CHAPTER 15

LEARNING WITH A PHYSICAL STATE

All of the problems that we have considered in earlier chapters have one characteristic in common. In each of these problems, the decision we make depends only on our belief state about the problem. The decision itself can be discrete (as in Chapter 3), a subset (Chapter **??**), a scalar (Chapter 10), or continuous (Chapter 14), and we have considered many different types of objective functions (online, offline, or linked to a complex implementation decision) and different belief structures (lookup table, parametric and nonparametric). However, within each specific problem class, the set of possible decisions has always stayed the same over time, and every policy we have looked at has made decisions based purely on the available information.

There are many problems where this is not the case, and our decision also depends on a *physical state* as well as a belief state. A simple example arises when we are trying to learn the travel times on links in a transportation network. We can traverse from node i to node j to learn more about the time τ_{ij} to get from i to j, but doing so puts us at node j, which changes the decisions we can consider next. If we decide to go from i to j, we have to balance the information we gain about τ_{ij} against the impact of now being at location j.

Another application arises in medical treatments. Consider the problem of treating a diabetes patient. The doctor has certain beliefs about the effectiveness of different treatments. At the same time, the doctor also observes the results of blood sugar tests that are regularly undergone by the patient. The decision to recommend a

Copyright © 2018 John Wiley & Sons, Inc.

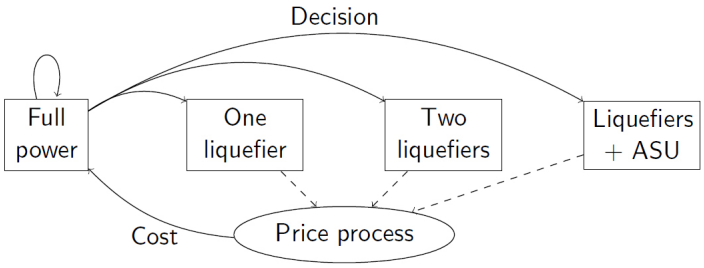

Figure 15.1 An operational flexibility problem, where the state of the power plant affects the energy consumed and the cost of resuming operations.

particular type of treatment is based on the doctor's beliefs about the effectiveness of that treatment, but also on the patient's physical condition. For example, a class of diabetes drugs known as "secretagogues" works by stimulating pancreatic cells to release insulin. Secretagogues can work well in reducing blood sugar, but may cause weight gain as a side effect. The decision to try a treatment can change the physical state of the patient (captured by his weight), which can then change the decisions we can make in the future.

A second example involves the management of a sensor moving around to collect information. This might be a robot sensing radiation near an accident, or a medical technician collecting information about the prevalence of a viral outbreak in the population. The information can be used to help us design response strategies. If we collect information at one location, it reduces the cost of collecting information at nearby locations. Our decision of what information to collect depends on the physical location of the sensor.

We use the term "physical state" somewhat loosely, but we always mean a state variable that changes the decisions that we are allowed to make. For example, imagine that we are selling a product where we can change the price and then learn how the market might respond to the price. We may feel that the market will respond badly to rapid changes in the price, and as a result we impose a restriction that we cannot change the price by more than one dollar. If we charge $20 during one week, we cannot then try charging $28 or $15 the next week. While we may not think of price as a physical state, this falls within the problem class that we wish to consider.

There are many practical problems where learning arises which also have a physical state. Below are some additional examples of problems that have a physical state:

- Operational flexibility - A manufacturer of industrial gases buys energy on the real-time market to run its cryogenic plants. The spot price of energy is subject to very sudden spikes, known as "coincident peaks" in the energy literature. Each plant runs an air separation unit (ASU) which breaks up air into component gases, as well as one or two recycle compressors or liquefiers. Shutting down all of these components allows us to save more money during a coincident peak, but if the peak never arrives, we will waste a lot of energy ramping back up to normal operations (which can take upwards of 24 hours). We have a belief state that helps us predict coincident peaks, but the physical state of the power plant also affects the costs incurred.

- Competitive games - Imagine a game of chess. Each player has a certain belief about the opponent's strategy, and uses this belief to decide on the next move. However, the decisions a player can make depends on the state of the board.
- Pricing with finite inventory - A store sets a price for a product with the goal of maximizing revenue. As in Chapters 7 and **??**, we may wish to experiment with different prices and observe the effect on revenues. However, if we have finite inventory, our decision will depend on the amount we have in stock as well as on our beliefs about the revenue curve. Experimentation is less useful when there is less inventory remaining.
- Mutual fund cash balance - A mutual fund needs cash to meet shareholder redemptions. At the same time, keeping more cash on hand means having less money to invest, leading to lower profits. Over time, we gradually learn about shareholder behavior and the rate at which redemptions arise. Our ability to meet redemptions at any given time, however, is constrained by the amount of cash on hand.

The physical state introduces a whole new level of challenge. The question of how to make decisions based on both physical and belief states is at the very frontier of optimal learning. In this chapter, we give a framework for where to begin thinking about this problem. We also suggest a few approaches that integrate the optimal learning concepts we have developed throughout this book into the world of physical states, and close with a discussion of how some of these concepts might be taken further.

15.1 INTRODUCTION TO DYNAMIC PROGRAMMING

Sequential, stochastic decision problems have long been approached using dynamic programming. Assume that we have a process that may be in a physical state $S_t = s$ at time t. (Throughout this book, we have denoted time by n; the reason why we switch to t here will become clear shortly.) If we choose a (discrete) action x, we may land in a state $S_{t+1} = s'$ with probability $p(s'|s, x)$. Assume that we earn a contribution $C(S_t, x)$ from taking action x when we are in state S_t, and that contributions are discounted by $0 < \gamma < 1$ with each passing time period. If $V_t(S_t)$ is the value of being in state S_t at time t, then we would like to choose our action x_t by solving

$$x_t = \arg\max_x \left(C(S_t, x) + \gamma \sum_{s'} p(s'|S_t, x) V_{t+1}(s') \right). \tag{15.1}$$

The problem is that we do not know $V_{t+1}(s')$. If we have a finite horizon problem with T as the last time period, we might assume $V_T(s) = 0$ then compute the remaining value functions using

$$V_t(S_t) = \max_x \left(C(S_t, x) + \gamma \sum_{s'} p(s'|S_t, x) V_{t+1}(s') \right). \tag{15.2}$$

Equation (15.2) needs to be computed by looping over all states S_t. If S_t is discrete and very low dimensional (as in, three or less), this approach can work quite well.

But there are many problems where the state S_t has more than three dimensions, or it may be continuous. For these problems (which are quite common), we encounter what is widely known as the "curse of dimensionality."

15.1.1 Approximate dynamic programming

The dynamic programming community has developed an algorithmic strategy known broadly as *approximate dynamic programming* (ADP) that is designed to circumvent the curse of dimensionality. ADP is actually an umbrella for a variety of algorithms, but its most popular form works as follows. Let's begin by assuming that the state S_t is discrete. Further assume that we are given an approximation $\overline{V}_t^0(s)$ for all states s and times $t = 1, \ldots, T$ (for example, we might start with $\overline{V}_t^0(s) = 0$).

If we use equation (15.2), we are solving the problem by proceeding backward in time. It is this process that requires that we loop over all possible states, because we do not know in advance which states that we might actually visit. With approximate dynamic programming, we are going to progress *forward* in time, starting with a given initial state S_0. If we are in state S_t at time t and choose action x_t (according to some rule that we discuss below), we are then going to observe a sample realization of a random variable W_{t+1} (which was unknown at time t). Finally, we are going to use a *transition function* (also known as the state transition model), which gives us the next state using

$$S_{t+1} = S^M(S_t, x_t, W_{t+1}).$$

We are going to simulate our way from time 0 to time T iteratively. To firm up our notation, let n the iteration index. In iteration n, let S_t^n be the state that we visit at time t. Also let ω^n index the sample path of observations of W_t, which means that $W_{t+1}(\omega^n)$ is the sample realization of what we observe between t and $t+1$ while following sample path ω^n.

We have to figure out how we are going to make decisions. The most natural policy is to mimic equation (15.1) and solve

$$x_t^n = \arg\max_x \left(C(S_t^n, x) + \gamma \sum_{s'} p(s'|S_t^n, x) \overline{V}_{t+1}^{n-1}(s') \right). \tag{15.3}$$

Not surprisingly, we are going to refer to this as a pure exploitation policy, because it means to choose the best action based on our current belief about the downstream values captured by $\overline{V}_{t+1}^{n-1}(s')$. Before we start criticizing this policy (as we will), we need to close the loop by mentioning how we are coming up with these value function approximations. First compute

$$\hat{v}_t^n = C(S_t^n, x_t^n) + \gamma \sum_{s'} p(s'|S_t^n, x_t^n) \overline{V}_{t+1}^{n-1}(s'). \tag{15.4}$$

We can think of $\hat{v}_t^n$ as a sample estimate of the value of being in state S_t^n. This is the value that we experienced while we were following our sample path through the states $S_0^n, S_1^n, \ldots, S_T^n$. It would be nice to mimic (15.2) and just set $V_t(S_t^n) = \hat{v}_t^n$,

but $\hat{v}_t^n$ is a noisy sample estimate. For this reason, we have to do smoothing, which we can do using

$$\overline{V}_t^n(S_t^n) = (1-\alpha_{n-1})\overline{V}_t^{n-1}(S_t^n) + \alpha_{n-1}\hat{v}_t^n.$$

Here, α_{n-1} is known as a stepsize, which is a parameter between 0 and 1. It might be a constant (such as $\alpha_{n-1} = .001$), but more often it is assigned a declining sequence such as $\alpha_n = a/(a+n-1)$, where a is a tunable parameter.

Notice that, in the above presentation, we have two time indices, n and t. This is because we considered a finite-horizon problem ending at time T. The index t denotes a stage of the problem, whereas the index n denotes the number of times we have updated our approximation $\bar{V}$ of the value function. The algorithm works by making multiple passes through the problem, so that we solve (15.3) backward for $t = T-1, ..., 1$ for each fixed value of n.

We can also accommodate infinite-horizon problems ($T \rightarrow \infty$), for which the optimality equation (15.2) becomes

$$V(S) = \max_x \big(C(S,x) + \gamma \sum_{s'} p(s'|S,x)V(s')\big). \tag{15.5}$$

for all states S. In this case, the ADP algorithm merges the two time indices into one, such that (15.3) becomes

$$x^n = \arg\max_x \left(C(S^n,x) + \gamma \sum_{s'} p(s'|S^n,x)\overline{V}^{n-1}(s') \right). \tag{15.6}$$

and (15.4) becomes

$$\hat{v}^n = C(S^n,x^n) + \gamma \sum_{s'} p(s'|S^n,x^n)\overline{V}^{n-1}(s'). \tag{15.7}$$

The approximation is updated using

$$\overline{V}^n(S^n) = (1-\alpha_{n-1})\overline{V}^{n-1}(S^n) + \alpha_{n-1}\hat{v}^n. \tag{15.8}$$

We still run the algorithm by progressing forward in time starting from S^0. Now, a single iteration consists of a visit to a single state S^n, and n is our only time index, as in the rest of the book. We hope that, after a large number of iterations, $\overline{V}^n$ will be close to the true solution of (15.5). To simplify our presentation, we will focus on infinite-horizon problems when developing our algorithms in this chapter, though it is important to keep in mind the wealth of finite-horizon applications.

We now have the beginnings of an algorithm that seems to have considerable appeal. We can handle arbitrarily complex state variables S_t, and even the random information W_t can be quite complex (problems have been solved with thousands of dimensions), because at any point in time we are working with a single state and a single sample realization of W_t. It almost seems like magic, so you know that there has to be something wrong. Indeed, the problem is known as the exploration vs. exploitation problem, something that should be now be quite familiar to readers of this book.

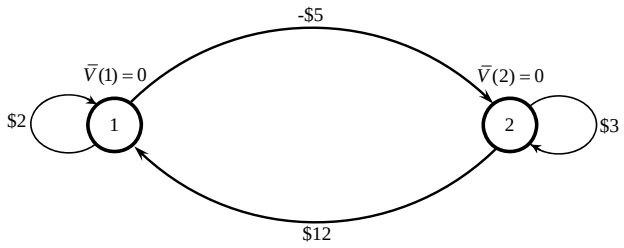

Figure 15.2 A two state dynamic program, where the estimated value of being in each state is currently zero.

15.1.2 The exploration vs. exploitation problem

Unfortunately, the problem with our elementary approximate dynamic programming algorithm is not just that it may not work very well, but that it can, and generally, will, work terribly. To see why, we only need the very simple two-state dynamic program illustrated in Figure 15.2. Assume that we are initially in state 1, and we have to choose an action. We can choose to stay in state 1, in which case we earn a $2 contribution plus the approximate value of being in state 1, where our initial approximation is $\overline{V}(1) = 0$. Alternatively, we can choose to go to state 2, where we incur a loss of $5, but then we receive the value of being in state 2, where again we have an approximate value of $\overline{V}(2) = 0$. Looking at these two alternatives, we would naturally decide to stay in state 1. As a result, we never discover that the transition from state 2 back to state 1 would earn us $12.

This is hardly an artificial example. A more real-world setting is illustrated with an interstate truck driver who arrives in a city and is offered a set of loads of freight. He has to choose one of the loads, which has him moving from his current location to the destination of the load. Let S_t be the current location of the driver (typically a city or three-digit zip code), let x_t be his choice of load from a set of offered loads (these arrive randomly and only become known when he arrives at a location), and let $C(S_t, x)$ be the net profit he receives from accepting the load and moving to its destination. For this simple problem, $S_{t+1} = S^M(S_t, x_t, W_{t+1}) = S^M(S_t, x_t)$ captures the destination of the load (which is deterministic).

A new truck driver might not have any information about the value of being in a city, and as a result might reasonably start with $\overline{V}^0(s) = 1$ for each state s. This means that he would initially take the load paying the most. Over time, he will learn about the value of being in states that he visits, but this will lead to situations where he is choosing between a load that returns him to a city which he has already visited (in which case $\overline{V}^{n-1}(s') > 0$), and a load that takes him to a city he has never visited (which means $\overline{V}^{n-1}(s') = 0$). As a result, there is a natural bias to take loads to cities he has visited before. Sound familiar? This is like returning to restaurants that you know, rather than trying new restaurants. Needless to say, this is fairly intuitive behavior.

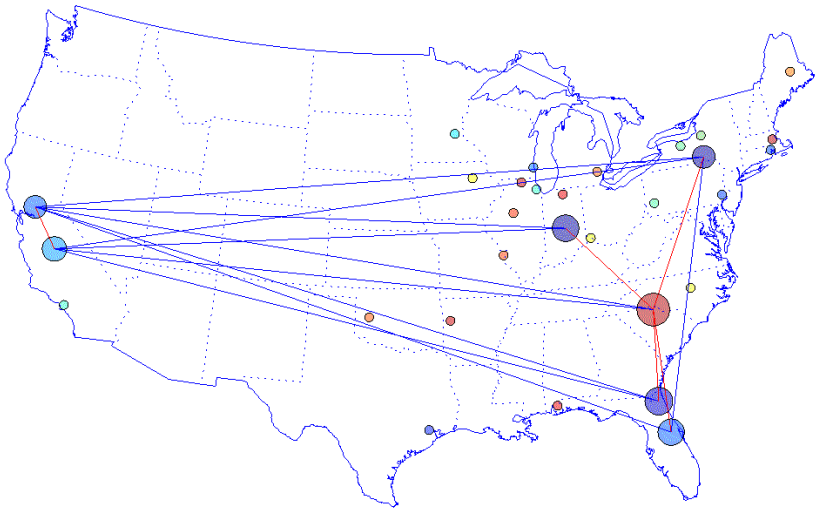

Figure 15.3 The effect of a pure exploitation policy for our nomadic trucker.

The problem with this policy is that, as we would expect, there is a tendency to get caught visiting a small number of cities. Figure 15.3 shows the results of a simulation of this policy, and out of a set of 30 possible locations, the driver finds that he is constantly returning to the same seven cities. This might seem familiar, but it is hardly optimal.

15.1.3 Discussion

Our ADP algorithm has not really solved the curse of dimensionality. While we have eliminated the need to loop over all states to calculate the value of being in a state, we still need good approximations of the value of each state that we *might* visit, as shown in equation (15.3). While this may be much smaller than the size of the full state space, this is still a large number of states.

There are two changes we have to consider to overcome this problem. The first is that we have to introduce *exploration* steps, where we visit a state just to collect information about the state. We have seen this idea throughout this volume, so there should be no surprise that it arises in this setting as well. The second change is that we have to use some form of *generalized learning*. That is, we need to learn more than just the value of being in a single state. Again, this has been a common theme that we have seen before, first through the mechanism of correlated beliefs, and later when we made the transition to parametric belief models as we did in Chapter 7.

First, we present some simple heuristics for exploration. Then, we progress to more formal methods that address both of the issues raised in this discussion. As a rule, the exploration strategies in this chapter are designed for infinite-horizon problems, so we use a single time index n.

15.2 SOME HEURISTIC LEARNING POLICIES

One very simple policy that is frequently used in practice is our old acquaintance epsilon-greedy: at time n, we make a random decision with probability ε, or follow the pure exploitation policy (15.6) with probability $1-\varepsilon$. We face the usual problem of tuning ε, but otherwise, it will be possible to achieve good performance as long as $\mathcal{X}$ is fairly small.

Two more sophisticated exploration strategies developed within the computer science community are known as R-max and E^3. Both of these policies are based on the idea of categorizing the states according to whether we have "enough" or "not enough" information about them.

The R-max policy makes decisions according to the rule

$$X^{Rmax,n}\left(S^n;\alpha^n\right)=\arg\max_x\left(C\left(S^n,x\right)+\gamma\sum_s\rho_s^n\left(S^n,x\right)F\left(s\right)\right)$$

where

$$F\left(s\right)=\begin{cases}R^{\max} & \text{if } s \text{ has been visited fewer than } m \text{ times,}\\ V^n\left(s\right) & \text{if } s \text{ has been visited at least } m \text{ times.}\end{cases}$$

The integer m is a tunable parameter representing the number of times we need to visit a state in order to obtain "enough" information. The value R^{max} is deliberately chosen to be very large, such that we are more likely to choose an action x if it is more likely to lead us to states for which we have little information. Eventually, once we visit every state enough times, R^{max} reduces to the pure exploitation policy . We are still, however, left with the issue of tuning m and R^{max}.

The E^3 policy ("Explicit Explore or Exploit") can be viewed as a modification of epsilon-greedy. As in R-max, we have a parameter m representing the amount of information that is "sufficient" for us to know the value of being in a state. Upon reaching state S^n, our decision is made as follows. If S^n is a state that we have never visited before, we make a random decision. If we have visited S^n before, but fewer than m times, we make the decision that we have tried the fewest number of times among all our previous visits to the state. Finally, if we have visited S^n more than m times, we follow the pure exploitation policy and make our decision according to (15.6). Once again, the policy reduces to pure exploitation once we have sufficiently explored the state space. In the early stages, the policy encourages us to make decisions with which we are unfamiliar, in order to learn about them.

15.3 THE LOCAL BANDIT APPROXIMATION

Our first formal technique, known as the local bandit approximation or LBA, attempts to convert the dynamic program into a multi-armed bandit problem at each time step, and then uses a Gittins-like calculation to make the next decision. The multi-armed bandit problem, which we covered in Chapter 5, has no physical state. We make our decision based purely on our knowledge about the rewards obtainable from M

different arms. To apply bandit-style thinking to our current problem, we need a way to remove the physical state.

Let us start by defining a policy π_n that always makes decisions according to (15.6) for $\bar{V}^{n-1}$ fixed at the current value of n. At first glance, this sounds like the pure exploitation policy. However, pure exploitation updates the approximation $\bar{V}^{n-1}$ using (15.8) after $\hat{v}^n$ is observed. That is, pure exploitation assumes that the approximation is fixed when we make a decision, but we continue to update the approximation in every iteration. This is exactly the concept of experiential learning discussed back in Section **??**.

By contrast, the policy π_n *always* makes decisions according to the *fixed* approximation $\bar{V}^{n-1}$. It is analogous to the "Stop" policy discussed in Section 5.4. If we use this policy, we will commit to our current value function approximation and stop learning altogether. The dynamic program is thus reduced to a Markov chain (Y_n) whose transition probabilities are given by

$$P\left(Y_{n+1} = s' \,|\, Y_n = s\right) = p\left(s' \,|\, s, X^{\pi,n}\left(s\right)\right)$$

where $X^{\pi,n}\left(s\right)$ is the decision that solves (15.6) for $S^n = s$. Our reward for visiting state s is thus $\tilde{C}\left(s\right) = C\left(s, X^{\pi,n}\left(s\right)\right)$.

We use this policy much the same way that we did in Section **??**: we assume that we will make only one more decision before switching to this naive policy. Suppose that S^n is our state at time n. Let us adopt a perspective that is centered around the state S^n. In this time step, we will make a decision x^n and transition to a different state. Suppose that, from that point onward, we will only make decisions according to the policy π_n. This policy will lead us to visit many different states, before we eventually return to S^n. We use the word *sojourn* to describe the period of time in which we travel around states other than S^n. The sojourn ends as soon as we return to this state. Figure 15.4 provides a visual illustration, with states represented by circles and decisions represented by squares.

As long as we use the policy π_n to make decisions, we can study the sojourn via the Markov chain (Y_n). Let

$$\tau_{s,s'} = \min\left\{n \geq 0 \,|\, Y_n = s', Y_0 = s\right\}$$

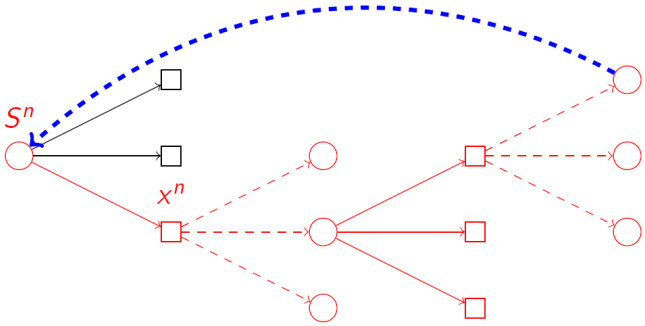

Figure 15.4 If we visit a state S^n and make decision x^n, we enter a sojourn that takes us to other states and actions. The sojourn ends as soon as we return to S^n.

be the number of transitions needed for the Markov chain (Y_n) to reach state s' for the first time, provided that it started in state s. The quantities

$$\begin{aligned} R(s,s') &= \mathbb{E}\left(\sum_{n=0}^{\tau_{s,s'}-1} \gamma^n f(Y_n) \mid Y_0 = s\right) \\ T(s,s') &= \mathbb{E}\left(\sum_{n=0}^{\tau_{s,s'}-1} \gamma^n \mid Y_0 = s\right) \end{aligned}$$

represent the average discount reward collected and the average discounted time elapsed, respectively, before our first visit to state s'. We can compute these quantities using first-transition analysis on Y_n. First,

$$R(s,s') = \begin{cases} f(s) + \gamma \sum_{s'' \in \mathcal{S}} P(Y_{n+1} = s'' \mid Y_n = s) R(s'',s') & \text{if } s \neq s' \\ 0 & \text{if } s = s' \end{cases}.$$

Similarly,

$$T(s,s') = \begin{cases} 1 + \gamma \sum_{s'' \in \mathcal{S}} P(Y_{n+1} = s'' \mid Y_n = s) T(s'',s') & \text{if } s \neq s' \\ 0 & \text{if } s = s' \end{cases}.$$

It follows that, if we visit state S^n and choose action x, the expected reward collected during the ensuing sojourn, and the expected length of the sojourn, are given by

$$\begin{aligned} R^{sojourn}(S^n,x) &= C(S^n,x) + \gamma \sum_{s' \in \mathcal{S}} p(s' \mid S^n,x) R(s',S^n), \\ T^{sojourn}(S^n,x) &= 1 + \gamma \sum_{s' \in \mathcal{S}} p(s' \mid S^n,x) T(s',S^n). \end{aligned}$$

The quantity

$$\nu^{LBA,n}(S^n,x) = \frac{R^{sojourn}(S^n,x)}{T^{sojourn}(S^n,x)} \tag{15.9}$$

represents the expected reward per expected unit time that we receive during the sojourn.

It turns out that the ratio in (15.9) resembles a Gittins index. In Chapter 5, we defined the Gittins index of a bandit arm as a risk-free reward that would make us indifferent between playing the arm repeatedly or just collecting the reward. An equivalent definition describes the Gittins index as the long-term expected reward per play that we can obtain from playing the arm over and over.

Thus, in a sense, this analysis is converting the dynamic program into a bandit problem. When we visit state S^n, we behave as if S^n were the only state in the dynamic program. All other states are grouped together into the sojourn. The different decisions available in state S^n are viewed as "arms" of a bandit, and the expected reward that we collect during the sojourn after pulling the "arm" x is treated as the "reward" of that arm. Thus, (15.9) is viewed as the Gittins index of decision x. We then make decisions according to the simple rule

$$X^{LBA,n}(S^n) = \arg\max_x \nu_x^{LBA,n},$$

analogous to the Gittins index policy for multi-armed bandits.

Unlike the original Gittins index policy, LBA is not optimal, because we are using the policy π_n instead of the optimal policy in the analysis of the sojourn. However, when this approach was first introduced, it was a pioneering example of an attempt to bring optimal learning ideas over to learning with a physical state. It also illustrates some of the difficulties inherent in carrying optimal learning concepts over to the physical state setting.

15.4 THE KNOWLEDGE GRADIENT IN DYNAMIC PROGRAMMING

We are going to develop the concept of the knowledge gradient in the context of dynamic programs with a physical state. We do this in two steps. First, we are going to introduce the idea of approximating a value function using a linear regression model. This makes it possible to generalize what we learn from a visit to a single state to what we know about all states. Then, we show how to compute the knowledge gradient in this setting.

15.4.1 Generalized learning using basis functions

A limitation of all the policies we have reviewed up to now is that they are limited by the use of a lookup table belief model for value functions. A much more powerful way of learning uses some form of generalization. While there are several ways of doing this, by far the most popular involves using linear regression to approximate the value of being in a state.

Before we do this, we have to solve a small technical difficulty. Our pure exploitation policy would have chosen an action using

$$x_t^n = \arg\max_x \left(C(S_t^n, x) + \gamma \sum_{s'} p(s'|S_t^n, x) \overline{V}_{t+1}^{n-1}(s') \right).$$

If we have a large state space (which is the only time where learning is an issue), then we would never be able to compute the one-step transition matrix $p(s'|S_t^n, x_t^n)$. A more natural way to write this equation is using its expectation form, where we would write

$$x_t^n = \arg\max_x \left(C(S_t^n, x) + \gamma \mathbb{E}\{\overline{V}_{t+1}^{n-1}(S'|S_t^n, x\} \right)$$

with S' denoting the randomly determined next state. We have not solved anything yet, because we can only compute the expectation for special cases. We can get around this by using the concept of the *post-decision state variable* which we designate S_t^x. The post-decision state is the state immediately after we have made a decision, but before we have learned any new information. For example, imagine we have a simple inventory problem where at time t we need to serve a random demand $\hat{D}_t$ with some resource, where R_t is the amount of resource (medicine, money, vaccines, water, power plants...) currently available. The (pre-decision) state is

given by $S_t = (R_t, \hat{D}_t)$. Unsatisfied demands are lost. The resource variable evolves according to

$$R_{t+1} = R_t - x_t + \hat{R}_{t+1},$$

where $0 \leq x_t \leq R_t$ is our decision of how much to withdraw to satisfy demands, and $\hat{R}_{t+1}$ represents random additions (blood donation, cash deposits). The post decision state $S_t^x = R_t^x$, where

$$R_t^x = R_t - x_t.$$

The key idea with the post-decision state is that it is a deterministic function of the state S_t and the action x_t. More generally, we assume we have a function $S^{M,x}(S_t, x_t)$ that returns the post-decision state S_t^x.

Using this concept, our exploitation policy would be computed using

$$x_t^n = \arg\max_x \left(C(S_t^n, x) + \gamma \overline{V}_t^{x,n-1}(S^{M,x}(S_t^n, x)) \right), \tag{15.10}$$

where $\overline{V}_t^{x,n-1}(S_t^x)$ is our value function approximation around the post-decision state S_t^x. Note that this is a deterministic optimization problem, which is much easier to solve. Below, we will again focus on the infinite-horizon formulation, in which (15.10) becomes

$$x^n = \arg\max_x \left(C(S^n, x) + \gamma \overline{V}^{x,n-1}(S^{M,x}(S^n, x)) \right), \tag{15.11}$$

We now just have to resolve the problem of approximating the value function. For this purpose, we propose using a simple linear model of the form

$$V(S^{x,n}) = \sum_{f \in \mathcal{F}} \theta_f \phi_f \left(S^{x,n} \right) = (\phi(S^{x,n}))^T \theta.$$

Here, $\phi_f(S)$ is known as a *feature* or *basis function*. These are functions that capture what an expert feels are the important characteristics of the state variable. Instead of having to estimate the value of being in each state, we now have to find the regression vector θ.

In Chapter 7, we showed how we could use the knowledge gradient to estimate the parameters of a linear regression model. We are going to do the same here, but now we have to deal with the issue of a physical state. We are also going to adopt another convention that we have used widely in our presentation which is the idea that we have a prior for our regression model. That is, we are going to assume that we have an initial estimate θ^0 as a starting point that is an unbiased estimate of the true value. This is like saying that we have a rough idea of what the value function looks like. This assumption allows us to claim that $\hat{v}^n$ is an unbiased observation of the true value $V(S^{x,n-1})$ of being in the post-decision state $S^{x,n-1}$. We are going to further assume that the error $\epsilon = \hat{v}^n - V(S^{x,n-1})$, in addition to having mean zero (since we assume that our prior is unbiased) also has a known error with variance σ_ϵ^2.

With this foundation, we can quickly describe the equations needed to update our regression vector θ. We start by defining the matrix

$$X^n = \begin{bmatrix} \phi_1\left(S^{x,0}\right) & \cdots & \phi_F\left(S^{x,0}\right) \\ \vdots & \ddots & \vdots \\ \phi_1\left(S^{x,n-1}\right) & \cdots & \phi_F\left(S^{x,n-1}\right) \end{bmatrix}.$$

This matrix consists of the value of each basis function evaluated at every state that we have visited. Also define the vector of observations

$$y^n = \begin{bmatrix} \hat{v}^1 \\ \vdots \\ \hat{v}^n \end{bmatrix}.$$

If we were using traditional linear regression, we could compute the best regression vector θ^n using

$$\theta^n = \left[\left(X^n\right)^T X^n\right]^{-1} \left(X^n\right)^T y^n.$$

We can compute θ^n recursively without having to perform matrix inversion. First let $B^n = \left[\left(X^n\right)^T X^n\right]^{-1}$. The updating equation for θ^n is given by

$$\theta^n = \theta^{n-1} + \frac{\hat{v}^n - \phi\left(S^{x,n-1}\right)^T \theta^{n-1}}{1 + \phi\left(S^{x,n-1}\right)^T B^{n-1} \phi\left(S^{x,n-1}\right)} B^{n-1} \phi\left(S^{x,n-1}\right). \quad (15.12)$$

We can also compute the matrix B^n recursively using

$$B^n = B^{n-1} - \frac{B^{n-1}\phi\left(S^{x,n-1}\right)\phi\left(S^{x,n-1}\right)^T B^{n-1}}{1 + \phi\left(S^{x,n-1}\right)^T B^{n-1}\phi\left(S^{x,n-1}\right)}. \quad (15.13)$$

We are going to make the somewhat heroic assumption that the observations $\hat{v}^n$ are independent and identically distributed with variance σ_ϵ^2. This means that the covariance matrix for the observation vector Y^n is $I\sigma_\epsilon^2$, where I is the identity matrix. Our estimate of the regression vector is given by

$$\theta^n = B^n (X^n)^T Y^n.$$

That is, our approximation of the value of state $S^{x,n}$ at time n is $\bar{V}^n\left(S^n\right) = \phi\left(S^{x,n}\right)^T \theta^n$. It is also possible to show (as we did in Chapter 7) that the covariance matrix for θ is given by

$$\Sigma^{\theta,n} = \sigma_\epsilon^2 B^n.$$

We can now rewrite equation (15.12) as

$$\theta^n = \theta^{n-1} - \frac{\hat{v}^n - \phi\left(S^{x,n-1}\right)^T \theta^{n-1}}{\sigma_\varepsilon^2 + \phi\left(S^{x,n-1}\right)^T \Sigma^{\theta,n-1}\phi\left(S^{x,n-1}\right)} \Sigma^{\theta,n-1}\phi\left(S^{x,n-1}\right) \quad (15.14)$$

and

$$\Sigma^{\theta,n} = \Sigma^{\theta,n-1} - \frac{\Sigma^{\theta,n-1}\phi\left(S^{x,n-1}\right)\phi\left(S^{x,n-1}\right)^T \Sigma^{\theta,n-1}}{\sigma_\varepsilon^2 + \phi\left(S^{x,n-1}\right)^T \Sigma^{\theta,n-1}\phi\left(S^{x,n-1}\right)}. \quad (15.15)$$

15.4.2 The knowledge gradient

We are now ready to compute a knowledge gradient for each potential action in a dynamic program. We are going to build on the principles we have developed earlier in this volume, but care has to be used in the presence of a physical state.

We start by writing our exploitation policy (doing the best given what we know now) in the form

$$X^{Exp,n}\left(S^n\right) = \arg\max_x Q^n(S^n, x) \tag{15.16}$$

where

$$Q^n(S^n, x) = C(S^n, x) + \gamma\phi\left(S^{x,n}\right)^T \theta^n.$$

When we use the knowledge gradient, we capture the fact that our decision will generate information (presumably from an exogenous distribution) that would allow us to update our regression vector, giving us an updated estimate θ^{n+1} (which is random at time n, when we choose our action). We can write the knowledge gradient policy as

$$X^{KG,n}\left(S^n\right) = \arg\max_x Q^{KG,n}\left(S^n, x\right) \tag{15.17}$$

where

$$Q^{KG,n}(S^n, x) = C(S^n, x) + \gamma\mathbb{E}_x^n \overline{V}^{n+1}(S^{x,n}). \tag{15.18}$$

To understand this, it is useful to talk through the steps. If we are in state S^n and choose action x, we would then observe a random quantity W^{n+1} that would determine the next pre-decision state S^{n+1}. Once in state S^{n+1}, we would observe $\hat{v}^{n+1}$, which would then be used to update our parameter vector θ^n to give us an updated estimate θ^{n+1}. However, we are still in state S^n and W^{n+1} (and therefore S^{n+1} and $\hat{v}^{n+1}$) is a random variable. For this reason, we have to take the expectation $\mathbb{E}_x^n$ of $\overline{V}^{n+1}$ given what we know now (given by S^n, which captures both our physical state as well as our belief state), and the action x that we are considering.

Note that while we are in state S^n, the information we gain from a transition from S^n to S^{n+1} from taking action x updates our belief about the value function that may benefit us even though our next decision is from state S^{n+1} (which is random when we are still in state S^n). We did not enjoy this property when we used a lookup table representation. If we learned something about state s' from a decision to move from s to s', this new information was of little or no value while we were in state s'. The information did not add value until we were in some other state where we might consider a transition back to state s. We could apply the same principle using a lookup table representation if we exploited the idea of correlated beliefs.

We start by expanding equation (15.18) using

$$\begin{aligned} Q^{KG,n}(S^n, x) = \; & C(S^n, x) \\ & +\gamma \sum_{S^{n+1}} p(S^{n+1}|S^n, x)\mathbb{E}_x^n \max_{x'} Q^{n+1}(S^{n+1}, x') \end{aligned} \tag{15.19}$$

To compute the expectation in (15.19) we build on ideas we first presented in Chapter 4 and write θ^{n+1} using

$$\theta^{n+1} \sim \theta^n + \frac{\Sigma^{\theta,n+1}\phi(S^{x,n})}{\sqrt{\sigma_\varepsilon^2 + \phi(S^{x,n})^T\Sigma^{\theta,n}\phi(S^{x,n})}}Z, \tag{15.20}$$

where Z is a standard normal random variable with mean 0 and variance 1. Multiplying both sides in (15.20) by $\phi(S^y)$, for some decision y not necessarily equal to x, gives us the relationship between $\bar{V}^n$ and $\bar{V}^{n+1}$ which we can write as

$$\bar{V}^{n+1}(S^y) \sim \bar{V}^n(S^y) + \frac{\phi(S^y)^T\Sigma^{\theta,n-1}\phi(S^{x,n})}{\sqrt{\sigma^2_\varepsilon + \phi(S^{x,n})^T\Sigma^{\theta,n}\phi(S^{x,n})}}Z. \tag{15.21}$$

The quantity $\phi(S^y)^T\Sigma^{\theta,n-1}\phi(S^{x,n})$ is precisely $Cov^n(S^y, S^{x,n})$.

Using equation (15.21), the last term in (15.19) can be written as

$$\mathbb{E}^n_x \max_{x'} Q^{n+1}(S^{n+1}, x') = \mathbb{E} \max_{x'} (a^n_{x'} + b^n_{x'} Z) \tag{15.22}$$

where

$$\begin{aligned} a^n_{x'} &= C(S^{n+1}, x') + \gamma \bar{V}^n(S^{M,x}(S^{n+1}, x')), \\ b^n_{x'} &= \gamma \frac{\phi(S^{M,x}(S^{n+1}, x'))^T\Sigma^{\theta,n-1}\phi(S^{x,n})}{\sqrt{\sigma^2_\varepsilon + \phi(S^{x,n})^T\Sigma^{\theta,n}\phi(S^{x,n})}}. \end{aligned}$$

Now we have to use the methods for computing the knowledge gradient with correlated beliefs (see Section 4.5) to compute (15.22) using

$$\mathbb{E} \max_{x'}(a^n_{x'} + b^n_{x'} Z) = \left(\max_{x'} a^n_{x'}\right) + \sum_{x'} (b^n_{x'+1} - b^n_{x'}) f(-|c_{x'}|),$$

where f is defined as $f(z) = z\Phi(z) + \phi(z)$, where ϕ and Φ denote the standard Gaussian pdf and cdf. Readers familiar with Section 4.5 will remember that it is important that the actions be sorted so that the slopes b^n_y are sorted in increasing order, and that dominated lines have been eliminated.

We can now compute the knowledge gradient using

$$\begin{aligned} \nu^{KG,n}(S^{x,n}, S^{n+1}) &= \mathbb{E} \max_{x'} (a^n_{x'} + b^n_{x'} Z) - \max_{x'} a^n_{x'} \\ &= \sum_{x'} \left(b^n_{x'+1} - b^n_{x'}\right) f(-|c_{x'}|). \end{aligned}$$

Since $a^n_{x'} = Q^n(S^{n+1}, x')$, the quantity $\nu^{KG,n}(S^{x,n}, S^{n+1})$ can be viewed as the expected improvement in our estimate of the value of being in state S^{n+1}, obtained as a result of making the random transition from $S^{x,n}$ (which depends on our action) to S^{n+1}. We can substitute the definition of the knowledge gradient back into (15.19) to obtain

$$\begin{aligned} &\sum_{S^{n+1}} p\left(S^{n+1}|S^n, x\right) \mathbb{E}^n_x \max_{x'} Q^{n+1}\left(S^{n+1}, x'\right) \\ &\quad = \sum_{S^{n+1}} p\left(S^{n+1}|S^n, x\right) \max_{x'} Q^n\left(S^{n+1}, x'\right) \\ &\qquad + \sum_{S^{n+1}} p\left(S^{n+1}|S^n, x\right) \nu^{KG,n}\left(S^{x,n}, S^{n+1}\right). \end{aligned}$$

The value of being in the post-decision state is the expected value of being in the next pre-decision state. We can write this expectation using

$$\sum_{S^{n+1}} p(S^{n+1}|S^n, x) \max_{x'} Q^n\left(S^{n+1}, x'\right) \approx \bar{V}^n\left(S^{x,n}\right)$$

and (15.17) reduces to

$$X^{KG,n}(S^n) = \max_x C(S^n,x) + \gamma \bar{V}^n(S^{x,n}) + \sum_{S^{n+1}} p(S^{n+1}|S^n,x)\nu^{KG,n}(S^{x,n},S^{n+1}). \quad (15.23)$$

We see that our knowledge gradient policy looks very similar to the pure exploitation policy in equation (15.16), with the exception that we now have one more term that captures the value of information.

One potential problem with equation (15.23) is that we may not be able to compute the conditional probability $p(S^{n+1}|S^n,x)$, especially if the random variables are continuous. In this case, a quick work-around is to to approximate the expectation in the value of information term using Monte Carlo simulation. In this case, we would use

$$\sum_{S^{n+1}} p\left(S^{n+1}|S^n,x\right)\nu^{KG,n}\left(S^{x,n},S^{n+1}\right) \approx \frac{1}{K}\sum_{k=1}^{K}\nu^{KG,n}\left(S^{x,n},S_k^{n+1}\right),$$

where $S_k^{n+1} = S^{M,W}\left(S^{x,n},W^{n+1}(\omega_k)\right)$ for the kth sample path generated. This technique works well for relatively low values of K, e.g. $K \approx 20$, as long as we have some sort of simulation model from which we can generate transitions.

The structure of the policy in (15.23) is suited for online learning, as we saw in Section 5.4. If we are trying to learn the value functions as quickly as possible in an offline setting, we would focus purely on the value of information, giving us a policy of the form

$$X^{Off,n}(S^n) = \arg\max_x \sum_{S^{n+1}} p\left(S^{n+1}|S^n,x\right)\nu^{KG,n}\left(S^{x,n},S^{n+1}\right). \quad (15.24)$$

This policy will look odd to readers familiar with approximate dynamic programming, since it appears to ignore the contribution from an action and the downstream value. However, in an offline setting, our goal is to just learn the value function approximation, with the hope that if we can learn this function quickly, then a pure exploitation policy (holding the value function fixed) will return good results.

15.4.3 Experiments

Experience with this use of the knowledge gradient is quite limited, but we report on experimental work that is available as this book went to press. We are going to use these ideas to optimize the performance of an energy storage device such as a battery. Let R^n be the amount of energy in the battery as a percentage of total capacity, which is taken to be 35 megawatt-hours. We allow R^n to take integer values between 0 and 100. We can buy energy on the spot market at a price P^n which evolves according to the model

$$\log\frac{P^{n+1}}{P^n} = -\alpha\log\frac{P^n}{P^0} + \sigma Z^{n+1} \quad (15.25)$$

where Z, as before, is a standard normal random variable with mean 0 and variance 1. We use the values $\alpha = 0.0633$ and $\sigma = 0.2$, along with an initial price $P^0 = 30$.

The decision x^n is an integer from -50 to 50 representing how much to charge ($x^n \geq 0$) or discharge ($x^n < 0$) the storage device. The decisions x^n are constrained so that R^n does not go negative or above the maximum capacity of the battery. The post-decision state is given by

$$\begin{aligned} R^{x,n} &= R^n + x^n, \\ P^{x,n} &= P^n. \end{aligned}$$

The next pre-decision state is obtained by letting $R^{n+1} = R^{x,n}$ (we assume no exogenous changes to the amount stored in the battery) and generating P^{n+1} using equation (15.25). The single-period reward is given by $C(S^n, x^n) = -P^n x^n$, the cost incurred or revenue obtained as a result of our decision. Our goal is to determine a storage and withdrawal policy that maximizes total discounted profits over time, using a discount factor of $\gamma = .99$.

The spot price P^n is continuous, and thus we cannot solve the problem exactly. We use a parametric value function approximation with six polynomial basis functions given by $\phi\left(S^{x,n}\right) = \left(1, R^{x.n}, \left(R^{x,n}\right)^2, P^{x,n}, \left(P^{x,n}\right)^2, R^{x,n}P^{x,n}\right)$. We run the KG policy with the Monte Carlo approximation from equation (15.24), with $K = 20$.

For each sample path, the algorithm started with a prior $\theta_1^0 = 15000$, $\theta_2^0, ..., \theta_6^0 = 0$ and a diagonal covariance matrix $\Sigma^{\theta,0}$ with all diagonal elements equal to 500^2. An optimistic value for θ^0 was chosen heuristically to reduce the likelihood of getting stuck in a subset of the state space. The observation noise was chosen to be $\sigma_\varepsilon^2 = 2000^2$.

Most experiments with approximate dynamic programming use a series of training iterations to estimate the value function approximation, after which θ is fixed and a series of simulations are run to determine how well the policy works. This is classical off-line learning, and yet it is most common to use the exploitation policy given in equation (15.10) which has more of an online structure. For this reason, we report the results of four comparisons: we are going to use the online policy $X^{KG,n}(S^n)$ in equation (15.23) and the offline policy $X^{Off,n}(S^n)$ in equation (15.24), tested in two different objective functions. The first is an online objective function, where we assume that we are accumulating contributions in a real setting as we are learning the value function. The second is a more traditional offline setting, where we use a budget to fit the value function approximation and then evaluate the policy using a series of testing iterations. Naturally, we expect the online policy $X^{KG,n}(S^n)$ to do better on the online objective function, and we expect the offline policy $X^{Off,n}(S^n)$ to do better on the offline objective function.

Table 15.1 compares our two learning policies using basis functions against an algorithm that uses a lookup table representation, and an epsilon-greedy learning policy that we introduced in Section 3.2.3. The average performance of each algorithm, averaged over several hundred problems, shows that as we hoped, the offline KG policy (using either basis functions or a lookup table representation) does better on the offline objective function, and the online KG policy does better on the online objective function. Interestingly, the basis functions worked best for the offline objective function while the lookup table worked best for the online objective function. The epsilon-greedy policy performed poorly on both objective functions.

Table 15.1 Means and standard errors for the storage problem.

	Offline objective		Online objective	
	Mean	Avg. SE	Mean	Avg. SE
Offline KG (basis functions)	1136.20	3.54	-342.23	19.96
Online KG (basis functions)	871.13	3.15	44.58	27.71
Offline KG (lookup)	210.43	0.33	-277.38	15.90
Online KG (lookup)	79.36	0.23	160.28	5.43
Epsilon-greedy (param.)	-475.54	2.30	-329.03	25.31

A separate but important issue involves the computational effort to support a policy. For this problem, we can safely say that the epsilon-greedy policy requires virtually no computational effort, and for this reason could perhaps be run for more iterations. The knowledge gradient policy when using a lookup table representation required 0.025 seconds for each calculation. The KG policy when using basis functions required .205 seconds to determine the best action which is small, but not negligible. In a true online problem, these execution times are probably negligible (perhaps we are making decisions once a day or once an hour). However, in many approximate dynamic programming applications, this may be considered a fairly significant amount of overhead. The value of the policy would have to be judged based on the degree to which it accelerated convergence.

One final note of caution has to do with the basis function model itself. If the vector ϕ of basis functions is poorly chosen – for instance, if the features do not adequately describe the value function, or if V is nonlinear in the features – then ADP can perform very poorly. The literature has found cases in which the parameters θ^n will never converge, regardless of how long we run the algorithm. This issue has nothing to do with our particular choice of exploration strategy, but rather is intrinsic to the linear model. Notwithstanding, basis functions continue to be a very popular algorithmic strategy in ADP, due to their remarkable ease of use. In the above model, performance can often be vastly improved by tuning σ_ε^2 or the starting covariance matrix $\Sigma^{\theta,0}$. Unfortunately, we are not able to avoid tunable parameters altogether in our ADP algorithm, the way we did in Chapter 4. It is important to bear in mind, however, that other policies such as epsilon-greedy would add even more tunable parameters if we were to use them together with basis functions.

15.5 AN EXPECTED IMPROVEMENT POLICY

As in Sections **??** and 14.3, our version of KG for learning with a physical state also has a close relative in the form of an expected improvement policy. This time, EI appears under the name "value of perfect information" or VPI. The difference in name is due to the fact that this method was developed independently in the computer science community. Recalling that

$$Q^n(S^n, x) = C(S^n, x) + \gamma\phi\left(S^{x,n}\right)^T \theta^n,$$

and defining

$$\sigma_x^{2,n} = \phi\left(S^{x,n}\right)^T \Sigma^{\theta,n} \phi\left(S^{x,n}\right),$$

we calculate

$$\nu^{VPI,n}\left(S^{x,n}\right) = \sigma_x^{2,n} f\left(-\frac{\left|\phi\left(S^{x,n}\right)^T \theta^n - \max_{x' \neq x} \phi\left(S^{x',n}\right)^T \theta^n\right|}{\sigma_x^{2,n}}\right).$$

We then make our decision according to the formula

$$X^{VPI,n}\left(S^n\right) = \arg\max_x C(S^n, x) + \gamma V^n(S^{x,n}) + \nu^{VPI,n}(S^{x,n}).$$

Just as the other variants of EI, this policy implicitly assumes that we will learn the true value of $S^{x,n}$ immediately after choosing the decision x. Furthermore, just like the KG policy, VPI makes a decision by solving Bellman's equation, plus a value of information term.

15.6 BIBLIOGRAPHIC NOTES

In addition to the models discussed in this chapter, there is a separate stream of literature within the computer science community on learning unknown transition probabilities in a Markov decision process. This particular learning problem uses the Dirichlet distribution that we briefly touched on in Section 2.6 to model the unknown probabilities; the resulting algorithms tend to have a high degree of computational complexity. The LBA method of Duff & Barto (1996) also grew out of this literature. An example of an early approach is Silver (1963). See e.g. Dearden et al. (1999), Steele (2000) and Duff (2003) for additional work on this topic.

Section 15.1 - We give a very streamlined introduction to ADP based on Chapters 3-4 of Powell (2011). Other good references include Puterman (1994), Bertsekas & Tsitsiklis (1996), and Si et al. (2005).

Section 3.2.3 - The R-max method was developed by Brafman & Tennenholtz (2003), whereas E^3 is due to Kearns & Singh (2002). Our heuristics from Chapter 3 can also be used in this setting; implementation is described e.g. in Sutton & Barto (1998) and Kaelbling (1993).

Section 15.3 - The LBA policy was originally proposed by Duff & Barto (1996) as a conceptual algorithm. The full implementation was worked out by Ryzhov et al. (2010), where some experimental results are given. The expression of the Gittins index as expected reward per expected unit time is due to Katehakis & Veinott (1987); see also Dupacova (1995).

Section 15.4 - Basis functions are a standard model for generalized learning; see e.g. Tesauro (1992) for an early treatment. The model continues to see widespread use; see Sutton et al. (2009) for some recent advances. The development of the knowledge gradient for dynamic programming is based on Ryzhov & Powell (2011*b*). The first