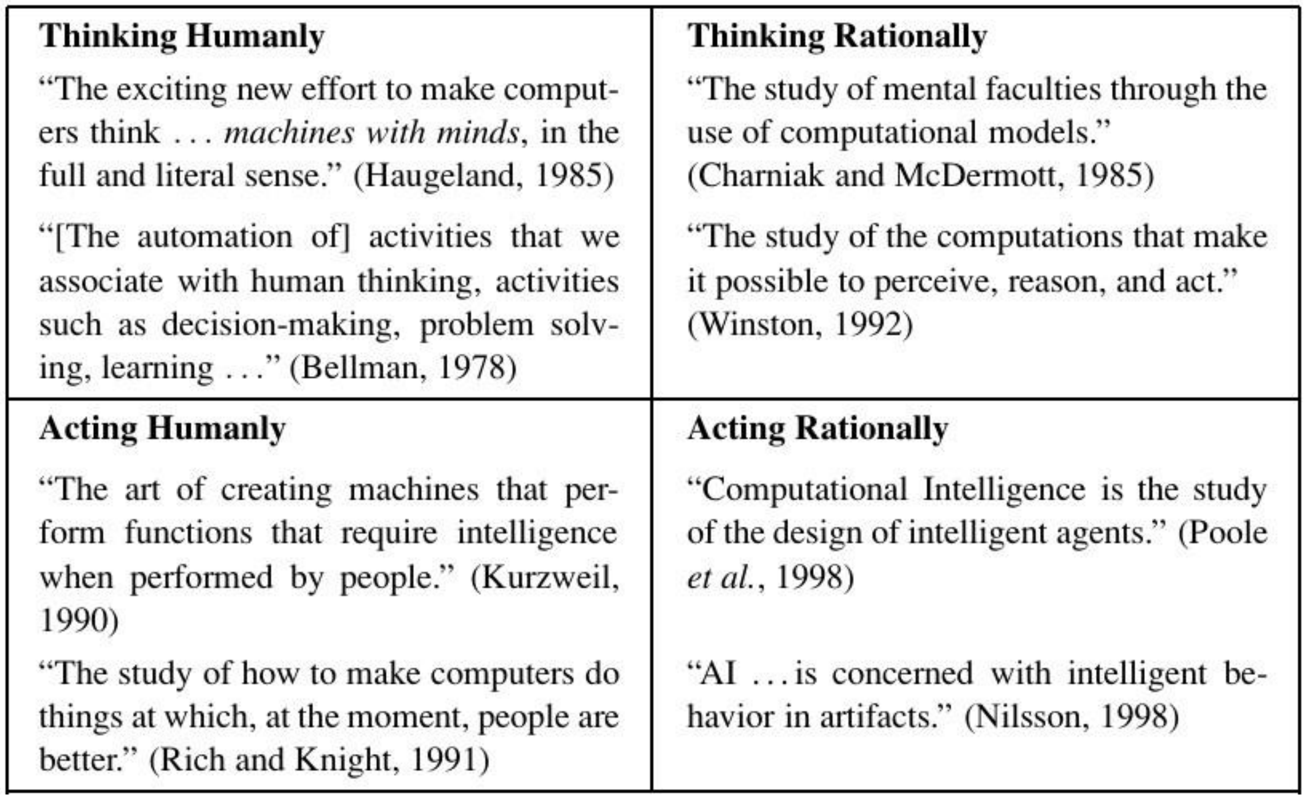
MODULE 1

INTRODUCTION

WhatisAI?

The definitions of AI:



Thinking Humanly	Thinking Rationally
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Figure 1: Some definitions of artificial intelligence, organized into four categories.

The definitions on the top, **(a)** and **(b)** are concerned with **thought processes and reasoning**, whereas those on the bottom**, (c)** and **(d)** address **behavior.** The definitions on the left, **(a)** and**(c)** measure success in terms of **human performance**, and those on the right, **(b)** and**(d)** measure the ideal concept of intelligence called **rationality**.

1)Acting humanly: The Turing Test approach

The Turing Test, proposed by **Alan Turing** (1950), was designed to provide a satisfactory operational definition of intelligence.

A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer.

The computer would need to possess the following capabilities:

- **natural language processing** to enable it to communicate successfully in English
- **knowledge representation** to store what it knows or hears
- **automated reasoning** to use the stored information to answer questions and to draw new conclusions
- **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

To pass the total Turing Test, the computer will need

- **computer vision** to perceive objects, and
- **robotics** to manipulate objects and move about.

2) Thinking humanly: The cognitive modeling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think.

There are three ways to do this:

- **introspection** - trying to catch our own thoughts as they go by;
- **psychological** experiments - observing a person in action;
- **brain imaging** - observing the brain in action.

Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program.

This can be achieved through **cognitive science**

Cognitive AI discuss the theories about how human mind works– for example, theories about how we recognize faces and other objects, or about how we solve abstract problem.

3) Thinking Rationally: The "Laws of thought" approach

- The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking," that is, irrefutable reasoning processes.
- His syllogisms provided patterns for argument structures that always yielded correct

conclusions when given correct premises.

- **example,** “Socrates is a man; all men are mortal; therefore, Socrates is mortal.”
- These laws of thought were supposed to govern the operation of the mind; their study initiated the field called logic.

4)Acting Rationally: The Rational Agent approach

- An agent is just something that acts: operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals.
- A **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

INTELLIGENT AGENTS

Agents and Environments

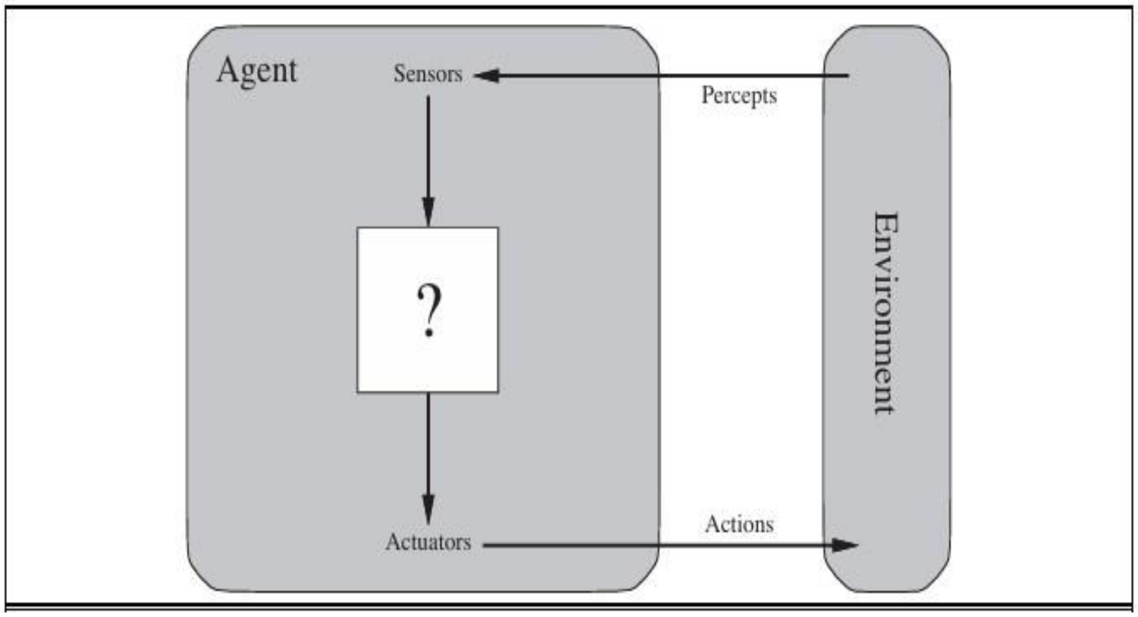


Figure 1.1: Agents interact with environments through sensors and actuators.

Agent:

An Agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

An agent can be:

- A **human agent** has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- A **robotic agent** might have cameras and infrared range finders for sensors and various motors for actuators.
- A **software agent** receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

Percept:

We use the term percept to refer to the agent's perceptual inputs at any given instant.

Percept Sequence:

An agent's percept sequence is the complete history of everything the agent has ever perceived.

Agent function:

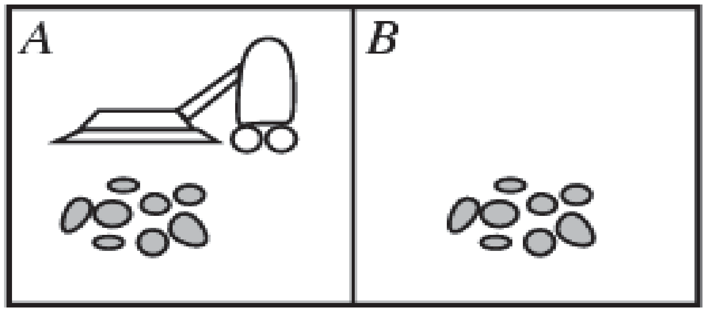
Mathematically speaking, we say that an agent's behavior is described by the agent function that maps any **given percept sequence to an action**.

Agent program

Internally, the agent function for an artificial agent will be implemented by an agent program. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

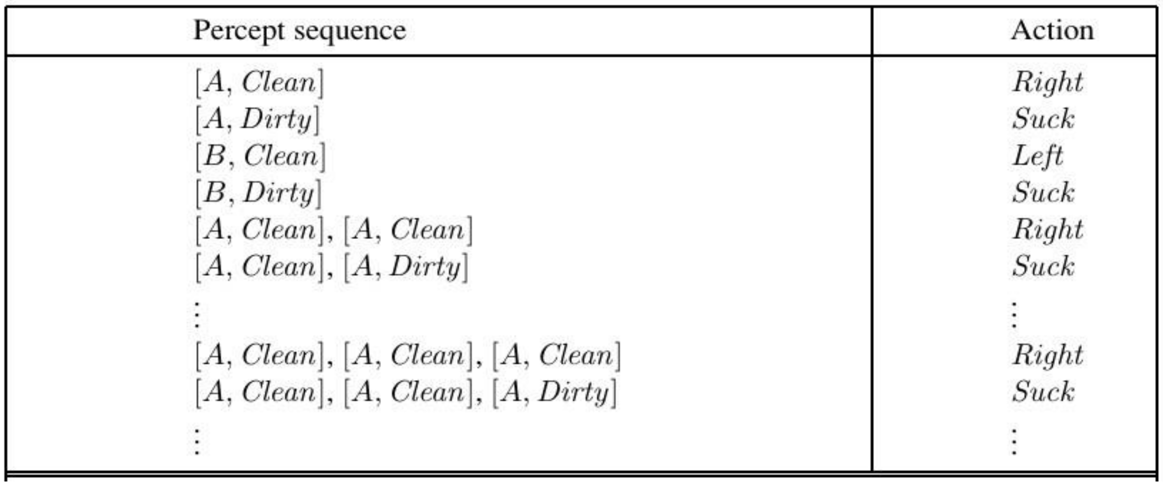
<u>To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world</u>

This particular world has just two locations: **squares A and B.** The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in Figure 1.2



A vacuum-cleaner world with just two locations.

<u>Agent function</u>



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

Figure 1.2: Partial tabulation of a simple agent function for the vacuum-cleaner world

Good Behavior: The Concept of Rationality

A **rational agent** is one that does the right thing every entry in the table for the agent function is filled out correctly.

- When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives.
- This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well.
- This notion of desirability is captured by a **performance measure** that evaluates any given sequence of environment states.

1) Rationality

The rationality of an agent is measured by its performance measure:-

- 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”.

2) Omniscience, learning, and autonomy

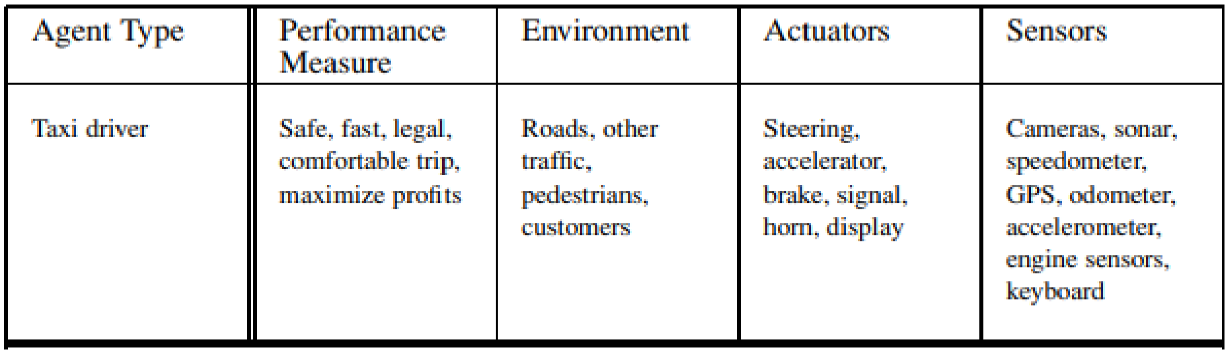
- We need to be careful to distinguish between rationality and omniscience.
- An **omniscient agent** knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.

The Nature of Environment

1) Specifying the Task Environment

The environment is the Task Environment (problem) for which the Rational Agent is the solution. Any task environment is characterized on the basis of **PEAS (Performance, Environment, Actuators, and Sensors).**

Example:- an automated taxi driver.



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

Figure 1.3: PEAS description of the task environment for an automated taxi.

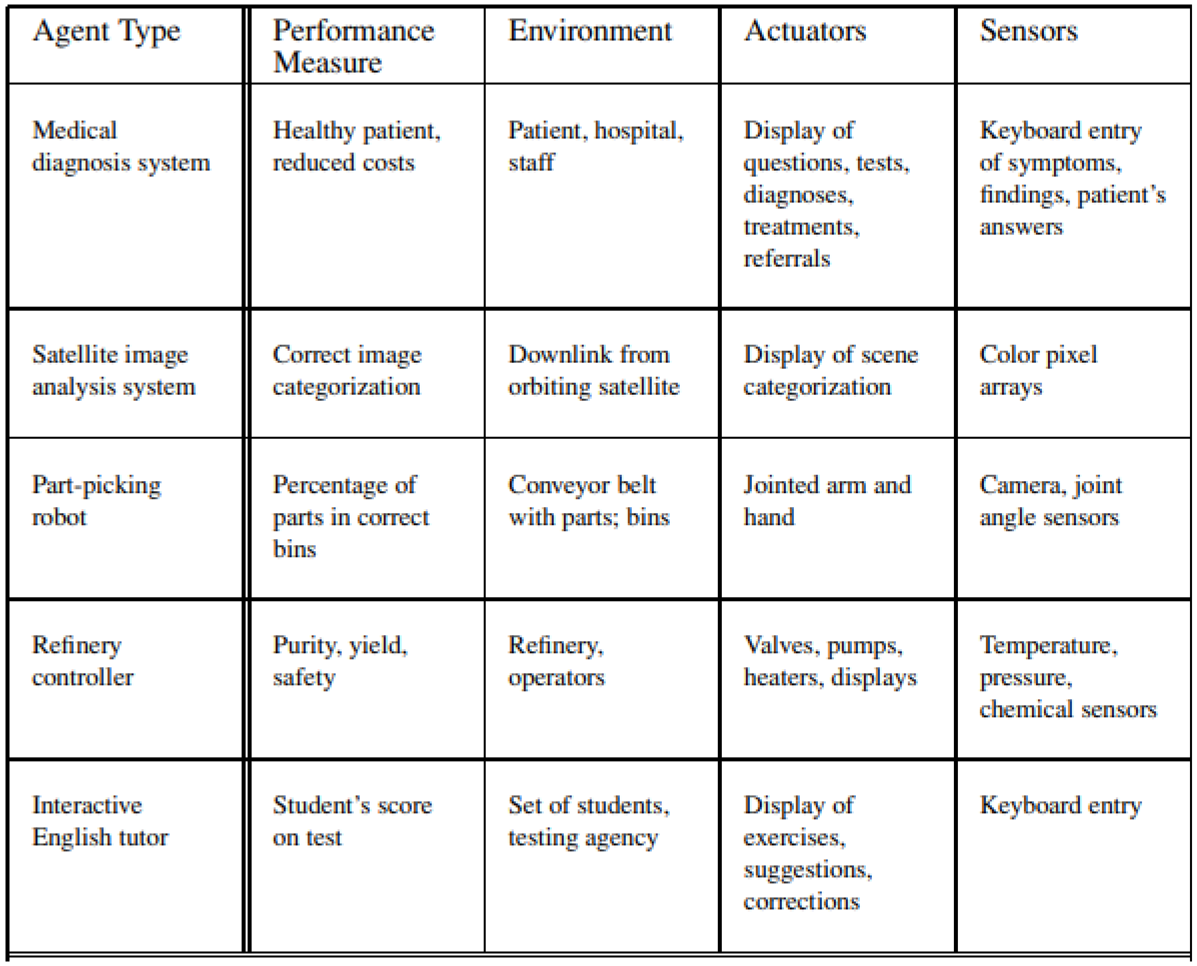
Performance measure:- includes getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time or cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits.

Environment:- Any taxi driver must deal with a variety of roads, ranging from rural lanes and urban alleys to 12-lane freeways. The roads contain other traffic, pedestrians, stray animals etc.

Actuators:- : control over the engine through the accelerator and control over steering and braking. In addition, it will need output to a display screen or voice synthesizer to talk back to the passengers.

<u>Sensors:-</u> include one or more controllable video cameras so that it can see the road; it might augment these with infrared or sonar sensors to detect distances to other cars and obstacles.

Basic PEAS elements for a number of additional agent types



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

Figure 1.4: Examples of agent types and their PEAS descriptions.

<u>2) Properties of Task Environment</u>

a) Fully observable vs Partially observable

b) Single-agent vs Multi-agent

c) Deterministic vs Stochastic

d) Episodic vs sequential

e) Static vs Dynamic
f) Discrete vs Continuous
g) Known vs Unknown

a) Fully Observable vs Partially Observable

- If an agent sensor can sense or access the complete state of an environment at each point of time then it is a fully observable environment, else it is partially observable.
- A fully observable environment is easy as there is no need to maintain the internal state to keep track of the world.
- An agent with no sensors in all environments then such an environment is called as unobservable.

b) Single-agent vs Multi-agent

- If only one agent is involved in an environment, and operating by itself then such an environment is called single agent environment.
  Ex: - crossword puzzle
- If multiple agents are operating in an environment, then such an environment is called multi-agent environment.
  Ex: - Chess

c) Deterministic vs Stochastic

- If an agent's current state and selected action can completely determine the next state of the environment, then such environment is called a deterministic environment.
- A stochastic environment is random in nature and cannot be determined completely by an agent.

d) Episodic vs Sequential

- In an episodic environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action.
  Ex: - an agent that has to spot effective parts on an assembly line bases each decision on

the current part

- In a sequential environment the agent engages in a series of connected episodes.
  Ex: - Chess and taxi driving are sequential

e) Static vs Dynamic

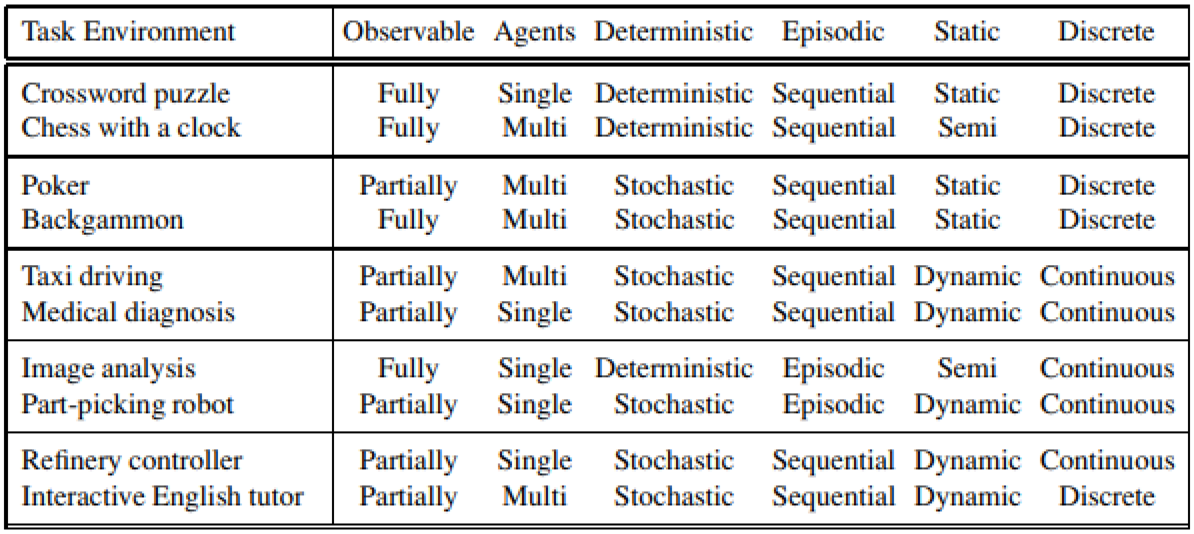
- If the environment can change itself while an agent is deliberating then such environment is called a dynamic environment else it is called a static environment.
  Ex: - Taxi driving
- Static environments are easy to deal because an agent does not need to continue looking at the world while deciding for an action.
  Ex: - Crossword puzzles
- If the environment itself does not change with the time but the agent's performance score does, then we say the environment is **semi-dynamic**.
  Ex: - Chess, when played with a clock

f) Discrete vs Continuous

- If in an environment there are a finite number of percepts and actions that can be performed within it, then such an environment is called a discrete environment else it is called continuous environment.
- A chess game comes under discrete environment as there is a finite number of moves that can be performed.
- A self-driving car is an example of a continuous environment.

g) Known vs Unknown

- Known and unknown are not actually a feature of an environment, but it is an agent's state of knowledge to perform an action.
- In a known environment, the results for all actions are known to the agent. While in unknown environment, agent needs to learn how it works in order to perform an action.
- It is quite possible for a known environment to be partially observable and an unknown environment can be fully observable.



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

Figure 1.5: Examples of task environments and their characteristics.

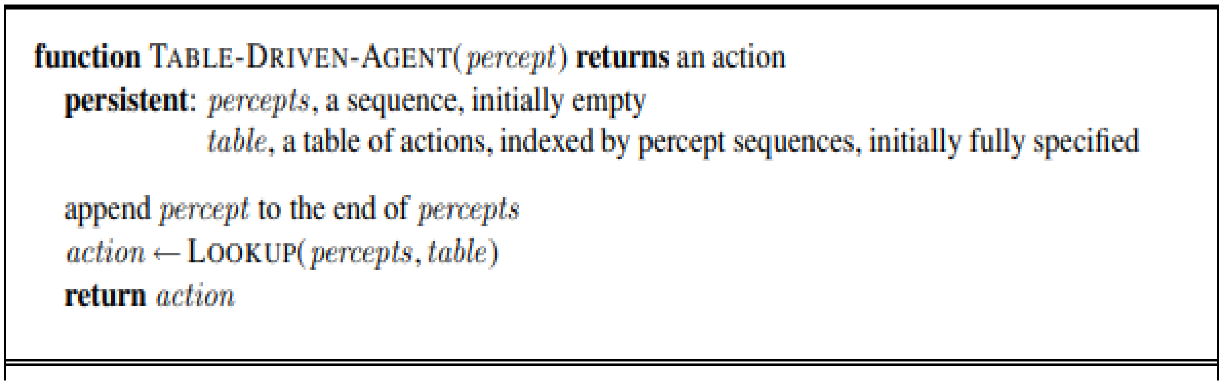
The Structure of Agents

- The task of AI is to design an agent program which implements the agent function- the mapping from percepts to actions.
- The structure of an intelligent agent is a combination of architecture and agent program. It can be viewed as:

Agent = architecture + program

Agent Programs

- The agent programs take the current percept as input from the sensors and return an action to the actuators.
- Simplest agent program is a **TABLE-DRIVEN-AGENT**



```
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
```

Figure 1.6: The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

Four basic kinds of Agent programs

a) Simple reflex agents

b) Model-based reflex agents

c) Goal-based agents

d) Utility-based agents

a) Simple Reflex Agent

- These agents select actions on the basis of the current percept, ignoring the rest of the percept history.
- Works on condition-action rules:

  If *condition* then *action*

- The agent will work only if the correct decision can be made on the basis of only the current percept - work only if the environment is fully observable.

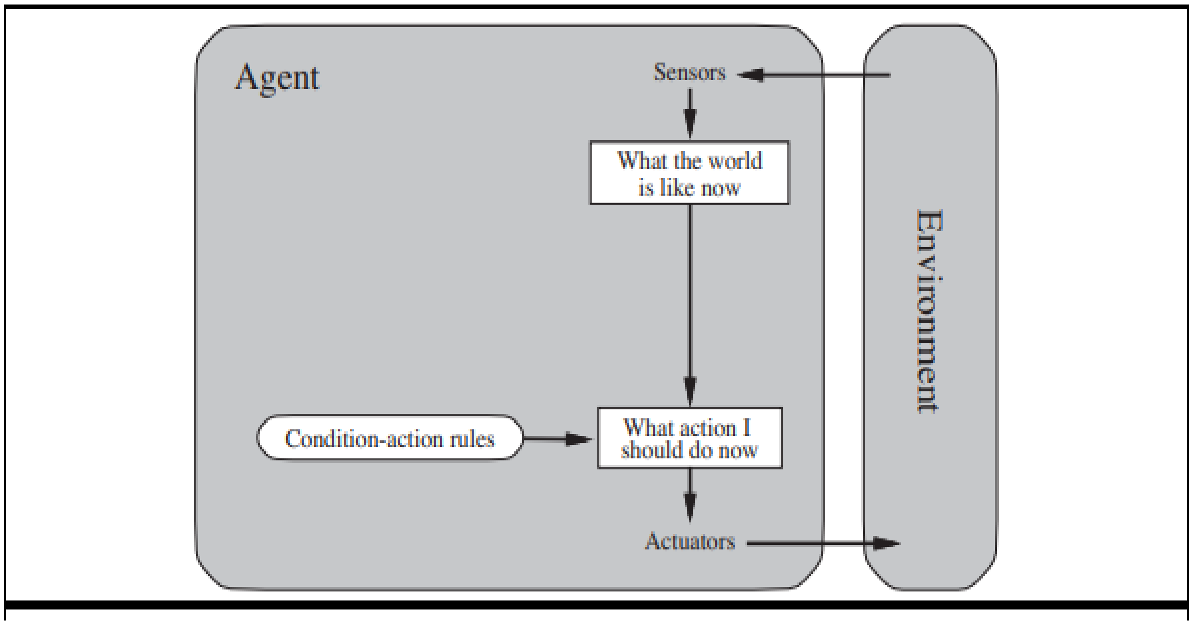


Figure 1.7: Schematic diagram of a simple reflex agent

```
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
```

Figure 1.8: A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

b) Model-Based Reflex Agent

- The most effective way to handle partial observability is to keep track of the part of the world it can't see now.
- That is, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.
- This can be done using two kinds of knowledge:-
    - how the world evolves independently of the agent
    - how the agent's own actions affect the world

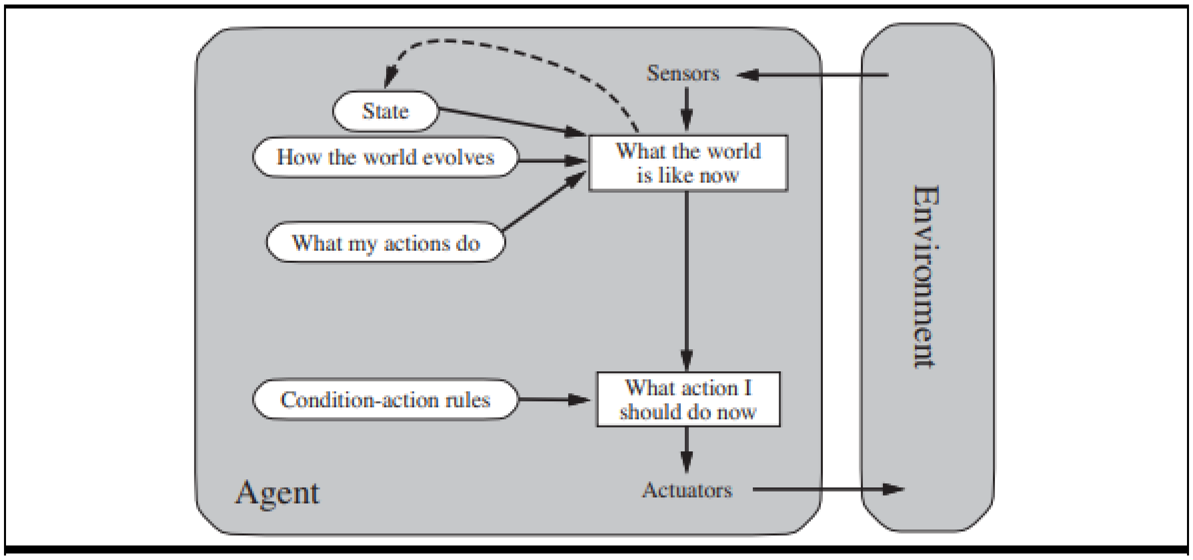


Figure 1.9: A model-based reflex agent.

```
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
```

Figure 1.10: A model-based reflex agent. It 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

c) Goal-Based Agent

- Knowing the current state of the environment is not enough. The agent needs some goal information.

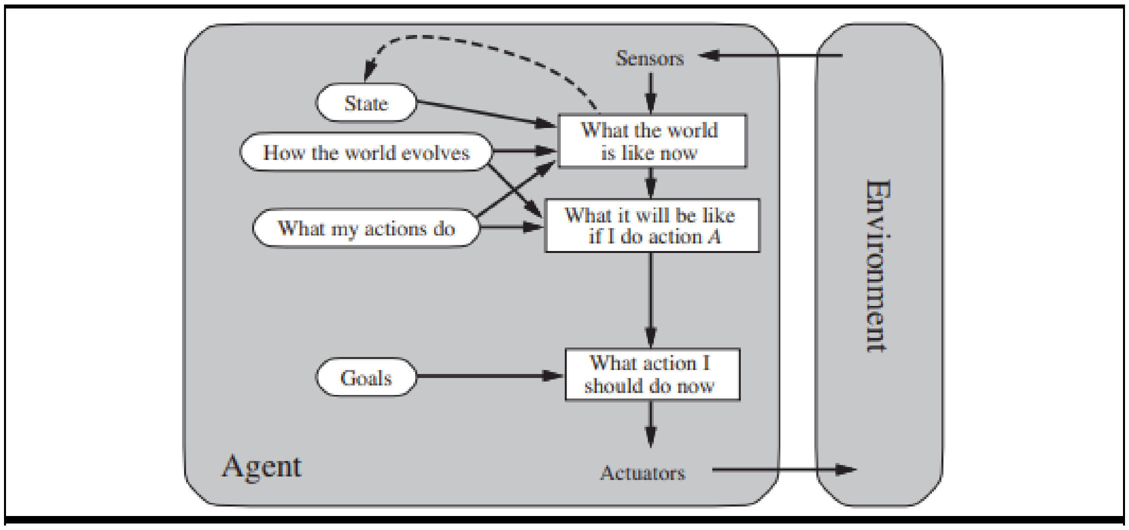

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Figure 1.11: A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

d) Utility-Based Agent

- Sometimes achieving the desired goal is not enough. We may look for quicker, safer, cheaper trip to reach a destination.

- Agent happiness should be taken into consideration. We call it **utility.**
- A utility function is the agent's performance measure.
- Because of the uncertainty in the world, a utility agent chooses the action that maximizes the expected utility.

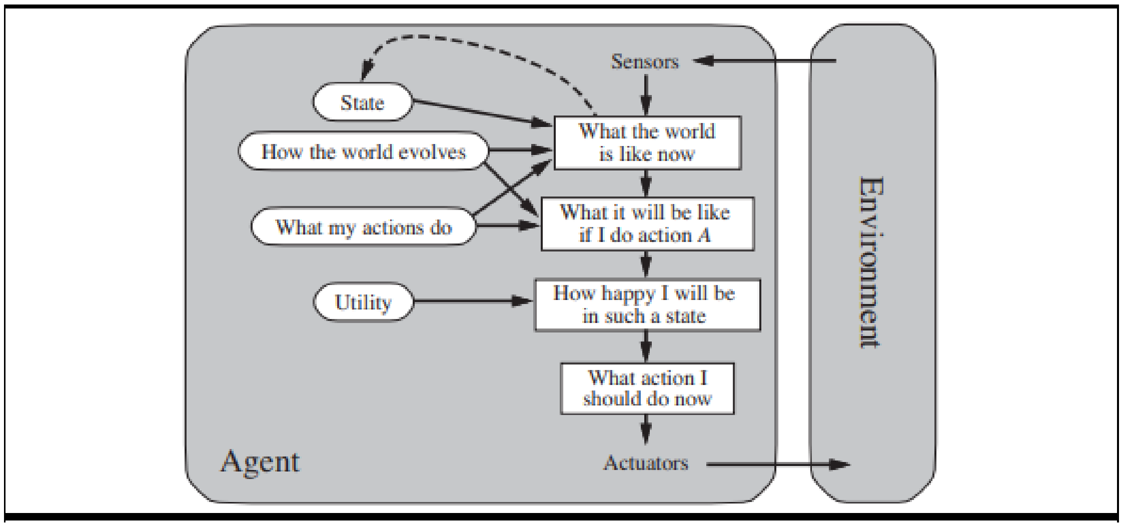

Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Figure 1.12: A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

Learning Agents

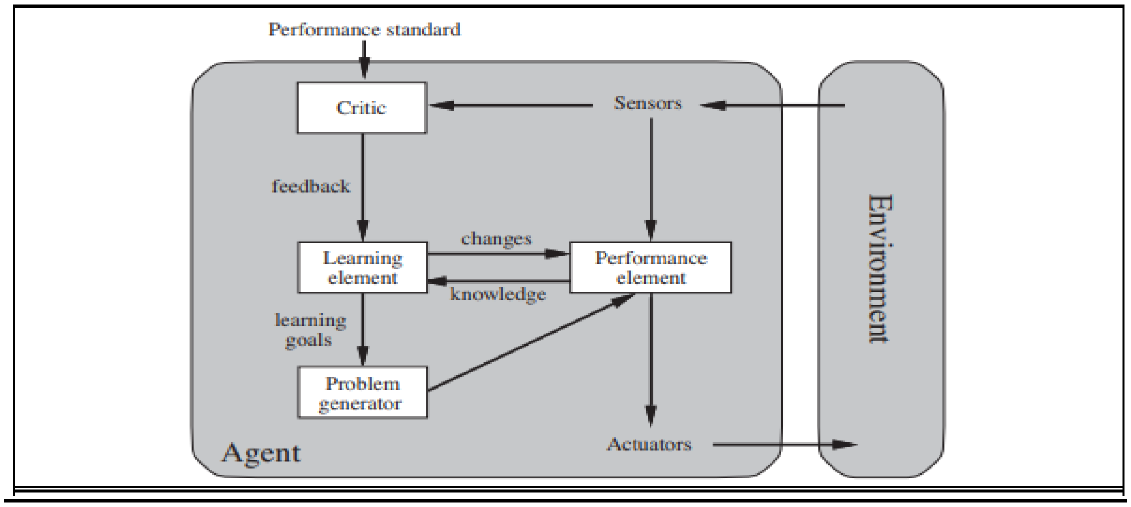


Figure 1.13: A general learning agent.

A learning agent can be divided into four conceptual components

- **Learning element:** responsible for making improvements.
- **Performance element:** responsible for selecting external actions.
- **Critic:** how well the agent is doing w.r.t. a fixed performance standard.
- **Problem generator:** allows the agent to explore.

How the components of Agent Program works

1. **Atomic Representation:** each state of the world is indivisible—it has no internal structure. **Ex-** finding a driving route from one end of a country to the other via some sequence of cities, AI algorithms: search, games, Markov decision processes, Hidden Markov models.
2. **Factored Representation:** splits up each state into a fixed set of variables or attributes, each of which can have a value. **Ex-** GPS location, amount of gas in the tank, AI algorithms: constraint satisfaction and Bayesian networks.
3. **Structured Representation:** relationships between the objects of a state can be explicitly expressed. **Ex-** AI algorithms: first order logic, knowledge-based learning.

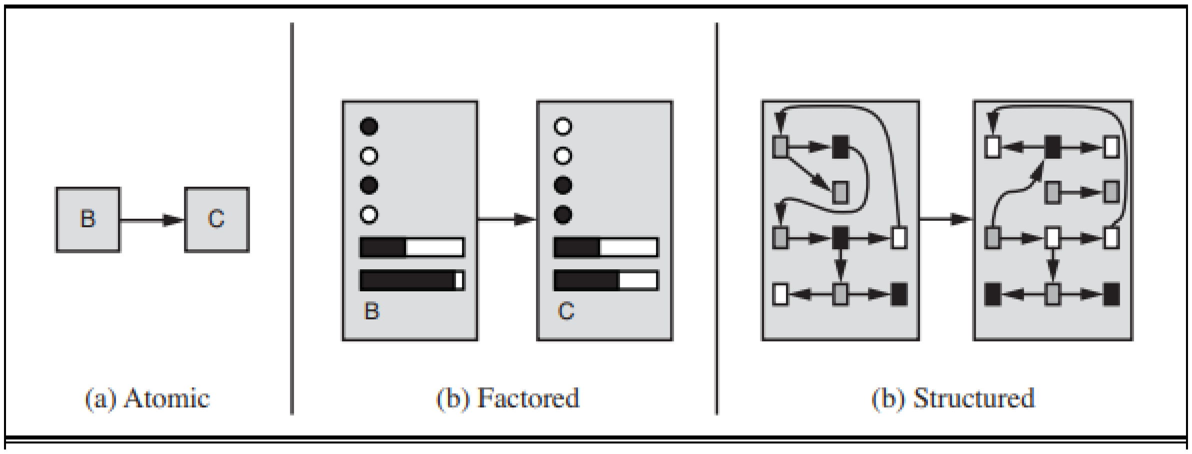


Figure 1.14: Three ways to represent states and the transitions between them. **(a) Atomic representation:** a state (such as B or C) is a black box with no internal structure; **(b) Factored representation:** a state consists of a vector of attribute values; values can be Boolean, realvalued, or one of a fixed set of symbols. **(c) Structured representation:** a state includes objects, each of which may have attributes of its own as well as relationships to other objects

SOLVING PROBLEMS BY SEARCHING

Problem-Solving Agents

Romania Problem

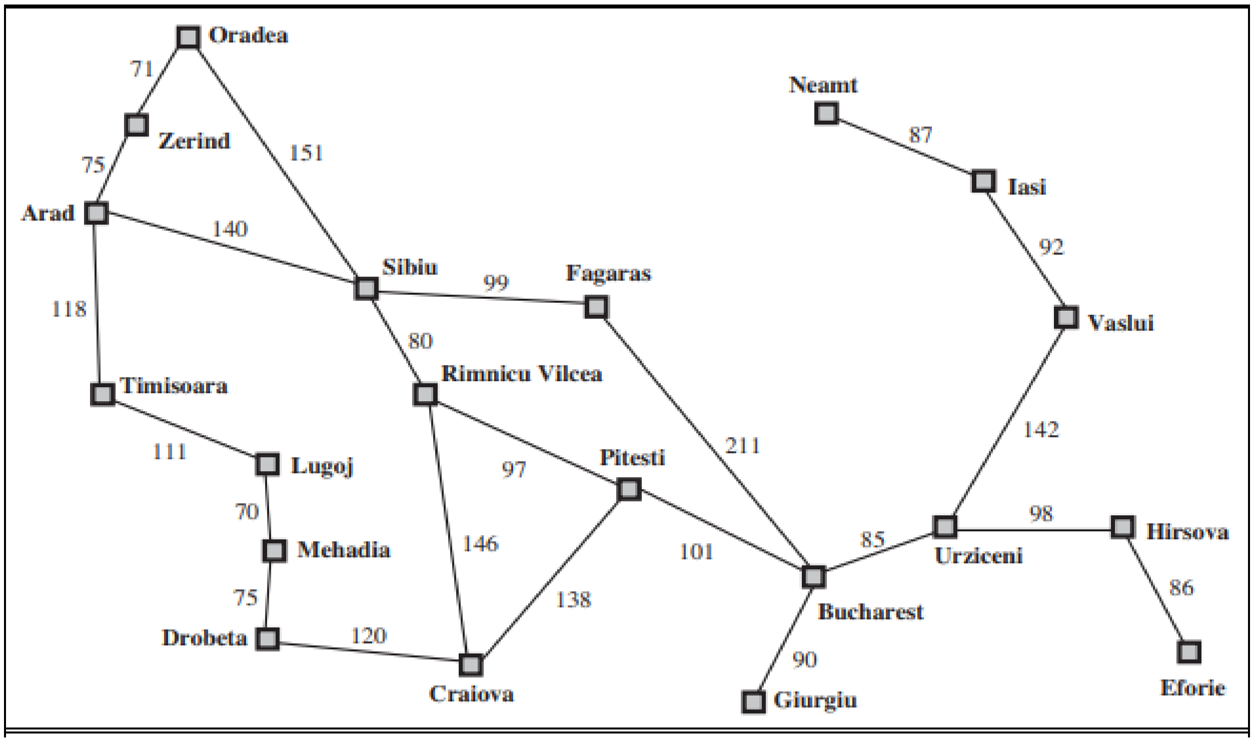
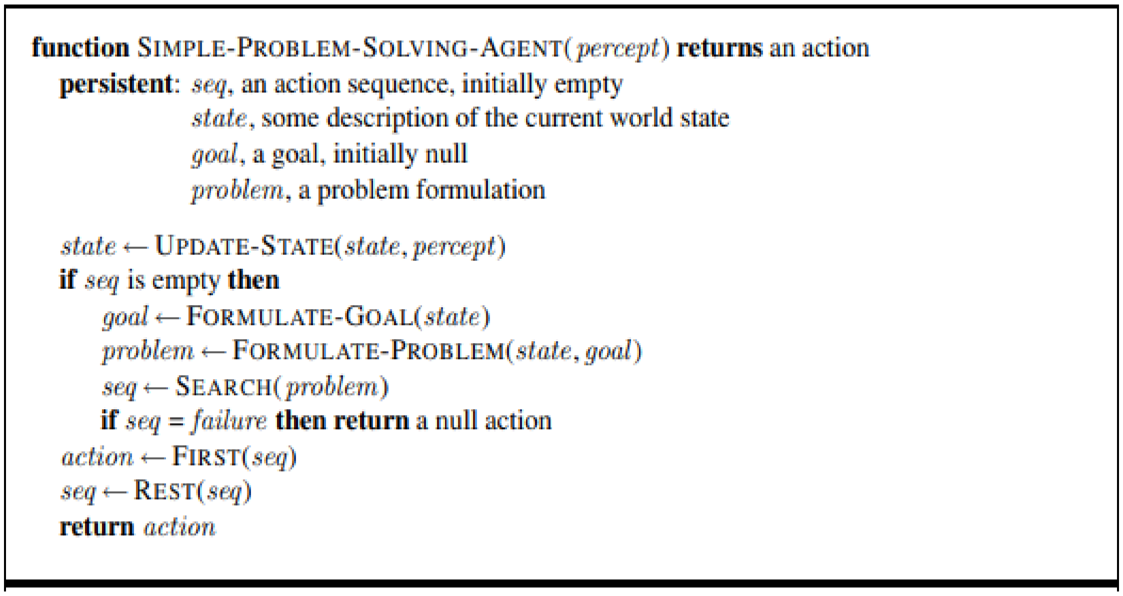


Figure 1.15: A simplified road map of part of Romania

- Imagine an agent in the city of Arad, Romania, enjoying a touring holiday.
- The agent's performance measure contains many factors:
    - ➢ it wants to improve its suntan,
    - ➢ improve its Romanian, take in the sights,
    - ➢ enjoy the nightlife (such as it is),
    - ➢ avoid hangovers, and so on.
- The agent has a nonrefundable ticket to fly out of Bucharest the following day.
- The agent observes that there are three roads leading out of Arad: one toward Sibiu, one to Timisoara, and one to Zerind.

Agent follows this **four phase problem-solving process:**

- **Goal Formulation:** the agent adopts the goal of reaching Bucharest.
- **Problem Formulation:** the agent devises a description of the states and actions necessary to reach the goal. Agent considers the actions of travelling from one city to an adjacent city; the only fact that will change due to an action is the current city.
- **Search:** before taking any action in the real world, the agent simulates sequences of actions, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called **solution.**
- **Execution:** the agent can now execute the actions in the solution, one at a time.



```
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
```

Figure 1.16: 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.

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

- **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. For example, we have **RESULT (In (Arad), Go (Zerind)) = In (Zerind) .**
- The **goal** test, which determines whether a given state is a goal state.
- A path **cost function** that assigns a numeric cost to each path.

Example Problems

- **Toy problem** is intended to illustrate or exercise various problem-solving methods.
- A **real-world problem** is one whose solutions people actually care about.

Toy Problems

1) Vacuum world

This can be formulated as a problem as follows:

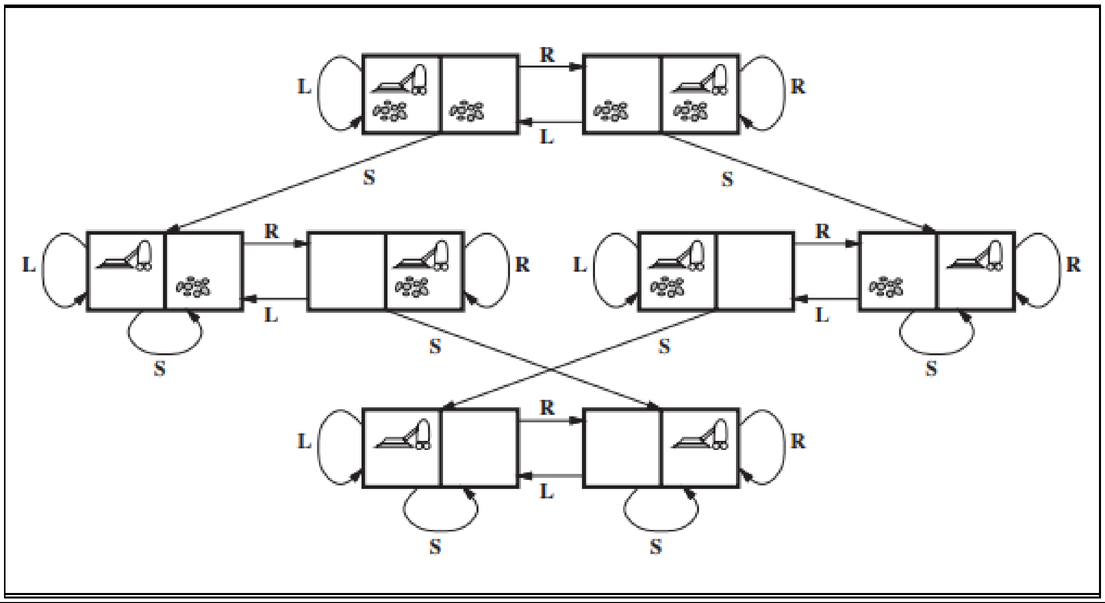


Figure 1.17: The state space for the vacuum world. Links denote actions: L = Left, R= Right, S= Suck.

- **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

$$\mathbf{n \cdot 2^n} \text{ states.}$$

- **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 Sucking 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.

2) The 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, such as the one shown on the right of the figure. The standard formulation is as follows:

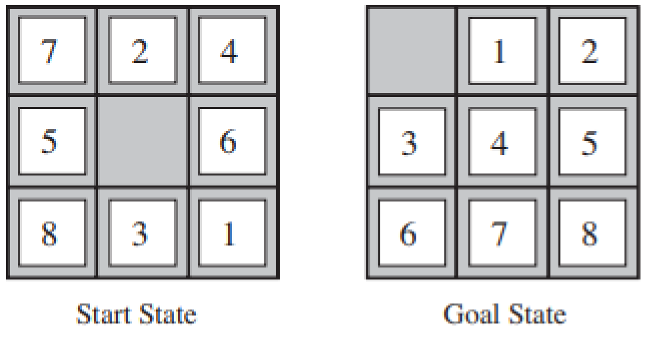


Figure 1.18: A typical instance of the 8-puzzle.

- **States:** specifies the location of each of the 8 tiles and the blank in one of the nine squares.
- **Initial state:** Any state can be designated as the initial state.
- **Actions:** 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 in Figure 1.18, the resulting state has the 5 and the blank

switched.

- **Goal test:** This checks whether the state matches the goal configuration shown in Figure 1.18. (Other goal configurations are possible.)
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

3) The 8-queens problem

The goal is to place eight queens on a chessboard such that no queen attacks any other.

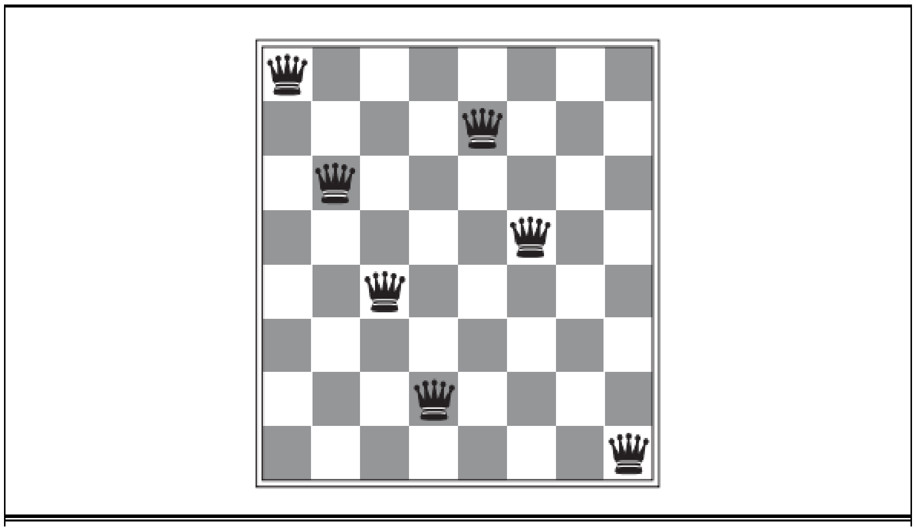


Figure 1.19: solution to the 8-queens problem.

- **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.

4) Final toy problem was devised by Donald Knuth (1964)

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.

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

For example, we can reach 5 from 4 as follows:

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 Problems

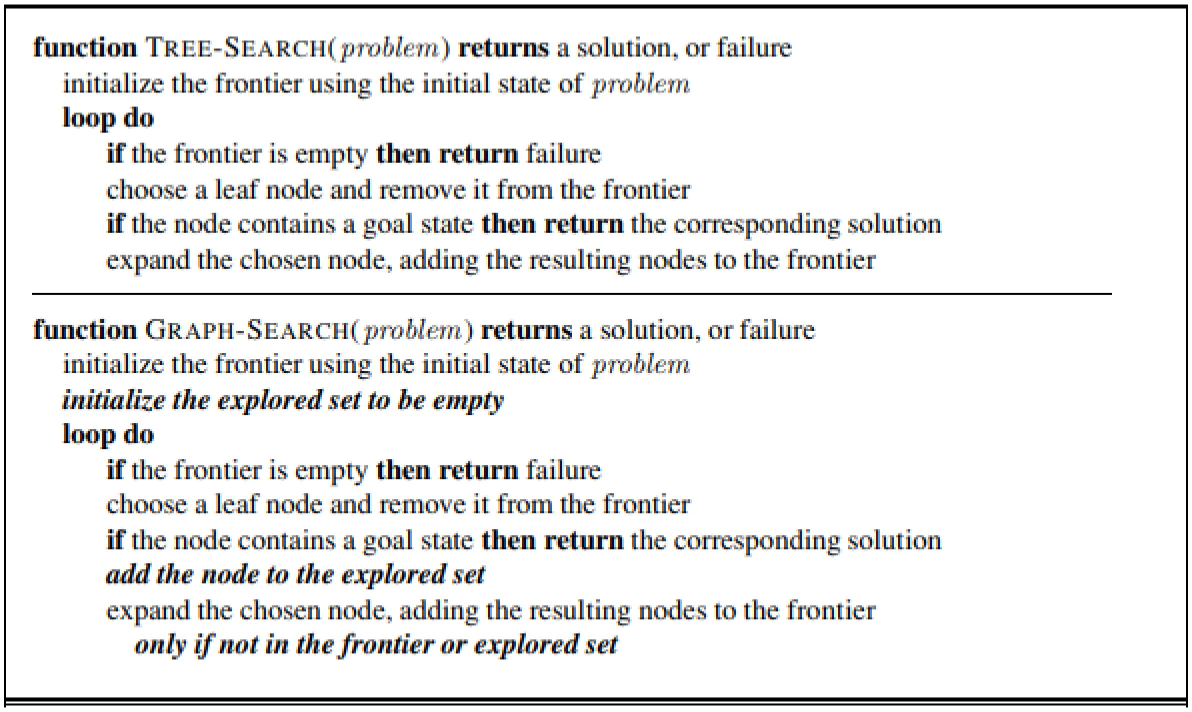
- Route-finding problem
- airline travel problem
- Touring problems
- traveling salesperson problem (TSP)
- A VLSI layout problem
- Robot navigation
- Automatic assembly sequencing

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, and so on

Searching for solution

- A solution is an action sequence, so search algorithms work by considering various possible action sequences.
- The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem.
- Expanding the current state.
- Generating a new set of states.
- Current state - parent node In(Arad)
- child nodes: In(Sibiu), In(Timisoara), and In(Zerind)
- **Leaf node,** that is, a node with no children in the tree.
- The set of all leaf nodes available for expansion at any given point is called the **frontier.**
- how they choose which state to expand next—the so-called search strategy
- repeated state in the search tree - **a loopy path.**



```
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
```

Figure 1.20: An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

1) Infrastructure for search algorithms

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.

2) Measuring problem-solving performance

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, as defined on page 68?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

Uninformed search strategies:

These strategies have no additional information about states beyond that provided in the problem definition. Hence it is also called as **Blind Search.**

1. Breadth-first search
2. Uniform-cost search
3. Depth-first search
4. Depth-limited search
5. Iterative deepening depth-first search
6. Bidirectional search

1) Breadth First Search

- Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- Breadth-first search is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion.

- This is achieved very simply by using a **FIFO queue for the frontier.**
- Thus, new nodes go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 1.21: Breadth-first search on a graph.

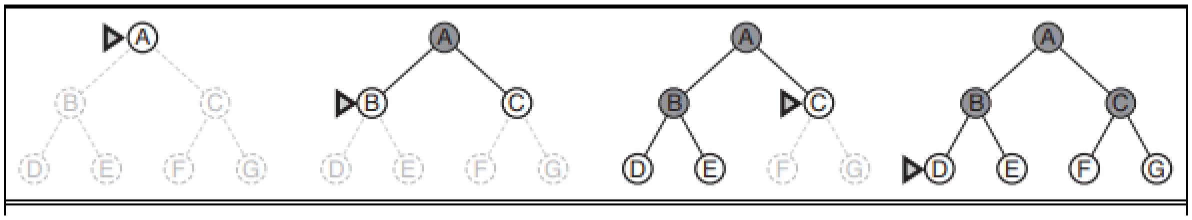


Figure 1.22: Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

<u>Properties of BFS</u>

- **Complete -** if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes
- **Optimal -** optimal if the path cost is a non-decreasing function of the depth of the node (all actions have the same cost)
- **Time Complexity – $1+b+b^2+b^3+\ldots..+b^d = O(b^d)$**
- **Space Complexity - $O(b^d)$**

2) Uniform Cost Search

- Expands the node n with the lowest path cost g(n).
- This is done by storing the frontier as a priority queue ordered by g.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Figure 1.23: Uniform-cost search on a graph.

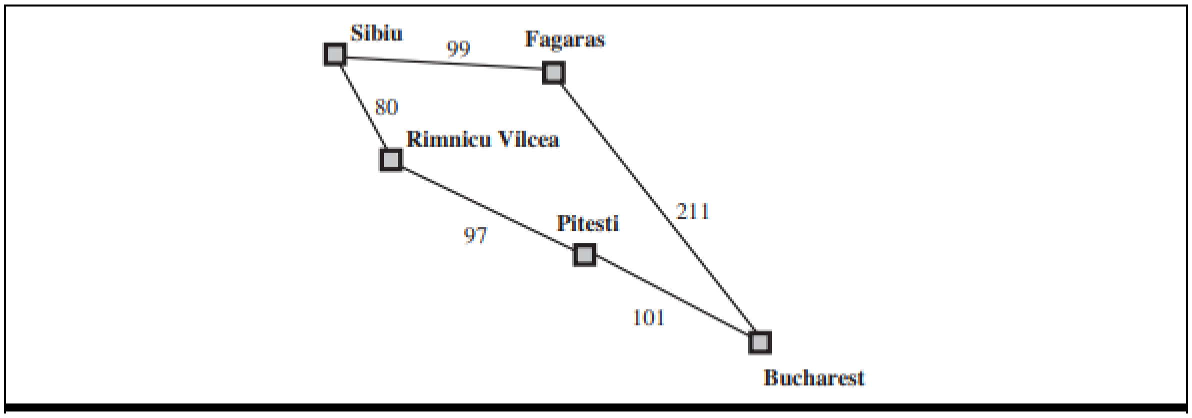


Figure 1.24: Part of the Romania state space, selected to illustrate uniform-cost search

- The problem is to get from **Sibiu to Bucharest.**
- The successors of **Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99**, respectively.
- The least-cost node, **Rimnicu Vilcea, is expanded next**, adding Pitesti with cost **80 + 97 = 177.**

- The least-cost node is now **Fagaras, so it is expanded**, adding Bucharest with cost **99 + 211 = 310.**
- Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost **80+ 97+ 101 = 278.**
- Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with **g-cost 278**, is selected for expansion and the solution is returned.

Properties of UCS

- **Completeness** – is guaranteed provided the cost of every step exceeds some small positive constant €.
- **Optimal -** uniform-cost search is optimal in general**.**
- **Time Complexity – $O(b^{1+[C*/€]})$**
- **Space Complexity - $O(b^{1+[C*/€]})$**

3) Depth First Search

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.
- The depth-first search algorithm is an instance of the graph-search algorithm, whereas **breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.**
- A **LIFO queue** means that the most recently generated node is chosen for expansion.

Properties of DFS

- **Completeness –** is complete in finite state spaces because it will eventually expand every node.

- **Optimal –** DFS is not optimal, as it may generate a large number of steps or high cost to reach the goal node.
- **Time Complexity – O (b^m)**

  Where, m = maximum depth of any node and this can be much larger than d
- **Space Complexity - O (b^m)**

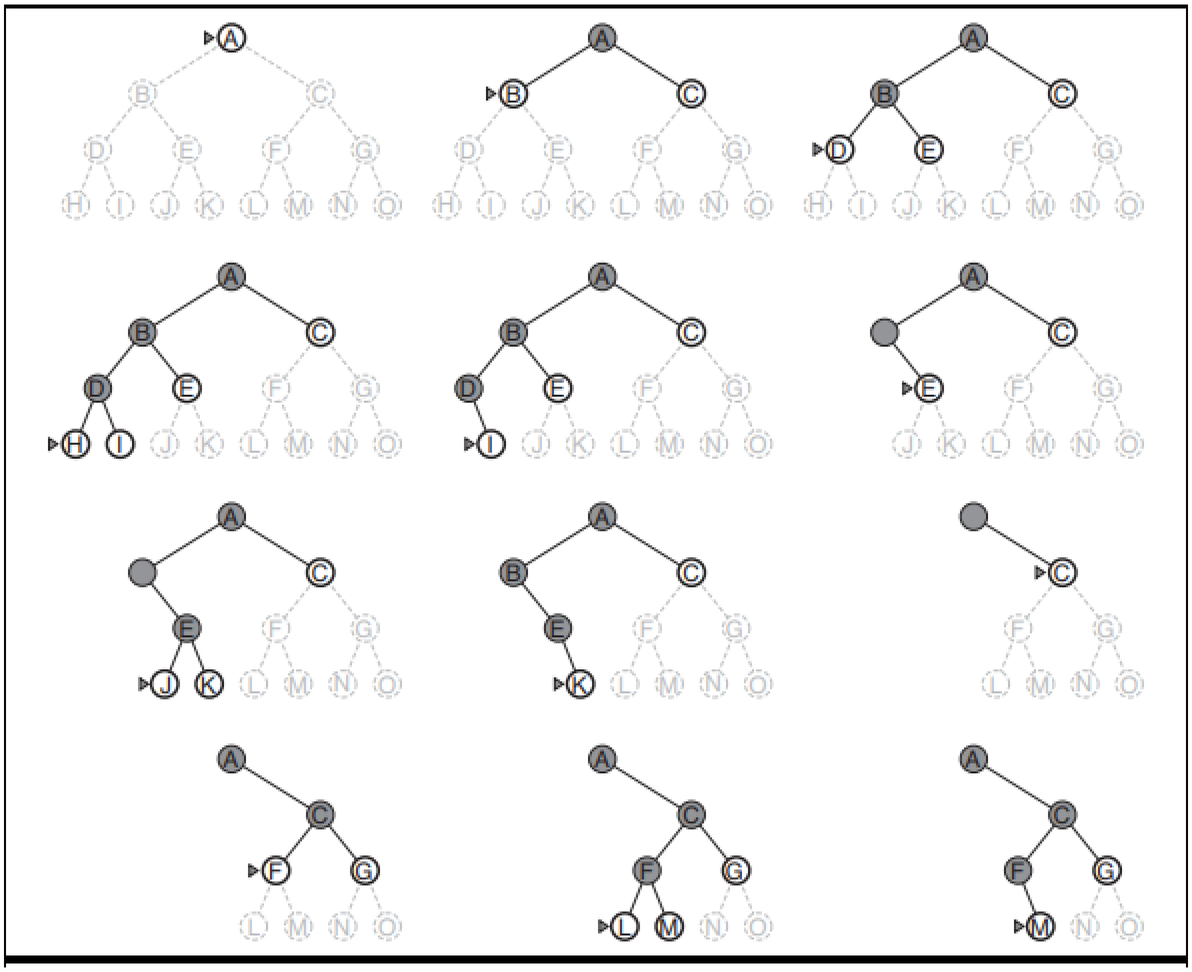
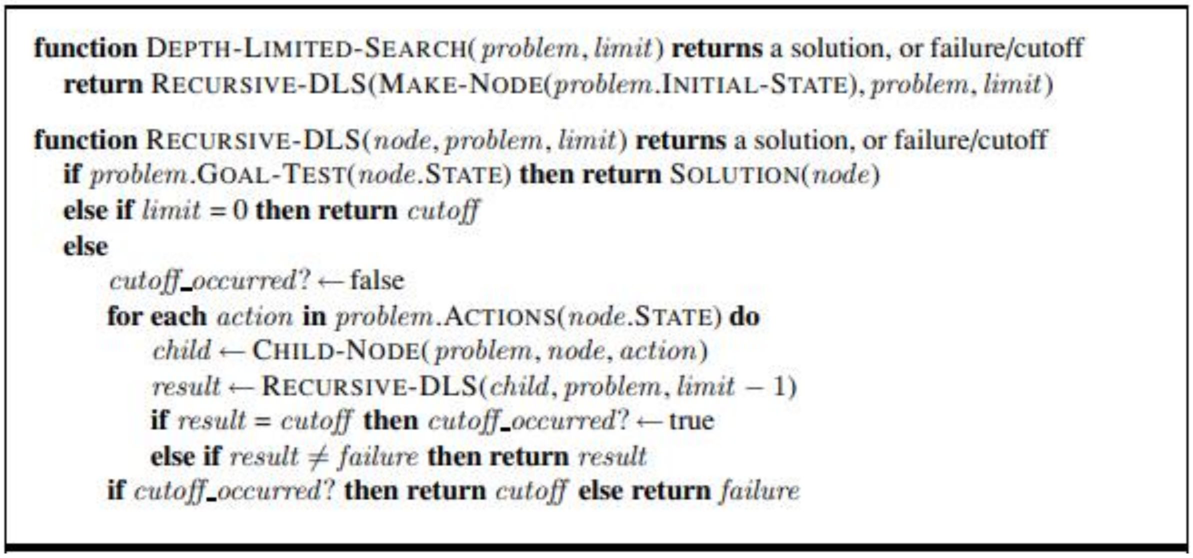


Figure 1.25: Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

4) Depth-Limited Search

- The failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a pre-determined depth limit ***l***.

- That is, nodes at depth ***l*** are treated as if they have no successors.
- This approach is called **depth-limited search**.



```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 1.26: A recursive implementation of depth-limited tree search.

<u>**Properties of DLS**</u>

- **Completeness –** DLS is complete if the solution is above the depth-limit. Otherwise, incomplete.
- **Optimal –** not optimal even if $l > d$
- **Time Complexity – O (b^l)**
- **Space Complexity – O (b x *l*)**

<u>5) Iterative Deepening Depth-First Search</u>

- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.
- This will occur when the depth limit reaches d, the depth of the shallowest goal node.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Figure 1.27: The iterative deepening search algorithm, which repeatedly applies depthlimited search with increasing limits. It terminates when a solution is found or if the depthlimited search returns failure, meaning that no solution exists.

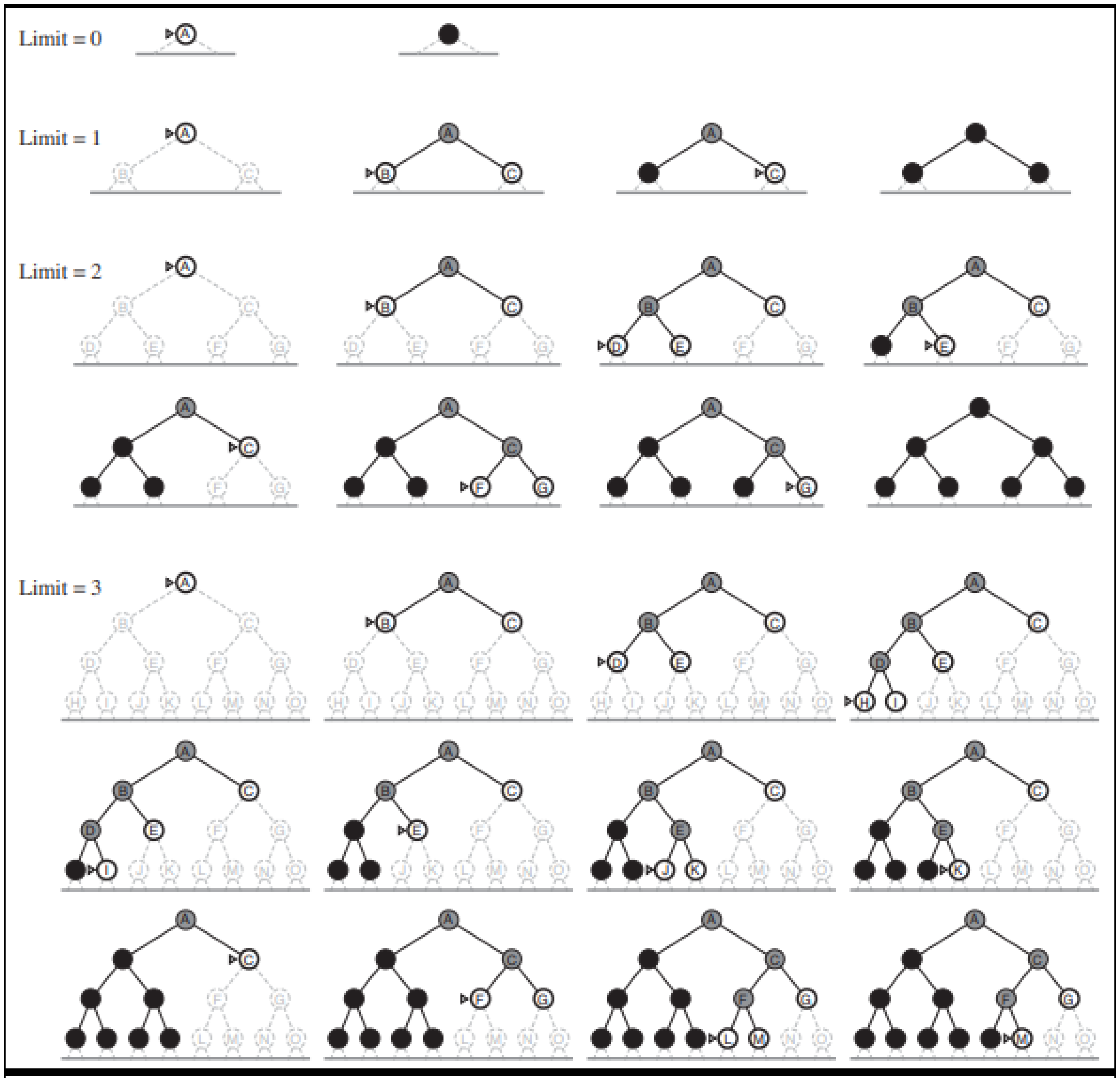


Figure 1.28: Four iterations of iterative deepening search on a binary tree.

Properties of IDDFS

- **Completeness –** Yes
- **Optimal –** Yes
- **Time Complexity – O (b^d)**
- **Space Complexity – O (b x d)**

6) Bidirectional Search

- The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d.

Properties of BS

- **Completeness –** Yes
- **Optimal –** Yes
- **Time Complexity – O ($b^{d/2}$)**
- **Space Complexity – O ($b^{d/2}$)**