Module 2

Problem-solving

Agents

- How an agent can find a sequence of actions that achieves its goals when no single action will do.
- Reflex agents – simplest agents which base their actions on a direct mapping from states to actions. Such agents cannot operate well in environments for which this mapping would be too large to store and would take too long to learn.
- Goal-based agents - consider future actions and the desirability of their outcomes.

Problem-solving Agent

- Problem-solving agent – one kind of goal-based agent.
- Problem-solving agents use **atomic** representations, that is, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms.
- Goal-based agents that use more advanced **factored** or **structured** representations are usually called **planning agents.**

Solving Problems by Searching

- **Uninformed** search algorithms — algorithms that are given no information about the problem other than its definition.

  Although some of these algorithms can solve any solvable problem, none of them can do so efficiently.

- **Informed** search algorithms - can do quite well given some guidance on where to look for solutions.

PROBLEM-SOLVING AGENTS

- Intelligent agents are supposed to maximize their performance measure. This can be , achieved if the agent can adopt a goal and aim at satisfying it.
- *Ex. Imagine an agent in the city of Arad, Romania, enjoying a touring holiday.*
- *Now, suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day. In that case, it makes sense for the agent to adopt the goal of getting to Bucharest.*
- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
- .

Steps In Solving a Problem

1. Goal formulation

- It organizes finite steps to formulate a target/ goals which requires some actions to achieve the goal.

2. Problem formulation

- Decides what action should be taken to achieve the formulated the goal.

- *We will consider a goal to be a set of world states—exactly those states in which the goal is satisfied.*
- *The agent's task is to find out how to act, now and in the future, so that it reaches a goal state. Before it can do this, it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider.*

- Problem formulation is the process of deciding what actions and states to consider, given a goal.
- Eg : driving from Bangalore to Chennai.
- States: places in between Bangalore to Chennai
- Actions: turn left, turn right, accelerate & brake etc.
- In general, an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.

Properties of Environment:

- Ex- *Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad.*
- If the agent has no additional information—i.e., if the environment is **unknown** in the sense-—then it is has no choice but to try one of the actions at random.
- *Ex-But suppose the agent has a map of Romania.*
- In general, an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of **known** value.

- We assume that the environment is observable, so the agent always knows the current state.
- Ex- *For the agent driving in Romania, it's reasonable to suppose that each city on the map has a sign indicating its presence to arriving drivers.*
- Environment is discrete, so at any given state there are only finitely many actions to choose from.
- *Ex-This is true for navigating in Romania because each city is connected to a small number of other cities.*

- We will assume the environment is known, so the agent knows which states are reached by each action.
- Finally, we assume that the environment is deterministic, so each action has exactly one outcome.

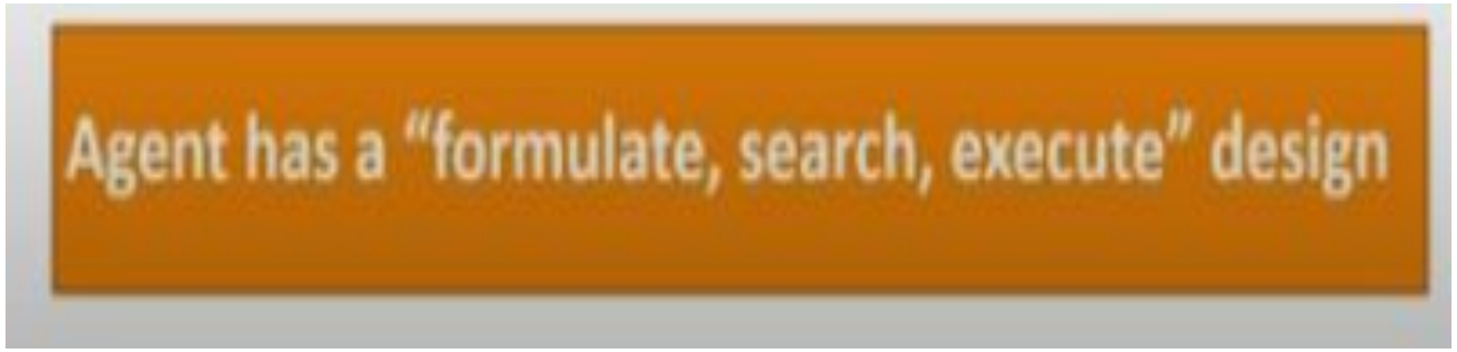
- *Under these assumptions, the solution to any problem is a fixed sequence of actions.*

Problem-solving Agents

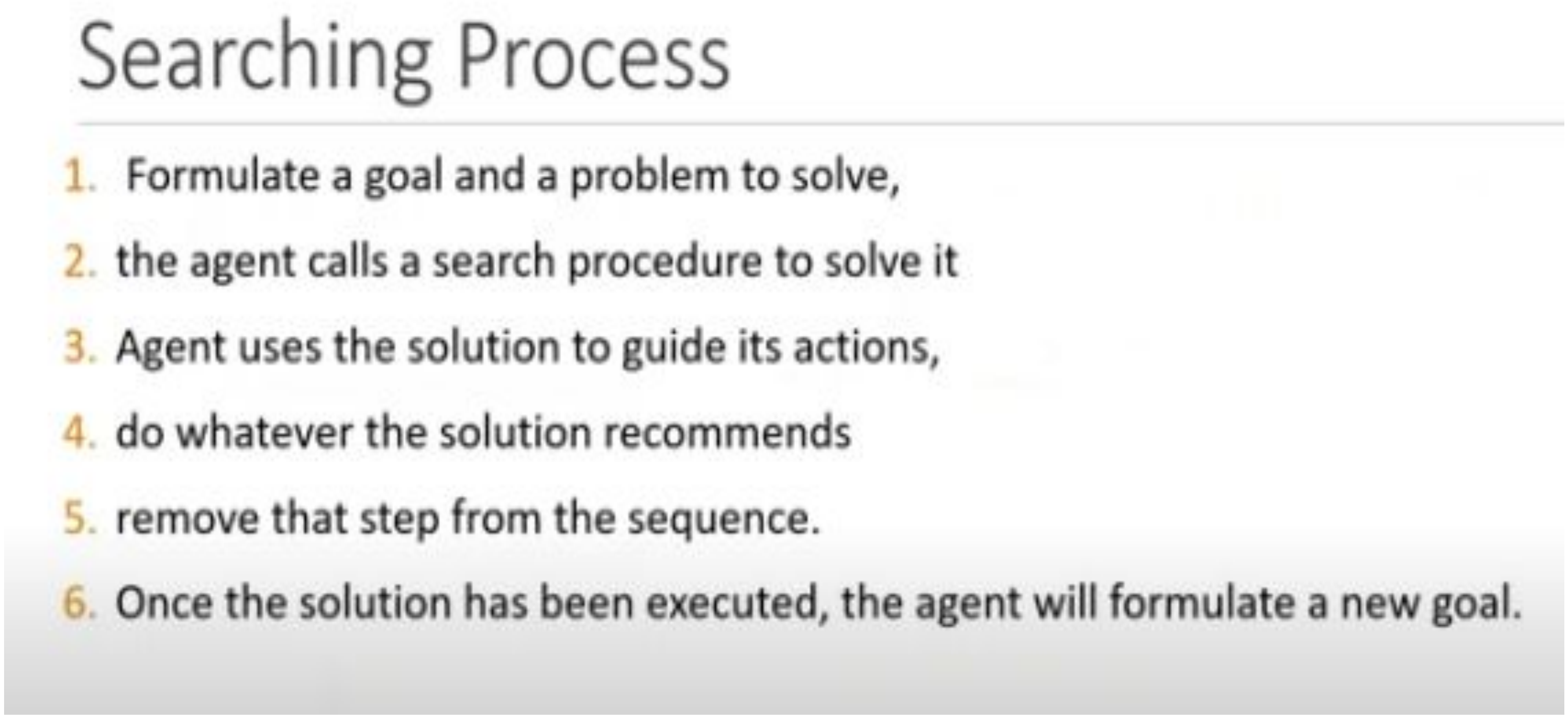
- In general, it could be a branching strategy that recommends different actions in the future depending on what percepts arrive.
- If the agent knows the initial state and the environment is known and deterministic, it knows exactly where it will be after the first action and what it will perceive.
- Since only one percept is possible after the first action, the solution can specify only one possible second action, and so on.

Search

- The process of looking for a sequence of actions that reaches the goal is called search.
- A search algorithm takes a problem as input and returns a solution in the form of an action sequence.
- Ex-Travelling salesman problem
- Once a solution is found, the actions it recommends can be carried out. This is called the execution phase.
- Thus, we have a simple “formulate, search, execute” design for the agent.


Agent has a "formulate, search, execute" design

- A simple problem-solving agent first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.



Searching Process

1. Formulate a goal and a problem to solve,
2. the agent calls a search procedure to solve it
3. Agent uses the solution to guide its actions,
4. do whatever the solution recommends
5. remove that step from the sequence.
6. Once the solution has been executed, the agent will formulate a new goal.

```
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

  $state \leftarrow$ UPDATE-STATE($state, percept$)
  if $seq$ is empty then
      $goal \leftarrow$ FORMULATE-GOAL($state$)
      $problem \leftarrow$ FORMULATE-PROBLEM($state, goal$)
      $seq \leftarrow$ SEARCH($problem$)
      if $seq = failure$ then return a null action
  $action \leftarrow$ FIRST($seq$)
  $seq \leftarrow$ REST($seq$)
  return $action$
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

- After formulating a goal and a problem to solve, the agent calls a search procedure to solve it.
- It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence.
- Once the solution has been executed, the agent will formulate a new goal.

Open-loop system

while the agent is executing the solution sequence it *ignores its percepts* when choosing an action because it knows in advance what they will be

An agent that carries out its plans with its eyes closed, so to speak, must be quite certain of what is going on is an open loop.

ignoring the percepts breaks the loop between agent and environment.

Well-defined problems and solutions

- A problem can be defined formally by five components:

- The initial state that the agent starts in.

For example, the initial state for our agent in Romania might be described as In(Arad).

- A description of the possible actions available to the agent.

Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is applicable in s.

For example, from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.

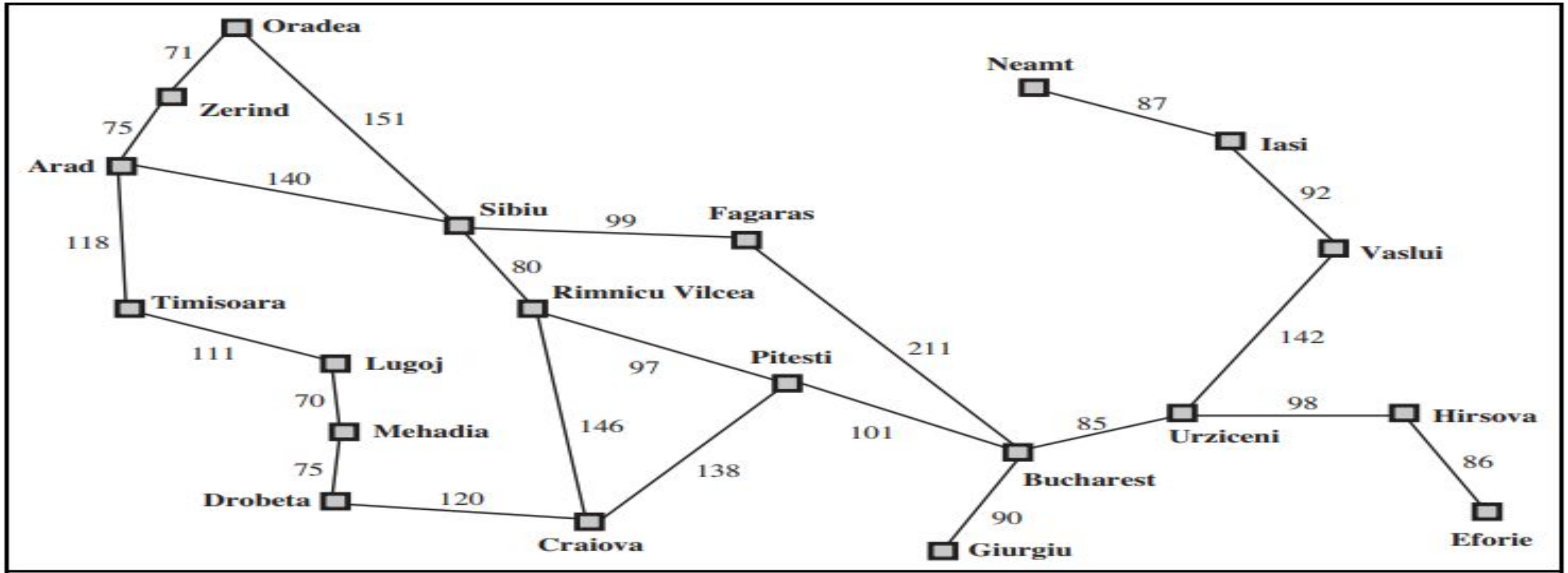


Figure 3.2 A simplified road map of part of Romania.

- A description of what each action does; the formal name for this is the transition model, specified by a function RESULT(s, a) that returns the state that results from doing action a in state s.

- We also use the term successor to refer to any state reachable from a given state by a single action.

RESULT(In(Arad),Go(Zerind)) = In(Zerind) .

- Together, the initial state, actions, and transition model implicitly define the state space of the problem—the set of all states reachable from the initial state by any sequence of actions.
- The state space forms a directed network or graph in which the nodes are states and the links between nodes are actions.
- A path in the state space is a sequence of states connected by a sequence of actions.
- The goal test, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.

- The agent’s goal in Romania is the singleton set {In(Bucharest)}.

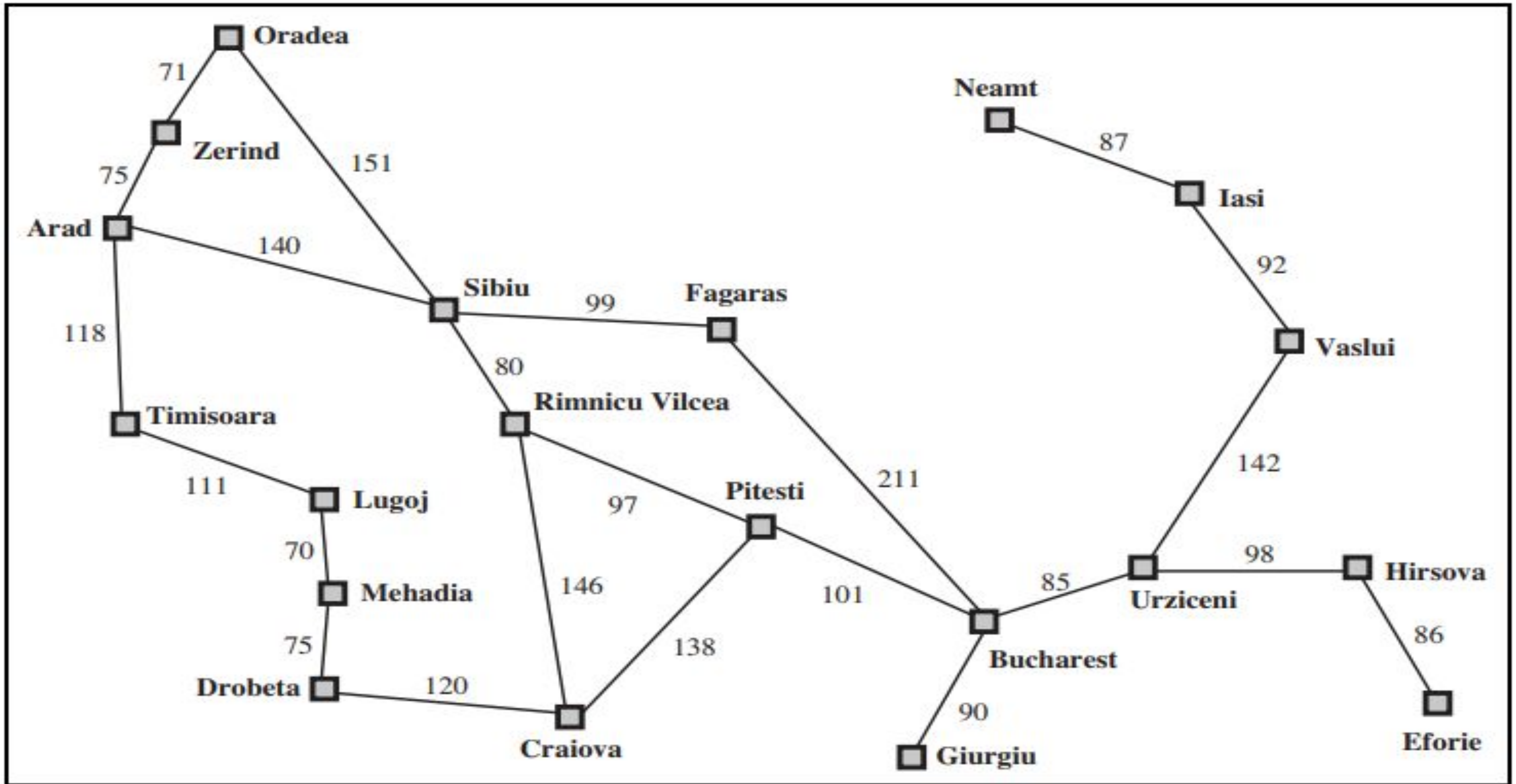


Figure 3.2 A simplified road map of part of Romania.

- A path cost function that assigns a numeric cost to each path.

The problem-solving agent chooses a cost function that reflects its own performance measure.

For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers.

We assume that the cost of a path can be described as the sum of the costs of the individual actions along the path.

The step cost of taking action a in state s to reach state s’ is denoted by c(s, a, s’).

The step costs for Romania are shown in Figure 3.2 as route distances. We assume that step costs are nonnegative.

- A solution to a problem is an action sequence that leads from the initial state to a goal state.

- Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

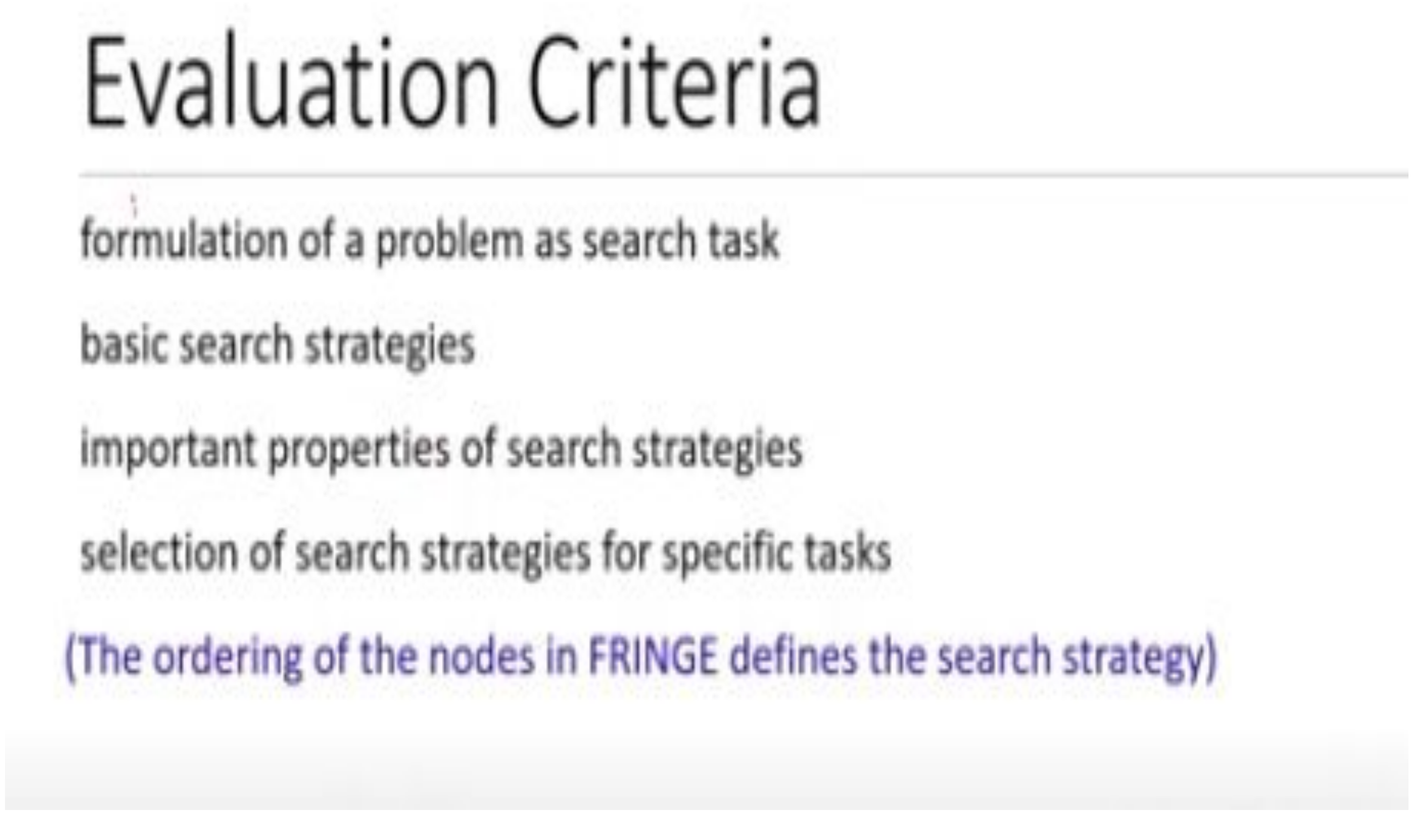
A **problem** can be defined formally by five components:

- Initial state.
- Actions.
- Transition model.
- Goal test.
- Path cost.

Abstraction

- The process of removing detail from a representation is called abstraction.
- The abstraction is *useful* if carrying out each of the actions in the solution is easier than the original problem.
- The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.
- Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

- *In(Arad), to an actual crosscountry trip, where the state of the world includes so many things: the traveling companions, the current radio program, the scenery out of the window, the proximity of law enforcement officers, the distance to the next rest stop, the condition of the road, the weather, and so on. All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest.*
- The process of removing detail from a representation is called abstraction.
- In addition to abstracting the state description, we must abstract the actions themselves. A driving action has many effects. Besides changing the location of the vehicle and its occupants, it takes up time, consumes fuel, generates pollution, and changes the agent.



Evaluation Criteria

formulation of a problem as search task

basic search strategies

important properties of search strategies

selection of search strategies for specific tasks

(The ordering of the nodes in FRINGE defines the search strategy)

Example: Romania

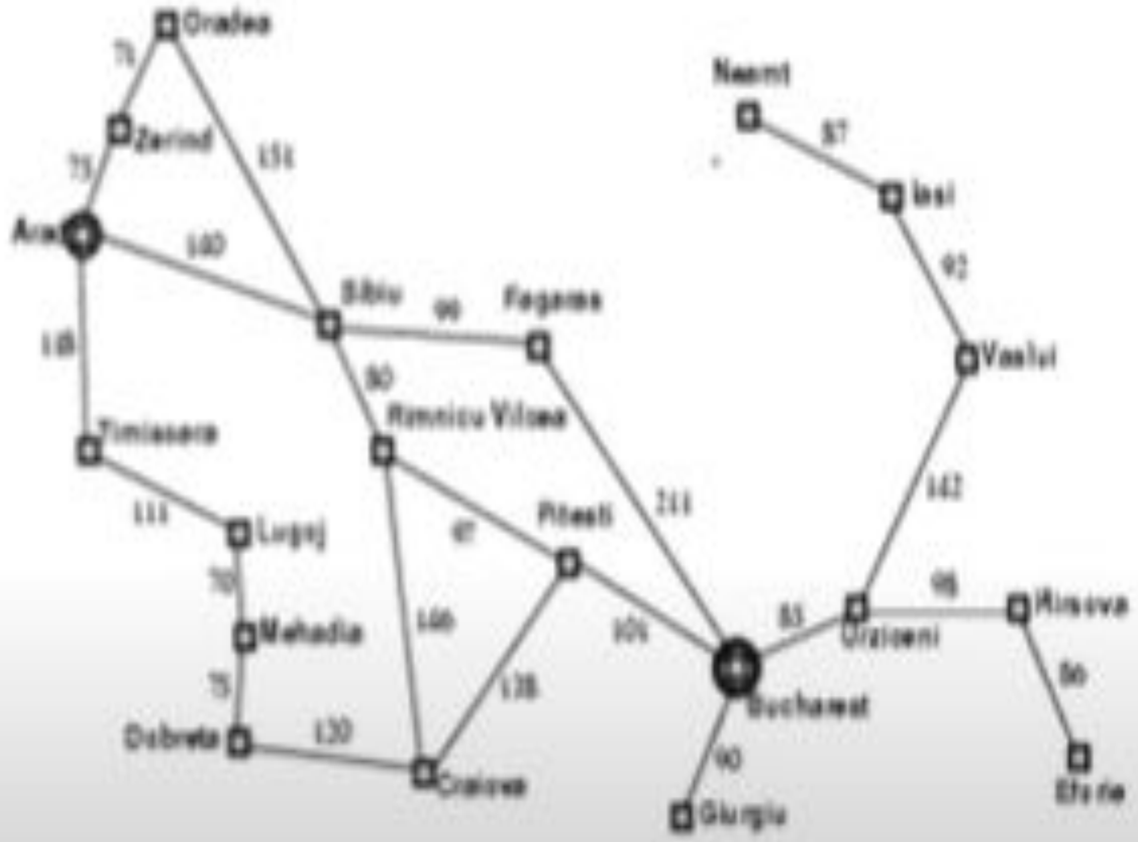
On holiday in Romania; currently in Arad.

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. **Problem : To Go from** Arad **to** Bucharest
2. **Initial State :** Arad
3. **Operator : Go from One City To another .**
4. **State Space : {**Sibiu, Fagaras, Timisora,....**}**
5. **Goal Test : are the agent in** Bucharest**.**
6. **Path Cost Function : Get The Cost From The Map.**
7. **Solution :** $\left\{ \underline{\{Ar \rightarrow sib \rightarrow Fr \rightarrow Bu\}} \;,\; \underline{\{Ar \rightarrow Ti \rightarrow Lu \rightarrow Me \rightarrow Cr, \rightarrow Pi \rightarrow Bu\}} \ldots\ldots \right\}$
8. **State Set Space :** {Arad $\underline{\rightarrow}$ Sibiu $\underline{\rightarrow}$ Fagaras $\underline{\rightarrow}$ Bucharest}

Example Problems

- A **toy problem** is intended to illustrate or exercise various problem-solving methods.

  - It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms.

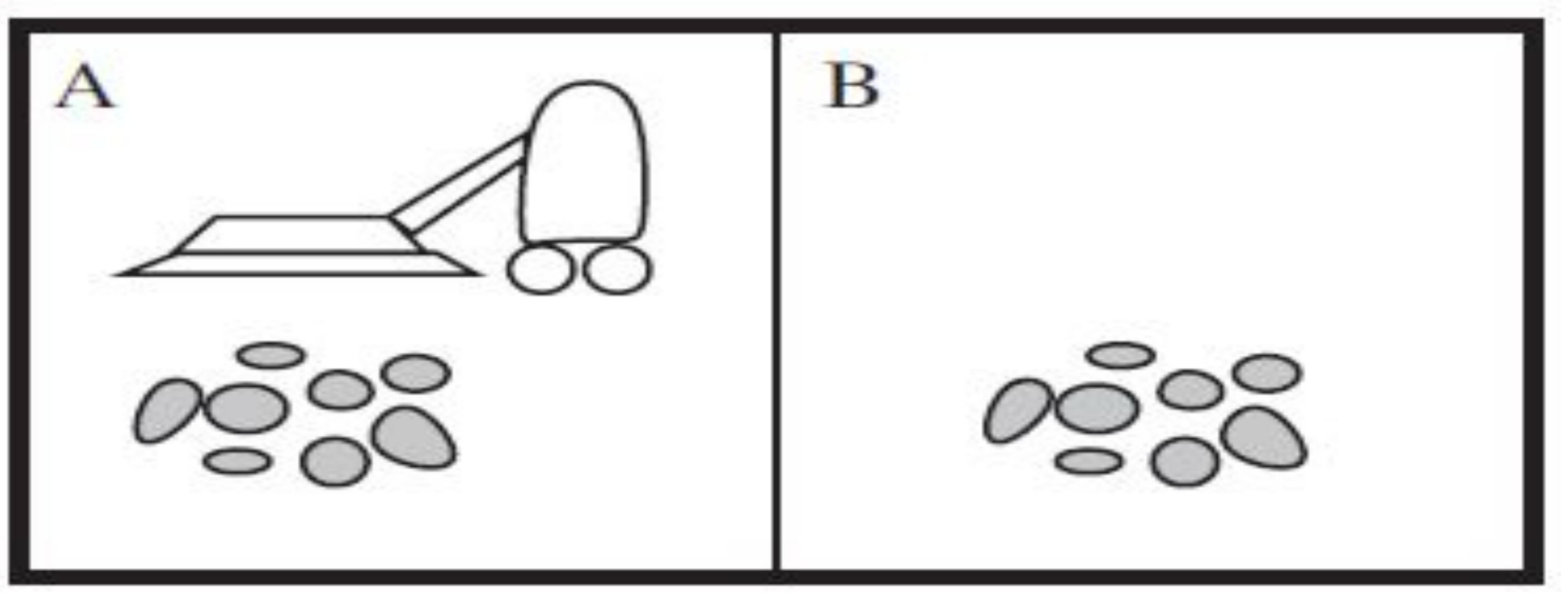
- A **real-world problem** is one whose solutions people actually care about.

  - Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

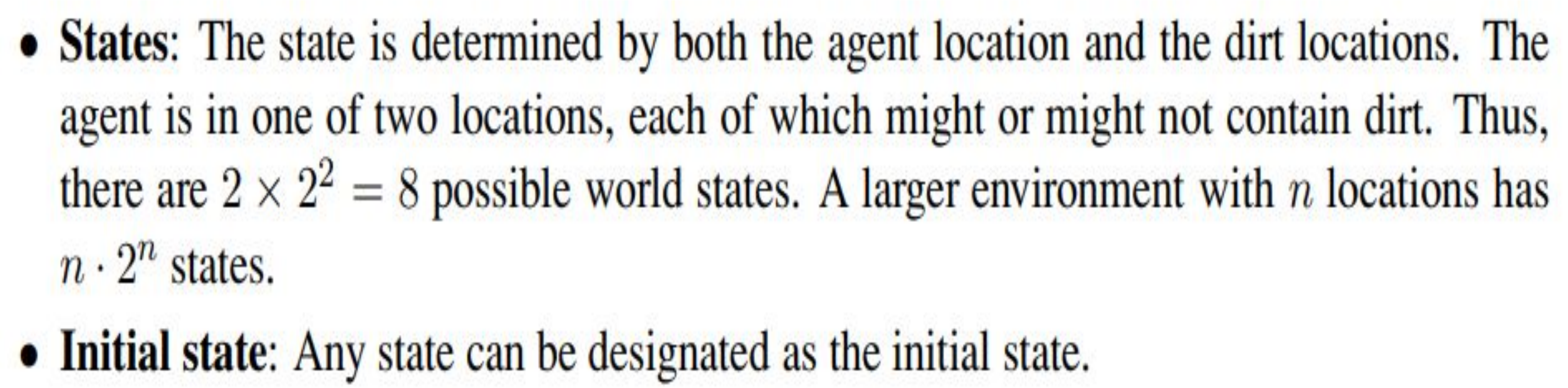
EXAMPLE PROBLEMS

Toy problems

- The first example we examine is the vacuum world.

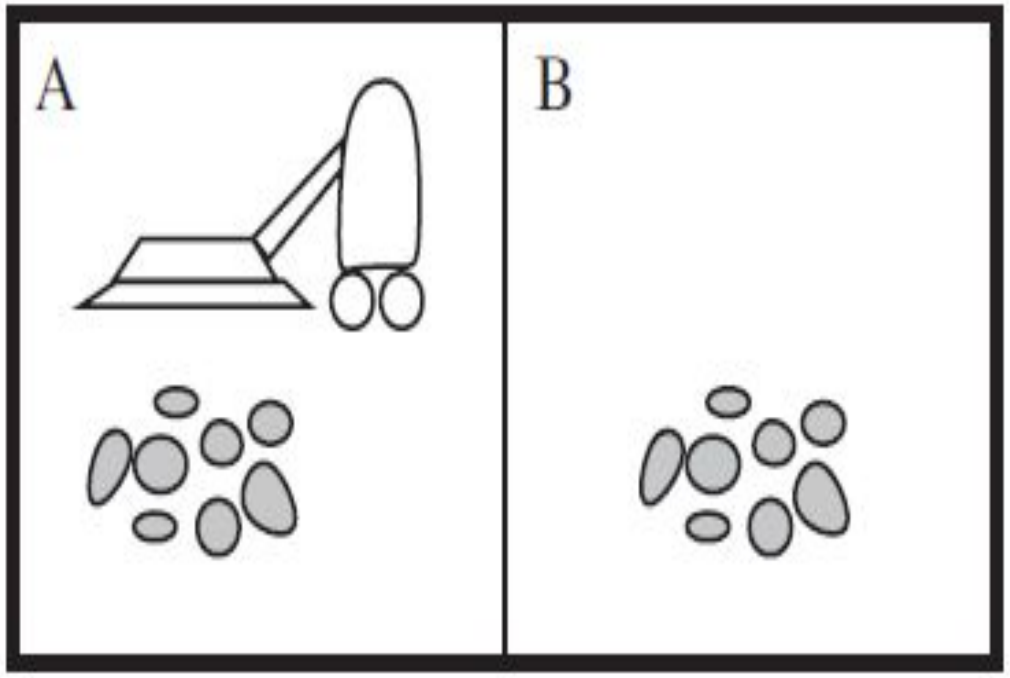


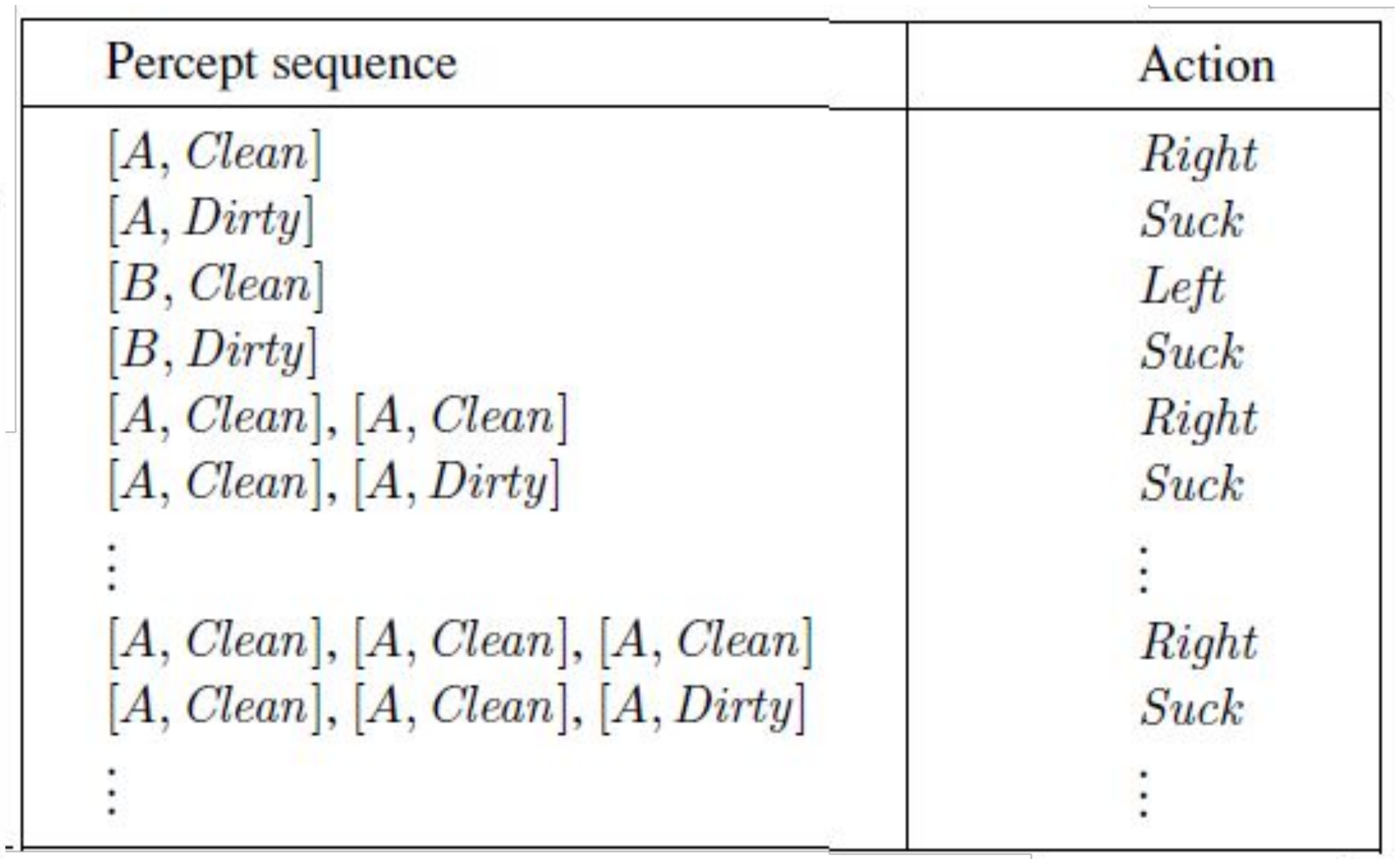
EXAMPLE PROBLEMS



- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ states.
- **Initial state**: Any state can be designated as the initial state.

A vacuum-cleaner world with just two locations





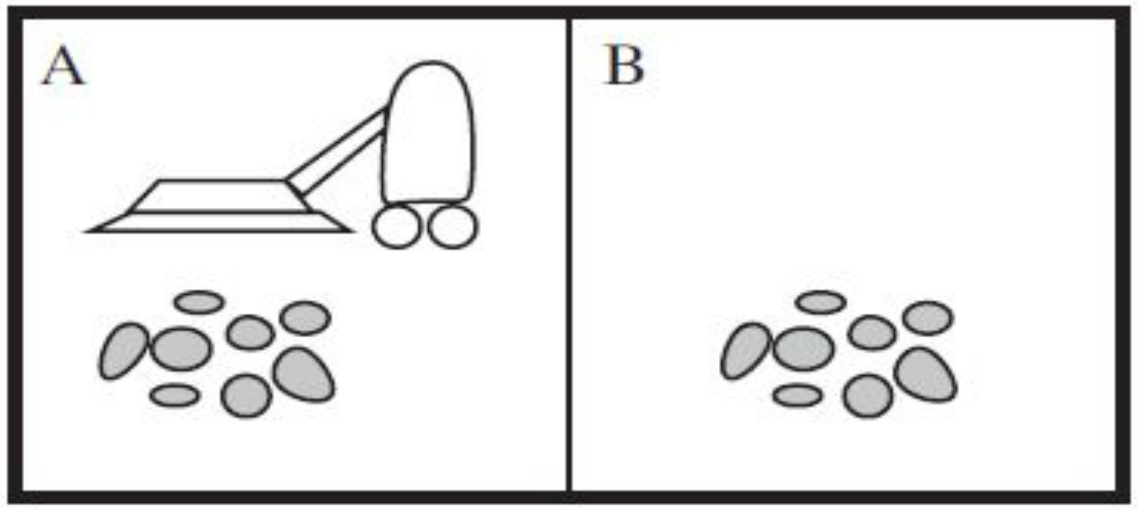
Percept sequence	Action
$[A, Clean]$	$Right$
$[A, Dirty]$	$Suck$
$[B, Clean]$	$Left$
$[B, Dirty]$	$Suck$
$[A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Dirty]$	$Suck$
$\vdots$	$\vdots$
$[A, Clean], [A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Clean], [A, Dirty]$	$Suck$
$\vdots$	$\vdots$

Partial tabulation of a simple agent function for the vacuum-cleaner world

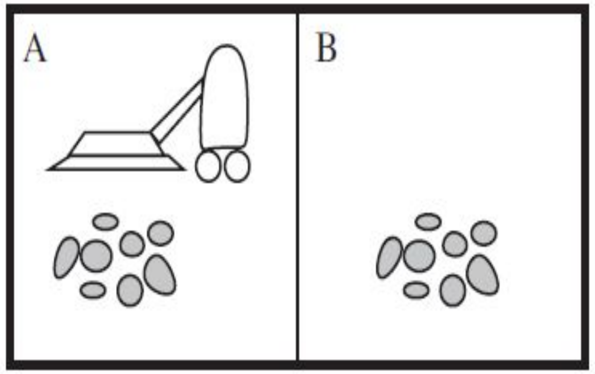
A vacuum-cleaner world with just two locations

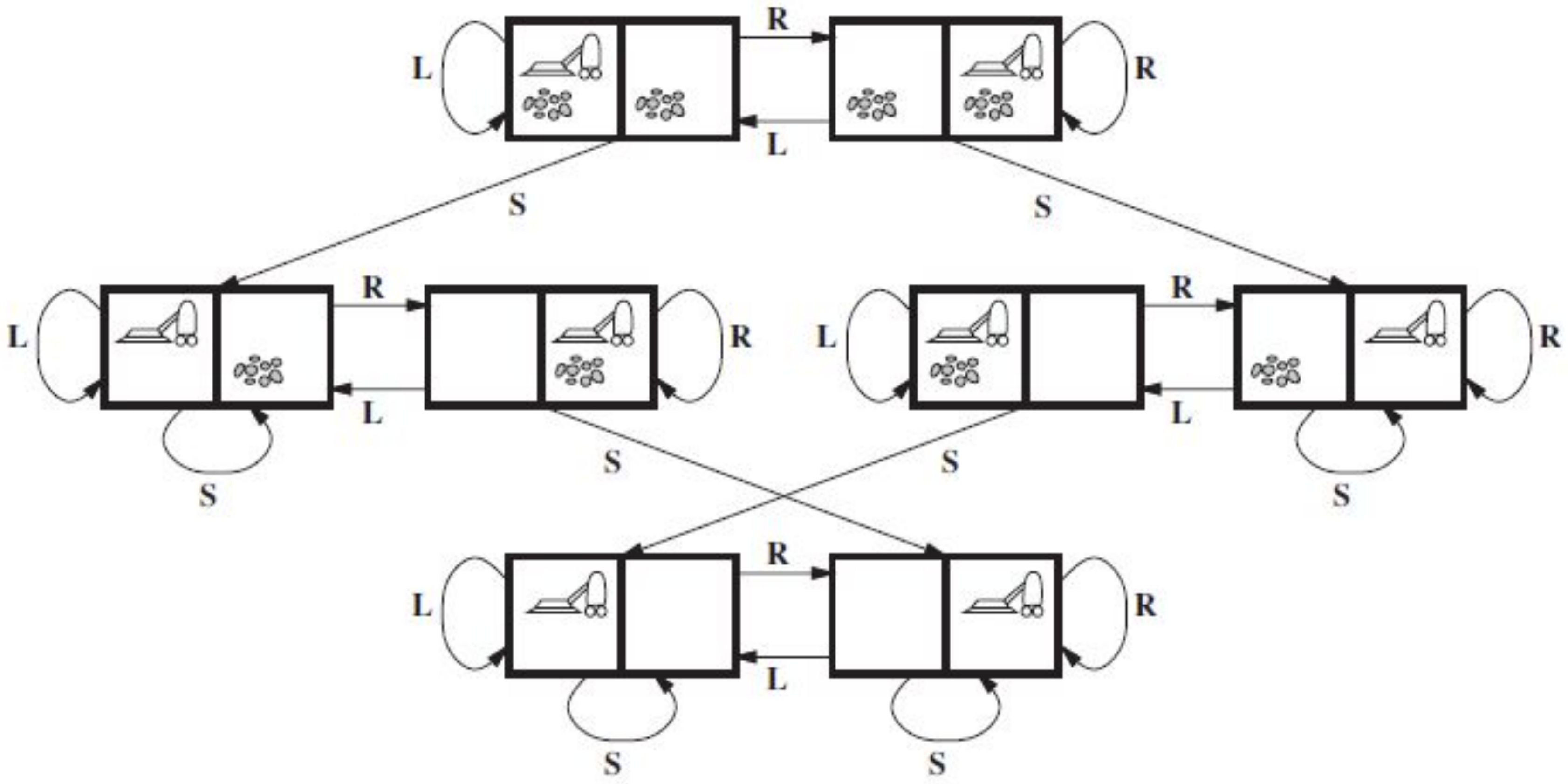
- The agent program for a simple reflex agent in the two-state vacuum environment.

```
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
```



A vacuum-cleaner world with just two locations





The state space for the vacuum world.
Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

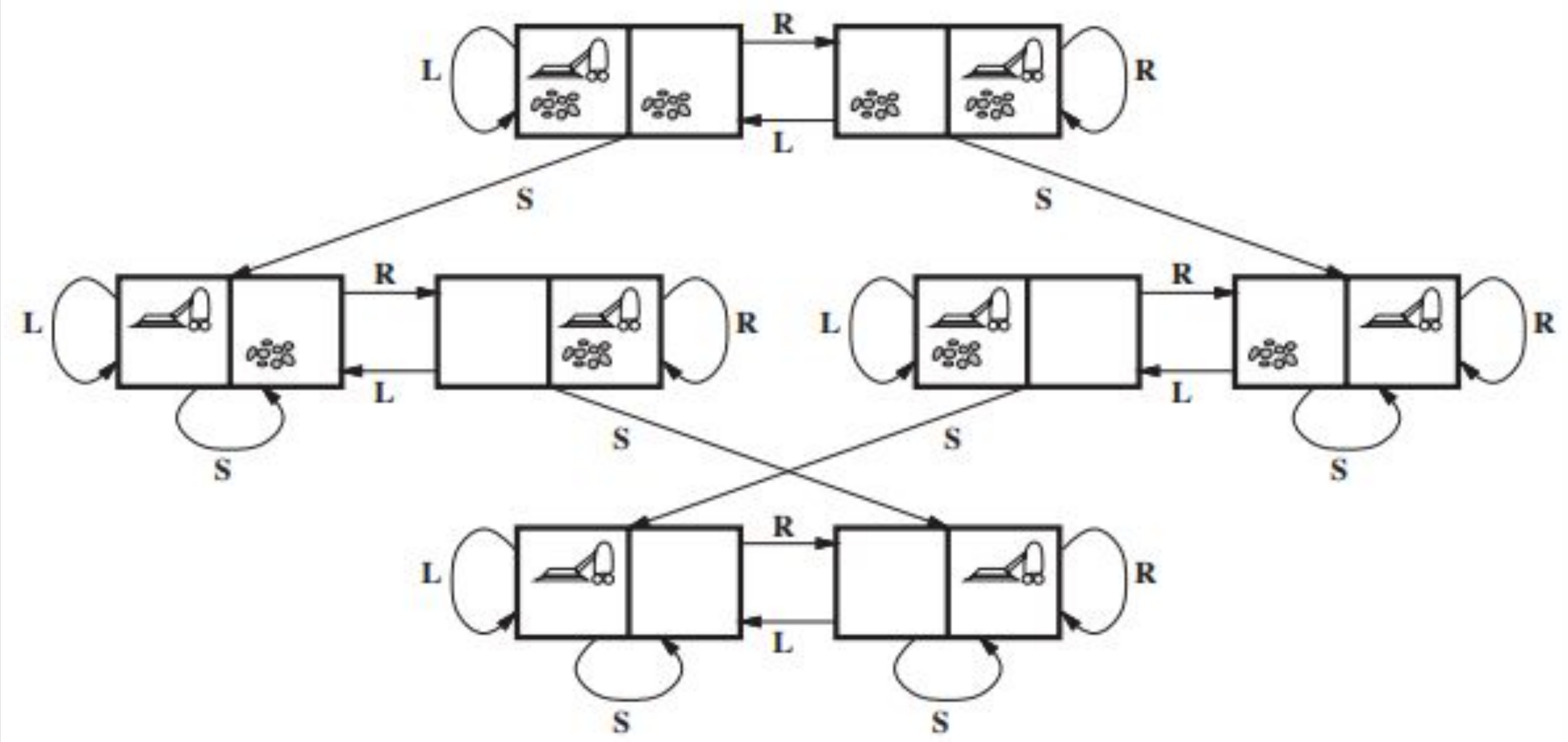
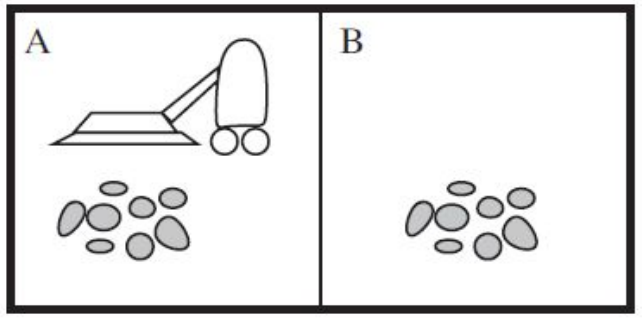


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

A vacuum-cleaner world with just two locations

- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Total possible states - $2 \times 2^2 = 8$.
  A larger environment with n locations has $n \cdot 2^n$ states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: Each state has just three actions: *Left*, *Right*, and *Suck*.
  Larger environments might also include *Up* and *Down*.
- **Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect. The complete state space is previous slide.
- **Goal test**: Checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.



- Path Coast =3

8-puzzle

- Consists of a 3×3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state.

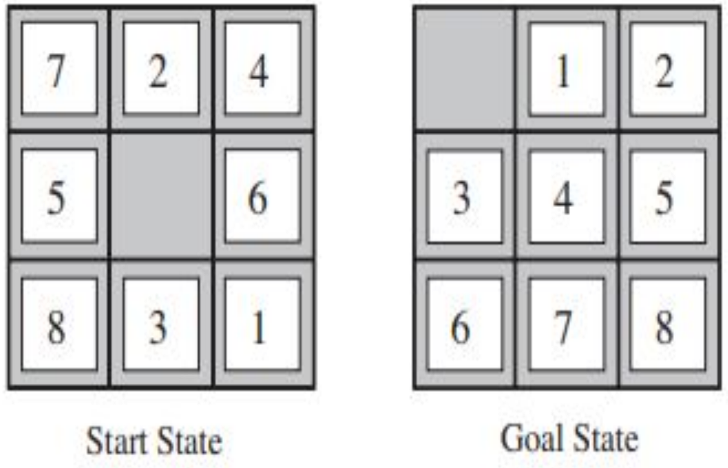


Figure 3.4 A typical instance of the 8-puzzle.

- **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states (Exercise 3.4).
- **Actions**: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
- **Transition model**: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

- The 8-puzzle belongs to the family of sliding-block puzzles, which are often used as test problems for new search algorithms in AI.
- The 8-puzzle has 9!/2 = 181, 440 reachable states and is easily solved.
- The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms.
- The 24-puzzle (on a 5 × 5 board) has around 1025 states, and random instances take several hours to solve optimally.

8-queens problem

- Place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.)

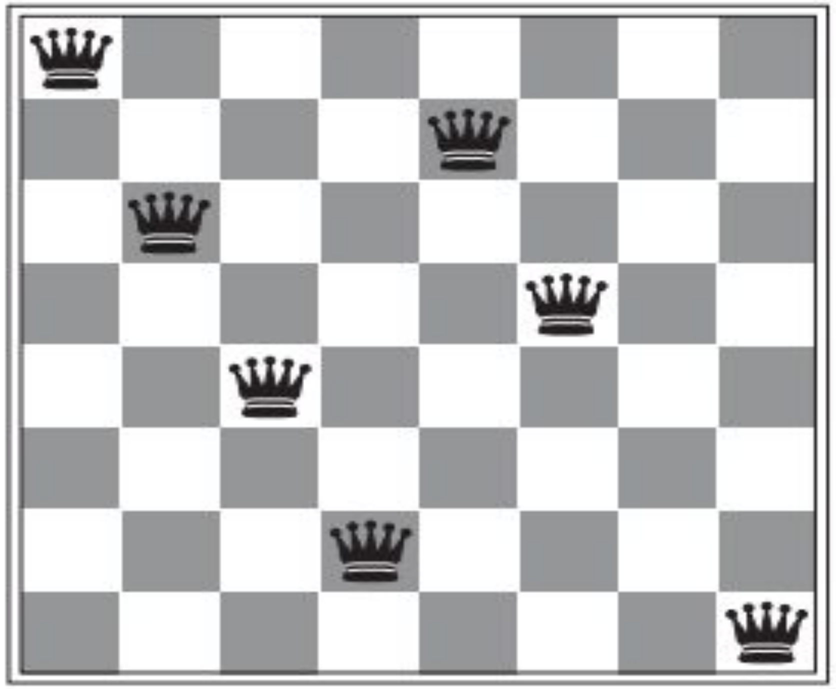


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

There are two main kinds of formulation

- An **incremental formulation**
  - involves operators that augment the state description starting from an empty state
  - Each action adds a queen to the state
  - States:
    - ✓any arrangement of 0 to 8 queens on board
  - Successor function:
  - add a queen to any empty square
- A **complete-state formulation**
  - starts with all 8 queens on the board
  - move the queens individually around
  - States:
    - ✓any arrangement of 8 queens, one per column in the leftmost columns
  - Operators: move an attacked queen to a row, not attacked by any other
- the right formulation makes a big difference to the size of the search space

Incremental formulation

- States: Any arrangement of 0 to 8 queens on the board is a state.
- Initial state: No queens on the board.
- Actions: Add a queen to any empty square.
- Transition model: Returns the board with a queen added to the specified square.
- Goal test: 8 queens are on the board, none attacked.

In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 1014$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

Complete state formulation

- Incremental formulation prohibiting placing a queen in any square that is already attacked:
- **States**: All possible arrangements of n queens ($0 \le n \le 8$), one per column in the leftmost n columns, with no queen attacking another.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
- **Transition model**: Returns the board with a queen added to the specified square.
- • **Goal test**: 8 queens are on the board, none attacked.
- This formulation reduces the 8-queens state space from 1.8×1014 to just 2,057, and solutions are easy to find.

How infinite state spaces can arise?

- Our final toy problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise.
- Knuth conjectured that, starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer.
- For example, we can reach 5 from 4 as follows:

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

- The problem definition is very simple:
- States: Positive numbers.
- Initial state: 4.
- Actions: Apply factorial, square root, or floor operation (factorial for integers only).
- Transition model: As given by the mathematical definitions of the operations.
- Goal test: State is the desired positive integer.

- To our knowledge there is no bound on how large a number might be constructed in the process of reaching a given target—for example, the number 620,448,401,733,239,439,360,000 is generated in the expression for 5—so the state space for this problem is infinite.
- Such state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.

Route-finding problem – Airline Travel Problem

- **States**: Each state obviously includes a location (e.g., an airport) and the current time.

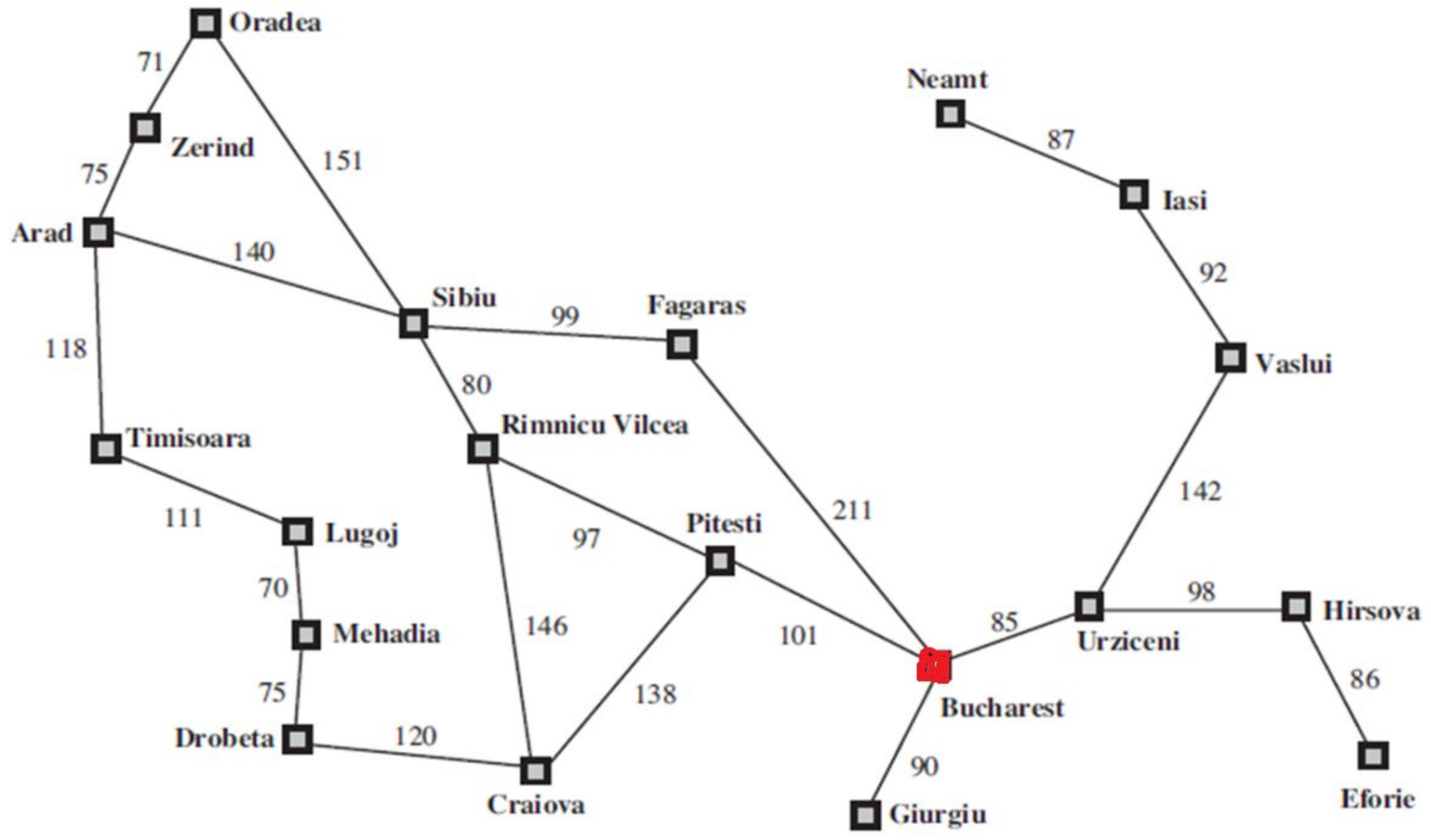
  Further, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.

- **Initial state**: This is specified by the user's query.
- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.

Route-finding problem – Airline Travel Problem

- **Transition model**: The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.
- **Goal test**: Are we at the final destination specified by the user?
- **Path cost**: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

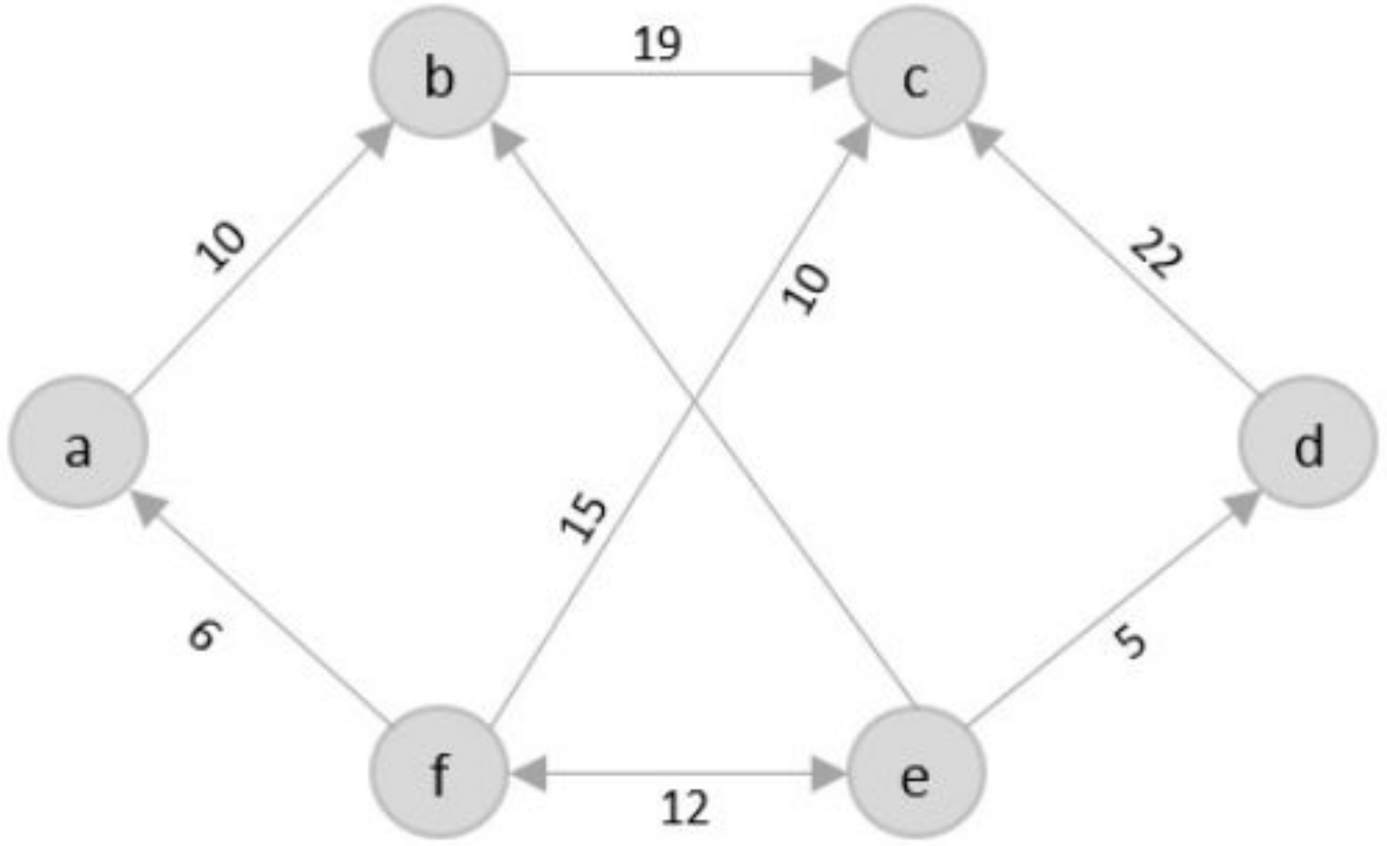
Touring Problem



The problem:
"Visit every city in Figure at least once, starting and ending in Bucharest."

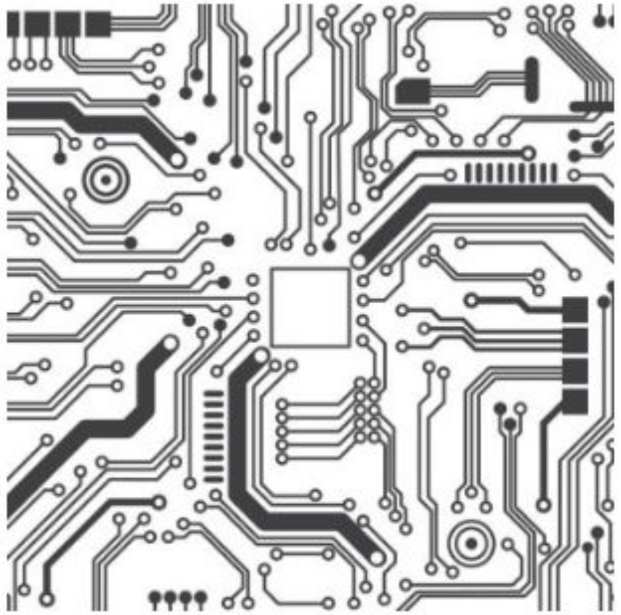
Real-world problems

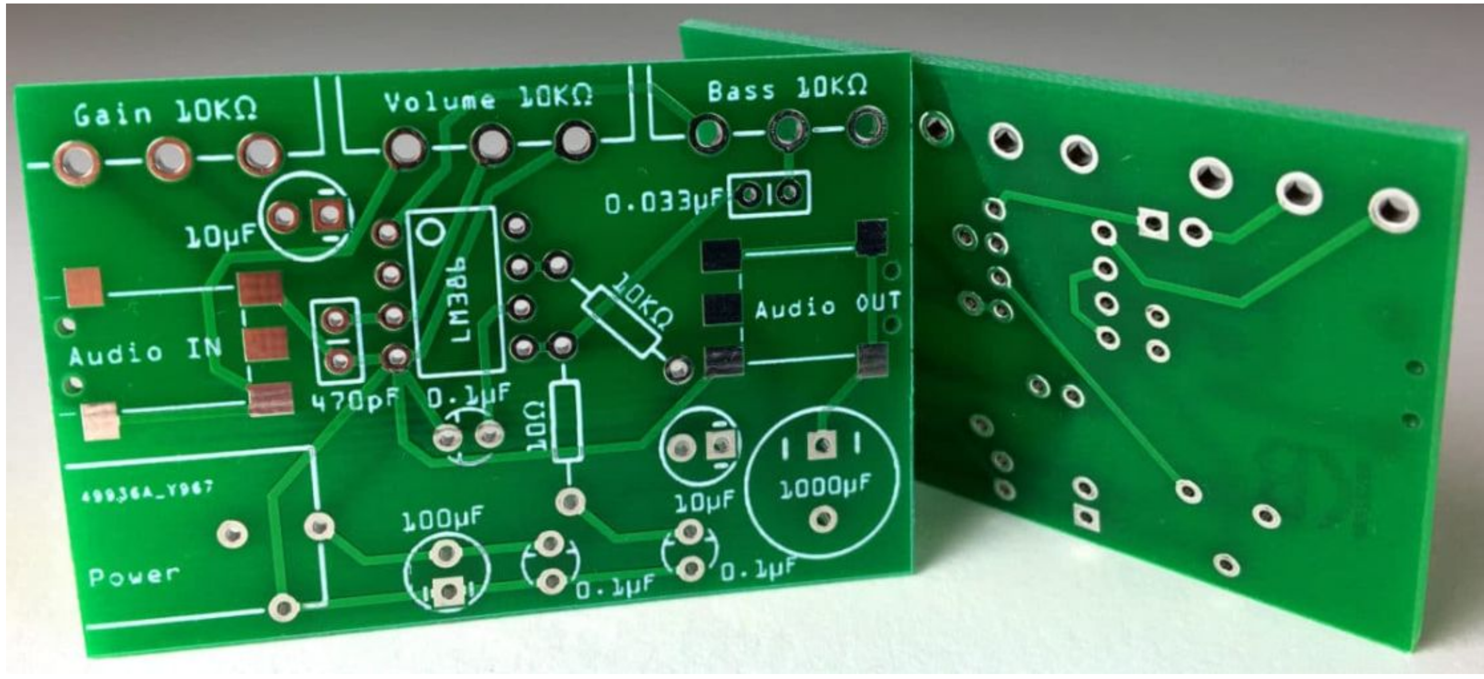
- Route-finding problem
- Touring problem
- Travelling Salesperson problem



Real-world problems

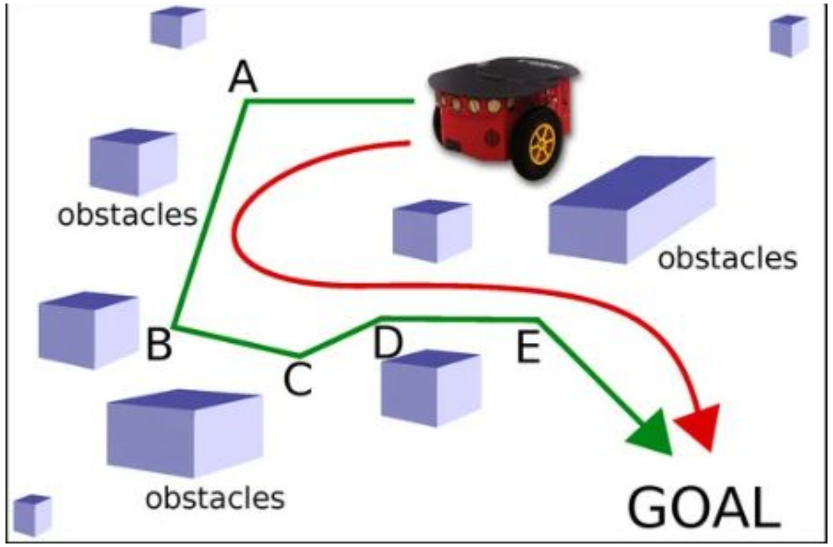
- Route-finding problem
- Touring problem
- Travelling Salesperson problem
- VLSI Layout problem

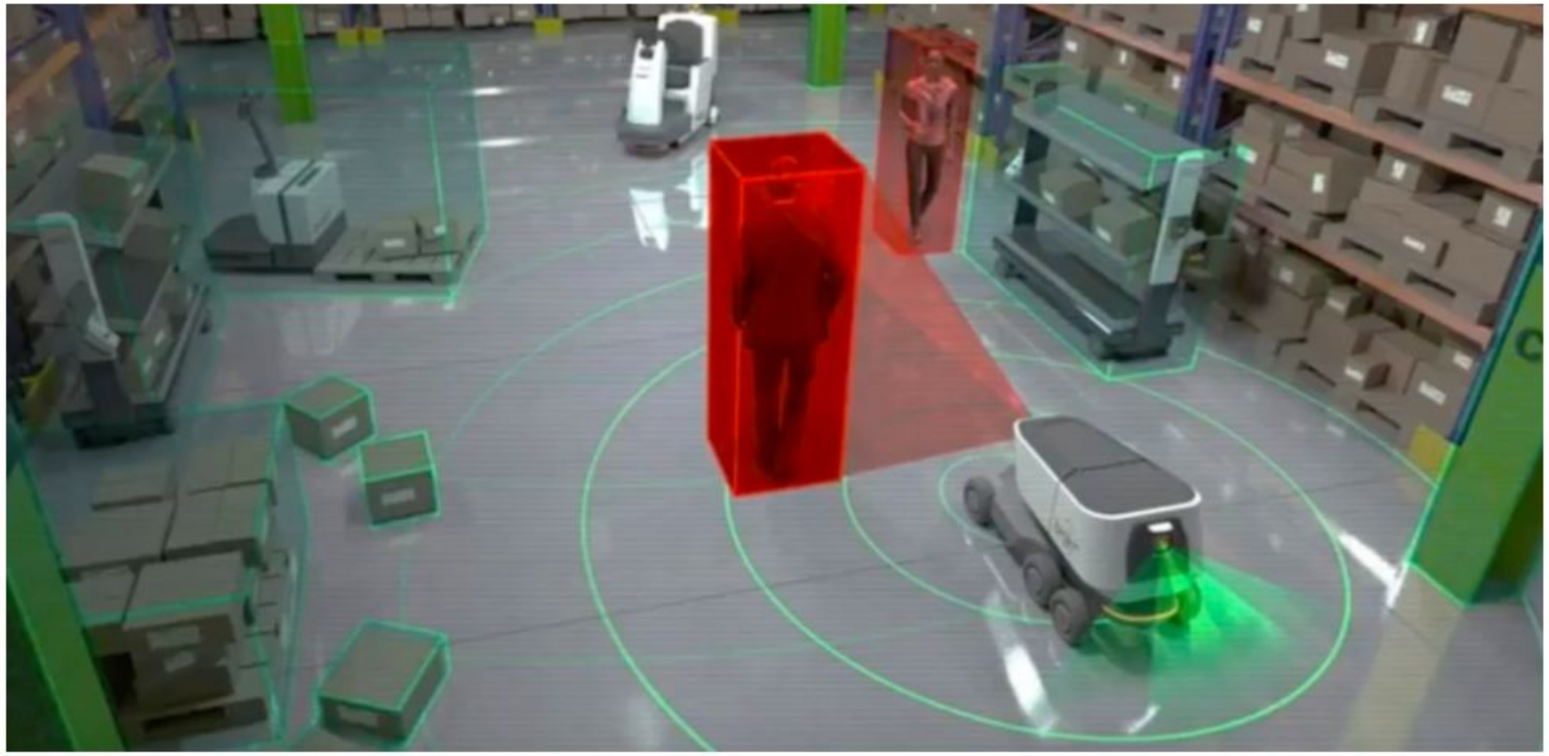




Real-world problems

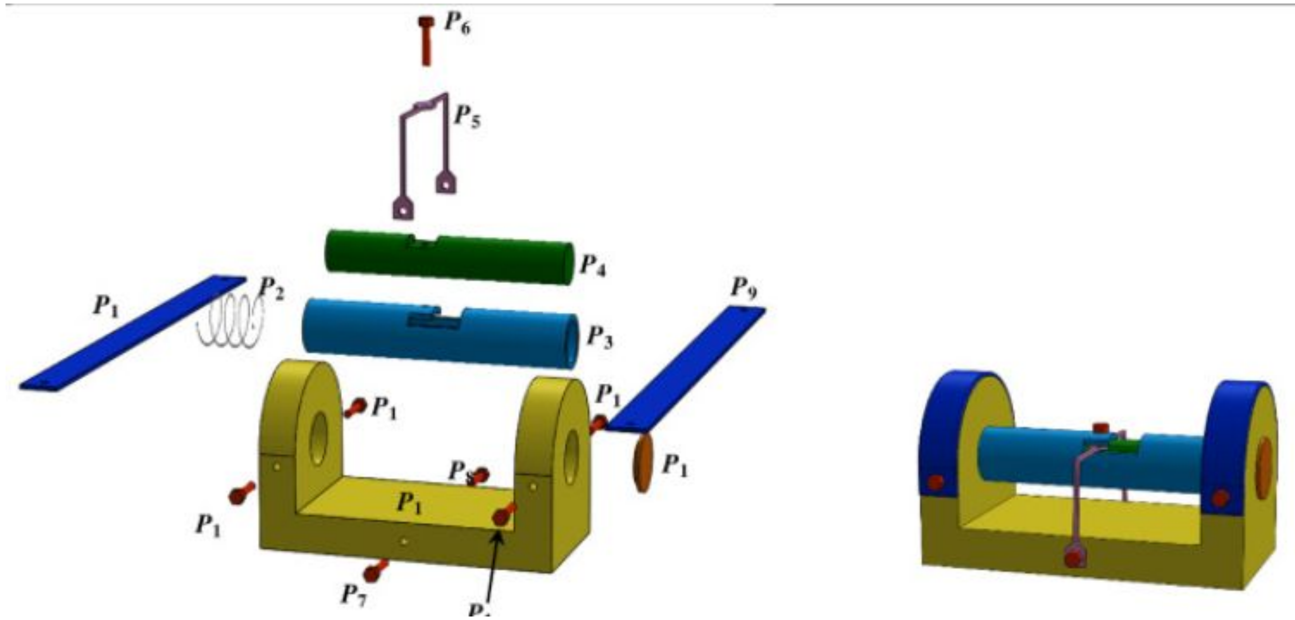
- Route-finding problem
- Touring problem
- Travelling Salesperson problem
- VLSI Layout problem
- Robot Navigation problem





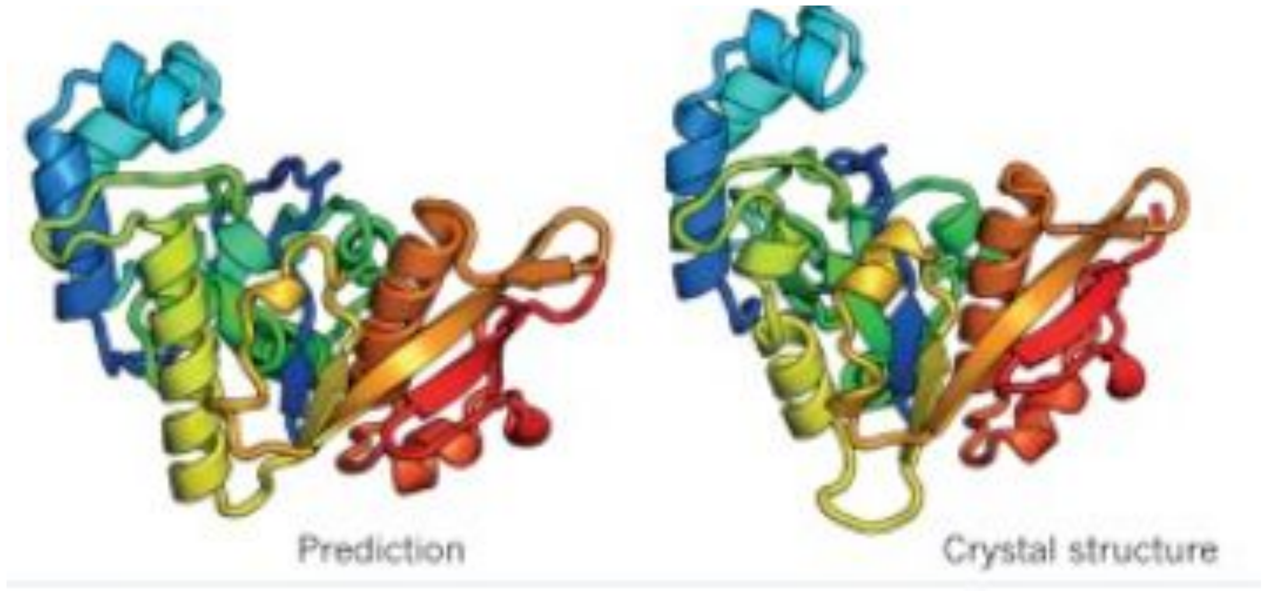
Real-world problems

- Route-finding problem
- Touring problem
- Travelling Salesperson problem
- VLSI Layout problem
- Robot Navigation problem
- Automatic Assembly Sequencing problem



Real-world problems

- Route-finding problem
- Touring problem
- Travelling Salesperson problem
- VLSI Layout problem
- Robot Navigation problem
- Automatic Assembly Sequencing problem
- Protein design problem



Real-world problems

- Route-finding problem
- Touring problem
- Travelling Salesperson problem
- VLSI Layout problem
- Robot Navigation problem
- Automatic Assembly Sequencing problem
- Protein design problem