17 MAKING COMPLEX DECISIONS

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

SEQUENTIAL DECISION PROBLEMS

In this chapter, we address the computational issues involved in making decisions. Whereas Chapter 16 was concerned with one-shot or episodic decision problems, in which the utility of each action's outcome was well known, we will be concerned here with **sequential decision problems,** in which the agent's utility depends on a sequence of decisions. Sequential decision problems, which include utilities, uncertainty, and sensing, generalize the search and planning problems described in Parts II and IV. Section 17.1 explains how sequential decision problems are defined, and Sections 17.2 and 17.3 explain how they can be solved to produce optimal behavior that balances the risks and rewards of acting in an uncertain environment. Section 17.4 extends these ideas to the case of partially observable environments, and Section 17.5 develops a complete design for decision-theoretic agents in partially observable environments, combining dynamic Bayesian networks from Chapter 15 with decision networks from Chapter 16.

The second part of the chapter covers environments with multiple agents. In such environments, the notion of optimal behavior becomes much more complicated by the interactions among the agents. Section 17.6 introduces the main ideas of **game theory,** including the idea that rational agents might need to behave randomly. Section 17.7 looks at how multiagent systems can be designed so that multiple agents can achieve a common goal.

17.1 SEQUENTIAL DECISION PROBLEMS

An example

Suppose that an agent is situated in the 4 x **3** environment shown in Figure 17.1(a). Beginning in the start state, it must choose an action at each time step. The interaction with the environment terminates when the agent reaches one of the goal states, marked +1 or −1. In each location, the available actions are called Up, *Down, Left,* and *Right.* We will assume for now that the environment is **fully observable,** so that the agent always knows where it is.

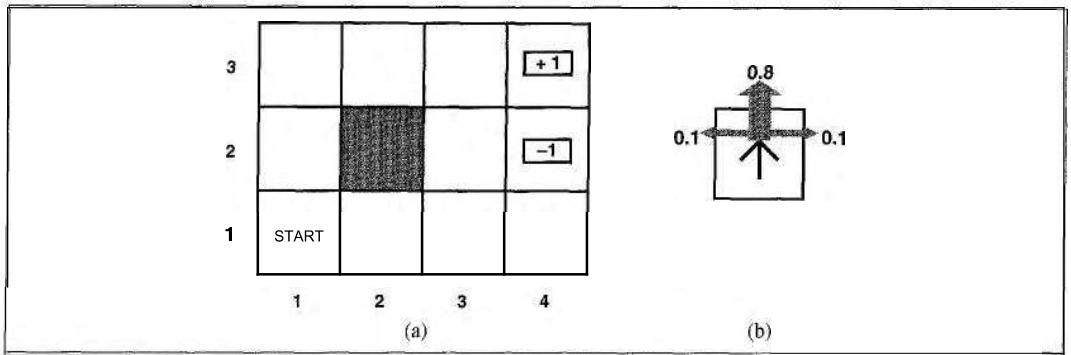

Figure 17.1 (a) A simple 4 x **3** environment that presents the agent with a sequential decision problem. (b) Illustration of the transition model of the environment: the "intended" outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction.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

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

TRANSITION MODEL

A specification of the outcome probabilities for each action in each possible state is called a transition model (or just "model," whenever no confusion can arise). We will use $T(s, a, s')$ to denote the probability of reaching state s' if action a is done in state s. We will assume that transitions are Markovian in the sense of Chapter 15, that is, the probability of reaching s' from s depends only on s and not on the history of earlier states. For now, you can think of $T(s, a, s')$ as a big three-dimensional table containing probabilities. Later, in Section 17.5, we will see that the transition model can be represented as a dynamic Bayesian network, just as in Chapter 15.

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

REWARD

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

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

MARKOV DECISION PROCESS

The specification of a sequential decision problem for a fully observable environment with a Markovian transition model and additive rewards is called a Markov decision process, or MDP. An MDP is defined by the following three components:

Initial State: S_0

Transition Model: $T(s, a, s')$

Reward Function:[1] $R(s)$

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

POLICY

Each time a given policy is executed starting from the initial state, the stochastic nature of the environment will 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.

OPTIMAL POLICY

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

The balance of risk and reward changes depends on the value of $R(s)$ for the nonterminal states. Figure 17.2(b) shows optimal policies for four different ranges of $R(s)$. When $R(s) \leq -1.6284$, life is so painful that the agent heads straight for the nearest exit, even if the exit is worth −1. When $-0.4278 \leq R(s) \leq -0.0850$, life is quite unpleasant; the agent takes the shortest route to the +1 state and is willing to risk falling into the −1 state by accident. In particular, the agent takes the shortcut from (3,1). When life is only slightly dreary ($-0.0221 < R(s) < 0$), the optimal policy takes no *risks* ***at all.*** In (4,1) and (3,2), the agent

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

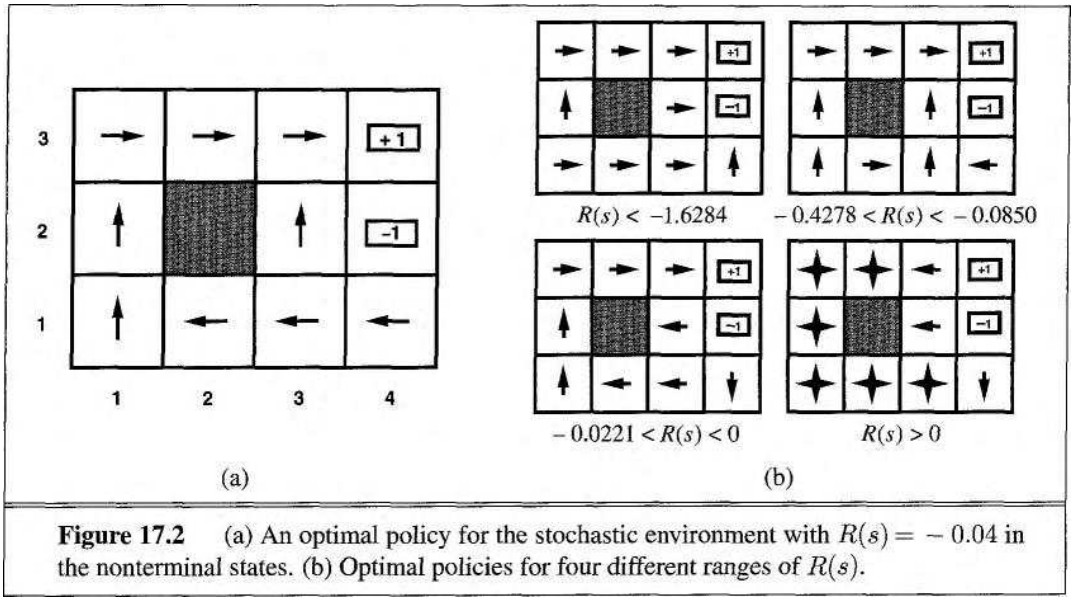

Figure 17.2 (a) An optimal policy for the stochastic environment with $R(s) = -0.04$ in the nonterminal states. (b) Optimal policies for four different ranges of $R(s)$.

heads directly away from the –1 state so that it cannot fall in by accident, even though this means banging its head against the wall quite a few times. Finally, if $R(s) > 0$, then life is positively enjoyable and the agent avoids *both* exits. As long as the actions in (4,1), (3,2), and (3,3) are as shown, every policy is optimal, and the agent obtains infinite total reward because it never enters a terminal state. Surprisingly, it turns out that there are six other optimal policies for various ranges of $R(s)$; Exercise 17.7 asks you to find them.

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

Optimality in sequential decision problems

In the MDP example in Figure 17.1, the performance of the agent was measured by a sum of rewards for the states visited. This choice of performance measure is not arbitrary, but it is not the only possibility. This section investigates the possible choices for the performance measure—that is, choices for the utility function on environment histories, which we will write as $U_h([s_0, s_1, \ldots, s_n])$. The section draws on ideas from Chapter 16 and is somewhat technical; the main points are summarized at the end.

FINITE HORIZON
INFINITE HORIZON

The first question to answer is whether there is a **finite horizon** or an **infinite horizon** for decision making. A finite horizon means that there is a *fixed* time N after which nothing matters—the game is over, so to speak. Thus, $U_h([s_0, s_1, \ldots, s_{N+k}]) = U_h([s_0, s_1, \ldots, s_N])$ for all $k > 0$. For example, suppose an agent starts at (3,1) in the 4 x **3** world of Figure 17.1,

NONSTATIONARY POLICY

STATIONARY POLICY

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, the optimal action in a given state could change over time. We* say that the optimal policy for a finite horizon is **nonstationary.** With no fixed time limit, on the other hand, there is no reason to behave differently in the same state at different times. Hence, the optimal action depends only on the current state, and the optimal policy is **stationary.** Policies for the infinite-horizon case are therefore simpler than those for the finite-horizon case, and we will deal mainly with the infinite-horizon case in this chapter.[2] Note that "infinite horizon" does not necessarily mean that all state sequences are infinite; it just means that there is no fixed deadline. In particular, there can be finite state sequences in an infinite-horizon MDP containing a terminal state.

The next question we must decide is how to calculate the utility of state sequences. We can view this as a question in **multiattribute utility theory** (see Section 16.4), where each state s_i is viewed as an attribute of the state sequence $[s_0, s_1, s_2 \ldots]$. To obtain a simple expression in terms of the attributes, we will need to make some sort of preference independence assumption. The most natural assumption is that the agent's preferences between state sequences are **stationary.** Stationarity for preferences means the following: if two state sequences $[s_0, s_1, s_2, \ldots]$ and $[s'_0, s'_1, s'_2, \ldots]$ begin with the same state (i.e., $s_0 = s'_0$) then the two sequences should be preference-ordered the same way as the sequences $[s_1, s_2, \ldots]$ and $[s'_1, s'_2, \ldots]$. In English, this means that if you prefer one future to another starting tomorrow, then you should still prefer that future if it were to start today. Stationarity is a fairly innocuous-looking assumption with very strong consequences: it turns out that under stationarity there are just two ways to assign utilities to sequences:

STATIONARY PREFERENCE

ADDITIVE REWARDS

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

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

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

DISCOUNTED REWARDS

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

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

DISCOUNT FACTOR

where the **discount factory** 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 y is 1, discounted rewards are exactly equivalent to additive rewards, so additive rewards are a special case of discounted rewards. Discounting appears to be a good model of both animal and human preferences over time. A discount factor of γ is equivalent to an interest rate of $(1/\gamma) - 1$.

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

[2] This is for completely observable environments. We will see later that for partially observable environments, the infinite-horizon case is not so simple.

Lurking beneath our choice of infinite horizons is a problem: if the environment does not contain a terminal state, or if the agent never reaches one, then all environment histories will be infinitely long, and utilities with additive rewards will generally be infinite. Now, we can agree that $+\infty$ is better than $-\infty$, but comparing two state sequences, both having $+\infty$ 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 rewards are bounded by $R_{\max}$ and $\gamma < 1$, we have
$$U_h([s_0, s_1, s_2, \ldots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = R_{\max}/(1-\gamma)\,, \qquad (17.1)$$
using the standard formula for the sum of an infinite geometric series.
2. If the environment contains terminal states *and if the agent is guaranteed to get to one eventually,* then we will never need to compare infinite sequences. A policy that is guaranteed to reach a terminal state is called a proper policy. With proper policies, we can use $\gamma = 1$ (i.e., additive rewards). The first three policies shown in Figure 17.2(b) are proper, but the fourth is improper. It gains infinite total reward by staying away from the terminal states when the reward for the nonterminal states is positive. The existence of improper policies can cause the standard algorithms for solving MDPs to fail with additive rewards, and so provides a good reason for using discounted rewards.

PROPER POLICY

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

AVERAGE REWARD

In sum, the use of discounted rewards presents the fewest difficulties in evaluating state sequences. The final step is to show how to choose between policies, bearing in mind that a given policy π generates not one state sequence, but a whole range of possible state sequences, each with a specific probability determined by the transition model for the environment. Thus, the value of a policy is the *expected* sum of discounted rewards obtained, where the expectation is taken over all possible state sequences that could occur, given that the policy is executed. An optimal policy π^* satisfies
$$\pi^* = \operatorname*{argmax}_{\pi} E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi\right]\,. \qquad (17.2)$$
The next two sections describe algorithms for finding optimal policies.

17.2 VALUE ITERATION

VALUE ITERATION

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

Utilities of states

The utility of states is defined in terms of the utility of state sequences. Roughly speaking, the utility of a state is the expected utility of the state sequences that might follow it. Obviously, the state sequences depend on the policy that is executed, so we begin by defining the utility $U^{\pi}(s)$ with respect to a specific policy π. If we let s_t be the state the agent is in after executing π for t steps (note that s_t is a random variable), then we have

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s\right] \tag{17.3}$$

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

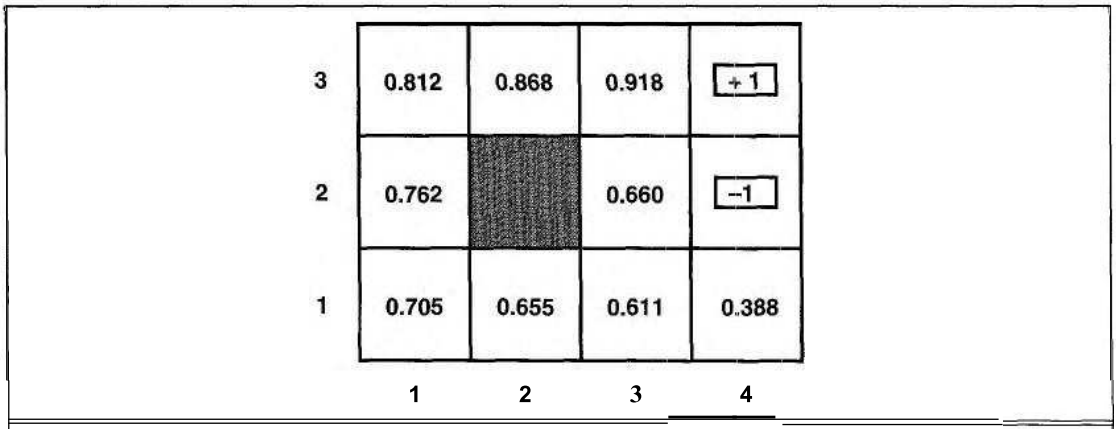

Figure 17.3 The utilities of the states in the 4 x **3** worlld, calculated with $\gamma = 1$ and $R(s) = -0.04$ for nonterminal states.

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

$$\pi^*(s) = \operatorname*{argmax}_a \sum_{s'} T(s, a, s') U(s') \,. \tag{17.4}$$

Now, if the utility of a state is the expected sum of discounted rewards from that point onwards, then there is a direct relationship between the utility of a state and the utility of its neighbors: *the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.* That is, the utility of a state is given by

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U(s') \,. \tag{17.5}$$

BELLMAN EQUATION

Equation (17.5) is called the **Bellman** equation, after Richard Bellman (1957). The utilities of the states—defined by Equation (17.3) 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 the next two sections.

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

$$U(1,1) = -0.04 + \gamma \max\{ \begin{array}{ll} 0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), & (Up) \\ 0.9U(1,1) + 0.1U(1,2), & (Left) \\ 0.9U(1,1) + 0.1U(2,1), & (Down) \\ 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1) \} & (Right) \end{array}$$

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

The value iteration algorithm

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

BELLMAN UPDATE

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{s'} \sum T(s,a,s')U_i(s') \,. \tag{17.6}$$

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

We can apply value iteration to the 4 x **3** world in Figure 17.1(a). Starting with initial values of zero, the utilities evolve as shown in Figure 17.5(a). Notice how the states at different distances from (4,3) accumulate negative reward until, at some point, 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.

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 VALUE-ITERATION(mdp, ϵ) returns a utility function
  inputs: mdp, an MDP with states S, transition model T, reward function R, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    6, the maximum change in the utility of any state in an iteration

repeat
  $U \leftarrow U'$; $\delta \leftarrow 0$
  for each state s in S do
    $U'[s] \leftarrow R[s] + \gamma \max_a \sum_{s'} T(s, a, s') U[s']$
    if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
until $6 < \epsilon(1-\gamma)/\gamma$
return U

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

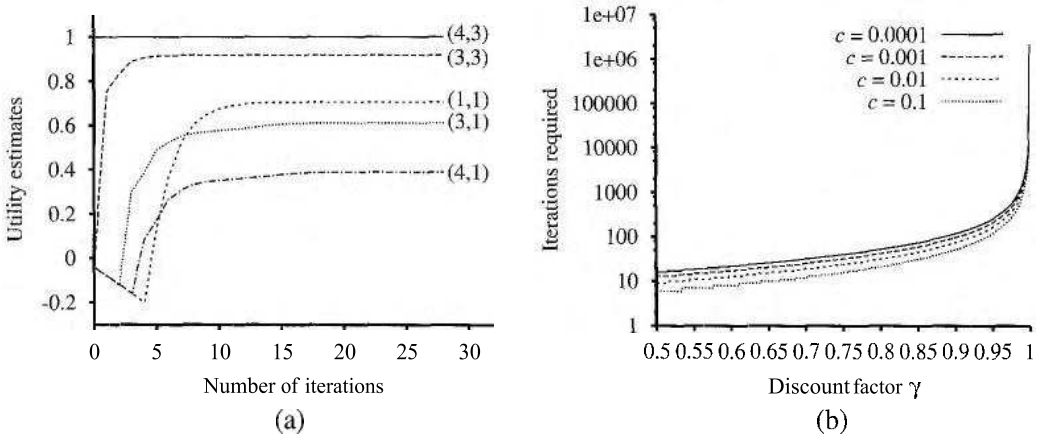

Figure 17.5 (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations k required to guarantee an error of at most $\epsilon = c \cdot R_{\max}$, for different values of c, as a function of the discount factor γ.

function returned when the algorithm is terminated early; this is useful because it means that we don't have to run forever. The 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 amount, than the original arguments. 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 un-

changed by the application of the function. From this example, we can discern two important properties of contractions:

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

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

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

MAX NORM

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

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

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

$$||B\,U_i - B\,U_i'|| \leq \gamma\,||U_i - U_i'||\ . \tag{17.7}$$

That is, the Bellman update is a contraction by a factor of γ on the space of utility vectors. Hence, value iteration always converges to a unique solution of the Bellman equations.

In particular, we can replace U_i' in Equation (17.7) with the *true* utilities U, for which $B\,U = U$. Then we obtain the inequality

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

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

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

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

The error bound in the preceding paragraph gives some idea of the factors influencing the runtime of the algorithm, but is sometimes overly conservative as a method of deciding when to stop the iteration. For the latter purpose, we can use a bound relating the error

to the size of the Bellman update on any given iteration. From the contraction property (Equation (17.7)), it can be shown that if the update is small (i.e., no state's utility changes by much), then the error, compared with the true utility function, also is small. More precisely,

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

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

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

POLICY LOSS

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

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

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

Now we have everything we need to use value iteration in practice. We know that it converges to the correct utilities, we can bound the error in the utility estimates if we stop after a finite number of iterations, and we can bound the policy loss that results from executing the corresponding MEU policy. As a final note, all of the results in this section depend on discounting with $\gamma < 1$. If $\gamma = 1$ and the environment contains terminal states, then a similar set of convergence results and error bounds can be derived whenever certain technical conditions are satisfied.

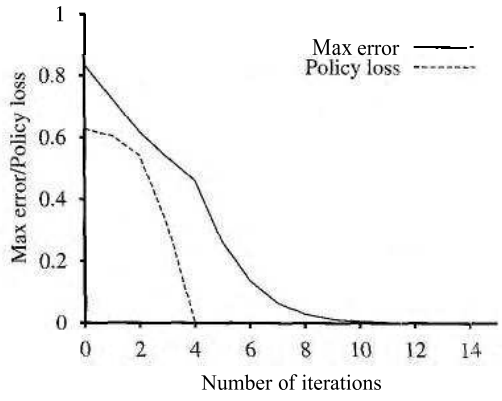

Figure 17.6 The maximum error $||U_i - U||$ of the utility estimates and the policy loss $||U^{\pi_i} - U||$ compared with the optimal policy, as a function of the number of iterations of value iteration.

17.3 POLICY ITERATION

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

POLICY ITERATION

POLICY EVALUATION

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

POLICY IMPROVEMENT

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

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

The policy improvement step is obviously straightforward, but how do we implement the POLICY-EVALUATION routine? It turns out that doing so is much simpler than solving the standard Bellman equations (which is what value iteration does), because the action in each state is fixed by the policy. At the ith iteration, the policy π_i specifies the action $\pi_i(s)$ in state s. This means that we have a simplified version of the Bellman equation (17.5) relating

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

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s **in** S **do**
      if $\max_a \sum_{s'} T(s,a,s')\, U[s'] > \sum_{s'} T(s,\pi[s],s')\, U[s']$ **then**
        $\pi[s] \leftarrow \text{argmax}_a \sum_{s'} T(s,a,s')\, U[s']$
        $unchanged? \leftarrow$ false
  until $unchanged?$
  return π

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

the utility of s (under π_i) to the utilities of its neighbors:

$$U_i(s) = R(s) + \gamma \sum_{s'} T(s, \pi_i(s), s') U_i(s') \ . \tag{17.10}$$

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

$$U_i(1,1) = -0.04 + 0.8U_i(1,2) + 0.1U_i(1,1) + 0.1U_i(2,1) \ ,$$
$$U_i(1,2) = -0.04 + 0.8U_i(1,3) + 0.2U_i(1,2) \ ,$$

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

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

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} T(s, \pi_i(s), s') U_i(s')$$

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

MODIFIED 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 utility function, asynchronous policy iteration is guaranteed to converge to an optimal policy. The freedom to choose any states to work on means that we can design much more efficient heuristic algorithms—for example, algorithms that concentrate on updating the values of states that are likely to be reached by a good policy. This makes a lot of sense In real life: if one has no intention of throwing oneself off a cliff, one should not spend time worrying about the exact value of the resulting states.

17.4 PARTIALLY OBSERVABLE MDPS

The description of Markov decision processes in Section 17.1 assumed that the environment was **fully observable.** With this assumption, the agent always knows which state it is in. This, combined with the Markov assumption for the transition model, means that the optimal policy depends only on the current state. When the environment is only **partially observable,** the situation is, one might say, much less clear. The agent does not necessarily know which

state it is in, so it cannot execute the action $\pi(s)$ recommended for that state. Furthermore, the utility of a state s and the optimal action in s depend not just on s, but also on ***how much the agent knows*** when it is in s. For these reasons, partially observable **MDPs** (or POMDPs—pronounced "pom-dee-pees")are usually viewed as much more difficult than ordinary MDPs. We cannot avoid POMDPs, however, because the real world is one.

PARTIALLY OBSERVABLE MDP

As an example, consider again the 4 x **3** world of Figure 17.1, but now let's suppose that the agent has ***no sensors whatsoever*** and has ***no*** idea *where* ***it is.*** More precisely, let's suppose the agent's initial state is equally likely to be any of the nine nonterminal states (Figure 17.8(a)). Clearly, if the agent ***knew*** it was in (3,3), it would move ***Right;*** if it ***knew*** it was in (1,1), it would move Up; but since it could be anywhere, what should it do? One possible answer is that the agent should first act so as to reduce its uncertainty, and only then should it try heading for the +1 exit. For example, if the agent moves Left five times, then it is quite likely to be at the left wall (Figure 17.8(b)). Then, if it moves Up five times, it is quite likely to be at the top, probably in the top left corner (Figure 17.8(c)). Finally, if it moves ***Right*** five times, it has a good chance—about 77.5%--of reaching the +1 exit (Figure 17.8(d)). Continuing to move right thereafter increases its chances to 81.8%. This policy is therefore surprisingly safe, but under it, the agent is rather slow to reach the exit, and has an expected utility of only about 0.08. The optimal policy, which we will describe shortly, does much better.

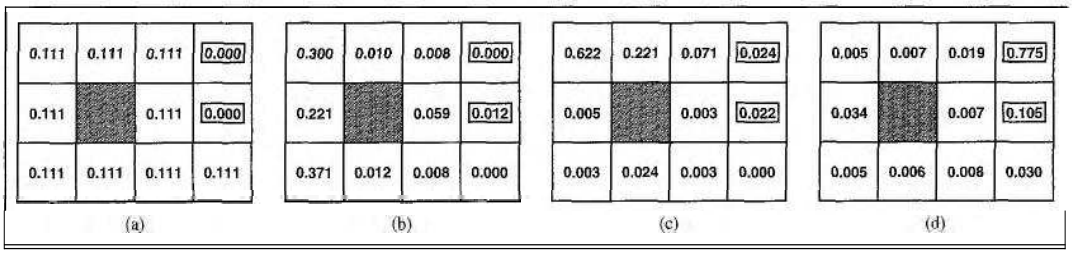

Figure 17.8 (a) The initial probability distribution for the agent's location. (b) After moving ***Left*** five times. (c) After moving Up five times. (d) After moving Right five times.

To get a handle on POMDPs, we must first define them properly. A POMDP has the same elements as an MDP—the transition model $T(s, \mathrm{a}, s')$ and the reward function $R(s)$—but it also has an observation model $O(s, o)$ that specifies the probability of perceiving the observation o in state s.[3] For example, our agent with no sensors has only one possible observation (the empty observation), and this occurs with probability 1 in every state.

OBSERVATION MODEL

In Chapters 3 and 12, 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 concept is refined somewhat. A belief state b is now a *probability* ***distribution*** over all possible states. For example, the initial belief state in Figure 17.8(a) could be written as $\langle \$, \$, \$, \$; \ \$,; \ \$, \$, 0, 0 \rangle$.

[3] The observation model is essentially identical to the **sensor model** for temporal processes, as described in Chapter 15. As with the reward function for MDPs, the observation model can also depend on the action and outcome state, but again this change is not fundamental.

We will write $b(s)$ for 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 observations and actions so far. This is essentially the **filtering** task. (See Chapter 15.) The basic recursive filtering equation (15.3 on page 543) shows how to calculate the new belief state from the previous belief state and the new observation. For POMDPs, we also have an action to consider and a slightly different notation, but the result is essentially the same. If $b(s)$ was the previous belief state, and the agent does action a and perceives observation o, then the new belief state is given by

$$b'(s') = \alpha\, O(s', \mathrm{o}) \sum_s T(s, \mathrm{a}, s') b(s) \tag{17.11}$$

where α is a normalizing constant that makes the belief state sum to 1. We can abbreviate this equation as $b' = \text{FORWARD}(b, \mathrm{a}, o)$.

The fundamental insight required to understand POMDPs is this: *the optimal action depends only on the agent's current belief state.* That is, the 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 is this:

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

Now we can think of POMDPs as requiring a search in belief state space, just like the methods for sensorless and contingency problems in Chapter 3. 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 x **3** world is a point in an 11-dimensional continuous space. An action changes the belief state, not just the physical state, so it is evaluated according to the information the agent acquires as a result. POMDPs therefore include the value of information (Section 16.6) 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 observation,* then Equation (17.11) would provide a *deterministic* update to the belief state: $b' = \text{FORWARD}(b, \mathrm{a}, \mathrm{o})$. Of course, the subsequent observation is not yet known, so the agent might arrive in one of several possible belief states b', depending on the observation that occurs. The probability of perceiving o, 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:

$$\begin{aligned} P(o|a,b) &= \sum_{s'} P(o|a,s',b) P(s'|a,b) \\ &= \sum_{s'} O(s',o) P(s'|a,b) \\ &= \sum O(s',o) \sum T(s,a,s') b(s) \,. \end{aligned}$$

Let us write the probability of reaching b' from b, given action a, as $\tau(b,a,b')$. Then that gives us

$$\begin{aligned}\tau(b,a,b') &= P(b'|a,b) = \sum P(b'|o,a,b)P(o|a,b) \\ &= \sum_o P(b'|o,a,b) \sum_{s'} O(s',o) \sum_s T(s,a,s')b(s) , \end{aligned} \tag{17.12}$$

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

Equation (17.12) can be viewed as defining a transition model for the belief state space. We can also define a reward function for belief states (i.e., the expected reward for the actual states the agent might be in):

$$\rho(b) = \sum_s b(s)R(s) .$$

So it seems that $\tau(b,a,b')$ and $\rho(b)$ together 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 aphysical 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.

Notice that, although we have reduced POMDPs to MDPs, the MDP we obtain has a continuous (and usually high-dimensional) state space. None of the MDP algorithms described in Sections 17.2 and 17.3 applies directly to such MDPs. It turns out that we *can* develop versions of value and policy iteration that apply to continuous-state MDPs. The basic idea is that a policy $\pi(b)$ can be represented as a set of *regions* of belief state space, each of which is associated with a particular optimal action.[4] The value function associates a distinct *linear* function of b with each region. Each value- or policy-iteration step refines the boundaries of the regions and might introduce new regions.

The details of the algorithms are beyond the scope of this book, but we will report the solution for the sensorless 4 x **3** world. The optimal policy is the following:

$$[Left, Up, Up, Right, Up, Up, Right, Up, Up, Right, Up, Right, Up, Right, Up, \ldots] .$$

The policy is a sequence because this problem is *deterministic* in belief state space—there are no observations. The "trick" it embodies is to have the agent move *Left* once to ensure that it's *not* in (4,1), so that it's then fairly safe to keep moving *Up* and *Right* to reach the +1 exit. The agent reaches the +1 exit 86.6% of the time and does so much faster than the policy given earlier in the section, so its expected utility is 0.38 compared with 0.08.

For more complex POMDPs with observations, finding approximately optimal policies is very difficult (PSPACE-hard, in fact—i.e., very hard indeed). Problems with a few dozen states are often infeasible. The next section describes a different, approximate method for solving POMDPs, one based on look-ahead search.

[4] For some POMDPs, the optimal policy has infinitely many regions, so the simple list-of-regions approach fails and more ingenious methods are needed to find even an approximation.

17.5 DECISION-THEORETIC AGENTS

In this section, we outline a comprehensive approach to agent design for partially observable, stochastic environments. The basic elements of the design are already familiar:

- The transition and observation models are represented by a **dynamic Bayesian network** (as described in Chapter 15).
- The dynamic Bayesian network is extended with decision and utility nodes, as used in **decision networks** in Chapter 16. The resulting model is called a **dynamic decision network** or **DDN.**

DYNAMIC DECISION NETWORK

- A filtering algorithm is used to incorporate each new percept and action and to update the belief state representation.
- Decisions are made by projecting forward possible action sequences and choosing the best one.

The primary advantage of using a dynamic Bayesian network to represent the transition and sensor models is that it decomposes the state description into a set of random variables, much as planning algorithms use logical representations to decompose the state space used by search algorithms. The agent design is therefore a practical implementation of the **utility-based agent** sketched in Chapter 2.

Because we are using dynamic Bayesian networks, we will revert to the notation of Chapter 15, where $\mathbf{X}_t$ referred to the set of state variables for time t and $\mathbf{E}_t$ referred to the evidence variables. Thus, where we have used s_t (the: state at time t) so far in this chapter, we will now use $\mathbf{X}_t$. We will use A_t to refer to the action at time t, so the transition model $T(s, \mathrm{a}, \mathrm{s}')$ is the same as $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, A_t)$ and the observation model $O(s, \mathrm{o})$ is the same as $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$. We will use R_t to refer to the reward received at time t and U_t to refer to the utility of the state at time t. With this notation, a dynamic deersion network looks like the one shown in Figure 17.9.

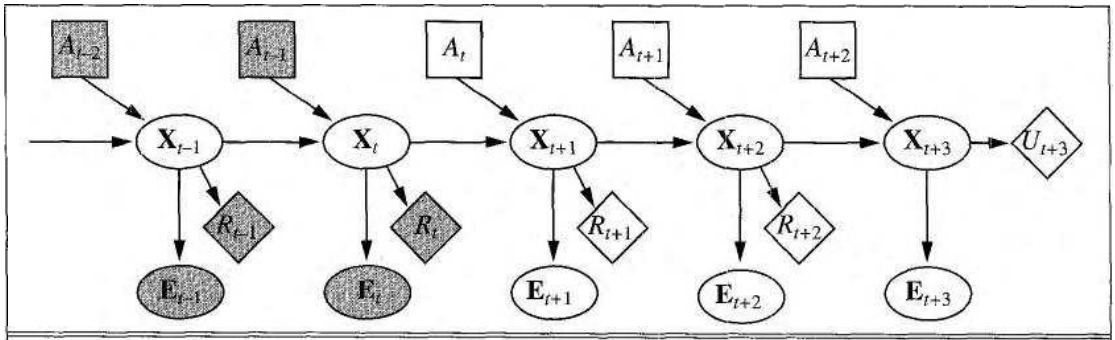

Figure 17.9 The generic structure of a dynamic decision network. Variables with known values are shaded. The current time is t and the agent must decide what to do--that is, choose a value for A_t. The network has been unrolled into the future for three steps and represents future rewards, as well as the utility of the state at the look-ahead horizon.

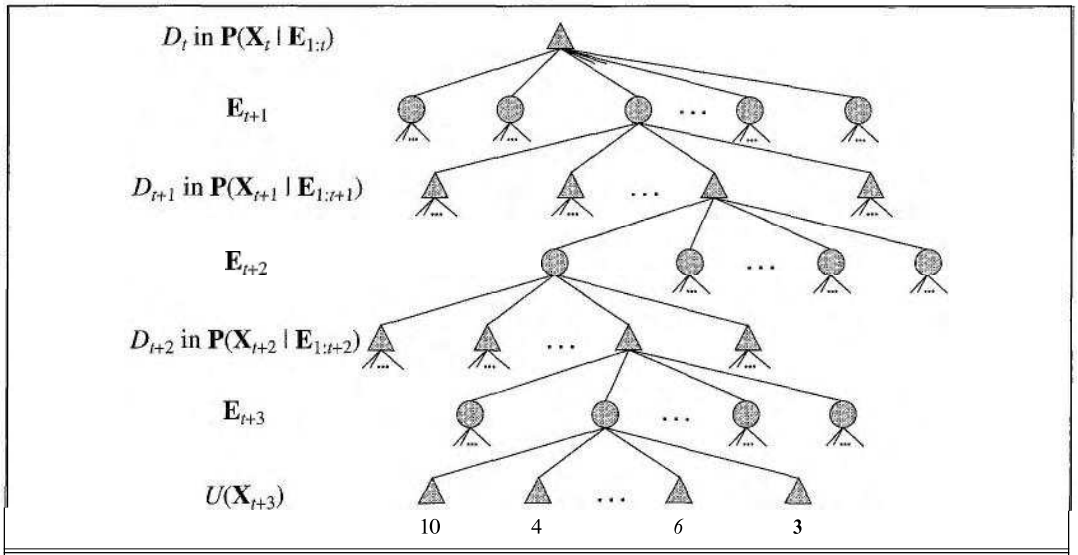

Figure 17.10 Part of the look-ahead solution of the DDN in Figure 17.9.

Dynamic decision networks provide a concise representation for large POMDPs, so they can be used as inputs for any POMDP algorithm including those for value- and policy-iteration methods. In this section, we will focus on look-ahead methods that project action sequences forward from the current belief state in much the same way as do the game-playing algorithms of Chapter 6. The network in Figure 17.9 has been projected three steps into the future; the current and future decisions and the future observations and rewards are all unknown. Notice that the network includes nodes for the *rewards* for $\mathbf{X}_{t+1}$ and $\mathbf{X}_{t+2}$, but the *utility* for $\mathbf{X}_{t+3}$. This is because the agent must maximize the (discounted) sum of all future rewards, and $U(\mathbf{X}_{t+3})$ represents the reward for $\mathbf{X}_{t+3}$ and all subsequent rewards. As in Chapter 6, we assume that U is available only in some approximate form: if exact utility values were available, there would be no need for look-ahead beyond depth 1.

Figure 17.10 shows part of the search tree corresponding to the three-step look-ahead DDN in Figure 17.9. Each of the triangular nodes is a belief state in which the agent makes a decision A_{t+i} for $i = 0, 1, 2, \ldots$. The round nodes correspond to choices by the environment, namely, what observation $\mathbf{E}_{t+i}$ occurs. Notice that there are no chance nodes corresponding to the action outcomes; this is because the belief state update for an action is deterministic regardless of the actual outcome.

The belief state at each triangular node can be computed by applying a filtering algorithm to the sequence of observations and actions leading to it. In this way, the algorithm takes into account the fact that, for decision A_{t+i}, the agent *will* have available percepts $\mathbf{E}_{t+1}, \ldots, \mathbf{E}_{t+i}$, even though at time t it does not know what those percepts will be. In this way, a decision-theoretic agent automatically takes into account the value of information and will execute information-gathering actions where appropriate.

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. This is similar to the EXPECTIMINIMAX algorithm for game trees with chance nodes, except that (1) there can also be rewards at non-leaf states and (2) the decision nodes correspond to belief states rather than actual states. The time complexity of an exhaustive search to depth d is $O(|D|^d \cdot |\mathbf{E}|^d)$, where $|D|$ is the number of available actions and E is the number of possible observations. For problems in which the discount factor γ is not too close to 1, a shallow search is often good enough to give near-optimal decisions. It is also possible to approximate the averaging step at the chance nodes, by sampling from the set of possible observations instead of summing over all possible observations. There are various other ways of finding good approximate solutions quickly, but we defer them to Chapter 21.

Decision-theoretic agents based on dynamic decision networks have a number of advantages compared with other, simpler agent designs presented in earlier chapters. In particular, they handle partially observable, uncertainty environments and can easily revise their "plans" to handle unexpected observations. 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 most important defect of our DDN-based algorithm is its reliance on forward search, just like the state-space search algorithms of Part II. In Part IV, we explained how the ability to consider partially ordered, abstract plans via goal-directed search provided a massive increase in problem-solving power, particularly when combined with plan libraries. There have been attempts to extend these methods into the probabilistic domain, but so far they have proven to be inefficient. A second, related problem is the basically propositional nature of the DDN language. We would like to be able to extend some of the ideas for first-order probabilistic languages in Section 14.6 to the problem of decision making. Current research has shown that this extension is possible and has significant benefits, as discussed in the notes at the end of the chapter.

17.6 DECISIONS WITH MULTIPLE AGENTS: GAME THEORY

This chapter has concentrated on making decisions in uncertain environments. But what if the uncertainty is due to other agents and the decisions they make? And what if the decisions of those agents are in turn influenced by our decisions? We addressed this question once before, when we studied games in Chapter 6. There, however, we were concerned with turn-talung games with perfect information, for which minimax search can be used to find optimal moves. In this section we study the aspects of **game theory** that can be used to analyze games with simultaneous moves. To simplify matters, we will look first at games that are only one move long. The word "game" and the simplification to single moves might make this seem trivial, but in fact, game theory is used in very serious decision making situations including bankruptcy proceedings, the auctioning of wireless frequency spectrums, product development and pricing decisions, and national defense, situations involving billions of dollars and hundreds of thousands of lives. Game theory can be used in at least two ways:

GAME THEORY

1. Agent design: Game theory can analyze the agent's decisions, and compute the expected utility for each decision (under the assumption that other agents are acting optimally according to game theory). For example, in the game two-finger Morra, two players, O and E, simultaneously display one or two fingers. Let the total number of fingers be f. If f is odd, O collects f dollars from E, and if f is even, E collects f dollars from O. Game theory can determine the best strategy against a rational player and the expected return for each player.[5]
2. Mechanism Design: When an environment is inhabited by many agents, it might be possible to define the rules of the environment (i.e., the game that the agents must play) so that the collective good of all agents is maximized when each agent adopts the game-theoretic solution that maximizes its own utility. For example, game theory can help design the protocols for a collection of Internet traffic routers so that each router has an incentive to act in such a way that global throughput is maximized. Mechanism design can also be used to construct intelligent multiagent systems that solve complex problems in a distributed fashion without the need for each agent to know about the whole problem being solved.

A game in game theory is defined by the following components:

PLAYERS
- Players or agents who will be making decisions. Two-player games have received the most attention, although n-player games for n > 2 are also common. We will give players capitalized names, like *Alice* and *Bob* or O and E.

ACTIONS
- Actions that the players can choose. We will give actions lowercase names, like *one* or *testify*. The players may or may not have the same set of actions available.

PAYOFF MATRIX
- A payoff matrix that gives the utility to each player for each combination of actions by all the players. The payoff matrix for two-finger Morra is as follows:

	O: one	*0 : two*
E: one	$E = 2, O = -2$	$E = -3, O = 3$
E: two	$E = -3, O = 3$	$E = 4, O = -4$

For example, the lower-right corner shows that when O chooses action *two* and E also chooses *two,* the payoff is 4 for E and *–4* for O.

Each player in a game must adopt and then execute a strategy (which is the name used in game theory for a policy). A pure strategy is a deterministic policy specifying a particular action to take in each situation; for a one-move game, a pure strategy is just a single action. The analysis of games leads to the idea of a mixed strategy, which is a randomized policy that selects particular actions according to a specific probability distribution over actions. The mixed strategy that chooses action a with probability p and action b otherwise is written $[p: a; (1 - p): b]$. For example, a mixed strategy for two-finger Morra might be $[0.5: one; 0.5: two]$. A strategy profile is an assignment of a strategy to each player; given the strategy profile, the game's outcome is a numeric value for each player.

PURE STRATEGY

MIXEDSTRATEGY

STRATEGY PROFILE

OUTCOME

[5] Morra is a recreational version of an **inspection game.** In such games, an inspector chooses a day to inspect a facility (such as a restaurant or a biological weapons plant), and the facility operator chooses a day to hide all the nasty stuff. The inspector wins if the days are different and the facility operator wins if they are the same.

SOLUTION

A **solution** to a game is a strategy profile in which each player adopts a rational strategy. We will see that the most important issue in game theory is to define what "rational" means when each agent chooses only part of the strategy profile that determines the outcome. It is important to realize that outcomes are actual results of playing a game, while solutions are theoretical constructs used to analyze a game. We will see that some games only have a solution in mixed strategies. But that does not mean that a player must literally be adopting a mixed strategy to be rational.

Consider the following story: Two alleged burglars, Alice and Bob, are caught red-handed near the scene of a burglary and are interrogated separately by the police. Both know that if they both confess to the crime, they will each serve 5 years in prison for burglary, but if both refuse to confess, they will serve only 1 year each for the lesser charge of possessing stolen property. However, the police separately offer each a deal: if you testify against your partner as the leader of a burglary ring, you'll go free, while the partner will serve 10 years.

PRISONER'S DILEMMA

Now Alice and Bob face the so-called **prisoner's dilemma**: should they testify or refuse? Being rational agents, Alice and Bob each want to maximize their own expected utility. Let's assume that Alice is callously unconcerned about her partner's fate, so her utility decreases in proportion to the number of years she will spend in prison, regardless of what happens to Bob. Bob feels exactly the same way. To help reach a rational decision, they both construct the following payoff matrix:

	Alice:testify	*Alice:refuse*
Bob:testify	$A = -5, B = -5$	$A = -10, B = 0$
Bob:refuse	$A = 0, B = -10$	$A = -1, B = -1$

Alice analyzes the payoff matrix as follows: Suppose Bob testifies. Then I get 5 years if I testify and 10 years if I don't, so in that case testifying is better. On the other hand, if Bob refuses, then I get 0 years if I testify and 1 year if I refuse, so in that case as well testifying is better. So in either case, it's better for me to testify, so that's what I must do.

DOMINANT STRATEGY
STRONGLY DOMINATES

Alice has discovered that *testify* is a **dominant strategy** for the game. We say that a strategy s for player p **strongly dominates** strategy s' if the outcome for s is better for p than the outcome for s', for every choice of strategies by the other players. Strategy s **weakly**

WEAKLY DOMINATES

dominates s' if **s** is better than s' on at least one strategy profile and no worse on any other. A dominant strategy is a strategy that dominates all others. It is irrational to play a strongly dominated strategy, and irrational not to play a dominant strategy if one exists. Being rational, Alice chooses the dominant strategy. We need just a bit more terminology before we go on:

PARETO OPTIMAL

we say that an outcome is **Pareto optimal**[6] if there is no other outcome that all players would

PARETO DOMINATED

prefer. An outcome is **Pareto dominated** by another outcome if all players would prefer the other outcome.

If Alice is clever as well as rational, she will continue to reason as follows: Bob's dominant strategy is also to testify. Therefore, he will testify and we will both get five years. When each player has a dominant strategy, the combination of those strategies is called a

DOMINANT STRATEGY EQUILIBRIUM
EQUILIBRIUM

dominant strategy equilibrium. In general, a strategy profile forms an **equilibrium** if no

[6] Pareto optimality is named after the economist Vilfredo Pareto (1848–1923).

player can benefit by switching strategies, given that every other player sticks with the same strategy. An equilibrium is essentially a **local optimum** in the space of policies; it is the top of a peak that slopes downward along every dimension, where a dimension corresponds to a player's strategy choices.

The *dilemma* in the prisoner's dilemma is that the outcome of the equilibrium point is worse for both players than the outcome they would get if they both refused to testify. In other words, the outcome for the equilibrium solution is Pareto dominated by the (-1, -1) outcome of *(refuse,refuse)*.

Is there any way for Alice and Bob to arrive at the (-1, -1) outcome? It is certainly an *allowable* option for both of them to refuse to testify, but is is an *unlikely* option. Either player contemplating playing *refuse* will realize that he or she would do better by playing *testzfy*. That is the attractive power of an equilibrium point.

NASH EQUILIBRIUM

The mathematician John Nash (1928–) proved that *every game has an equilibrium of the type defined here.* It is now called a **Nash equilibrium** in his honor. Clearly, a dominant strategy equilibrium is a Nash equilibrium (Exercise 17.9), but not all games have dominant strategies. Nash's theorem means that there are equilibrium strategies even when there is no dominant strategy.

For the prisoner's dilemma, only the strategy profile *(testify, testzfy)* is a Nash equilibrium. It is hard to see how rational players can avoid this outcome, because in any proposed non-equilibrium solution at least one of the players will be tempted to change strategies. Game theorists agree that being a Nash equilibrium is a necessary condition for being a solution—although they disagree whether it is a sufficient condition.

It is easy enough to avoid the *(testify, testzfy)* solution if we change the game (or the players) in some way. For example, we could change to an iterated game in which the players know that they will meet again (but crucially, they must be uncertain about how many times they will meet again). Or if the agents have moral beliefs that encourage cooperation and fairness, we could change the payoff matrix to reflect the utility to each agent of cooperating with the other. We will see later that changing the agents to have limited computational powers, rather than the ability to reason absolutely rationally, can also affect the outcome, as can telling one agent that the other has limited rationality.

Now, let's look at a game that has no dominant strategy. Acme, a video game hardware manufacturer, has to decide whether its next game machine will use DVDs or CDs. Meanwhile, the video game software producer Best needs to decide whether to produce its next game on DVD or CD. The profits for both will be positive if they agree and negative if they disagree, as shown in the following payoff matrix:

	Acme:dvd	*Acme:cd*
Best:dvd	$A = 9, B = 9$	$A = -4, B = -1$
Best:cd	$A = -3, B = -1$	$A = 5, B = 5$

There is no dominant strategy equilibrium for this game, but there are *two* Nash equilibria: (dvd, *dvd*) and (cd, *cd*). We know these are Nash equilibria because, if either player unilaterally moves to a different strategy, that player will be worse off. Now the agents have a problem: *there are multiple acceptable solutions, but if each agent chooses a different*

solution then the resulting strategy *profile* won't be *a* solution at all and both agents will *suffer.* How can they agree on a solution? One answer is that both should choose the Pareto-optimal solution (dvd, *dvd)*; that is, we can restrict the definition of "solution" to the unique Pareto-optimal Nash equilibrium provided that one exists. Every game has at least one Pareto-optimal solution, but a game might have several, or they might not be equilibrium points. For example, we could set the payoffs for (dvd, dvd) to 5 instead of 9. In that case, there are two equal Pareto-optimal equilibrium points. To choose between them the agents can either guess or communicate, which can be done either by establishing a convention that orders the solutions before the game begins or by negotiating to reach a mutually beneficial solution during the game (which would mean including communicative actions as part of a multimove game). Communication thus arises in game theory for exactly the same reasons that it arose in multiagent planning in Chapter 12. Games in which players need to communicate like this are called **coordination games.**

COORDINATION GAME

We have seen that a game can have more than one Nash equilibrium; how do we know that every game must have at least one? It can be that a game has no pure-strategy Nash equilibria. Consider, for example, any pure strategy profile for two-finger Morra (page 632). If the total number of fingers is even then O will want to switch; if the total is odd then E will want to switch. Therefore no pure strategy profile can be an equilibrium and we must look to mixed strategies.

But which mixed strategy? In 1928, von Neumann developed a method for finding the optimal mixed strategy for two-player, **zero-sum games.** A zero-sum game is a game in which the payoffs in each cell of the payoff matrix sum to zero.[7] Clearly, Morra is such a game. For two-player, zero-sum games, we know that the payoffs are equal and opposite, so we need consider the payoffs of only one player, who will be the maximizer (just as in Chapter 6). For Morra, we pick the even player E to be the maximizer, so we can define the payoff matrix by the values $U_E(e, o)$—the payoff to E if E does e and **O** does o.

ZERO-SUMGAME

MAXIMIN

Von Neumann's method is called the the **maximin** technique, and it works as follows:

- Suppose we change the rules to force E to reveal his or her strategy first, followed by O. Then we have a turn-taking game to which we can apply the standard **minimax** algorithm from Chapter 6. Let's suppose this gives an outcome $U_{E,O}$. Clearly, this game favors O, so the true utility U of the game (from E's point of view) is at least $U_{E,O}$. For example, if we just look at pure strategies, the minimax game tree has a root value of **−3** (see Figure 17.11(a)), so we know that $U \geq -3$.
- Now suppose we change the rules to force **O** to reveal his or her strategy first, followed by E. Then the minimax value of this game is $U_{O,E}$, and because this game favors E we know that U is at most $U_{O,E}$. With pure strategies, the value is +2 (see Figure 17.11(b)), so we know $U \leq +2$.

[7] More general is the concept of **constant-sum games,** in which the sum of every cell in the game adds up to a constant, c. An n-person constant-sum game can be turned into a zero-sum game by subtracting c/n from every payoff. Thus chess, with traditional payoff of 1 for a win, 1/2 for a draw, and 0 for a loss, is technically a constant-sum game with $c = 1$, but can easily be transformed into a zero-sum game by subtracting 1/2 from every payoff.

Combining these two arguments, we see that the true utility U of the solution must satisfy

$$U_{E,O} \leq U \leq U_{O,E} \qquad \text{or in this case,} \qquad -3 \leq U \leq 2 .$$

To pinpoint the value of U, we need to turn our analysis to mixed strategies. First, observe the following: *once the first player has revealed his or her strategy, the secondplayer cannot lose by playing a pure strategy.* The reason is simple: if the second player plays a mixed strategy, $[p: one; (1-p): two]$,its expected utility is a linear combination $(p.u_{one} + (1-p).u_{two})$ of the utilities of the pure strategies, u_{one} and u_{two}. This linear combination can never be better than the best of u_{one} and u_{two}, so the second player might as well play a pure strategy.

With this observation in mind, the minimax trees can be thought of as having infinitely many branches at the root, corresponding to the infinitely many mixed strategies the first player can choose. Each of these leads to a node with two branches corresponding to the pure strategies for the second player. We can depict these infinite trees finitely by having one "parameterized" choice at the root:

- If E moves first, the situation is as shown in Figure 17.11(c). E plays $[p: one; (1-p): two]$ at the root, and then O chooses a move given the value of p. If O chooses *one,* the expected payoff (to E) is $2p - 3(1-p) = 5p - 3$; if O chooses *two,* the expected payoff is $-3p + 4(1-p) = 4 - 7p$. We can draw these two payoffs as straight lines on a graph, where p ranges from 0 to 1 on the x-axis, as shown in Figure 17.11(e). 0 , the minimizer, will always choose the lower of the two lines, as shown by the heavy lines in the figure. Therefore, the best that E can do at the root is to choose p to be at the intersection point, which is where

  $$5p - 3 = 4 - 7p \qquad\qquad p = 7/12 .$$

  The utility for E at this point is $U_{E,O} = -1/12$.
- If O moves first, the situation is as shown in Figure 17.11(d). O plays $[q: one; (1-q): two]$ at the root, and then E chooses a move given the value of q. The payoffs are $2q - 3(1-q) = 5q - 3$ and $-3q + 4(1-q) = 4 - 7q$.[8] Again, Figure 17.11(f) shows that the best O can do at the root is to choose the intersection point:

  $$5q - 3 = 4 - 7q \qquad \Rightarrow \qquad q = 7/12 .$$

  The utility for E at this point is $U_{O,E} = -1/12$.

Now we know that the true utility of the game lies between $-1/12$ and $-1/12$, that is, it is exactly $-1/12$! (The moral is that it is better to be O than E if you are playing this game.) Furthermore, the true utility is attained by the mixed strategy $[7/12: one; 5/12: two]$,which should be played by both players. This strategy is called the **maximin equilibrium** of the game, and is a Nash equilibrium. Note that each component strategy in an equilibrium mixed strategy has the same expected utility. In this case, both *one* and *two* have the same expected utility, $-1/12$, as the mixed strategy itself.

MAXIMIN EQUILIBRIUM

Our result for two-finger Morra is an example of the general result by von Neumann: *every two-player zero-sum game has a maximin equilibrium when you allow mixed strategies.*

[8] It is a coincidence that these equations are the same as those for p; the coincidence arises because $U_E(one, two) = U_E(two, one) = -3$. This also explains why the optimal strategy is the same for both players.

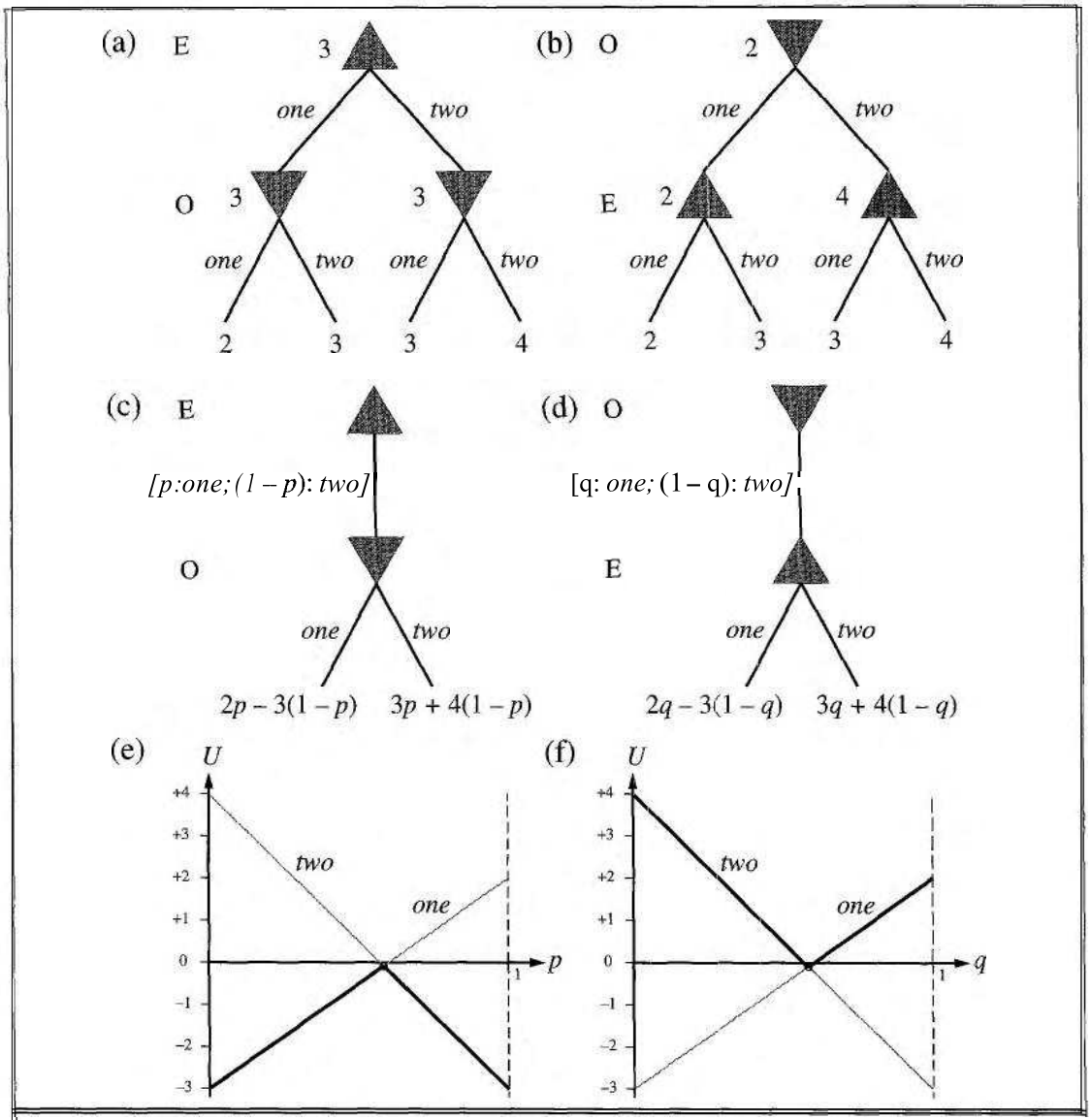

Figure 17.11 (a) and (b): Minimax game trees for two-finger Morra if the players take turns playing pure strategies. (c) and (d): Paramete ed game trees where the first player plays a mixed strategy. The payoffs depend on the probability parameter (p or q) in the mixed strategy. (e) and (f): For any particular value of the probability parameter, the second player will choose the "better" of the two actions, so the value of the first player's mixed strategy is given by the heavy lines. The first player will choose the probability parameter for the mixed strategy at the intersection point.

Furthermore, every Nash equilibrium in a zero-sum game is a maximin for both players. The general algorithm for finding maximin equilibria in zero-sum games is somewhat more involved than Figures 17.11(e) and (f) might suggest. When there are n possible actions, a mixed strategy is a point in n-dimensional space and the lines become hyperplanes. It's

also possible for some pure strategies for the second player to be dominated by others, so that they are not optimal against any strategy for the first player. After removing all such strategies (which might have to be done repeatedly), the optimal choice at the root is the highest (or lowest) intersection point of the remaining hyperplanes. Finding this choice is an example of a **linear programming** problem: maximizing an objective function subject to linear constraints. Such problems can be solved by standard techniques in time polynomial in the number of actions (and in the number of bits used to specify the reward function, if you want to get technical).

The question remains, what should a rational agent actually do in playing a single game of Morra? The rational agent will have derived the fact that [7/12: one; 5/12: two] is the maximin equilibrium strategy, and will assume that this is mutual knowledge with a rational opponent. The agent could use a 12-sided die or a random number generator to pick randomly according to this mixed strategy, in which case the expected payoff would be -1/12 for E. Or the agent could just decide to play one, or two. In either case, the expected payoff remains -1/12 for E. Curiously, unilaterally choosing a particular action does not harm one's expected payoff, but allowing the other agent to know that one has made such a unilateral decision does affect the expected payoff, because then the opponent can adjust his strategy accordingly.

Finding solutions to non-zero-sum finite games (i.e., Nash equilibria) is somewhat more complicated. The general approach has two steps: (1) Enumerate all possible subsets of actions that might form mixed strategies. For example, first try all strategy profiles where each player uses a single action, then those where each player uses either one or two actions, and so on. This is exponential in the number of actions, and so only applies to relatively small games. (2) For each strategy profile enumerated in (1), check to see if it is an equilibrium. This is done by solving a set of equations and inequalities that are similar to the ones used in the zero-sum case. For two players these equations are linear and can be solved with basic linear programming techniques, but for three or more players they are nonlinear and may be very difficult to solve.

REPEATEDGAME

So far we have looked only at games that last a single move. The simplest kind of multiple-move game is the **repeated game,** in which players face the same choice repeatedly, but each time with knowledge of the history of all players' previous choices. A strategy profile for a repeated game specifies an action choice for each player at each time step for every possible history of previous choices. As with MDPs, payoffs are additive over time.

Let's consider the repeated version of the prisoner's dilemma. Will Alice and Bob work together and refuse to testify, knowing that they will meet again? The answer depends on the details of the engagement. For example, suppose Alice and Bob know that they must play exactly 100 rounds of prisoner's dilemma. Then they both know that the 100th round will not be a repeated game—that is, its outcome can have no effect on future rounds—and therefore they will both choose the dominant strategy, testify, in that round. But once the 100th round is determined, the 99th round can have no effect on subsequent rounds, so it too will have a dominant strategy equilibrium at $(testify, testify)$. By induction, both players will choose testify on every round, earning a total jail sentence of 500 years each.

We can get different solutions by changing the rules of the interaction. For example, suppose that after each round there is a 99% chance that the players will meet again. Then

the expected number of rounds is still 100, but neither player knows for sure which round will be the last. Under these conditions, more cooperative behavior is possible. For example, one equilibrium strategy is for each player to *refuse* unless the other player has ever played *testify*. This strategy could be called **perpetual punishment.** Suppose both players have adopted this strategy, and this is mutual knowledge. Then as long as neither player has played *testify,* then at any point in time the expected future total payoff for each player is

PERPETUAL PUNISHMENT

$$\sum_{t=0}^{\infty} 0.99^t \cdot (-1) = -100 .$$

A player who chooses *testify* will gain a score of 0 rather than -1 on the very next move, but his or her total expected future payoff becomes

$$0 + \sum_{t=1}^{\infty} 0.99^t \cdot (-10) = -999$$

Therefore, at every step, there is no incentive to deviate from *(refuse, refuse).* Perpetual punishment is the "mutually assured destruction" strategy of the prisoner's dilemma: once either player decides to *testify,* it assures that both players suffer a great deal. But it only works as a deterrent if the other player believes you have adopted this strategy--or at least that you might have adopted it.

There are other strategies that are more forgiving. The most famous, called **tit-for-tat,** calls for starting with *refuse* and then echoing the other player's previous move on all subsequent moves. So Alice would refuse as long as Bob refuses and would testify the move after Bob testified, but would go back to refusing if Bob did. Although very simple, this strategy has proven to be highly robust and effective against a wide variety of strategies.

TIT-FOR-TAT

We can also get different solutions by changing the agents, rather than changing the rules of engagement. Suppose the agents are finite-state machines with n states and they are playing a game with $m > n$ total steps. The agents are thus incapable of representing the number of remaining steps, and must treat it as an unknown. Therefore they cannot do the induction, and are free to arrive at the more favorable (refuse, refuse) equilibrium. In this case, ignorance is bliss—or rather, having your opponent believe that you are ignorant is bliss. Your success in these repeated games depends on the other player's perception of you as a bully or a simpleton, and not on your actual characteristics.

Repeated games in full generality are beyond the scope of this book, but they arise in many settings. For example, we can construct a sequential game by putting two agents in the 4 x 3 world of Figure 17.1. If we specify that no movement occurs when the two agents try to move into the same square simultaneously (a common problem at many traffic intersections), then certain pure strategies can get stuck forever. One solution is for each agent to randomize its choice between moving forward and staying put; the stalemate will be resolved quickly and both agents will be happy. This is exactly what is done to resolve packet collisions in Ethernet networks.

Currently known solution methods for repeated games resemble those for turn-taking games in Chapter 6, in that a game tree can be constnucted from the root downwards and solved from the leaves upwards. The main difference is that, instead of simply taking the

maximum or minimum of the child values, the algorithm must solve a game in mixed strategies at each level, assuming that the child nodes have been solved and have well-defined values to work with.

PARTIAL INFORMATION

Repeated games in *partially observable* environments are called games of partial information. Examples include card games such as poker and bridge, wherein each player can see only a subset of the cards, and more serious "games" such as abstractions of nuclear war, where neither side knows the location of all its opponent's weapons. Games of partial information are solved by considering a tree of belief states, as in POMDPs. (See Section 17.4.) One important difference is that, while one's own belief state is observable, the opponent's belief state is not. Only recently have practical algorithms been developed for such games. Some simplified versions of poker have been solved, proving that bluffing is indeed a rational choice, as part of a well balanced mixed strategy. One important insight to emerge from such studies is that mixed strategies are useful not just for making one's actions unpredictable, but also for minimizing the amount of information that one's opponent can learn from observing one's actions. It is interesting that, although designers of programs for playing bridge are well aware of the importance of gathering and hiding information, none has yet proposed the use of randomized strategies.

So far, there have been some barriers that have prevented game theory from being widely used in agent design. First, note that in a Nash equilibrium solution, a player is assuming that the opponents will definitely play the equilibrium strategy. This means that the player is unable to incorporate any beliefs it might have about how the other players are likely to act, and therefore that it might be wasting some of its value defending against threats that will never materialize. The notion of a Bayes–Nash equilibrium partially addresses this point: it is an equilibrium with respect to a player's prior probability distribution over the other players' strategies—in other words, it expresses a player's beliefs about the other players' likely strategies. Second, there is currently no good way to combine game theoretic and POMDP control strategies. Because of these and other problems, game theory has been used primarily to *analyze* environments that are at equilibrium, rather than to *control* agents within an environment. We shall soon see how it can help *design* environments.

BAYES–NASH EQUILIBRIUM

17.7 MECHANISM DESIGN

In the previous section, we looked at the question "Given a game, what is a rational strategy?" In this section, we ask "Given that agents are rational, what game should we design?" More specifically, we would like to design a game whose solutions, consisting of each agent pursuing its own rational strategy, result in the maximization of some global utility function. This problem is called mechanism design, or sometimes inverse game theory. Mechanism design is a staple of economics and political science. For collections of agents, it holds the possibility of using game-theoretic mechanisms to construct smart systems out of a collection of more limited systems--even noncooperative systems—in much the same way that teams of humans can achieve goals far beyond the reach of any individual.

MECHANISM DESIGN

Examples of mechanism design include auctioning off cheap airline tickets, routing TCP packets between computers, deciding how medical interns will be assigned to hospitals, and deciding how robotic soccer players will cooperate with their teammates. Mechanism design became more than an academic subject in the 1990s when several nations, faced with the problem of auctioning off licenses to broadcast in various frequency bands, lost hundreds of millions of dollars in potential revenue as a result of poor mechanism design. Formally, a

MECHANISM

mechanism consists of (1) a language for describing the (possibly infinite) set of allowable strategies that agents may adopt and (2) an outcome rule G that determines the payoffs to the agents given a strategy profile of allowable strategies.

At first sight, the mechanism design problem can seem trivial. Suppose that the global utility function U is decomposed into any set of individual agent utility functions U_i, such that $\mathrm{U} = \sum_i U_i$. Then, one might say, if each agent maximizes its own utility, surely that will lead automatically to the maximization of the global utility. (For example, Capitalism 101 says that if everyone tries to get rich, the total wealth of society will increase.) Unfortunately, this doesn't work. The actions of each agent could affect the well-being of other agents in

TRAGEDY OF THE COMMONS

ways that decrease global utility. One example of this is the **tragedy of the commons,** a situation in which individual farmers all bring their livestock to graze for free on the town commons, with the result being the destruction of the commons and a negative utility for all the farmers. Each farmer individually acted rationally, reasoning that the use of the commons was free and that, although using the commons could lead to its destruction, refraining from using it would not help (because the others would use it anyway). Similar arguments apply to the use of the atmosphere and the oceans for free dumping of pollutants.

A standard approach in mechanism design for dealing with such problems is to charge

EXTERNALITIES

each agent for using the commons. More generally, we need to ensure that all **externalities**—effects on global utility that are not recognized in the individual agents' transactions—are made explicit. Setting the prices correctly is the difficult part. In the limit, this approach amounts to creating a mechanism in which each agent is effectively required to maximize global utility. This is an impossibly difficult task for the agent, who can neither assess the current state of the world nor observe the effects of its actions on all other agents. Mechanism design therefore concentrates on finding mechanisms for which the decision problem for the individual agents is straightforward.

Let's consider auctions first. In the most common form, an auction is a mechanism for selling some goods to members of a pool of bidders. The strategies are the bids and the outcome determines who gets the goods and how much they pay. One example of where auctions can come into play within AI is when a collection of agents are deciding whether to cooperate on a joint plan. Hunsberger and Grosz (2000) show that this can be accomplished efficiently with an auction in which the agents bid for roles in the joint plan.

For now, we'll consider auctions wherein (1) there is a single good, (2) each bidder has a utility value v_i for the good, and (3) these values are known only to the bidder. Bidders make bids b_i, and the highest bid wins the goods, but the mechanism determines how the bids are made and the price paid by the winner (it need not be b_i). The best-known type of

ENGLISH AUCTION

auction is the **English auction,** in which the auctioneer increments the price of the goods, checking whether bidders are still interested, until only one bidder is left. This mechanism

has the property that the bidder with the highest value v_i gets the goods at a price of $b_2 + d$, where b_2 is the highest bid among all the other players and d is the auctioneer's increment between bids.[9] The English auction also has the property that bidders have a simple dominant strategy: keep bidding as long as the current cost is below your personal value. Recall that "dominant" means that the strategy works against all other strategies, which in turn means that a player can adopt it without regard for the other strategies. Therefore, players don't have to waste time and energy contemplating other players' possible strategies. A mechanism where players have a dominant strategy that involves revealing their true incentives is called a **strategy-proof** mechanism.

STRATEGY-PROOF

One negative property of the English auction is its high communication costs, so either the auction takes place in one room or all bidders have to have high-speed, secure communication lines. An alternative mechanism that requires much less communication is the **sealed bid auction.** Here, each bidder makes a single bid and communicates it to the auctioneer, and the highest bid wins. With this mechanism, the strategy of bidding your true value is no longer dominant. If your value is v_i and you believe that the maximum of all the other players' bids will be b_2 then you should bid the lower of v_i and $b_2 + \epsilon$. Two drawbacks of the sealed bid auction are that the player with the highest v_i might not get the goods and that players must spend effort contemplating the other players' strategies.

SEALED BID AUCTION

A small change in the rules for sealed bid auctions produces the **sealed bid second-price auction,** also known as a **Vickrey auction.**[10] In such auctions, the winner pays the price of the second highest bid, rather than paying his own bid. This simple modification completely eliminates the complex deliberations required for standard (or **first-price)** sealed bid auctions, because the dominant strategy is now to bid your actual value. To see that, we note that any player can think of the auction as a two-player game, ignoring all players except himself and the highest bidder among the other players. The utility of player i in terms of his bid b_i, his value v_i, and the best bid among the other players, b_m is

SEALED BID SECOND-PRICE AUCTION
VICKREY AUCTION

$$u_i(b_i, b_m) = \begin{cases} (v_i - b_m) & \text{if } b_i > b_m \\ 0 & \text{otherwise.} \end{cases}$$

To see that $b_i = v_i$ is a dominant strategy, note that when $(v_i - b_m)$ is positive, any bid that wins the auction is optimal, and bidding v_i in particular wins the auction. On the other hand, when $(v_i - b_m)$ is negative, any bid that loses the auction is optimal, and bidding v_i in particular loses the auction. So bidding v_i is optimal for all possible values of b_m, and in fact, v_i is the only bid that has this property. Because of its simplicity and the minimal computation requirements for both seller and bidders, the Vickrey auction is widely used in constructing distributed AI systems.

Now let's consider the Internet routing problem. The players correspond to edges in the graph of network connections. Each player knows the cost c_i of sending a message along its own edge; the cost of not having a message to send is 0. The goal is to find the cheapest path for a message to travel from origin to destination, where the cost of the whole route

[9] There is actually a small chance that the player with highest v_i fails to get the goods, in the case where $b_2 < v_i < b_2 + d$. The chance of this happening can be made arbitrarily small by decreasing the increment d.

[10] Named after William Vickrey (1914–1996), winner of the 1996 Nobel prize in economics.

is the sum of the individual edge costs. Chapter 4 gives several algorithms for computing the shortest path, given the edge costs, so all we have to do is get each agent to report its true cost, c_i. Unfortunately, if we just ask each agent, it will report costs that are high, to encourage us to send traffic elsewhere. We need to develop a strategy-proof mechanism. One such mechanism is to pay each player a payoff p_i equal to the length of the shortest path that does not contain the ith edge minus the length of the shortest path (as computed by a search algorithm) where the cost of the ith edge is assumed to be 0:

$$p_i = \text{LENGTH}(\text{path with } c_i = \infty) - \text{LENGTH}(\text{path with } c_i = 0) .$$

We can show that, under this mechanism, the dominant strategy for each player is to report c_i truthfully and that doing so will result in a cheapest path. Despite this desirable property, the mechanism outlined here is not used in practice, because of the high communication and central computation cost. The mechanism designer rnust communicate with all n players and then must solve the optimization problem. This might be worth it if the costs could be amortized over many messages, but in a real network tlhe costs c_i would fluctuate constantly, because of traffic congestion and because some machines will crash and others will come online. No completely satisfactory solution has yet been devised.

17.8 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 uncertain 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 utility of the state sequences encountered when an optimal policy is executed, starting in that state. The **value iteration** algorithm for solving MDPs works by iteratively solving the equations relating the utilities of each state to that 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. 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 observation models, to update its belief state, and to project forward possible action sequences.
- **Game theory** describes rational behavior for agents in situations where multiple agents interact simultaneously. Solutions of games are **Nash** equilibria — strategy profiles in which no agent has an incentive to deviate from the specified strategy.
- **Mechanism design** can be used to set the rules by which agents will interact, in order to maximize some global utility through the operation of individually rational agents. Sometimes, mechanisms exist that achieve this goal without requiring each agent to consider the choices made by other agents.

We shall return to the world of MDPs and POMDP in Chapter 21, when we study **reinforcement learning** methods that allow an agent to improve its behavior from experience in sequential, uncertain environments.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Richard Bellman (1957) initiated the modem approach to sequential decision problems and proposed the dynamic programming approach in general and the value iteration algorithm in particular. 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). 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 (17.9). The analysis of discounting in terms of stationary preferences is due to Koopmans (1972). The texts by Bertsekas (1987), Puterman (1994), and Bertsekas and Tsitsiklis (1996) provide a rigorous introduction to sequential decision problems. Papadimitriou and Tsitsiklis (1987) describe results on the computational complexity of MDPs.

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, as did the later survey by Barto *et al.* (1995). (Earlier work by Werbos (1977) contained many similar ideas, but was not taken up to the same extent.) The connection between MDPs and AI planning problems was made first by Sven Koenig (1991), who showed how probabilistic STRIPS operators provide a compact representation for transition models. (See also Wellman (1990b).) Work by Dean *et al.* (1993) and Tash and Russell (1994) attempted to overcome the combinatorics of large state spaces by using a limited search horizon and abstract states. Heuristics based on the value of information can be used to select areas of the state space where a local expansion of the horizon will yield a significant improvement in decision quality. Agents using this approach can tailor their effort to handle time pressure and generate some interesting behaviors such as using familiar "beaten paths" to find their way around the state space quickly without having to recompute optimal decisions at each point. Recent work by Boutilier and others (Boutilier *et al.*, 2000,2001) has focused on dynamic program-

ming using *symbolic* representations of both transition models and value functions, based on propositional and first-order formulæ.

The observation that a partially observable MDP can be transformed into a regular MDP by using the belief states is due to Astrom (1965). The first complete algorithm for the exact solution of partially-observable Markov decision processes (POMDPs) 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) surveys the state of the art in POMDPs. 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. In this policy representation, the belief state corresponds directly to a particular state in the automaton. Approximately optimal policies for POMDPs can be constructed by forward search combined with sampling of possible observations and action outcomes (Kearns *et* al., 2000; Ng and Jordan, 2000). Additional work on POMDP algorithms is covered in Chapter 21.

The basic ideas for an agent architecture using dynamic decision networks were proposed by Dean and Kanazawa (1989a). The book *Planning and Control* by Dean and Wellman (1991) goes into much greater depth, making connections between DBN/DDN models and the classical control literature on filtering. Tatman and Shachter (1990) showed how to apply dynamic programming algorithms to DDN models. Russell (1998) explains various ways in which such agents can be scaled up and identifies a number of open research issues.

The early roots of game theory can be traced back to proposals made in the 17th century by Christiaan Huygens and Gottfried Leibniz to study competitive and cooperative human interactions scientifically and mathematically. Throughout the 19th century, several leading economists created simple mathematical examples to analyze particular examples of competitive situations. The first formal results in game theory are due to Zermelo (1913) (who had, the year before, suggested a form of minimax search for games, albeit an incorrect one). Emile Borel (1921) introduced the notion of a mixed strategy. John von Neumann (1928) proved that every two-person, zero-sum game has a maximin equilibrium in mixed strategies and a well-defined value. Von Neumann's collaboration with the economist Oskar Morgenstern led to the publication in 1944 of the *Theory of Games and Economic Behavior,* the defining book for game theory. Publication of the book was delayed by the wartime paper shortage until a member of the Rockefeller family personally subsidized its publication.

In 1950, at the age of 21, John Nash published his ideas concerning equilibria in general games. His definition of an equilibrium solution, although originating in the work of Cournot (1838), became known as Nash equilibrium. After a long delay due to the schizophrenia he suffered from 1959 onwards, Nash was awarded the Nobel prize in Economics (along with Reinhart Selten and John Harsanyi) in 1994.

The Bayes–Nash equilibrium is described by Harsanyi (1967) and discussed by Kadane and Larkey (1982). Some issues in the use of game theory for agent control are covered by Binmore (1982).

The prisoner's dilemma was invented as a classroom exercise by Albert W. Tucker in 1950 and is covered extensively by Axelrod (1985). Repeated games were introduced by

Luce and Raiffa (1957) as were games of partial information by Kuhn (1953). The first practical algorithm for partial information games was developed within AI by Koller *et* al. (1996); the paper by Koller and Pfeffer (1997) provides a readable introduction to the general area and describes a working system for representing and solving sequential games. Game theory and MDPs are combined in the theory of Markov games (Littman, 1994). Shapley (1953) actually described the value iteration algorithm before Bellman, but his results were not widely appreciated, perhaps because they were presented in the context of Markov games. Textbooks on game theory include those by Myerson (1991), Fudenberg and Tirole (1991), and Osborne and Rubinstein (1994).

The tragedy of the commons, a motivating problem for the field of mechanism design, was presented by Hardin (1968). Hurwicz (1973) created a mathematical foundation for mechanism design. Milgrom (1997) writes about the multibillion-dollar spectrum auction mechanism he designed. Auctions can also be used in planning (Hunsberger and Grosz, 2000) and scheduling (Rassenti *et* al., 1982). Varian (1995) gives a brief overview with connections to the computer science literature, and Rosenschein and Zlotkin (1994) present a book-length treatment with applications to distributed AI. Related work on distributed AI also goes under other names, including Collective Intelligence (Turner and Wolpert, 2000) and market-based control (Clearwater, 1996). Papers on computational issues in auctions often appear in the ACM Conferences on Electronic Commerce.

EXERCISES

17.1 For the 4 x **3** world shown in Figure 17.1, calculate which squares can be reached from (1,1) by the action sequence *[Up,Up, Right, Right, Right]* and with what probabilities. Explain how this computation is related to the task of projecting a hidden Markov model.

17.2 Suppose that we define the utility of a state sequence to be the ***maximum*** reward obtained in any state in the sequence. Show that this utility function does not result in stationary preferences between state sequences. Is it still possible to define a utility function on states such that MEU decision making gives optimal behavior?

17.3 Can any finite search problem be translated exactly into a Markov decision problem such that an optimal solution of the latter is also an optimal solution of the former? If so, explain ***precisely*** how to translate the problem and how to translate the solution back; if not, explain ***precisely*** why not (i.e., give a counterexample).

17.4 Consider an undiscounted MDP having three states, (1, 2, 3), with rewards $-1, -2, 0$ respectively. State 3 is a terminal state. In states 1 and 2 there are two possible actions: a and b. The transition model is as follows:

- In state 1, action a moves the agent to state 2 with probability 0.8 and makes the agent stay put with probability 0.2.
- In state 2, action a moves the agent to state 1 with probability 0.8 and makes the agent stay put with probability 0.2.

- In either state 1 or state 2, action b moves the agent to state 3 with probability 0.1 and makes the agent stay put with probability 0.9.

Answer the following questions:

a. What can be determined *qualitatively* about the optimal policy in states 1 and 2?

b. Apply policy iteration, showing each step in full, to determine the optimal policy and the values of states 1 and 2. Assume that the initial policy has action b in both states.

c. What happens to policy iteration if the initial policy has action a in both states? Does discounting help? Does the optimal policy depend on the discount factor?

17.5 Sometimes MDPs are formulated with a reward function $R(s, a)$ that depends on the action taken.or a reward function $R(s, \mathrm{a}, s')$ that also depends on the outcome state.

a. Write the Bellman equations for these formulations.

b. Show how an MDP with reward function $R(s, a, s')$ can be transformed into a different MDP with reward function $R(s, \mathrm{a})$, such that optimal policies in the new MDP correspond exactly to optimal policies in the original MDP.

c. Now do the same to convert MDPs with $R(s, a)$ into MDPs with $R(s)$.

17.6 Consider the 4 x **3** world shown in Figure 17.1.

a. Implement an environment simulator for this environment, such that the specific geography of the environment is easily altered. Some code for doing this is already in the online code repository.

b. Create an agent that uses policy iteration, and measure its performance in the environment simulator from various starting states. Perform several experiments from each starting state, and compare the average total reward received per run with the utility of the state, as determined by your algorithm.

c. Experiment with increasing the size of the environment. How does the runtime for policy iteration vary with the size of the environment?

17.7 For the environment shown in Figure 17.1, find all the threshold values for $R(s)$ such that the optimal policy changes when the threshold is crossed. You will need a way to calculate the optimal policy and its value for fixed $R(s)$. *[Hint:* Prove that the value of any fixed policy varies linearly with $R(s)$.**]**

17.8 In this exercise we will consider two-player MDPs that correspond to zero-sum, turn-taking games like those in Chapter 6. Let the players be: A and B, and let $R(s)$ be the reward for player A in s. (The reward for B is always equal and opposite.)

a. Let $U_A(s)$ be the utility of state s when it is A's turn to move in s, and let $U_B(\mathrm{s})$ be the utility of state s when it is B's turn to move in s. All rewards and utilities are calculated from A's point of view (just as in a minimax game tree). Write down Bellman equations defining $U_A(s)$ and $U_B(s)$.

b. Explain how to do two-player value iteration with these equations, and define a suitable stopping criterion.

c. Consider the game described in Figure 6.14 on page 190. Draw the state space (rather than the game tree), showing the moves by A as solid lines and moves by B as dashed lines. Mark each state with $R(s)$. You will find it helpful to arrange the states (s_A, s_B) on a two-dimensional grid, using s_A and s_B as "coordinates."

d. Now apply two-player value iteration to solve this game, and derive the optimal policy.

17.9 Show that a dominant strategy equilibrium is a Nash equilibrium, but not vice versa.

17.10 In the children's game of rock–paper–scissors each player reveals at the same time a choice of rock, paper, or scissors. Paper wraps rock, rock blunts scissors, and scissors cut paper. In the extended version rock–paper–scissors–fire–water, fire beats rock, paper and scissors; rock, paper and scissors beat water, and water beats fire. Write out the payoff matrix and find a mixed-strategy solution to this game.

17.11 Solve the game of three-finger Morra.

17.12 Prior to 1999, teams in the National Hockey League received 2 points for a win, 1 for a tie, and 0 for a loss. Is this a constant-sum game? In 1999, the rules were amended so that a team receives 1 point for a loss in overtime. The winning team still gets 2 points. How does this modification change the answers to the questions above? If it were legal to do so, when would it be rational for the two teams to secretly agree to end regulation play in a tie and then battle it out in overtime? Assume that the utility to each team is the number of points it receives, and that there is a mutually known prior probability p that the first team will win in overtime. For what values of p would both teams agree to this arrangement?

17.13 The following payoff matrix, from Blinder (1983) by way of Bernstein (1996), shows a game between politicians and the Federal Reserve.

	Fed: contract	Fed: do nothing	Fed: expand
Pol: contract	$F = 7, P = 1$	$F = 9, P = 4$	$F = 6, P = 6$
Pol: do nothing	$F = 8, P = 2$	$F = 5, P = 5$	$F = 4, P = 9$
Pol: expand	$F = 3, P = 3$	$F = 2, P = 7$	$F = 1, P = 8$

Politicians can expand or contract fiscal policy, while the Fed can expand or contract monetary policy. (And of course either side can choose to do nothing.) Each side also has preferences for who should do what—neither side wants to look like the bad guys. The payoffs shown are simply the rank orderings: 9 for first choice through 1 for last choice. Find the Nash equilibrium of the game in pure strategies. Is this a Pareto optimal solution? The reader might wish to analyze the policies of recent administrations in this light.