Chapter 2

Title	General Introduction
Subject	Principals and Approaches Of Artificial Intelligence
Author	Chourouk Guettas
Grade level	Master 1
Objective	Learn the basics and the approaches of AI
Materials	Book; Artificial Intelligence: A modern Approach
Activities and procedures	Presentations and exercises
Lecture's Plan	**Chapter 2 Intelligent Agents** 1- Introduction 2- The concept of Rationality 3- Task environment (PEAS) 4- Environment proprieties 5- Agent structure

In which we discuss the nature of agents, perfect or otherwise, the diversity of environments, and the resulting menagerie of agent types.

2.1 Introduction

1. Definition of an Agent

An agent is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**. This simple idea is illustrated in **Figure 2.1.**

- Examples:

- A human agent has *eyes, ears, and other organs* for sensors and *hands, legs, vocal tract, and so on* 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.

The final element is the connection between knowledge and action, which is vital to AI. Intelligence requires actions as well as reasoning and only by understanding how actions are justified can we understand how to build an agent whose actions are **justifiable** (or rational).

- Perception: is used to refer to the agent's perceptual inputs at any given instant. An agent's ***percept sequence*** is the complete history of everything the agent has ever perceived. In general, an agent's choice of action at any given instant can depend on the entire percept sequence observed to date, but not on anything it hasn't perceived.

- Action: By specifying the agent's choice of action for every possible percept sequence, an agent is defined. Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

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; while the agent program is a concrete implementation, running within some physical system.

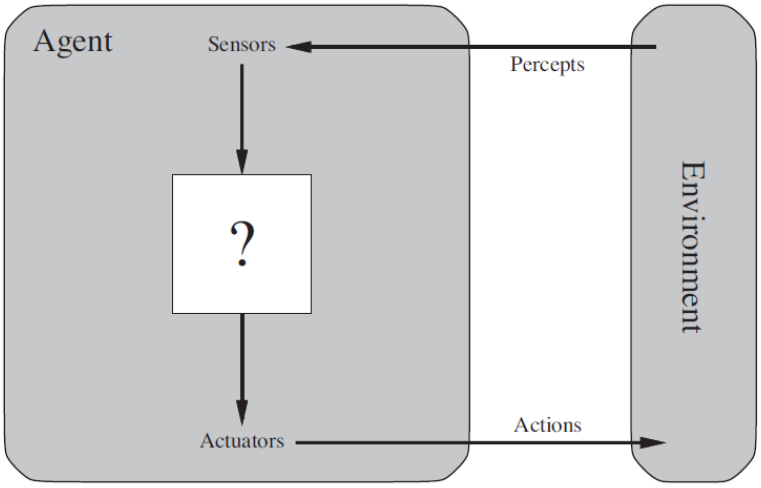

Figure 2.1 Agents interact with environments through sensors and actuators.

2. Vacuum Cleaner Example

The vacuum-cleaner world shown in **Figure 2.2.** This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. 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 2.3** and an agent program that implements it appears later in **Figure 2.8**.

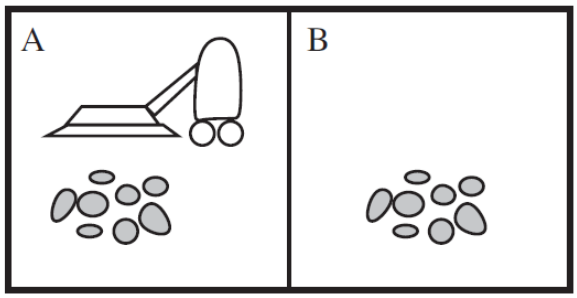

Figure 2.2 A vacuum-cleaner world with just two locations.

Looking at **Figure 2.3**, the obvious question, then, is this: What is the right way to fill out the table? In other words, what makes an agent good or bad, intelligent or stupid? We answer these questions in the next section.

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 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

2.1 The Concept of 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:*

1. 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.

Example:

Consider the simple vacuum-cleaner agent that cleans a square if it is dirty and moves to the other square if not; this is the agent function tabulated in **Figure 2.3**. Is this a rational agent? That depends! First, we need to

say what the *performance measure* is, what is known about the environment, and what sensors and actuators the agent has. Let us assume the following:

- The performance measure awards one point for each clean square at each time step, over a "lifetime" of 1000 time steps.
- The "geography" of the environment is known a priori (Figure 2.2) but the dirt distribution and the initial location of the agent are not. Clean squares stay clean and sucking cleans the current square. The Left and Right actions move the agent left and right except when this would take the agent outside the environment, in which case the agent remains where it is.
- The only available actions are Left, Right, and Suck.
- The agent correctly perceives its location and whether that location contains dirt.

We claim that under these circumstances the agent is indeed rational; its expected performance is at least as high as any other agent's.

You can see easily that the same agent would be irrational under different circumstances. ***For example***, once all the dirt is cleaned up, the agent will oscillate needlessly back and forth; if the performance measure includes a penalty of one point for each movement left or right, the agent will fare poorly. A better agent for this case would do nothing once it is sure that all the squares are clean. If clean squares can become dirty again, the agent should occasionally check and re-clean them if needed. If the geography of the environment is unknown, the agent will need to explore it rather than stick to squares A and B.

2. Omniscience, learning, and autonomy

- There is a difference between rationality and **omniscience**. An omniscient agent knows the *actual* outcome of its actions and can act accordingly; but omniscience is impossible in reality. Rationality maximizes *expected* performance, while perfection maximizes *actual* performance; if we expect an agent to do what turns out to be the best action after the fact, it will be impossible to design an agent to fulfil this specification. Lastly, rationality does not require **omniscience**, because the rational choice depends only on the percept sequence *to date*.

- A rational agent needs to **learn** as much as possible from what it perceives. The agent's initial configuration could reflect some prior knowledge of the environment, but as the agent gains experience this may be modified and augmented. There are extreme cases in which the environment is completely known *a priori*. In such cases, the agent need not perceive or learn; it simply acts correctly.

- When an agent relies on the prior knowledge of its designer rather than on its own *percepts*, we say that the agent lacks **autonomy**. A rational agent should be autonomous—it should learn what it can to compensate for partial or incorrect prior knowledge. An artificial rational intelligent agent must be provided with some initial knowledge as well as an ability to learn. After sufficient experience of its environment, its behaviour can become effectively *independent* of its prior knowledge.

2.2 Task environment (PEAS)

In designing an agent, the first step must always be to specify the task environment as fully as possible; through identifying the performance measure, the environment, and the agent's actuators and sensors. All these are grouped under the heading of the **task environment** and referred acronymically by **PEAS** (**P**erformance, **E**nvironment, **A**ctuators, and **Sensors**) description.

Example (Automated Taxi Driver):

Taking into consideration that A fully automated taxi is currently somewhat beyond the capabilities of existing technology. **Figure 2.4** summarizes the PEAS description for the taxi's task environment. Each element is discussed in more detail in the following paragraphs.

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 2.4 PEAS description of the task environment for an automated taxi.

- **Performance measure:** desirable qualities for an automated driver that we aspire for, more likely, would be: getting to the correct destination; minimizing fuel consumption; minimizing the trip time or cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits. Obviously, some of these goals conflict, so trade-offs will be required.
- **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, road works, police cars, puddles, and potholes. The taxi must also interact with potential and actual passengers. There are also some optional choices. The taxi might need to operate in a snowy area or it could be driving on the right or on the left depending on the country. Obviously, the more restricted the environment, the easier the design problem.
- **Actuators:** include those available to a human driver: 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, and perhaps some way to communicate with other vehicles.
- **Sensors:** for the taxi will 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. To avoid speeding tickets, the taxi should have a speedometer, and to control the vehicle properly, especially on curves, it should have an accelerometer. To determine the mechanical state of the vehicle,

it will need the usual array of engine, fuel, and electrical system sensors. Like many human drivers, it needs a global positioning system (GPS). Finally, it will need a keyboard or microphone for the passenger to request a destination.

What matters is not the distinction between "real" and "artificial" environments, but the complexity of the relationship among the behaviour of the agent, the percept sequence generated by the environment, and the performance measure. Some "real" environments are actually quite simple while some artificial environment are so complicated, for example: an inspector robot of an industrial conveyor belt vs. a softbot Web site operator to scan Internet news sources.

2.3 Proprieties of task environments

The range of task environments that might arise in AI is obviously vast. We can, however, identify a fairly small number of dimensions along which task environments can be categorized. These dimensions determine, to a large extent, the appropriate agent design and the applicability of each of the principal families of techniques for agent implementation. First, we list the dimensions, then we analyse several task environments to illustrate the ideas. In **Figure 2.5**, the basic PEAS elements for a number of additional agent types are presented.

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 2.5 Examples of agent types and their PEAS descriptions.

- **Fully observable vs. partially observable:** If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is **fully observable**. A task environment is effectively fully observable if the sensors detect all aspects that are relevant to the choice of action; relevance, in turn, depends on the performance measure. Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world. An environment might be **partially observable** because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data—for example, a vacuum agent with only a local dirt sensor cannot tell whether there is dirt in other squares, and an automated taxi cannot see what other drivers are thinking. If the agent has no sensors at all then the environment is **unobservable.**
- **Single agent vs. multiagent:** The distinction between single-agent and multiagent environments may seem simple enough. For example, an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment. However, we can distinguish different types for of behaviours in multiagent environments. For example: chess is a **competitive** multiagent environment. In the taxi-driving environment, on the other hand, avoiding collisions maximizes the performance measure of all agents, so it is a partially **cooperative** multiagent environment. It is also partially competitive considering the parking case. The agent-design problems in multiagent environments are often quite different from those in single-agent environments; for example, **communication** often emerges as a rational behaviour in multiagent environments; in some competitive environments, **randomized behaviour** is rational because it avoids the pitfalls of predictability.
- **Deterministic** vs. **stochastic**: If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic. Most real situations are so complex that it is impossible to keep track of all the unobserved aspects; for practical purposes, they must be treated as stochastic. Taxi driving is clearly stochastic in this sense, because one can never predict the behaviour of traffic exactly; moreover, one's tires blow out and one's engine seizes up without warning. The vacuum world as we described it is deterministic, but variations can include stochastic elements such as randomly appearing dirt and an unreliable suction mechanism.
- **Episodic vs. sequential:** In an episodic task environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action. Crucially, the next episode does not depend on the actions taken in previous episodes. For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions; moreover, the current decision doesn't affect whether the next part is defective. In sequential environments, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential: in both cases, short-term actions can have long-term consequences. Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

- **Static vs. dynamic:** If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. Dynamic environments, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing. If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic.** Taxi driving is clearly dynamic: the other cars and the taxi itself keep moving while the driving algorithm dithers about what to do next. Chess, when played with a clock, is semidynamic. Crossword puzzles are static.
- **Discrete vs. continuous:** The discrete/continuous distinction applies to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent. For example, the chess environment has a finite number of distinct states (excluding the clock). Chess also has a discrete set of percepts and actions. Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time. Taxi-driving actions are also continuous (steering angles, etc.). Input from digital cameras is discrete, strictly speaking, but is typically treated as representing continuously varying intensities and locations.
- **Known vs. unknown:** Strictly speaking, this distinction refers not to the environment itself but to the agent's (or designer's) state of knowledge about the "laws of physics" of the environment. In a known environment, the outcomes (or outcome probabilities if the environment is stochastic) for all actions are given. Obviously, if the environment is unknown, the agent will have to learn how it works in order to make good decisions. Note that the distinction between known and unknown environments is not the same as the one between fully and partially observable environments. It is quite possible for a *known* environment to be *partially* observable—for example, in solitaire card games, I know the rules but am still unable to see the cards that have not yet been turned over. Conversely, an *unknown* environment can be *fully* observable—in a new video game, the screen may show the entire game state but I still don't know what the buttons do until I try them.

Note that the hardest case is *partially observable, multiagent, stochastic sequential, dynamic, continuous, and unknown*. Taxi driving is hard in all these senses, except that for the most part the driver's environment is known. Driving a rented car in a new country with unfamiliar geography and traffic laws is a lot more exciting.

Figure 2.6 lists the properties of a number of familiar environments. Note that the answers are not always cut and dried. *For example*, we describe the part-picking robot as episodic, because it normally considers each part in isolation. But if one day there is a large batch of defective parts, the robot should learn from several observations that the distribution of defects has changed, and should modify its behavior for subsequent parts. "known/unknown" column is not included because, as explained earlier, this is not strictly a property of the environment. For some environments, such as chess and poker, it is quite easy to supply the agent with full knowledge of the rules, but it is nonetheless interesting to consider how an agent might learn to play these games without such knowledge.

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 2.6 Examples of task environments and their characteristics.

The characteristics depend on how the task environment is defined. We have listed the medical-diagnosis task as single-agent because the disease process in a patient is not profitably modeled as an agent; but a medical-diagnosis system might also have to deal with recalcitrant patients and skeptical staff, so the environment could have a multiagent aspect. Furthermore, medical diagnosis is episodic if one conceives of the task as selecting a diagnosis given a list of symptoms; the problem is sequential if the task can include proposing a series of tests, evaluating progress over the course of treatment, and so on. Also, many environments are episodic at higher levels than the agent's individual actions. For example, a chess tournament consists of a sequence of games; each game is an episode because (by and large) the contribution of the moves in one game to the agent's overall performance is not affected by the moves in its previous game. On the other hand, decision making within a single game is certainly sequential. You can refer to (https://github.com/aimacode) for the implementations of a number of environments with a general-purpose environment simulator that places one or more agents in a simulated environment.

2.4 The structure of Agents

The job of AI is to design an agent program that implements the agent function— the mapping from percepts to actions. Assuming that the program will run on some sort of computing device with physical sensors and actuators—we call this the architecture:

agent = architecture + program

Obviously, the chosen program has to be one that is appropriate for the architecture. If the program is going to recommend actions like *Walk*, the architecture had better have legs. The architecture might be just an ordinary PC, or it might be a robotic car with several on-board computers, cameras, and other sensors. In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the actuators as they are generated.

1. Agent programs

The agent programs will take the current percept as input from the sensors and return an action to the actuators. There is a difference between the *agent program*, which takes the current percept as input, and the *agent function*, which takes the entire percept history. The agent program takes just the current percept as input because nothing more is available from the environment; if the agent's actions need to depend on the entire percept sequence, the agent will have to remember the percepts.

Figure 2.7 shows a rather trivial agent program that keeps track of the percept sequence and then uses it to index into a table of actions to decide what to do. The table—an example of which is given for the vacuum world in **Figure 2.3**—represents explicitly the agent function that the agent program embodies. To build a rational agent in this way, we as designers must construct a table that contains the appropriate action for every possible percept sequence.

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

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

Figure 2.7 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.

It is instructive to consider why the table-driven approach to agent construction is doomed to failure. Let $\boldsymbol{P}$ be the set of possible percepts and let $\boldsymbol{T}$ be the lifetime of the agent (the total number of percepts it will receive). The lookup table will contain $\sum_{t=1}^{T} |\boldsymbol{P}|^{t}$ entries. Consider the automated taxi: the visual input from a single camera comes in at the rate of roughly 27 megabytes per second (30 frames per second, 640 × 480 pixels with 24 bits of color information). This gives a lookup table with over 10250,000,000,000 entries for an hour's driving. Even the lookup table for chess—a tiny, well-behaved fragment of the real world—would have at least 10150 entries. The daunting size of these tables means that:

- No physical agent in this universe will have the space to store the table,
- The designer would not have time to create the table,
- No agent could ever learn all the right table entries from its experience, and
- Even if the environment is simple enough to yield a feasible table size, the designer still has no guidance about how to fill in the table entries.

The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from a vast table.

There are four basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

- Simple reflex agents;
- Model-based reflex agents;
- Goal-based agents; and
- Utility-based agents.

Each kind of agent program combines particular components in particular ways to generate actions. The next section explains how to convert all these agents into *learning* agents that can improve the performance of their components so as to generate better actions.

2. Simple reflex agents

The simplest kind of agent is the **simple reflex agent**. These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in **Figure 2.3** is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt. An agent program for this agent is shown in **Figure 2.8**.

```
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
```

Figure 2.8 The agent program for a simple reflex agent in the two-state vacuum environment. This program implements the agent function tabulated in Figure 2.3.

Notice that the vacuum agent program is very small compared to the corresponding table. The most obvious reduction comes from ignoring the percept history, which cuts down the number of possibilities from 4^T to just 4. A further, small reduction comes from the fact that when the current square is dirty, the action does not depend on the location.

Simple reflex behaviors occur even in more complex environments. Imagine yourself as the driver of the automated taxi. If the car in front brakes and its brake lights come on, then you should notice this and initiate braking. In other words, some processing is done on the visual input to establish the condition we call "The car in front is braking." Then, this triggers some established connection in the agent program to the action "initiate braking." We call such a connection a **condition–action rule**, written as

if *car-in-front-is-braking* **then** *initiate-braking*.

The program in **Figure 2.8** is specific to one particular vacuum environment. A more general and flexible approach is first to build a general-purpose interpreter for condition– action rules and then to create rule sets

for specific task environments. **Figure 2.9** gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action.

We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process. The agent program, which is also very simple, is shown in **Figure 2.10**. The INTERPRET-INPUT function generates an abstracted description of the current state from the percept, and the RULE-MATCH function returns the first rule in the set of rules that matches the given state description.

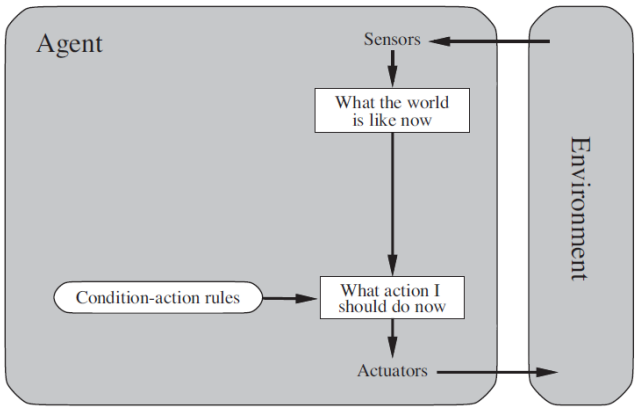

Figure 2.9 Schematic diagram of a simple reflex agent.

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

Simple reflex agents have the admirable property of **being simple**, but they turn out to be of **limited intelligence**. The agent in **Figure 2.10** will work *only if the correct decision can be made on the basis of only the current percept—that is, only if the environment is fully observable.* Even a little bit of unobservability can cause serious trouble.

- **Example:**

Suppose that a simple reflex vacuum agent is deprived of its location sensor and has only a dirt sensor. Such an agent has just two possible percepts: [Dirty] and [Clean]. It can Suck in response to [Dirty]; what should it do in response to [Clean]? Moving Left fails (forever) if it happens to start in square A, and moving Right fails (forever) if it happens to start in square B. Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments.

- **Randomization**:

Escape from infinite loops is possible if the agent can **randomize** its actions. For example, if the vacuum agent perceives [Clean], it might flip a coin to choose between Left and Right. Eventually, the agent will be able to reach the other square. Then, if that square is dirty, the agent will clean it and the task will be complete. Hence, a *randomized simple reflex agent* might outperform a *deterministic simple reflex agent*.

3. Model-based reflex agents

The most effective way to handle partial observability is for the agent 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.

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program:

- First, we need some information about how the world evolves independently of the agent—for example, that an overtaking car generally will be closer behind than it was a moment ago.
- Second, we need some information about how the agent's own actions affect the world—for example, that when the agent turns the steering wheel clockwise, the car turns to the right.

This knowledge about "*how the world works*"—whether implemented in simple Boolean circuits or in complete scientific theories—is called a **model of the world**. An agent that uses such a model is called a **model-based agent.**

Figure 2.11 gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works. The agent program is shown in **Figure 2.12**. The interesting part is the function UPDATE-STATE, which is responsible for creating the new internal state description. The details of how models and states are represented vary widely depending on the type of environment and the particular technology used in the agent design.

Regardless of the kind of representation used, it is seldom possible for the agent to determine the current state of a partially observable environment *exactly*. Instead, the box labelled "what the world is like now" (Figure 2.11) represents the agent's "best guess". Uncertainty about the current state may be unavoidable, but the agent still has to make a decision.

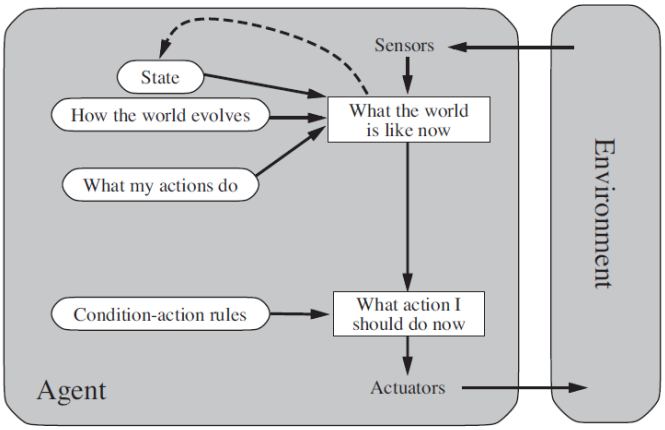

Figure 2.11 A model-based reflex agent.

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 2.12 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.

4. Goal-based agents

Knowing something about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, as well as a current state description, the agent needs some sort of **goal** information that describes situations that are desirable. The agent program can combine this with the model to choose actions that achieve the goal. **Figure 2.13** shows the goal-based agent's structure.

Sometimes goal-based action selection is straightforward—for example, when goal satisfaction results immediately from a single action. Sometimes it will be trickier—for example, when the agent has to consider long sequences of twists and turns in order to find a way to achieve the goal.

Notice that decision making of this kind is fundamentally different from the condition– action rules described earlier, in that it involves consideration of the future—both "What will happen if I do such-and-such?" and "Will that make me happy?".

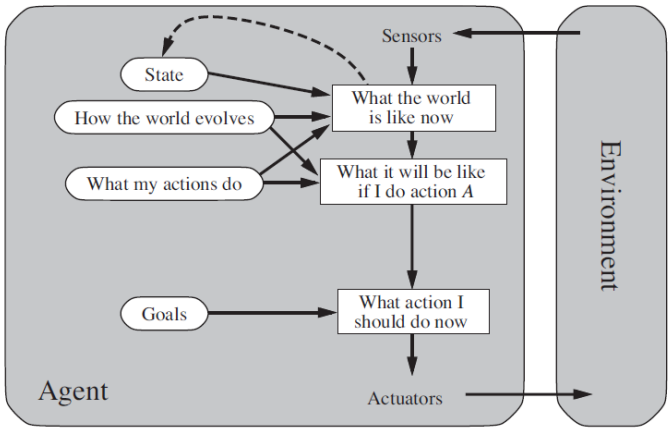

Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. If it starts to rain, the agent can update its knowledge of how effectively its brakes will operate; this will automatically cause all of the relevant behaviors to be altered to suit the new conditions. For the reflex agent, on the other hand, we would have to rewrite many condition–action rules. The goal-based agent's behavior can easily be changed to go to a different destination, simply by specifying that destination as the goal. The reflex agent's rules for when to turn and when to go straight will work only for a single destination; they must all be replaced to go somewhere new.

5. Utility-based agents

Goals alone are not enough to generate high-quality behavior in most environments. *For example*, many action sequences will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between **"happy"** and **"unhappy"** states. The term **utility** instead is used as a general performance measure that allows a comparison of different world states according to exactly how happy they would make the agent.

The performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of getting to the taxi's destination. An agent's **utility function** is essentially an internalization of the performance measure. If the internal utility function and the external performance measure are in agreement, then an agent that chooses actions to maximize its utility will be rational according to the external performance measure.

Like goal-based agents, a utility-based agent has many advantages in terms of flexibility and learning. Furthermore, in two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions. *First,* when there are **conflicting goals**, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff. *Second*, when there are **several goals** that the

agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.

Partial observability and stochasticity are ubiquitous in the real world, and so, therefore, is decision making under uncertainty. Technically speaking, a rational utility-based agent chooses the action that maximizes the **expected utility** of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome. The utility-based agent structure is shown in Figure 2.14.

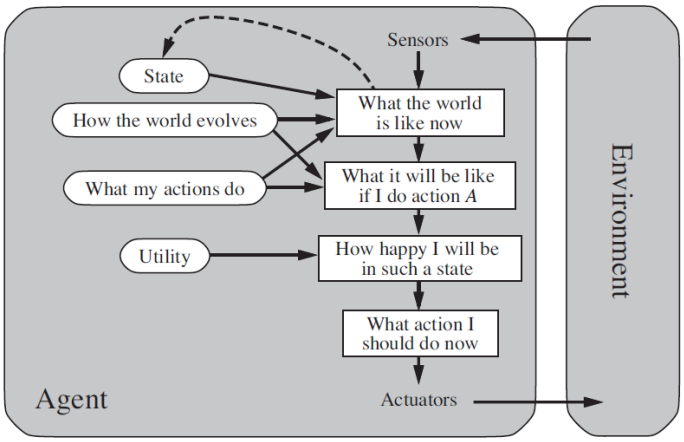

Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

6. Learning agents

A learning agent can be divided into four conceptual components, as shown in Figure 2.15.

- **Learning element**: which is responsible for making improvements.
- **Performance element**: which is responsible for selecting external actions.
- **Critic**: tells the learning element how well the agent is doing with respect to a fixed performance standard.
- **Problem generator**: It is responsible for suggesting actions that will lead to new and informative experiences.

To make the overall design more concrete, let us return to the *automated taxi example*. The **performance element** consists of whatever collection of knowledge and procedures the taxi has for selecting its driving actions. The taxi goes out on the road and drives, using this performance element. The **critic** observes the world and passes information along to the learning element. For example, after the taxi makes a quick left turn across three lanes of traffic, the critic observes the shocking language used by other drivers. From this experience, the learning element is able to formulate a rule saying this was a bad action, and the **performance element** is modified by

installation of the new rule. 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.

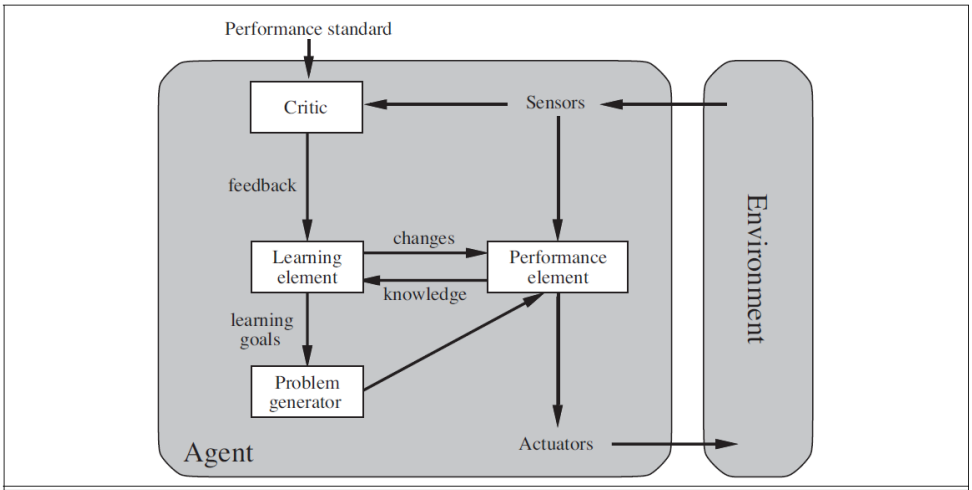

Figure 2.15 A general learning agent.

To sum up, the **performance element** is what we have previously considered to be the entire agent: it takes in percepts and decides on actions. The **learning element** uses 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. Whereas, the **problem generator**'s job is to suggest new exploratory actions with the aim of discovering much better actions for the long run.

In summary, agents have a variety of components, and those components can be represented in many ways within the agent program, so there appears to be great variety among learning methods. There is, however, a single unifying theme. Learning in intelligent agents can be summarized as a process of modification of each component of the agent to bring the components into closer agreement with the available feedback information, thereby improving the overall performance of the agent.

7. How the components of agent programs work

Agent programs were described (in very high-level terms) as consisting of various components, whose function it is to answer questions such as: "What is the world like now?" "What action should I do now?" "What do my actions do?" But roughly speaking, we can place these components' representations along an axis of increasing complexity and expressive power—**atomic**, **factored**, and **structured**. Figure 2.16 provides schematic depictions of how those transitions might be represented.

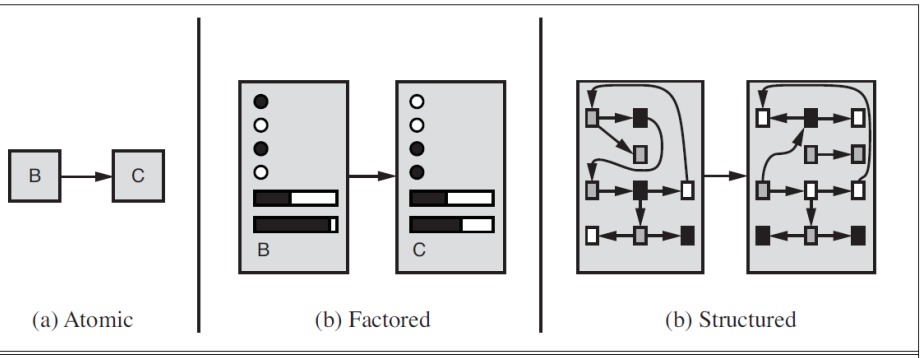

Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed 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.

- ***Atomic representation***: each state of the world is indivisible—it has no internal structure. Consider the problem of finding a driving route from one end of a country to the other via some sequence of cities. For the purposes of solving this problem, it may suffice to reduce the state of world to just the name of the city we are in—a single atom of knowledge; a "black box" whose only discernible property is that of being identical to or different from another black box. The algorithms underlying **search** and **game-playing**, **Hidden Markov models**, and **Markov decision processes** all work with atomic representations—or, at least, they treat representations *as if* they were atomic.
- ***Factored representation***: splits up each state into a fixed set of variables or attributes, each of which can have a value. Now we can consider a higher-fidelity description for the same problem, where we need to be concerned with more than just atomic location in one city or another; we might need to pay attention to how much gas is in the tank, our current GPS coordinates, whether or not the oil warning light is working, and so on. With factored representations, we can also represent *uncertainty*—for example, ignorance about the amount of gas in the tank can be represented by leaving that attribute blank. Many important areas of AI are based on factored representations, including **constraint satisfaction** algorithms, **propositional logic**, **planning**, **Bayesian networks**, and the **machine learning** algorithms.
- ***Structured representation***: For many purposes, we need to understand the world as having things in it that are related to each other, not just variables with values. For example, we might notice that a large truck ahead of us is reversing into the driveway of a dairy farm but a cow has got loose and is blocking the truck's path. A factored representation is unlikely to be pre-equipped with the attribute *TruckAheadBackingIntoDairyFarmDrivewayBlockedByLooseCow* with value true or false. However, a **structured representation**, in which objects such as cows and trucks and their various and varying relationships can be described explicitly. Structured representations underlie **relational databases** and **first-order logic**, **first-order probability models**, **knowledge-based learning** and much of **natural language understanding.** In fact, almost everything that humans express in natural language concerns objects and their relationships.