2 INTELLIGENT AGENTS

In which we discuss what an intelligent agent does, how it is related to its environment, how it is evaluated, and how we might go about building one.

2.1 INTRODUCTION

An **agent** is anything that can be viewed as **perceiving** its environment through **sensors** and **acting** upon that environment through **effectors**. A human agent has eyes, ears, and other organs for sensors, and hands, legs, mouth, and other body parts for effectors. A robotic agent substitutes cameras and infrared range finders for the sensors and various motors for the effectors. A software agent has encoded bit strings as its percepts and actions. A generic agent is diagrammed in Figure 2.1.

Our aim in this book is to design agents that do a good job of acting on their environment. First, we will be a little more precise about what we mean by a good job. Then we will talk about different designs for successful agents—filling in the question mark in Figure 2.1. We discuss some of the general principles used in the design of agents throughout the book, chief among which is the principle that agents should *know* things. Finally, we show how to couple an agent to an environment and describe several kinds of environments.

2.2 HOW AGENTS SHOULD ACT

RATIONAL AGENT

A **rational agent** is one that does the right thing. Obviously, this is better than doing the wrong thing, but what does it mean? As a first approximation, we will say that the right action is the one that will cause the agent to be most successful. That leaves us with the problem of deciding *how* and *when* to evaluate the agent's success.

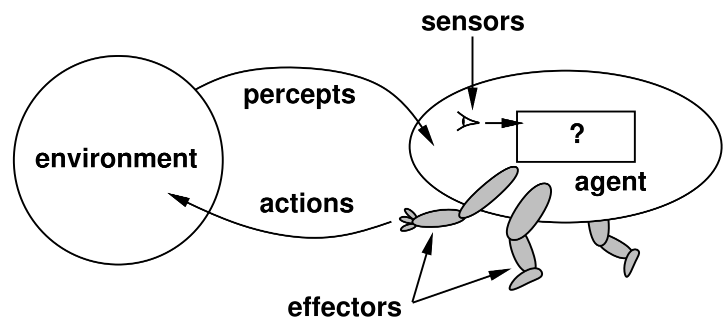


Figure 2.1 Agents interact with environments through sensors and effectors.

PERFORMANCE MEASURE

We use the term **performance measure** for the *how*—the criteria that determine how successful an agent is. Obviously, there is not one fixed measure suitable for all agents. We could ask the agent for a subjective opinion of how happy it is with its own performance, but some agents would be unable to answer, and others would delude themselves. (Human agents in particular are notorious for "sour grapes"—saying they did not really want something after they are unsuccessful at getting it.) Therefore, we will insist on an objective performance measure imposed by some authority. In other words, we as outside observers establish a standard of what it means to be successful in an environment and use it to measure the performance of agents.

As an example, consider the case of an agent that is supposed to vacuum a dirty floor. A plausible performance measure would be the amount of dirt cleaned up in a single eight-hour shift. A more sophisticated performance measure would factor in the amount of electricity consumed and the amount of noise generated as well. A third performance measure might give highest marks to an agent that not only cleans the floor quietly and efficiently, but also finds time to go windsurfing at the weekend.[1]

The *when* of evaluating performance is also important. If we measured how much dirt the agent had cleaned up in the first hour of the day, we would be rewarding those agents that start fast (even if they do little or no work later on), and punishing those that work consistently. Thus, we want to measure performance over the long run, be it an eight-hour shift or a lifetime.

OMNISCIENCE

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. Consider the following example: I am walking along the Champs Elysées one day and I see an old friend across the street. There is no traffic nearby and I'm not otherwise engaged, so, being rational, I start to cross the street. Meanwhile, at 33,000 feet, a cargo door falls off a passing airliner,[2] and before I make it to the other side of the street I am flattened. Was I irrational to cross the street? It is unlikely that my obituary would read "Idiot attempts to cross

[1] There is a danger here for those who establish performance measures: you often get what you ask for. That is, if you measure success by the amount of dirt cleaned up, then some clever agent is bound to bring in a load of dirt each morning, quickly clean it up, and get a good performance score. What you really want to measure is how clean the floor is, but determining that is more difficult than just weighing the dirt cleaned up.

[2] See N. Henderson, "New door latches urged for Boeing 747 jumbo jets," *Washington Post*, 8/24/89.

street." Rather, this points out that rationality is concerned with *expected* success *given what has been perceived*. Crossing the street was rational because most of the time the crossing would be successful, and there was no way I could have foreseen the falling door. Note that another agent that was equipped with radar for detecting falling doors or a steel cage strong enough to repel them would be more successful, but it would not be any more rational.

In other words, we cannot blame an agent for failing to take into account something it could not perceive, or for failing to take an action (such as repelling the cargo door) that it is incapable of taking. But relaxing the requirement of perfection is not just a question of being fair to agents. The point is that if we specify that an intelligent agent should always do what is *actually* the right thing, it will be impossible to design an agent to fulfill this specification—unless we improve the performance of crystal balls.

In summary, what is rational at any given time depends on four things:

- The performance measure that defines degree of success.
- Everything that the agent has perceived so far. We will call this complete perceptual history the **percept sequence**.

PERCEPT SEQUENCE

- What the agent knows about the environment.
- The actions that the agent can perform.

IDEAL RATIONAL AGENT

This leads to a definition of an **ideal rational agent**: *For each possible percept sequence, an ideal rational agent should do whatever action is expected to maximize its performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.*

We need to look carefully at this definition. At first glance, it might appear to allow an agent to indulge in some decidedly underintelligent activities. For example, if an agent does not look both ways before crossing a busy road, then its percept sequence will not tell it that there is a large truck approaching at high speed. The definition seems to say that it would be OK for it to cross the road. In fact, this interpretation is wrong on two counts. First, it would not be rational to cross the road: the risk of crossing without looking is too great. Second, an ideal rational agent would have chosen the "looking" action before stepping into the street, because looking helps maximize the expected performance. Doing actions *in order to obtain useful information* is an important part of rationality and is covered in depth in Chapter 16.

The notion of an agent is meant to be a tool for analyzing systems, not an absolute characterization that divides the world into agents and non-agents. Consider a clock. It can be thought of as just an inanimate object, or it can be thought of as a simple agent. As an agent, most clocks always do the right action: moving their hands (or displaying digits) in the proper fashion. Clocks are a kind of degenerate agent in that their percept sequence is empty; no matter what happens outside, the clock's action should be unaffected.

Well, this is not quite true. If the clock and its owner take a trip from California to Australia, the right thing for the clock to do would be to turn itself back six hours. We do not get upset at our clocks for failing to do this because we realize that they are acting rationally, given their lack of perceptual equipment.[3]

[3] One of the authors still gets a small thrill when his computer successfully resets itself at daylight savings time.

The ideal mapping from percept sequences to actions

Once we realize that an agent's behavior depends only on its percept sequence to date, then we can describe any particular agent by making a table of the action it takes in response to each possible percept sequence. (For most agents, this would be a very long list—infinite, in fact, unless we place a bound on the length of percept sequences we want to consider.) Such a list is called

MAPPING

a **mapping** from percept sequences to actions. We can, in principle, find out which mapping correctly describes an agent by trying out all possible percept sequences and recording which actions the agent does in response. (If the agent uses some randomization in its computations, then we would have to try some percept sequences several times to get a good idea of the agent's

IDEAL MAPPINGS

average behavior.) And if mappings describe agents, then **ideal mappings** describe ideal agents. *Specifying which action an agent ought to take in response to any given percept sequence provides a design for an ideal agent.*

This does not mean, of course, that we have to create an explicit table with an entry for every possible percept sequence. It is possible to define a specification of the mapping without exhaustively enumerating it. Consider a very simple agent: the square-root function on a calculator. The percept sequence for this agent is a sequence of keystrokes representing a number, and the action is to display a number on the display screen. The ideal mapping is that when the percept is a positive number x, the right action is to display a positive number z such that $z^2 \approx x$, accurate to, say, 15 decimal places. This specification of the ideal mapping does not require the designer to actually construct a table of square roots. Nor does the square-root function have to use a table to behave correctly: Figure 2.2 shows part of the ideal mapping and a simple program that implements the mapping using Newton's method.

The square-root example illustrates the relationship between the ideal mapping and an ideal agent design, for a very restricted task. Whereas the table is very large, the agent is a nice, compact program. It turns out that it is possible to design nice, compact agents that implement

Percept x	Action z
1.0	1.000000000000000
1.1	1.048808848170152
1.2	1.095445115010332
1.3	1.140175425099138
1.4	1.183215956619923
1.5	1.224744871391589
1.6	1.264911064067352
1.7	1.303840481040530
1.8	1.341640786499874
1.9	1.378404875209022
⋮	⋮

function SQRT(x)
  $z \leftarrow 1.0$ /* *initial guess* */
  repeat until $|z^2 - x| < 10^{-15}$
    $z \leftarrow z - (z^2 - x)/(2z)$
  end
  return z

Figure 2.2 Part of the ideal mapping for the square-root problem (accurate to 15 digits), and a corresponding program that implements the ideal mapping.

the ideal mapping for much more general situations: agents that can solve a limitless variety of tasks in a limitless variety of environments. Before we discuss how to do this, we need to look at one more requirement that an intelligent agent ought to satisfy.

Autonomy

AUTONOMY

There is one more thing to deal with in the definition of an ideal rational agent: the "built-in knowledge" part. If the agent's actions are based completely on built-in knowledge, such that it need pay no attention to its percepts, then we say that the agent lacks **autonomy**. For example, if the clock manufacturer was prescient enough to know that the clock's owner would be going to Australia at some particular date, then a mechanism could be built in to adjust the hands automatically by six hours at just the right time. This would certainly be successful behavior, but the intelligence seems to belong to the clock's designer rather than to the clock itself.

An agent's behavior can be based on both its own experience and the built-in knowledge used in constructing the agent for the particular environment in which it operates. *A system is autonomous*[4] *to the extent that its behavior is determined by its own experience.* It would be too stringent, though, to require complete autonomy from the word go: when the agent has had little or no experience, it would have to act randomly unless the designer gave some assistance. So, just as evolution provides animals with enough built-in reflexes so that they can survive long enough to learn for themselves, it would be reasonable to provide an artificial intelligent agent with some initial knowledge as well as an ability to learn.

Autonomy not only fits in with our intuition, but it is an example of sound engineering practices. An agent that operates on the basis of built-in assumptions will only operate successfully when those assumptions hold, and thus lacks flexibility. Consider, for example, the lowly dung beetle. After digging its nest and laying its eggs, it fetches a ball of dung from a nearby heap to plug the entrance; if the ball of dung is removed from its grasp *en route*, the beetle continues on and pantomimes plugging the nest with the nonexistent dung ball, never noticing that it is missing. Evolution has built an assumption into the beetle's behavior, and when it is violated, unsuccessful behavior results. A truly autonomous intelligent agent should be able to operate successfully in a wide variety of environments, given sufficient time to adapt.

2.3 STRUCTURE OF INTELLIGENT AGENTS

AGENT PROGRAM

ARCHITECTURE

So far we have talked about agents by describing their *behavior*—the action that is performed after any given sequence of percepts. Now, we will have to bite the bullet and talk about how the insides work. The job of AI is to design the **agent program**: a function that implements the agent mapping from percepts to actions. We assume this program will run on some sort of computing device, which we will call the **architecture**. Obviously, the program we choose has

[4] The word "autonomous" has also come to mean something like "not under the immediate control of a human," as in "autonomous land vehicle." We are using it in a stronger sense.

to be one that the architecture will accept and run. The architecture might be a plain computer, or it might include special-purpose hardware for certain tasks, such as processing camera images or filtering audio input. It might also include software that provides a degree of insulation between the raw computer and the agent program, so that we can program at a higher level. 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 effectors as they are generated. The relationship among agents, architectures, and programs can be summed up as follows:

agent = architecture + program

Most of this book is about designing agent programs, although Chapters 24 and 25 deal directly with the architecture.

Before we design an agent program, we must have a pretty good idea of the possible percepts and actions, what goals or performance measure the agent is supposed to achieve, and what sort of environment it will operate in.[5] These come in a wide variety. Figure 2.3 shows the basic elements for a selection of agent types.

It may come as a surprise to some readers that we include in our list of agent types some programs that seem to operate in the entirely artificial environment defined by keyboard input and character output on a screen. "Surely," one might say, "this is not a real environment, is it?" In fact, what matters is not the distinction between "real" and "artificial" environments, but the complexity of the relationship among the behavior of the agent, the percept sequence generated by the environment, and the goals that the agent is supposed to achieve. Some "real" environments are actually quite simple. For example, a robot designed to inspect parts as they come by on a conveyer belt can make use of a number of simplifying assumptions: that the lighting is always just so, that the only thing on the conveyer belt will be parts of a certain kind, and that there are only two actions—accept the part or mark it as a reject.

SOFTWARE AGENTS
SOFTBOTS

In contrast, some **software agents** (or software robots or **softbots**) exist in rich, unlimited domains. Imagine a softbot designed to fly a flight simulator for a 747. The simulator is a very detailed, complex environment, and the software agent must choose from a wide variety of actions in real time. Or imagine a softbot designed to scan online news sources and show the interesting items to its customers. To do well, it will need some natural language processing abilities, it will need to learn what each customer is interested in, and it will need to dynamically change its plans when, for example, the connection for one news source crashes or a new one comes online.

Some environments blur the distinction between "real" and "artificial." In the ALIVE environment (Maes *et al.*, 1994), software agents are given as percepts a digitized camera image of a room where a human walks about. The agent processes the camera image and chooses an action. The environment also displays the camera image on a large display screen that the human can watch, and superimposes on the image a computer graphics rendering of the software agent. One such image is a cartoon dog, which has been programmed to move toward the human (unless he points to send the dog away) and to shake hands or jump up eagerly when the human makes certain gestures.

---

[5] For the acronymically minded, we call this the PAGE (Percepts, Actions, Goals, Environment) description. Note that the goals do *not* necessarily have to be represented within the agent; they simply describe the performance measure by which the agent design will be judged.

Agent Type	Percepts	Actions	Goals	Environment
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
Satellite image analysis system	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
Refinery controller	Temperature, pressure readings	Open, close valves; adjust temperature	Maximize purity, yield, safety	Refinery
Interactive English tutor	Typed words	Print exercises, suggestions, corrections	Maximize student's score on test	Set of students

Figure 2.3 Examples of agent types and their PAGE descriptions.

The most famous artificial environment is the Turing Test environment, in which the whole point is that real and artificial agents are on equal footing, but the environment is challenging enough that it is very difficult for a software agent to do as well as a human. Section 2.4 describes in more detail the factors that make some environments more demanding than others.

Agent programs

We will be building intelligent agents throughout the book. They will all have the same skeleton, namely, accepting percepts from an environment and generating actions. The early versions of agent programs will have a very simple form (Figure 2.4). Each will use some internal data structures that will be updated as new percepts arrive. These data structures are operated on by the agent's decision-making procedures to generate an action choice, which is then passed to the architecture to be executed.

There are two things to note about this skeleton program. First, even though we defined the agent mapping as a function from percept *sequences* to actions, the agent program receives only a single percept as its input. It is up to the agent to build up the percept sequence in memory, if it so desires. In some environments, it is possible to be quite successful without storing the percept sequence, and in complex domains, it is infeasible to store the complete sequence.

```
function SKELETON-AGENT( percept) returns action
   static: memory, the agent's memory of the world

   memory ← UPDATE-MEMORY(memory, percept)
   action ← CHOOSE-BEST-ACTION(memory)
   memory ← UPDATE-MEMORY(memory, action)
   return action
```

Figure 2.4 A skeleton agent. On each invocation, the agent's memory is updated to reflect the new percept, the best action is chosen, and the fact that the action was taken is also stored in memory. The memory persists from one invocation to the next.

Second, the goal or performance measure is *not* part of the skeleton program. This is because the performance measure is applied externally to judge the behavior of the agent, and it is often possible to achieve high performance without explicit knowledge of the performance measure (see, e.g., the square-root agent).

Why not just look up the answers?

Let us start with the simplest possible way we can think of to write the agent program—a lookup table. Figure 2.5 shows the agent program. It operates by keeping in memory its entire percept sequence, and using it to index into *table*, which contains the appropriate action for all possible percept sequences.

It is instructive to consider why this proposal is doomed to failure:

1. The table needed for something as simple as an agent that can only play chess would be about 35^{100} entries.
2. It would take quite a long time for the designer to build the table.
3. The agent has no autonomy at all, because the calculation of best actions is entirely built-in. So if the environment changed in some unexpected way, the agent would be lost.

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

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

Figure 2.5 An agent based on a prespecified lookup table. It keeps track of the percept sequence and just looks up the best action.

4. Even if we gave the agent a learning mechanism as well, so that it could have a degree of autonomy, it would take forever to learn the right value for all the table entries.

Despite all this, TABLE-DRIVEN-AGENT *does* do what we want: it implements the desired agent mapping. It is not enough to say, "It can't be intelligent;" the point is to understand why an agent that *reasons* (as opposed to looking things up in a table) can do even better by avoiding the four drawbacks listed here.

An example

At this point, it will be helpful to consider a particular environment, so that our discussion can become more concrete. Mainly because of its familiarity, and because it involves a broad range of skills, we will look at the job of designing an automated taxi driver. We should point out, before the reader becomes alarmed, that such a system is currently somewhat beyond the capabilities of existing technology, although most of the components are available in some form.[6] The full driving task is extremely *open-ended*—there is no limit to the novel combinations of circumstances that can arise (which is another reason why we chose it as a focus for discussion).

We must first think about the percepts, actions, goals and environment for the taxi. They are summarized in Figure 2.6 and discussed in turn.

Agent Type	Percepts	Actions	Goals	Environment
Taxi driver	Cameras, speedometer, GPS, sonar, microphone	Steer, accelerate, brake, talk to passenger	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers

Figure 2.6 The taxi driver agent type.

The taxi will need to know where it is, what else is on the road, and how fast it is going. This information can be obtained from the **percepts** provided by one or more controllable TV cameras, the speedometer, and odometer. To control the vehicle properly, especially on curves, it should have an accelerometer; it will also need to know the mechanical state of the vehicle, so it will need the usual array of engine and electrical system sensors. It might have instruments that are not available to the average human driver: a satellite global positioning system (GPS) to give it accurate position information with respect to an electronic map; or infrared or sonar sensors to detect distances to other cars and obstacles. Finally, it will need a microphone or keyboard for the passengers to tell it their destination.

The **actions** available to a taxi driver will be more or less the same ones available to a human driver: control over the engine through the gas pedal and control over steering and braking. In addition, it will need output to a screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles.

6 See page 26 for a description of an existing driving robot, or look at the conference proceedings on Intelligent Vehicle and Highway Systems (IVHS).

What **performance measure** would we like our automated driver to aspire to? Desirable qualities include getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time and/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 there will be trade-offs involved.

Finally, were this a real project, we would need to decide what kind of driving **environment** the taxi will face. Should it operate on local roads, or also on freeways? Will it be in Southern California, where snow is seldom a problem, or in Alaska, where it seldom is not? Will it always be driving on the right, or might we want it to be flexible enough to drive on the left in case we want to operate taxis in Britain or Japan? Obviously, the more restricted the environment, the easier the design problem.

Now we have to decide how to build a real program to implement the mapping from percepts to action. We will find that different aspects of driving suggest different types of agent program. We will consider four types of agent program:

- Simple reflex agents
- Agents that keep track of the world
- Goal-based agents
- Utility-based agents

Simple reflex agents

The option of constructing an explicit lookup table is out of the question. The visual input from a single camera comes in at the rate of 50 megabytes per second (25 frames per second, 1000×1000 pixels with 8 bits of color and 8 bits of intensity information). So the lookup table for an hour would be $2^{60 \times 60 \times 50M}$ entries.

However, we can summarize portions of the table by noting certain commonly occurring input/output associations. For example, if the car in front brakes, and its brake lights come on, then the driver 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**[7] written as

CONDITION–ACTION RULE

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

Humans also have many such connections, some of which are learned responses (as for driving) and some of which are innate reflexes (such as blinking when something approaches the eye). In the course of the book, we will see several different ways in which such connections can be learned and implemented.

Figure 2.7 gives the structure of a simple reflex agent in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action. (Do not worry if this seems trivial; it gets more interesting shortly.) We use rectangles to denote

[7] Also called **situation–action rules**, **productions**, or **if–then rules**. The last term is also used by some authors for logical implications, so we will avoid it altogether.

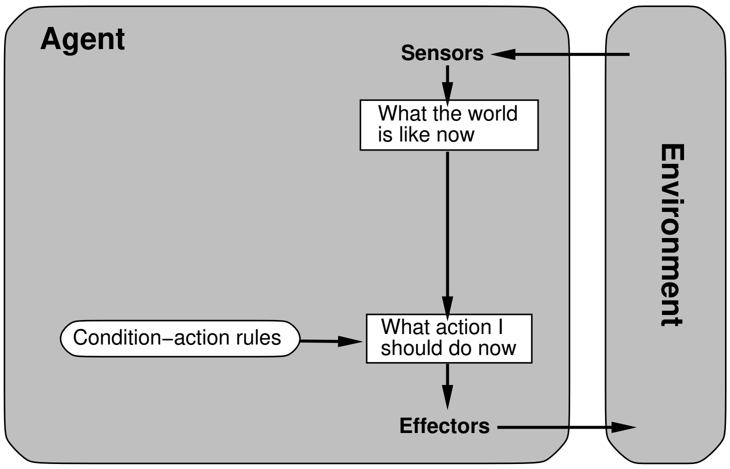


Figure 2.7 Schematic diagram of a simple reflex agent.

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

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

Figure 2.8 A simple reflex agent. It works by finding a rule whose condition matches the current situation (as defined by the percept) and then doing the action associated with that rule.

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 2.8. 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. Although such agents can be implemented very efficiently (see Chapter 10), their range of applicability is very narrow, as we shall see.

Agents that keep track of the world

The simple reflex agent described before will work only if the correct decision can be made on the basis of the current percept. If the car in front is a recent model, and has the centrally mounted brake light now required in the United States, then it will be possible to tell if it is braking from a single image. Unfortunately, older models have different configurations of tail lights, brake lights, and turn-signal lights, and it is not always possible to tell if the car is braking. Thus, even for the simple braking rule, our driver will have to maintain some sort of **internal state** in order to choose an action. Here, the internal state is not too extensive—it just needs the previous frame from the camera to detect when two red lights at the edge of the vehicle go on or off simultaneously.

INTERNAL STATE

Consider the following more obvious case: from time to time, the driver looks in the rear-view mirror to check on the locations of nearby vehicles. When the driver is not looking in the mirror, the vehicles in the next lane are invisible (i.e., the states in which they are present and absent are indistinguishable); but in order to decide on a lane-change maneuver, the driver needs to know whether or not they are there.

The problem illustrated by this example arises because the sensors do not provide access to the complete state of the world. In such cases, the agent may need to maintain some internal state information in order to distinguish between world states that generate the same perceptual input but nonetheless are significantly different. Here, "significantly different" means that different actions are appropriate in the two states.

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 changes lanes to the right, there is a gap (at least temporarily) in the lane it was in before, or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago.

Figure 2.9 gives the structure of the reflex agent, showing how the current percept is combined with the old internal state to generate the updated description of the current state. The agent program is shown in Figure 2.10. The interesting part is the function UPDATE-STATE, which is responsible for creating the new internal state description. As well as interpreting the new percept in the light of existing knowledge about the state, it uses information about how the world evolves to keep track of the unseen parts of the world, and also must know about what the agent's actions do to the state of the world. Detailed examples appear in Chapters 7 and 17.

Goal-based agents

Knowing 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, right, or go straight on. The right 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, which describes situations that are desirable—for example, being at the passenger's destination. The agent program can combine this with information about the results of possible actions (the same information as was used to update internal state in the reflex agent) in order to choose actions that achieve the goal. Sometimes this will be simple, when goal satisfaction results immediately from a single action; sometimes, it will be more tricky, when the agent has to consider long sequences of twists and turns to find a way to achieve the goal. **Search** (Chapters 3 to 5) and **planning** (Chapters 11 to 13) are the subfields of AI devoted to finding action sequences that do achieve the agent's goals.

GOAL

SEARCH

PLANNING

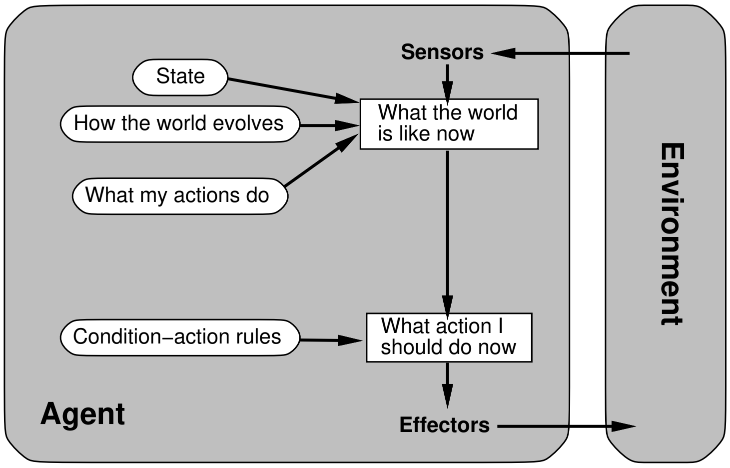


Figure 2.9 A reflex agent with internal state.

function REFLEX-AGENT-WITH-STATE(*percept*) **returns** *action*
  static: *state*, a description of the current world state
    rules, a set of condition-action rules

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

Figure 2.10 A reflex agent with internal state. It works by finding a rule whose condition matches the current situation (as defined by the percept and the stored internal state) and then doing the action associated with that rule.

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?" In the reflex agent designs, this information is not explicitly used, because the designer has precomputed the correct action for various cases. The reflex agent brakes when it sees brake lights. A goal-based agent, in principle, could reason that if the car in front has its brake lights on, it will slow down. From the way the world usually evolves, the only action that will achieve the goal of not hitting other cars is to brake. Although the goal-based agent appears less efficient, it is far more flexible. 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 a large number of condition–action rules. Of course, the goal-based agent is also more flexible with respect to reaching different destinations. Simply by specifying a new destination, we can get the goal-based agent to come up with a new behavior. The reflex agent's rules for when to turn and when to go straight will only work for a single destination; they must all be replaced to go somewhere new.

Figure 2.11 shows the goal-based agent's structure. Chapter 13 contains detailed agent programs for goal-based agents.

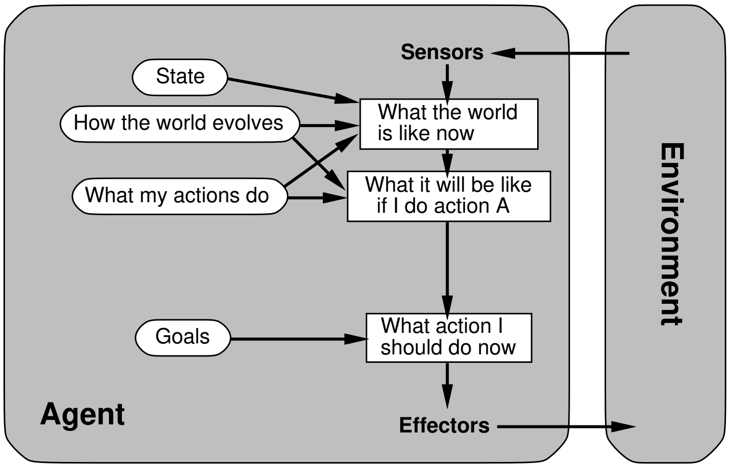


Figure 2.11 An agent with explicit goals.

Utility-based agents

Goals alone are not really enough to generate high-quality behavior. For example, there are many action sequences that 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 distinction between "happy" and "unhappy" states, whereas a more general performance measure should allow a comparison of different world states (or sequences of states) according to exactly how happy they would make the agent if they could be achieved. Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher **utility** for the agent.[8]

UTILITY

Utility is therefore a function that maps a state[9] onto a real number, which describes the associated degree of happiness. A complete specification of the utility function allows rational decisions in two kinds of cases where goals have trouble. First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate trade-off. Second, when there are several goals that the agent can aim for, none

[8] The word "utility" here refers to "the quality of being useful," not to the electric company or water works.

[9] Or sequence of states, if we are measuring the utility of an agent over the long run.

of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed up against the importance of the goals.

In Chapter 16, we show that any rational agent can be described as possessing a utility function. An agent that possesses an *explicit* utility function therefore can make rational decisions, but may have to compare the utilities achieved by different courses of actions. Goals, although cruder, enable the agent to pick an action right away if it satisfies the goal. In some cases, moreover, a utility function can be translated into a set of goals, such that the decisions made by a goal-based agent using those goals are identical to those made by the utility-based agent.

The overall utility-based agent structure appears in Figure 2.12. Actual utility-based agent programs appear in Chapter 5, where we examine game-playing programs that must make fine distinctions among various board positions; and in Chapter 17, where we tackle the general problem of designing decision-making agents.

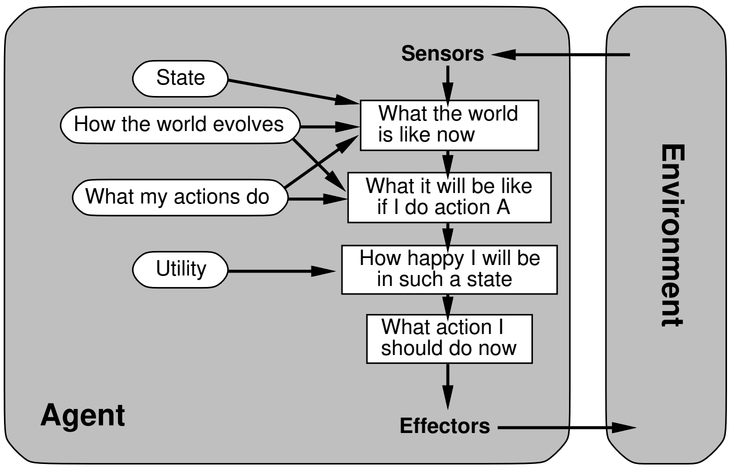


Figure 2.12 A complete utility-based agent.

2.4 ENVIRONMENTS

In this section and in the exercises at the end of the chapter, you will see how to couple an agent to an environment. Section 2.3 introduced several different kinds of agents and environments. In all cases, however, the nature of the connection between them is the same: actions are done by the agent on the environment, which in turn provides percepts to the agent. First, we will describe the different types of environments and how they affect the design of agents. Then we will describe environment programs that can be used as testbeds for agent programs.

Properties of environments

Environments come in several flavors. The principal distinctions to be made are as follows:

ACCESSIBLE

◇ **Accessible** vs. **inaccessible**.
If an agent's sensory apparatus gives it access to the complete state of the environment, then we say that the environment is accessible to that agent. An environment is effectively accessible if the sensors detect all aspects that are relevant to the choice of action. An accessible environment is convenient because the agent need not maintain any internal state to keep track of the world.

DETERMINISTIC

◇ **Deterministic** vs. **nondeterministic**.
If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic. In principle, an agent need not worry about uncertainty in an accessible, deterministic environment. If the environment is inaccessible, however, then it may *appear* to be nondeterministic. This is particularly true if the environment is complex, making it hard to keep track of all the inaccessible aspects. Thus, it is often better to think of an environment as deterministic or nondeterministic *from the point of view of the agent*.

EPISODIC

◇ **Episodic** vs. **nonepisodic**.
In an episodic environment, the agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

STATIC

◇ **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. If the environment does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic**.

SEMIDYNAMIC

DISCRETE

◇ **Discrete** vs. **continuous**.
If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Chess is discrete—there are a fixed number of possible moves on each turn. Taxi driving is continuous—the speed and location of the taxi and the other vehicles sweep through a range of continuous values.[10]

We will see that different environment types require somewhat different agent programs to deal with them effectively. It will turn out, as you might expect, that the hardest case is *inaccessible*, *nonepisodic*, *dynamic*, and *continuous*. It also turns out that most real situations are so complex that whether they are *really* deterministic is a moot point; for practical purposes, they must be treated as nondeterministic.

[10] At a fine enough level of granularity, even the taxi driving environment is discrete, because the camera image is digitized to yield discrete pixel values. But any sensible agent program would have to abstract above this level, up to a level of granularity that is continuous.

Figure 2.13 lists the properties of a number of familiar environments. Note that the answers can change depending on how you conceptualize the environments and agents. For example, poker is deterministic if the agent can keep track of the order of cards in the deck, but it is nondeterministic if it cannot. 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 next game. On the other hand, moves within a single game certainly interact, so the agent needs to look ahead several moves.

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	No	No	Yes

Figure 2.13 Examples of environments and their characteristics.

Environment programs

The generic environment program in Figure 2.14 illustrates the basic relationship between agents and environments. In this book, we will find it convenient for many of the examples and exercises to use an environment simulator that follows this program structure. The simulator takes one or more agents as input and arranges to repeatedly give each agent the right percepts and receive back an action. The simulator then updates the environment based on the actions, and possibly other dynamic processes in the environment that are not considered to be agents (rain, for example). The environment is therefore defined by the initial state and the update function. Of course, an agent that works in a simulator ought also to work in a real environment that provides the same kinds of percepts and accepts the same kinds of actions.

The RUN-ENVIRONMENT procedure correctly exercises the agents in an environment. For some kinds of agents, such as those that engage in natural language dialogue, it may be sufficient simply to observe their behavior. To get more detailed information about agent performance, we insert some performance measurement code. The function RUN-EVAL-ENVIRONMENT, shown in Figure 2.15, does this; it applies a performance measure to each agent and returns a list of the resulting scores. The *scores* variable keeps track of each agent's score.

In general, the performance measure can depend on the entire sequence of environment states generated during the operation of the program. Usually, however, the performance measure

```
procedure RUN-ENVIRONMENT(state, UPDATE-FN, agents, termination)
   inputs: state, the initial state of the environment
           UPDATE-FN, function to modify the environment
           agents, a set of agents
           termination, a predicate to test when we are done

   repeat
      for each agent in agents do
         PERCEPT[agent] ← GET-PERCEPT(agent, state)
      end
      for each agent in agents do
         ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
      end
      state ← UPDATE-FN(actions, agents, state)
   until termination(state)
```

Figure 2.14 The basic environment simulator program. It gives each agent its percept, gets an action from each agent, and then updates the environment.

```
function RUN-EVAL-ENVIRONMENT(state, UPDATE-FN, agents,
                              termination, PERFORMANCE-FN) returns scores
   local variables: scores, a vector the same size as agents, all 0

   repeat
      for each agent in agents do
         PERCEPT[agent] ← GET-PERCEPT(agent, state)
      end
      for each agent in agents do
         ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
      end
      state ← UPDATE-FN(actions, agents, state)
      scores ← PERFORMANCE-FN(scores, agents, state)
   until termination(state)
   return scores                                          /* change */
```

Figure 2.15 An environment simulator program that keeps track of the performance measure for each agent.

works by a simple accumulation using either summation, averaging, or taking a maximum. For example, if the performance measure for a vacuum-cleaning agent is the total amount of dirt cleaned in a shift, *scores* will just keep track of how much dirt has been cleaned up so far.

RUN-EVAL-ENVIRONMENT returns the performance measure for a a single environment, defined by a single initial state and a particular update function. Usually, an agent is designed to

ENVIRONMENT CLASS

work in an **environment class**, a whole set of different environments. For example, we design a chess program to play against any of a wide collection of human and machine opponents. If we designed it for a single opponent, we might be able to take advantage of specific weaknesses in that opponent, but that would not give us a good program for general play. Strictly speaking, in order to measure the performance of an agent, we need to have an environment generator that selects particular environments (with certain likelihoods) in which to run the agent. We are then interested in the agent's average performance over the environment class. This is fairly straightforward to implement for a simulated environment, and Exercises 2.5 to 2.11 take you through the entire development of an environment and the associated measurement process.

A possible confusion arises between the state variable in the environment simulator and the state variable in the agent itself (see REFLEX-AGENT-WITH-STATE). As a programmer implementing both the environment simulator and the agent, it is tempting to allow the agent to peek at the environment simulator's state variable. This temptation must be resisted at all costs! The agent's version of the state must be constructed from its percepts alone, without access to the complete state information.

2.5 SUMMARY

This chapter has been something of a whirlwind tour of AI, which we have conceived of as the science of agent design. The major points to recall are as follows:

- An **agent** is something that perceives and acts in an environment. We split an agent into an architecture and an agent program.
- An **ideal agent** is one that always takes the action that is expected to maximize its performance measure, given the percept sequence it has seen so far.
- An agent is **autonomous** to the extent that its action choices depend on its own experience, rather than on knowledge of the environment that has been built-in by the designer.
- An **agent program** maps from a percept to an action, while updating an internal state.
- There exists a variety of basic agent program designs, depending on the kind of information made explicit and used in the decision process. The designs vary in efficiency, compactness, and flexibility. The appropriate design of the agent program depends on the percepts, actions, goals, and environment.
- **Reflex agents** respond immediately to percepts, **goal-based agents** act so that they will achieve their goal(s), and **utility-based agents** try to maximize their own "happiness."
- The process of making decisions by reasoning with knowledge is central to AI and to successful agent design. This means that representing knowledge is important.
- Some environments are more demanding than others. Environments that are inaccessible, nondeterministic, nonepisodic, dynamic, and continuous are the most challenging.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The analysis of rational agency as a mapping from percept sequences to actions probably stems ultimately from the effort to identify rational behavior in the realm of economics and other forms of reasoning under uncertainty (covered in later chapters) and from the efforts of psychological behaviorists such as Skinner (1953) to reduce the psychology of organisms strictly to input/output or stimulus/response mappings. The advance from behaviorism to functionalism in psychology, which was at least partly driven by the application of the computer metaphor to agents (Putnam, 1960; Lewis, 1966), introduced the internal state of the agent into the picture. The philosopher Daniel Dennett (1969; 1978b) helped to synthesize these viewpoints into a coherent "intentional stance" toward agents. A high-level, abstract perspective on agency is also taken within the world of AI in (McCarthy and Hayes, 1969). Jon Doyle (1983) proposed that rational agent design is the core of AI, and would remain as its mission while other topics in AI would spin off to form new disciplines. Horvitz *et al.* (1988) specifically suggest the use of rationality conceived as the maximization of expected utility as a basis for AI.

The AI researcher and Nobel-prize-winning economist Herb Simon drew a clear distinction between rationality under resource limitations (procedural rationality) and rationality as making the objectively rational choice (substantive rationality) (Simon, 1958). Cherniak (1986) explores the minimal level of rationality needed to qualify an entity as an agent. Russell and Wefald (1991) deal explicitly with the possibility of using a variety of agent architectures. *Dung Beetle Ecology* (Hanski and Cambefort, 1991) provides a wealth of interesting information on the behavior of dung beetles.

EXERCISES

2.1 What is the difference between a performance measure and a utility function?

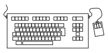
2.2 For each of the environments in Figure 2.3, determine what type of agent architecture is most appropriate (table lookup, simple reflex, goal-based or utility-based).

2.3 Choose a domain that you are familiar with, and write a PAGE description of an agent for the environment. Characterize the environment as being accessible, deterministic, episodic, static, and continuous or not. What agent architecture is best for this domain?

2.4 While driving, which is the best policy?

a. Always put your directional blinker on before turning,
b. Never use your blinker,
c. Look in your mirrors and use your blinker only if you observe a car that can observe you?

What kind of reasoning did you need to do to arrive at this policy (logical, goal-based, or utility-based)? What kind of agent design is necessary to carry out the policy (reflex, goal-based, or utility-based)?



The following exercises all concern the implementation of an environment and set of agents in the vacuum-cleaner world.

2.5 Implement a performance-measuring environment simulator for the vacuum-cleaner world. This world can be described as follows:

◊ **Percepts**: Each vacuum-cleaner agent gets a three-element percept vector on each turn. The first element, a touch sensor, should be a 1 if the machine has bumped into something and a 0 otherwise. The second comes from a photosensor under the machine, which emits a 1 if there is dirt there and a 0 otherwise. The third comes from an infrared sensor, which emits a 1 when the agent is in its home location, and a 0 otherwise.

◊ **Actions**: There are five actions available: go forward, turn right by 90°, turn left by 90°, suck up dirt, and turn off.

◊ **Goals**: The goal for each agent is to clean up and go home. To be precise, the performance measure will be 100 points for each piece of dirt vacuumed up, minus 1 point for each action taken, and minus 1000 points if it is not in the home location when it turns itself off.

◊ **Environment**: The environment consists of a grid of squares. Some squares contain obstacles (walls and furniture) and other squares are open space. Some of the open squares contain dirt. Each "go forward" action moves one square unless there is an obstacle in that square, in which case the agent stays where it is, but the touch sensor goes on. A "suck up dirt" action always cleans up the dirt. A "turn off" command ends the simulation.

We can vary the complexity of the environment along three dimensions:

◊ **Room shape**: In the simplest case, the room is an $n \times n$ square, for some fixed n. We can make it more difficult by changing to a rectangular, L-shaped, or irregularly shaped room, or a series of rooms connected by corridors.

◊ **Furniture**: Placing furniture in the room makes it more complex than an empty room. To the vacuum-cleaning agent, a piece of furniture cannot be distinguished from a wall by perception; both appear as a 1 on the touch sensor.

◊ **Dirt placement**: In the simplest case, dirt is distributed uniformly around the room. But it is more realistic for the dirt to predominate in certain locations, such as along a heavily travelled path to the next room, or in front of the couch.

2.6 Implement a table-lookup agent for the special case of the vacuum-cleaner world consisting of a 2×2 grid of open squares, in which at most two squares will contain dirt. The agent starts in the upper left corner, facing to the right. Recall that a table-lookup agent consists of a table of actions indexed by a percept sequence. In this environment, the agent can always complete its task in nine or fewer actions (four moves, three turns, and two suck-ups), so the table only needs entries for percept sequences up to length nine. At each turn, there are eight possible percept vectors, so the table will be of size $\sum_{i=1}^{9} 8^i = 153,391,688$. Fortunately, we can cut this down by realizing that the touch sensor and home sensor inputs are not needed; we can arrange so that the agent never bumps into a wall and knows when it has returned home. Then there are only two relevant percept vectors, ?0? and ?1?, and the size of the table is at most $\sum_{i=1}^{9} 2^i = 1022$. Run the environment simulator on the table-lookup agent in all possible worlds (how many are there?). Record its performance score for each world and its overall average score.

2.7 Implement an environment for a $n \times m$ rectangular room, where each square has a 5% chance of containing dirt, and n and m are chosen at random from the range 8 to 15, inclusive.

2.8 Design and implement a pure reflex agent for the environment of Exercise 2.7, ignoring the requirement of returning home, and measure its performance. Explain why it is impossible to have a reflex agent that returns home and shuts itself off. Speculate on what the best possible reflex agent could do. What prevents a reflex agent from doing very well?

2.9 Design and implement several agents with internal state. Measure their performance. How close do they come to the ideal agent for this environment?

2.10 Calculate the size of the table for a table-lookup agent in the domain of Exercise 2.7. Explain your calculation. You need not fill in the entries for the table.

2.11 Experiment with changing the shape and dirt placement of the room, and with adding furniture. Measure your agents in these new environments. Discuss how their performance might be improved to handle more complex geographies.