INFORMS Journal on Computing
Vol. 22, No. 1, Winter 2010, pp. 2–17
ISSN 1091-9856 | EISSN 1526-5528 | 10 | 2201 | 0002

DOI 10.1287/ijoc.1090.0349

Feature Article

Merging AI and OR to Solve High-Dimensional Stochastic Optimization Problems Using Approximate Dynamic Programming

Warren B. Powell
Department of Operations Research and Financial Engineering, Princeton University,
Princeton, New Jersey 08544, powell@princeton.edu

We consider the problem of optimizing over time hundreds or thousands of discrete entities that may be characterized by relatively complex attributes, in the presence of different forms of uncertainty. Such problems arise in a range of operational settings such as transportation and logistics, where the entities may be aircraft, locomotives, containers, or people. These problems can be formulated using dynamic programming but encounter the widely cited "curse of dimensionality." Even deterministic formulations of these problems can produce math programs with millions of rows, far beyond anything being solved today. This paper shows how we can combine concepts from artificial intelligence and operations research to produce practical solution methods that scale to industrial-strength problems. Throughout, we emphasize concepts, techniques, and notation from artificial intelligence and operations research to show how these fields can be brought together for complex stochastic, dynamic problems.

Key words: approximate dynamic programming; reinforcement learning; stochastic optimization; machine learning; curse of dimensionality
History: Accepted by Edward Wasil, Area Editor for Feature Articles; received July 2006; revised July 2007, March 2008, September 2008; accepted March 2009. Published online in *Articles in Advance* October 2, 2009.

1. Introduction

Consider the problem of training a computer to play a game such as backgammon. The computer has to recognize the state of the board, evaluate a number of choices, and determine which move will put the player into the best position. The logic has to consider the decisions of the opponent and the roll of the dice. To determine the best decision, the computer has to compute the value of the state resulting from the decision.

Now, consider the problem of managing a fleet of hundreds or thousands of trucks moving goods around the country, where a truck moves an entire load from one location to the next. For each truck, we have to decide which loads to move to maximize profits while achieving other goals such as minimizing empty miles, obeying work rules (which limit how many hours a driver can move each day and week), and getting the driver home on time. Before a company assigns a driver to move a load, it is necessary to think about where the driver will be after moving a load and the prospects for the driver in the future.

Both of these problems represent decisions that have to be made over time, where decisions that are made now need to anticipate the effect on future rewards. Both problems involve forms of uncertainty (the roll of the dice, the phone calls of customers). The first example is a classic problem addressed by the artificial intelligence (AI) community, whereas the second is the type of problem addressed by the operations research (OR) community. Both problems can be formulated compactly as Markov decision processes. If we let S_t be the state of our system at time t (for the case of the board game, t indexes pairs of plays, or the number of plays by a particular player), d_t be the decision (typically represented as action a_t in the AI community), and $C(S_t, d_t)$ be the contribution of making decision d_t when in state S_t, we can find the value of being in state S_t by using Bellman's equation

$$V_t(S_t) = \max_{d_t} \left(C(S_t, d_t) + \sum_{s' \in \mathcal{S}} p_t(s' \mid S_t, d_t) V_{t+1}(s') \right), \quad (1)$$

where $\mathcal{S}$ is the set of possible states and $p_t(s' \mid S_t, d_t)$ is the probability of going to state s' if we are currently in state S_t and make decision d_t. Note that this is the form of Bellman's equation for finite-horizon or nonstationary problems. It is popular in both the

AI community as well as the OR community to focus on steady-state problems, where Bellman's equation would be written as

$$V(s) = \max_d \left(C(s, d) + \sum_{s' \in \mathcal{S}} p(s' \mid S, d) V(s') \right). \quad (2)$$

This form of Bellman's equation is more compact and elegant, but it hides the timing of information, which becomes important as we focus on the modeling of complex problems (an issue that seems more important in OR). Also, it is easy to convert the time-indexed version in Equation (1) to the steady-state version in (2) (one can just drop any time index), but not the other way around. Finally, even for a stationary problem, we may need a time-dependent formulation if our policy depends on the initial starting state.

Both the AI and OR communities may start with Equation (1), but they have evolved very different solution strategies for these problems. The reason reflects differences in the nature of the problems addressed by each community. The AI community often works on problems that could be described as managing a single, complex entity (such as a backgammon board), whereas the OR community is often trying to optimize multiple (sometimes many) entities that are relatively simple. The AI community will sometimes address multientity problems but often in a multiagent framework, where each agent is effectively solving a single-entity problem. The AI community will measure complexity in terms of the size of the state space, whereas the OR community will measure the number of decision variables or the rows (constraints) in a linear program and the number of scenarios (random outcomes) in a stochastic program.

The trucking problem, for example, is described by the number of drivers with a particular set of attributes (such as location) and a set of decisions that covers how to assign a set of drivers to a set of loads. Consider how the problem looks when viewed as a traditional dynamic program. If at time t there are R_t trucks looking at L_t loads, then a decision involves a matching of up to R_t drivers to up to L_t loads. Let $x = (x_{rl})_{r,l}$ be the decision vector where $x_{rl} = 1$ if we assign driver r to load l. A problem with 100 drivers being matched to 100 loads produces a decision vector with 10,000 dimensions (with $100! \approx 10^{157}$ distinct "actions").

The state space is even larger. Assume each driver is described by a vector $a = (a_1, \ldots, a_n)$ of attributes (such as location, equipment type, driver characteristics, etc.). If there are R drivers, each of which can take up to A attributes, then it is possible to show that the size of the state space for our trucking problem is

$$|\mathcal{S}| = \binom{R + A - 1}{A - 1}. \quad (3)$$

Table 1 Size of State Space for a Fleet Management Problem: 10, 100, and 250 Trucks with 10, 100, and 1,000 Attributes

	Truck attributes		
Trucks	10	100	1,000
10	9×10^4	4×10^{13}	3×10^{23}
100	4×10^{12}	5×10^{58}	1×10^{144}
250	1×10^{16}	1×10^{89}	1×10^{270}

Table 1 illustrates how the state space grows as a function of both the number of trucks we are managing and the number of possible attributes each truck can take. We note that real problems can have thousands of trucks (for the very largest companies), with attribute spaces in the millions.

The problem of playing the game or managing a single truck driver can be viewed as single-entity problems (in the language of OR, we might describe our truck driver as a single resource or asset). Managing a fleet of drivers would be a multiple-entity/resource/asset problem (managing multiple games of backgammon at the same time makes little sense). Clearly, increasing the number of entities that we are managing increases the complexity of the problem.

A second dimension that affects the difficulty of the problem is the complexity of each entity. Managing a single truck whose only attribute is location is fairly simple. However, our driver may have other attributes: How many days has he been away from home? How many hours has he been awake and driving this day? How many hours did he drive on each of the last seven days? What type of equipment is he pulling? In realistic problems, a driver might be described by over a dozen attributes. This makes the problem much harder and impossible to solve exactly. In other words, managing a complex driver starts looking like the problem of playing a game such as backgammon or chess.

For the purposes of our discussion, we categorize our problems along two major dimensions: (1) the number of entities and (2) the complexity of the entities being managed. These two dimensions create four major problem classes as shown in Table 2. Single, simple-entity problems are the easiest and can be solved using standard textbook methods for Markov decision processes (see, for example, Puterman 1994). The AI community has accumulated a substantial literature on problems that can be described as a single, complex entity (playing board games, controlling

Table 2 Major Problem Classes

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

robots). The OR community has similarly developed a rich set of tools for problems with multiple, simple entities (managing fleets of vehicles, routing and scheduling problems). The problem of managing multiple, complex entities under uncertainty (e.g., people, machines, transportation equipment) has seen relatively little attention.

In this paper, we are going to show how we can bring together the fields of AI (in the form of machine learning and reinforcement learning) and OR (in the form of math programming) to solve classes of dynamic programs with 10^{100} states or more (using the most common measure of size in dynamic programming) or, equivalently, multistage stochastic linear programs with 10^{10} rows or more (using the most common measure of size in math programming). The solutions are approximate but are supported by a strong theoretical foundation. The point of the presentation is not to present a specific algorithm for a specific problem but rather to illustrate how the techniques of AI and OR can be combined to solve (approximately) problems that would have previously been viewed as intractable by both communities.

We begin in §2 by introducing specific notation for modeling single-entity and multientity problems. We also introduce a specific candidate application, which serves to illustrate the ideas throughout the remainder of this paper. Section 3 provides different forms of Bellman's equations, and introduces the concept of the post-decision state variable, which is central to solving large-scale problems. The remainder of this paper is a tour of the four major problem classes described in Table 2. Section 4 points out that if we have a single, simple entity, the problem can be easily solved using standard techniques from Markov decision processes. Section 5 addresses the problems involving a single, complex entity that are so often the focus of papers in AI. Section 6 turns to problems involving multiple simple entities that are typically addressed in OR. We note that the OR community does not traditionally view dynamic programming as a starting point for this problem class. For both of these problems, we use the same foundation of Markov decision processes that we used for the simplest problems but introduce different approximation techniques suited to the problem class. Section 7 combines the techniques from these communities to produce a practical solution strategy for a problem with multiple, complex entities. We close with §8, which provides a general set of guidelines for designing strategies for solving a wide range of complex, stochastic resource allocation problems. This section provides important guidelines for how to adapt the general strategy described in this paper to a much broader set of problems.

2. Modeling Dynamic Programs

It is common to model dynamic programs using generic notation such as s for a discrete state in a set $\mathcal{S} = \{1, 2, \ldots, s, \ldots, |\mathcal{S}|\}$. This is known in the AI community as a *flat representation*. Although the generality is attractive, this is precisely what makes general dynamic programs impossible to solve. It is far more effective to use a *factored representation* (Boutilier et al. 1999, Guestrin et al. 2003), where we retain the specific structure of the state variable. The term "flat representation" is also used in the AI community to distinguish nonhierarchical from hierarchical representations. Some refer to "unstructured representation" to describe an indexed (flat) state representation, whereas "structured" refers to representations that are factored but not necessarily hierarchical.

For our purposes, we need to introduce more specific modeling elements—in particular, to allow us to distinguish modeling single and multiple entities. In §2.1, we provide the notation for modeling a single entity, and §2.2 provides the notation for handling multiple entities. Our notation represents a compromise between AI, Markov decision processes, math programming, and simulation. A major goal of this presentation is to provide a richer notation that captures the structure of the problem.

We will illustrate the concepts using a number of examples, but we are going to use one particular example as a unifying theme throughout the presentation. This example involves the management of one or more drivers for a trucking company in an industry known as truckload trucking, where a driver picks up an entire load of freight and moves it to its destination before picking up a new load (there is no consolidation of shipments on the vehicle). This problem was our original motivating application and helps to illustrate all four problem classes.

2.1. Modeling a Single-Entity Problem

We are going to deal with both simple and complex entities. When we have a simple entity, it is most natural to represent the states of the entity using

$\mathcal{I}$ = the set of potential states of the entity, given by $(1, 2, \ldots, |\mathcal{I}|)$. We let i and j index elements of $\mathcal{I}$.

If we are making a decision at time t, we might let i_t be the state of our system (entity). When we use this notation, we assume that the set $\mathcal{I}$ is small enough

to enumerate (that is, it might have thousands or tens of thousands of elements, but probably not millions). In our trucking example, i would be a city or region where we might divide the country into 100 or 1,000 locations.

It is common in the AI and Markov decision process communities (see Sutton and Barto 1998, Puterman 1994) to represent a decision as "action a." This can be thought of as a flat representation of decisions. We are going to consider problems (in particular, those with multiple entities) where we are not able to enumerate all the decisions (just as we may not be able to enumerate all the states). We also wish to distinguish between a type of action ("switch the machine to paint blue," "fly to Chicago") and quantity decisions ("sell 500 cars," "fly the plane at 200 miles per hour"). We represent a type of decision using

d = a type of decision that can be used to act on an entity,
$\mathscr{D}$ = the set of different decision types that can be used to act on a single entity.

We always assume that the decision set $\mathscr{D}$ is small enough to enumerate. For our trucking example, a decision d might be to move to a location or to pick up a load (in this case, $\mathscr{D}$ would have to include all possible locations and all possible loads).

A decision generates a contribution (often designated a reward if maximizing or a cost if minimizing), which we define as

$C(i, d)$ = contribution generated by acting on an entity in state i using decision $d \in \mathscr{D}$.

Whenever we represent the state of the system using $i \in \mathscr{I}$, we are using a flat representation. For many applications, it makes more sense to represent the state as a vector. We are going to need to distinguish between the state of a single entity versus the state of a system with multiple entities. The OR community often refers to the "attribute" of an entity (or resource) as it evolves over time. For this reason, we define the state of a single entity as

a = the vector of attributes (these may be categorical, discrete numerical, or continuous numerical)
$= (a_1, a_2, \ldots, a_I)$,
$\mathscr{A}$ = set of possible values that a may take.

For our trucking example, the attribute of a driver might include his location, his home domicile, the type of equipment he is using, the number of days he has been away from home, and his driving history (federal law requires that he keep a record of how many hours he drives on each of the last eight days). The AI community would refer to an element a_i of the vector a as a *feature* of an entity. The OR community sometimes refers to the attribute vector a as a *label*, a term that is adapted from the shortest path algorithms.

If we are solving a single-entity problem (e.g., the game of backgammon), then a_t (the attribute of our entity at time t) is also the state of our system, where we might use S_t as our state variable. Our only reason for introducing new notation for the state of a single entity is our interest in making the transition to planning multiple entities. We also do not make any structural assumptions about the specific nature of an attribute vector; all the attributes could be categorical, with no particular ordering or metric.

Dynamic systems evolve over time. We assume that we are working with a stochastic system in discrete time (defined as the points in time when we make decisions). We let $t = 0$ be "here and now." It is useful to view information as arriving in continuous time, where time interval t is the period between points in time $t-1$ and t. This eliminates any ambiguity surrounding the information available when we make a decision. We model information using the generic notation

W_t = the exogenous information that arrives to the system during time interval t.

The exogenous information W_t will normally be one or more functions (random variables) describing new information that arrived between the decision made at time $t-1$ and the decision made at time t. W_t could be the roll of the dice, the play of an opponent, new customer demands, equipment failures, and so on.

Given the information process, we describe the evolution of our system using the *attribute transition function*

$$a_{t+1} = a^M(a_t, d_t, W_{t+1}). \tag{4}$$

If we represent the state of the system using the flat representation $\mathscr{I}$, we might define the transition function using $i_{t+1} = i^M(i_t, d_t, W_{t+1})$. There are many stochastic problems where the attribute transition function is deterministic. For example, a truck may drive from i to j (deterministically arriving to j), but where the demands to use the truck once it arrives to j are random, or where the cost of transportation is random. In these cases, we can simply drop the argument W_{t+1} and let $a_{t+1} = a^M(a_t, d_t)$ (or $i_{t+1} = i^M(i_t, d_t)$). For algebraic purposes, it is useful to define the indicator function:

$$\delta_{a'}(a, d) = \begin{cases} 1 & \text{if decision } d \in \mathscr{D} \text{ modifies an entity with attribute } a \text{ to attribute } a', \\ 0 & \text{otherwise.} \end{cases}$$

If we use state $i \in \mathscr{I}$, we would write $\delta_j(i, d)$.

2.2. Modeling Multientity Problems

When we are modeling multiple entities (or resources), we define

R_{ti} = the number of entities with attribute (state) i at time t,
$R_t = (R_{ti})_{i \in \mathcal{I}}$.

If we wish to model multiple complex entities, we would use R_{ta} to represent the number of resources with attribute a at time t. To use our trucking example, we might let R_{ti} be the number of trucks in city i, but R_{ta} would be the number of drivers with attribute a. We refer to R_t as the *resource state vector* (using the vocabulary from OR where these problems are most common) because it describes the state of all the resources.

The number of possible values that R_t can take is given by Equation (3), which, as we showed earlier, grows very quickly with both the number of resources and the size of the attribute space that describes the resources.

We always view d as a decision acting on a single entity. For multiple entities, we represent decisions using

x_{tad} = the number of entities with attribute a that we act on with a decision of type $d \in \mathcal{D}$,
$x_t = (x_{tad})_{a \in \mathcal{A},\, d \in \mathcal{D}}$.

Whereas d is a type of decision that may be defined using categorical descriptions (move an aircraft to Chicago, clean the truck, or perform a machine setup), x_{tad} is a numerical value. Furthermore, a decision d_t is always applied to a single entity, whereas x_t is a vector that describes decisions being applied to multiple entities. If we are managing discrete entities (R_t is integer), then x_t needs to be integer. Decisions have to be made subject to the following constraints:

$$\sum_{d \in \mathcal{D}} x_{tad} = R_{ta}, \tag{5}$$

$$x_{tad} \geq 0. \tag{6}$$

For specific applications, there may be other constraints. We denote the feasible region by $\mathcal{X}_t$, is the set of all x_t that satisfy constraints (5) and (6). Other constraints can be added as needed, and hence, we simply refer to $\mathcal{X}_t$ as the feasible region.

As with the single-entity case, we let W_t represent the information arriving during time interval t. This information can cover new demands, travel delays, equipment failures, and unexpected costs. The resource transition function can be written as

$$R_{t+1} = R^M(R_t, x_t, W_{t+1}). \tag{7}$$

To illustrate with a simple example, imagine that R_t is a scalar quantity giving the cash held by a mutual fund at time t. Between $t-1$ and t, we have random deposits $\hat{R}_t^D$ and withdrawals $\hat{R}_t^W$, giving us the information process $W_t = (\hat{R}_t^D, \hat{R}_t^W)$. The mutual fund can then subtract from or add to this cash level by making or liquidating investments. R_t is governed by the inventory equation

$$R_{t+1} = \max\{0, R_t + \hat{R}_{t+1}^D - \hat{R}_{t+1}^W + x_t\},$$

where $x_t > 0$ means we are liquidating investments, and $x_t < 0$ means we are making investments. This is the resource transition function for this problem.

In many problems, R_t (as a scalar or a vector) captures the state of the system. However, it is often the case that there is other information available when we make a decision. For example, a traveler driving over a network might arrive at a node i and then observe random costs $\hat{c}_{ij}$ on each link (i, j) out of node i. His "resource state" would be node i, but $\hat{c}$ represents information he can use to make his decision. To capture all the information that is needed to make a decision, we define a state variable S_t, which we represent generically using

$$S_t = (R_t, \rho_t),$$

where R_t is the "resource state" and ρ_t represents (for the moment) "other information" (such as $\hat{c}$) that is needed to make a decision. ρ_t is sometimes referred to as the state of the environment. We assume that we are given a set of equations that describes how S_t evolves over time, which is represented using the state transition function

$$S_{t+1} = S^M(S_t, x_t, W_{t+1}). \tag{8}$$

In some communities, $S^M(\cdot)$ is known as the "plant model" (literally, the model of a physical production plant) or "system model" (hence, our use of the superscript M), but we feel that "transition function" is the most descriptive. We use $S^M(\cdot)$ to describe the transition function for the full state vector. Similarly, $R^M(\cdot)$ describes the transition function for the vector R_t, and $a^M(\cdot)$ represents the transition function for a single entity with attribute a_t.

We make decisions using a decision function or *policy* that defines the action x_t we are going to take given that we are in state S_t. The decision function is represented using

$X_t^\pi(S_t)$ = a decision function that returns a feasible vector $x_t \in \mathcal{X}_t$ given the state S_t,

where $\mathcal{X}_t$ is the feasible region for x_t. We let Π be our family of policies (decision functions). Our contribution is given by

$C(S_t, x_t)$ = the contribution returned by making decision x_t when we are in state S_t.

Now our optimization problem can be written as

$$\max_{\pi\in\Pi} \mathbb{E}\sum_{t=0}^{T} \gamma^t C(S_t, X_t^\pi(S_t)), \tag{9}$$

where T is the planning horizon and γ is a discount factor. The evolution of S_t is governed by (8).

2.3. Notation and Problem Structure

Perhaps one of the biggest challenges when looking at research in another community is understanding the implicit assumptions being made about problem scale and structure. In our presentation, we assume that $\mathcal{I}$ can be enumerated. The attribute vector a (which describes the state of a single entity) may have several—or several dozen—dimensions, but not thousands (or more). An attribute may be discrete (categorical or numeric) or continuous (but most of our presentation will assume it is discrete). As a result, we assume that $\mathcal{A}$ cannot be enumerated. We also assume that $\mathcal{D}$ is "not too large" (that is, we can enumerate all the elements in $\mathcal{D}$), but the set $\mathcal{X}$ is too large to enumerate. If we have a single simple entity, we can enumerate $\mathcal{I}$, but if we have multiple simple entities, $R_t = (R_{ti})_{i\in\mathcal{I}}$ is a vector that (we assume) is too large to enumerate.

3. Solving the Problem

Solving the optimization problem in Equation (9) directly is usually computationally intractable. As a result, a number of computational strategies have evolved, spanning dynamic programming (Puterman 1994), stochastic programming (Birge and Louveaux 1997), stochastic search and control (Spall 2003), and optimization of simulation (Chang et al. 2007). Our interest is primarily in the line of research devoted to finding approximate solutions to a dynamic programming model. This research has evolved under names such as neurodynamic programming (Bertsekas and Tsitsiklis 1996), reinforcement learning (Sutton and Barto 1998), and approximate dynamic programming (Si et al. 2004, Powell 2007).

We start with Bellman's equations in §3.1 and review the three curses of dimensionality that plague its solution in §3.2. We then introduce a powerful concept known as the post-decision state variable in §3.3, which sets up an algorithmic strategy that scales to very large problems.

3.1. Bellman's Equation

The most common representation of Bellman's equation is given in Equation (1), when a decision is a discrete, scalar quantity. However, we state it again where the decision is potentially a high-dimensional vector x_t, giving us

$$V_t(S_t) = \max_{x_t\in\mathcal{X}_t}\left(C(S_t, x_t) + \gamma\sum_{s'\in\mathcal{S}} p_t(s' \mid S_t, x_t)V_{t+1}(s')\right), \tag{10}$$

where $\mathcal{S}$ is our discrete state space, and $p_t(s' \mid S_t, x_t)$ is the one-step transition matrix giving the probability that $S_{t+1} = s'$, given that we are in state S_t and take action x_t. We refer to Equation (10) as the *standard form* of Bellman's equation because it is so widely used in textbooks on Markov decision processes (see Bellman 1957, Howard 1971, Puterman 1994).

For a finite horizon problem, the standard way of solving (10) is to assume that we are given $V_{T+1}(S_{T+1})$ and then compute $V_t(S_t)$ from $V_{t+1}(S_{t+1})$ by stepping backward in time. This strategy requires finding $V_t(S_t)$ for all $S_t \in \mathcal{S}$. If S_t is a scalar, this is usually very easy. However, if it is a vector, we encounter the classic "curse of dimensionality" that is so widely cited as a reason why we cannot use dynamic programming.

For our purposes, it is useful to adopt an equivalent but slightly different form of Bellman's equation. Instead of using the one-step transition matrix, we recognize that this is being used to compute an expectation, allowing us to write Bellman's equation as

$$V_t(S_t) = \max_{x_t\in\mathcal{X}_t}(C(S_t, x_t) + \gamma\mathbb{E}\{V_{t+1}(S^M(S_t, x_t, W_{t+1})) \mid S_t\}). \tag{11}$$

We refer to (11) as the *expectation form* of Bellman's equation.

3.2. The Three Curses of Dimensionality

Regardless of whether we use the standard form or expectation form of Bellman's equation, finding $V_t(S_t)$ for each (presumably discrete) value of S_t is computationally intractable when S_t is a vector with more than a few dimensions. For problems in OR, our challenge is actually much harder. There are, in fact, three curses of dimensionality. The first is the state variable, where we might have to describe the number of resources (e.g., people, equipment, facilities, product) of different types at a point in time. If we are modeling simple resources such as trucks, we might have a resource state vector $R_t = (R_{ti})_{i\in\mathcal{I}}$ with a hundred dimensions or more. However, if we are modeling more complex resources such as drivers, $R_t = (R_{ta})_{a\in\mathcal{A}}$, and we obtain a resource vector whose dimensionality is the size of the attribute space $\mathcal{A}$ (we might call this the "curse of curses").

The second is the expectation, which implies a sum (or integral) over all the dimensions of the information variable W_{t+1}. There are applications where W_{t+1} refers to the demand in period t or the price of an asset. However, there are many other applications where W_{t+1} is a vector. For example, in our trucking example, an element of W_{t+1} might be the demand to move freight from location i to location j of type k. If we have 100 locations and 10 product types, W_{t+1} would have 100,000 dimensions. If the

information includes the possibility of equipment failures, then W_{t+1} is the number of failures of equipment with attribute a (W_{t+1} might have $|\mathscr{A}|$ dimensions). The problem of computing the expectation is disguised by the standard form of Bellman's equation, which uses a one-step transition matrix (often viewed as input data). However, a one-step transition matrix is itself an expectation. If $1_{\{s'=S^M(s,x,W)\}}$ is the event that we transition from s to s' given x and W, then the one-step transition matrix is given by $p(s' \mid s, x) = \mathbb{E}\{1_{s'=S^M(s,x,W)}\}$. If W is a vector, computing the expectation can be computationally intractable.

The third curse of dimensionality is the decision vector x_t, which can have dimensionality up to $|\mathscr{A}| \times |\mathscr{D}|$. Searching over all possible values of $x_t \in \mathscr{X}_t$ is computationally intractable for problems with more than a few dimensions. The math programming community might express surprise at this perspective because problems where x_t has thousands of dimensions are considered routine, but if we use a discrete representation of the value function (a common assumption), then it is not possible to use major classes of algorithms such as the simplex method to solve the problem.

These "curses" all assume that states, information, and actions are discrete and that we have to enumerate all possible values. In many (most?) applications, we can exploit structure to avoid a brute-force enumeration. In the remainder of this paper, we build up the tools needed to solve high-dimensional resource allocation problems by combining the tools of artificial intelligence, approximate dynamic programming, and math programming.

It has been our finding that the most problematic of the "curses" is the expectation. In the next section, we lay the foundation for circumventing this problem.

3.3. The Post-Decision State Variable

The first step of our development requires formulating Bellman's equation around the post-decision state variable. Let S_t be our traditional state variable measured at time t (alternatively, at the end of time interval t) just before we make a decision. Let S_t^x be the state immediately after we make a decision. We can write the sequence of states, information, and actions using

$$(S_0, x_0, S_0^x, W_1, S_1, x_1, S_1^x, \ldots, W_t, S_t, x_t, S_t^x, \ldots).$$

The post-decision state variable is relatively unknown in textbooks on Markov decision processes, where Equation (1) and its variations are completely standard. However, it is an idea that seems to come up in specific settings. Sutton and Barto (1998) refer to it as the after-state variable, as in the position of a game board after one player has made his move but before his opponent has moved. Judd (1998) refers to it as the end-of-period state. The concept is standard in the modeling of decision trees, which uses decision nodes (predecision state variables) and outcome nodes (post-decision state variables) (see Skinner 1999). We prefer the term "post-decision state," which was first used by Van Roy et al. (1997), because this emphasizes the property that there is no new information in the post-decision state (hence it is indexed by time t). For our work, the concept of the post-decision state is particularly powerful when decisions are vectors and we need the tools of math programming to solve the one-period optimization problem.

If we model the evolution from S_t to S_t^x to S_{t+1}, we might introduce the transition functions $S^{M,W}(\cdot)$ and $S^{M,x}(\cdot)$ such that

$$S_t = S^{M,W}(S_{t-1}^x, W_t), \tag{12}$$

$$S_t^x = S^{M,x}(S_t, x_t). \tag{13}$$

We similarly use $R^{M,W}(\cdot)$ and $R^{M,x}(\cdot)$ to represent the transition functions for the resource state variable R_t and $a^{M,W}(\cdot)$ and $a^{M,x}(\cdot)$ for the attribute transition function (when talking about a single entity).

To illustrate using our fleet management application, assume that we wish to act on a truck in state i with decision d. The indicator function $\delta_j(i,d) = 1$ if the truck then ends up in state j as a result of the decision d. The post-decision resource vector would be given by

$$R_{tj}^x = \sum_{i \in \mathscr{I}} \sum_{d \in \mathscr{D}} \delta_j(i,d) x_{tid}. \tag{14}$$

After that, we have random changes in our fleet because of departures (drivers quitting) and additions (new hires), which we represent using $\hat{R}_{t+1,i}$. The next predecision resource state is then

$$R_{t+1,i} = R_{ti}^x + \hat{R}_{t+1,i}. \tag{15}$$

The predecision resource state is R_t along with the state of the environment ρ_t, which might include market prices, customer demands, changes in technology, and the weather. We might let $\hat{\rho}_{t+1}$ be exogenous changes to ρ_t, which evolves according to $\rho_{t+1} = \rho_t + \hat{\rho}_{t+1}$. We might assume that ρ_t evolves purely as a result of exogenous changes, which means the pre- and post-decision version of ρ_t are the same. As a result, we would write

$$S^t = (R_t, \rho_t),$$

$$S_t^x = (R_t^x, \rho_t).$$

Now let $V_t(S_t)$ be the value of being in state S_t just before we make a decision (as we have done), and let $V_t^x(S_t^x)$ be the value of being in state S_t^x just after we

have made a decision. $V_t(S_t)$ and $V_t^x(S_t^x)$ are related to each other via

$$V_t(S_t) = \max_{x_t \in \mathscr{X}_t} (C(S_t, x_t) + \gamma V_t^x(S^{M,x}(S_t, x_t))), \quad (16)$$

$$V_t^x(S_t^x) = \mathbb{E}\{V_{t+1}(S^{M,W}(S_t^x, W_{t+1})) \mid S_t^x\}. \quad (17)$$

We note that in writing (16), the feasible region $\mathscr{X}_t$ is a function of the state S_t. Rather than write $\mathscr{X}_t(S_t)$, we let the subscript t indicate that the feasible region depends on the information available at time t (and hence depends on S_t).

Substituting (17) into (16) gives us the expectation form of Bellman's equation (11). However, we are going to work with Equations (16) and (17) individually. We are going to use (17) to develop an approximation of $V_t^x(S_t^x)$ and then use the fact that (16) is a deterministic problem. If we design our approximation of $V_t^x(S_t^x)$ in a suitable way, we can solve (16) using a vast library of optimization algorithms. For example, if $C(S_t, x_t)$ is linear in x_t and if we replace $V_t^x(S_t^x)$ with a function that is linear in S_t^x (which is also linear in x_t), then we can solve the optimization problem using linear programming. If we require x_t to be integer, we can use integer programming or our favorite metaheuristic (see Blum and Roli 2003, Glover and Kochenberger 2003 for introductions).

4. A Single, Simple Entity

A nice illustration of our single, simple entity is a truck driver whose state is his current location $i \in \mathscr{I}$. Imagine that when the driver arrives at location i, he learns about a set of demands $\widehat{D}_t = (\widehat{D}_{tij})_{i,j \in \mathscr{I}}$, where $\widehat{D}_{tij}$ is the number of loads going from i to j. The state of our driver is his location i, but the state of the system is his location and the demands that are available to be moved, $\widehat{D}_t$. Let $R_{ti} = 1$ if the driver is in location i. Then R_t captures his physical state, but the complete state variable is given by $S_t = (R_t, \widehat{D}_t)$. R_t is effectively a scalar, but $\widehat{D}_t$ is a vector, which means that S_t is a vector. If we were to try to model this using Bellman's equation, we would encounter the first curse of dimensionality.

We can get around this problem by using the post-decision state. Assume that if a demand is not served at time t, it is lost. If our driver is at i and has accepted to move a load to j, then his post-decision state is $S_t^x = R_t^x$, where $R_{tj}^x = 1$ if he is headed to location j. The number of possible values of R_t^x is equal to the number of locations, which is quite small. Thus, although it may be computationally impossible to find the value function $V_t(S_t)$ around the predecision state, it is quite easy to find it around the post-decision state. Aside from using the concept of the post-decision state, we can solve the problem exactly using the classical techniques of Markov decision processes (Puterman 1994).

5. A Single, Complex Entity

There are many single-entity problems where we need to describe the entity using a vector of attributes a (normally, we would use S to denote the state of our system). Assuming the attributes are discrete, we could associate the index i with a specific attribute vector a and form a state space $\mathscr{I}$ (a classical flat representation). However, when the attribute vector has as few as five or six elements (depending on the number of outcomes each dimension can take), the attribute space may be too large to enumerate. In this case, we use a factored representation by retaining the explicit structure of the attribute vector a.

Given an entity in state a, we can act on the entity using one of a set of decisions $d \in \mathscr{D}$ (which generally depends on the state we are in). We let $C(a, d)$ be the contribution earned from this decision (this may be stochastic, but for our presentation, we assume it is deterministic). For a finite horizon problem defined over $t = 0, 1, \ldots, T$, we can find the optimal policy by solving Bellman's equation (1), where the state $S_t = a_t$ is the attribute of the entity.

The problem is that we are no longer able to solve this problem for each attribute a_t. In the remainder of this section, we describe how approximate dynamic programming can be used to produce practical solutions to this problem. We note that as a by-product, we provide an algorithm that easily handles vectors of information (the second curse of dimensionality), but for the moment, we are going to assume that we have a relatively small set of decisions $\mathscr{D}$ that can be enumerated.

5.1. Approximate Dynamic Programming

Classical dynamic programming works backward in time, where we assume that we have a value function $V_{t+1}(a')$, which allows us to find $V_t(a)$ for each attribute (state) a. When we use approximate dynamic programming, we can step forward in time, using an approximate value function to make decisions. Classical treatments of approximate dynamic programming (see, for example, Bertsekas and Tsitsiklis 1996, Sutton and Barto 1998) replace the value function in Equation (11) with an approximation, giving us an optimization problem that looks like

$$\hat{v}_t^n = \max_{d_t \in \mathscr{D}} \big(C(a_t^n, d_t) + \gamma \mathbb{E}\{\overline{V}_{t+1}^{n-1}(a^M(a_t^n, d_t, W_{t+1})) \mid a_t^n\}\big). \quad (18)$$

Here, a_t^n is the current state of our entity at time t in iteration n, and $\hat{v}_t^n$ is a sample estimate of the value of being in state a_t^n. $\overline{V}_{t+1}^{n-1}(a')$ is the value function approximation obtained from iteration $n-1$. We update our estimate of the value of being in state a_t^n using the simple updating formula

$$\overline{V}_t^n(a_t^n) = (1 - \alpha_{n-1})\overline{V}_t^{n-1}(a_t^n) + \alpha_{n-1}\hat{v}_t^n, \quad (19)$$

where α_{n-1} is a stepsize between 0 and 1. This type of updating assumes that we are using a lookup table representation, where we have a value of being in state a_t^n.

There are a number of variations in how states are traversed and updated. After updating a state a_t^n, we may simply choose another state at random, or we may take the decision d_t^n that solves (18) and then simulate new information W_{t+1} to obtain the next state a_{t+1}^n, a strategy that is sometimes referred to as *real-time dynamic programming* (RTDP) (Barto et al. 1995). There are also different ways of updating the value function. Equations (18) and (19) are purely illustrative.

Let d_t^n be the decision produced by solving (18). We then choose a random observation of W_{t+1}. We represent this mathematically by letting ω^n denote the sample path, where $W_{t+1}^n = W_{t+1}(\omega^n)$ is a sample realization of the random variable W_{t+1}. The next predecision state is given by $a_{t+1}^n = a^M(a_t^n, d_t^n, W_{t+1}(\omega^n))$, which is pure simulation, allowing us to use arbitrarily complex information processes. However, if W_t is vector valued, we are probably going to have difficulty computing the expectation in (18) (the second curse of dimensionality).

We overcome the second curse of dimensionality by using Equations (16) and (17). Instead of producing a value function approximation around the predecision state a_t^n, it is easier to update the value function around the post-decision state, $a_{t-1}^{x,n}$. There are several ways to represent a post-decision state, but it always captures what we know about the state of the system after we make a decision, before any new information has arrived (see Powell 2007, §4.4 for a more complete discussion of post-decision state variables). If we are playing backgammon, it would be the state of the board after we make our move but before the next roll of the dice. If we are moving a truck from Dallas to Chicago, it would be the attributes that we forecast the truck will have when it arrives in Chicago but before it has left Dallas. The post-decision state can also use an estimate of what might happen (a forecast) before knowing what actually happens (for example, the expected travel time from Dallas to Chicago). For our single, complex entity, this means that we replace the updating Equation (19) with

$$\overline{V}_{t-1}^n(a_{t-1}^{x,n}) = (1-\alpha_{n-1})\overline{V}_{t-1}^{n-1}(a_{t-1}^{x,n}) + \alpha_{n-1}\hat{v}_t^n.$$

Note that we are using $\hat{v}_t^n$, the value observed when our entity had attribute a_t^n, to update $\overline{V}_{t-1}^{n-1}(a_{t-1}^{x,n})$, the value around the previous post-decision attribute (or state) a_{t-1}^n. Equation (20) is a mechanism for approximating the expectation of the value of being in post-decision state $a_{t-1}^{x,n}$.

Step 0. Initialization:

Step 0(a). Initialize $\overline{V}_t^0$, $t \in \mathcal{T}$.
Step 0(b). Set $n = 1$.
Step 0(c). Initialize a_0^1.

Step 1. Choose a sample path ω^n.

Step 2. Do for $t = 0, 1, \ldots, T$:

Step 2(b). Solve:

$$\hat{v}_t^n = \max_{d_t}(C(a_t^n, d_t) + \gamma \overline{V}_t^{n-1}(a^{M,x}(a_t^n, d_t)))$$

Let d_t^n be the best decision, and let $a_t^{x,n} = a^{M,x}(a_t^n, d_t^n)$.

Step 2(c). Update the value function:

$$\overline{V}_{t-1}^n(a_{t-1}^{x,n}) = (1-\alpha_{n-1})\overline{V}_{t-1}^{n-1}(a_{t-1}^{x,n}) + \alpha_{n-1}\hat{v}_t^n$$

Step 2(d). Compute the new predecision state

$$a_{t+1}^n = a^{M,W}(a_t^{x,n}, W_{t+1}(\omega^n)).$$

Step 3. Increment n. If $n \le N$ go to Step 1.
Step 4. Return the value functions $(\overline{V}_t^n)_{t=1}^T$.

Figure 1 An Approximate Dynamic Programming Algorithm for a Single, Complex Entity

This step means that the optimization problem in (18) is replaced with the deterministic problem

$$\hat{v}_t^n = \max_{d_t \in \mathcal{D}}(C(a_t^n, d_t) + \gamma \overline{V}_t^{n-1}(a^{M,x}(a_t^n, d_t))). \tag{20}$$

Thus we no longer have an expectation, and we still simulate our way from a_t^n to $a_{t+1}^n = a^{M,W}(a_t^{x,n}, W_{t+1}(\omega^n))$, as we did before. This means that the methodology easily handles complex stochastic processes, but we have not yet solved the problem of large action spaces (we still assume that the set $\mathcal{D}$ is small enough to enumerate). Figure 1 describes the complete algorithm.

Specialists in reinforcement learning will recognize the basic updating step in Equation (18) as a form of temporal-difference learning (see Sutton and Barto 1998, Bertsekas and Tsitsiklis 1996) known as "TD(0)." Another popular technique is known as Q-learning (Watkins and Dayan 1992, Sutton and Barto 1998), which involves starting with a state-action pair ((a, d) in the notation of this section), then choosing a random outcome $W_{t+1}^n = W_{t+1}(\omega^n)$ of what might happen in the future, and estimating a q-factor using

$$\hat{q}_t^n(a_t^n, d_t^n) = C(a_t^n, d_t) + \gamma \max_{d \in \mathcal{D}} \overline{Q}_t^{n-1}(a_t^n, d_t),$$

which is then smoothed to estimate

$$\overline{Q}_t^n(a_t^n, d_t^n) = (1-\alpha_{n-1})\overline{Q}_t^{n-1}(a_t^n, d_t^n) + \alpha_{n-1}\hat{q}_t^n(a_t^n, d_t^n). \tag{21}$$

There are similarities between Q-learning and approximate dynamic programming (ADP) with a post-decision state variable. Q-Learning requires that you learn a value $\bar{Q}_t(a_t, d_t)$ for each state a_t and decision d_t, whereas the method we have described requires only that we learn the value of being in a specific post-decision state. Q-Learning is often described as a technique for "model-free" dynamic programs, a concept that is relatively unfamiliar to the OR community. These applications arise when we can observe a state, but we do not have access to a transition function (the "model") to predict the state resulting from an action (this might also happen when we do not have a mathematical model for the information process). Model-free dynamic programming arises when there is a physical process that is used to observe the results of decisions.

5.2. Approximation Techniques

Approximate dynamic programming gives the appearance of overcoming the challenge of large state spaces because we do not have to loop over all states. This is somewhat of a myth. In practice, the computational burden of looping over all states in backward dynamic programming has been replaced with the statistical problem of estimating the value of many states. It is not enough to know the value of being in states that we actually visit. If we are making good decisions, we have to have good estimates of the value of states that we *might* visit. This may be far smaller than the entire set of states, but it can still be an extremely large number.

There is a vast array of statistical techniques that can be used to approximate the value function for a single, complex entity, especially if we are willing to take advantage of the structure of the attribute vector. In this section, we review two of the most popular strategies. The first uses simple or mixed aggregation, which makes almost no assumptions about the structure of the attribute vector. The second uses the concept of "basis functions," which is simply a way of describing statistical regression models. This strategy requires that we make explicit use of the properties of specific attributes.

5.2.1. Aggregation. Perhaps the most widely used method for approximating value functions is aggregation. Assume that we have a family of aggregation functions where

$\mathcal{G}=$ the set of indices corresponding to the levels of aggregation;

$G^g: \mathcal{A} \rightarrow \mathcal{A}^{(g)}$, $g \in \mathcal{G}$;

$a^{(g)} = G^g(a)$, the gth-level aggregation of the attribute vector a.

Thus, G^g is an aggregation function that acts on the attribute space $\mathcal{A}$ producing a smaller attribute space $\mathcal{A}^{(g)}$. In the AI community, aggregation is a form of state abstraction. The function G could discretize a numerical attribute, or it might simply ignore an attribute. For example, we could list attributes $a = (a_1, a_2, \ldots, a_N)$ in decreasing order of importance and create a series of aggregations using the attribute vectors

$$a^{(0)} = (a_1, a_2, \ldots, a_{N-1}, a_N),$$
$$a^{(1)} = (a_1, a_2, \ldots, a_{N-1}, \text{-}),$$
$$a^{(2)} = (a_1, a_2, \ldots, \text{-}, \text{-}),$$

and so on. For our trucking application, the attribute vector might be (location, equipment type, driver domicile, days from home). We could then create aggregations by ignoring days from home, then ignoring driver domicile, and so on. Also, we could create aggregations on a single attribute such as location or domicile; instead of using a five-digit zip code, we could use a three-digit zip code.

If we visit state a_t^n at time t during iteration n, the updating equation for an aggregated estimate would be given by

$$\bar{V}_t^{(g,n)}(a) = \begin{cases} (1-\alpha_{n-1})\bar{V}_t^{(g,n-1)}(a) + \alpha_{n-1}\hat{v}_t^n \\ \quad \text{if } G(a_t^n) = a, \\ \bar{V}_t^{(g,n)}(a^{(g)}) \quad \text{otherwise,} \end{cases} \tag{22}$$

where $\bar{V}_t^{(g,n)}(a)$ is an estimate of the value of being in aggregated state $a^{(g)} = G^g(a)$. This uses a traditional form of aggregation where each state is aggregated up to a unique aggregate state. Another strategy is soft state aggregation (Singh et al. 1995), where aggregations can be overlapping. Soft state aggregation can be viewed as a form of nonparametric regression for state spaces that lack a distance metric (for example, one or more attributes may be categorical).

Aggregation is a widely used method for solving the problem of large state spaces (for a sample of relevant research, see Rogers et al. 1991). The limitation is that there is no good answer to determining the right level of aggregation. The answer depends on how many iterations you can run your algorithm. In fact, it would be best to change the level of aggregation as the algorithm progresses (Bertsekas and Castanon 1989, Luus 2000). A strategy that is more flexible than choosing a single level of aggregation is to define a family of aggregation functions G^g, $g \in \mathcal{G}$. We can then use a weighted sum of estimates at different levels of aggregation such as

$$\bar{v}_a^n = \sum_{g \in \mathcal{G}} w_a^{(g)} \bar{v}_a^{(g)}, \tag{23}$$

where $w_a^{(g)}$ is the weight applied to the estimate of the value of being in state a at the gth level of aggregation

if we are in state a. This model allows us to adjust weights based on how often we visit a state. Using our trucking application, we will have few observations of drivers in less-populated states such as Montana as opposed to denser states such as Illinois. However, it does mean that we have a lot of weights to compute. In fact, when the attribute space is large, we will often find that we have no observations for many attributes at the most disaggregate level, but we always have something at the more aggregate levels.

There are different strategies for estimating the weights $w_a^{(g)}$. A simple method that has proven very effective in this context is to use weights that are proportional to the variance of $\bar{v}_a^{(g)}$ (see George et al. 2008 or Powell 2007, Chapter 6). The field of machine learning offers a range of techniques for choosing the relevant attributes and levels of aggregation (Hastie et al. 2001 contains a nice overview of these methods).

5.2.2. Basis Functions. A more general approximation strategy has evolved in the ADP literature under the name of basis functions. We create these by assuming that we can identify *features* that are functions that act on the attribute vector, which we feel have a measurable impact on the value function. People in OR will recognize this strategy as statistical regression, where basis functions are simply explanatory variables in a statistical model.

Unlike aggregation, which assumes very little about the attribute vector, basis functions require that we take advantage of the specific properties of individual attributes. For example, we may feel that the value of a truck driver is linearly related to the number of days he has been away from home. Alternatively, we may use an indicator variable to capture the value of a driver whose home is in Texas. We begin by defining

$\mathcal{F}$ = a set of features,
$\phi_f(a)$ = a function of the state vector a, which is felt to capture important relationships between the state variable and the value function.

The value function approximation might then be written as

$$\bar{V}_t(a_t \mid \theta) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(a_t).$$

This is typically referred to as a linear model because it is linear in the parameters (the basis functions themselves may, of course, be highly nonlinear functions of the attribute vector). In a traditional regression model, we might take a series of observations of features, call them $(a^m)_{m=1}^n$, and relate them to observations of the value of the entity corresponding to these attributes, which we denote by $(\hat{v}^m)_{m=1}^n$. Techniques from linear regression would then give us a set of parameters θ^n, which reflect the n observations.

In dynamic programming, we may have a parameter vector θ^{n-1}, after which we observe an attribute vector $\hat{a}^n$ and the value $\hat{v}^n$ of the entity with attribute $\hat{a}^n$. We could reestimate θ (giving us θ^n), but it is much better to do this recursively. There are several techniques for doing this, but one of the simplest is a stochastic gradient algorithm, given by

$$\begin{aligned}\theta^n &= \theta^{n-1} - \alpha_{n-1}(\bar{V}(\hat{a}^n \mid \theta^{n-1}) - \hat{v}^n(\hat{a}^n))\nabla_\theta \bar{V}_t(\hat{a}^n \mid \theta^{n-1}) \\ &= \theta^{n-1} - \alpha_{n-1}(\bar{V}(\hat{a}^n \mid \theta^{n-1}) - \hat{v}^n(\hat{a}^n)) \begin{pmatrix} \phi_1(\hat{a}^n) \\ \phi_2(\hat{a}^n) \\ \vdots \\ \phi_F(\hat{a}^n) \end{pmatrix}. \quad (24)\end{aligned}$$

This method requires that we start with an initial estimate θ^0 before we have any observations. We then get an updated estimate after each observation. The stepsize α_{n-1} determines how much weight we put on the most recent observation.

Basis functions, and stochastic gradient algorithms for fitting the parameters, are widely used in the approximate dynamic programming literature (Tsitsiklis and Van Roy 1996, 1997; Van Roy et al. 1997; Van Roy 2001). A major strength of this strategy is that it allows us to exploit structure in the attribute vector. However, this also means that we have to identify functions that recognize this structure, a potential problem if the attributes are not well defined (or if we do not fully understand how they interact with each other).

6. Multiple Entities, Simple Attributes

We now make the transition to handling multiple entities, where there are different types of entities. We assume that the number of different types of entities is not too large. A nice illustration is managing blood inventories where blood is characterized by one of eight blood types and age (zero to five weeks), giving us a total of 48 usable "blood attributes" (blood type plus age). For our trucking example, we would be modeling the inventories of trucks around the country (rather than the availability of drivers). For this class of applications, we have all three curses of dimensionality: (1) the state variable (e.g., the 48 blood attributes), (2) the information process (16 dimensions, covering 8 dimensions of blood donations and 8 dimensions for blood demands), (3) and the decision variables (more than 200 dimensions to describe all the different types of allowable blood uses).

We can model a problem with multiple, simple entities using the same notation we introduced in §2.2 but where we would use the index i instead of the attribute vector a. Thus the resource state vector would be $R_t = (R_{ti})_{i \in \mathcal{I}}$, where R_{ti} is the number of resources in state i.

We can use the same basic algorithm that we described in Figure 1, with a few changes. Instead of state a_t, we use R_t. Instead of choosing a decision $d_t \in \mathcal{D}$, we are going to choose a vector $x_t \in \mathcal{X}_t$ (the third curse of dimensionality). We solve this problem by using the tools of math programming. For our illustration, we will need to solve the decision problem using a linear programming package. This, in turn, means that we cannot use a lookup table representation for our value function. Instead, we need to use a functional approximation that retains the structure of our decision problem as a linear program.

Following the approximation strategy we introduced in §5.1, we start in state R_t^n (at time t in iteration n) and then solve the optimization problem using

$$X^\pi(R_t^n) = \arg\max_{x_t \in \mathcal{X}_t^n} \left(C(R_t^n, x_t) + \gamma \bar{V}_t^{n-1}(R_t^x) \right), \quad (25)$$

where $R_t^x = R^{M,x}(R_t^n, x_t)$ is a deterministic function that depends on x_t. Note that we avoid the need to deal with an expectation imbedded within the optimization problem. We face our first application where x_t is a vector, with possibly thousands of dimensions. If we were just using a myopic policy ($\bar{V}_t(R_t^x) = 0$), the problem would be a straightforward mathematical program, where handling decision functions with thousands of dimensions is routine. Not surprisingly, we want to design a value function approximation that allows us to take advantage of this structure.

The two simplest strategies that can be used for approximating the value function are linear (in the resource state), which is written as

$$\bar{V}_t(R_t^x) = \sum_{j \in \mathcal{J}} \bar{v}_{tj} R_{tj}^x, \quad (26)$$

and nonlinear but separable, given by

$$\bar{V}_t(R_t^x) = \sum_{j \in \mathcal{J}} \bar{V}_{tj}(R_{tj}^x). \quad (27)$$

The approximation $\bar{V}_{tj}(R_{tj}^x)$ can be piecewise linear or a continuously differentiable nonlinear function. If it is linear or piecewise linear, then (25) can be solved as a linear program. Figure 2 illustrates the linear program when we use a separable, piecewise linear approximation. This linear program scales to large industrial applications, where $\mathcal{J}$ might have thousands of elements (such as locations).

We illustrate the process of updating the value function using a linear value function approximation. In this case, Equation (25) becomes

$$x_t^n = \arg\max_{x_t \in \mathcal{X}_t^n} \left\{ \sum_{i \in \mathcal{I}} \sum_{d \in \mathcal{D}} c_{tid} x_{tid} + \gamma \sum_{j \in \mathcal{J}} \bar{v}_{tj}^{n-1} R_{tj}^x \right\}. \quad (28)$$

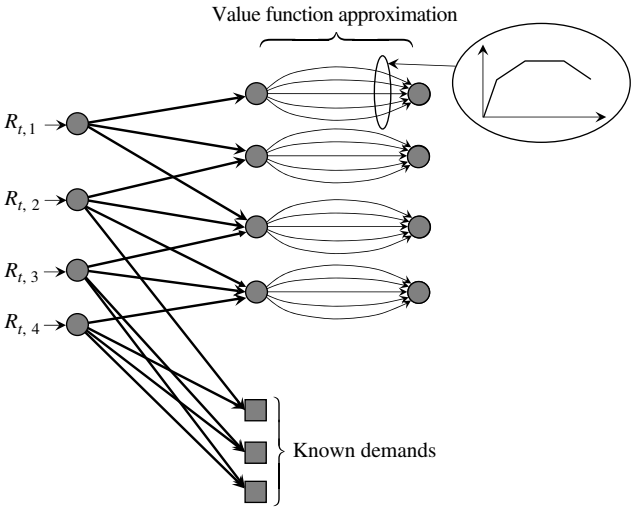

Figure 2 Illustration of Decision Problem Using Separable, Piecewise Linear Value Function Approximation

Using Equation (14) and a little algebra takes us to

$$x_t^n = \arg\max_{x_t \in \mathcal{X}_t^n} \left\{ \sum_{i \in \mathcal{I}} \sum_{d \in \mathcal{D}} (c_{tid} + \gamma \bar{v}_{t,i_j^M(i,d)}^{n-1}) x_{tid} \right\}. \quad (29)$$

This is a linear program that can be solved for very large-scale problems.

Once we know that we can solve the decision problem, we next face the challenge of updating the value function approximation. In §5, we updated the value function approximation using an estimate $\hat{v}_t^n$ of the value of being in a state a_t^n. Now, we are going to use derivatives to update slopes of the value function approximation rather than using the value of being in a state. This is a fairly common strategy in the control theory community, where approximate dynamic programming is often called "heuristic dynamic programming," whereas the use of derivatives is known as "dual heuristic dynamic programming" (see Werbos 1990, Ferrari and Stengel 2004).

Obtaining slopes is relatively easy when we note that solving (29) as a linear program subject to the constraints in (5) and (6) produces dual variables for the flow conservation constraints (5). Let $\hat{v}_{ti}^n$ represent the dual variable *for each of these constraints* $i \in \mathcal{I}$ obtained in the nth iteration of the algorithm. We emphasize that it is important that we are able to obtain an estimate of the slope for each element of R_t, because this will no longer be the case when we make the final step to handling multiple, complex entities. If we were managing a single entity, obtaining $\hat{v}_{ti}^n$ for all $i \in \mathcal{I}$ is the same as getting an updated estimate of the value of being in each state at each iteration (also known as synchronous dynamic programming).

The ability to use slopes is particularly effective in the context of solving optimization problems involving multiple entities. First, it is generally the case that

we need estimates of the slopes of the value function instead of the value function itself (adding a constant does not change the solution). Second, instead of obtaining a single observation $\hat{v}_t^n$ of the value of being in state S_t^n, we obtain a vector of slopes, one for each $i \in \mathcal{I}$. If the set $\mathcal{I}$ has hundreds or thousands of elements, then at each iteration, instead of getting a single update, we get hundreds or thousands of pieces of new information, which dramatically improves the overall rate of convergence. The idea of using derivatives was first introduced by Werbos (1990) in the control theory literature under the name dual heuristic dynamic programming, but it has been used throughout the stochastic programming literature (see Higle and Sen 1996, Birge and Louveaux 1997 for reviews).

Once we obtain the gradients $\hat{v}_{ti}^n$, our value function approximation is updated using

$$\bar{v}_{t-1,\,i}^n = (1-\alpha_n)\bar{v}_{t-1,\,i}^{n-1} + \alpha_n \hat{v}_{ti}^n.$$

Note that $\hat{v}_{ti}^n$, which contains information that becomes known in time period t, is used to update $\bar{v}_{t-1,\,i}^{n-1}$, which is an approximation of an expectation that only contains information up through time $t-1$.

Linear value function approximations work well when we are primarily interested in making sure we use the right type of resource. For example, Simão et al. (2009) describes a problem where we have to decide which driver should move a load to New York. The important issue is not how many drivers are in New York, but whether we should have a driver in New York whose home is Texas or California. In other words, we want the right *type* of driver. Piecewise linear value function approximations allow us to capture the declining marginal value of additional resources and are therefore more useful when we are trying to decide how many resources.

If separability is too strong of an approximation, a powerful technique is to use Benders' decomposition (see Higle and Sen 1991; Birge and Louveaux 1997; or Powell 2007, Chapter 11). At time t, iteration n we would solve a problem with the form

$$\max_{x_t,\,z} C(S_t, x_t) + z \tag{30}$$

subject to $x_t \in \mathcal{X}_t$, and

$$z \le (\alpha_t^n(v))^T x_t + \beta_t^n(v) \quad \forall\, v \in \mathcal{V}_t^n. \tag{31}$$

Equation (31) represents a series of cuts that approximates the value function. The set $\mathcal{V}_t^n$ refers to a set of vertices (or cuts) that are generated as the algorithm progresses. The coefficients $\alpha_t^n(v)$ and $\beta_t^n(v)$ are computed by solving the linear program at time $t+1$ and using the dual to infer the coefficients. This method will produce optimal solutions (in the limit), although convergence can become slow as the dimensionality of the resource state vector increases (see Powell et al. 2004). Another strategy is to use a regression model (basis functions), where we might specify a value function approximation of the form

$$\bar{V}_t(R_t) = \theta_{t0} + \sum_{i\in\mathcal{I}} \theta_{1i} R_{ti} + \sum_{i\in\mathcal{I}} \theta_{2i} R_{ti}^2.$$

For the determined reader, Klabjan and Adelman (2007) describe how to use ridge regression to produce a novel class of approximation strategies.

6.1. Discussion

This section has demonstrated that approximate dynamic programming can be used to solve (approximately) certain classes of resource allocation problems that are much larger than would have been achievable using classical approximate dynamic programming techniques. These methods have been demonstrated on problems where the number of dimensions of the state vector R_t is in the thousands. However, this work assumes that there is a dual variable $\hat{v}_{ti}^n$ for each dimension $i \in \mathcal{I}$ of R_t. Although this is certainly a breakthrough, we are interested in even larger problems. In the next section, we show how we return to the fundamentals of approximate dynamic programming to solve these very large-scale problems.

7. Multiple, Complex Assets

There are many applications where we have to manage multiple, complex entities. These may be people, aircraft, or complex equipment (e.g., power generators). Return to our notation where a_t is a vector of attributes describing an entity. Let R_{ta} be the number of entities with this attribute. If we manage a single, complex entity with attribute a_t, where a_t might have several dozen dimensions, we have seen that we are in the domain of AI problems where the state space $\mathcal{A}$ is too large to enumerate, requiring us to use approximation methods. When we have multiple entities, we have a state variable where the number of *dimensions* is comparable to the size of the state space of a large dynamic program.

Although this problem may seem intractably complex, we already have the tools we need to solve it (approximately). We have already seen in §6 that we can obtain high-quality solutions if we use value function approximations that are linear in R_{ta} or nonlinear but separable in R_{ta}. The major limitation of this approach is that we assumed that we could measure the dual variable $\hat{v}_{ta}$ (or obtain a gradient) for each type of entity of type a (we used type i in §6). When our space of entities is not too large (10,000 is a reasonable upper limit), this is a good assumption. However, with as few as three or four attributes, the

attribute space $\mathcal{A}$ can grows exponentially. As with the types of dynamic programs considered in the AI community, the state space of a single entity can easily reach numbers such as 10^{10} or more. For all practical purposes, the vector R_t is infinite dimensional.

If our value function approximation is linear in the resource state (Equation (26)), we can approximate the marginal value of a unit of resource with attribute a using the approximation methods of §5.2.1, where

$$\bar{v}_{ta}^{n} = \sum_{g \in \mathcal{G}} w_a^{(g,n)} \bar{v}_{ta}^{(g,n)}. \tag{32}$$

This approximation can then be used just as we did in the simple entity case. If we repeat the derivation of Equation (29), we would obtain the following linear program:

$$x_t^n = \arg\max_{x_t \in \mathcal{X}_t^n} \left\{ \sum_{a \in \mathcal{A}} \sum_{d \in \mathcal{D}} (c_{tad} + \gamma \bar{v}_{t,a^M(a,d)}^{n-1}) x_{tad} \right\}. \tag{33}$$

Equation (33) requires that when we compute the cost c_{tad} for a given (a, d) pair, we also have to compute $\bar{v}_{t,a^M(a,d)}^{n-1}$ using Equation (32). This logic works for attribute spaces that are arbitrarily large, because we always have an estimate of the value of an attribute (at least at some level of aggregation). In effect, we are using statistical methods to overcome the challenge of high-dimensional state vectors (or linear programs with a virtually infinite number of constraints).

Because the attribute space is too large to enumerate, we need to select a strategy to determine which elements of the vector R_t are generated. We only obtain a dual $\hat{v}_{ta}$ if we generate the resource constraint for attribute a. Let $\mathcal{A}_t^n$ be the attributes that we have explicitly enumerated, which means our resource state vector at iteration n is given by $R_t^n = (R_{ta}^n)_{a \in \mathcal{A}^n}$. A natural strategy is to enumerate the attributes that receive flow. For example, it might be the case that we have never visited the attribute $a' = a^{M,x}(a, d)$, even though we can compute (using hierarchical aggregation) an estimate of the value of being in this state. Now, assume that after solving the linear program in (33), we obtain $x_{tad} > 0$. We could then add a' to the set $\mathcal{A}^{n-1}$. We note that we might add an attribute at an earlier iteration, which then does not receive flow at a later iteration; thus, it is certainly possible that we can have $R_{ta}^n = 0$ for some $a \in \mathcal{A}^n$. We have found that it is important to retain this attribute in the set $\mathcal{A}^n$. These are known as "zero flow buckets" and are important in the stability of the algorithm.

Viewed in the language of approximate dynamic programming, this is a form of exploration that is critical to the performance of the algorithm. It is quite natural to resample the value of all states that we may have already visited (we may also use rules to include a state in $\mathcal{A}^n$ even if we do not create entities with these attributes). Interestingly, we have never seen this idea in the approximate dynamic programming literature. The ADP community distinguishes between *synchronous* algorithms (which sample all states at every iteration) and *asynchronous* algorithms (which generally sample only one state at every iteration). The idea of resampling a subset of visited states is a potential opportunity for the ADP literature, especially when resampling a visited state offers computational advantages over states that have never been visited (for example, costs may be calculated and stored).

8. More Complex Problems

This paper has demonstrated how approximate dynamic programming, combined with math programming (linear and integer), can be used to solve extremely large-scale problems (approximately), producing solutions that are much more than mere simulations of complex, dynamic problems. The techniques of approximate dynamic programming allow us to approximate linear programs effectively with an infinite number of rows (in one time period), with only linear growth in run times for multiple time periods. The resulting formulations are much smaller and easier to solve than deterministic models that encompass the entire planning horizon.

There is a vast range of more complex problems that fall under the umbrella of "dynamic resource management," spanning such difficult problems as research and development (R&D) portfolio optimization (Beaujon et al. 2001), stochastic multicommodity flow problems (Topaloglu and Powell 2006), stochastic machine scheduling (Pinedo 1995), and dynamic vehicle routing (Ichoua et al. 2005, Adelman 2004). However, as of this writing, the use of ADP for these more difficult problems has to be viewed in its infancy.

Approximate dynamic programming offers the potential of breaking down very hard combinatorial problems into sequences of much easier ones. The challenge is obtaining solutions of high quality (and proving this). This strategy can be followed on a broad range of problems by following four simple steps.

First, imagine that you are solving the problem myopically (for example, assigning machines to serve the tasks you know about, or routing a vehicle to pick up customers that have already called in). Now, design an algorithm for this problem. You may feel that you can solve the problem optimally, especially because these problems tend to be much smaller than problems where everything is known. However, you may need to use your favorite heuristic (local search, tabu search, genetic algorithms), as illustrated

in Gendreau et al. (1999). Your choice of algorithm will determine how you have to approximate your value function.

Second, you have to identify a value function approximation that you feel captures the important behaviors of your problem but that does not complicate unnecessarily your algorithmic strategy for the myopic problem. If you are using a heuristic search algorithm, you can live with almost any functional approximation. However, if you are solving it optimally, your approximation may have to have special structure. If your decision problem requires the use of a linear, nonlinear, or integer programming package, you have to choose your value function approximation accordingly.

Third, you need to design an updating strategy to improve your approximation. In our motivating application, we could use dual variables or finite differences to capture the value of an entity with a particular type of attribute. For more complex combinatoric problems, even approximating the incremental value of an entity may be quite hard (and unreliable). For example, our heuristic may produce routes for a fleet of vehicles. From this solution, we can estimate the cost (or contribution) of each vehicle by following its route. However, this is not the same as estimating the change that would happen if we did not have that particular vehicle (in other words, a derivative). An open research question would be to estimate the error from approximating the value of a vehicle by using the cost of the path that it follows instead of using the marginal value (which involves dropping the vehicle from the solution and reoptimizing).

Finally, you need to develop a method for testing your approximation. For some problems, it is possible to obtain optimal or near-optimal solutions for deterministic versions of the problem that serve as useful metrics. In most cases, you are just hoping to obtain solutions that are better than your myopic approximation (which ignores the impact of decisions now on the future).

We believe that this simple strategy can be applied to a vast range of dynamic resource management problems. Developing an effective approximate dynamic programming algorithm, which includes finding a value function approximation that is computationally tractable, proposing an updating strategy, and showing that the value function produces higher-quality solutions than a simple myopic model, represents a significant (and potentially patentable) contribution.

Acknowledgments

This research was supported, in part, by Grant AFOSR-F49620-93-1-0098 from the Air Force Office of Scientific Research.

References

Adelman, D. 2004. A price-directed approach to stochastic inventory/routing. *Oper. Res.* **52**(4) 499–514.

Barto, A. G., S. J. Bradtke, S. P. Singh. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* (*Special Volume on Computational Res. Interaction Agency*) **72**(1-2) 81–138.

Beaujon, G. J., S. P. Marin, G. C. McDonald. 2001. Balancing and optimizing a portfolio of R&D projects. *Naval Res. Logist.* **48**(1) 18–40.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Bertsekas, D., D. Castanon. 1989. Adaptive aggregation methods for infinite horizon dynamic programming. *IEEE Trans. Automatic Control* **34**(6) 589–598.

Bertsekas, D. P., J. N. Tsitsiklis. 1996. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA.

Birge, J. R., F. Louveaux. 1997. *Introduction to Stochastic Programming*. Springer-Verlag, New York.

Blum, C., A. Roli. 2003. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surveys* **35**(3) 268–308.

Boutilier, C., T. Dean, S. Hanks. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *J. Artificial Intelligence Res.* **11** 1–94.

Chang, H. S., M. C. Fu, J. Hu, S. I. Marcus. 2007. *Simulation-Based Algorithms for Markov Decision Processes*. Springer-Verlag, Berlin.

Ferrari, S., R. F. Stengel. 2004. Model-based adaptive critic designs. J. Si, A. G. Barto, W. B. Powell, D. Wunsch, eds. *Handbook of Learning and Approximate Dynamic Programming*. IEEE Press, New York, 64–94.

Gendreau, M., F. Guertin, J.-Y. Potvin, E. Taillard. 1999. Parallel tabu search for real-time vehicle routing and dispatching. *Transportation Sci.* **33**(4) 381–390.

George, A., W. B. Powell, S. Kulkarni. 2008. Value function approximation using multiple aggregation for multiattribute resource management. *J. Machine Learning Res.* **9** 2079–2111.

Glover, F., G. A. Kochenberger, eds. 2003. *Handbook of Metaheuristics*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Guestrin, C., D. Koller, R. Parr. 2003. Efficient solution algorithms for factored MDPs. *J. Artificial Intelligence Res.* **19** 399–468.

Hastie, T., R. Tibshirani, J. Friedman. 2001. *The Elements of Statistical Learning*. Springer Series in Statistics, Springer, New York.

Higle, J. L, S. Sen. 1991. Stochastic decomposition: An algorithm for two-stage linear programs with recourse. *Math. Oper. Res.* **16**(3) 650–669.

Higle, J. L., S. Sen. 1996. *Stochastic Decomposition: A Statistical Method for Large Scale Stochastic Linear Programming*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Howard, R. 1971. *Dynamic Probabilistic Systems, Volume II: Semimarkov and Decision Processes*. John Wiley & Sons, New York.

Ichoua, S., M. Gendreau, J.-Y. Potvin. 2005. Exploiting knowledge about future demands for real-time vehicle dispatching. *Transportation Sci.* **40**(2) 211–225.

Judd, K. L. 1998. *Numerical Methods in Economics*. MIT Press, Cambridge, MA.

Klabjan, D., D. Adelman. 2007. An infinite-dimensional linear programming algorithm for deterministic semi-Markov decision processes on Borel spaces. *Math. Oper. Res.* **32**(3) 528–550.

Luus, R. 2000. *Iterative Dynamic Programming*. Chapman & Hall/CRC, New York.

Pinedo, M. 1995. *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, Englewood Cliffs, NJ.

Powell, W. 2007. *Approximate Dynamic Programming: Solving the curses of Dimensionality*. John Wiley & Sons, New York.

Powell, W., A. Ruszczyński, H. Topaloglu. 2004. Learning algorithms for separable approximations of discrete stochastic optimization problems. *Math. Oper. Res.* **29**(4) 814–836.

Puterman, M. L. 1994. *Markov Decision Processes*. John Wiley & Sons, New York.

Rogers, D. F., R. D. Plante, R. T. Wong, J. R. Evans. 1991. Aggregation and disaggregation techniques and methodology in optimization. *Oper. Res.* **39**(4) 553–582.

Si, J., A. G. Barto, W. B. Powell, D. Wunsch, eds. 2004. *Handbook of Learning and Approximate Dynamic Programming*. IEEE Press, New York.

Simão, H. P., J. Day, A. P. George, T. Gifford, J. Nienow, W. B. Powell. 2009. An approximate dynamic programming algorithm for large-scale fleet management: A case application. *Transportation Sci.* **43**(2) 178–197.

Singh, S. P., T. Jaakkola, M. I. Jordan. 1995. Reinforcement learning with soft state aggregation. G. Tesauro, D. Touretzky, T. K. Leen, eds. *Advances in Neural Information Processing Systems*, Vol. 7. MIT Press, Cambridge, MA, 361–368.

Skinner, D. C. 1999. *Introduction to Decision Analysis*. Probabilistic Publishing, Gainesville, FL.

Spall, J. C. 2003. *Introduction to Stochastic Search and Optimization: Estimation, Simulation and Control*. John Wiley & Sons, Hoboken, NJ.

Sutton, R. S., A. G. Barto. 1998. *Reinforcement Learning*. MIT Press, Cambridge, MA.

Topaloglu, H., W. B. Powell. 2006. Dynamic programming approximations for stochastic, time-staged integer multicommodity flow problems. *INFORMS J. Comput.* **18**(1) 31–42.

Tsitsiklis, J. N., B. Van Roy. 1996. Feature-based methods for large scale dynamic programming. *Machine Learn.* **22**(1–3) 59–94.

Tsitsiklis, J. N., B. Van Roy. 1997. An analysis of temporal-difference learning with function approximation. *IEEE Trans. Automatic Control* **42**(5) 674–690.

Van Roy, B. 2001. Neuro-dynamic programming: Overview and recent trends. E. Feinberg, A. Shwartz, eds. *Handbook of Markov Decision Processes: Methods and Applications*. Kluwer, Boston, 431–460.

Van Roy, B., D. P. Bertsekas, Y. Lee, J. N. Tsitsiklis. 1997. A neuro-dynamic programming approach to retailer inventory management. *Proc. IEEE Conf. Decision and Control, San Diego*, Vol. 4. Institute of Electrical and Electronics Engineers, Washington, DC, 4052–4057.

Watkins, C. J. C. H., P. Dayan. 1992. Q-learning. *Machine Learn.* **8**(3–4) 279–292.

Werbos, P. J. 1990. A menu of designs for reinforcement learning over time. R. S. Sutton, W. T. Miller, P. J. Werbos, eds. *Neural Networks for Control*. MIT Press, Cambridge, MA, 67–96.