UNIT – 1

Chapter – 1

Introduction

- What is AI?
- Formal definitions of AI
  - Acting Humanly
  - Thinking Humanly
  - Thinking Rationally
  - Acting Rationally
- Abilities/Qualities of AI Machine
  - NLP
  - Knowledge base
  - Automated Reasoning
  - Machine learning
  - Computer vision
  - Robotics
- What is Machine Learning?

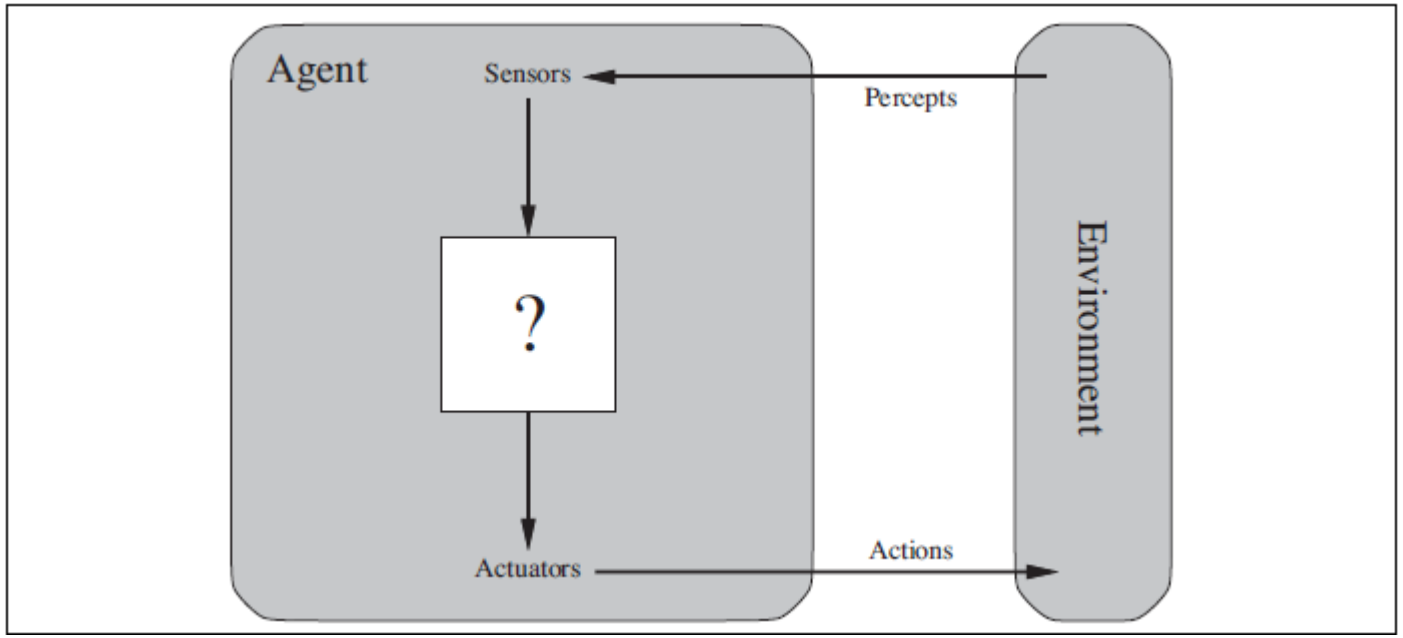
UNIT – 1

Chapter – 2

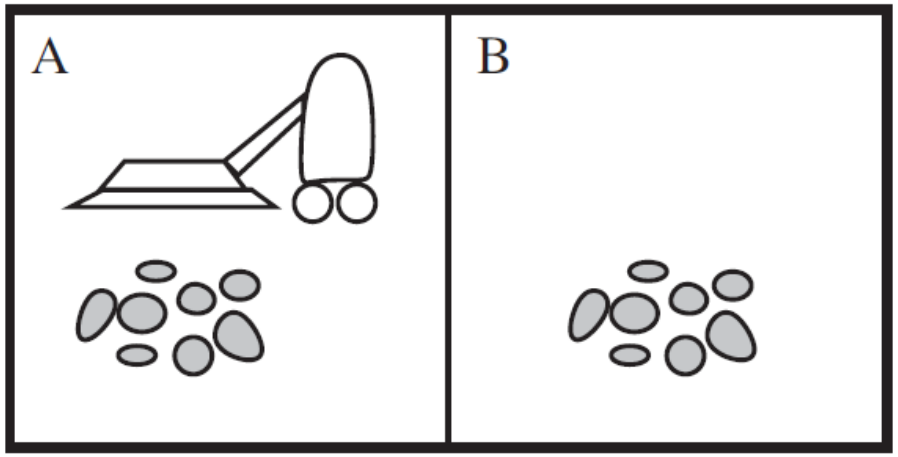
Artificial Intelligent Agents

Artificial Intelligent Agents

- An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**.
- The term **percept** refer to the agent's perceptual inputs at any given instant
- Mathematically, an agent's behavior is described by the **agent function** that maps any given percept sequence to an action



Ex: Vacuum Cleaner



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

Good Behavior: The Concept of Rationality

A **rational agent** is one that does the right thing

(a). Rationality:

✓What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

✓This leads to a **definition of a rational agent**:

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

(b). Omniscience, learning, and autonomy

The Nature of Environments

✓**Specifying the task environment :** **R**epresent the environment with PEAS description

P : Performance

E : Environment

A : Actuators

S : Sensors

Ex: Driverless car

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

The Nature of Environments
Cntd...

Examples of agent types and their PEAS descriptions

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

The Nature of Environments Cntd..

✓Properties of task environments :

- **Fully observable** vs. **partially observable**
- **Single agent** vs. **multiagent**
- **Deterministic** vs. **Stochastic**
- **Episodic** vs. **Sequential**
- **Static** vs. **dynamic**
- **Discrete** vs. **Continuous**
- **Known** vs. **unknown**

The Nature of Environments Cntd..

Examples of task environments and their characteristics

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

The Structure of the Agent

Discussed agents via their behaviour
The action that is performed after any given sequence of percepts.

Now we talk about insides work.
The goal of AI is to design an **agent program** that implements the agent function

i.e. percepts $\longrightarrow$ actions.

This program will run on some sort of computing device with physical sensors and actuators which is called as the **architecture**:

Thus, *agent = architecture + program* .

Types of Agent

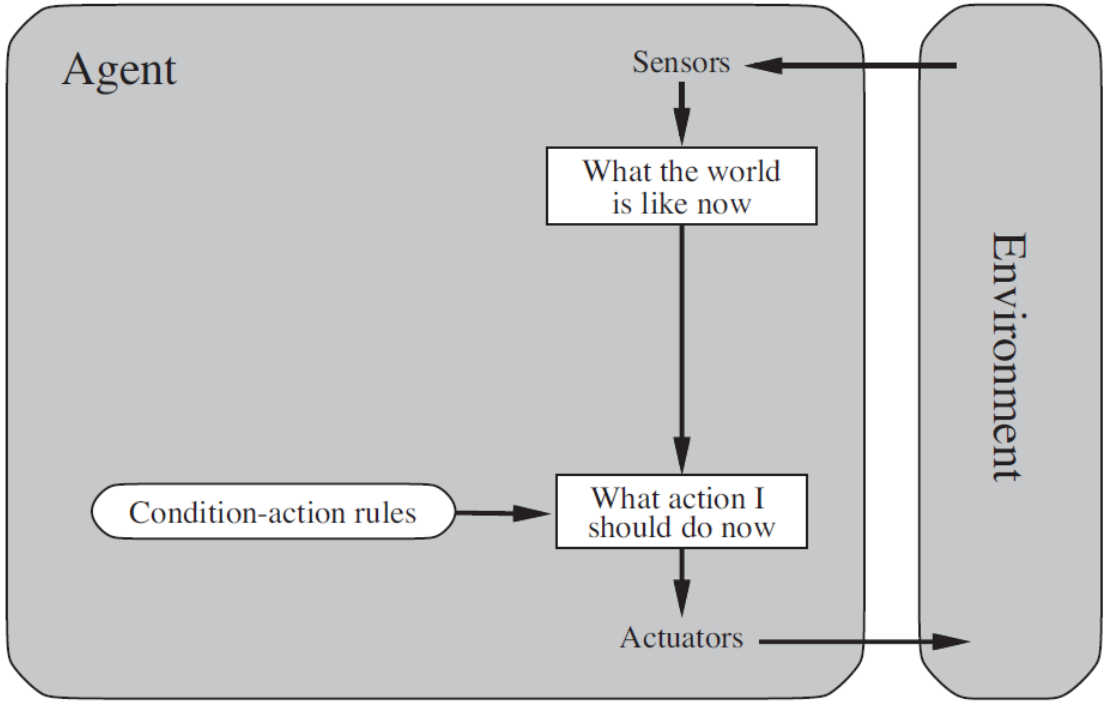
Four basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

1. Simple reflex agents;
2. Model-based reflex agents;
3. Goal-based agents; and
4. Utility-based agents.

1. Simple reflex agent

These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history.

It works in fully observable environment
It functions based on if-then-else rules



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*

Simple reflex agent example

The agent program for a simple reflex agent in the two-state vacuum environment. This program implements the agent function tabulated

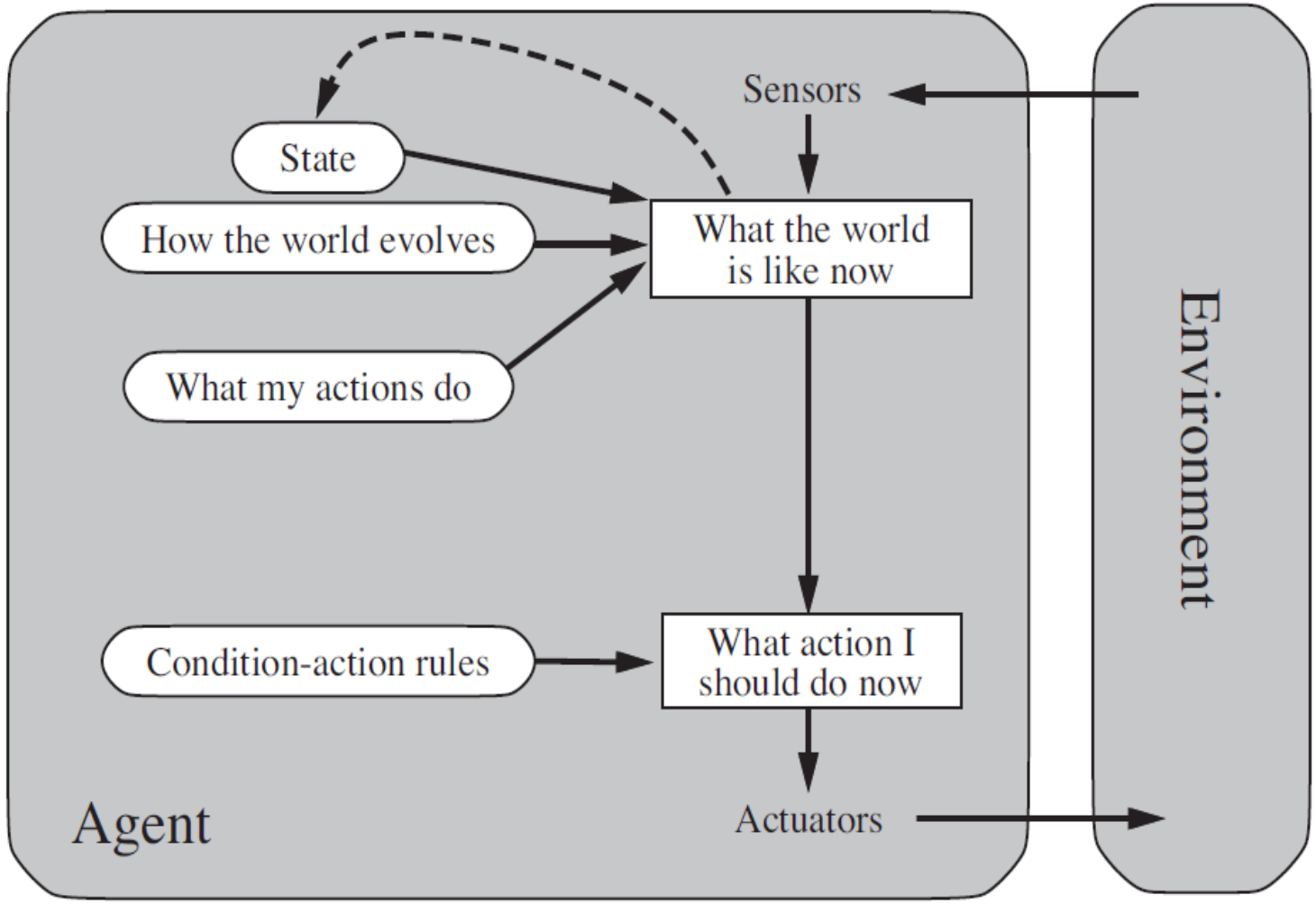
function REFLEX-VACUUM-AGENT([*location*,*status*]) **returns** an action

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

2. Model-based reflex agent

These agents select actions on the basis of the both *current* percept and percept history.

It works in partially observable environment



2. Model-based reflex agent Cntd...

A model-based reflex agent keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent

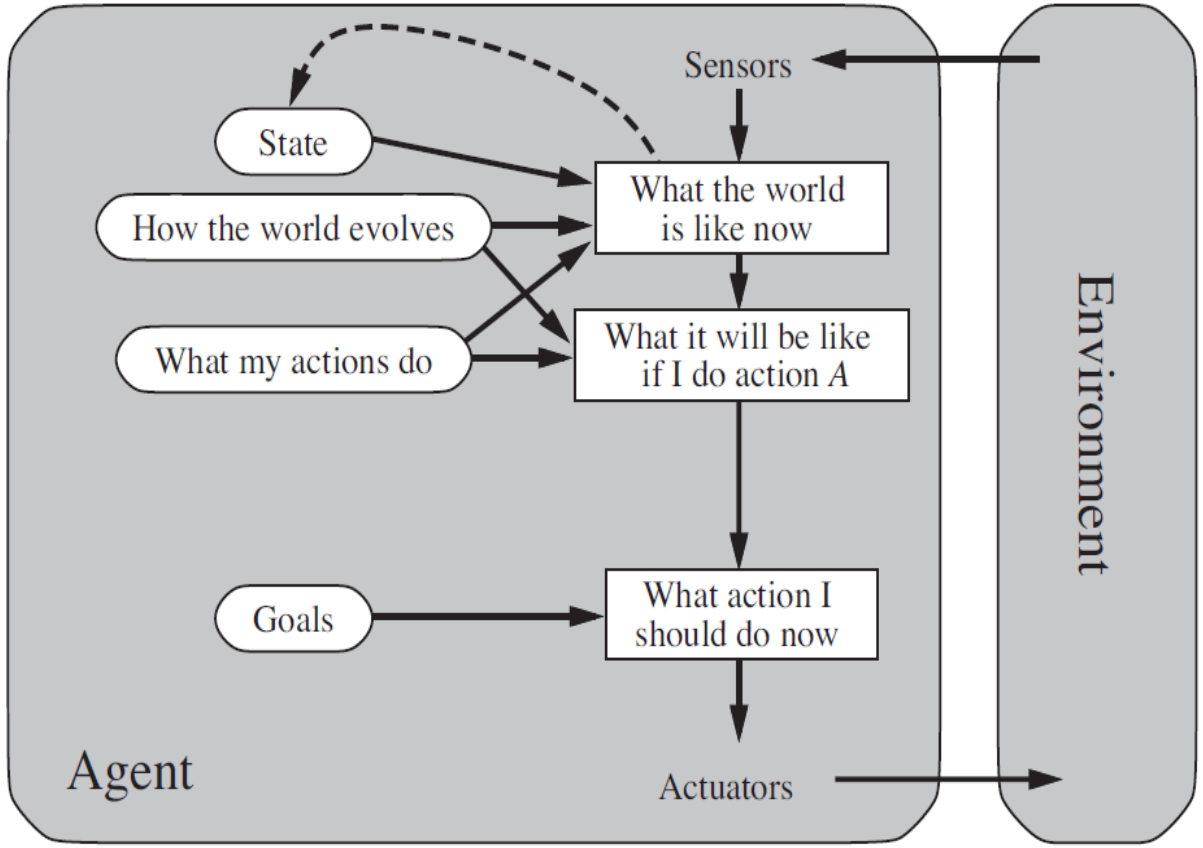
function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action

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*

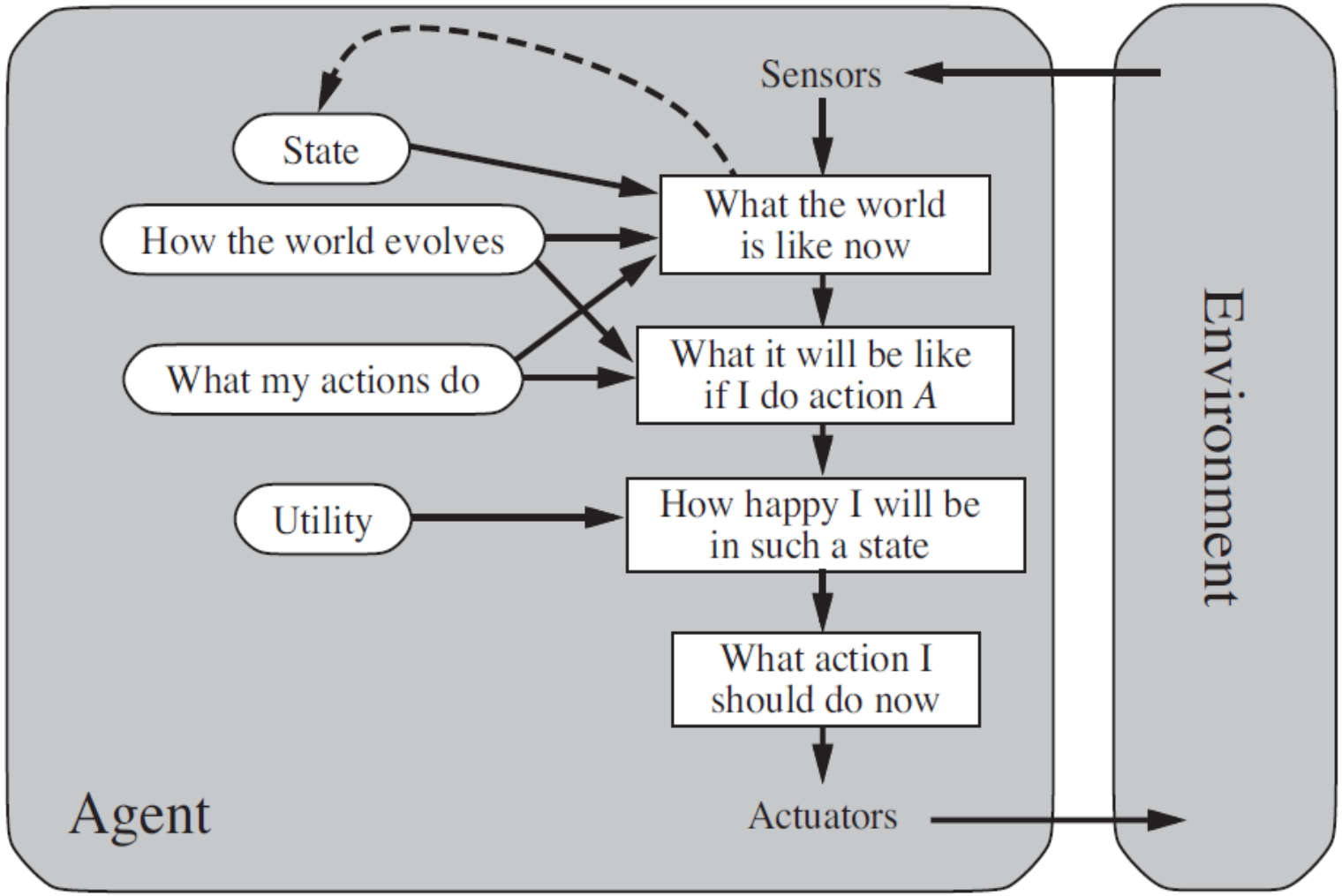
3. Goal-based agent

- Expansion of Model-based.
- Along with the current state description, the agent needs some sort of **goal** information that describes situations that are desirable
- Searching and planning are the subfields of AI devoted to finding action sequences that achieve the agent's goals



4. Utility-based agent

- Expansion of Model-based.
- Goals alone are not enough to generate high-quality behavior in most environments
- Agent can be in “happy” or “Unhappy” state based on the action performed



Learning agent

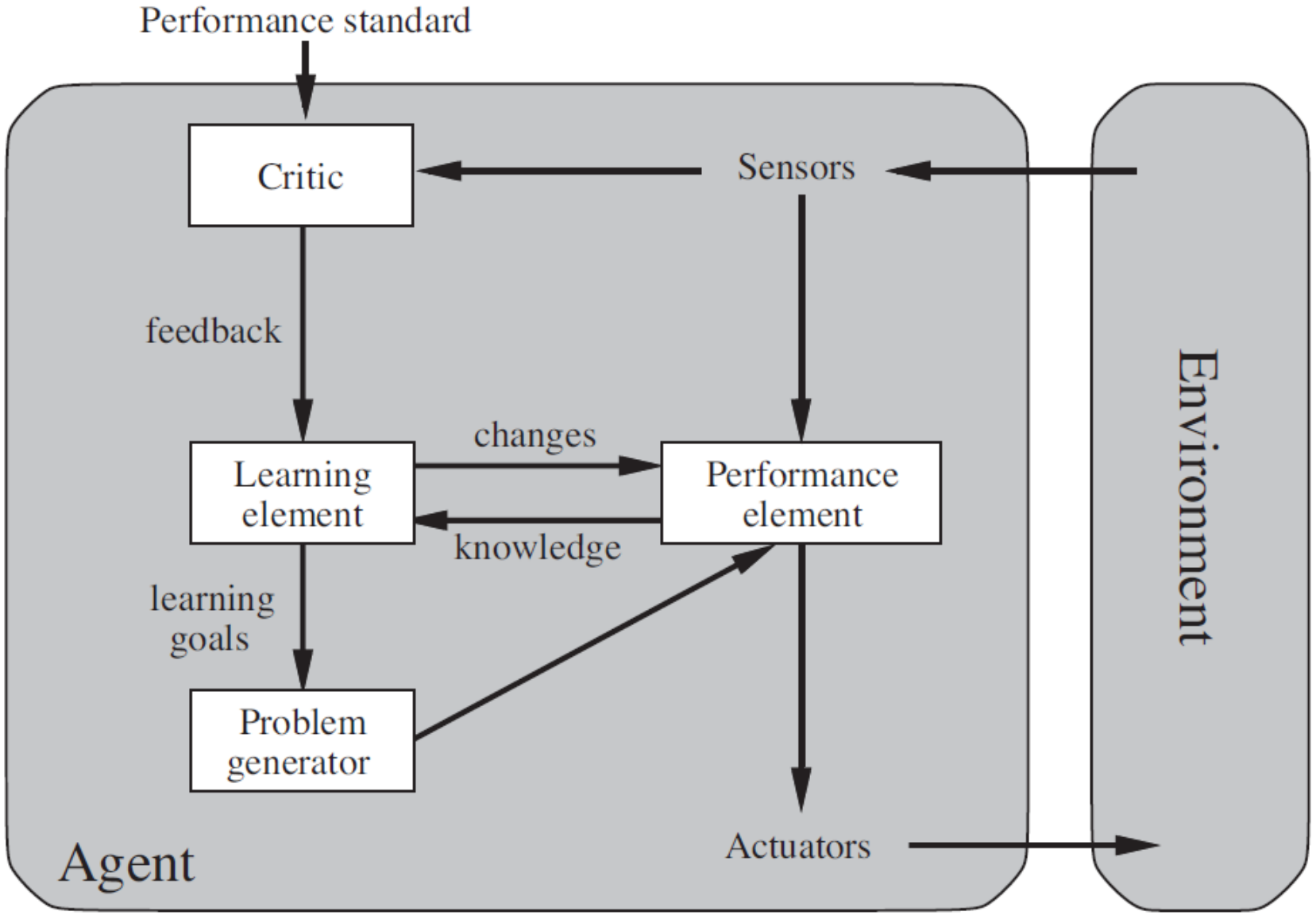
- Learning element : Is responsible for making improvements in agent. It gets feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.

- Performance element: It responsible for selecting external actions i.e. it is an agent (Maps percepts to actions)

- Critic: Feedback generator

- Problem generator : It is responsible for suggesting actions that will lead to new and informative experiences. The problem generator might identify certain areas of behavior in need of improvement and suggest experiments, such as trying out the brakes on different road surfaces under different conditions

Learning agent



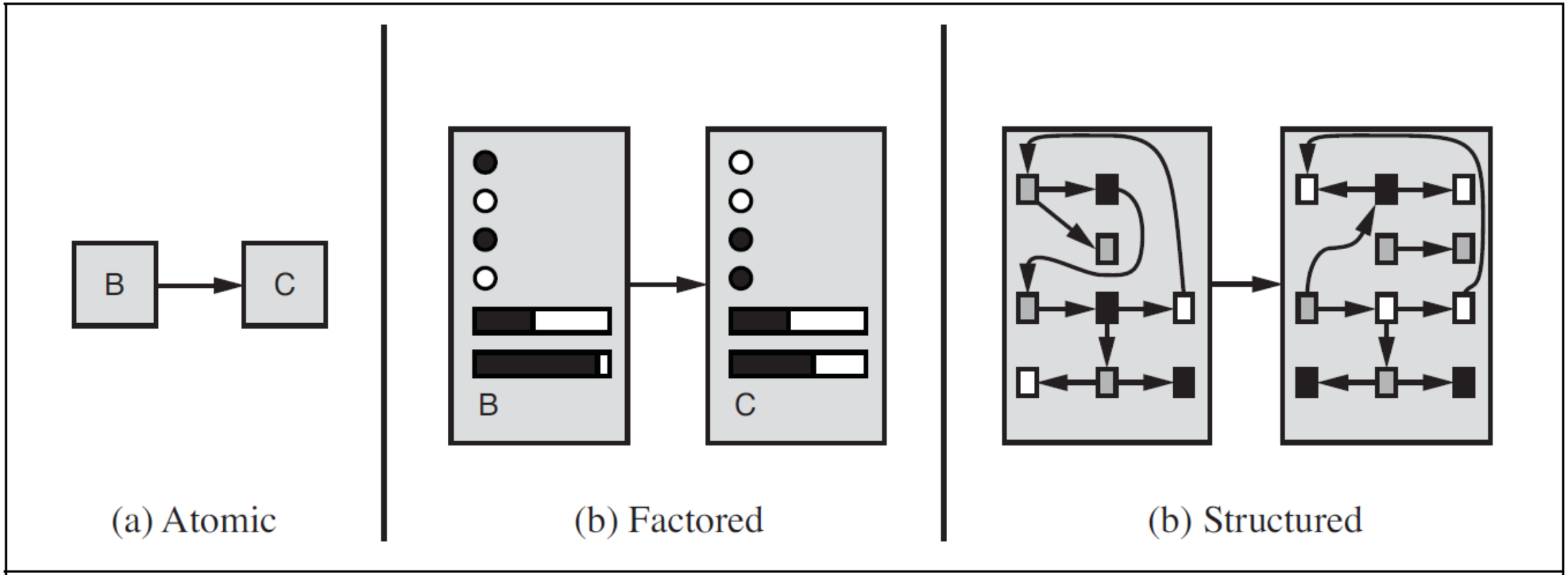
How the components of agent programs work

We described agent programs as consisting of various components, whose function is to answer questions such as:

- What is the world like now?
- What action should I do now?
- What do my actions do?

- These programs can be structured based on how they represent and reason about the world.
- The *three* main types of state representations in agent programs are
    1. Atomic
    2. Factored
    3. structured.
- Each of these representations impacts how the agent perceives, processes, and acts on the environment.

How the components of agent programs work Cntd...



How the components of agent programs work Cntd…

Atomic Representation :

- The state of the world is represented as a single, indivisible unit, often without any internal structure or details.
- Each state is a unique, distinct entity, and the agent does not decompose or analyze it further.
- These agents treat each state as a black box and make decisions purely based on the atomic state.

•**Example**: Chess game where the agent only considers the current position of pieces on the board and the possible moves.

How the components of agent programs work Cntd...

Factored Representation :

- The state of the world is described by a set of variables or factors, each of which can take on different values.
- These variables represent different attributes or aspects of the world.
- The agent can analyze these individual factors to make more informed decisions.

- **Example**: In a car driving scenario, the state of the environment can be described by variables like the car's speed, the traffic light status, the road conditions, and the positions of other vehicles. Each of these is a factor, and the agent's decision is based on the combination of these variable values.

How the components of agent programs work Cntd…

Structured Representation :

- The state is described as a set of objects and the relationships between them.
- This allows for a much richer understanding of the world, including the ability to represent complex interactions between objects.
- It underlie relational databases and first-order logic, first-order probability models, knowledge-based learning and much of natural language understanding

- **Example**: A robot in a factory could represent its environment using structured representations, where objects like machines, tools, and workpieces are defined, and the relationships between them are explicitly modeled. This allows the agent to reason about complex relationships and perform tasks like assembling components based on these relationships.

UNIT – 1

Chapter – 3

Solving Problems by Searching

Problem-Solving Agents

- This chapter describes one kind of goal-based agent called a **problem-solving agent**.
- These 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**
- Intelligent agents are supposed to maximize their performance measure.
- Achieving this is sometimes simplified if the agent can adopt a **goal** and aim at satisfying it.
- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.

Problem-Solving Agents Contd..

- **Goal formulation,** based on the current situation and the agent's performance measure, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.
- **Unknown** environment i.e **i**f the agent has no additional information, then it has no choice but to try one of the actions at random.
- 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

Problem-Solving Agents Contd..

- **Assumptions:**
- ***Observable*** **-**The agent always knows the current state
- ***Discrete*** **-** At any state there are finite number of actions to choose
- ***Known*** **-** The agent knows, in which state it will be after performing specific action
- ***Deterministic*** **:** Each action will have axactctly one outcome

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

Problem-Solving Agents Contd..

- 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.
- Once a solution is found, the actions it recommends can be carried out, and is called the execution phase.
- Thus, we have a simple "**formulate, search, execute**" design for the agent.

"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."

A Simple Problem-Solving Agent Function

```
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
```

Well defined Problems and Solutions

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

1. **Initial state :** that the agent starts in
   Eg: In(Arad).

2. **Actions : Possible actions available to the agent, given a particular state s,** ACTIONS(s) returns the set of actions that can be executed in s.
   Eg: From the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.

3. **Transition model** : A description of what each action does; 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.
   Eg: we have RESULT(In(Arad),Go(Zerind)) = In(Zerind) .

Well defined Problems and Solutions Contd..

4. **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.

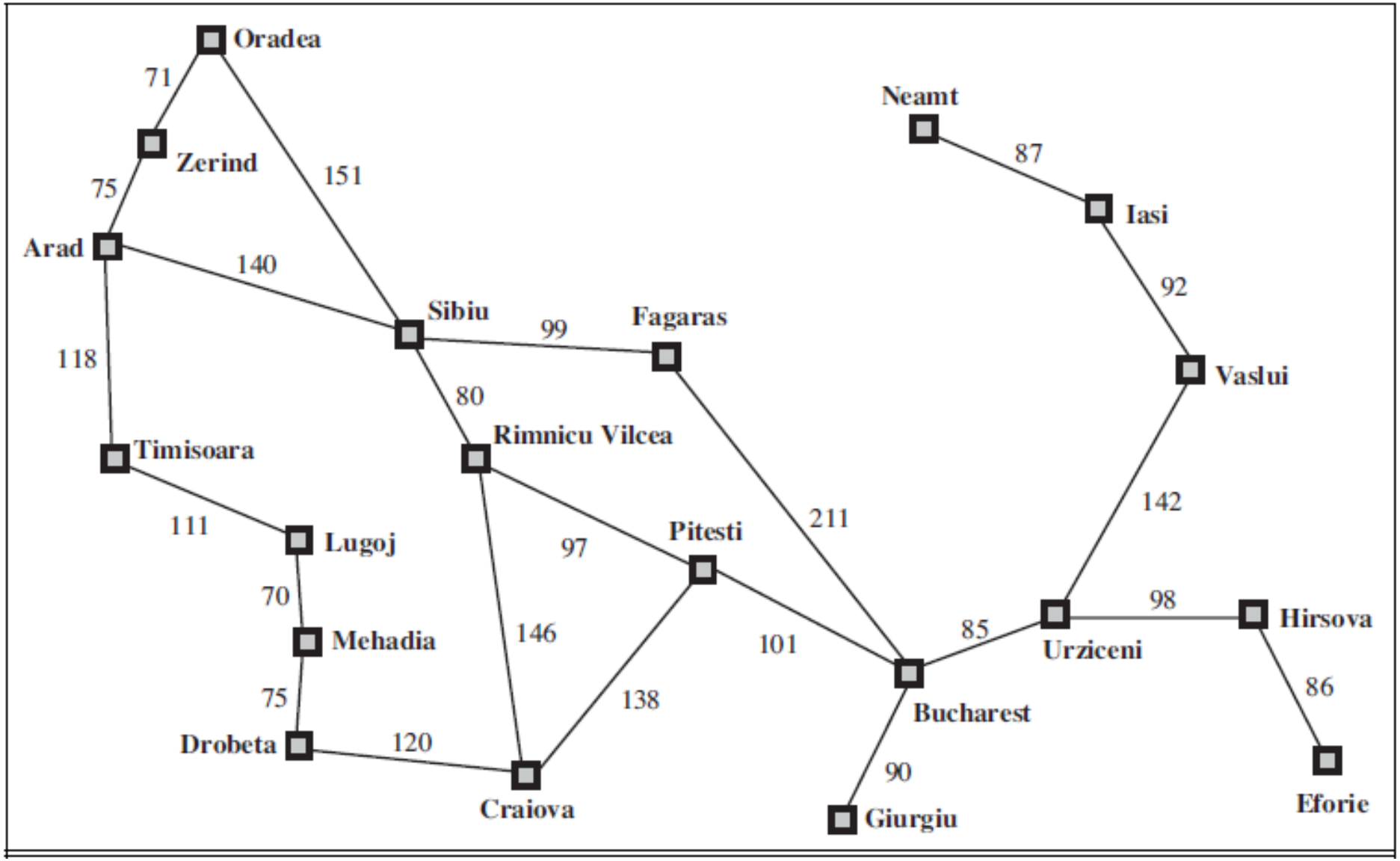
   Eg: The agent's goal in Romania is the singleton set {In(Bucharest)}.

5. **Path cost:** It is a function that assigns a numeric cost to each path.

   The step cost of taking action *'a'* in state *'s'* to reach state *'s^l'* is denoted by $C(s, a, s^l)$

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.**

Well defined Problems and Solutions Contd..



Toy Problem : Vacuum Cleaner

The **vacuum cleaner world** formulated as a problem as follows:

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 x 2n) states.

Initial state: Any state can be designated as the initial state.

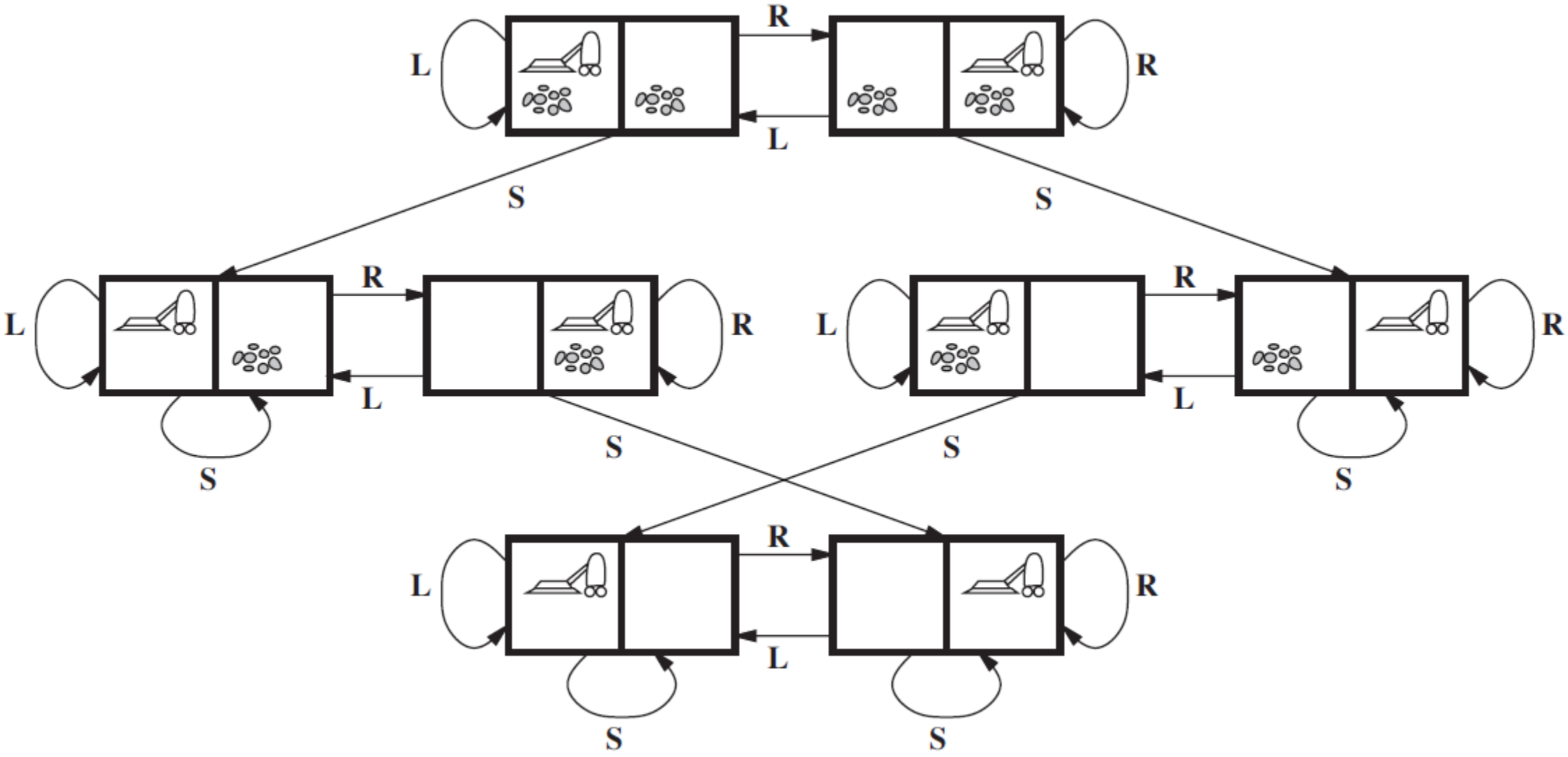
Actions: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.

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.

Goal test: This checks whether all the squares are clean.

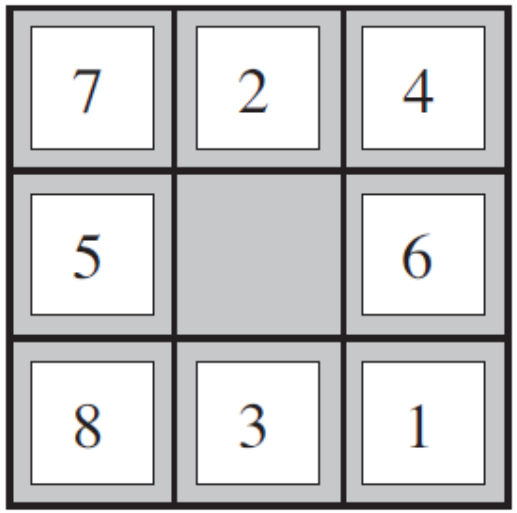
Path cost: Each step costs 1, so the path cost is the number of steps in the path.

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

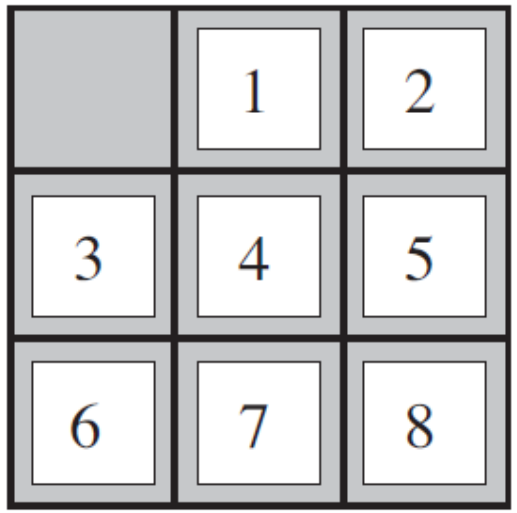


Toy Problem : 8-Puzzle

The **8-puzzle**, an instance of which is shown in Figure 3.4, 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, such as the one shown on the right of the figure. The standard formulation is as in following slide:



Start State



Goal State

Toy Problem : 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.

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.

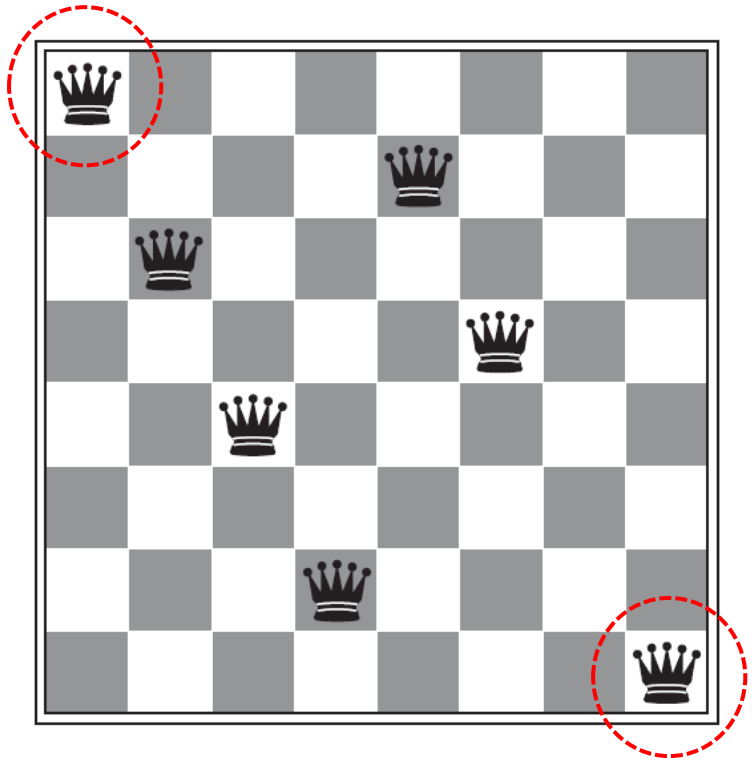
Transition model: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in above figure, the resulting state has the 5 and the blank switched.

Goal test: This checks whether the state matches the goal configuration shown in Figure. (Other goal configurations are possible.)

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

Toy Problem : 8-Queens Problem

The goal of the **8-queens problem** is to 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 below shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.



Toy Problem : 8-Queens Problem

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.

Toy Problem : A problem devised by Donald Knuth

The 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.

Eg. 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.

Real-world Problem : Route Finding Problem

Consider the airline travel problems that must be solved by a travel-planning Web site:

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

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.

Real-world Problem : Touring (Travel) Problem

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem "Visit every city in Figure 3.2 at least once, starting and ending in Bucharest." As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the *set of cities the agent has visited*. So the initial state would be In(Bucharest), Visited({Bucharest}), a typical intermediate state would be In(Vaslui), Visited({Bucharest , Urziceni , Vaslui}), and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

Real-world Problem : Traveling Salesperson Problem

The **traveling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

Real-world Problem : VLSI Layout Problem

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually split into two parts: **cell layout** and **channel routing**.

In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

Real-world Problem : Robot Navigation Problem

Robot navigation is a generalization of the route-finding problem described earlier. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite. In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls.

Real-world Problem : Automatic Assembly Sequencing

Automatic assembly sequencing of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972). In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the ***generation of legal actions*** is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space.

Another important assembly problem is **protein design**, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

Searching for Solutions

- Solution to the formulated problem is sequence of actions.
- Hence the agent should search for the action sequence
- Searching is done by considering various possible combinations of action sequence
- The possible action sequences starting at initial state form a **search tree** with initial state at the root
- The branches are actions and the **nodes** correspond to states in the state space of the problem.
- **Expand** the current state by considering any of the leaf node by applying various possible actions
- The set of all leaf nodes available for expansion at any given point is called the **frontier**.
- Loopy paths are a special case of the more general concept of **redundant paths**, which exist whenever there is more than one way to get from one state to another.

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 GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  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
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Infrastructure for the Search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

- n.STATE : The state in the state space to which the node corresponds
- n.PARENT : The node in the search tree that generated this node
- n.ACTION : The action that was applied to the parent to generate the node
- n.PATH-COST : The cost, traditionally denoted by *g(n)*, of the path from the initial state to the node, as indicated by the parent pointers.

Infrastructure for the Search algorithms

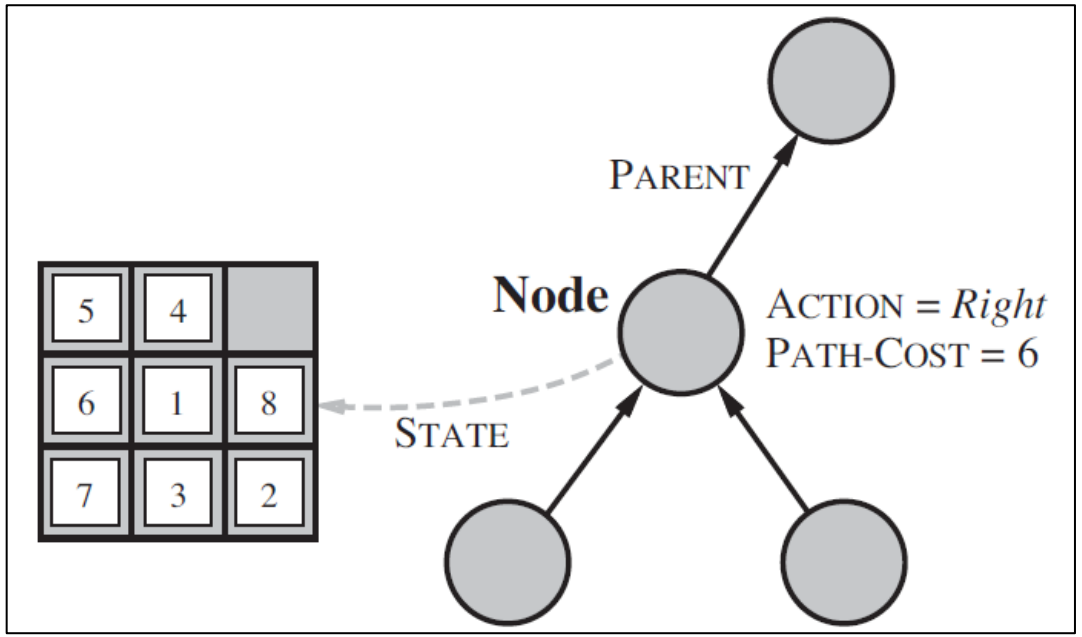
function CHILD-NODE(*problem*, *parent*, *action*) **returns** a node

  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*)



Measuring problem-solving performance

Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. We can evaluate an algorithm's performance in four ways:

Completeness : Is the algorithm guaranteed to find a solution when there is one?

Optimality : Does the strategy find the optimal solution

Time complexity : How long does it take to find a solution?

Space complexity : How much memory is needed to perform the search?