

CHAPTER 16

MAKING COMPLEX DECISIONS

In which we examine methods for deciding what to do today, given that we may face another decision tomorrow.

Sequential decision problem

In this chapter, we address the computational issues involved in making decisions in a stochastic environment. Whereas Chapter 15 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 16.1 explains how sequential decision problems are defined, and Section 16.2 describes methods for solving them to produce behaviors that are appropriate for a stochastic environment. Section 16.3 covers **multi-armed bandit** problems, a specific and fascinating class of sequential decision problems that arise in many contexts. Section 16.4 explores decision problems in partially observable environments and Section 16.5 describes how to solve them.

16.1 Sequential Decision Problems

Suppose that an agent is situated in the 4×3 environment shown in Figure 16.1(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.

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 16.1(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 16.MDPX.)

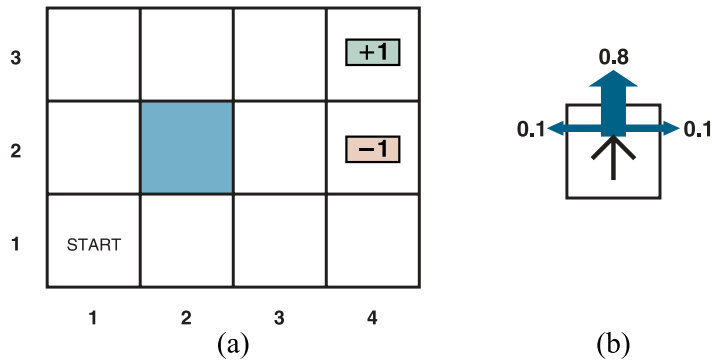


Figure 16.1 (a) A simple, stochastic 4×3 environment that presents the agent with a sequential decision problem. (b) Illustration of the transition model of the environment: the “intended” outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. A collision with a wall results in no movement. Transitions into the two terminal states have reward +1 and −1, respectively, and all other transitions have a reward of −0.04.

As in Chapter 3, the **transition model** (or just “model,” when the meaning is clear) describes the outcome of each action in each state. Here, the outcome is stochastic, so we write $P(s' | s, a)$ for the probability of reaching state s' if action a is done in state s . (Some authors write $T(s, a, s')$ for the transition model.) We will assume that transitions are **Markovian**: the probability of reaching s' from s depends only on s and not on the history of earlier states.

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 and actions—an **environment history**—rather than on a single state. Later in this section, we investigate the nature of utility functions on histories; for now, we simply stipulate that for every transition from s to s' via action a , the agent receives a **reward** $R(s, a, s')$. The rewards may be positive or negative, but they are bounded by $\pm R_{\max}$.¹

Reward

For our particular example, the reward is −0.04 for all transitions except those entering 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 $(9 \times -0.04) + 1 = 0.64$. 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 it wants to leave as soon as possible.

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 $P(s' | s, a)$; and a reward function $R(s, a, s')$. Methods for solving MDPs usually involve **dynamic programming**: simplifying a problem by recursively breaking it into smaller pieces and remembering the optimal solutions to the pieces.

Markov decision process

Dynamic programming

¹ It is also possible to use costs $c(s, a, s')$, as we did in the definition of search problems in Chapter 3. The use of rewards is, however, standard in the literature on sequential decisions under uncertainty.

Policy

The next question is, what does a solution to the problem look like? No fixed action sequence can 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 . No matter what the outcome of the action, the resulting state will be in the policy, and the agent will 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.

The optimal policies for the world of Figure 16.1 are shown in Figure 16.2(a). There are two policies because the agent is exactly indifferent between going left and going up from (3,1): going left is safer but longer, while going up is quicker but risks falling into (4,2) by accident. In general there will often be multiple optimal policies.

The balance of risk and reward changes depending on the value of $r = R(s, a, s')$ for transitions between nonterminal states. The policies shown in Figure 16.2(a) are optimal for $-0.0850 < r < -0.0273$. Figure 16.2(b) shows optimal policies for four other ranges of r . When $r < -1.6497$, life is so painful that the agent heads straight for the nearest exit, even if the exit is worth -1 . When $-0.7311 < r < -0.4526$, life is quite unpleasant; the agent takes the shortest route to the $+1$ state from (2,1), (3,1), and (3,2), but from (4,1) the cost of reaching $+1$ is so high that the agent prefers to dive straight into -1 . When life is only slightly dreary ($-0.0274 < r < 0$), 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 $r > 0$, then life is positively enjoyable and the agent avoids *both* exits. As long as the actions in (4,1), (3,2), and (3,3) are as shown, every policy is optimal, and the agent obtains infinite total reward because it never enters a terminal state. It turns out that there are nine optimal policies in all for various ranges of r ; Exercise 16.THRC asks you to find them.

The introduction of uncertainty brings MDPs closer to the real world than deterministic search problems. For this reason, MDPs have been studied in several fields, including AI, operations research, economics, and control theory. Dozens of solution algorithms have been proposed, several of which we discuss in Section 16.2. First, however, we spell out in more detail the definitions of utilities, optimal policies, and models for MDPs.

16.1.1 Utilities over time

In the MDP example in Figure 16.1, the performance of the agent was measured by a sum of rewards for the transitions experienced. This choice of performance measure is not arbitrary, but it is not the only possibility for the utility function² on environment histories, which we write as $U_h([s_0, a_0, s_1, a_1 \dots, s_n])$.

² In this chapter we use U for the utility function (to be consistent with the rest of the book), but many works about MDPs use V (for *value*) instead.

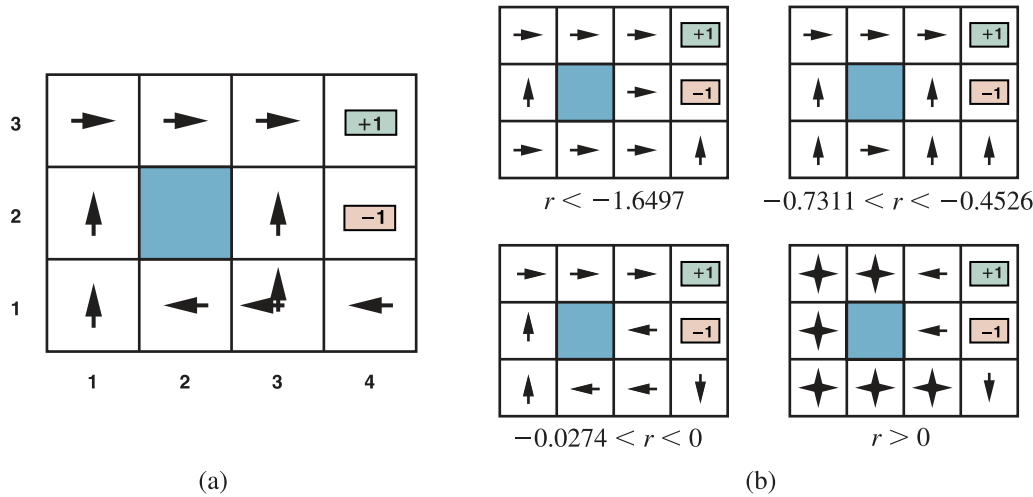


Figure 16.2 (a) The optimal policies for the stochastic environment with $r = -0.04$ for transitions between nonterminal states. There are two policies because in state (3,1) both *Left* and *Up* are optimal. (b) Optimal policies for four different ranges of r .

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,

$$U_h([s_0, a_0, s_1, a_1, \dots, s_{N+k}]) = U_h([s_0, a_0, s_1, a_1, \dots, s_N])$$

for all $k > 0$. For example, suppose an agent starts at (3,1) in the 4×3 world of Figure 16.1, 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*, an optimal action in a given state may depend on how much time is left. A policy that depends on the time is called **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, an 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. There can be finite state sequences in an infinite-horizon MDP that contains a terminal state.

The next question we must decide is how to calculate the utility of state sequences. Throughout this chapter, we will **additive discounted rewards**: the utility of a history is

$$U_h([s_0, a_0, s_1, a_1, s_2, \dots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \dots,$$

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 close to 1, an agent is more willing to wait for long-term rewards. When γ is exactly 1, discounted rewards reduce to the special

Finite horizon

Infinite horizon

Nonstationary policy

Stationary policy

Additive discounted reward

Discount factor

Additive reward

case of purely **additive rewards**. Notice that additivity was used implicitly in our use of path cost functions in heuristic search algorithms (Chapter 3).

There are several reasons why additive discounted rewards make sense. One is empirical: both humans and animals appear to value near-term rewards more highly than rewards in the distant future. Another is economic: if the rewards are monetary, then it really is better to get them sooner rather than later because early rewards can be invested and produce returns while you're waiting for the later rewards. In this context, a discount factor of γ is equivalent to an interest rate of $(1/\gamma) - 1$. For example, a discount factor of $\gamma = 0.9$ is equivalent to an interest rate of 11.1%.

A third reason is uncertainty about the true rewards: they may never arrive for all sorts of reasons that are not taken into account in the transition model. Under certain assumptions, a discount factor of γ is equivalent to adding a probability $1 - \gamma$ of accidental termination at every time step, independent of the action taken.

A fourth justification arises from a natural property of preferences over histories. In the terminology of multiattribute utility theory (see Section 15.4), each transition $s_t \xrightarrow{a_t} s_{t+1}$ can be viewed as an **attribute** of the history $[s_0, a_0, s_1, a_1, s_2, \dots]$. In principle, the utility function could depend in arbitrarily complex ways on these attributes. There is, however, a highly plausible preference-independence assumption that can be made, namely that the agent's preferences between state sequences are **stationary**.

Stationary preference

Assume two histories $[s_0, a_0, s_1, a_1, s_2, \dots]$ and $[s'_0, a'_0, s'_1, a'_1, s'_2, \dots]$ begin with the same transition (i.e., $s_0 = s'_0$, $a_0 = a'_0$, and $s_1 = s'_1$). Then stationarity for preferences means that the two histories should be preference-ordered the same way as the histories $[s_1, a_1, s_2, \dots]$ and $[s'_1, a'_1, s'_2, \dots]$. 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, but additive discounting is the only form of utility on histories that satisfies it.

A final justification for discounted rewards is that it conveniently makes some nasty infinities go away. With infinite horizons there is a potential difficulty: 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

$$U_h([s_0, a_0, s_1, \dots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = \frac{R_{\max}}{1 - \gamma}, \quad (16.1)$$

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 undiscounted rewards). The first three policies shown in Figure 16.2(b) are proper, but the fourth is improper. It gains infinite total reward by staying away from the terminal states when the reward for transitions between non-terminal states is positive. The existence of improper policies can cause the standard

Proper policy

algorithms for solving MDPs to fail with additive rewards, and so provides a good reason for using discounted rewards.

3. Infinite sequences can be compared in terms of the **average reward** obtained per time step. Suppose that transitions to square (1,1) in the 4×3 world have a reward of 0.1 while transitions to 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 complex. Average reward

Additive discounted rewards present the fewest difficulties in evaluating histories, so we shall use them henceforth.

16.1.2 Optimal policies and the utilities of states

Having decided that the utility of a given history is the sum of discounted rewards, 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, \dots , 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

$$U^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \right], \quad (16.2)$$

where the expectation E 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:

$$\pi_s^* = \operatorname{argmax}_{\pi} U^\pi(s). \quad (16.3)$$

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.³ 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 15 for the utility of an outcome. Figure 16.3 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.

³ Although this seems obvious, it does not hold for finite-horizon policies or for other ways of combining rewards over time, such as taking the max. The proof follows directly from the uniqueness of the utility function on states, as shown in Section 16.2.1.

3	0.8516	0.9078	0.9578	+1
2	0.8016		0.7003	-1
1	0.7453	0.6953	0.6514	0.4279
	1	2	3	4

Figure 16.3 The utilities of the states in the 4×3 world with $\gamma = 1$ and $r = -0.04$ for transitions to nonterminal states.

The utility function $U(s)$ allows the agent to select actions by using the principle of maximum expected utility from Chapter 15—that is, choose the action that maximizes the reward for the next step plus the expected discounted utility of the subsequent state:

$$\pi^*(s) = \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U(s')]. \quad (16.4)$$

We have defined the utility of a state, $U(s)$, 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 expected reward for the next transition plus the discounted utility of the next state, assuming that the agent chooses the optimal action.* That is, the utility of a state is given by

$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U(s')]. \quad (16.5)$$

Bellman equation

This is called the **Bellman equation**, after Richard Bellman (1957). The utilities of the states—defined by Equation (16.2) 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 16.2.1.

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

$$\max \{ \begin{aligned} & [0.8(-0.04 + \gamma U(1,2)) + 0.1(-0.04 + \gamma U(2,1)) + 0.1(-0.04 + \gamma U(1,1))], \\ & [0.9(-0.04 + \gamma U(1,1)) + 0.1(-0.04 + \gamma U(1,2))], \\ & [0.9(-0.04 + \gamma U(1,1)) + 0.1(-0.04 + \gamma U(2,1))], \\ & [0.8(-0.04 + \gamma U(2,1)) + 0.1(-0.04 + \gamma U(1,2)) + 0.1(-0.04 + \gamma U(1,1))] \end{aligned} \}$$

where the four expressions correspond to *Up*, *Left*, *Down* and *Right* moves. When we plug in the numbers from Figure 16.3, with $\gamma = 1$, we find that *Up* is the best action.

Q-function

Another important quantity is the **action-utility function**, or **Q-function**: $Q(s, a)$ is the expected utility of taking a given action in a given state. The Q-function is related to utilities in the obvious way:

$$U(s) = \max_a Q(s, a). \quad (16.6)$$

Furthermore, the optimal policy can be extracted from the Q -function as follows:

$$\pi^*(s) = \operatorname{argmax}_a Q(s, a). \quad (16.7)$$

We can also develop a Bellman equation for Q -functions, noting that the expected total reward for taking an action is its immediate reward plus the discounted utility of the outcome state, which in turn can be expressed in terms of the Q -function:

$$\begin{aligned} Q(s, a) &= \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U(s')] \\ &= \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma \max_{a'} Q(s', a')] \end{aligned} \quad (16.8)$$

Solving the Bellman equations for U (or for Q) gives us what we need to find an optimal policy. The Q -function shows up again and again in algorithms for solving MDPs, so we shall use the following definition:

function $Q\text{-VALUE}(mdp, s, a, U)$ **returns** a utility value
return $\sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U[s']]$

16.1.3 Reward scales

Chapter 15 noted that the scale of utilities is arbitrary: an affine transformation leaves the optimal decision unchanged. We can replace $U(s)$ by $U'(s) = mU(s) + b$ where m and b are any constants such that $m > 0$. It is easy to see, from the definition of utilities as discounted sums of rewards, that a similar transformation of rewards will leave the optimal policy unchanged in an MDP:

$$R'(s, a, s') = mR(s, a, s') + b.$$

It turns out, however, that the additive reward decomposition of utilities leads to significantly more freedom in defining rewards. Let $\Phi(s)$ be *any* function of the state s . Then, according to the **shaping theorem**, the following transformation leaves the optimal policy unchanged: Shaping theorem

$$R'(s, a, s') = R(s, a, s') + \gamma\Phi(s') - \Phi(s). \quad (16.9)$$

To show that this is true, we need to prove that two MDPs, M and M' , have identical optimal policies as long as they differ only in their reward functions as specified in Equation (16.9). We start from the Bellman equation for Q , the Q -function for MDP M :

$$Q(s, a) = \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma \max_{a'} Q(s', a')].$$

Now let $Q'(s, a) = Q(s, a) - \Phi(s)$ and plug it into this equation; we get

$$Q'(s, a) + \Phi(s) = \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma \max_{a'} (Q'(s', a') + \Phi(s'))].$$

which then simplifies to

$$\begin{aligned} Q'(s, a) &= \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma\Phi(s') - \Phi(s) + \gamma \max_{a'} Q'(s', a')] \\ &= \sum_{s'} P(s' | s, a) [R'(s, a, s') + \gamma \max_{a'} Q'(s', a')]. \end{aligned}$$

In other words, $Q'(s, a)$ satisfies the Bellman equation for MDP M' . Now we can extract the optimal policy for M' using Equation (16.7):

$$\pi_{M'}^*(s) = \operatorname{argmax}_a Q'(s, a) = \operatorname{argmax}_a Q(s, a) - \Phi(s) = \operatorname{argmax}_a Q(s, a) = \pi_M^*(s).$$

The function $\Phi(s)$ is often called a **potential**, by analogy to the electrical potential (voltage) that gives rise to electric fields. The term $\gamma\Phi(s') - \Phi(s)$ functions as a gradient of the potential. Thus, if $\Phi(s)$ has higher value in states that have higher utility, the addition of $\gamma\Phi(s') - \Phi(s)$ to the reward has the effect of leading the agent “uphill” in utility.

At first sight, it may seem rather counterintuitive that we can modify the reward in this way without changing the optimal policy. It helps if we remember that *all policies are optimal* with a reward function that is zero everywhere. This means, according to the shaping theorem, that all policies are optimal for any potential-based reward of the form $R(s, a, s') = \gamma\Phi(s') - \Phi(s)$. Intuitively, this is because with such a reward it doesn’t matter which way the agent goes from A to B . (This is easiest to see when $\gamma = 1$: along any path the sum of rewards collapses to $\Phi(B) - \Phi(A)$, so all paths are equally good.) So adding a potential-based reward to any other reward shouldn’t change the optimal policy.

The flexibility afforded by the shaping theorem means that we can actually help out the agent by making the immediate reward more directly reflect what the agent should do. In fact, if we set $\Phi(s) = U(s)$, then the greedy policy π_G with respect to the modified reward R' is also an optimal policy:

$$\begin{aligned} \pi_G(s) &= \operatorname{argmax}_a \sum_{s'} P(s' | s, a) R'(s, a, s') \\ &= \operatorname{argmax}_a \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma\Phi(s') - \Phi(s)] \\ &= \operatorname{argmax}_a \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U(s') - U(s)] \\ &= \operatorname{argmax}_a \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U(s')] \\ &= \pi^*(s) \quad (\text{by Equation (16.4)}). \end{aligned}$$

Of course, in order to set $\Phi(s) = U(s)$, we would need to know $U(s)$; so there is no free lunch, but there is still considerable value in defining a reward function that is helpful to the extent possible. This is precisely what animal trainers do when they provide a small treat to the animal for each step in the target sequence.

16.1.4 Representing MDPs

The simplest way to represent $P(s' | s, a)$ and $R(s, a, s')$ is with big, three-dimensional tables of size $|S|^2|A|$. This is fine for small problems such as the 4×3 world, for which the tables have $11^2 \times 4 = 484$ entries each. In some cases, the tables are **sparse**—most entries are zero because each state s can transition to only a bounded number of states s' —which means the tables are of size $O(|S||A|)$. For larger problems, even sparse tables are far too big.

Just as in Chapter 15, where Bayesian networks were extended with action and utility nodes to create decision networks, we can represent MDPs by extending dynamic Bayesian networks (DBNs, see Chapter 14) with decision, reward, and utility nodes to create **dynamic decision networks**, or DDNs. DDNs are **factored representations** in the terminology of

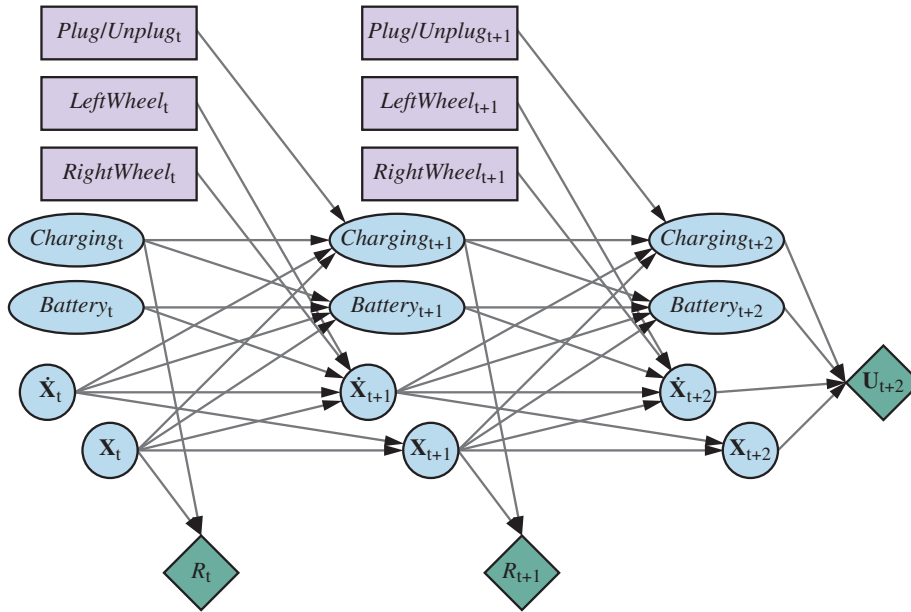


Figure 16.4 A dynamic decision network for a mobile robot with state variables for battery level, charging status, location, and velocity, and action variables for the left and right wheel motors and for charging.

Chapter 2; they typically have an exponential complexity advantage over atomic representations and can model quite substantial real-world problems.

Figure 16.4, which is based on the DBN in Figure 14.13(b) (page 504), shows some elements of a slightly realistic model for a mobile robot that can charge itself. The state S_t is decomposed into four state variables:

- \mathbf{X}_t consists of the two-dimensional location on a grid plus the orientation;
- $\dot{\mathbf{X}}_t$ is the rate of change of \mathbf{X}_t ;
- Charging_t is true when the robot is plugged in to a power source;
- Battery_t is the battery level, which we model as an integer in the range $0, \dots, 5$.

The state space for the MDP is the Cartesian product of the ranges of these four variables. The action is now a set \mathbf{A}_t of action variables, comprised of *Plug/Unplug*, which has three values (*plug*, *unplug*, and *noop*); *LeftWheel* for the power sent to the left wheel; and *RightWheel* for the power sent to the right wheel. The set of actions for the MDP is the Cartesian product of the ranges of these three variables. Notice that each action variable affects only a subset of the state variables.

The overall transition model is the conditional distribution $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, \mathbf{A}_t)$, which can be computed as a product of conditional probabilities from the DDN. The reward here is a single variable that depends only on the location \mathbf{X} (for, say, arriving at a destination) and *Charging*, as the robot has to pay for electricity used; in this particular model, the reward doesn't depend on the action or the outcome state.

The network in Figure 16.4 has been projected two steps into the future. Notice that the network includes nodes for the *rewards* for times t and $t+1$, but the *utility* for time $t+2$. This

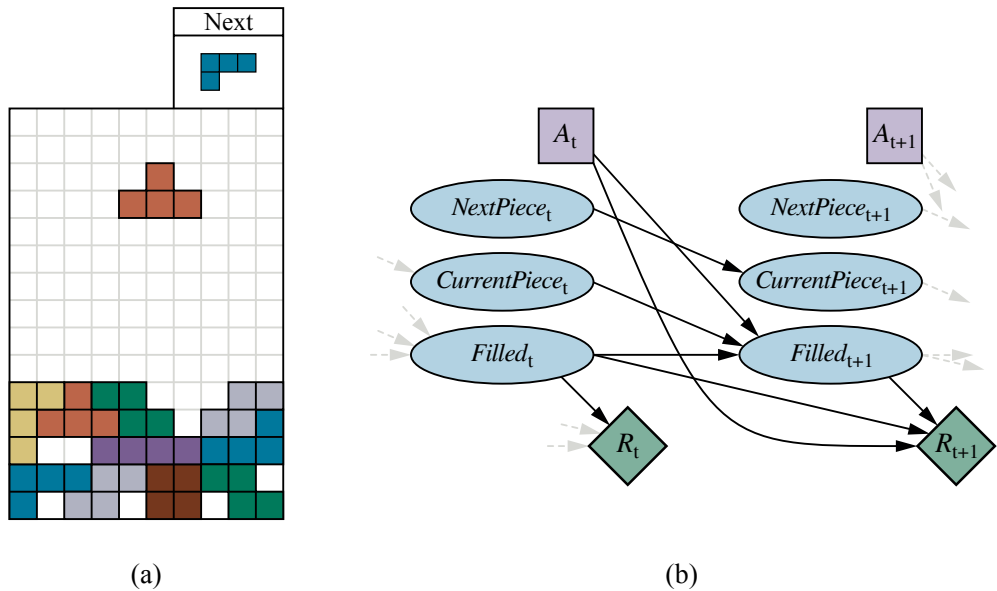


Figure 16.5 (a) The game of Tetris. The T-shaped piece at the top center can be dropped in any orientation and in any horizontal position. If a row is completed, that row disappears and the rows above it move down, and the agent receives one point. The next piece (here, the L-shaped piece at top right) becomes the current piece, and a new next piece appears, chosen at random from the seven piece types. The game ends if the board fills up to the top. (b) The DDN for the Tetris MDP.

is because the agent must maximize the (discounted) sum of all future rewards, and $U(\mathbf{X}_{t+3})$ represents the reward for all rewards from $t + 3$ onwards. If a heuristic approximation to U is available, it can be included in the MDP representation in this way and used in lieu of further expansion. This approach is closely related to the use of bounded-depth search and heuristic evaluation functions for games in Chapter 6.

Another interesting and well-studied MDP is the game of Tetris (Figure 16.5(a)). The state variables for the game are the *CurrentPiece*, the *NextPiece*, and a bit-vector-valued variable *Filled* with one bit for each of the 10×20 board locations. Thus, the state space has $7 \times 7 \times 2^{200} \approx 10^{62}$ states. The DDN for Tetris is shown in Figure 16.5(b). Note that $Filled_{t+1}$ is a deterministic function of $Filled_t$ and A_t . It turns out that every policy for Tetris is proper (reaches a terminal state): eventually the board fills despite one’s best efforts to empty it.

16.2 Algorithms for MDPs

In this section, we present four different algorithms for solving MDPs. The first three, **value iteration**, **policy iteration**, and **linear programming**, generate exact solutions offline. The fourth is a family of online approximate algorithms that includes **Monte Carlo planning**.

16.2.1 Value Iteration

The Bellman equation (Equation (16.5)) 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

Monte Carlo
planning

Value iteration

```

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

  repeat
     $U \leftarrow U'; \delta \leftarrow 0$ 
    for each state  $s$  in  $S$  do
       $U'[s] \leftarrow \max_{a \in A(s)} Q\text{-VALUE}(\text{mdp}, s, a, U)$ 
      if  $|U'[s] - U[s]| > \delta$  then  $\delta \leftarrow |U'[s] - U[s]|$ 
  until  $\delta \leq \epsilon(1 - \gamma)/\gamma$ 
  return  $U$ 

```

Figure 16.6 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (16.12).

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 i th iteration. The iteration step, called a **Bellman update**, looks like this:

Bellman update

$$U_{i+1}(s) \leftarrow \max_{a \in A(s)} \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U_i(s')], \quad (16.10)$$

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 “convergence of value iteration” below), 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 (16.4)) is optimal. The detailed algorithm, including a termination condition when the utilities are “close enough,” is shown in Figure 16.6. Notice that we make use of the Q-VALUE function defined on page 559.

We can apply value iteration to the 4×3 world in Figure 16.1(a). Starting with initial values of zero, the utilities evolve as shown in Figure 16.7(a). Notice how the states at different 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.

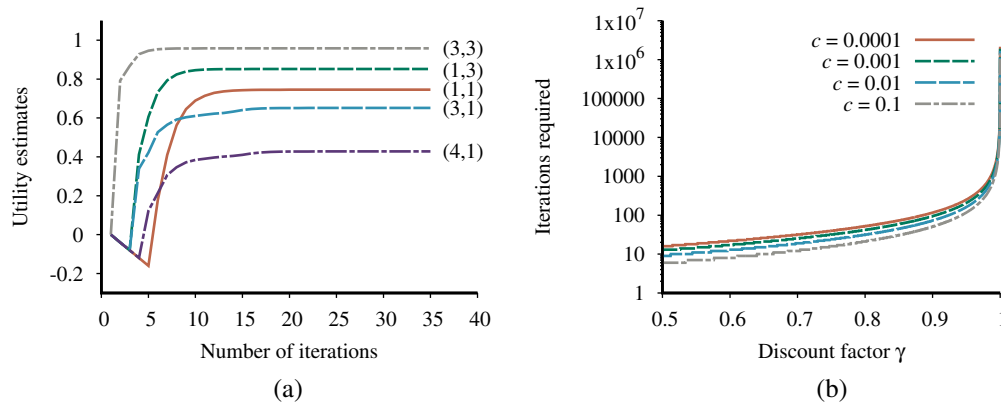


Figure 16.7 (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations required to guarantee an error of at most $\epsilon = c \cdot R_{\max}$, for different values of c , as a function of the discount factor γ .

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 (16.10)) as an operator B that is applied simultaneously to update the utility of every state. Then the Bellman equation becomes $U = BU$ and the Bellman update equation can be written as

$$U_{i+1} \leftarrow BU_i.$$


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

$$\|BU_i - BU'_i\| \leq \gamma \|U_i - U'_i\|. \quad (16.11)$$

That is, *the Bellman update is a contraction by a factor of γ on the space of utility vectors.* (Exercise 16.VICT 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 (16.11) with the *true* utilities U , for which $BU = U$. Then we obtain the inequality

$$\|BU_i - U\| \leq \gamma \|U_i - U\|.$$

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. Thus, value iteration converges exponentially fast. We can calculate the number of iterations required as follows: First, recall from Equation (16.1) 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 that


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

iterations suffice. Figure 16.7(b) shows how N varies with γ , for different values of the ratio ϵ/R_{\max} . The good news is that, because of the exponentially fast convergence, N does not depend much on the ratio ϵ/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 (16.11)), 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,

$$\text{if } \|U_{i+1} - U_i\| < \epsilon(1 - \gamma)/\gamma \text{ then } \|U_{i+1} - U\| < \epsilon. \quad (16.12)$$

This is the termination condition used in the VALUE-ITERATION algorithm of Figure 16.6.

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 maximum expected utility (MEU) policy π_i based on one-step look-ahead using U_i (as in Equation (16.4)). 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** $\|U^{\pi_i} - U\|$ is the most the agent can lose by  Policy loss

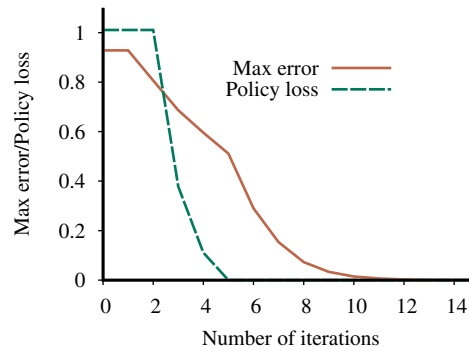


Figure 16.8 The maximum error $\|U_i - U\|$ of the utility estimates and the policy loss $\|U^{\pi_i} - U\|$, as a function of the number of iterations of value iteration on the 4×3 world.

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:

$$\text{if } \|U_i - U\| < \epsilon \quad \text{then} \quad \|U^{\pi_i} - U\| < 2\epsilon. \quad (16.13)$$

In practice, it often occurs that π_i becomes optimal long before U_i has converged. Figure 16.8 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 $i=5$, even though the maximum error in U_i is still 0.51.

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.

16.2.2 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 (16.4)).

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

```

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

  repeat
     $U \leftarrow \text{POLICY-EVALUATION}(\pi, U, \textit{mdp})$ 
    unchanged?  $\leftarrow$  true
    for each state s in S do
       $a^* \leftarrow \underset{a \in A(s)}{\text{argmax}} \text{ Q-VALUE}(\textit{mdp}, s, a, U)$ 
      if Q-VALUE(mdp, s,  $a^*$ , U) > Q-VALUE(mdp, s,  $\pi[s]$ , U) then
         $\pi[s] \leftarrow a^*$ ; unchanged?  $\leftarrow$  false
  until unchanged?
  return  $\pi$ 

```

Figure 16.9 The policy iteration algorithm for calculating an optimal policy.

better policy, policy iteration must terminate. The algorithm is shown in Figure 16.9. As with value iteration, we use the Q-VALUE function defined on page 559.

How do we implement POLICY-EVALUATION? It turns out that doing so is 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 *i*th iteration, the policy π_i specifies the action $\pi_i(s)$ in state *s*. This means that we have a simplified version of the Bellman equation (16.5) relating the utility of *s* (under π_i) to the utilities of its neighbors:

$$U_i(s) = \sum_{s'} P(s' | s, \pi_i(s)) [R(s, \pi_i(s), s') + \gamma U_i(s')]. \quad (16.14)$$

For example, suppose π_i is the policy shown in Figure 16.2(a). Then we have $\pi_i(1, 1) = Up$, $\pi_i(1, 2) = Up$, and so on, and the simplified Bellman equations are

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

and so on for all the states. 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. If the transition model is sparse—that is, if each state transitions only to a small number of other states—then the solution process can be faster still.

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

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

Modified policy
iteration

and this is repeated several times to efficiently produce the next utility estimate. The resulting algorithm is called **modified policy iteration**.

Asynchronous policy
iteration

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. There's no sense planning for the results of an action you will never do.

16.2.3 Linear programming

Linear programming or LP, which was mentioned briefly in Chapter 4 (page 139), is a general approach for formulating constrained optimization problems, and there are many industrial-strength LP solvers available. Given that the Bellman equations involve a lot of sums and maxes, it is perhaps not surprising that solving an MDP can be reduced to solving a suitably formulated linear program.

The basic idea of the formulation is to consider as variables in the LP the utilities $U(s)$ of each state s , noting that the utilities for an optimal policy are the highest utilities attainable that are consistent with the Bellman equations. In LP language, that means we seek to minimize $U(s)$ for all s subject to the inequalities

$$U(s) \geq \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U(s')]$$

for every state s and every action a .

This creates a connection from dynamic programming to linear programming, for which algorithms and complexity issues have been studied in great depth. For example, from the fact that linear programming is solvable in polynomial time, one can show that MDPs can be solved in time polynomial in the number of states and actions and the number of bits required to specify the model. In practice, it turns out that LP solvers are seldom as efficient as dynamic programming for solving MDPs. Moreover, polynomial time may sound good, but the number of states is often very large. Finally, it's worth remembering that even the simplest and most uninformed of the search algorithms in Chapter 3 runs in linear time in the number of states and actions.

16.2.4 Online algorithms for MDPs

Value iteration and policy iteration are *offline* algorithms: like the A* algorithm in Chapter 3, they generate an optimal solution for the problem, which can then be executed by a simple agent. For sufficiently large MDPs, such as the Tetris MDP with 10^{62} states, exact offline solution, even by a polynomial-time algorithm, is not possible. Several techniques have been developed for approximate offline solution of MDPs; these are covered in the notes at the end of the chapter and in Chapter 23 (Reinforcement Learning).

Here we will consider online algorithms, analogous to those used for game playing in Chapter 6, where the agent does a significant amount of computation at each decision point rather than operating primarily with precomputed information.

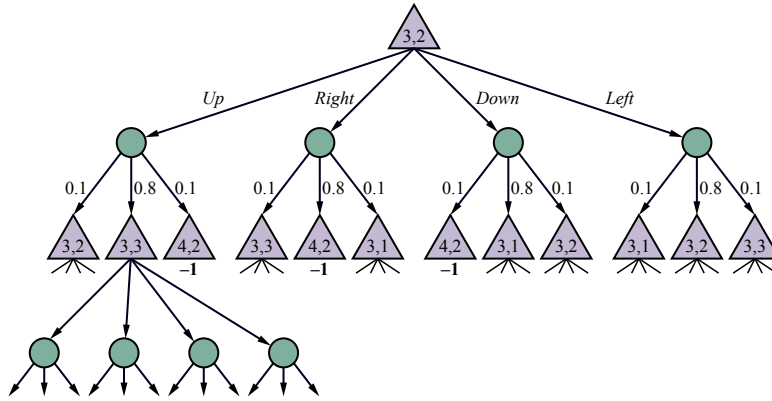


Figure 16.10 Part of an expectimax tree for the 4×3 MDP rooted at $(3,2)$. The triangular nodes are max modes and the circular nodes are chance nodes.

The most straightforward approach is actually a simplification of the EXPECTIMINIMAX algorithm for game trees with chance nodes: the EXPECTIMAX algorithm builds a tree of alternating max and chance nodes, as illustrated in Figure 16.10. (There is a slight difference from standard EXPECTIMINIMAX in that there are rewards on nonterminal as well as terminal transitions.) An evaluation function can be applied to the nonterminal leaves of the tree, or they can be given a default value. A decision can be extracted from the search tree by backing up the utility values from the leaves, taking an average at the chance nodes and taking the maximum at the decision nodes.

For problems in which the discount factor γ is not too close to 1, the ϵ -horizon is a useful concept. Let ϵ be a desired bound on the absolute error in the utilities computed from an expectimax tree of bounded depth, compared to the exact utilities in the MDP. Then the ϵ -horizon is the tree depth H such that the sum of rewards beyond any leaf at that depth is less than ϵ —roughly speaking, anything that happens after H is irrelevant because it's so far in the future. Because the sum of rewards beyond H is bounded by $\gamma^H R_{\max} / (1 - \gamma)$, a depth of $H = \lceil \log_{\gamma} \epsilon(1 - \gamma) / R_{\max} \rceil$ suffices. So, building a tree to this depth gives near-optimal decisions. For example, with $\gamma = 0.5$, $\epsilon = 0.1$, and $R_{\max} = 1$, we find $H = 5$, which seems reasonable. On the other hand, if $\gamma = 0.9$, $H = 44$, which seems less reasonable!

In addition to limiting the depth, it is also possible to avoid the potentially enormous branching factor at the chance nodes. (For example, if all the conditional probabilities in a DBN transition model are nonzero, the transition probabilities, which are given by the product of the conditional probabilities, are also nonzero, meaning that every state has *some* probability of transitioning to every other state.)

As noted in Section 13.4, expectations with respect to a probability distribution P can be approximated by generating N samples from P and using the sample mean. In mathematical form, we have

$$\sum_x P(x) f(x) \approx \frac{1}{N} \sum_{i=1}^N f(x_i).$$

So, if the branching factor is very large, meaning that there are very many possible x values, a good approximation to the value of the chance node can be obtained by sampling a bounded

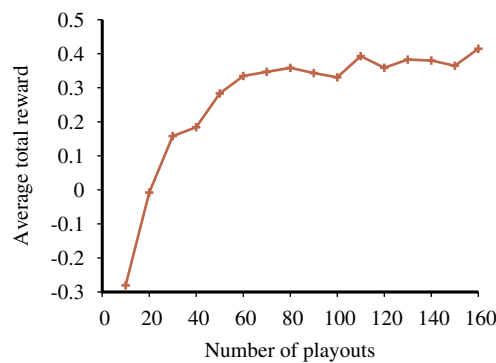


Figure 16.11 Performance of UCT as a function of the number of playouts per move for the 4×3 world using a random playout policy, averaged over 1000 runs per data point.

number of outcomes from the action. Typically, the samples will focus on the *most likely* outcomes because those are most likely to be generated.

If you look closely at the tree in Figure 16.10, you will notice something: it isn't really a tree. For example, the root (3,2) is also a leaf, so one ought to consider this as a graph, and one ought to constrain the value of the leaf (3,2) to be the same as the value of the root (3,2), since they are the same state. In fact, this line of thinking quickly brings us back to the Bellman equations that relate the values of states to the values of neighboring states. The explored states actually constitute a sub-MDP of the original MDP, and this sub-MDP can be solved using any of the algorithms in this chapter to yield a decision for the current state. (Frontier states are typically given a fixed estimated value.)

Real-time dynamic
programming
(RTDP)

This general approach is called **real-time dynamic programming (RTDP)** and is quite analogous to LRTA* in Chapter 4. Algorithms of this kind can be quite effective in moderate-sized domains such as grid worlds; in larger domains such as Tetris, there are two issues. First, the state space is such that any manageable set of explored states contains very few repeated states, so one might as well use a simple expectimax tree. Second, a simple heuristic for frontier nodes may not be enough to guide the agent, particularly if rewards are sparse.

One possible fix is to apply reinforcement learning to generate a much more accurate heuristic (see Chapter 23). Another approach is to look further ahead in the MDP using the Monte Carlo approach of Section 6.4. In fact, the UCT algorithm from Figure 6.10 was developed originally for MDPs rather than games. The changes required to solve MDPs rather than games are minimal: they arise primarily from the fact that the opponent (nature) is stochastic and from the need to keep track of rewards rather than just wins and losses.

When applied to the 4×3 world, the performance of UCT is not especially impressive. As Figure 16.11 shows, it takes 160 playouts on average to reach a total reward of 0.4, whereas an optimal policy has an expected total reward of 0.7453 from the initial state (see Figure 16.3). One reason UCT can have difficulty is that it builds a tree rather than a graph and uses (an approximation to) expectimax rather than dynamic programming. The 4×3 world is very “loopy”: although there are only 9 nonterminal states, UCT's playouts often continue for more than 50 actions.

UCT seems better suited for Tetris, where the playouts go far enough into the future to give the agent a sense of whether a potentially risky move will work out in the end or cause a massive pile-up. Exercise 16.UCTT explores the application of UCT to Tetris. One particularly interesting question is how much a simple simulation policy can help—for example, one that avoids creating overhangs and puts pieces as low as possible.

16.3 Bandit Problems

In Las Vegas, a *one-armed bandit* is a slot machine. A gambler can insert a coin, pull the lever, and collect the winnings (if any). An **n-armed bandit** has n levers. Behind each lever is a fixed but unknown probability distribution of winnings; each pull samples from that unknown distribution.

N-armed bandit

The gambler must choose which lever to play on each successive coin—the one that has paid off best, or maybe one that has not been tried yet? This is an example of the ubiquitous tradeoff between **exploitation** of the current best action to obtain rewards and **exploration** of previously unknown states and actions to gain information, which can in some cases be converted into a better policy and better long-term rewards. In the real world, one constantly has to decide between continuing in a comfortable existence, versus striking out into the unknown in the hopes of a better life.

The n -armed bandit problem is a formal model for real problems in many vitally important areas, such as deciding which of n possible new treatments to try to cure a disease, which of n possible investments to put part of your savings into, which of n possible research projects to fund, or which of n possible advertisements to show when the user visits a particular web page.

Early work on the problem began in the U.S. during World War II; it proved so recalcitrant that Allied scientists proposed that “the problem be dropped over Germany, as the ultimate instrument of intellectual sabotage” (Whittle, 1979).

It turns out that the scientists, both during and after the war, were trying to prove “obviously true” facts about bandit problems that are, in fact, false. (As Bradt *et al.* (1956) put it, “There are many nice properties which optimal strategies do not possess.”) For example, it was generally assumed that an optimal policy would eventually settle on the best arm in the long run; in fact, there is a finite probability that an optimal policy settles on a suboptimal arm. We now have a solid theoretical understanding of bandit problems as well as useful algorithms for solving them.

There are several different definitions of **bandit problems**; one of the cleanest and most general is as follows:

Bandit problems

- Each arm M_i is a **Markov reward process** or MRP, that is, an MDP with only one possible action a_i . It has states S_i , transition model $P_i(s' | s, a_i)$, and reward $R_i(s, a_i, s')$. The arm defines a distribution over sequences of rewards $R_{i,0}, R_{i,1}, R_{i,2}, \dots$, where each $R_{i,t}$ is a random variable.
- The overall bandit problem is an MDP: the state space is given by the Cartesian product $S = S_1 \times \dots \times S_n$; the actions are a_1, \dots, a_n ; the transition model updates the state of whichever arm M_i is selected, according to its specific transition model, leaving the other arms unchanged; and the discount factor is γ .

Markov reward process

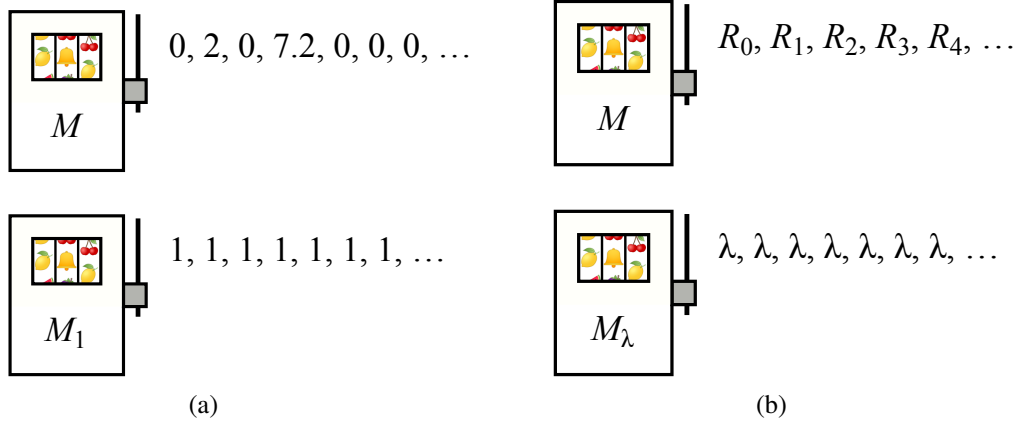


Figure 16.12 (a) A simple deterministic bandit problem with two arms. The arms can be pulled in any order, and each yields the sequence of rewards shown. (b) A more general case of the bandit in (a), where the first arm gives an arbitrary sequence of rewards and the second arm gives a fixed reward λ .

This definition is very general, covering a wide range of cases. The key property is that the arms are independent, coupled only by the fact that the agent can work on only one arm at a time. It's possible to define a still more general version in which fractional efforts can be applied to all arms simultaneously, but the total effort across all arms is bounded; the basic results described here carry over to this case.

We will see shortly how to formulate a typical bandit problem within this framework, but let's warm up with the simple special case of deterministic reward sequences. Let $\gamma = 0.5$, and suppose that there are two arms labeled M and M_1 . Pulling M multiple times yields the sequence of rewards 0, 2, 0, 7.2, 0, 0, ..., while pulling M_1 yields 1, 1, 1, ... (Figure 16.12(a)). If, at the beginning, one had to commit to one arm or the other and stick with it, the choice would be made by computing the utility (total discounted reward) for each arm:

$$U(M) = (1.0 \times 0) + (0.5 \times 2) + (0.5^2 \times 0) + (0.5^3 \times 7.2) = 1.9$$

$$U(M_1) = \sum_{t=0}^{\infty} 0.5^t = 2.0.$$

One might think the best choice is to go with M_1 , but a moment's more thought shows that starting with M and then switching to M_1 after the fourth reward gives the sequence $S = 0, 2, 0, 7.2, 1, 1, 1, \dots$, for which

$$U(S) = (1.0 \times 0) + (0.5 \times 2) + (0.5^2 \times 0) + (0.5^3 \times 7.2) + \sum_{t=4}^{\infty} 0.5^t = 2.025.$$

Hence the strategy S that switches from M to M_1 at the right time is better than either arm individually. In fact, S is optimal for this problem: all other switching times give less reward.

Let's generalize this case slightly, so that now the first arm M yields an arbitrary sequence R_0, R_1, R_2, \dots (which may be known or unknown) and the second arm M_λ yields $\lambda, \lambda, \lambda, \dots$ for some known fixed constant λ (see Figure 16.12(b)). This is called a **one-armed bandit** in the literature, because it is formally equivalent to the case where there is one arm M that produces rewards R_0, R_1, R_2, \dots and costs λ for each pull. (Pulling arm M is equivalent to not

pulling M_λ , so it gives up a reward of λ each time.) With just one arm, the only choice is to whether to pull again or to stop. If you pull the first arm T times (i.e., at times $0, 1, \dots, T-1$ we say that the **stopping time** is T . Stopping time

Going back to our version with M and M_λ , let's assume that after T pulls of the first arm, an optimal strategy eventually pulls the second arm for the first time. Since no information is gained from this move (we already know the payoff will be λ), at time $T+1$ we will be in the same situation and thus an optimal strategy must make the same choice.

Equivalently, we can say that an optimal strategy is to run arm M up to time T and then switch to M_λ for the rest of time. It's possible that $T=0$ if the strategy chooses M_λ immediately, or $T=\infty$ if the strategy never chooses M_λ , or somewhere in between. Now let's consider the value of λ such that an optimal strategy is *exactly indifferent* between (a) running M up to the best possible stopping time and then switching to M_λ forever, and (b) choosing M_λ immediately. At the tipping point we have

$$\max_{T \geq 0} E \left[\left(\sum_{t=0}^{T-1} \gamma^t R_t \right) + \sum_{t=T}^{\infty} \gamma^t \lambda \right] = \sum_{t=0}^{\infty} \gamma^t \lambda,$$

which simplifies to

$$\lambda = \max_{T \geq 0} \frac{E \left(\sum_{t=0}^{T-1} \gamma^t R_t \right)}{E \left(\sum_{t=0}^{T-1} \gamma^t \right)}. \quad (16.15)$$

This equation defines a kind of “value” for M in terms of its ability to deliver a stream of timely rewards; the numerator of the fraction represents a utility while the denominator can be thought of as a “discounted time,” so the value describes the maximum obtainable utility per unit of discounted time. (It's important to remember that T in the equation is a stopping time, which is governed by a rule for stopping rather than being a simple integer; it reduces to a simple integer only when M is a deterministic reward sequence.) The value defined in Equation (16.15) is called the **Gittins index** of M . Gittins index

The remarkable thing about the Gittins index is that it provides a very simple optimal policy for any bandit problem: *pull the arm that has the highest Gittins index, then update the Gittins indices*. Furthermore, because the index of arm M_i depends only on the properties of that arm, an optimal decision on the first iteration can be calculated in $O(n)$ time, where n is the number of arms. And because the Gittins indices of the arms that are not selected remain unchanged, each decision after the first one can be calculated in $O(1)$ time. ◀

16.3.1 Calculating the Gittins index

To get more of a feel for the index, let's calculate the value of the numerator, denominator, and ratio in Equation (16.15) for different possible stopping times on the deterministic reward sequence $0, 2, 0, 7.2, 0, 0, 0, \dots$:

T	1	2	3	4	5	6
R_t	0	2	0	7.2	0	0
$\sum \gamma^t R_t$	0.0	1.0	1.0	1.9	1.9	1.9
$\sum \gamma^t$	1.0	1.5	1.75	1.875	1.9375	1.9687
ratio	0.0	0.6667	0.5714	1.0133	0.9806	0.9651

Clearly, the ratio will decrease from here on, because the numerator remains constant while the denominator continues to increase. Thus, the Gittins index for this arm is 1.0133, the

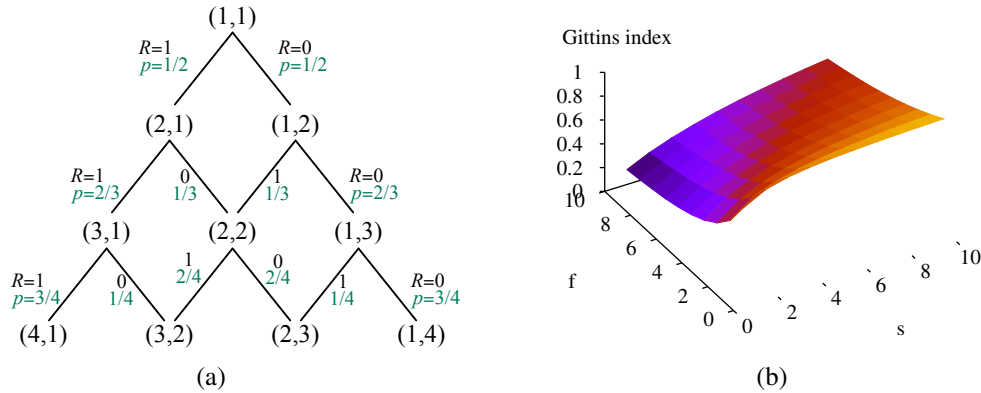


Figure 16.14 (a) States, rewards, and transition probabilities for the Bernoulli bandit. (b) Gittins indices for the states of the Bernoulli bandit process.

We cannot quite apply the transformation of the preceding section to calculate the Gittins index of the Bernoulli arm because it has infinitely many states. We can, however, obtain a very accurate approximation by solving the truncated MDP with states up to $s_i + f_i = 100$ and $\gamma = 0.9$. The results are shown in Figure 16.14(b). The results are intuitively reasonable: we see that, generally speaking, arms with higher payoff probabilities are preferred, but there is also an **exploration bonus** associated with arms that have only been tried a few times. For example, the index for the state (3,2) is higher than the index for the state (7,4) (0.7057 vs. 0.6922), even though the estimated value at (3,2) is lower (0.6 vs. 0.6364).

Exploration bonus

16.3.3 Approximately optimal bandit policies

Calculating Gittins indices for more realistic problems is rarely easy. Fortunately, the general properties observed in the preceding section—namely, the desirability of some combination of estimated value and uncertainty—lend themselves to the creation of simple policies that turn out to be “nearly as good” as optimal policies.

The first class of methods uses the **upper confidence bound** or UCB heuristic, previously introduced for Monte Carlo tree search (Figure 6.11 on page 209). The basic idea is to use the samples from each arm to establish a **confidence interval** for the value of the arm, that is, a range within which the value can be estimated to lie with high confidence; then choose the arm with the highest upper bound on its confidence interval. The upper bound is the current mean value estimate $\hat{\mu}_i$ plus some multiple of the standard deviation of the uncertainty in the value. The standard deviation is proportional to $\sqrt{1/N_i}$, where N_i is the number of times arm M_i has been sampled. So we have an approximate index value for arm M_i given by

Upper confidence bound

$$UCB(M_i) = \hat{\mu}_i + g(N)/\sqrt{N_i},$$

where $g(N)$ is an appropriately chosen function of N , the total number of samples drawn from all arms. A UCB policy simply picks the arm with the highest UCB value. Notice that the UCB value is not strictly an index because it depends on N , the total number of samples drawn across all arms, and not just on the arm itself.

The precise definition of g determines the **regret** relative to the clairvoyant policy, which simply picks the best arm and yields average reward μ^* . A famous result due to Lai and

Robbins (1985) shows that, for the undiscounted case, no possible algorithm can have regret that grows more slowly than $O(\log N)$. Several different choices of g lead to a UCB policy that matches this growth; for example, we can use $g(N) = (2\log(1 + N\log^2 N))^{1/2}$.

Thompson sampling

A second method, **Thompson sampling** (Thompson, 1933), chooses an arm randomly according to the probability that the arm is in fact optimal, given the samples so far. Suppose that $P_i(\mu_i)$ is the current probability distribution for the true value of arm M_i . Then a simple way to implement Thompson sampling is to generate one sample from each P_i and then pick the best sample. This algorithm also has a regret that grows as $O(\log N)$.

16.3.4 Non-indexable variants

Bandit problems were motivated in part by the task of testing new medical treatments on seriously ill patients. For this task, the goal of maximizing the total number of successes over time clearly makes sense: each successful test means a life saved, each failure a life lost.

If we change the assumptions slightly, however, a different problem emerges. Suppose that, instead of determining the best medical treatment for each new human patient, we are instead testing different drugs on samples of bacteria with the goal of deciding which drug is best. We will then put that drug into production and forgo the others. In this scenario there is no additional cost if the bacteria dies—there is a fixed cost for each test, but we don't have to minimize test failures; rather we are just trying to make a good decision as fast as possible.

Selection problem

The task of choosing the best option under these conditions is called a **selection problem**. Selection problems are ubiquitous in industrial and personnel contexts. One often must decide which supplier to use for a process; or which candidate employees to hire. Selection problems are superficially similar to the bandit problem but have different mathematical properties. In particular, *no index function exists for selection problems*. The proof of this fact requires showing any scenario where the optimal policy switches its preferences for two arms M_1 and M_2 when a third arm M_3 is added (see Exercise 16.SELC).

Chapter 6 introduced the concept of **metalevel** decision problems such as deciding what computations to make during a game-tree search prior to making a move. A metalevel decision of this kind is also a selection problem rather than a bandit problem. Clearly, a node expansion or evaluation *costs* the same amount of time whether it produces a high or a low output value. It is perhaps surprising, then, that the Monte Carlo tree search algorithm (see page 209) has been so successful, given that it tries to solve selection problems with the UCB heuristic, which was designed for bandit problems. Generally speaking, one expects optimal bandit algorithms to explore much less than optimal selection algorithms, because the bandit algorithm assumes that a failed trial costs real money.

Bandit superprocess
BSP

An important generalization of the bandit process is the **bandit superprocess** or **BSP**, in which each arm is a full Markov decision process in its own right, rather than being a Markov reward process with only one possible action. All other properties remain the same: the arms are independent, only one (or a bounded number) can be worked on at a time, and there is a single discount factor.

Examples of BSPs include daily life, where one can attend to one task at a time, even though several tasks may need attention; project management with multiple projects; teaching with multiple pupils needing individual guidance; and so on. The ordinary term for this is **multitasking**. It is so ubiquitous as to be barely noticeable: when formulating a real-world decision problem, decision analysts rarely ask if their client has other, unrelated problems.

Multitasking

One might reason as follows: “If there are n disjoint MDPs then it is obvious that an optimal policy overall is built from the optimal solutions of the individual MDPs. Given its optimal policy π_i , each MDP becomes a Markov reward process where there is only one action $\pi_i(s)$ in each state s . So we have reduced the n -armed bandit superprocess to an n -armed bandit process.” For example, if a real-estate developer has one construction crew and several shopping centers to build, it seems to be just common sense that one should devise the optimal construction plan for each shopping center and then solve the bandit problem to decide where to send the crew each day.

While this sounds highly plausible, it is incorrect. In fact, the globally optimal policy for a BSP may include actions that are locally suboptimal from the point of view of the constituent MDP in which they are taken. The reason for this is that the availability of other MDPs in which to act changes the balance between short-term and long-term rewards in a component MDP. In fact, it tends to lead to greedier behavior in each MDP (seeking short-term rewards) because aiming for long-term reward in one MDP would delay rewards in all the other MDPs.

For example, suppose the locally optimal construction schedule for one shopping center has the first shop available for rent by week 15, whereas a suboptimal schedule costs more but has the first shop available by week 5. If there are four shopping centers to build, it might be better to use the locally suboptimal schedule in each so that rents start coming in from weeks 5, 10, 15, and 20, rather than weeks 15, 30, 45, and 60. In other words, what would be only a 10-week delay for a single MDP turns into a 40-week delay for the fourth MDP. In general, the globally and locally optimal policies necessarily coincide only when the discount factor is 1; in that case, there is no cost to delaying rewards in any MDP.

The next question is how to solve BSPs. Obviously, the globally optimal solution for a BSP could be computed by converting it into a global MDP on the Cartesian-product state space. The number of states would be exponential in the number of arms of the BSP, so this would be horrendously impractical.

Instead, we can take advantage of the loose nature of the interaction between the arms. This interaction arises only from the agent’s limited ability to attend to the arms simultaneously. To some extent, the interaction can be modeled by the notion of **opportunity cost**: how much utility is given up per time step by not devoting that time step to another arm. The higher the opportunity cost, the more necessary it is to generate early rewards in a given arm. In some cases, an optimal policy in a given arm is unaffected by the opportunity cost. (Trivially, this is true in a Markov reward process because there is only one policy.) In that case, an optimal policy can be applied, converting that arm into a Markov reward process.

Opportunity cost

Such an optimal policy, if it exists, is called a **dominating policy**. It turns out that by adding actions to states, it is always possible to create a relaxed version of an MDP (see Section 3.6.2) so that it has a dominating policy, which thus gives an upper bound on the value of acting in the arm. A lower bound can be computed by solving each arm separately (which may yield a suboptimal policy overall) and then computing the Gittins indices. If the lower bound for acting in one arm is higher than the upper bounds for all other actions, then the problem is solved; if not, then a combination of look-ahead search and recomputation of bounds is guaranteed to eventually identify an optimal policy for the BSP. With this approach, relatively large BSPs (10^{40} states or more) can be solved in a few seconds.

Dominating policy

16.4 Partially Observable MDPs

The description of Markov decision processes in Section 16.1 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

16.4.1 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 $P(s'|s, a)$, actions $A(s)$, and reward function $R(s, a, s')$ —but, like the partially observable search problems of Section 4.4, it also has a **sensor model** $P(e|s)$. Here, as in Chapter 14, the sensor model specifies the probability of perceiving evidence e in state s .⁵ For example, we can convert the 4×3 world of Figure 16.1 into a POMDP by adding a noisy or partial sensor instead of assuming that the agent knows its location exactly. The noisy four-bit sensor from page 494 could be used, which reports the presence or absence of a wall in each compass direction with accuracy $1 - \epsilon$.

As with MDPs, we can obtain compact representations for large POMDPs by using dynamic decision networks (see Section 16.1.4). We add sensor variables \mathbf{E}_t , assuming that the state variables \mathbf{X}_t may not be directly observable. The POMDP sensor model is then given by $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$. For example, we might add sensor variables to the DDN in Figure 16.4 such as *BatteryMeter_t* to estimate the actual charge *Battery_t* and *Speedometer_t* to estimate the magnitude of the velocity vector $\dot{\mathbf{X}}_t$. A sonar sensor *Walls_t* might give estimated distances to the nearest wall in each of the four cardinal directions relative to the robot’s current orientation; these values depends on the current position and orientation \mathbf{X}_t .

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 14. For example, the initial belief state for the 4×3 POMDP could be the uniform distribution over the nine nonterminal states along with 0s for the terminal states, that is, $\langle \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, 0 \rangle$.

We use the notation $b(s)$ to refer to the probability assigned to the actual state s by belief state b . The agent can calculate its current belief state as the conditional probability distribution over the actual states given the sequence of percepts and actions so far. This is essentially the **filtering** task described in Chapter 14. The basic recursive filtering equation (14.5 on page 485) shows how to calculate the new belief state from the previous belief state and the new evidence. For POMDPs, we also have an action to consider, but the result is essentially the same. If b was the previous belief state, and the agent does action a and then

⁵ The sensor model can also depend on the action and outcome state, but this change is not fundamental.

perceives evidence e , then the new belief state is obtained by calculating the probability of now being in state s' , for each s' , with the following formula:

$$b'(s') = \alpha P(e|s') \sum_s P(s'|s, a) b(s),$$

where α is a normalizing constant that makes the belief state sum to 1. By analogy with the update operator for filtering (page 485), we can write this as

$$b' = \alpha \text{FORWARD}(b, a, e). \quad (16.16)$$

In the 4×3 POMDP, suppose the agent moves *Left* and its sensor reports one adjacent wall; then it's quite likely (although not guaranteed, because both the motion and the sensor are noisy) that the agent is now in (3,1). Exercise 16.POMD asks you to calculate the exact probability values for the new belief state.

The fundamental insight required to understand POMDPs is this: *the optimal action depends only on the agent's current belief state*. That is, an optimal policy can be described by a mapping $\pi^*(b)$ from belief states to actions. It does *not* depend on the *actual* state the agent is in. This is a good thing, because the agent does not know its actual state; all it knows is the belief state. Hence, the decision cycle of a POMDP agent can be broken down into the following three steps:

1. Given the current belief state b , execute the action $a = \pi^*(b)$.
2. Observe the percept e .
3. Set the current belief state to $\text{FORWARD}(b, a, e)$ and repeat.

We can think of POMDPs as requiring a search in belief-state space, just like the methods for sensorless and contingency problems in Chapter 4. The main difference is that the POMDP belief-state space is *continuous*, because a POMDP belief state is a probability distribution. For example, a belief state for the 4×3 world is a point in an 11-dimensional continuous space. An action changes the belief state, not just the physical state, because it affects the percept that is received. Hence, the action is evaluated at least in part according to the information the agent acquires as a result. POMDPs therefore include the value of information (Section 15.6) as one component of the decision problem.

Let's look more carefully at the outcome of actions. In particular, let's calculate the probability that an agent in belief state b reaches belief state b' after executing action a . Now, if we knew the action *and the subsequent percept*, then Equation (16.16) would provide a *deterministic* update to the belief state: $b' = \text{FORWARD}(b, a, e)$. Of course, the subsequent percept is not yet known, so the agent might arrive in one of several possible belief states b' , depending on the percept that is received. The probability of perceiving e , given that a was performed starting in belief state b , is given by summing over all the actual states s' that the agent might reach:

$$\begin{aligned} P(e|a, b) &= \sum_{s'} P(e|a, s', b) P(s'|a, b) \\ &= \sum_{s'} P(e|s') P(s'|a, b) \\ &= \sum_{s'} P(e|s') \sum_s P(s'|s, a) b(s). \end{aligned}$$

Let us write the probability of reaching b' from b , given action a , as $P(b' | b, a)$. This probability can be calculated as follows:

$$\begin{aligned} P(b' | b, a) &= \sum_e P(b' | e, a, b) P(e | a, b) \\ &= \sum_e P(b' | e, a, b) \sum_{s'} P(e | s') \sum_s P(s' | s, a) b(s), \end{aligned} \quad (16.17)$$

where $P(b' | e, a, b)$ is 1 if $b' = \text{FORWARD}(b, a, e)$ and 0 otherwise.

Equation (16.17) can be viewed as defining a transition model for the belief-state space. We can also define a reward function for belief-state transitions, which is derived from the expected reward of the real state transitions that might be occurring. Here, we use the simple form $\rho(b, a)$, the expected reward if the agent does a in belief state b :

$$\rho(b, a) = \sum_s b(s) \sum_{s'} P(s' | s, a) R(s, a, s').$$

Together, $P(b' | b, a)$ and $\rho(b, a)$ define an *observable* MDP on the space of belief states. Furthermore, it can be shown that an optimal policy for this MDP, $\pi^*(b)$, is also an optimal policy for the original POMDP. In other words, *solving a POMDP on a physical state space can be reduced to solving an MDP on the corresponding belief-state space*. This fact is perhaps less surprising if we remember that the belief state is always observable to the agent, by definition.

16.5 Algorithms for Solving POMDPs

We have shown how to reduce POMDPs to MDPs, but the MDPs we obtain have a continuous (and usually high-dimensional) state space. This means we will have to redesign the dynamic programming algorithms from Sections 16.2.1 and 16.2.2, which assumed a finite state space and a finite number of actions. Here we describe a value iteration algorithm designed specifically for POMDPs, followed by an online decision-making algorithm similar to those developed for games in Chapter 6.

16.5.1 Value iteration for POMDPs

Section 16.2.1 described a value iteration algorithm that computed one utility value for each state. With infinitely many belief states, we need to be more creative. Consider an optimal policy π^* and its application in a specific belief state b : the policy generates an action, then, for each subsequent percept, the belief state is updated and a new action is generated, and so on. For this specific b , therefore, the policy is exactly equivalent to a **conditional plan**, as defined in Chapter 4 for nondeterministic and partially observable problems. Instead of thinking about policies, let us think about conditional plans and how the expected utility of executing a fixed conditional plan varies with the initial belief state. We make two observations:

1. Let the utility of executing a *fixed* conditional plan p starting in physical state s be $\alpha_p(s)$. Then the expected utility of executing p in belief state b is just $\sum_s b(s) \alpha_p(s)$, or $b \cdot \alpha_p$ if we think of them both as vectors. Hence, the expected utility of a fixed conditional plan varies *linearly* with b ; that is, it corresponds to a hyperplane in belief space.
2. At any given belief state b , an optimal policy will choose to execute the conditional plan with highest expected utility; and the expected utility of b under an optimal policy is just

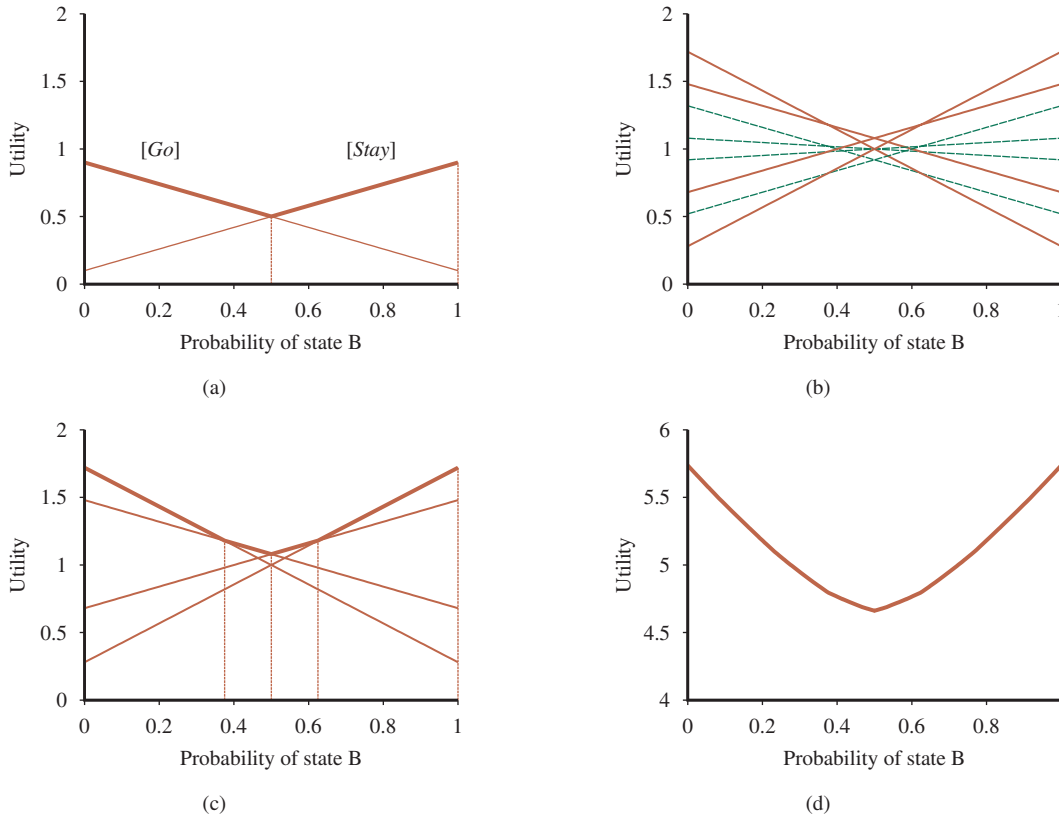


Figure 16.15 (a) Utility of two one-step plans as a function of the initial belief state $b(B)$ for the two-state world, with the corresponding utility function shown in bold. (b) Utilities for 8 distinct two-step plans. (c) Utilities for four undominated two-step plans. (d) Utility function for optimal eight-step plans.

the utility of that conditional plan: $U(b) = U^{\pi^*}(b) = \max_p b \cdot \alpha_p$. If an optimal policy π^* chooses to execute p starting at b , then it is reasonable to expect that it might choose to execute p in belief states that are very close to b ; in fact, if we bound the depth of the conditional plans, then there are only finitely many such plans and the continuous space of belief states will generally be divided into *regions*, each corresponding to a particular conditional plan that is optimal in that region.

From these two observations, we see that the utility function $U(b)$ on belief states, being the maximum of a collection of hyperplanes, will be *piecewise linear* and *convex*.

To illustrate this, we use a simple two-state world. The states are labeled A and B and there are two actions: *Stay* stays put with probability 0.9 and *Go* switches to the other state with probability 0.9. The rewards are $R(\cdot, \cdot, A) = 0$ and $R(\cdot, \cdot, B) = 1$; that is, any transition ending in A has reward zero and any transition ending in B has reward 1. For now we will assume the discount factor $\gamma = 1$. The sensor reports the correct state with probability 0.6. Obviously, the agent should *Stay* when it's in state B and *Go* when it's in state A . The problem is that it doesn't know where it is!

The advantage of a two-state world is that the belief space can be visualized in one dimension, because the two probabilities $b(A)$ and $b(B)$ sum to 1. In Figure 16.15(a), the x -axis

represents the belief state, defined by $b(B)$, the probability of being in state B . Now let us consider the one-step plans $[Stay]$ and $[Go]$, each of which receives the reward for one transition as follows:

$$\begin{aligned}\alpha_{[Stay]}(A) &= 0.9R(A, Stay, A) + 0.1R(A, Stay, B) = 0.1 \\ \alpha_{[Stay]}(B) &= 0.1R(B, Stay, A) + 0.9R(B, Stay, B) = 0.9 \\ \alpha_{[Go]}(A) &= 0.1R(A, Go, A) + 0.9R(A, Go, B) = 0.9 \\ \alpha_{[Go]}(B) &= 0.9R(B, Go, A) + 0.1R(B, Go, B) = 0.1\end{aligned}$$

The hyperplanes (lines, in this case) for $b \cdot \alpha_{[Stay]}$ and $b \cdot \alpha_{[Go]}$ are shown in Figure 16.15(a) and their maximum is shown in bold. The bold line therefore represents the utility function for the finite-horizon problem that allows just one action, and in each “piece” of the piecewise linear utility function an optimal action is the first action of the corresponding conditional plan. In this case, the optimal one-step policy is to *Stay* when $b(B) > 0.5$ and *Go* otherwise.

Once we have utilities $\alpha_p(s)$ for all the conditional plans p of depth 1 in each physical state s , we can compute the utilities for conditional plans of depth 2 by considering each possible first action, each possible subsequent percept, and then each way of choosing a depth-1 plan to execute for each percept:

$[Stay; \text{ if } Percept=A \text{ then } Stay \text{ else } Stay]$
 $[Stay; \text{ if } Percept=A \text{ then } Stay \text{ else } Go]$
 $[Go; \text{ if } Percept=A \text{ then } Stay \text{ else } Stay]$
 \dots

There are eight distinct depth-2 plans in all, and their utilities are shown in Figure 16.15(b). Notice that four of the plans, shown as dashed lines, are suboptimal across the entire belief space—we say these plans are **dominated**, and they need not be considered further. There are four undominated plans, each of which is optimal in a specific region, as shown in Figure 16.15(c). The regions partition the belief-state space.

Dominated plan

We repeat the process for depth 3, and so on. In general, let p be a depth- d conditional plan whose initial action is a and whose depth- $(d-1)$ subplan for percept e is $p.e$; then

$$\alpha_p(s) = \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma \sum_e P(e | s') \alpha_{p.e}(s')]. \quad (16.18)$$

This recursion naturally gives us a value iteration algorithm, which is given in Figure 16.16. The structure of the algorithm and its error analysis are similar to those of the basic value iteration algorithm in Figure 16.6 on page 563; the main difference is that instead of computing one utility number for each state, POMDP-VALUE-ITERATION maintains a collection of undominated plans with their utility hyperplanes.

The algorithm’s complexity depends primarily on how many plans get generated. Given $|A|$ actions and $|E|$ possible observations, there are $|A|^{O(|E|^{d-1})}$ distinct depth- d plans. Even for the lowly two-state world with $d=8$, that’s 2^{255} plans. The elimination of dominated plans is essential for reducing this doubly exponential growth: the number of undominated plans with $d=8$ is just 144. The utility function for these 144 plans is shown in Figure 16.15(d).

Notice that the intermediate belief states have lower value than state A and state B , because in the intermediate states the agent lacks the information needed to choose a good action. This is why information has value in the sense defined in Section 15.6 and optimal policies in POMDPs often include information-gathering actions.

```

function POMDP-VALUE-ITERATION(pomdp,  $\epsilon$ ) returns a utility function
  inputs: pomdp, a POMDP with states  $S$ , actions  $A(s)$ , transition model  $P(s' | s, a)$ ,
           sensor model  $P(e | s)$ , rewards  $R(s, a, s')$ , discount  $\gamma$ 
            $\epsilon$ , the maximum error allowed in the utility of any state
  local variables:  $U, U'$ , sets of plans  $p$  with associated utility vectors  $\alpha_p$ 

   $U' \leftarrow$  a set containing all one-step plans  $[a]$ , with  $\alpha_{[a]}(s) = \sum_{s'} P(s' | s, a) R(s, a, s')$ 
  repeat
     $U \leftarrow U'$ 
     $U' \leftarrow$  the set of all plans consisting of an action and, for each possible next percept,
                a plan in  $U$  with utility vectors computed according to Equation (16.18)
     $U' \leftarrow \text{REMOVE-DOMINATED-PLANS}(U')$ 
  until MAX-DIFFERENCE( $U, U'$ )  $\leq \epsilon(1 - \gamma)/\gamma$ 
  return  $U$ 

```

Figure 16.16 A high-level sketch of the value iteration algorithm for POMDPs. The REMOVE-DOMINATED-PLANS step and MAX-DIFFERENCE test are typically implemented as linear programs.

Given such a utility function, an executable policy can be extracted by looking at which hyperplane is optimal at any given belief state b and executing the first action of the corresponding plan. In Figure 16.15(d), the corresponding optimal policy is still the same as for depth-1 plans: *Stay* when $b(B) > 0.5$ and *Go* otherwise.

In practice, the value iteration algorithm in Figure 16.16 is hopelessly inefficient for larger problems—even the 4×3 POMDP is too hard. The main reason is that given n undominated conditional plans at level d , the algorithm constructs $|A| \cdot n^{|E|}$ conditional plans at level $d + 1$ before eliminating the dominated ones. With the four-bit sensor, $|E|$ is 16, and n can be in the hundreds, so this is hopeless.

Since this algorithm was developed in the 1970s, there have been several advances, including more efficient forms of value iteration and various kinds of policy iteration algorithms. Some of these are discussed in the notes at the end of the chapter. For general POMDPs, however, finding optimal policies is very difficult (PSPACE-hard, in fact—that is, very hard indeed). The next section describes a different, approximate method for solving POMDPs, one based on look-ahead search.

16.5.2 Online algorithms for POMDPs

The basic design for an online POMDP agent is straightforward: it starts with some prior belief state; it chooses an action based on some deliberation process centered on its current belief state; after acting, it receives an observation and updates its belief state using a filtering algorithm; and the process repeats.

One obvious choice for the deliberation process is the expectimax algorithm from Section 16.2.4, except with belief states rather than physical states as the decision nodes in the tree. The chance nodes in the POMDP tree have branches labeled by possible observations and leading to the next belief state, with transition probabilities given by Equation (16.17). A fragment of the belief-state expectimax tree for the 4×3 POMDP is shown in Figure 16.17.

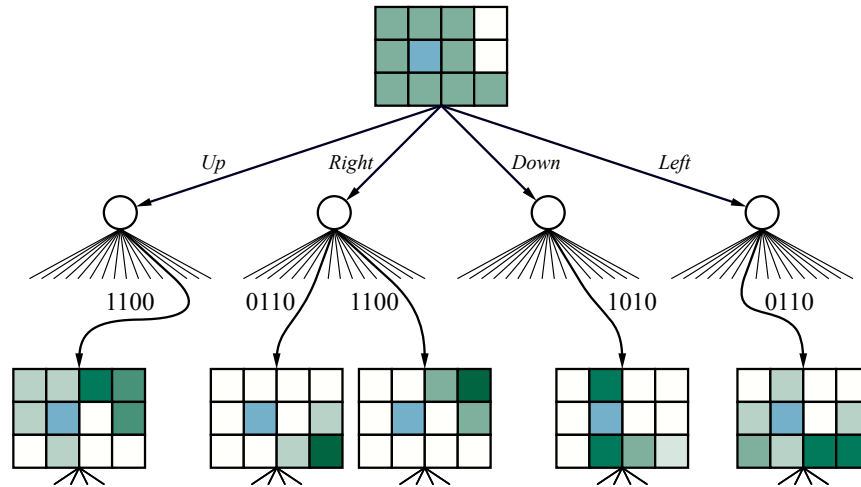


Figure 16.17 Part of an expectimax tree for the 4×3 POMDP with a uniform initial belief state. The belief states are depicted with shading proportional to the probability of being in each location.

The time complexity of an exhaustive search to depth d is $O(|A|^d \cdot |E|^d)$, where $|A|$ is the number of available actions and $|E|$ is the number of possible percepts. (Notice that this is far less than the number of possible depth- d conditional plans generated by value iteration.) As in the observable case, sampling at the chance nodes is a good way to cut down the branching factor without losing too much accuracy in the final decision. Thus, the complexity of approximate online decision making in POMDPs may not be drastically worse than that in MDPs.

For very large state spaces, exact filtering is infeasible, so the agent will need to run an approximate filtering algorithm such as particle filtering (see page 510). Then the belief states in the expectimax tree become collections of particles rather than exact probability distributions. For problems with long horizons, we may also need to run the kind of long-range playouts used in the UCT algorithm (Figure 6.11). The combination of particle filtering and UCT applied to POMDPs goes under the name of partially observable Monte Carlo planning or **POMCP**. With a DDN representation for the model, the POMCP algorithm is, at least in principle, applicable to very large and realistic POMDPs. Details of the algorithm are explored in Exercise 16.POMC. POMCP is capable of generating competent behavior in the 4×3 POMDP. A short (and somewhat fortunate) example is shown in Figure 16.18.

POMDP agents based on dynamic decision networks and online decision making have a number of advantages compared with other, simpler agent designs presented in earlier chapters. In particular, they handle partially observable, stochastic environments and can easily revise their “plans” to handle unexpected evidence. With appropriate sensor models, they can handle sensor failure and can plan to gather information. They exhibit “graceful degradation” under time pressure and in complex environments, using various approximation techniques.

So what is missing? The principal obstacle to real-world deployment of such agents is the inability to generate successful behavior over long time-scales. Random or near-random playouts have no hope of gaining any positive reward on, say, the task of laying the table

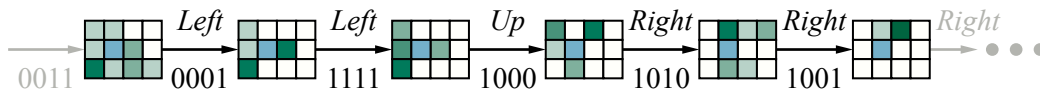


Figure 16.18 A sequence of percepts, belief states, and actions in the 4×3 POMDP with a wall-sensing error of $\epsilon=0.2$. Notice how the early *Left* moves are safe—they are very unlikely to fall into $(4,2)$ —and coerce the agent’s location into a small number of possible locations. After moving *Up*, the agent thinks it is probably in $(3,3)$, but possibly in $(1,3)$. Fortunately, moving *Right* is a good idea in both cases, so it moves *Right*, finds out that it had been in $(1,3)$ and is now in $(2,3)$, and then continues moving *Right* and reaches the goal.

for dinner, which might take tens of millions of motor-control actions. It seems necessary to borrow some of the hierarchical planning ideas described in Section 11.4. At the time of writing, there are not yet satisfactory and efficient ways to apply these ideas in stochastic, partially observable environments.

Summary

This chapter shows how to use knowledge about the world to make decisions even when the outcomes of an action are uncertain and the rewards for acting might not be reaped until many actions have passed. The main points are as follows:

- Sequential decision problems in stochastic environments, also called **Markov decision processes**, or MDPs, are defined by a **transition model** specifying the probabilistic outcomes of actions and a **reward function** specifying the reward in each state.
- The utility of a state sequence is the sum of all the rewards over the sequence, possibly discounted over time. The solution of an MDP is a **policy** that associates a decision with every state that the agent might reach. An optimal policy maximizes the utility of the state sequences encountered when it is executed.
- The utility of a state is the expected sum of rewards when an optimal policy is executed from that state. The **value iteration** algorithm iteratively solves a set of equations relating the utility of each state to those of its neighbors.
- **Policy iteration** alternates between calculating the utilities of states under the current policy and improving the current policy with respect to the current utilities.
- Partially observable MDPs, or POMDPs, are much more difficult to solve than are MDPs. They can be solved by conversion to an MDP in the continuous space of belief states; both value iteration and policy iteration algorithms have been devised. Optimal behavior in POMDPs includes information gathering to reduce uncertainty and therefore make better decisions in the future.
- A decision-theoretic agent can be constructed for POMDP environments. The agent uses a **dynamic decision network** to represent the transition and sensor models, to update its belief state, and to project forward possible action sequences.

We shall return MDPs and POMDPs in Chapter 23, which covers **reinforcement learning** methods that allow an agent to improve its behavior from experience.

Bibliographical and Historical Notes

Richard Bellman developed the ideas underlying the modern approach to sequential decision problems while working at the RAND Corporation beginning in 1949. According to his autobiography (Bellman, 1984), he coined the term “dynamic programming” to hide from a research-phobic Secretary of Defense, Charles Wilson, the fact that his group was doing mathematics. (This cannot be strictly true, because his first paper using the term (Bellman, 1952) appeared before Wilson became Secretary of Defense in 1953.) Bellman’s book, *Dynamic Programming* (1957), gave the new field a solid foundation and introduced the value iteration algorithm.

Shapley (1953b) actually described the value iteration algorithm independently of Bellman, but his results were not widely appreciated in the operations research community, perhaps because they were presented in the more general context of Markov games. Although the original formulations included discounting, its analysis in terms of stationary preferences was suggested by Koopmans (1972). The shaping theorem is due to Ng *et al.* (1999).

Ron Howard’s Ph.D. thesis (1960) introduced policy iteration and the idea of average reward for solving infinite-horizon problems. Several additional results were introduced by Bellman and Dreyfus (1962). The use of contraction mappings in analyzing dynamic programming algorithms is due to Denardo (1967). Modified policy iteration is due to van Nunen (1976) and Puterman and Shin (1978). Asynchronous policy iteration was analyzed by Williams and Baird (1993), who also proved the policy loss bound in Equation (16.13). The general family of **prioritized sweeping** algorithms aims to speed up convergence to optimal policies by heuristically ordering the value and policy update calculations (Moore and Atkeson, 1993; Andre *et al.*, 1998; Wingate and Seppi, 2005).

The formulation of MDP-solving as a linear program is due to de Ghellinck (1960), Manne (1960), and D’Épenoux (1963). Although linear programming has traditionally been considered inferior to dynamic programming as an exact solution method for MDPs, de Farias and Roy (2003) show that it is possible to use linear programming and a linear representation of the utility function to obtain provably good approximate solutions to very large MDPs. Papadimitriou and Tsitsiklis (1987) and Littman *et al.* (1995) provide general results on the computational complexity of MDPs. Yinyu Ye (2011) analyzes the relationship between policy iteration and the simplex method for linear programming and proves that for fixed γ , the runtime of policy iteration is polynomial in the number of states and actions.

Seminal work by Sutton (1988) and Watkins (1989) on reinforcement learning methods for solving MDPs played a significant role in introducing MDPs into the AI community. (Earlier work by Werbos (1977) contained many similar ideas, but was not taken up to the same extent.) AI researchers have pushed MDPs in the direction of more expressive representations that can accommodate much larger problems than the traditional atomic representations based on transition matrices.

The basic ideas for an agent architecture using dynamic decision networks were proposed by Dean and Kanazawa (1989a). Tatman and Shachter (1990) showed how to apply dynamic programming algorithms to DDN models. Several authors made the connection between MDPs and AI planning problems, developing probabilistic forms of the compact STRIPS representation for transition models (Wellman, 1990b; Koenig, 1991). The book *Planning and Control* by Dean and Wellman (1991) explores the connection in great depth.

Later work on **factored MDPs** (Boutilier *et al.*, 2000; Koller and Parr, 2000; Guestrin *et al.*, 2003b) uses structured representations of the value function as well as the transition model, with provable improvements in complexity. **Relational MDPs** (Boutilier *et al.*, 2001; Guestrin *et al.*, 2003a) go one step further, using structured representations to handle domains with many related objects. Open-universe MDPs and POMDPs (Srivastava *et al.*, 2014b) also allow for uncertainty over the existence and identity of objects and actions.

Factored MDP

Relational MDP

Many authors have developed approximate online algorithms for decision making in MDPs, often borrowing explicitly from earlier AI approaches to real-time search and game-playing (Werbos, 1992; Dean *et al.*, 1993; Tash and Russell, 1994). The work of Barto *et al.* (1995) on RTDP (real-time dynamic programming) provided a general framework for understanding such algorithms and their connection to reinforcement learning and heuristic search. The analysis of depth-bounded expectimax with sampling at chance nodes is due to Kearns *et al.* (2002). The UCT algorithm described in the chapter is due to Kocsis and Szepesvari (2006) and borrows from earlier work on random playouts for estimating the values of states (Abramson, 1990; Brügmann, 1993; Chang *et al.*, 2005).

Bandit problems were introduced by Thompson (1933) but came to prominence after World War II through the work of Herbert Robbins (1952). Bradt *et al.* (1956) proved the first results concerning stopping rules for one-armed bandits, which led eventually to the breakthrough results of John Gittins (Gittins and Jones, 1974; Gittins, 1989). Katehakis and Veinott (1987) suggested the restart MDP as a method of computing Gittins indices. The text by Berry and Fristedt (1985) covers many variations on the basic problem, while the pellucid online text by Ferguson (2001) connects bandit problems with stopping problems.

Lai and Robbins (1985) initiated the study of the asymptotic regret of optimal bandit policies. The UCB heuristic was introduced and analyzed by Auer *et al.* (2002). Bandit superprocesses (BSPs) were first studied by Nash (1973) but have remained largely unknown in AI. Hadfield-Menell and Russell (2015) describe an efficient branch-and-bound algorithm capable of solving relatively large BSPs. Selection problems were introduced by Bechhofer (1954). Hay *et al.* (2012) developed a formal framework for metareasoning problems, showing that simple instances mapped to selection rather than bandit problems. They also proved the satisfying result that expected computation cost of the optimal computational strategy is never higher than the expected gain in decision quality—although there are cases where the optimal policy may, with some probability, keep computing long past the point where any possible gain has been used up.

The observation that a partially observable MDP can be transformed into a regular MDP over belief states is due to Astrom (1965) and Aoki (1965). The first complete algorithm for the exact solution of POMDPs—essentially the value iteration algorithm presented in this chapter—was proposed by Edward Sondik (1971) in his Ph.D. thesis. (A later journal paper by Smallwood and Sondik (1973) contains some errors, but is more accessible.) Lovejoy (1991) surveyed the first twenty-five years of POMDP research, reaching somewhat pessimistic conclusions about the feasibility of solving large problems.

The first significant contribution within AI was the Witness algorithm (Cassandra *et al.*, 1994; Kaelbling *et al.*, 1998), an improved version of POMDP value iteration. Other algorithms soon followed, including an approach due to Hansen (1998) that constructs a policy incrementally in the form of a finite-state automaton whose states define the possible belief states of the agent.

More recent work in AI has focused on **point-based** value iteration methods that, at each iteration, generate conditional plans and α -vectors for a finite set of belief states rather than for the entire belief space. Lovejoy (1991) proposed such an algorithm for a fixed grid of points, an approach taken also by Bonet (2002). An influential paper by Pineau *et al.* (2003) suggested generating reachable points by simulating trajectories in a somewhat greedy fashion; Spaan and Vlassis (2005) observe that one need generate plans for only a small, randomly selected subset of points to improve on the plans from the previous iteration for all points in the set. Shani *et al.* (2013) survey these and other developments in point-based algorithms, which have led to good solutions for problems with thousands of states. Because POMDPs are PSPACE-hard (Papadimitriou and Tsitsiklis, 1987), further progress on offline solution methods may require taking advantage of various kinds of structure in value functions arising from a factored representation of the model.

The online approach for POMDPs—using look-ahead search to select an action for the current belief state—was first examined by Satia and Lave (1973). The use of sampling at chance nodes was explored analytically by Kearns *et al.* (2000) and Ng and Jordan (2000). The POMCP algorithm is due to Silver and Veness (2011).

With the development of reasonably effective approximation algorithms for POMDPs, their use as models for real-world problems has increased, particularly in education (Rafferty *et al.*, 2016), dialog systems (Young *et al.*, 2013), robotics (Hsiao *et al.*, 2007; Huynh and Roy, 2009), and self-driving cars (Forbes *et al.*, 1995; Bai *et al.*, 2015). An important large-scale application is the Airborne Collision Avoidance System X (ACAS X), which keeps airplanes and drones from colliding midair. The system uses POMDPs with neural networks to do function approximation. ACAS X significantly improves safety compared to the legacy TCAS system, which was built in the 1970s using expert system technology (Kochenderfer, 2015; Julian *et al.*, 2018).

Complex decision making has also been studied by economists and psychologists. They find that decision makers are not always rational, and may not be operating exactly as described by the models in this chapter. For example, when given a choice, a majority of people prefer \$100 today over a guarantee of \$200 in two years, but those same people prefer \$200 in eight years over \$100 in six years. One way to interpret this result is that people are not using additive exponentially discounted rewards; perhaps they are using **hyperbolic rewards** (the hyperbolic function dips more steeply in the near term than does the exponential decay function). This and other possible interpretations are discussed by Rubinstein (2003).

The texts by Bertsekas (1987) and Puterman (1994) provide rigorous introductions to sequential decision problems and dynamic programming. Bertsekas and Tsitsiklis (1996) include coverage of reinforcement learning. Sutton and Barto (2018) cover similar ground but in a more accessible style. Sigaud and Buffet (2010), Mausam and Kolobov (2012) and Kochenderfer (2015) cover sequential decision making from an AI perspective. Krishnamurthy (2016) provides thorough coverage of POMDPs.

Hyperbolic reward