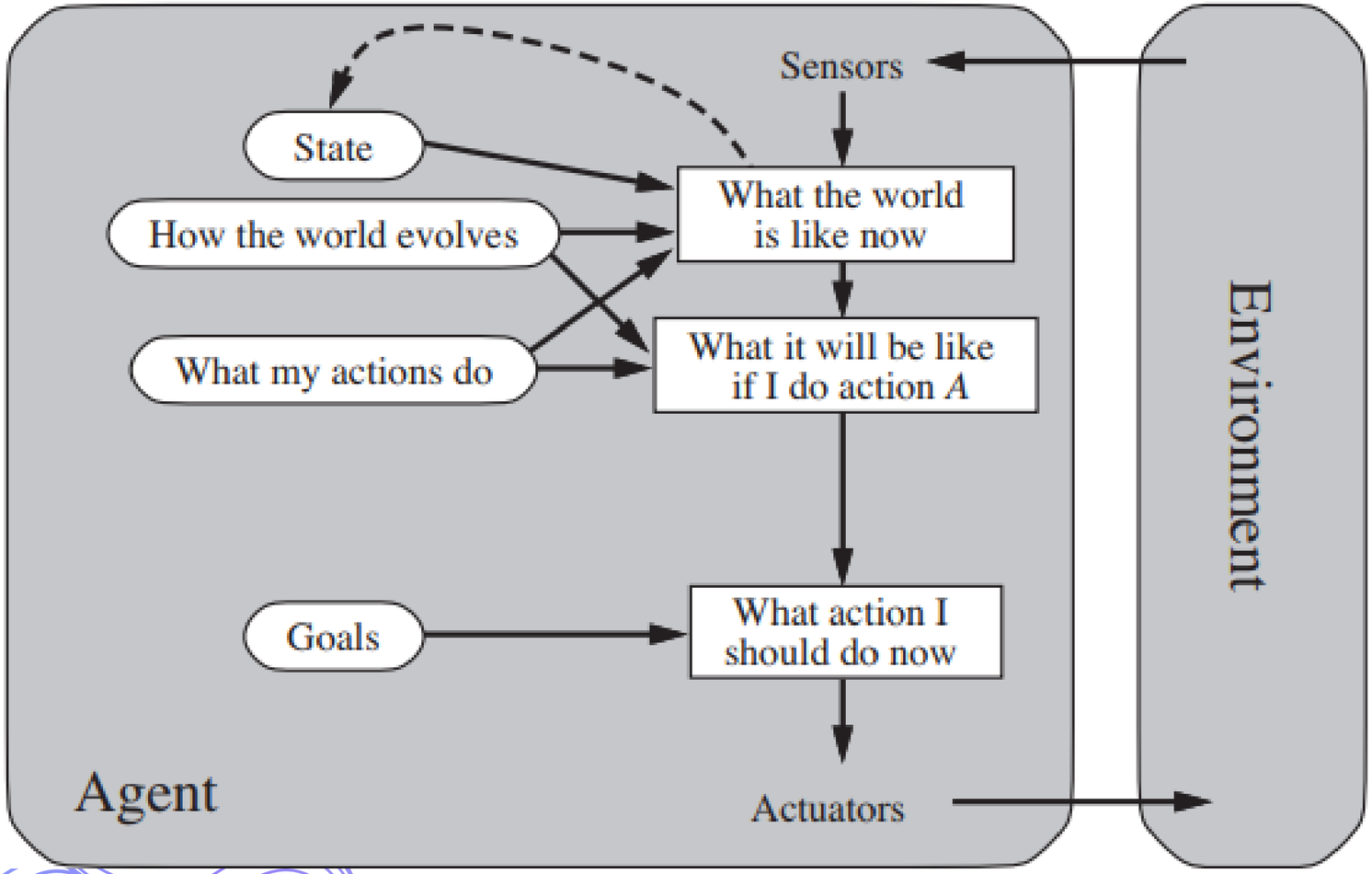
MODEL-BASED REFLEX AGENTS

function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action

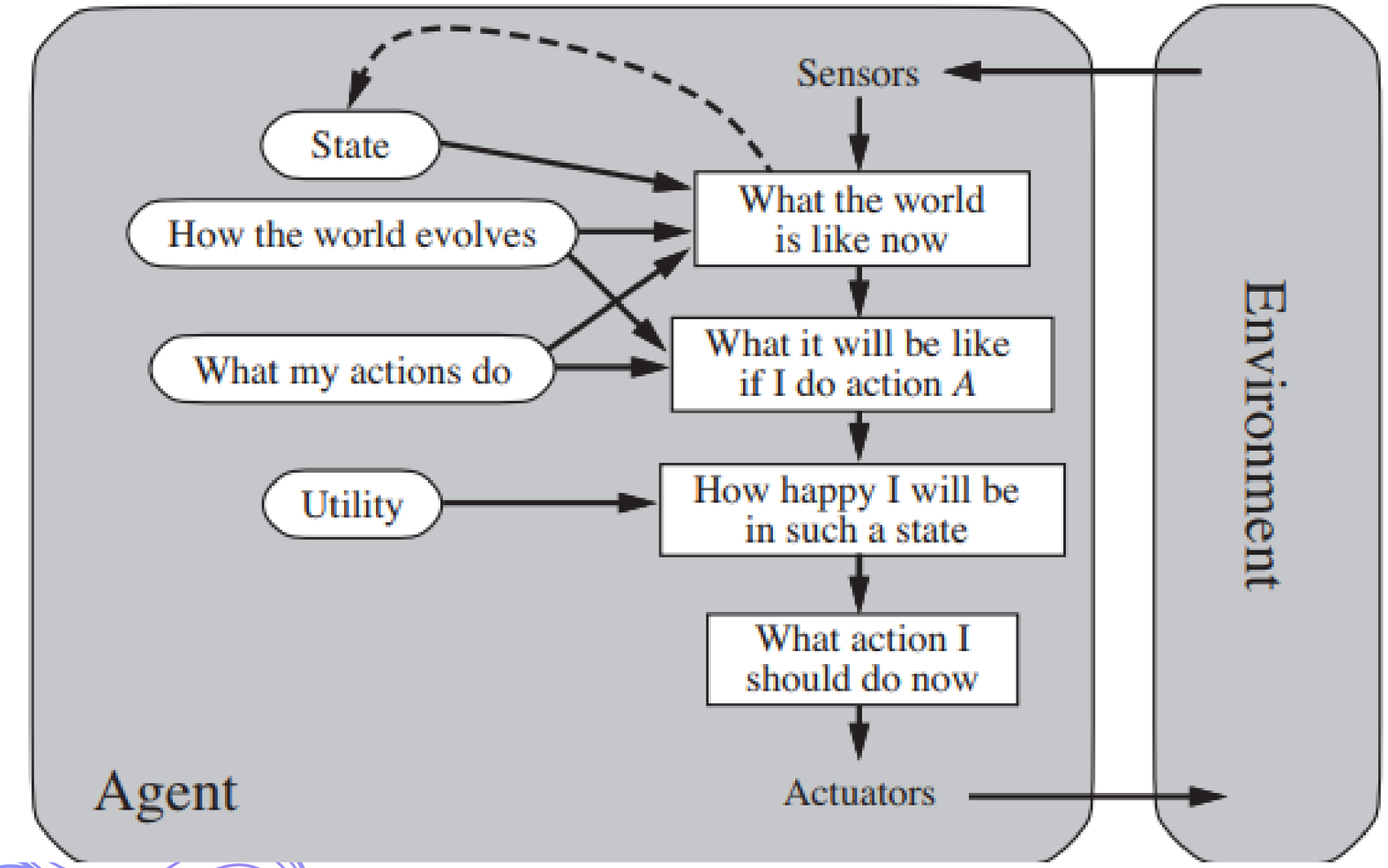
persistent: *state*, the agent's current conception of the world state
model, a description of how the next state depends on current state and action
rules, a set of condition–action rules
action, the most recent action, initially none

state ← UPDATE-STATE(*state*, *action*, *percept*, *model*)
rule ← RULE-MATCH(*state*, *rules*)
action ← *rule*.ACTION
return *action*

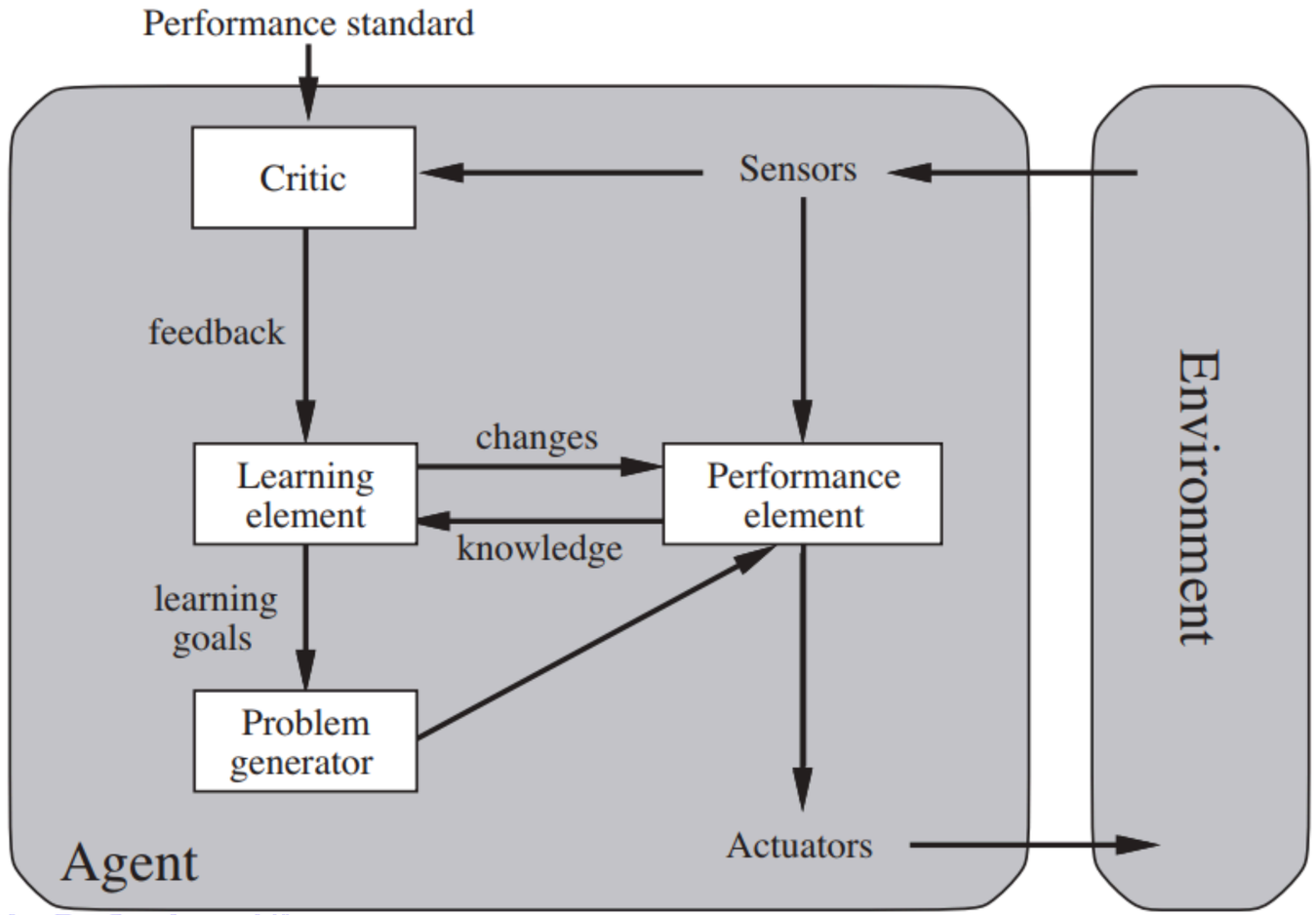
GOAL-BASED AGENT



UTILITY-BASED AGENTS

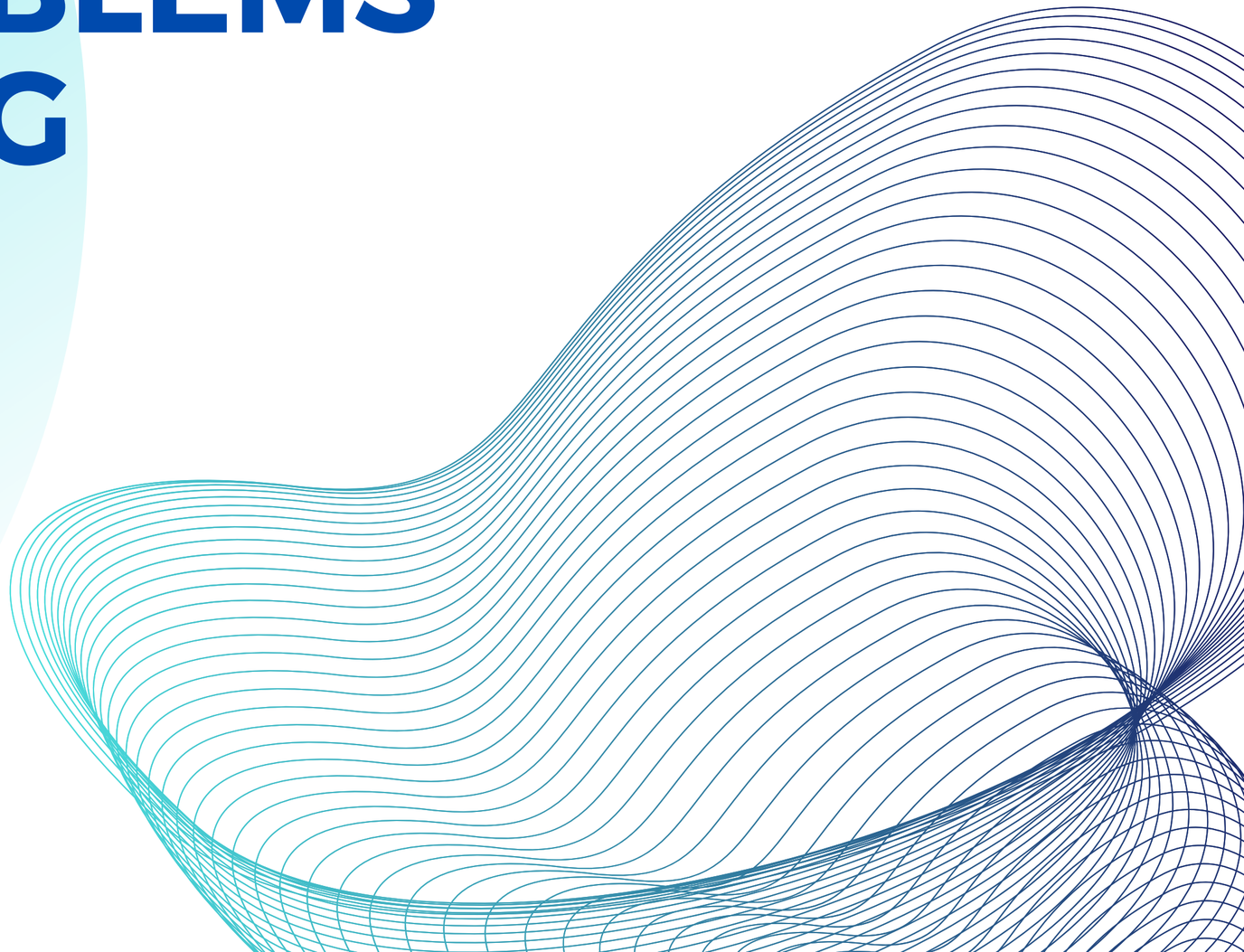


LEARNING AGENTS



SOLVING PROBLEMS BY SEARCHING

- Problem-solving agents
- Problem types
- Problem formulation
- Examples
- Basic search algorithms



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
      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
```

PROBLEM-SOLVING AGENTS

- On holiday in Vietnam; at the moment in Nha Trang.
- Flight leaves tomorrow in Hochiminh city.
- Formulate goal: be in Hochiminh city.
- Formulate problem:
  - States: Various provinces.
  - Actions: drive between provinces.
  - Find solutions: sequence of provinces.

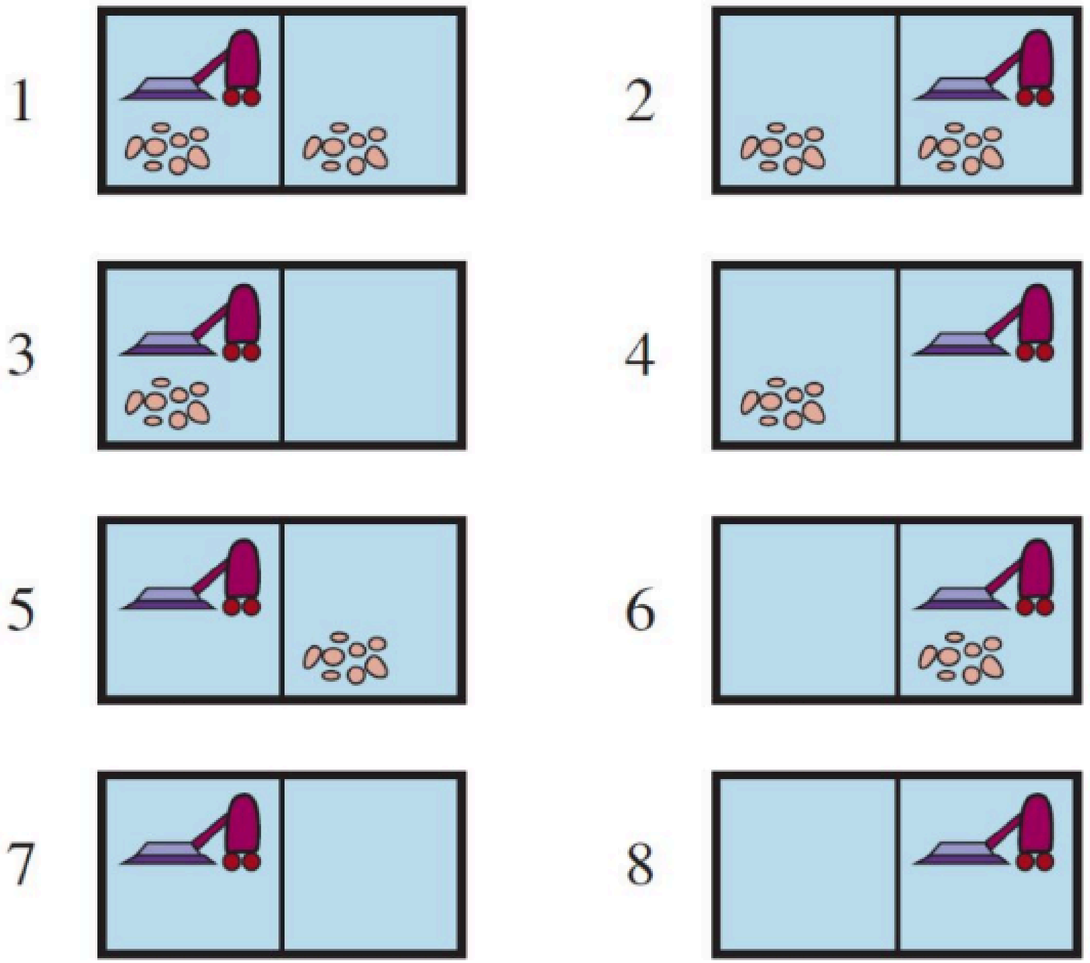
PROBLEM TYPES

- **Deterministic, fully observable**: single-state problem. Agents know exactly which state it will be in; solution is a sequence.
- **No-observable**: sensorless problem (conformant problem). Agent may have no idea where it is and solution is a sequence.

PROBLEM TYPES

- **Nondeterministic and/or partially observable**: contingency problem: percepts provide new information about the current state often interleave search, execution
- **Unknown state space**: exploration problem

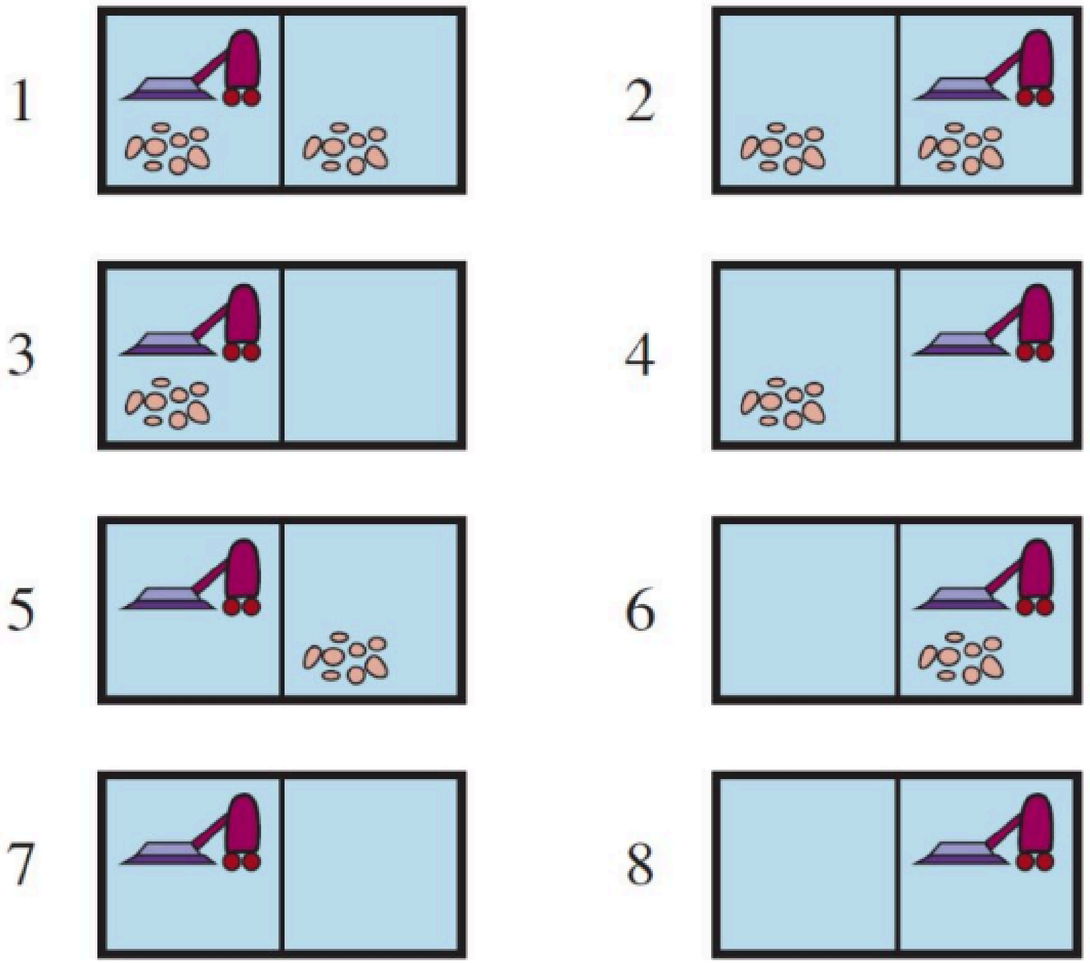
EXAMPLE: VACUUM WORLD



Single-state, start in state 5.

Solutions?

EXAMPLE: VACUUM WORLD

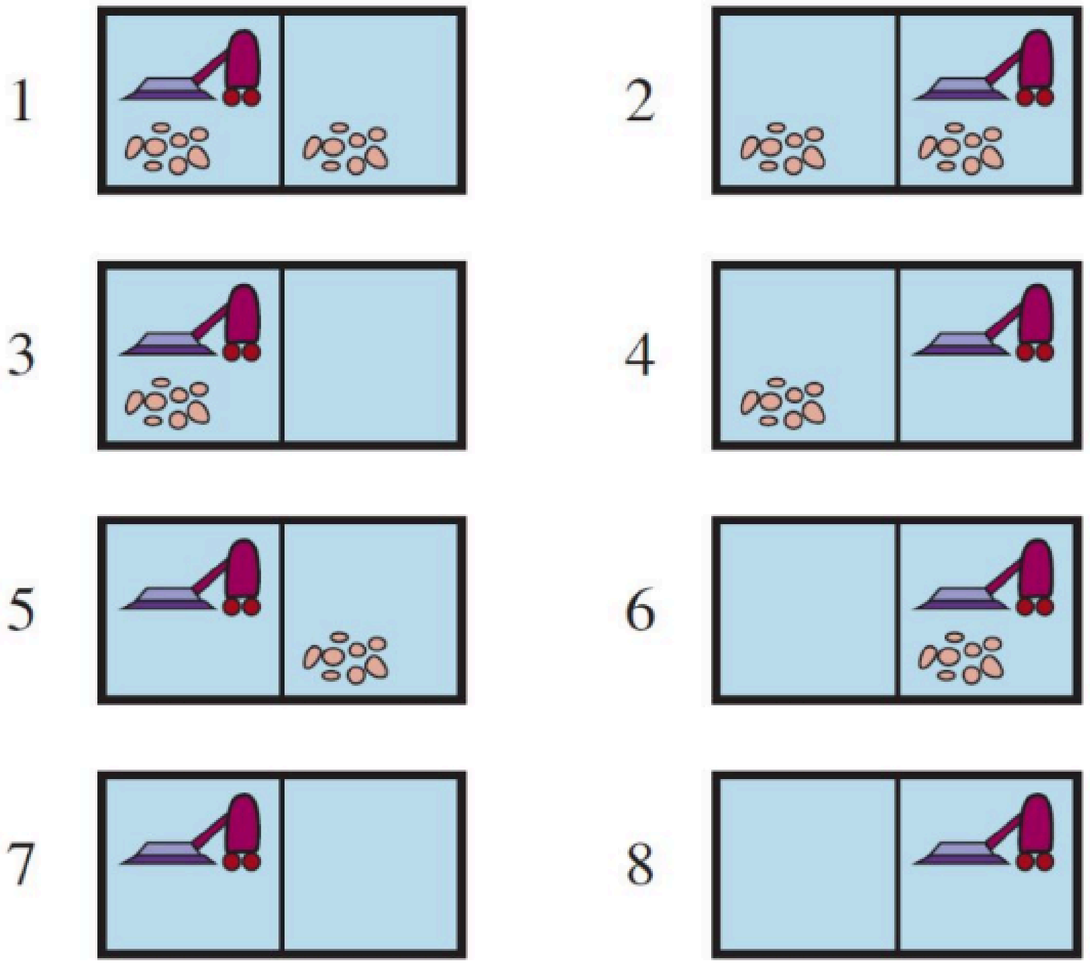


Single-state, start in state 5.

Solutions?

[Right,Suck]

EXAMPLE: VACUUM WORLD



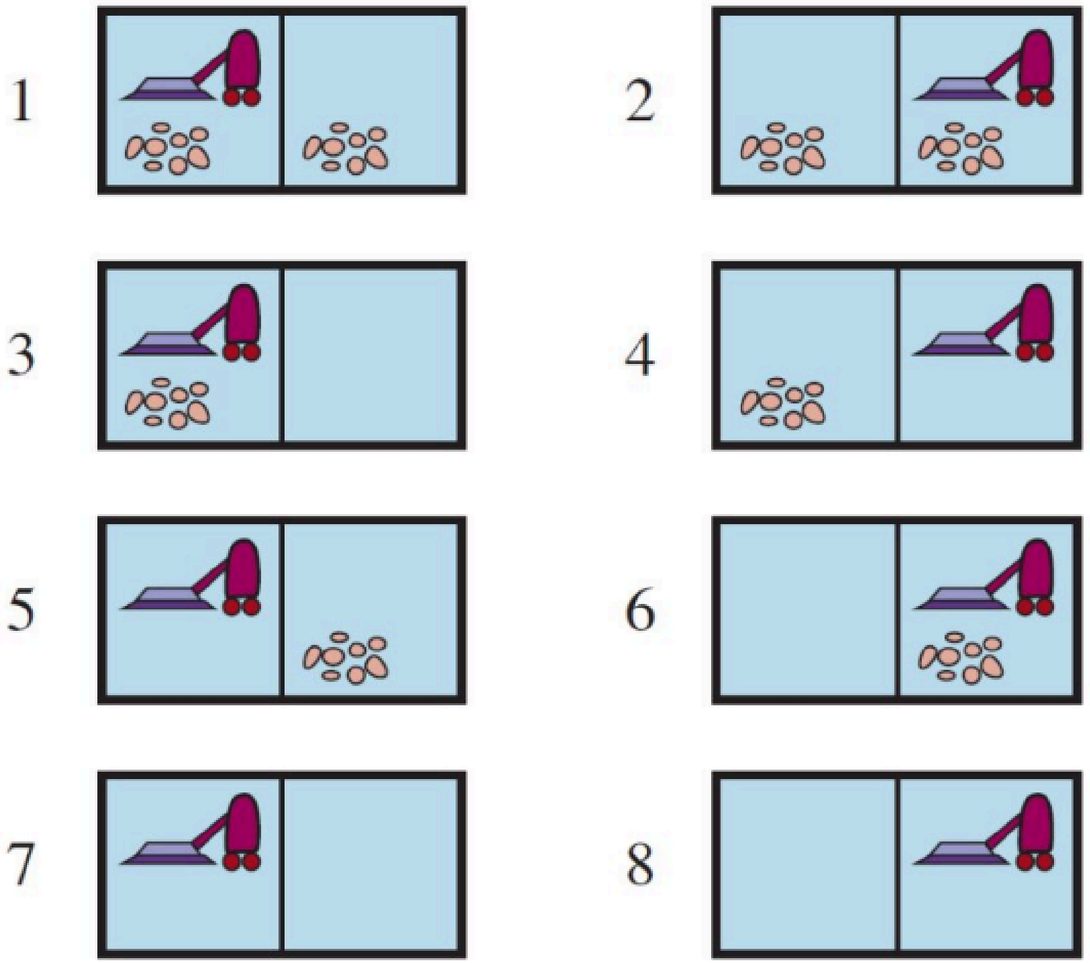
Sensorless:

Start in {1, 2, .., 6, 7, 8}

Right goes to {2, 4, 6, 8}

Solutions?

EXAMPLE: VACUUM WORLD



Contingency problems:

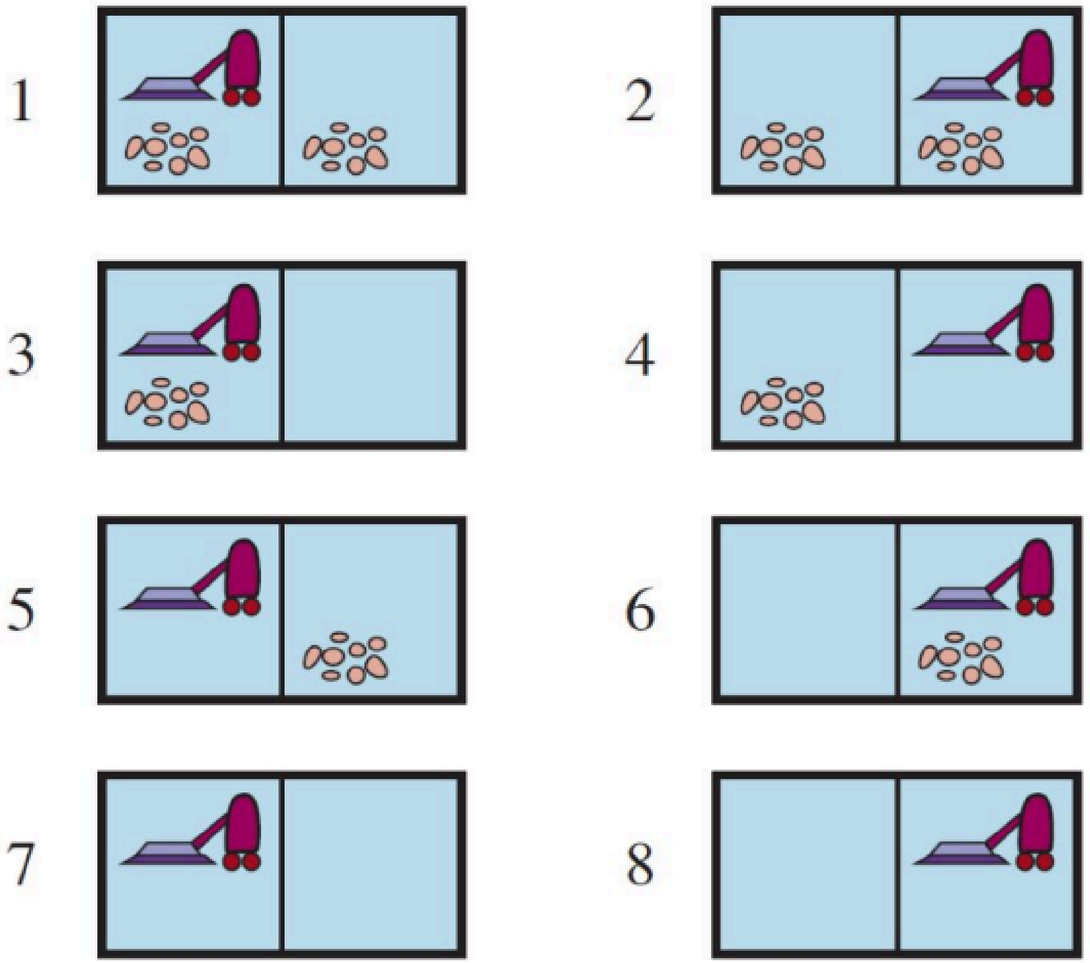
Nondeterministic: Suck may dirty a clean carpet.

Partially observable: location, dirty at current location.

Percept: [L,Clean], i.e., start in the state 5 or 7.

Solutions?

EXAMPLE: VACUUM WORLD



Contingency problems:

Nondeterministic: Suck may dirty a clean carpet.

Partially observable: location, dirty at current location.

Percept: [L,Clean], i.e., start in the state 5 or 7.

Solutions?

[Right, if Dirty then Suck]

SINGLE-STATE PROBLEM FORMULATION

- One problem can be determined by the following items:
  - **Initial state**: where the agent starts in, e.g., “at Nha Trang”
  - **Actions or successor function** S(x) = set of action-state pairs.
  - **Goal test**: explicit/implicit to determine whether a given state is a goal state.
  - **Path cost**: to show a numeric cost to each path, e.g., sum of distances, number of actions executed,...
    - C(x,a,y) = step cost (non-negative)

SINGLE-STATE PROBLEM FORMULATION

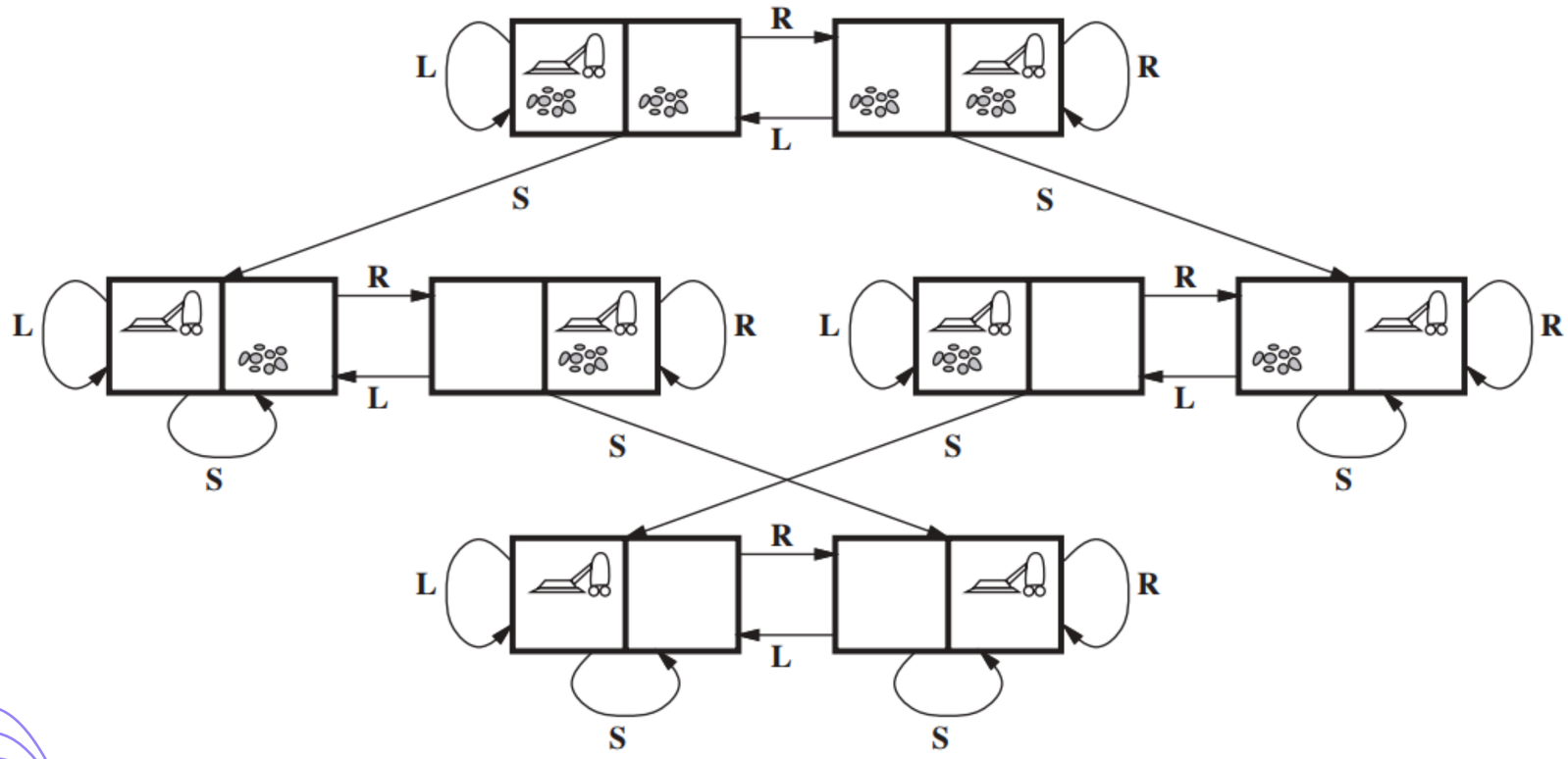
- A solution of the problem is a sequence of necessary actions from the initial state to achieve the goal state.
- Solution quality is measured by the path cost function and an optimal solution has the lowest path cost among all possible solutions.

SELECTING A STATE SPACE

- Real world is complicated. **State space** must be abstracted for problem solving.
- Abstract state = a set of real states.
- Abstract action = a complex combination of real actions.
- Abstract solution = a set of real paths that are solutions in the real world.

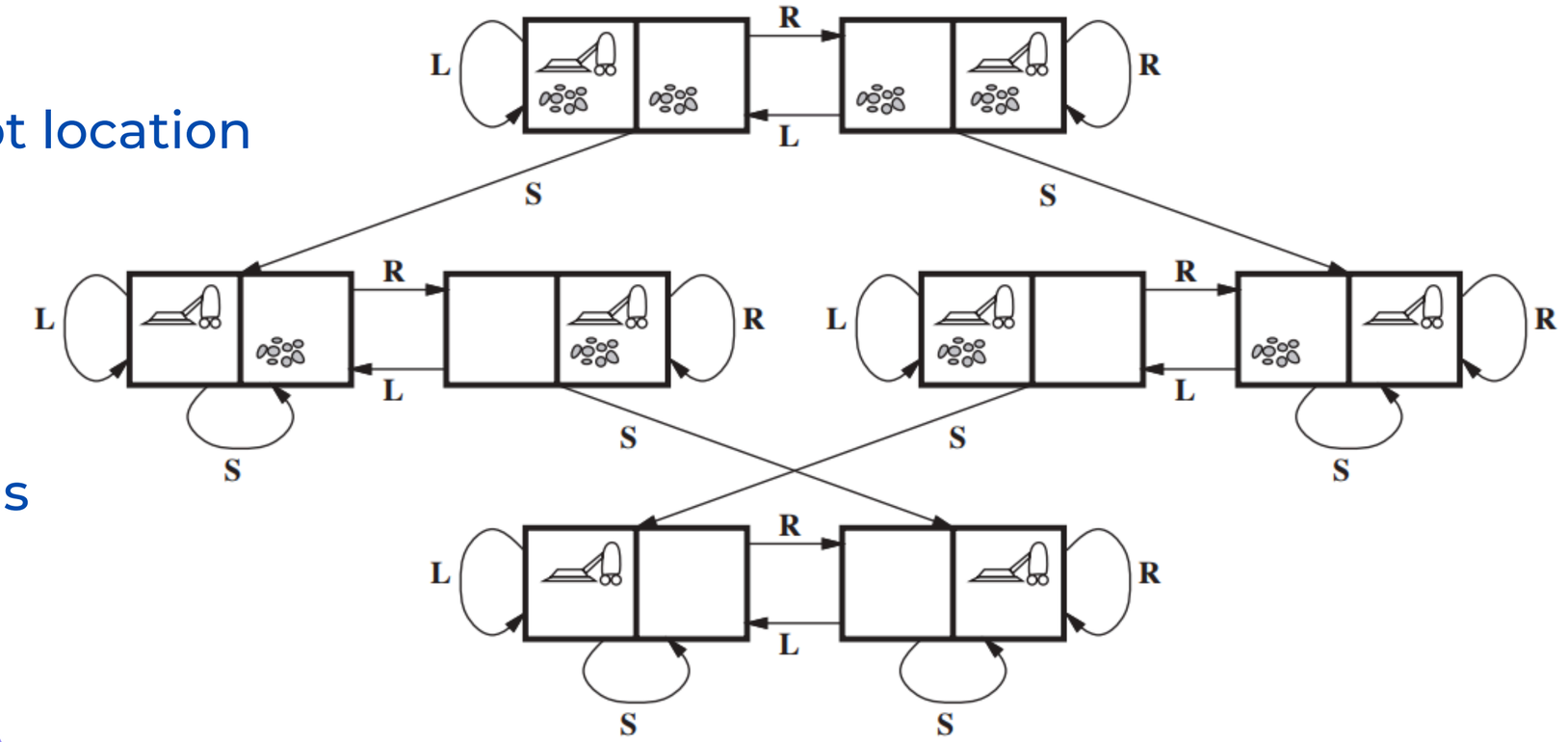
VACUUM WORLD STATE SPACE GRAPH

- States?
- Actions?
- Goal test?
- Path cost?



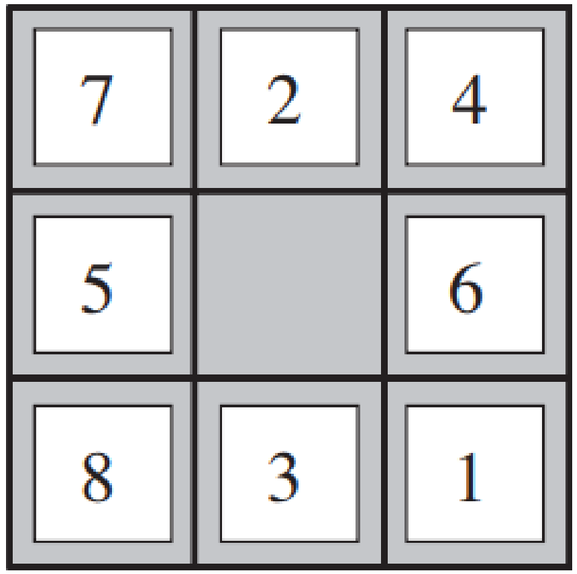
VACUUM WORLD STATE SPACE GRAPH

- **States?**
  - integer dirt and robot location
- **Actions?**
  - Left, Right, Suck
- **Goal test?**
  - No dirt at all locations
- **Path cost?**
  - 1 per action

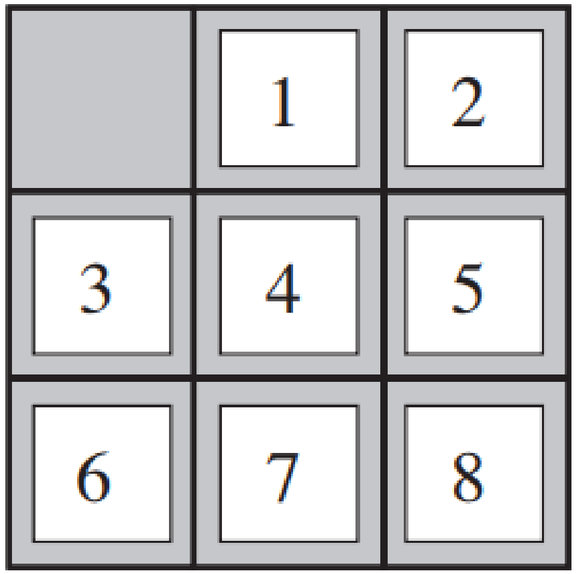


THE 8-PUZZLE

- States?
- Actions?
- Goal test?
- Path cost?



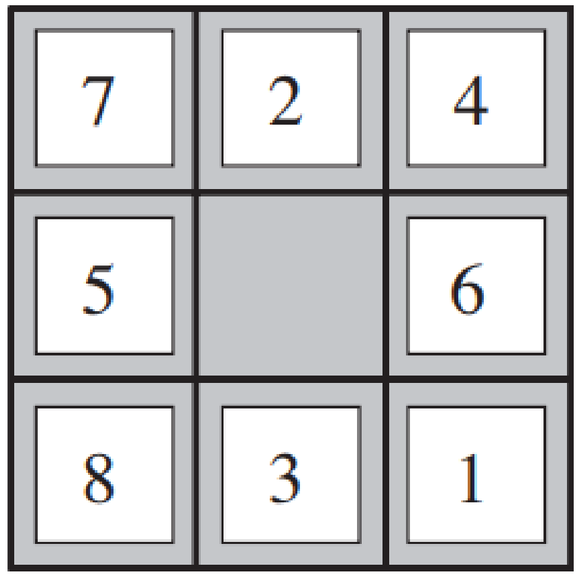
Start State



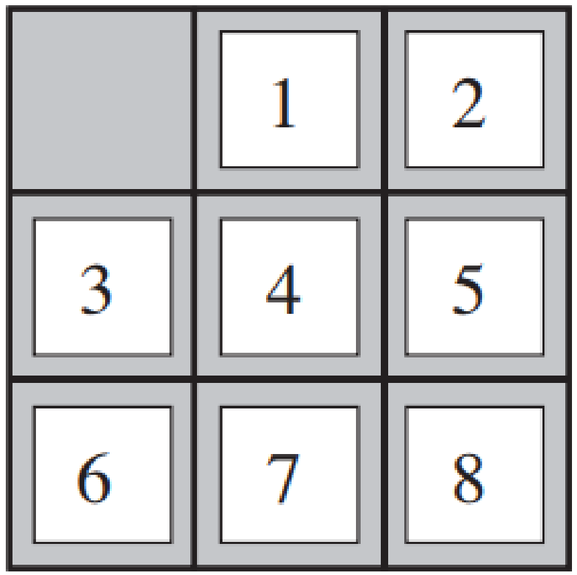
Goal State

THE 8-PUZZLE

- **States?** locations of tiles
- **Actions?** move blank left, right, up, down
- **Goal test?** a given goal state
- **Path cost?** 1 per move
- Optimal solution of n-Puzzle family is NP-hard!!!



Start State



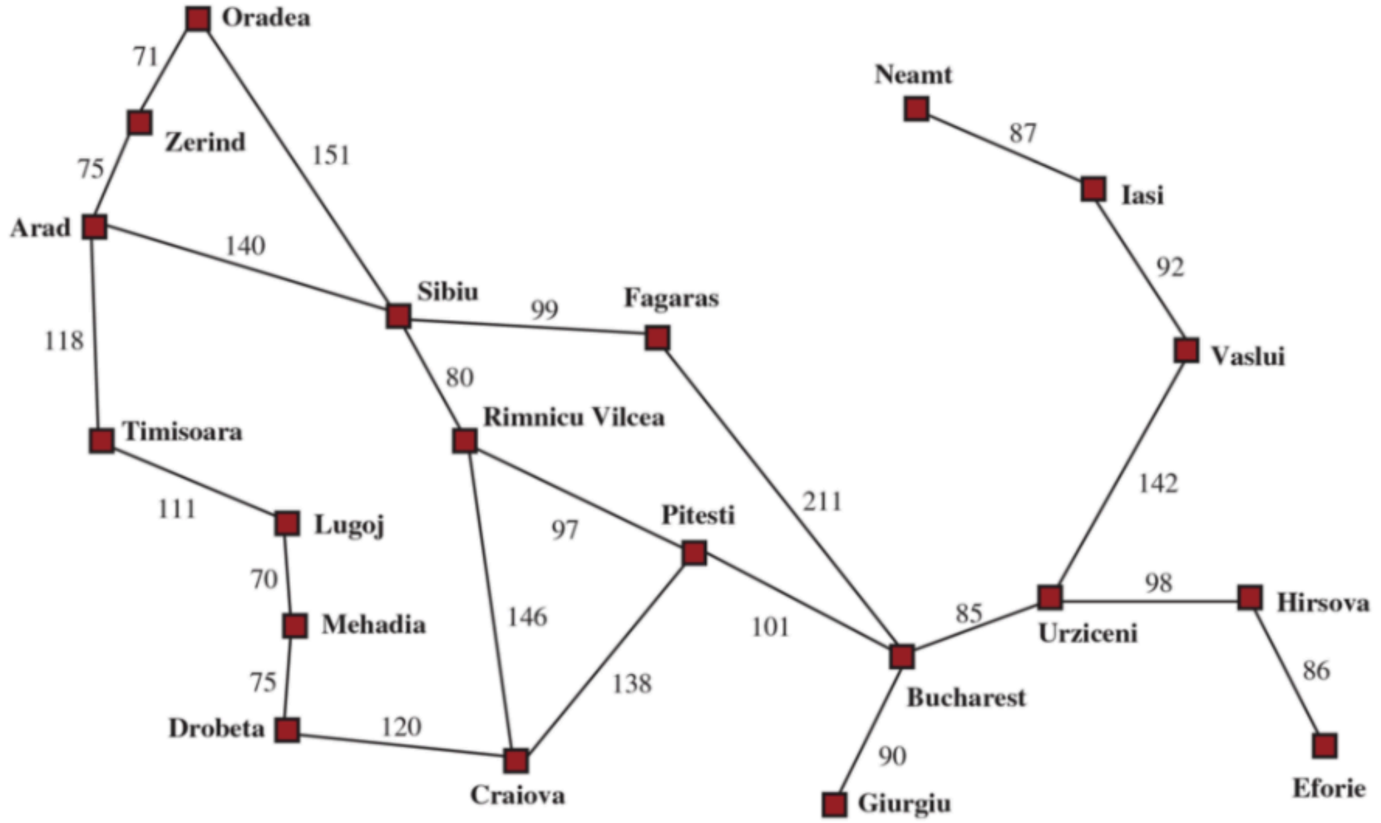
Goal State

TREE SEARCH ALGORITHMS

- After formulating the problem, we have to solve it. It can be done by a search through the state space.
- Basic idea:

```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

TREE SEARCH ALGORITHMS



A simplified road map of part of Romania, with road distances in miles.

TREE SEARCH ALGORITHMS

General tree-search algorithm

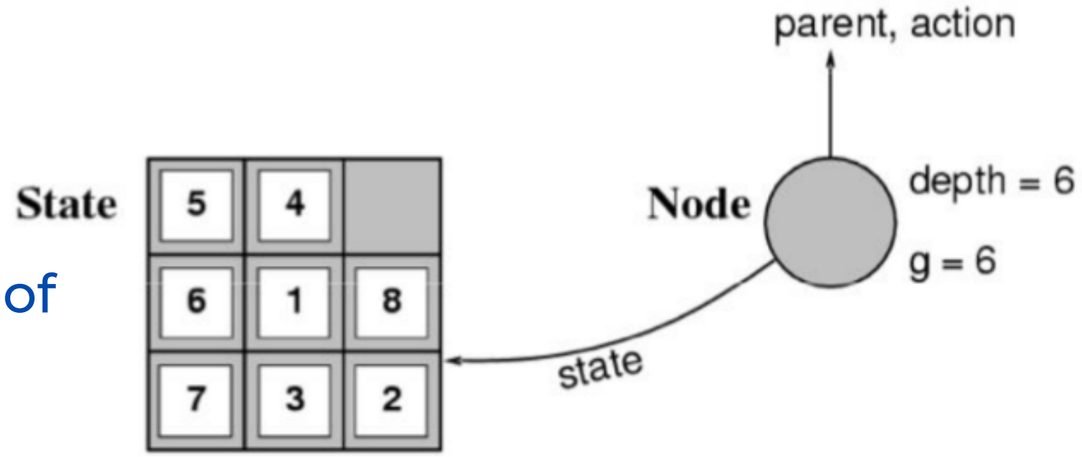
```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

GENERAL TREE-SEARCH ALGORITHMS

Implementation: States vs Nodes

- **A state** is a representation of a physical configuration
- **A node** is a data structure constituting part of a search tree, including: state, parent node, action, path-cost and depth (the number of steps along the path from the initial state).



GENERAL TREE-SEARCH ALGORITHMS

- The **“Expand” functions** creates new nodes, filling in the various fields and using the “SuccessorFn” of the problem to create the corresponding states.
- **A fringe** is a collection of nodes that have been generated but not expanded.

TREE SEARCH ALGORITHMS

- A **search strategy** is defined by picking the **order of node expansion.**
- Strategies are **evaluated** along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: number of nodes generated
  - **space complexity**: How much memory is needed to perform the search
  - **optimality**: does it always find a least-cost solution?
- **Time and space complexity** are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space