17 MAKING COMPLEX DECISIONS

In which we examine methods for deciding what to do today, given that we may decide again tomorrow.

SEQUENTIAL DECISION PROBLEM

In this chapter, we address the computational issues involved in making decisions in a stochastic environment. Whereas Chapter 16 was concerned with one-shot or episodic decision problems, in which the utility of each action's outcome was well known, we are concerned here with **sequential decision problems**, in which the agent's utility depends on a sequence of decisions. Sequential decision problems incorporate utilities, uncertainty, and sensing, and include search and planning problems as special cases. Section 17.1 explains how sequential decision problems are defined, and Sections 17.2 and 17.3 explain how they can be solved to produce optimal behavior that balances the risks and rewards of acting in an uncertain environment. Section 17.4 extends these ideas to the case of partially observable environments, and Section 17.4.3 develops a complete design for decision-theoretic agents in partially observable environments, combining dynamic Bayesian networks from Chapter 15 with decision networks from Chapter 16.

The second part of the chapter covers environments with multiple agents. In such environments, the notion of optimal behavior is complicated by the interactions among the agents. Section 17.5 introduces the main ideas of **game theory**, including the idea that rational agents might need to behave randomly. Section 17.6 looks at how multiagent systems can be designed so that multiple agents can achieve a common goal.

17.1 SEQUENTIAL DECISION PROBLEMS

Suppose that an agent is situated in the 4×3 environment shown in Figure 17.1(a). Beginning in the start state, it must choose an action at each time step. The interaction with the environment terminates when the agent reaches one of the goal states, marked +1 or –1. Just as for search problems, the actions available to the agent in each state are given by $\text{ACTIONS}(s)$, sometimes abbreviated to $A(s)$; in the 4×3 environment, the actions in every state are *Up*, *Down*, *Left*, and *Right*. We assume for now that the environment is **fully observable**, so that the agent always knows where it is.

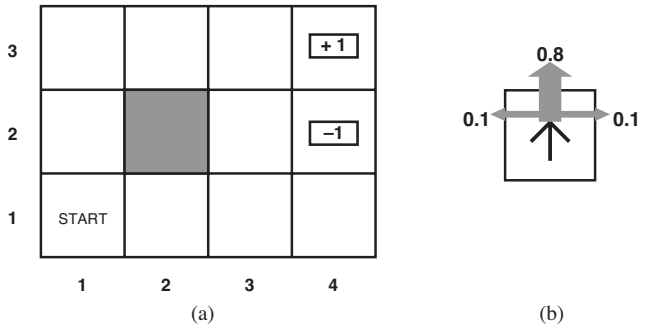

Figure 17.1 (a) A simple 4×3 environment that presents the agent with a sequential decision problem. (b) Illustration of the transition model of the environment: the "intended" outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. A collision with a wall results in no movement. The two terminal states have reward +1 and –1, respectively, and all other states have a reward of –0.04.

If the environment were deterministic, a solution would be easy: [*Up, Up, Right, Right, Right*]. Unfortunately, the environment won't always go along with this solution, because the actions are unreliable. The particular model of stochastic motion that we adopt is illustrated in Figure 17.1(b). Each action achieves the intended effect with probability 0.8, but the rest of the time, the action moves the agent at right angles to the intended direction. Furthermore, if the agent bumps into a wall, it stays in the same square. For example, from the start square (1,1), the action *Up* moves the agent to (1,2) with probability 0.8, but with probability 0.1, it moves right to (2,1), and with probability 0.1, it moves left, bumps into the wall, and stays in (1,1). In such an environment, the sequence $[Up, Up, Right, Right, Right]$ goes up around the barrier and reaches the goal state at (4,3) with probability $0.8^5 = 0.32768$. There is also a small chance of accidentally reaching the goal by going the other way around with probability $0.1^4 \times 0.8$, for a grand total of 0.32776. (See also Exercise 17.1.)

As in Chapter 3, the **transition model** (or just "model," whenever no confusion can arise) describes the outcome of each action in each state. Here, the outcome is stochastic, so we write $P(s' \mid s, a)$ to denote the probability of reaching state s' if action a is done in state s. We will assume that transitions are **Markovian** in the sense of Chapter 15, that is, the probability of reaching s' from s depends only on s and not on the history of earlier states. For now, you can think of $P(s' \mid s, a)$ as a big three-dimensional table containing probabilities. Later, in Section 17.4.3, we will see that the transition model can be represented as a **dynamic Bayesian network**, just as in Chapter 15.

To complete the definition of the task environment, we must specify the utility function for the agent. Because the decision problem is sequential, the utility function will depend on a sequence of states—an **environment history**—rather than on a single state. Later in this section, we investigate how such utility functions can be specified in general; for now,

REWARD

we simply stipulate that in each state s, the agent receives a **reward** $R(s)$, which may be positive or negative, but must be bounded. For our particular example, the reward is -0.04 in all states except the terminal states (which have rewards +1 and –1). The utility of an

environment history is just (for now) the *sum* of the rewards received. For example, if the agent reaches the +1 state after 10 steps, its total utility will be 0.6. The negative reward of –0.04 gives the agent an incentive to reach (4,3) quickly, so our environment is a stochastic generalization of the search problems of Chapter 3. Another way of saying this is that the agent does not enjoy living in this environment and so wants to leave as soon as possible.

MARKOV DECISION PROCESS

To sum up: a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a **Markov decision process**, or **MDP**, and consists of a set of states (with an initial state s_0); a set $\text{ACTIONS}(s)$ of actions in each state; a transition model $P(s' \mid s, a)$; and a reward function $R(s)$.[1]

POLICY

The next question is, what does a solution to the problem look like? We have seen that any fixed action sequence won't solve the problem, because the agent might end up in a state other than the goal. Therefore, a solution must specify what the agent should do for *any* state that the agent might reach. A solution of this kind is called a **policy**. It is traditional to denote a policy by π, and $\pi(s)$ is the action recommended by the policy π for state s. If the agent has a complete policy, then no matter what the outcome of any action, the agent will always know what to do next.

OPTIMAL POLICY

Each time a given policy is executed starting from the initial state, the stochastic nature of the environment may lead to a different environment history. The quality of a policy is therefore measured by the *expected* utility of the possible environment histories generated by that policy. An **optimal policy** is a policy that yields the highest expected utility. We use π^* to denote an optimal policy. Given π^*, the agent decides what to do by consulting its current percept, which tells it the current state s, and then executing the action $\pi^*(s)$. A policy represents the agent function explicitly and is therefore a description of a simple reflex agent, computed from the information used for a utility-based agent.

An optimal policy for the world of Figure 17.1 is shown in Figure 17.2(a). Notice that, because the cost of taking a step is fairly small compared with the penalty for ending up in (4,2) by accident, the optimal policy for the state (3,1) is conservative. The policy recommends taking the long way round, rather than taking the shortcut and thereby risking entering (4,2).

The balance of risk and reward changes depending on the value of $R(s)$ for the nonterminal states. Figure 17.2(b) shows optimal policies for four different ranges of $R(s)$. When $R(s) \leq -1.6284$, life is so painful that the agent heads straight for the nearest exit, even if the exit is worth –1. When $-0.4278 \leq R(s) \leq -0.0850$, life is quite unpleasant; the agent takes the shortest route to the +1 state and is willing to risk falling into the –1 state by accident. In particular, the agent takes the shortcut from (3,1). When life is only slightly dreary ($-0.0221 < R(s) < 0$), the optimal policy takes *no risks at all*. In (4,1) and (3,2), the agent heads directly away from the –1 state so that it cannot fall in by accident, even though this means banging its head against the wall quite a few times. Finally, if $R(s) > 0$, then life is positively enjoyable and the agent avoids *both* exits. As long as the actions in (4,1), (3,2),

[1] Some definitions of MDPs allow the reward to depend on the action and outcome too, so the reward function is $R(s, a, s')$. This simplifies the description of some environments but does not change the problem in any fundamental way, as shown in Exercise 17.4.

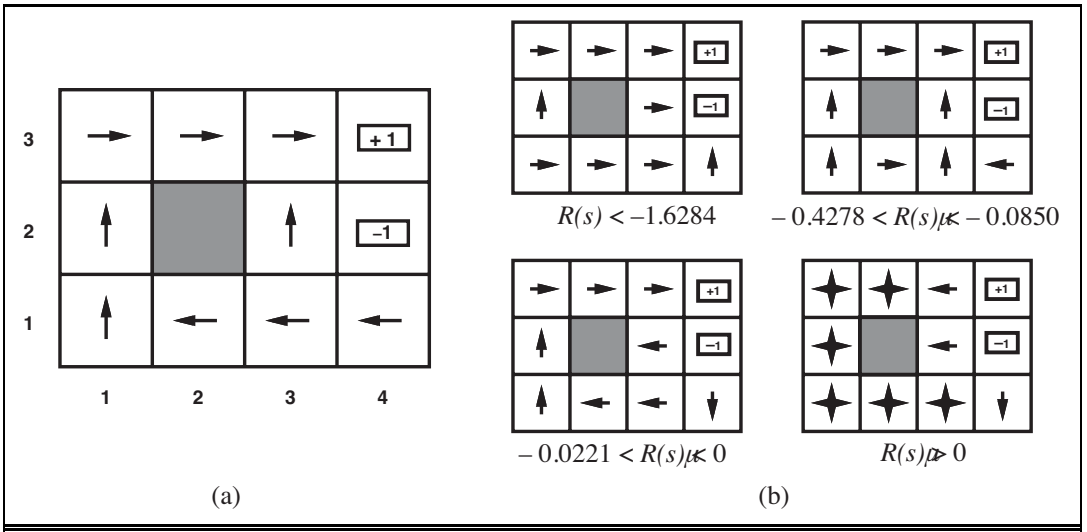

Figure 17.2 (a) An optimal policy for the stochastic environment with $R(s) = -0.04$ in the nonterminal states. (b) Optimal policies for four different ranges of $R(s)$.

and (3,3) are as shown, every policy is optimal, and the agent obtains infinite total reward because it never enters a terminal state. Surprisingly, it turns out that there are six other optimal policies for various ranges of $R(s)$; Exercise 17.5 asks you to find them.

The careful balancing of risk and reward is a characteristic of MDPs that does not arise in deterministic search problems; moreover, it is a characteristic of many real-world decision problems. For this reason, MDPs have been studied in several fields, including AI, operations research, economics, and control theory. Dozens of algorithms have been proposed for calculating optimal policies. In sections 17.2 and 17.3 we describe two of the most important algorithm families. First, however, we must complete our investigation of utilities and policies for sequential decision problems.

17.1.1 Utilities over time

In the MDP example in Figure 17.1, the performance of the agent was measured by a sum of rewards for the states visited. This choice of performance measure is not arbitrary, but it is not the only possibility for the utility function on environment histories, which we write as $U_h([s_0, s_1, \ldots, s_n])$. Our analysis draws on **multiattribute utility theory** (Section 16.4) and is somewhat technical; the impatient reader may wish to skip to the next section.

FINITE HORIZON
INFINITE HORIZON

The first question to answer is whether there is a **finite horizon** or an **infinite horizon** for decision making. A finite horizon means that there is a *fixed* time N after which nothing matters—the game is over, so to speak. Thus, $U_h([s_0, s_1, \ldots, s_{N+k}]) = U_h([s_0, s_1, \ldots, s_N])$ for all $k > 0$. For example, suppose an agent starts at (3,1) in the 4×3 world of Figure 17.1, and suppose that $N = 3$. Then, to have any chance of reaching the +1 state, the agent must head directly for it, and the optimal action is to go *Up*. On the other hand, if $N = 100$, then there is plenty of time to take the safe route by going *Left*. *So, with a finite horizon,*

NONSTATIONARY POLICY

STATIONARY POLICY

the optimal action in a given state could change over time. We say that the optimal policy for a finite horizon is **nonstationary**. With no fixed time limit, on the other hand, there is no reason to behave differently in the same state at different times. Hence, the optimal action depends only on the current state, and the optimal policy is **stationary**. Policies for the infinite-horizon case are therefore simpler than those for the finite-horizon case, and we deal mainly with the infinite-horizon case in this chapter. (We will see later that for partially observable environments, the infinite-horizon case is not so simple.) Note that "infinite horizon" does not necessarily mean that all state sequences are infinite; it just means that there is no fixed deadline. In particular, there can be finite state sequences in an infinite-horizon MDP containing a terminal state.

The next question we must decide is how to calculate the utility of state sequences. In the terminology of multiattribute utility theory, each state s_i can be viewed as an **attribute** of the state sequence $[s_0, s_1, s_2 \ldots]$. To obtain a simple expression in terms of the attributes, we will need to make some sort of preference-independence assumption. The most natural assumption is that the agent's preferences between state sequences are **stationary**. Stationarity for preferences means the following: if two state sequences $[s_0, s_1, s_2, \ldots]$ and $[s'_0, s'_1, s'_2, \ldots]$ begin with the same state (i.e., $s_0 = s'_0$), then the two sequences should be preference-ordered the same way as the sequences $[s_1, s_2, \ldots]$ and $[s'_1, s'_2, \ldots]$. In English, this means that if you prefer one future to another starting tomorrow, then you should still prefer that future if it were to start today instead. Stationarity is a fairly innocuous-looking assumption with very strong consequences: it turns out that under stationarity there are just two coherent ways to assign utilities to sequences:

STATIONARY PREFERENCE

ADDITIVE REWARD

1. **Additive rewards**: The utility of a state sequence is

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \cdots .$$

The 4×3 world in Figure 17.1 uses additive rewards. Notice that additivity was used implicitly in our use of path cost functions in heuristic search algorithms (Chapter 3).

DISCOUNTED REWARD

2. **Discounted rewards**: The utility of a state sequence is

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots ,$$

DISCOUNT FACTOR

where the **discount factor** γ is a number between 0 and 1. The discount factor describes the preference of an agent for current rewards over future rewards. When γ is close to 0, rewards in the distant future are viewed as insignificant. When γ is 1, discounted rewards are exactly equivalent to additive rewards, so additive rewards are a special case of discounted rewards. Discounting appears to be a good model of both animal and human preferences over time. A discount factor of γ is equivalent to an interest rate of $(1/\gamma) - 1$.

For reasons that will shortly become clear, we assume discounted rewards in the remainder of the chapter, although sometimes we allow $\gamma = 1$.

Lurking beneath our choice of infinite horizons is a problem: if the environment does not contain a terminal state, or if the agent never reaches one, then all environment histories will be infinitely long, and utilities with additive, undiscounted rewards will generally be

infinite. While we can agree that $+\infty$ is better than $-\infty$, comparing two state sequences with $+\infty$ utility is more difficult. There are three solutions, two of which we have seen already:

1. With discounted rewards, the utility of an infinite sequence is *finite*. In fact, if $\gamma < 1$ and rewards are bounded by $\pm R_{\max}$, we have
$$U_h([s_0, s_1, s_2, \ldots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = R_{\max}/(1-\gamma)\ , \tag{17.1}$$
using the standard formula for the sum of an infinite geometric series.
2. If the environment contains terminal states *and if the agent is guaranteed to get to one eventually*, then we will never need to compare infinite sequences. A policy that is guaranteed to reach a terminal state is called a **proper policy**. With proper policies, we can use $\gamma = 1$ (i.e., additive rewards). The first three policies shown in Figure 17.2(b) are proper, but the fourth is improper. It gains infinite total reward by staying away from the terminal states when the reward for the nonterminal states is positive. The existence of improper policies can cause the standard algorithms for solving MDPs to fail with additive rewards, and so provides a good reason for using discounted rewards.

PROPER POLICY

AVERAGE REWARD

3. Infinite sequences can be compared in terms of the **average reward** obtained per time step. Suppose that square (1,1) in the 4×3 world has a reward of 0.1 while the other nonterminal states have a reward of 0.01. Then a policy that does its best to stay in (1,1) will have higher average reward than one that stays elsewhere. Average reward is a useful criterion for some problems, but the analysis of average-reward algorithms is beyond the scope of this book.

In sum, discounted rewards present the fewest difficulties in evaluating state sequences.

17.1.2 Optimal policies and the utilities of states

Having decided that the utility of a given state sequence is the sum of discounted rewards obtained during the sequence, we can compare policies by comparing the *expected* utilities obtained when executing them. We assume the agent is in some initial state s and define S_t (a random variable) to be the state the agent reaches at time t when executing a particular policy π. (Obviously, $S_0 = s$, the state the agent is in now.) The probability distribution over state sequences $S_1, S_2, \ldots$, is determined by the initial state s, the policy π, and the transition model for the environment.

The expected utility obtained by executing π starting in s is given by
$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]\ , \tag{17.2}$$
where the expectation is with respect to the probability distribution over state sequences determined by s and π. Now, out of all the policies the agent could choose to execute starting in s, one (or more) will have higher expected utilities than all the others. We'll use π_s^* to denote one of these policies:
$$\pi_s^* = \operatorname*{argmax}_\pi U^\pi(s)\ . \tag{17.3}$$

Remember that π_s^* is a policy, so it recommends an action for every state; its connection with s in particular is that it's an optimal policy when s is the starting state. A remarkable consequence of using discounted utilities with infinite horizons is that the optimal policy is *independent* of the starting state. (Of course, the *action sequence* won't be independent; remember that a policy is a function specifying an action for each state.) This fact seems intuitively obvious: if policy π_a^* is optimal starting in a and policy π_b^* is optimal starting in b, then, when they reach a third state c, there's no good reason for them to disagree with each other, or with π_c^*, about what to do next.[2] So we can simply write π^* for an optimal policy.

Given this definition, the true utility of a state is just $U^{\pi^*}(s)$—that is, the expected sum of discounted rewards if the agent executes an optimal policy. We write this as $U(s)$, matching the notation used in Chapter 16 for the utility of an outcome. Notice that $U(s)$ and $R(s)$ are quite different quantities; $R(s)$ is the "short term" reward for being in s, whereas $U(s)$ is the "long term" total reward from s onward. Figure 17.3 shows the utilities for the 4×3 world. Notice that the utilities are higher for states closer to the +1 exit, because fewer steps are required to reach the exit.

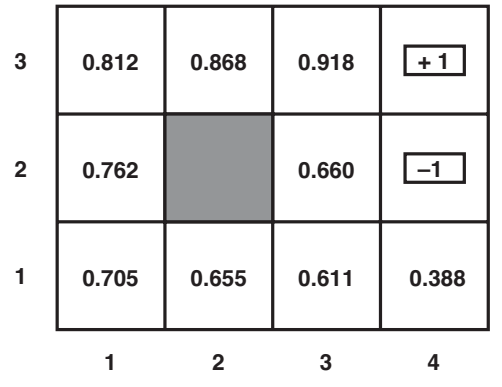

Figure 17.3 The utilities of the states in the 4×3 world, calculated with $\gamma = 1$ and $R(s) = -0.04$ for nonterminal states.

The utility function $U(s)$ allows the agent to select actions by using the principle of maximum expected utility from Chapter 16—that is, choose the action that maximizes the expected utility of the subsequent state:

$$\pi^*(s) = \operatorname*{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s') \ . \tag{17.4}$$

The next two sections describe algorithms for finding optimal policies.

[2] Although this seems obvious, it does not hold for finite-horizon policies or for other ways of combining rewards over time. The proof follows directly from the uniqueness of the utility function on states, as shown in Section 17.2.

17.2 VALUE ITERATION

VALUE ITERATION

In this section, we present an algorithm, called **value iteration**, for calculating an optimal policy. The basic idea is to calculate the utility of each state and then use the state utilities to select an optimal action in each state.

17.2.1 The Bellman equation for utilities

Section 17.1.2 defined the utility of being in a state as the expected sum of discounted rewards from that point onwards. From this, it follows that there is a direct relationship between the utility of a state and the utility of its neighbors: *the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.* That is, the utility of a state is given by

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s') \ . \tag{17.5}$$

BELLMAN EQUATION

This is called the **Bellman equation**, after Richard Bellman (1957). The utilities of the states—defined by Equation (17.2) as the expected utility of subsequent state sequences—are solutions of the set of Bellman equations. In fact, they are the *unique* solutions, as we show in Section 17.2.3.

Let us look at one of the Bellman equations for the 4×3 world. The equation for the state (1,1) is

$$\begin{aligned} U(1,1) = -0.04 + \gamma \max[\ & 0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), && (\mathit{Up}) \\ & 0.9U(1,1) + 0.1U(1,2), && (\mathit{Left}) \\ & 0.9U(1,1) + 0.1U(2,1), && (\mathit{Down}) \\ & 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1) \ ]. && (\mathit{Right}) \end{aligned}$$

When we plug in the numbers from Figure 17.3, we find that *Up* is the best action.

17.2.2 The value iteration algorithm

The Bellman equation is the basis of the value iteration algorithm for solving MDPs. If there are n possible states, then there are n Bellman equations, one for each state. The n equations contain n unknowns—the utilities of the states. So we would like to solve these simultaneous equations to find the utilities. There is one problem: the equations are *nonlinear*, because the "max" operator is not a linear operator. Whereas systems of linear equations can be solved quickly using linear algebra techniques, systems of nonlinear equations are more problematic. One thing to try is an *iterative* approach. We start with arbitrary initial values for the utilities, calculate the right-hand side of the equation, and plug it into the left-hand side—thereby updating the utility of each state from the utilities of its neighbors. We repeat this until we reach an equilibrium. Let $U_i(s)$ be the utility value for state s at the ith iteration. The iteration step, called a **Bellman update**, looks like this:

BELLMAN UPDATE

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s') \ , \tag{17.6}$$

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum change in the utility of any state in an iteration

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

Figure 17.4 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (17.8).

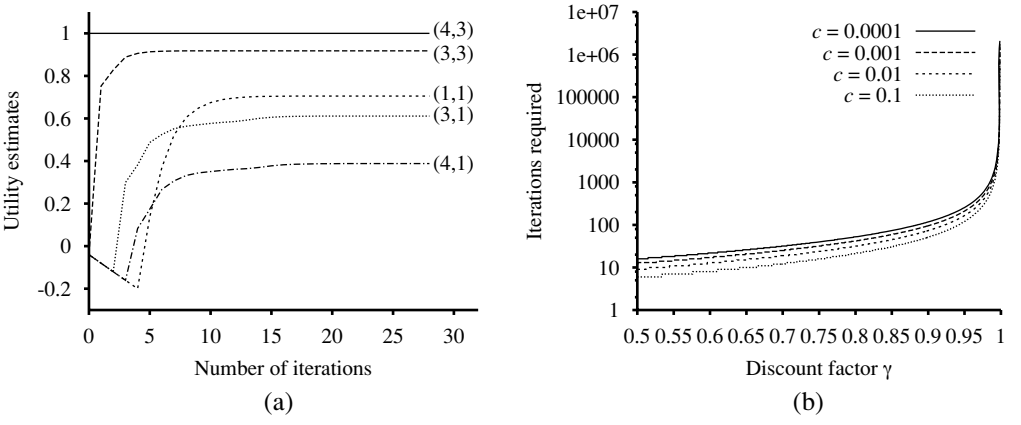

Figure 17.5 (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations k required to guarantee an error of at most $\epsilon = c \cdot R_{\max}$, for different values of c, as a function of the discount factor γ.

where the update is assumed to be applied simultaneously to all the states at each iteration. If we apply the Bellman update infinitely often, we are guaranteed to reach an equilibrium (see Section 17.2.3), in which case the final utility values must be solutions to the Bellman equations. In fact, they are also the *unique* solutions, and the corresponding policy (obtained using Equation (17.4)) is optimal. The algorithm, called VALUE-ITERATION, is shown in Figure 17.4.

We can apply value iteration to the 4×3 world in Figure 17.1(a). Starting with initial values of zero, the utilities evolve as shown in Figure 17.5(a). Notice how the states at differ-

ent distances from (4,3) accumulate negative reward until a path is found to (4,3), whereupon the utilities start to increase. We can think of the value iteration algorithm as *propagating information* through the state space by means of local updates.

17.2.3 Convergence of value iteration

We said that value iteration eventually converges to a unique set of solutions of the Bellman equations. In this section, we explain why this happens. We introduce some useful mathematical ideas along the way, and we obtain some methods for assessing the error in the utility function returned when the algorithm is terminated early; this is useful because it means that we don't have to run forever. This section is quite technical.

CONTRACTION

The basic concept used in showing that value iteration converges is the notion of a **contraction**. Roughly speaking, a contraction is a function of one argument that, when applied to two different inputs in turn, produces two output values that are "closer together," by at least some constant factor, than the original inputs. For example, the function "divide by two" is a contraction, because, after we divide any two numbers by two, their difference is halved. Notice that the "divide by two" function has a fixed point, namely zero, that is unchanged by the application of the function. From this example, we can discern two important properties of contractions:

- A contraction has only one fixed point; if there were two fixed points they would not get closer together when the function was applied, so it would not be a contraction.
- When the function is applied to any argument, the value must get closer to the fixed point (because the fixed point does not move), so repeated application of a contraction always reaches the fixed point in the limit.

Now, suppose we view the Bellman update (Equation (17.6)) as an operator B that is applied simultaneously to update the utility of every state. Let U_i denote the vector of utilities for all the states at the ith iteration. Then the Bellman update equation can be written as

$$U_{i+1} \leftarrow B\,U_i\ .$$

MAX NORM

Next, we need a way to measure distances between utility vectors. We will use the **max norm**, which measures the "length" of a vector by the absolute value of its biggest component:

$$||U|| = \max_s |U(s)|\ .$$

With this definition, the "distance" between two vectors, $||U - U'||$, is the maximum difference between any two corresponding elements. The main result of this section is the following: *Let U_i and U'_i be any two utility vectors. Then we have*

$$||B\,U_i - B\,U'_i|| \le \gamma\,||U_i - U'_i||\ . \tag{17.7}$$

That is, the Bellman update is a contraction by a factor of γ on the space of utility vectors. (Exercise 17.6 provides some guidance on proving this claim.) Hence, from the properties of contractions in general, it follows that value iteration always converges to a unique solution of the Bellman equations whenever $\gamma < 1$.

We can also use the contraction property to analyze the *rate* of convergence to a solution. In particular, we can replace U_i' in Equation (17.7) with the *true* utilities U, for which $B\,U = U$. Then we obtain the inequality

$$||B\,U_i - U|| \leq \gamma\,||U_i - U|| \;.$$

So, if we view $||U_i - U||$ as the *error* in the estimate U_i, we see that the error is reduced by a factor of at least γ on each iteration. This means that value iteration converges exponentially fast. We can calculate the number of iterations required to reach a specified error bound ϵ as follows: First, recall from Equation (17.1) that the utilities of all states are bounded by $\pm R_{\max}/(1-\gamma)$. This means that the maximum initial error $||U_0 - U|| \leq 2R_{\max}/(1-\gamma)$. Suppose we run for N iterations to reach an error of at most ϵ. Then, because the error is reduced by at least γ each time, we require $\gamma^N \cdot 2R_{\max}/(1-\gamma) \leq \epsilon$. Taking logs, we find

$$N = \lceil \log(2R_{\max}/\epsilon(1-\gamma))/\log(1/\gamma) \rceil$$

iterations suffice. Figure 17.5(b) shows how N varies with γ, for different values of the ratio $\epsilon/R_{\max}$. The good news is that, because of the exponentially fast convergence, N does not depend much on the ratio $\epsilon/R_{\max}$. The bad news is that N grows rapidly as γ becomes close to 1. We can get fast convergence if we make γ small, but this effectively gives the agent a short horizon and could miss the long-term effects of the agent's actions.

The error bound in the preceding paragraph gives some idea of the factors influencing the run time of the algorithm, but is sometimes overly conservative as a method of deciding when to stop the iteration. For the latter purpose, we can use a bound relating the error to the size of the Bellman update on any given iteration. From the contraction property (Equation (17.7)), it can be shown that if the update is small (i.e., no state's utility changes by much), then the error, compared with the true utility function, also is small. More precisely,

$$\text{if} \quad ||U_{i+1} - U_i|| < \epsilon(1-\gamma)/\gamma \quad \text{then} \quad ||U_{i+1} - U|| < \epsilon \;. \tag{17.8}$$

This is the termination condition used in the VALUE-ITERATION algorithm of Figure 17.4.

So far, we have analyzed the error in the utility function returned by the value iteration algorithm. *What the agent really cares about, however, is how well it will do if it makes its decisions on the basis of this utility function.* Suppose that after i iterations of value iteration, the agent has an estimate U_i of the true utility U and obtains the MEU policy π_i based on one-step look-ahead using U_i (as in Equation (17.4)). Will the resulting behavior be nearly as good as the optimal behavior? This is a crucial question for any real agent, and it turns out that the answer is yes. $U^{\pi_i}(s)$ is the utility obtained if π_i is executed starting in s, and the

POLICY LOSS

policy loss $||U^{\pi_i} - U||$ is the most the agent can lose by executing π_i instead of the optimal policy π^*. The policy loss of π_i is connected to the error in U_i by the following inequality:

$$\text{if} \quad ||U_i - U|| < \epsilon \quad \text{then} \quad ||U^{\pi_i} - U|| < 2\epsilon\gamma/(1-\gamma) \;. \tag{17.9}$$

In practice, it often occurs that π_i becomes optimal long before U_i has converged. Figure 17.6 shows how the maximum error in U_i and the policy loss approach zero as the value iteration process proceeds for the 4×3 environment with $\gamma = 0.9$. The policy π_i is optimal when $i = 4$, even though the maximum error in U_i is still 0.46.

Now we have everything we need to use value iteration in practice. We know that it converges to the correct utilities, we can bound the error in the utility estimates if we

stop after a finite number of iterations, and we can bound the policy loss that results from executing the corresponding MEU policy. As a final note, all of the results in this section depend on discounting with $\gamma < 1$. If $\gamma = 1$ and the environment contains terminal states, then a similar set of convergence results and error bounds can be derived whenever certain technical conditions are satisfied.

17.3 POLICY ITERATION

In the previous section, we observed that it is possible to get an optimal policy even when the utility function estimate is inaccurate. If one action is clearly better than all others, then the exact magnitude of the utilities on the states involved need not be precise. This insight suggests an alternative way to find optimal policies. The **policy iteration** algorithm alternates the following two steps, beginning from some initial policy π_0:

POLICY ITERATION

POLICY EVALUATION

- **Policy evaluation**: given a policy π_i, calculate $U_i = U^{\pi_i}$, the utility of each state if π_i were to be executed.

POLICY IMPROVEMENT

- **Policy improvement**: Calculate a new MEU policy π_{i+1}, using one-step look-ahead based on U_i (as in Equation (17.4)).

The algorithm terminates when the policy improvement step yields no change in the utilities. At this point, we know that the utility function U_i is a fixed point of the Bellman update, so it is a solution to the Bellman equations, and π_i must be an optimal policy. Because there are only finitely many policies for a finite state space, and each iteration can be shown to yield a better policy, policy iteration must terminate. The algorithm is shown in Figure 17.7.

The policy improvement step is obviously straightforward, but how do we implement the POLICY-EVALUATION routine? It turns out that doing so is much simpler than solving the standard Bellman equations (which is what value iteration does), because the action in each state is fixed by the policy. At the ith iteration, the policy π_i specifies the action $\pi_i(s)$ in

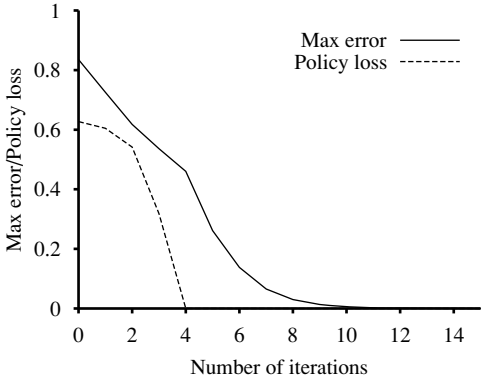

Figure 17.6 The maximum error $||U_i - U||$ of the utility estimates and the policy loss $||U^{\pi_i} - U||$, as a function of the number of iterations of value iteration.

state s. This means that we have a simplified version of the Bellman equation (17.5) relating the utility of s (under π_i) to the utilities of its neighbors:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s') \; . \tag{17.10}$$

For example, suppose π_i is the policy shown in Figure 17.2(a). Then we have $\pi_i(1,1) = Up$, $\pi_i(1,2) = Up$, and so on, and the simplified Bellman equations are

$$\begin{aligned} U_i(1,1) &= -0.04 + 0.8U_i(1,2) + 0.1U_i(1,1) + 0.1U_i(2,1) \; , \\ U_i(1,2) &= -0.04 + 0.8U_i(1,3) + 0.2U_i(1,2) \; , \\ &\vdots \end{aligned}$$

The important point is that these equations are *linear*, because the "$\max$" operator has been removed. For n states, we have n linear equations with n unknowns, which can be solved exactly in time $O(n^3)$ by standard linear algebra methods.

For small state spaces, policy evaluation using exact solution methods is often the most efficient approach. For large state spaces, $O(n^3)$ time might be prohibitive. Fortunately, it is not necessary to do *exact* policy evaluation. Instead, we can perform some number of simplified value iteration steps (simplified because the policy is fixed) to give a reasonably good approximation of the utilities. The simplified Bellman update for this process is

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s') \; ,$$

MODIFIED POLICY ITERATION

and this is repeated k times to produce the next utility estimate. The resulting algorithm is called **modified policy iteration**. It is often much more efficient than standard policy iteration or value iteration.

function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s **in** S **do**
      if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \; U[s'] \; > \; \sum_{s'} P(s' \mid s, \pi[s]) \; U[s']$ **then do**
        $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \; U[s']$
        $unchanged? \leftarrow$ false
  until $unchanged?$
  return π

Figure 17.7 The policy iteration algorithm for calculating an optimal policy.

The algorithms we have described so far require updating the utility or policy for all states at once. It turns out that this is not strictly necessary. In fact, on each iteration, we can pick *any subset* of states and apply *either* kind of updating (policy improvement or simplified value iteration) to that subset. This very general algorithm is called **asynchronous policy iteration**. Given certain conditions on the initial policy and initial utility function, asynchronous policy iteration is guaranteed to converge to an optimal policy. The freedom to choose any states to work on means that we can design much more efficient heuristic algorithms—for example, algorithms that concentrate on updating the values of states that are likely to be reached by a good policy. This makes a lot of sense in real life: if one has no intention of throwing oneself off a cliff, one should not spend time worrying about the exact value of the resulting states.

ASYNCHRONOUS POLICY ITERATION

17.4 PARTIALLY OBSERVABLE MDPS

The description of Markov decision processes in Section 17.1 assumed that the environment was **fully observable**. With this assumption, the agent always knows which state it is in. This, combined with the Markov assumption for the transition model, means that the optimal policy depends only on the current state. When the environment is only **partially observable**, the situation is, one might say, much less clear. The agent does not necessarily know which state it is in, so it cannot execute the action $\pi(s)$ recommended for that state. Furthermore, the utility of a state s and the optimal action in s depend not just on s, but also on *how much the agent knows* when it is in s. For these reasons, **partially observable MDPs** (or POMDPs—pronounced "pom-dee-pees") are usually viewed as much more difficult than ordinary MDPs. We cannot avoid POMDPs, however, because the real world is one.

PARTIALLY OBSERVABLE MDP

17.4.1 Definition of POMDPs

To get a handle on POMDPs, we must first define them properly. A POMDP has the same elements as an MDP—the transition model $P(s' \mid s, a)$, actions $A(s)$, and reward function $R(s)$—but, like the partially observable search problems of Section 4.4, it also has a **sensor model** $P(e \mid s)$. Here, as in Chapter 15, the sensor model specifies the probability of perceiving evidence e in state s.[3] For example, we can convert the 4×3 world of Figure 17.1 into a POMDP by adding a noisy or partial sensor instead of assuming that the agent knows its location exactly. Such a sensor might measure the *number of adjacent walls*, which happens to be 2 in all the nonterminal squares except for those in the third column, where the value is 1; a noisy version might give the wrong value with probability 0.1.

In Chapters 4 and 11, we studied nondeterministic and partially observable planning problems and identified the **belief state**—the set of actual states the agent might be in—as a key concept for describing and calculating solutions. In POMDPs, the belief state b becomes a *probability distribution* over all possible states, just as in Chapter 15. For example, the initial

[3] As with the reward function for MDPs, the sensor model can also depend on the action and outcome state, but again this change is not fundamental.