CHAPTER 1

The Challenges of Dynamic Programming

The optimization of problems over time arises in many settings, ranging from the control of heating systems to managing entire economies. In between are examples including landing aircraft, purchasing new equipment, managing blood inventories, scheduling fleets of vehicles, selling assets, investing money in portfolios, and just playing a game of tic-tac-toe or backgammon. These problems involve making decisions, then observing information, after which we make more decisions, and then more information, and so on. Known as *sequential decision problems*, they can be straightforward (if subtle) to formulate, but solving them is another matter.

Dynamic programming has its roots in several fields. Engineering and economics tend to focus on problems with continuous states and decisions (these communities refer to decisions as controls), which might be quantities such as location, speed, and temperature. By contrast, the fields of operations research and artificial intelligence work primarily with discrete states and decisions (or actions). Problems that are modeled with continuous states and decisions (and typically in continuous time) are often addressed under the umbrella of "control theory," whereas problems with discrete states and decisions, modeled in discrete time, are studied at length under the umbrella of "Markov decision processes." Both of these subfields set up recursive equations that depend on the use of a state variable to capture history in a compact way. There are many high-dimensional problems such as those involving the allocation of resources that are generally studied using the tools of mathematical programming. Most of this work focuses on deterministic problems using tools such as linear, nonlinear, or integer programming, but there is a subfield known as stochastic programming that incorporates uncertainty. Our presentation spans all of these fields.

Approximate Dynamic Programming: Solving the Curses of Dimensionality, Second Edition.
Warren B. Powell.

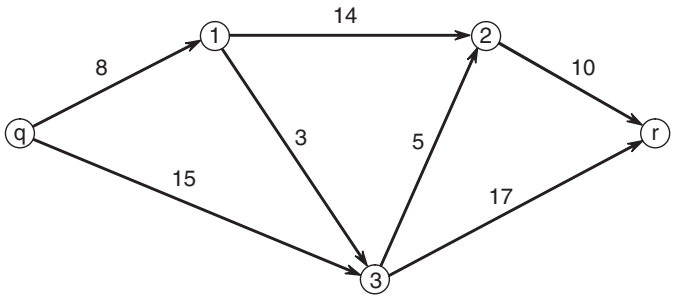

Figure 1.1 Illustration of a shortest path problem from origin q to destination r.

1.1 A DYNAMIC PROGRAMMING EXAMPLE: A SHORTEST PATH PROBLEM

Perhaps one of the best-known applications of dynamic programming is that faced by a driver choosing a path in a transportation network. For simplicity (and this is a real simplification for this application), we assume that the driver has to decide at each node (or intersection) which link to traverse next (we are not going to get into the challenges of left turns versus right turns). Let $\mathcal{I}$ be the set of intersections. If the driver is at intersection i, he can go to a subset of intersections $\mathcal{I}_i^+$ at a cost c_{ij}. He starts at the origin node $q \in \mathcal{I}$ and has to find his way to the destination node $r \in \mathcal{I}$ at the least cost. An illustration is shown in Figure 1.1.

The problem can be easily solved using dynamic programming. Let

$$v_i = \text{Cost to get from intersection } i \in \mathcal{I} \text{ to the destination node } r.$$

We assume that $v_r = 0$. Initially we do not know v_i, and so we start by setting $v_i = M$, where "M" is known as "big M" and represents a large number. We can solve the problem by iteratively computing

$$v_i \leftarrow \min \left\{ v_i, \min_{j \in \mathcal{I}^+} \{c_{ij} + v_j\} \right\} \qquad \text{for all } i \in \mathcal{I}. \tag{1.1}$$

Equation (1.1) has to be solved iteratively, where at each iteration, we loop over all the nodes i in the network. We stop when none of the values v_i change. It should be noted that this is not a very efficient way of solving a shortest path problem. For example, in the early iterations it may well be the case that $v_j = M$ for all $j \in \mathcal{I}^+$. However, we use the method to illustrate dynamic programming.

Table 1.1 illustrates the algorithm, assuming that we always traverse the nodes in the order (q, 1, 2, 3, r). Note that we handle node 2 before node 3, which is the reason why, even in the first pass, we learn that the path cost from node 3 to node r is 15 (rather than 17). We are done after iteration 3, but we require iteration 4 to verify that nothing has changed.

Shortest path problems arise in a variety of settings that have nothing to do with transportation or networks. Consider, for example, the challenge faced by a college

Table 1.1 Path cost from each node to node r after each node has been visited

	Cost from Node				
Iteration	q	1	2	3	r
	100	100	100	100	0
1	100	100	10	15	0
2	30	18	10	15	0
3	26	18	10	15	0
4	26	18	10	15	0

freshman trying to plan her schedule up to graduation. By graduation, she must take 32 courses overall, including eight departmentals, two math courses, one science course, and two language courses. We can describe the state of her academic program in terms of how many courses she has taken under each of these five categories. Let S_{tc} be the number of courses she has taken by the end of semester t in category $c = \{\text{Total courses, Departmentals, Math, Science, Language}\}$, and let $S_t = (S_{tc})_c$ be the state vector. Based on this state, she has to decide which courses to take in the next semester. To graduate, she has to reach the state $S_8 = (32, 8, 2, 1, 2)$. We assume that she has a measurable desirability for each course she takes, and that she would like to maximize the total desirability of all her courses.

The problem can be viewed as a shortest path problem from the state $S_0 = (0,\ 0,\ 0,\ 0,\ 0)$ to $S_8 = (32,\ 8,\ 2,\ 1,\ 2)$. Let S_t be her current state at the beginning of semester t, and let a_t represent the decisions she makes while determining what courses to take. We then assume we have access to a *transition function* $S^M(S_t,\ a_t)$, which tells us that if she is in state S_t and takes action a_t, she will land in state S_{t+1}, which we represent by simply using

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

In our transportation problem, we would have $S_t = i$ if we are at intersection i, and a_t would be the decision to "go to j," leaving us in the state $S_{t+1} = j$.

Finally, let $C_t(S_t,\ a_t)$ be the contribution or reward she generates from being in state S_t and taking the action a_t. The value of being in state S_t is defined by the equation

$$V_t(S_t) = \max_{x_t} \{C_t(S_t,\ a_t) + V_{t+1}(S_{t+1})\} \qquad \forall s_t \in \mathcal{S}_t,$$

where $S_{t+1} = S^M(S_t, a_t)$ and where $\mathcal{S}_t$ is the set of all possible (discrete) states that she can be in at the beginning of the year.

1.2 THE THREE CURSES OF DIMENSIONALITY

All dynamic programs can be written in terms of a recursion that relates the value of being in a particular state at one point in time to the value of the states that we

are carried into at the next point in time. For deterministic problems this equation can be written

$$V_t(S_t) = \max_{a_t} \big(C_t(S_t,\ a_t) + V_{t+1}(S_{t+1})\big). \tag{1.2}$$

where S_{t+1} is the state we transition to if we are currently in state S_t and take action a_t. Equation (1.2) is known as Bellman's equation, or the Hamilton–Jacobi equation, or increasingly, the Hamilton–Jacobi–Bellman equation (HJB for short). Some textbooks (in control theory) refer to them as the "functional equation" of dynamic programming (or the "recurrence equation"). We primarily use the term "optimality equation" in our presentation, but often use the term "Bellman equation" because this is so widely used in the dynamic programming community.

Most of the problems that we address in this volume involve some form of uncertainty (prices, travel times, equipment failures, weather). For example, in a simple inventory problem we might have S_t DVD players in stock. We might then order a_t new DVD players, after which we satisfy a random demand $\hat{D}_{t+1}$ that follows some probability distribution. The state variable would be described by the transition equation

$$S_{t+1} = \max\{0,\ S_t + a_t - \hat{D}_{t+1}\}.$$

Assume that $C_t(S_t, a_t)$ is the contribution we earn at time t, given by

$$C_t(S_t,\ a_t,\ \hat{D}_{t+1}) = p_t \min\{S_t + a_t,\ \hat{D}_{t+1}\} - ca_t.$$

To find the best decision, we need to maximize the contribution we receive from a_t plus the expected value of the state that we end up at (which is random). That means we need to solve

$$V_t(S_t) = \max_{a_t} \mathbb{E}\{C_t(S_t,\ a_t,\ \hat{D}_{t+1}) + V_{t+1}(S_{t+1})|S_t\}. \tag{1.3}$$

This problem is not too hard to solve. Assume that we know $V_{t+1}(S_{t+1})$ for each state S_{t+1}. We just have to compute (1.3) for each value of S_t, which then gives us $V_t(S_t)$. We can keep stepping backward in time to compute all the value functions.

For the vast majority of problems the state of the system is a vector. For example, if we have to track the inventory of N different products, where we might have 0, 1, $\ldots$, $M-1$ units of inventory of each product, then we would have M^N different states. As we can see, the size of the state space grows very quickly as the number of dimensions grows. This is the widely known "curse of dimensionality" of dynamic programming and is the most often-cited reason why dynamic programming cannot be used.

In fact, there are many applications where there are three curses of dimensionality. Consider the problem of managing blood inventories. There are eight blood types (AB+, AB−, A+, A−, B+, B−, O+, O−), which means we have eight types of blood supplies and eight types of blood demands. Let R_{ti} be the supply of blood type i at time t ($i = 1, 2, \ldots, 8$), and let D_{ti} be the demand for blood type

i at time t. Our state variable is given by $S_t = (R_t,\ D_t)$, where $R_t = (R_{ti})_{i=1}^{8}$ (D_t is defined similarly).

In each time period there are two types of randomness: random blood donations and random demands. Let $\hat{R}_{ti}$ be the random new donations of blood of type i in week t, and let $\hat{D}_{ti}$ be the random new demands for blood of type i in week t. We are going to let $W_t = (\hat{R}_t,\ \hat{D}_t)$ be the vector of random information (new supplies and demands) that becomes known in week t.

Finally, let x_{tij} be the amount of blood type i used to satisfy a demand for blood of type j. We switch to "x" for the action because this is the standard notation used in the field of mathematical programming for solving vector-valued decision problems. There are rules that govern what blood types can substitute for different demand types, shown in Figure 1.2.

We can quickly see that S_t and W_t have 16 dimensions each. If we have up to 100 units of blood of any type, then our state space has $100^{16} = 10^{32}$ states. If we have up to 20 units of blood being donated or needed in any week, then W_t has $20^{16} = 6.55 \times 10^{20}$ outcomes. We would need to evaluate 16 nested summations to evaluate the expectation. Finally, x_t has 27 dimensions (there are 27 feasible substitutions of blood types for demand types). Needless to say, evaluating all possible values of x_t is completely intractable.

This problem illustrates what is, for many applications, the three curses of dimensionality:

1. *State space*. If the state variable $S_t = (S_{t1}, S_{t2}, \ldots, S_{ti}, \ldots, S_{tI})$ has I dimensions, and if S_{ti} can take on L possible values, then we might have up to L^I different states.

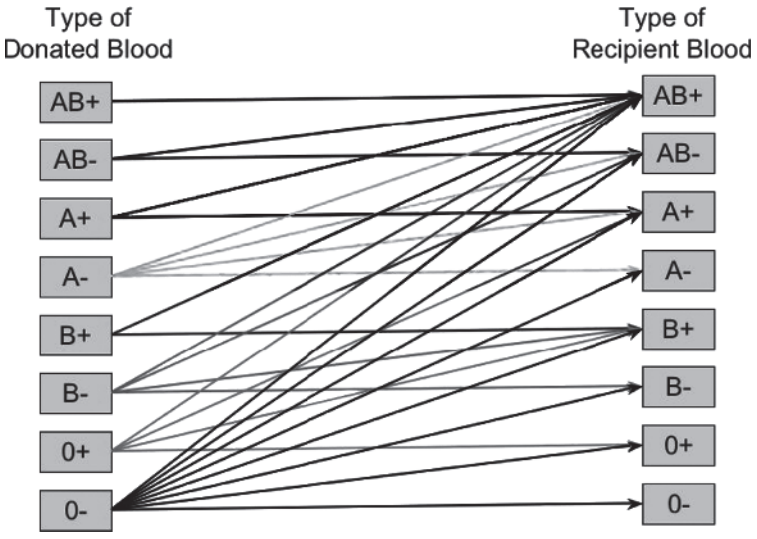

Figure 1.2 Different substitution possibilities between donated blood and patient types (from Cant, 2006).

2. *Outcome space*. The random variable $W_t = (W_{t1}, W_{t2}, \ldots, W_{tj}, \ldots, W_{tJ})$ might have J dimensions. If W_{tj} can take on M outcomes, then our outcome space might take on up to M^J outcomes.
3. *Action space*. The decision vector $x_t = (x_{t1}, x_{t2}, \ldots, x_{tk}, \ldots, x_{tK})$ might have K dimensions. If x_{tk} can take on N outcomes, then we might have up to N^K outcomes. In the language of math programming, we refer to the action space as the *feasible region*, and we may assume that the vector x_t is discrete (integer) or continuous.

By the time we get to Chapter 14, we will be able to produce high-quality, implementable solutions not just to the blood problem (see Section 14.2) but for problems that are far larger. The techniques that we are going to describe have produced production quality solutions to plan the operations of some of the largest transportation companies in the country. These problems require state variables with millions of dimensions, with very complex dynamics. We will show that these same algorithms converge to optimal solutions for special cases. For these problems we will produce solutions that are within 1 percent of optimality in a small fraction of the time required to find the optimal solution using classical techniques. However, we will also describe algorithms for problems with unknown convergence properties that produce solutions of uncertain quality and with behaviors that can range from the frustrating to the mystifying. This is a very young field.

Not all problems suffer from the three curses of dimensionality. Many problems have small sets of actions (do we buy or sell?), easily computable expectations (did a customer arrive or not?), and small state spaces (the nodes of a network). The field of dynamic programming has identified many problems, some with genuine industrial applications, that avoid the curses of dimensionality. Our goal is to provide guidance for the problems that do not satisfy some or all of these convenient properties.

1.3 SOME REAL APPLICATIONS

Our experiences using approximate dynamic programming have been driven by problems in transportation with decision vectors with thousands or tens of thousands of dimensions, and state variables with thousands to millions of dimensions. We have solved energy problems with 175,000 time periods. Our applications have spanned applications in air force, finance, and health.

As our energy environment changes, we have to plan new energy resources (such as the wind turbines in Figure 1.3). A challenging dynamic problem requires determining when to acquire or install new energy technologies (wind turbines, solar panels, energy storage using flywheels or compressed air, hydrogen cars) and how to operate them. These decisions have to be made when considering uncertainty in the demand, prices, and the underlying technologies for creating, storing, and using energy. For example, adding ethanol capacity has to include the possibility that oil prices will drop (reducing the demand for ethanol) or that government regulations may favor alternative fuels (increasing the demand).

Figure 1.3 Wind turbines are one form of alternative energy resources (from http://www.nrel.gov/data/pix/searchpix.cgi).

An example of a very complex resource allocation problem arises in railroads (Figure 1.4). In North America there are six major railroads (known as "Class I" railroads) that operate thousands of locomotives, many of which cost over $1 million. Decisions have to be made now to assign locomotives to trains, taking into account how the locomotives will be used at the destination. For example, a train may be going to a location that needs additional power. Or a locomotive might have to be routed to a maintenance facility, and the destination of a train may or may not offer good opportunities for getting the locomotive to the shop. There are many types of locomotives, and different types of locomotives are suited to different types of trains (e.g., trains moving coal, grain, or merchandise). Other applications of dynamic programming include the management of freight cars, where decisions about when, where, and how many to move have to be made in the presence of numerous sources of uncertainty, including customer demands, transit times, and equipment problems.

The military faces a broad range of operational challenges that require positioning resources to anticipate future demands. The problem may be figuring out when and where to position tankers for mid-air refueling (Figure 1.5), or whether

Figure 1.4 Major railroads in the United States have to manage complex assets such as boxcars, locomotives and the people who operate them. Courtesy Norfolk Southern.

a cargo aircraft should be modified to carry passengers. The air mobility command needs to think about not only what aircraft is best to move a particular load of freight but also the value of aircraft in the future (are there repair facilities near the destination?). The military is further interested in the value of more reliable aircraft and the impact of last-minute requests. Dynamic programming provides a means to produce robust decisions, allowing the military to respond to last-minute requests.

Managing the electric power grid requires evaluating the reliability of equipment such as the transformers that convert high-voltage power to the voltages used by homes and businesses. Figure 1.6 shows the high-voltage backbone network managed by PJM Interconnections that provides electricity to the northeastern United States. To ensure the reliability of the grid, PJM helps utilities maintain an appropriate inventory of spare transformers. They cost five million dollars each, weigh over 200 tons, and require at least a year to deliver. We must make decisions about how many to buy, how fast they should be delivered (fast delivery costs more), and where to put them when they do arrive. If a transformer fails, the electric power grid may have to purchase power from more expensive utilities to avoid a bottleneck, possibly costing millions of dollars per month. As a result it is not possible to wait until problems happen. Utilities also face the problem of pricing their energy in a dynamic market, and purchasing commodities such as coal and natural gas in the presence of fluctuating prices.

Figure 1.5 Mid-air refueling is a major challenge for air operations, requiring that tankers be positioned in anticipation of future needs (from http://www.amc.af.mil/photos/).

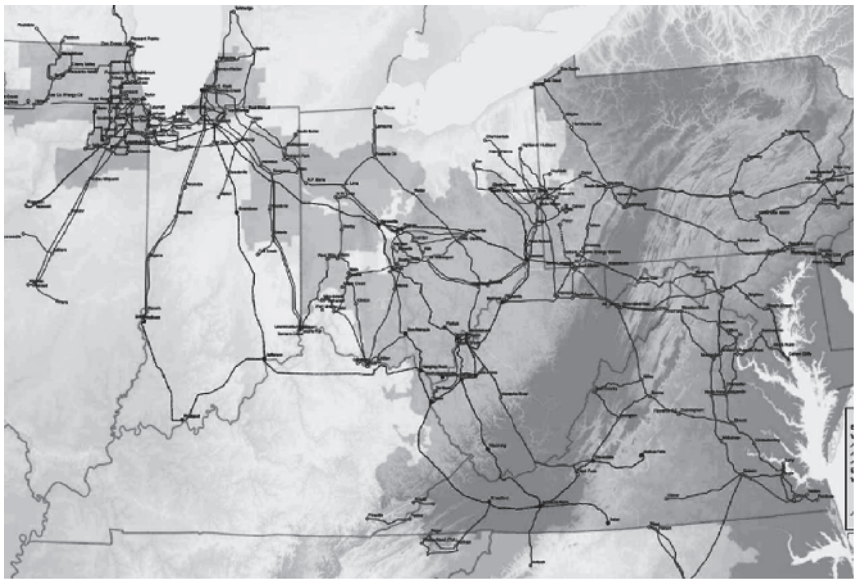

Figure 1.6 High-voltage backbone network managed by PJM Interconnections provides electricity to the northeastern United States. Courtesy PJM Interconnections.

Similar issues arise in the truckload motor carrier industry, where drivers are assigned to move loads that arise in a highly dynamic environment. Large companies manage fleets of thousands of drivers, and the challenge at any moment in time is to find the best driver (Figure 1.7 is from Schneider National, the largest truckload carrier in the United States). There is much more to the problem than

Figure 1.7 Schneider National, the largest truckload motor carrier in the United States, manages a fleet of over 15,000 drivers. Courtesy Schneider National.

simply finding the closest driver; each driver is characterized by attributes such as his or her home location and equipment type as well as his or her skill level and experience. There is a need to balance decisions that maximize profits now versus those that produce good long-run behavior. Approximate dynamic programming produced the first accurate model of a large truckload operation. Modeling this large-scale problem produces some of the advances described in this volume.

Challenging dynamic programs can be found in much simpler settings. A good example involves optimizing the amount of cash held in a mutual fund, which is a function of current market performance (should more money be invested?) and interest rates, illustrated in Figure 1.8. While this problem can be modeled with just three dimensions, the lack of structure and need to discretize at a fine level produced a very challenging optimization problem. Other applications include portfolio allocation problems and determining asset valuations that depend on portfolios of assets.

A third problem class is the acquisition of information. Consider the problem faced by the government that is interested in researching a new technology such as fuel cells or converting coal to hydrogen. There may be dozens of avenues to pursue, and the challenge is to determine the projects in which the government should invest. The state of the system is the set of estimates of how well different components of the technology work. The government funds research to collect information. The result of the research may be the anticipated improvement, or the

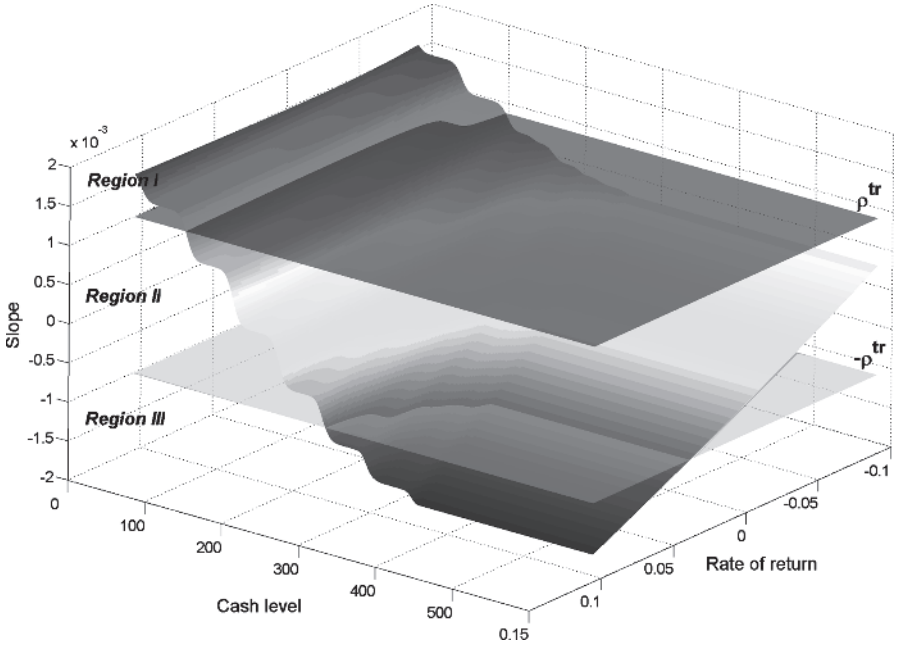

Figure 1.8 Value of holding cash in a mutual fund as a function of market performance and interest rates.

results may be disappointing. The government wants to plan a research program to maximize the likelihood that a successful technology is developed within a reasonable time frame (e.g., 20 years). Depending on time and budget constraints, the government may wish to fund competing technologies in the event that one does not work. Alternatively, it may be more effective to fund one promising technology and then switch to an alternative if the first does not work out.

1.4 PROBLEM CLASSES

Most of the problems that we use as examples in this book can be described as involving the management of physical, financial, or informational resources. Sometimes we use the term "assets," which carries the connotation of money or valuable resources (aircraft, real estate, energy commodities). But in some settings, even these terms may seem inappropriate, for example, training computers to play a game such as tic-tac-toe, where it will be more natural to think in terms of managing an "entity." Regardless of the term, there are a number of major problem classes we consider in our presentation:

Budgeting. Here we face the problem of allocating a fixed resource over a set of activities that incurs costs that are a function of how much we invest in the activity. For example, drug companies have to decide how much to

invest in different research projects or how much to spend on advertising for different drugs. Oil exploration companies have to decide how much to spend exploring potential sources of oil. Political candidates have to decide how much time to spend campaigning in different states.

Asset acquisition with concave costs. A company can raise capital by issuing stock or floating a bond. There are costs associated with these financial instruments independent of how much money is being raised. Similarly an oil company purchasing oil will be given quantity discounts (or it may face the fixed cost of purchasing a tanker-load of oil). Retail outlets get a discount if they purchase a truckload of an item. All of these are instances of acquiring assets with a concave (or, more generally, nonconvex) cost function, which means there is an incentive for purchasing larger quantities.

Asset acquisition with lagged information processes. We can purchase commodity futures that allow us to purchase a product in the future at a lower cost. Alternatively, we may place an order for memory chips from a factory in southeast Asia with one- to two-week delivery times. A transportation company has to provide containers for a shipper who may make requests several days in advance or at the last minute. All of these are asset acquisition problems with *lagged information processes*.

Buying/selling an asset. In this problem class the process stops when we either buy an asset when it looks sufficiently attractive or sell an asset when market conditions warrant. The game ends when the transaction is made. For these problems we tend to focus on the price (the purchase price or the sales price), and our success depends on our ability to trade off current value with future price expectations.

General resource allocation problems. This class encompasses the problem of managing reusable and substitutable resources over time (equipment, people, products, commodities). Applications abound in transportation and logistics. Railroads have to move locomotives and boxcars to serve different activities (moving trains, moving freight) over time. An airline has to move aircraft and pilots in order to move passengers. Consumer goods have to move through warehouses to retailers to satisfy customer demands.

Demand management. There are many applications where we focus on managing the demands being placed on a process. Should a hospital admit a patient? Should a trucking company accept a request by a customer to move a load of freight?

Storage problems. We face problems determining how much energy to store in a water reservoir or battery, how much cash to hold in a mutual fund, how many vaccines to order or blood to hold, and how much inventory to keep. These are all examples of "storage" problems, which may involve one, several or many types of resources, and where these decisions have to be made in the context of state-of-the-world variables such as commodity prices, interest rates, and weather.

Table 1.2 Major problem classes

Number of Entities	Attributes	
	Simple	Complex
Single	Single, simple entity	Single, complex entity
Multiple	Multiple, simple entities	Multiple, complex entities

Shortest paths. In this problem class we typically focus on managing a single, discrete entity. The entity may be someone playing a game, a truck driver we are trying to route to return him home, a driver who is trying to find the best path to his destination, or a locomotive we are trying to route to its maintenance shop. Shortest path problems, however, also represent a general mathematical structure that applies to a broad range of dynamic programs.

Dynamic assignment. Consider the problem of managing multiple entities, such as computer programmers, to perform different tasks over time (writing code or fixing bugs). Each entity and task is characterized by a set of attributes that determines the cost (or contribution) from assigning a particular resource to a particular task.

All these problems focus on the problem of managing physical or financial resources (or assets, or entities). It is useful to think of four major problem classes (depicted in Table 1.2) in terms of whether we are managing a single or multiple entities (e.g., one robot or a fleet of trucks), and whether the entities are simple (an entity may be described by its location on a network) or complex (an entity may be a truck driver described by a 10-dimensional vector of attributes).

If we are managing a single, simple entity, then this is a problem that can be solved exactly using classical algorithms described in Chapter 3. The problem of managing a single, complex entity (e.g., playing backgammon) is commonly studied in computer science under the umbrella of reinforcement learning, or in the engineering community under the umbrella of control theory (e.g., landing an aircraft). The problem of managing multiple, simple entities is widely studied in the field of operations research (managing fleets of vehicles or distribution systems), although this work most commonly focuses on deterministic models. By the end of this book, we are going to show the reader how to handle (approximately) problem classes that include multiple, complex entities, in the presence of different forms of uncertainty.

There is a wide range of problems in dynamic programming that involve controlling resources, where decisions directly involve transforming resources (purchasing inventory, moving robots, controlling the flow of water from reservoirs), but there are other important types of controls. Some examples include:

Pricing. Often the question being asked is, What price should be paid for an asset? The right price for an asset depends on how it is managed, so it should not be surprising that we often find asset prices as a by-product from determining how to best manage the asset.

Information collection. Since we are modeling sequential information and decision processes, we explicitly capture the information that is available when we make a decision, allowing us to undertake studies that change the information process. For example, the military uses unmanned aerial vehicles (UAVs) to collect information about targets in a military setting. Oil companies drill holes to collect information about underground geologic formations. Travelers try different routes to collect information about travel times. Pharmaceutical companies use test markets to experiment with different pricing and advertising strategies.

Technology switching. The last class of questions addresses the underlying technology that controls how the physical process evolves over time. For example, when should a power company upgrade a generating plant (e.g., to burn oil and natural gas)? Should an airline switch to aircraft that fly faster or more efficiently? How much should a communications company invest in a technology given the likelihood that better technology will be available in a few years?

Most of these problems entail both discrete and continuous states and actions. Continuous models would be used for money, for physical products such as oil, grain, and coal, or for discrete products that occur in large volume (most consumer products). In other settings, it is important to retain the integrality of the resources being managed (people, aircraft, locomotives, trucks, and expensive items that come in small quantities). For example, how do we position emergency response units around the country to respond to emergencies (bioterrorism, major oil spills, failure of certain components in the electric power grid)?

What makes these problems hard? With enough assumptions, none of these problems are inherently difficult. But in real applications, a variety of issues emerge that can make all of them intractable. These include:

- *Evolving information processes*. We have to make decisions now before we know the information that will arrive later. This is the essence of stochastic models, and this property quickly turns the easiest problems into computational nightmares.
- *High-dimensional problems*. Most problems are easy if they are small enough. In real applications, there can be many types of resources, producing decision vectors of tremendous size.
- *Measurement problems*. Normally we assume that we look at the state of our system and from this determine what decision to make. In many problems we cannot measure the state of our system precisely. The problem may be delayed information (stock prices), incorrectly reported information (the truck is in the wrong location), misreporting (a manager does not properly add up his total sales), theft (retail inventory), or deception (an equipment manager underreports his equipment so it will not be taken from him).
- *Unknown models (information, system dynamics)*. We can anticipate the future by being able to say something about what might happen (even if it is with

uncertainty) or the effect of a decision (which requires a model of how the system evolves over time).

- *Missing information*. There may be costs that simply cannot be computed and that are instead ignored. The result is a consistent model bias (although we do not know when it arises).
- *Comparing solutions*. Primarily as a result of uncertainty, it can be difficult comparing two solutions to determine which is better. Should we be better on average, or are we interested in the best and worst solution? Do we have enough information to draw a firm conclusion?

1.5 THE MANY DIALECTS OF DYNAMIC PROGRAMMING

Dynamic programming arises from the study of sequential decision processes. Not surprisingly, these arise in a wide range of applications. While we do not wish to take anything from Bellman's fundamental contribution, the optimality equations are, to be quite honest, somewhat obvious. As a result they were discovered independently by the different communities in which these problems arise.

The problems arise in a variety of engineering problems, typically in continuous time with continuous control parameters. These applications gave rise to what is now referred to as control theory. While uncertainty is a major issue in these problems, the formulations tend to focus on deterministic problems (the uncertainty is typically in the estimation of the state or the parameters that govern the system). Economists adopted control theory for a variety of problems involving the control of activities from allocating single budgets or managing entire economies (admittedly at a very simplistic level). Operations research (through Bellman's work) did the most to advance the theory of controlling stochastic problems, thereby producing the very rich theory of Markov decision processes. Computer scientists, especially those working in the realm of artificial intelligence, found that dynamic programming was a useful framework for approaching certain classes of machine learning problems known as reinforcement learning.

As different communities discovered the same concepts and algorithms, they invented their own vocabularies to go with them. As a result we can solve the Bellman equations, the Hamiltonian, the Jacobian, the Hamilton–Jacobian, or the all-purpose Hamilton–Jacobian–Bellman equations (typically referred to as the HJB equations). In our presentation we prefer the term "optimality equations," but "Bellman" has become a part of the language, imbedded in algorithmic strategies such as minimizing the "Bellman error" and "Bellman residual minimization."

There is an even richer vocabulary for the types of algorithms that are the focal point of this book. Everyone has discovered that the backward recursions required to solve the optimality equations in Section 1.1 do not work if the state variable is multidimensional. For example, instead of visiting node i in a network, we might visit state $S_t = (S_{t1}, S_{t2}, \ldots, S_{tB})$, where S_{tb} is the amount of blood on hand of type b. A variety of authors have independently discovered that an alternative strategy is to step forward through time, using iterative algorithms to help estimate

the value function. This general strategy has been referred to as forward dynamic programming, incremental dynamic programming, iterative dynamic programming, adaptive dynamic programming, heuristic dynamic programming, reinforcement learning, and neuro-dynamic programming. The term that is being increasingly adopted is *approximate dynamic programming*, although perhaps it is convenient that the initials, ADP, apply equally well to "adaptive dynamic programming."

The different communities have each evolved their own vocabularies and notational systems. The notation developed for Markov decision processes, and then adopted by the computer science community in the field of reinforcement learning, uses state S and action a. In control theory, it is state x and control u. The field of stochastic programming uses x for decisions. At first, it is tempting to view these as different words for the same quantities, but the cosmetic differences in vocabulary and notation tend to hide more fundamental differences in the nature of the problems being addressed by each community. In reinforcement learning, there is typically a small number of discrete actions. In control theory, u is usually a low-dimensional continuous vector. In operations research, it is not unusual for x to have hundreds or thousands of dimensions.

An unusual characteristic of the reinforcement learning community is their habit of naming algorithms after their notation. Algorithms such as Q-learning (named from the use of Q-factors), TD(λ), ϵ-greedy exploration, and SARSA (which stands for state-action–reward–state-action), are some of the best examples. As a result care has to be taken when designing a notational system.

The field continues to be characterized by a plethora of terms that often mean the same thing. The transition function (which models the evolution of a system over time) is also known as the system model, transfer function, state model, and plant model. The behavior policy is the same as the sampling policy, and a stepsize is also known as the learning rate or the gain.

There is a separate community that evolved from the field of deterministic math programming that focuses on problems with high-dimensional decisions. The reinforcement learning community focuses almost exclusively on problems with finite (and fairly small) sets of discrete actions. The control theory community is primarily interested in multidimensional and continuous actions (but not very many dimensions). In operations research it is not unusual to encounter problems where decisions are vectors with thousands of dimensions.

As early as the 1950s the math programming community was trying to introduce uncertainty into mathematical programs. The resulting subcommunity is called stochastic programming and uses a vocabulary that is quite distinct from that of dynamic programming. The relationship between dynamic programming and stochastic programming has not been widely recognized, despite the fact that Markov decision processes are considered standard topics in graduate programs in operations research.

Our treatment will try to bring out the different dialects of dynamic programming, although we will tend toward a particular default vocabulary for important concepts. Students need to be prepared to read books and papers in this field

that will introduce and develop important concepts using a variety of dialects. The challenge is realizing when authors are using different words to say the same thing.

1.6 WHAT IS NEW IN THIS BOOK?

As of this writing, dynamic programming has enjoyed a relatively long history, with many superb books. Within the operations research community, the original text by Bellman (Bellman, 1957) was followed by a sequence of books focusing on the theme of Markov decision processes. Of these, the current high-water mark is *Markov Decision Processes* by Puterman, which played an influential role in the writing of Chapter 3. The first edition appeared in 1994, followed in 2005 by the second edition. The dynamic programming field offers a powerful theoretical foundation, but the algorithms are limited to problems with very low-dimensional state and action spaces.

This book focuses on a field that is coming to be known as *approximate dynamic programming*; it emphasizes modeling and computation for much harder classes of problems. The problems may be hard because they are large (e.g., large state spaces), or because we lack a model of the underlying process that the field of Markov decision processes takes for granted. Two major references preceded the first edition of this volume. *Neuro-dynamic Programming* by Bertsekas and Tsitsiklis was the first book to appear (in 1996) that integrated stochastic approximation theory with the power of statistical learning to approximation value functions, in a rigorous if demanding presentation. *Reinforcement Learning* by Sutton and Barto, published in 1998 (but building on research that began in 1980), presents the strategies of approximate dynamic programming in a very readable format, with an emphasis on the types of applications that are popular in the computer science/artificial intelligence community. Along with these books, the survey of reinforcement learning in Kaelbling et al. (1996) is a major reference.

There is a sister community that goes by the name of *simulation optimization* that has evolved out of the simulation community that needs to select the best from a set of designs. Nice reviews of this literature are given in Fu (2002) and Kim and Nelson (2006). Books on the topic include Gosavi (2003), Chang et al. (2007), and Cao (2007). Simulation optimization is part of a larger community called stochastic search, which is nicely described in the book Spall (2003). As we show later, this field is directly relevant to policy search methods in approximate dynamic programming.

This volume presents approximate dynamic programming with a much stronger emphasis on modeling, with explicit and careful notation to capture the timing of information. We draw heavily on the modeling framework of control theory with its emphasis on transition functions, which easily handle complex problems, rather than transition matrices, which are used heavily in both Bertsekas and Tsitsiklis (1996) and Sutton and Barto (1998). We start with the classical notation of Markov decision processes that is familiar to the reinforcement learning community, but

we build bridges to math programming so that by the end of the book, we are able to solve problems with very high-dimensional decision vectors. For this reason we adopt two notational styles for modeling decisions: a for discrete actions common in the models solved in reinforcement learning, and x for the continuous and sometimes high-dimensional decision vectors common in operations research and math programming. Throughout the book, we use action a as our default notation, but switch to x in the context of applications that require continuous or multidimensional decisions.

Some other important features of this book are as follows:

- We identify the three curses of dimensionality that characterize some dynamic programs, and develop a comprehensive algorithmic strategy for overcoming them.
- We cover problems with discrete action spaces, denoted using a (standard in Markov decision processes and reinforcement learning), and vector-valued decisions, denoted using x (standard in mathematical programming). The book integrates approximate dynamic programming with math programming, making it possible to solve intractably large deterministic or stochastic optimization problems.
- We cover in depth the concept of the post-decision state variable, which plays a central role in our ability to solve problems with vector-valued decisions. The post-decision state offers the potential for dramatically simplifying many ADP algorithms by avoiding the need to compute a one-step transition matrix or otherwise approximate the expectation within Bellman's equation.
- We place considerable attention on the proper modeling of random variables and system dynamics. We feel that it is important to properly model a problem before attempting to solve it.
- The theoretical foundations of this material can be deep and rich, but our presentation is aimed at advanced undergraduate or masters level students with introductory courses in statistics, probability, and for Chapter 14, linear programming. For more advanced students, proofs are provided in "Why does it work" sections. The presentation is aimed primarily at students in engineering interested in taking real, complex problems, developing proper mathematical models, and producing computationally tractable algorithms.
- We identify four fundamental classes of policies (myopic, lookahead, policies based on value function approximations, and policy function approximations), with careful treatments of the last three. An entire chapter is dedicated to policy search methods, and three chapters develop the critical idea of using value function approximations.
- We carefully deal with the challenge of stepsizes, which depend critically on whether the algorithm is based on approximate value iteration (including Q-learning and TD learning) or approximate policy iteration. Optimal stepsize rules are given for each of these two major classes of algorithms.

Our presentation integrates the fields of Markov decision processes, math programming, statistics, and simulation. The use of statistics to estimate value functions dates back to Bellman and Dreyfus (1959). Although the foundations for proving convergence of special classes of these algorithms traces its origins to the seminal paper on stochastic approximation theory (Robbins and Monro, 1951), the use of this theory (in a more modern form) to prove convergence of special classes of approximate dynamic programming algorithms did not occur until 1994 (Tsitsiklis 1994; Jaakkola et al. 1994). The first book to bring these themes together is Bertsekas and Tsitsiklis (1996), which remains a seminal reference for researchers looking to do serious theoretical work.

1.7 PEDAGOGY

The book is roughly organized into four parts. Part I comprises Chapters 1 to 4, which provide a relatively easy introduction using a simple, discrete representation of states. Part II covers modeling, a description of major classes of policies and policy optimization. Part III covers policies based on value function approximations, along with efficient learning. Part IV describes specialized methods for resource allocation problems.

A number of sections are marked with an *. These can all be skipped when first reading the book without loss of continuity. Sections marked with ** are intended only for advanced graduate students with an interest in the theory behind the techniques.

Part I *Introduction to dynamic programming using simple state representations* —In the first four chapters we introduce dynamic programming, using what is known as a "flat" state representation. That is to say, we assume that we can represent states as $s = 1, 2, \ldots$. We avoid many of the rich modeling and algorithmic issues that arise in more realistic problems.

Chapter 1 Here we set the tone for the book, introducing the challenge of the three "curses of dimensionality" that arise in complex systems.

Chapter 2 Dynamic programs are best taught by example. Here we describe three classes of problems: deterministic problems, stochastic problems, and information acquisition problems. Notation is kept simple but precise, and readers see a range of different applications.

Chapter 3 This is an introduction to classic Markov decision processes. While these models and algorithms are typically dismissed because of "the curse of dimensionality," these ideas represent the foundation of the rest of the book. The proofs in the "why does it work" section are particularly elegant and help provide a deep understanding of this material.

Chapter 4 This chapter provides a fairly complete introduction to approximate dynamic programming, but focusing purely on estimating value functions using lookup tables. The material is particularly familiar to the reinforcement learning community. The presentation steps through classic algorithms, starting with Q-learning and SARSA, and then, progressing through real-time

dynamic programming (which assumes you can compute the one-step transition matrix), approximate value iteration using a pre-decision state variable, and finally, approximate value iteration using a post-decision state variable. Along the way the chapter provides a thorough introduction to the concept of the post-decision state variable, and introduces the issue of exploration and exploitation, as well as on-policy and off-policy learning.

Part II *Approximate dynamic programming with policy optimization*—This block introduces modeling, the design of policies, and policy optimization. Policy optimization is the simplest method for making good decisions, but it is generally restricted to relatively simple problems. As such, it makes for a good introduction to ADP before getting into the complexities of designing policies based on value function approximations.

Chapter 5 This chapter on modeling hints at the richness of dynamic problems. To help with assimilating this chapter, we encourage readers to skip sections marked with an * the first time they go through the chapter. It is also useful to reread this chapter from time to time as you are exposed to the rich set of modeling issues that arise in real applications.

Chapter 6 This chapter introduces four fundamental classes of policies: myopic policies, lookahead policies, policies based on value function approximations, and policy function approximations. We note that there are three classes of approximation strategies: lookup table, and parametric and nonparametric models. These fundamental categories appear to cover all the variations of policies that have been suggested.

Chapter 7 There are many problems where the structure of a policy is fairly apparent, but it depends on tunable parameters. Here we introduce the reader to communities that seek to optimize functions of deterministic parameters (which determines the policy) where we depend on noisy evaluations to estimate the performance of the policy. We cover classical stochastic search, add algorithms from the field of simulation optimization, and introduce the idea of the knowledge gradient, which has proved to be a useful general-purpose algorithmic strategy. In the process, the chapter provides an initial introduction to the exploration-exploitation problem for (offline) ranking and selection problems.

Part III *Approximate dynamic programming using value function approximations*—This is the best-known strategy for solving dynamic programs (approximately), and also the most difficult to master. We break this process into three steps, organized into the three chapters below:

Chapter 8 This chapter covers the basics of approximating functions using lookup tables (very briefly), parametric models (primarily linear regression) and a peek into nonparametric methods.

Chapter 9 Let $\overline{V}^{\pi}(s)$ be an approximation of the value of being in state s while following a fixed policy π. We fit this approximation using sample observations $\hat{v}^n$. This chapter focuses on different ways of computing $\hat{v}^n$ for finite and infinite horizon problems, which can then be used in conjunction with the methods in Chapter 8 to find $\overline{V}^{\pi}(s)$.

Chapter 10 The real challenge is estimating the value of a policy while simultaneously searching for better policies. This chapter introduces algorithms based on approximate value iteration (including Q-learning and TD learning) and approximate policy iteration. The discussion covers issues of convergence that arise while one simultaneously tries to estimate and optimize.

Chapter 11 Stepsizes are an often overlooked dimension of approximate dynamic programming. This chapter reviews four classes of stepsizes: deterministic formulas, heuristic stochastic formulas, optimal stepsize rules based on signal processing (ideally suited for policy iteration), and a new optimal stepsize designed purely for approximate value iteration.

Chapter 12 It is well known in the ADP/RL communities that it is sometimes necessary to visit a state in order to learn about the value of being in a state. Chapter 4 introduces this issue, and Chapter 7 returns to the issue again in the context of policy search. Here we address the problem in its full glory, making the transition from pure learning (no physical state) for both online and offline problems, but also from learning in the presence of a physical state.

Part IV *Resource allocation and implementation challenges* — We close with methods that are specifically designed for problems that arise in the context of resource allocation:

Chapter 13 Resource allocation problems have special structure such as concavity (or convexity for minimization problems). This chapter describes a series of approximation techniques that are directly applicable for these problems.

Chapter 14 There are many problems that can be described under the umbrella of "resource allocation" that offer special structure that we can exploit. These problems tend to be high-dimensional, with state variables that can easily have thousands or even millions of dimensions. However, when we combine concavity with the post-decision state variable, we produce algorithms that can handle industrial-strength applications.

Chapter 15 We close with a discussion of a number of more practical issues that arise in the development and testing of ADP algorithms.

This material is best covered in order. Depending on the length of the course and the nature of the class, an instructor may want to skip some sections, or to weave in some of the theoretical material in the "why does it work" sections. Additional material (exercises, solutions, datasets, errata) will be made available over time at the website http://www.castlelab.princeton.edu/adp.htm (this can also be accessed from the CASTLE Lab website at http://www.castlelab.princeton.edu/).

There are two faces of approximate dynamic programming, and we try to present both of them. The first emphasizes models and algorithms, with an emphasis on applications and computation. It is virtually impossible to learn this material without writing software to test and compare algorithms. The other face is a deeply theoretical one that focuses on proofs of convergence and rate of convergence. This material is advanced and accessible primarily to students with training in probability and stochastic processes at an advanced level.

1.8 BIBLIOGRAPHIC NOTES

There have been three major lines of investigation that have contributed to approximate dynamic programming. The first started in operations research with Bellman's seminal text (Bellman, 1957). Numerous books followed using the framework established by Bellman, each making important contributions to the evolution of the field. Selected highlights include Howard (1960), Derman (1970), Ross (1983), and Heyman and Sobel (1984). As of this writing, the best overall treatment of what has become known as the field of Markov decision processes is given in Puterman (2005). However, this work has focused largely on theory, since the field of discrete Markov decision processes has not proved easy to apply, as discrete representations of state spaces suffer from the well-known curse of dimensionality, which restricts this theory to extremely small problems. The use of statistical methods to approximate value functions originated with Bellman, in Bellman and Dreyfus (1959), but little subsequent progress was made within operations research.

The second originated with efforts by computer scientists to get computers to solve problems, starting with the work of Samuel (1959) to train a computer to play checkers, helping to launch the field that would become known in artificial intelligence as reinforcement learning. The real origins of the field lay in the seminal work in psychology initiated by Andy Barto and Richard Sutton (Sutton and Barto, 1981; Barto et al., 1981; Barto and Sutton, 1981). This team made many contributions over the next two decades, leading up to their landmark volume *Reinforcement Learning* (Sutton and Barto, 1998) which has effectively defined this field. Reinforcement learning evolved originally as an intuitive framework for describing human (and animal) behavior, and only later was the connection made with dynamic programming, when computer scientists adopted the notation developed within operations research. For this reason reinforcement learning as practiced by computer scientists and Markov decision processes as practiced by operations research share a common notation, but a very different culture.

The third line of investigation started completely independently under the umbrella of control theory. Instead of Bellman's optimality equation, it was the Hamiltonian or Jacobi equations, which evolved to the Hamilton–Jacobi equations. Aside from different notation, control theorists were motivated by problems of operating physical processes, and as a result focused much more on problems in continuous time, with continuous states and actions. While analytical solutions could be obtained for special cases, it is perhaps not surprising that control theorists quickly developed their own style of approximate dynamic programming, initially called heuristic dynamic programming (Werbos, 1974, 1989, 1992b). It was in this community that the first connection was made between the adaptive learning algorithms of approximate dynamic programming and reinforcement learning, and the field of stochastic approximation theory. The seminal papers that made this connection were Tsitsiklis (1994), Tsitsiklis and van Roy (1997), and the seminal book *Neuro-dynamic Programming* written by Bertsekas and Tsitsiklis (1996). A major breakthrough in control theory was

the recognition that the powerful technology of neural networks (Haykin, 1999) could be a general-purpose tool for approximating both value functions as well as policies. Major contributions were also made within the field of economics, including Rust (1997) and Judd (1998).

While these books have received the greatest visibility, special recognition is due to a series of workshops funded by the National Science Foundation under the leadership of Paul Werbos, some of which have been documented in several edited volumes (Werbos et al., 1990; White and Sofge, 1992; Si et al., 2004). These workshops have played a significant role in bringing different communities together, the effect of which can be found throughout this volume.

Our presentation is primarily written from the perspective of approximate dynamic programming and reinforcement learning as it is practiced in operations research and computer science, but there are substantial contributions to this field that have come from the engineering controls community. The edited volume by White and Sofge, (1992) provides the first comprehensive coverage of ADP/RL, primarily from the perspective of the controls community (there is a single chapter by Andy Barto on reinforcement learning). Bertsekas and Tsitsiklis (1996) is also written largely from a control perspective, although the influence of problems from operations research and artificial intelligence are apparent. A separate and important line of research grew out of the artificial intelligence community, which is nicely summarized in the review by Kaelbling et al. (1996) and the introductory textbook by Sutton and Barto (1998). More recently Si et al. (2004) brought together papers from the engineering controls community, artificial intelligence, and operations research.

Since the first edition of this book appeared in 2007, a number of other important references have appeared. The third edition of *Dynamic Programming and Optimal Control*, volume 2, by Bertsekas contains a lengthy chapter entitled *Approximate Dynamic Programming*, which he updates continuously and which can be downloaded directly from http://web.mit.edu/dimitrib/www/dpchapter.html. Sutton and Barto released a version of their 1998 book to the Internet at http://www.cs.wmich.edu/∼trenary/files/cs5300/RLBook/the-book.html, which makes their classic book easily accessible. Two recent additions to the literature include Busoniu et al. (2010) and Szepesvari (2010), the latter of which is also available as a free download at http://www.morganclaypool.com/doi/abs/10.2200/S00268ED1V01Y201005AIM009. These are more advanced monographs, but both contain recent theoretical and algorithmic developments.

Energy applications represent a growing area of research in approximate dynamic programming. Applications include Löhndorf and Minner (2010) and Powell et al. (2011). An application of ADP to truckload trucking at Schneider National is described in Simao et al. (2009, 2010). The work on Air Force operations is described in Wu et al. (2009). The application to the mutual fund cash balance problem is given in Nascimento and Powell (2010b).