Computation and Dynamic Programming

Huseyin Topaloglu

School of Operations Research and Information Engineering,
Cornell University, Ithaca, New York 14853, USA
topaloglu@orie.cornell.edu

June 25, 2010

Abstract

Dynamic programming provides a structured framework for solving sequential decision making problems under uncertainty, but its computational appeal has traditionally been limited when dealing with problems with large state and action spaces. In this chapter, we describe a variety of methods that are targeted towards alleviating the computational difficulties associated with dynamic programming. Among the methods that we describe are simulation based and model free methods, the linear programming approach to approximate dynamic programming, approximate policy iteration, rollout policies and state aggregation. An important aspect of the methods that we describe in this chapter is that they use parsimoniously parameterized value function approximations to ensure that the approximations can be stored in a tractable fashion and they utilize simulation to avoid computing expectations explicitly.

Dynamic programming is a powerful tool for solving sequential decision making problems that take place under uncertainty. One appeal of dynamic programming is that it provides a structured approach for computing the value function, which assesses the cost implications of being in different states. The value function can ultimately be used to construct an optimal policy to control the evolution of the system over time. However, the practical use of dynamic programming as a computational tool has traditionally been limited. In many applications, the number of possible states can be so large that it becomes intractable to compute the value function for every possible value of the state. This is especially the case when the state is itself a high dimensional vector and the number of possible values for the state grows exponentially with the number of dimensions of the vector. This difficulty is compounded by the fact that computing the value function requires taking expectations and it may be difficult to estimate or store the transition probability matrices that are involved in these expectations. Finally, we need to solve optimization problems to find the best action to take in each state and these optimization problems can be intractable when the number of possible actions is large.

In this chapter, we give an overview of computational dynamic programming approaches that are directed towards addressing the difficulties described above. Our presentation begins with the value and policy iteration algorithms. These algorithms are perhaps the most standard approaches for solving dynamic programs and they provide a sound starting point for the subsequent development, but they quickly lose their tractability when the number of states or actions is large. Following these standard approaches, we turn our attention to simulation based methods, such as the temporal difference learning and Q-learning algorithms, that avoid dealing with transition probability matrices explicitly when computing expectations. We cover variants of these methods that alleviate the difficulty associated with storing the value function by using parameterized approximation architectures. The idea behind these variants is to use a succinctly parameterized representation of the value function and tune the parameters of this representation by using simulated trajectories of the system. Another approach for approximating the value function is based on solving a large linear program to tune the parameters of a parameterized approximation architecture. The appealing aspect of the linear programming approach is that it naturally provides a lower bound on the value function. There are methods that use a combination of regression and simulation to construct value function approximations. In particular, we cover the approximate policy iteration algorithm that uses regression in conjunction with simulated cost trajectories of the system to estimate the discounted cost incurred by a policy. Following this, we describe rollout policies that build on an arbitrary policy and improve the performance of this policy with reasonable amount of work. Finally, we cover state aggregation, which deals with the large number of states by partitioning them into a number of subsets and assuming that the value function takes a constant value over each subset. Given the introductory nature of this chapter and the limited space, our coverage of computational dynamic programming approaches is not exhaustive. In our conclusions, we point out other approaches and extensions, such as linear and piecewise linear value function approximations and Lagrangian relaxation based decomposition methods.

There are excellent books on approximate dynamic programming that focus on computational aspects of dynamic programming. Bertsekas and Tsitsiklis (1996) lay out the connections of dynamic programming with the stochastic approximation theory. They give convergence results for some

computational dynamic programming methods by viewing these methods as stochastic approximation algorithms. Sutton and Barto (1998) provide a perspective from the computer science side and document a large body of work done by the authors on reinforcement learning. Si, Barto, Powell and Wunsch II (2004) cover computational dynamic programming methods with significant emphasis on applications from a variety of engineering disciplines. Powell (2007) focuses on dynamic programs where the state and the action are high dimensional vectors. Such dynamic programs pose major challenges as one can neither enumerate over all possible values of the state to compute the value function nor enumerate over all possible actions to decide which action to take. The author presents methods that can use mathematical programming tools to decide which action to take and introduces a post decision state variable that cleverly bypasses the necessity to compute expectations explicitly. The development in Powell (2007) is unique in the sense that it simultaneously addresses the computational difficulties associated with the number of states, the number of actions and the necessity to compute expectations. Bertsekas (2010) provides a variety of computational dynamic programming tools. The tools in that book chapter deal with the size of the state space by using parameterized representations of the value function and avoid computing expectations by using simulated trajectories of the system. Many value function approximation approaches are rooted in standard stochastic approximation methods. Comprehensive overviews of the stochastic approximation theory can be found in Kushner and Clark (1978), Benveniste, Metivier and Priouret (1991), Bertsekas and Tsitsiklis (1996) and Kushner and Yin (2003). In this paragraph, we have broadly reviewed the literature on computational dynamic programming and value function approximation methods, but throughout the chapter, we point to the relevant literature in detail at the end of each section.

The rest of the chapter is organized as follows. In Section 1, we formulate a Markov decision problem, give a characterization of the optimal policy and briefly describe the value and policy iteration algorithms for computing the optimal policy. In Section 2, we give simulation based approaches for computing the discounted cost incurred by a policy. We describe a simple procedure for iteratively updating an approximation to the discounted cost. This updating procedure is based on a standard stochastic approximation iteration and we use the updating procedure to motivate and describe the temporal difference learning method. In Section 3, we cover the Q-learning algorithm. An important feature of the Q-learning algorithm is that it avoids dealing with the transition probability matrices when deciding which action to take. This feature becomes useful when we do not have a precise model of the system that describes how the states visited by the system evolve over time. In Section 4, we demonstrate how we can use a large scale linear program to fit an approximation to the value function. This linear program often has too many constraints to be solved directly and we describe computational methods to solve the linear program. In Section 5, we describe an approximate version of the policy iteration algorithm that uses regression to estimate the discounted cost incurred by a policy. In Section 6, we explain rollout policies and show that a rollout policy always improves the performance of the policy from which it is derived. In Section 7, we show how to use state aggregation to partition the set of states into a number of subsets and assume that the value function is constant over each subset. In Section 8, we conclude by describing some other methods and possible extensions that can further enhance the computational appeal of dynamic programming.

1 Notation and Problem Setup

In this chapter, we are interested in infinite horizon, discounted cost Markov decision problems with finite sets of states and actions, which we respectively denote by $\mathcal{S}$ and $\mathcal{U}$. If the system is in state i and we use action u, then the system moves to state j with probability $p_{ij}(u)$ and we incur a cost of $g(i,u,j)$, where $|g(i,u,j)| < \infty$. The costs in the future time periods are discounted by a factor $\alpha \in [0,1)$ per time period. We use $\mathcal{U}(i)$ to denote the set of admissible actions when the system is in state i. We assume that $\mathcal{S} = \{1,2,\ldots,n\}$ so that there are n states. For brevity, we restrict our attention to infinite horizon, discounted cost Markov decision problems, but it is possible to extend the algorithms in this chapter to finite horizon problems or average cost criterion.

A Markovian deterministic policy π is a mapping from $\mathcal{S}$ to $\mathcal{U}$ that describes which action to take for each possible state. As a result, the states visited by the system under policy π evolve according to a Markov chain with the transition probability matrix $P^\pi = \{p_{ij}(\pi(i)) : i,j \in \mathcal{S}\}$. Letting $\{i_0^\pi, i_1^\pi, \ldots\}$ be the states visited by this Markov chain, if we start in state i and use policy π, then the discounted cost that we incur can be written as

$$J^\pi(i) = \lim_{T\to\infty} \mathbb{E}\left\{ \sum_{t=0}^{T} \alpha^t \, g(i_t^\pi, \pi(i_t^\pi), i_{t+1}^\pi) \,\middle|\, i_0^\pi = i \right\}.$$

Using Π to denote the set of Markovian deterministic policies, the optimal policy π^* satisfies $J^{\pi^*}(i) = \min_{\pi\in\Pi} J^\pi(i)$ for all $i \in \mathcal{S}$, giving the minimum discounted cost starting from each state. This policy can be obtained by solving the optimality equation

$$J(i) = \min_{u\in\mathcal{U}(i)} \left\{ \sum_{j\in\mathcal{S}} p_{ij}(u) \big[g(i,u,j) + \alpha \, J(j) \big] \right\} \tag{1}$$

for $\{J(i) : i \in \mathcal{S}\}$ and letting $\pi^*(i)$ be the optimal solution to the optimization problem on the right side above. If we let $J^* = \{J^*(i) : i \in \mathcal{S}\}$ be a solution to the optimality equation in (1), then $J^*(i)$ corresponds to the optimal discounted cost when we start in state i. We refer to $J^* \in \Re^n$ as the "value function" and J^* can be interpreted as a mapping from $\mathcal{S}$ to $\Re$, giving the optimal discounted cost when we start in a particular state.

The optimality equation in (1) characterizes the optimal policy, but it does not provide an algorithmic tool for actually computing the optimal policy. We turn our attention to standard algorithmic tools that can be used to solve the optimality equation in (1).

1. Value Iteration Algorithm. Throughout the rest of this chapter, it is useful to interpret the value function J^* not only as a mapping from $\mathcal{S}$ to $\Re$, but also as an n dimensional vector whose ith component $J^*(i)$ gives the optimal discounted cost when we start in state i. For $J \in \Re^n$, we define the nonlinear operator T on $\Re^n$ as

$$[TJ](i) = \min_{u\in\mathcal{U}(i)} \left\{ \sum_{j\in\mathcal{S}} p_{ij}(u) \big[g(i,u,j) + \alpha \, J(j) \big] \right\}, \tag{2}$$

where $[TJ](i)$ denotes the ith component of the vector TJ. In this case, the optimality equation in (1) can succinctly be written as $J = TJ$ and the value function J^* is a fixed point of the operator T. The idea behind the value iteration algorithm is to find a fixed point of the operator T by starting from an initial vector $J^1 \in \Re^n$ and successively applying the operator T. In particular, the value iteration algorithm generates a sequence of vectors $\{J^k\}_k$ with $J^{k+1} = TJ^k$.

It is possible to show that the operator T is a contraction mapping on $\Re^n$ so that it has a unique fixed point and the sequence of vectors $\{J^k\}_k$ with $J^{k+1} = TJ^k$ converge to the unique fixed point of the operator T. Since the value function J^* is a fixed point of the operator T, the value iteration algorithm indeed converges to J^*.

2. Policy Iteration Algorithm. The sequence of vectors $\{J^k\}_k$ generated by the value iteration algorithm do not necessarily correspond to the discounted cost $J^\pi = \{J^\pi(i) : i \in \mathcal{S}\}$ incurred by some policy π. In contrast, the policy iteration algorithm generates a sequence of vectors that correspond to the discounted costs incurred by different policies. To describe the policy iteration algorithm, for $J \in \Re^n$ and $\pi \in \Pi$, we define the linear operator T_π on $\Re^n$ as

$$[T_\pi J](i) = \sum_{j \in \mathcal{S}} p_{ij}(\pi(i)) \big[g(i, \pi(i), j) + \alpha\, J(j) \big]. \tag{3}$$

In this case, it is possible to show that the discounted cost J^π incurred by policy π is a fixed point of the operator T_π satisfying $J^\pi = T_\pi J^\pi$. Since the operator T_π is linear, we can find a fixed point of this operator by solving a system of linear equations. In particular, if we let P^π be the transition probability matrix $\{p_{ij}(\pi(i)) : i, j \in \mathcal{S}\}$, $g^\pi(i) = \sum_{j \in \mathcal{S}} p_{ij}(\pi(i))\, g(i, \pi(i), j)$ be the expected cost that we incur when we are in state i and follow policy π and $g^\pi = \{g^\pi(i) : i \in \mathcal{S}\}$ be the vector of expected costs, then $T_\pi J$ in (3) can be written in vector notation as

$$T_\pi J = g^\pi + \alpha\, P^\pi J.$$

Therefore, we can find a J^π that satisfies $J^\pi = T_\pi J^\pi$ by solving the system of linear equations $J^\pi = g^\pi + \alpha\, P^\pi J^\pi$, which has the solution $J^\pi = (I - \alpha\, P^\pi)^{-1} g^\pi$, where I denotes the $n \times n$ identity matrix and the superscript -1 denotes the matrix inverse. The policy iteration algorithm computes the optimal policy by starting from an initial policy π^1 and generating a sequence of policies $\{\pi^k\}_k$ through the following two steps.

Step 1. (Policy Evaluation) Compute the discounted cost J^{π^k} incurred by policy π^k, possibly by solving the system of linear equations $J^{\pi^k} = g^{\pi^k} + \alpha\, P^{\pi^k} J^{\pi^k}$.

Step 2. (Policy improvement) For all $i \in \mathcal{S}$, let policy π^{k+1} be defined such that we have

$$\pi^{k+1}(i) = \operatorname*{argmin}_{u \in \mathcal{U}(i)} \left\{ \sum_{j \in \mathcal{S}} p_{ij}(u) \left[g(i, u, j) + \alpha\, J^{\pi^k}(j) \right] \right\}. \tag{4}$$

Noting the definition of the operator T, policy π^{k+1} that is obtained in Step 2 satisfies $TJ^{\pi^k} = T_{\pi^{k+1}} J^{\pi^k}$. The algorithm stops when the policies at two successive iterations satisfy $\pi^k = \pi^{k+1}$. If we have $\pi^k = \pi^{k+1}$, then we obtain $TJ^{\pi^k} = T_{\pi^{k+1}} J^{\pi^k} = T_{\pi^k} J^{\pi^k} = J^{\pi^k}$ so that J^{π^k} is a fixed point

of the operator T, which implies that J^{π^k} is the solution to the optimality equation in (1). In the last chain of equalities, the first equality follows by the definition of the policy iteration algorithm, the second equality follows by the fact that $\pi^{k+1} = \pi^k$ at termination and the last equality follows by the fact that the discounted cost incurred by policy π^k is the fixed point of the operator T_{π^k}.

The value and policy iteration algorithms are perhaps the most standard approaches for solving Markov decision problems, but they quickly lose their computational tractability. If the number of states is large, then computing and storing J^k in the value iteration algorithm and J^{π^k} in the policy iteration algorithm becomes difficult. Furthermore, if the number of actions is large, then solving the optimization problems in (2) and (4) becomes intractable. Finally, solving these optimization problems requires computing expectations according to certain transition probability matrices and leaving the computation of such expectations aside, it can be very costly to even estimate and store the transition probability matrices. Throughout this chapter, we try to resolve these difficulties by giving algorithms that tend to be computationally more appealing than the value and policy iteration algorithms.

The books by Puterman (1994) and Bertsekas (2001) are modern and complete references on the theory of Markov decision processes. The notation that we use in this chapter follows the one in Bertsekas (2001) closely. Early analysis of the value iteration algorithm is attributed to Shapley (1953) and Blackwell (1965), whereas the policy iteration algorithm dates back to Bellman (1957) and Howard (1960). Bellman (1957) coins the term "curse of dimensionality" to refer to the difficulty associated with the large number of states, especially when the state is itself a high dimensional vector. Powell (2007) observes that not only the number of states, but also the number of admissible actions and the necessity to compute expectations can create computational difficulties.

There are a number of extensions for the value and policy iteration algorithms. It is possible to characterize the convergence rate of the value iteration algorithm and use this result to estimate the number of iterations to compute the value function with a certain precision. There is a Gauss Seidel variant of the value iteration algorithm that applies the operator T in an asynchronous manner. Modified policy iteration algorithm evaluates a policy only approximately in the policy evaluation step. There are methods to eliminate the suboptimal actions and such methods ensure that the operator T can be applied faster. Furthermore, the sequence of vectors generated by the value iteration algorithm converge to the value function only in the limit, but the action elimination methods may allow identifying the optimal policy without waiting for the value iteration algorithm to converge to the value function. We focus on problems with finite state and action spaces under the discounted cost criterion, but extensions to more general state and action spaces under average cost and undiscounted cost criteria are possible. Bertsekas and Shreve (1978), Sennott (1989), Puterman (1994), Bertsekas (2001) and Bertsekas (2010) are good references to explore the extensions that we mention in this paragraph.

2 Policy Evaluation with Monte Carlo Simulation

In many cases, it is necessary to compute the discounted cost J^π incurred by some policy π. Computing the discounted cost incurred by a policy requires solving a system of linear equations of the form

$J^\pi = g^\pi + \alpha P^\pi J^\pi$ and solving this system may be difficult, especially when the dimensions of the matrices are too large or when we simply do not have the data to estimate the transition probability matrix P^π. In these situations, we may want to use simulation to estimate the discounted cost incurred by a particular policy.

Assume that we generate a sequence of states $\{i_0, i_1, \ldots\}$ by following policy π, which is the policy that we want to evaluate. The initial state i_0 is chosen arbitrarily and the state transitions are sampled from the transition probability matrix P^π associated with policy π. Noting the discussion in the paragraph above, we may not have access to the full transition probability matrix P^π, but the sequence of states $\{i_0, i_1, \ldots\}$ may be generated either by experimenting with the system in real time or by making use of the data related to the evolution of the system in the past. Since the sequence of states $\{i_0, i_1, \ldots\}$ come from the transition probability matrix P^π, the discounted cost J^π incurred by policy π satisfies $J^\pi(i_k) = \mathbb{E}\{g(i_k, \pi(i_k), i_{k+1}) + \alpha J^\pi(i_{k+1})\}$. In this case, we can keep a vector $J \in \Re^n$ as an approximation to J^π and update one component of this vector after every state transition by

$$J(i_k) \leftarrow (1-\gamma)\, J(i_k) + \gamma\, [g(i_k, \pi(i_k), i_{k+1}) + \alpha\, J(i_{k+1})], \tag{5}$$

where $\gamma \in [0, 1]$ is a step size parameter. The idea behind the updating procedure above is that $J(i_k)$ on the right side of (5) is our estimate of $J^\pi(i_k)$ just before the kth state transition. Noting that $J^\pi(i_k) = \mathbb{E}\{g(i_k, \pi(i_k), i_{k+1}) + \alpha\, J^\pi(i_{k+1})\}$, if J and J^π are close to each other, then $g(i_k, \pi(i_k), i_{k+1}) + \alpha\, J(i_{k+1})$ can be interpreted as a noisy estimate $J^\pi(i_k)$ that we obtain during the kth state transition. Therefore, the updating procedure in (5) combines the current estimate and the new noisy estimate by putting the weights $1-\gamma$ and γ on them. This kind of an updating procedure is motivated by the standard stochastic approximation theory and if the step size parameter γ converges to zero at an appropriate rate, then it is possible to show that the vector J converges to J^π with probability one as long as every state is sampled infinitely often.

The simulation based approach outlined above does not require storing the transition probability matrix, but it still requires storing the n dimensional vector J, which can be problematic when the number of states is large. One approach to alleviate this storage requirement is to use a parameterized approximation architecture. In particular, we approximate $J^\pi(i)$ with

$$\tilde{J}(i, r) = \sum_{\ell=1}^{L} r_\ell\, \phi_\ell(i), \tag{6}$$

where $\{\phi_\ell(\cdot) : \ell = 1, \ldots, L\}$ are fixed functions specified by the model builder and $r = \{r_\ell : \ell = 1, \ldots, L\}$ are adjustable parameters. We can view $\{\phi_\ell(i) : \ell = 1, \ldots, L\}$ as the features of state i that are combined in a linear fashion to form an approximation to $J^\pi(i)$. For this reason, it is common to refer to $\{\phi_\ell(\cdot) : \ell = 1, \ldots, L\}$ as the "feature functions." The goal is to tune the adjustable parameters r so that $\tilde{J}(\cdot, r)$ is a good approximation to J^π. There are a number of algorithms to tune the adjustable parameters. One simulation based approach iteratively updates the vector $r = \{r_\ell : \ell = 1, \ldots, L\} \in \Re^L$ after every state transition by

$$\begin{aligned} r &\leftarrow r + \gamma \big[g(i_k, \pi(i_k), i_{k+1}) + \alpha\, \tilde{J}(i_{k+1}, r) - \tilde{J}(i_k, r) \big]\, \nabla_r \tilde{J}(i_k, r) \\ &= r + \gamma \big[g(i_k, \pi(i_k), i_{k+1}) + \alpha\, \tilde{J}(i_{k+1}, r) - \tilde{J}(i_k, r) \big]\, \phi(i_k) \end{aligned} \tag{7}$$

where we use $\nabla_r \tilde{J}(i_k, r)$ to denote the gradient of $\tilde{J}(i_k, r)$ with respect to r and $\phi(i_k)$ to denote the L dimensional vector $\{\phi_\ell(i_k) : \ell = 1, \ldots, L\}$. The equality above simply follows by the definition of $\tilde{J}(i_k, r)$ given in (6).

Bertsekas and Tsitsiklis (1996) justify the updating procedure in (7) as a stochastic gradient iteration to minimize the squared deviation between the discounted cost approximation $\tilde{J}(\cdot, r)$ and the simulated cost trajectory of the system. They are able to provide convergence results for this updating procedure, but as indicated by Powell (2007), one should be careful about the choice of the step size parameter γ to obtain desirable empirical convergence behavior. Another way to build intuition into the updating procedure in (7) is to consider the case where $L = n$ and $\phi_\ell(i) = 1$ whenever $\ell = i$ and $\phi_\ell(i) = 0$ otherwise, which implies that we have $\tilde{J}(i, r) = r_i$ for all $i \in \{1, \ldots, n\}$. This situation corresponds to the case where the number of feature functions is as large as the number of states and the discounted cost approximation $\tilde{J}(\cdot, r)$ in (6) can exactly capture the actual discounted cost J^π. In this case, $\phi(i_k)$ becomes the n dimensional unit vector with a one in the i_kth component and the updating procedure in (7) becomes $r_{i_k} \leftarrow r_{i_k} + \gamma\big[g(i_k, \pi(i_k), i_{k+1}) + \alpha\, r_{i_{k+1}} - r_{i_k}\big]$. We observe that the last updating procedure can be written as $r_{i_k} \leftarrow (1 - \gamma)\, r_{i_k} + \gamma\big[g(i_k, \pi(i_k), i_{k+1}) + \alpha\, r_{i_{k+1}}\big]$, which is identical to (5). Therefore, if the number of feature functions is as large as the number of states, then the updating procedures in (5) and (7) become equivalent.

We can build on the ideas above to construct more sophisticated tools for simulation based policy evaluation. The preceding development is based on the fact that J^π satisfies $J^\pi(i_k) = \mathbb{E}\{g(i_k, \pi(i_k), i_{k+1}) + \alpha\, J^\pi(i_{k+1})\}$, but it is also possible to show that J^π satisfies

$$J^\pi(i_k) = \mathbb{E}\left\{ \sum_{l=k}^{\tau-1} (\lambda\, \alpha)^{l-k} \left[g(i_l, \pi(i_l), i_{l+1}) + \alpha\, J^\pi(i_{l+1}) - J^\pi(i_l) \right] \right\} + J^\pi(i_k)$$

for any stopping time τ and any $\lambda \in (0, 1]$. The identity above can be seen by noting that the expectation of the expression in the square brackets is zero, but a more rigorous derivation can be found in Bertsekas and Tsitsiklis (1996). This identity immediately motivates the following stochastic approximation approach to keep and update an approximation $J \in \Re^n$ to the discounted cost J^π. We generate a sequence of states $\{i_0, i_1, \ldots, i_\tau\}$ by following policy π until the stopping time τ. Once the entire simulation is over, letting $d_l = g(i_l, \pi(i_l), i_{l+1}) + \alpha\, J(i_{l+1}) - J(i_l)$ for notational brevity, we update the approximation $J \in \Re^n$ by

$$J(i_k) \leftarrow J(i_k) + \gamma \sum_{l=k}^{\tau-1} (\lambda\, \alpha)^{l-k}\, d_l \tag{8}$$

for all $\{i_k : k = 0, \ldots, \tau - 1\}$. After updating J, we generate another sequence of states until the stopping time τ and continue in a similar fashion. When $\lambda = 1$ and τ deterministically takes value one, the updating procedures in (5) and (8) become equivalent.

The quantity d_l is referred to as the "temporal difference" and the updating procedures similar to that in (8) are called "temporal difference learning." For different values of $\lambda \in [0, 1]$, we obtain a whole class of algorithms and this class of algorithms is commonly denoted as TD(λ). The approach that we describe above works in batch mode in the sense that it updates the approximation for a number of

states after each simulation trajectory is over and it can be viewed as an offline version. There are also online versions of TD(λ) that update the approximation after every state transition. If the number of states is large, then carrying out the updating procedure in (8) for TD(λ) can be difficult as it requires storing the n dimensional vector J. It turns out that one can use a parameterized approximation architecture similar to the one in (6) to alleviate the storage problem. In this case, we update the L dimensional vector r of adjustable parameters by using

$$r \leftarrow r + \gamma \sum_{k=0}^{\tau-1} \nabla_r \tilde{J}(i_k, r) \sum_{l=k}^{\tau-1} (\lambda\,\alpha)^{l-k}\, d_l, \tag{9}$$

where the temporal difference above is defined as $d_l = g(i_l, \pi(i_l), i_{l+1}) + \alpha\, \tilde{J}(i_{l+1}, r) - \tilde{J}(i_l, r)$. Similar to the updating procedure in (7), we can build intuition for the updating procedure in (9) by considering the case where $L = n$ and $\phi_\ell(i) = 1$ whenever $\ell = i$ and $\phi_\ell(i) = 0$ otherwise. In this case, it is not difficult to see that the updating procedures in (8) and (9) become equivalent.

Temporal difference learning has its origins in Sutton (1984) and Sutton (1988). The presentation in this section is based on Bertsekas and Tsitsiklis (1996), where the authors give convergence results for numerous versions of TD(λ), including online and offline versions with different values for λ, applied to infinite horizon discounted cost and stochastic shortest path problems. The book by Sutton and Barto (1998) is another complete reference on temporal difference learning. Tsitsiklis and Van Roy (1997) analyze temporal difference learning with parameterized approximation architectures.

3 Model Free Learning

Assuming that we have a good approximation to the value function, to be able to choose an action by using this value function approximation, we need to replace the vector J in the right side of the optimality equation in (1) with the value function approximation and solve the resulting optimization problem. This approach requires computing an expectation that involves the transition probabilities $\{p_{ij}(u) : i, j \in \mathcal{S},\ u \in \mathcal{U}\}$. We can try to estimate this expectation by using simulation, but it is natural to ask whether we can come up with a method that bypasses estimating expectations explicitly. This is precisely the goal of model free learning, or more specifically the Q-learning algorithm.

The Q-learning algorithm is based on an alternative representation of the optimality equation in (1). If we let $Q(i,u) = \sum_{j\in\mathcal{S}} p_{ij}(u)\big[g(i,u,j) + \alpha\, J(j)\big]$, then (1) implies that $J(i) = \min_{u\in\mathcal{U}(i)} Q(i,u)$ and we can write the optimality equation in (1) as

$$Q(i,u) = \sum_{j\in\mathcal{S}} p_{ij}(u)\big[g(i,u,j) + \alpha \min_{v\in\mathcal{U}(j)} Q(j,v)\big]. \tag{10}$$

In this case, if $Q^* = \{Q^*(i,u) : i \in \mathcal{S},\ u \in \mathcal{U}(i)\}$ is a solution to the optimality equation above, then it is optimal to take the action $\mathrm{argmin}_{u\in\mathcal{U}(i)} Q^*(i,u)$ when the system is in state i. One interpretation of $Q^*(i,u)$ is that it is the optimal discounted cost given that the system starts in state i and the first action is u. The fundamental idea behind the Q-learning algorithm is to solve the optimality equation in (10) by using a stochastic approximation iteration. The algorithm generates a sequence of states

and actions $\{i_0, u_0, i_1, u_1, \ldots\}$ such that $u_k \in \mathcal{U}(i_k)$ for all $k = 0, 1, \ldots$. It keeps an approximation $Q \in \Re^{n \times |\mathcal{U}|}$ to Q^* and updates this approximation after every state transition by

$$Q(i_k, u_k) \leftarrow (1 - \gamma)\, Q(i_k, u_k) + \gamma \big[g(i_k, u_k, s_k) + \alpha \min_{v \in \mathcal{U}(s_k)} Q(s_k, v) \big], \tag{11}$$

where the successor state s_k of i_k is sampled according to the probabilities $\{p_{i_k,j}(u_k) : j \in \mathcal{S}\}$. The rationale behind the updating procedure above is similar to the one in (5) in the sense that $Q(i_k, u_k)$ on the right side of (11) is our current estimate of $Q^*(i_k, u_k)$ just before the kth state transition and the expression in the square brackets is our new noisy estimate of $Q^*(i_k, u_k)$. If every state action pair is sampled infinitely often in the trajectory of the system and the step size parameter γ satisfies certain conditions, then it can be shown that the approximation kept by the Q-learning algorithm converges to the solution to the optimality equation in (10).

For the convergence result to hold, the sequence of states and actions $\{i_0, u_0, i_1, u_1, \ldots\}$ can be sampled in an arbitrary manner as long as every state action pair is sampled infinitely often. However, the Q-learning algorithm is often used to control the real system as it evolves in real time. In such situations, given that the system is currently in state i_k, it is customary to choose $u_k = \text{argmin}_{u \in \mathcal{U}(i_k)} Q(i_k, u)$, where Q is the current approximation to Q^*. The hope is that if the approximation Q is close to Q^*, then the action u_k is the optimal action when the system is in state i_k. After implementing the action u_k, the system moves to some state i_{k+1} and it is also customary to choose the successor state s_k in the updating procedure in (11) as i_{k+1}. The reason behind this choice is that after implementing the action u_k in state i_k, the system naturally moves to state i_{k+1} according to the transition probabilities $\{p_{i_k,j}(u_k) : j \in \mathcal{S}\}$ and choosing $s_k = i_{k+1}$ ensures that the successor state s_k is also chosen according to these transition probabilities. To ensure that every state action pair is sampled infinitely often, with a small probability, the action u_k is randomly chosen among the admissible actions instead of choosing $u_k = \text{argmin}_{u \in \mathcal{U}(i_k)} Q(i_k, u)$. Similarly, with a small probability, the system is forced to move to a random state. This small probability is referred to as the "exploration probability."

An important advantage of the Q-learning algorithm is that once we construct a good approximation Q, if the system is in state i, then we can simply take the action $\text{argmin}_{u \in \mathcal{U}(i)} Q(i, u)$. In this way, we avoid dealing with transition probability matrices when choosing an action. However, the Q-learning algorithm still requires storing an $n \times |\mathcal{U}|$ dimensional approximation, which can be quite large in practical applications. There is a commonly used, albeit a heuristic, variant of the Q-learning algorithm that uses a parameterized approximation architecture similar to the one in (6). Letting $\tilde{Q}(i, u, r)$ be an approximation to $Q^*(i, u)$ parameterized by the L dimensional vector r of adjustable parameters, this variant uses the updating procedure

$$r \leftarrow r + \gamma \big[g(i_k, u_k, s_k) + \alpha \min_{v \in \mathcal{U}(s_k)} \tilde{Q}(s_k, v, r) - \tilde{Q}(i_k, u_k, r) \big] \nabla_r \tilde{Q}(i_k, u_k, r) \tag{12}$$

to tune the adjustable parameters. Bertsekas and Tsitsiklis (1996) provide a heuristic justification for the updating procedure in (12). In (11), if we have $g(i_k, u_k, s_k) + \alpha \min_{v \in \mathcal{U}(s_k)} Q(s_k, v) \geq Q(i_k, u_k)$, then we increase the value of $Q(i_k, u_k)$. Similarly, in (12), if we have $g(i_k, u_k, s_k) + \alpha \min_{v \in \mathcal{U}(s_k)} \tilde{Q}(s_k, v, r) \geq \tilde{Q}(i_k, u_k, r)$, then we would like to increase the value of $\tilde{Q}(i_k, u_k, r)$, but we can change the value of $\tilde{Q}(i_k, u_k, r)$ only by changing the adjustable parameters r. In (12), if we have $g(i_k, u_k, s_k) +$

$\alpha \min_{v \in \mathcal{U}(s_k)} \tilde{Q}(s_k, v, r) \geq \tilde{Q}(i_k, u_k, r)$ and the ℓth component of $\nabla_r \tilde{Q}(i_k, u_k, r)$ is nonnegative so that $\tilde{Q}(i_k, u_k, r)$ is an increasing function of r_ℓ, then we increase the ℓth component of r. The hope is that this changes the ℓth component of r in the right direction so that $\tilde{Q}(i_k, u_k, r)$ also increases. Of course, this is not guaranteed in general since all of the components of r change simultaneously in the updating procedure in (11). As a result, the updating procedure largely remains a heuristic.

The Q-learning algorithm was proposed by Watkins (1989) and Watkins and Dayan (1992). Barto, Bradtke and Singh (1995), Sutton and Barto (1998) and Si et al. (2004) give comprehensive overviews of the research revolving around the Q-learning algorithm. Tsitsiklis (1994) and Bertsekas and Tsitsiklis (1996) show that the Q-learning algorithm fits within the general framework of stochastic approximation methods and provide convergence results by building on and extending the standard stochastic approximation theory. The updating procedure in (11) has convergence properties, but the one in (12) with a parameterized approximation architecture is a heuristic. Bertsekas and Tsitsiklis (1996) indicate that the updating procedure in (12) has convergence properties when the approximation architecture corresponds to state aggregation, where the entire set of state action pairs is partitioned into L subsets $\{\mathcal{X}_\ell : \ell = 1, \ldots, L\}$ and the feature functions in the approximation architecture $\tilde{Q}(i, u, r) = \sum_{\ell=1}^{L} r_\ell \, \phi_\ell(i, u)$ satisfy $\phi_\ell(i, u) = 1$ whenever $(i, u) \in \mathcal{X}_\ell$ and $\phi_\ell(i, u) = 0$ otherwise. Furthermore, they note that the updating procedure in (12) is also convergent for certain optimal stopping problems. Tsitsiklis and Van Roy (2001) apply the Q-learning algorithm to optimal stopping problems arising from the option pricing setting. Kunnumkal and Topaloglu (2008) and Kunnumkal and Topaloglu (2009) use projections within the Q-learning algorithm to exploit the known structural properties of the value function so as to improve the empirical convergence rate. The projections used by Kunnumkal and Topaloglu (2008) are with respect to the L-2 norm and they can be computed as solutions to least squares regression problems. The authors exploit the fact that solutions to least squares regression problems can be computed fairly easily and one can even come up with explicit expressions for the solutions to least squares regression problems.

4 Linear Programming Approach

Letting $\{\theta(i) : i \in \mathcal{S}\}$ be an arbitrary set of strictly positive scalars that add up to $1 - \alpha$, the solution to the optimality equation in (1) can be found by solving the linear program

$$\max \quad \sum_{i \in \mathcal{S}} \theta(i) \, J(i) \tag{13}$$

$$\text{subject to} \quad J(i) - \sum_{j \in \mathcal{S}} \alpha \, p_{ij}(u) \, J(j) \leq \sum_{j \in \mathcal{S}} p_{ij}(u) \, g(i, u, j) \qquad i \in \mathcal{S}, \; u \in \mathcal{U}(i), \tag{14}$$

where the decision variables are $J = \{J(i) : i \in \mathcal{S}\}$. To see this, we note that a feasible solution J to the linear program satisfies $J(i) \leq \min_{u \in \mathcal{U}(i)} \{\sum_{j \in \mathcal{S}} p_{ij}(u) \, [g(i, u, j) + \alpha \, J(j)]\}$ for all $i \in \mathcal{S}$ so that we have $J \leq TJ$. It is a standard result in the Markov decision process literature that the operator T is monotone in the sense that if $J \leq J'$, then $TJ \leq TJ'$. Therefore, we obtain $J \leq TJ \leq T^2 J$ for any feasible solution J, where the last inequality follows by noting that $J \leq TJ$ and applying the operator T on both sides of the inequality. Continuing in this fashion, we obtain $J \leq T^k J$ for any

$k = 0, 1, \ldots$. In this case, letting J^* be the solution to the optimality equation in (1), since the value iteration algorithm implies that $\lim_{k\to\infty} T^k J = J^*$, we obtain $J \leq J^*$ for any feasible solution J to the linear program, which yields $\sum_{i\in\mathcal{S}} \theta(i)\, J(i) \leq \sum_{i\in\mathcal{S}} \theta(i)\, J^*(i)$. Therefore, $\sum_{i\in\mathcal{S}} \theta(i)\, J^*(i)$ is an upper bound on the optimal objective value of the linear program. We can check that J^* is a feasible solution to the linear program, in which case, J^* should also be the optimal solution. Associating the dual multipliers $\{x(i,u) : i \in \mathcal{S},\ u \in \mathcal{U}(i)\}$ with the constraints, the dual of the linear program is

$$\min \quad \sum_{i\in\mathcal{S}} \sum_{u\in\mathcal{U}(i)} \sum_{j\in\mathcal{S}} p_{ij}(u)\, g(i,u,j)\, x(i,u) \tag{15}$$

$$\text{subject to} \quad \sum_{u\in\mathcal{U}(i)} x(i,u) - \sum_{j\in\mathcal{S}} \sum_{u\in\mathcal{U}(j)} \alpha\, p_{ji}(u)\, x(j,u) = \theta(i) \qquad i \in \mathcal{S} \tag{16}$$

$$x(i,u) \geq 0 \qquad i \in \mathcal{S},\ u \in \mathcal{U}(i). \tag{17}$$

In problem (15)-(17), the decision variable $x(i,u)$ is interpreted as the stationary probability that we are in state i and take action u. In particular, if we add the first set of constraints over all $i \in \mathcal{S}$ and note that $\sum_{i\in\mathcal{S}} p_{ji}(u) = 1$ and $\sum_{i\in\mathcal{S}} \theta(i) = 1 - \alpha$, we obtain $\sum_{i\in\mathcal{S}} \sum_{u\in\mathcal{U}(i)} x(i,u) = 1$ so that $x = \{x(i,u) : i \in \mathcal{S},\ u \in \mathcal{U}(i)\}$ can indeed be interpreted as probabilities. If we let (J^*, x^*) be an optimal primal dual solution pair to problems (13)-(14) and (15)-(17), then by the discussion above, J^* is the solution to the optimality equation in (1). Furthermore, if $x^*(i,u) > 0$, then we obtain

$$\min_{v\in\mathcal{U}(i)} \left\{ \sum_{j\in\mathcal{S}} p_{ij}(v) \big[g(i,v,j) + \alpha\, J^*(j) \big] \right\} = J^*(i) = \sum_{j\in\mathcal{S}} p_{ij}(u) \big[g(i,u,j) + \alpha\, J^*(j) \big],$$

where the second equality follows from the complementary slackness condition for the constraint in problem (13)-(14) that corresponds to $x(i,u)$ and the fact that $x^*(i,u) > 0$. Therefore, if we have $x^*(i,u) > 0$ in the optimal solution to problem (15)-(17), then it is optimal to take action u whenever the system is in state i.

Problem (13)-(14) has n decision variables and $n \times |\mathcal{U}|$ constraints, which can both be very large for practical applications. One way to overcome the difficulty associated with the large number of decision variables is to use a parameterized approximation architecture as in (6). To choose the adjustable parameters $\{r_\ell : \ell = 1, \ldots, L\}$ in the approximation architecture, we plug $\sum_{\ell=1}^{L} r_\ell\, \phi_\ell(i)$ for $J(i)$ in problem (13)-(14) and solve the linear program

$$\max \quad \sum_{i\in\mathcal{S}} \sum_{\ell=1}^{L} \theta(i)\, r_\ell\, \phi_\ell(i) \tag{18}$$

$$\text{subject to} \quad \sum_{\ell=1}^{L} r_\ell\, \phi_\ell(i) - \sum_{j\in\mathcal{S}} \sum_{\ell=1}^{L} \alpha\, p_{ij}(u)\, r_\ell\, \phi_\ell(j) \leq \sum_{j\in\mathcal{S}} p_{ij}(u)\, g(i,u,j) \qquad i \in \mathcal{S},\ u \in \mathcal{U}(i), \tag{19}$$

where the decision variables are $r = \{r_\ell : \ell = 1, \ldots, L\}$. The number of decision variables in problem (18)-(19) are L, which can be manageable, but the number of constraints is still $n \times |\mathcal{U}|$. In certain cases, it may be possible to solve the dual of problem (18)-(19) by using column generation, but this requires a lot of structure in the column generation subproblems. One other approach is to randomly sample some state action pairs and include the constraints only for the sampled state action pairs.

An important feature of problem (18)-(19) is that it naturally provides a lower bound on the optimal discounted cost. In particular, if r^* is an optimal solution to problem (18)-(19), then letting $\tilde{J}(i,r^*) = \sum_{\ell=1}^{L} r_\ell^* \, \phi_\ell(i)$, we observe that $\{\tilde{J}(i,r^*) : i \in \mathcal{S}\}$ is a feasible solution to problem (13)-(14), in which case, the discussion at the beginning of this section implies that $\tilde{J}(\cdot,r^*) \leq J^*$. Such lower bounds on the optimal discounted cost can be useful when we try to get a feel for the optimality gap of a heuristic policy. On the other hand, an undesirable aspect of the approximation strategy in problem (18)-(19) is that the quality of the approximation that we obtain from this problem can depend on the choice of $\theta = \{\theta(i) : i \in \mathcal{S}\}$. We emphasize that this is in contrast to problem (13)-(14), which computes the optimal discounted cost for an arbitrary choice of θ. The paper by de Farias and Weber (2008) investigates the important question of how to choose θ and more work is needed in this area. Letting $\tilde{J}(\cdot,r) = \sum_{\ell=1}^{L} r_\ell \, \phi_\ell(\cdot)$ and using $\|\cdot\|_\theta$ to denote the θ weighted L-1 norm on $\Re^n$, de Farias and Van Roy (2003*b*) show that problem (18)-(19) computes an r that minimizes the error $\|J^* - \tilde{J}(\cdot,r)\|_\theta$ subject to the constraint that $\tilde{J}(\cdot,r) \leq T\tilde{J}(\cdot,r)$. This result suggests that we should use larger values of $\theta(i)$ for the states at which we want to approximate the value function more accurately. These states may correspond to the ones that we visit frequently or the ones that have serious cost implications. For this reason, $\{\theta(i) : i \in \mathcal{S}\}$ are referred to as the "state relevance weights." One common idea for choosing the state relevance weights is to simulate the trajectory of a reasonable policy and choose the state relevance weights as the frequency with which we visit different states. There are also some approximation guarantees for problem (18)-(19). Letting r^* be an optimal solution to this problem, de Farias and Van Roy (2003*b*) show that

$$\|J^* - \tilde{J}(\cdot,r^*)\|_\theta \leq \frac{2}{1-\alpha} \min_r \|J^* - \tilde{J}(\cdot,r)\|_\infty. \tag{20}$$

The left side above is the error in the value function approximation provided by problem (18)-(19), whereas the right side above is the smallest error possible in the value function approximation if we were allowed to use any value for r. The right side can be viewed as the power of the parameterized approximation architecture in (6). Therefore, (20) shows that if our approximation architecture is powerful, then we expect problem (18)-(19) to return a good value function approximation.

The linear programming approach for constructing value function approximations was proposed by Schweitzer and Seidmann (1985) and it has seen revived interest with the work of de Farias and Van Roy (2003*b*). Noting that the right side of (20) measures the distance between J^* and $\tilde{J}(\cdot,r)$ according to the max norm and it can be quite large in practice, de Farias and Van Roy (2003*b*) provide refinements on this approximation guarantee. Since the number of constraints in problem (18)-(19) can be large, de Farias and Van Roy (2004) study the effectiveness of randomly sampling some of the constraints to approximately solve this problem. Adelman and Mersereau (2008) compare the linear programming approach to other value function approximation methods. Desai, Farias and Moallemi (2009) observe that relaxing constraints (19) may actually improve the quality of the approximation and they propose relaxing these constraints subject to a budget on the total relaxation amount. The papers by de Farias and Van Roy (2003*a*) and de Farias and Van Roy (2006) extend the linear programming approach to average cost criterion. Adelman (2007) and Veatch and Walker (2008) give examples in the revenue management and queueing control settings, where the column generation subproblem for the dual of problem (18)-(19) becomes tractable.

5 Approximate Policy Iteration

In practice, both steps of the policy iteration algorithm in Section 1 can be problematic. In the policy evaluation step, we need to compute the discounted cost incurred by a policy π, potentially by solving the system of linear equations $J = T_\pi J$ for $J \in \Re^n$. In the policy improvement step, we need to find a policy π that satisfies $TJ = T_\pi J$ for a given vector $J \in \Re^n$. Carrying out these two steps exactly is almost never tractable, as it requires dealing with large dimensional matrices.

It is possible to use regression and simulation methods, along with parameterized approximation architectures as in (6), to approximately carry out the policy iteration algorithm. In the approximate version of the policy evaluation step, letting π^k be the policy that we need to evaluate, we generate a sequence of states $\{i_0, i_1, \ldots\}$ by following policy π^k. If we let $C_l = \sum_{t=l}^{\infty} \alpha^{t-l} g(i_t, \pi^k(i_t), i_{t+1})$ for all $l = 0, 1, \ldots$, then C_l provides a sample of the discounted cost incurred by policy π^k given that we start in state i_l. Therefore, letting r^k be the solution to the regression problem

$$\min_r \left\{ \sum_{l=0}^{\infty} \left[\tilde{J}(i_l, r) - C_l \right]^2 \right\} = \min_r \left\{ \sum_{l=0}^{\infty} \Big[\sum_{\ell=1}^{L} r_\ell \, \phi_\ell(i_l) - C_l \Big]^2 \right\}, \tag{21}$$

we can use $\tilde{J}(\cdot, r^k)$ as an approximation to the discounted cost J^{π^k} incurred by policy π^k. In the policy improvement step, on the other hand, we need to find a policy π^{k+1} such that $\pi^{k+1}(i)$ is the optimal solution to the problem $\min_{u \in \mathcal{U}(i)} \{\sum_{j \in \mathcal{S}} p_{ij}(u) \left[g(i, u, j) + \alpha \, J^{\pi^k}(j) \right]\}$ for all $i \in \mathcal{S}$. In the approximate version of the policy improvement step, we let policy π^{k+1} be such that $\pi^{k+1}(i)$ is the optimal solution to the problem $\min_{u \in \mathcal{U}(i)} \{\sum_{j \in \mathcal{S}} p_{ij}(u) \left[g(i, u, j) + \alpha \, \tilde{J}(j, r^k) \right]\}$ for all $i \in \mathcal{S}$. If the expectation inside the curly brackets cannot be computed exactly, then we resort to simulation. We note that J^{π^k} used in the policy improvement step of the policy iteration algorithm corresponds to the discounted cost incurred by policy π^k, whereas $\tilde{J}(\cdot, r^k)$ used in the policy improvement step of the approximate policy iteration algorithm does not necessarily correspond to the discounted cost incurred by policy π^k. It is also worthwhile to point out that since we need the action taken by policy π^k when generating the sequence of states $\{i_0, i_1, \ldots\}$, we do not have to compute the action taken by policy π^k in every possible state. We can simply compute the action taken by policy π^k as needed when generating the sequence of states $\{i_0, i_1, \ldots\}$.

There is some theoretical justification for the approximate policy iteration algorithm. We observe that there are two potential sources of error in the algorithm. First, the regression in the policy evaluation step may not accurately capture the discounted cost incurred by policy π^k. We assume that this error is bounded by ϵ in the sense that $|\tilde{J}(i, r^k) - J^{\pi^k}(i)| \leq \epsilon$ for all $i \in \mathcal{S}$ and $k = 0, 1, \ldots$. Second, we may not exactly compute the action taken by policy π^{k+1} in the policy improvement step. This is especially the case when we use simulation to estimate the expectations. We assume that this error is bounded by δ in the sense that $|[T_{\pi^{k+1}} \tilde{J}(\cdot, r^k)](i) - [T\tilde{J}(\cdot, r^k)](i)| \leq \delta$ for all $i \in \mathcal{S}$ and $k = 0, 1, \ldots$, where we use $[T_{\pi^{k+1}} \tilde{J}(\cdot, r^k)](i)$ and $[T\tilde{J}(\cdot, r^k)](i)$ to respectively denote the ith components of the vectors $T_{\pi^{k+1}} \tilde{J}(\cdot, r^k)$ and $T\tilde{J}(\cdot, r^k)$. Bertsekas and Tsitsiklis (1996) show that the approximate policy iteration algorithm generates a sequence of policies whose optimality gaps are bounded by ϵ and δ. In particular, the sequence of policies $\{\pi^k\}_k$ generated by the approximate policy iteration algorithm

satisfy $\limsup_{k\to\infty}\|J^{\pi^k} - J^*\|_\infty \leq (\delta + 2\,\alpha\,\epsilon)/(1-\alpha)$. This result provides some justification for the approximate policy iteration algorithm, but it is clearly difficult to measure ϵ and δ. If we use simulation to estimate the expectations, then there is always a small probability of incurring a large simulation error and coming up with a uniform bound on the simulation error may also not be possible.

A practical issue that requires clarification within the context of the approximate policy iteration algorithm is that the computation of $\{C_l : l = 0, 1, \ldots\}$ and the regression problem in (21) involve infinite sums, which cannot be computed in practice. To address this difficulty, for large finite integers N and M with $M < N$, we can generate a sequence of N states $\{i_0, i_1, \ldots, i_N\}$ and compute the discounted cost starting from each one of the states $\{i_0, i_1, \ldots, i_{N-M}\}$ until we reach the final state i_N. This amounts to letting $C_l = \sum_{t=l}^{N-1} \alpha^{t-l}\, g(i_t, \pi^k(i_t), i_{t+1})$ for $l = 0, 1, \ldots, N-M$ and we can use these sampled discounted costs in the regression problem. It is straightforward to check that if we have $l \in \{0, 1, \ldots, N-M\}$, then the two sums $\sum_{t=l}^{N-1} \alpha^{t-l}\, g(i_t, \pi^k(i_t), i_{t+1})$ and $\sum_{t=l}^{\infty} \alpha^{t-l}\, g(i_t, \pi^k(i_t), i_{t+1})$ differ by at most $\alpha^M\, \bar{G}/(1-\alpha)$, where we let $\bar{G} = \max_{i,j\in\mathcal{S}, u\in\mathcal{U}(i)} |g(i,u,j)|$. Therefore, truncating the sequence of states may not create too much error as long as M is large.

Bertsekas and Tsitsiklis (1996) describe an approximate version of the value iteration algorithm that also uses regression. At any iteration k, this algorithm keeps a vector of adjustable parameters r^k that characterize a value function approximation through the parameterized approximation architecture $\tilde{J}(\cdot, r^k)$ as in (6). We apply the operator T on $\tilde{J}(\cdot, r^k)$ to compute $\{[T\tilde{J}(\cdot, r^k)](i) : i \in \tilde{\mathcal{S}}\}$ for only a small set of representative states $\tilde{\mathcal{S}} \subseteq \mathcal{S}$. The adjustable parameters r^{k+1} at the next iteration are given by the optimal solution to the regression problem $\min_r \sum_{i\in\tilde{\mathcal{S}}} \left[\tilde{J}(i,r) - [T\tilde{J}(\cdot, r^k)](i)\right]^2$. Bertsekas and Tsitsiklis (1996) give performance bounds for this approximate value iteration algorithm. They indicate that one way of selecting the set of representative states $\tilde{\mathcal{S}}$ is to simulate the trajectory of a reasonable policy and focus on the states visited in the simulated trajectory. Noting the definition of the operator T, computing $[T\tilde{J}(\cdot, r^k)](i)$ requires taking an expectation and one can try to approximate this expectation by using simulation, especially when we do not have the data to estimate the transition probability matrices. Approximate value and policy iteration algorithms that we describe in this section are due to Bertsekas and Tsitsiklis (1996) and Bertsekas (2001). Tsitsiklis and Van Roy (1996) also give an analysis for the approximate value iteration algorithm.

6 Rollout Policies

The idea behind rollout policies is to build on a given policy and improve the performance of this policy with reasonable amount of work. For a given policy π with discounted cost J^π, we define policy π' such that $\pi'(i)$ is the optimal solution to the problem $\min_{u\in\mathcal{U}(i)}\{\sum_{j\in\mathcal{S}} p_{ij}(u)\,[g(i,u,j) + \alpha\, J^\pi(j)]\}$ for all $i \in \mathcal{S}$. In this case, we can use elementary properties of the policy iteration algorithm to show that policy π' is at least as good as policy π. In other words, the discounted cost $J^{\pi'}$ incurred by policy π' is no larger than the discounted cost J^π incurred by policy π. To see this result, we observe that the definition of policy π' implies that

$$\sum_{j\in\mathcal{S}} p_{ij}(\pi'(i))\,[g(i,\pi'(i),j) + \alpha\, J^\pi(j)] \leq \sum_{j\in\mathcal{S}} p_{ij}(\pi(i))\,[g(i,\pi(i),j) + \alpha\, J^\pi(j)] = J^\pi(i)$$

for all $i \in \mathcal{S}$, where the equality follows by the fact that J^π is the fixed point of the operator T_π. We write the inequality above as $T_{\pi'} J^\pi \leq J^\pi$. It is a standard result in the Markov decision process literature that the operator $T_{\pi'}$ is monotone and applying this operator on both sides of the last inequality, we obtain $T_{\pi'}^2 J^\pi \leq T_{\pi'} J^\pi \leq J^\pi$. Continuing in this fashion, we have $T_{\pi'}^k J^\pi \leq J^\pi$ for all $k = 0, 1, \ldots$. By using the value iteration algorithm under the assumption that the only possible policy is π', we have $\lim_{k\to\infty} T_{\pi'}^k J^\pi = J^{\pi'}$. Therefore, the last inequality implies that $J^{\pi'} \leq J^\pi$.

The biggest hurdle in finding the action chosen by policy π' is the necessity to know J^π. If policy π has a simple structure, then it may be possible to compute J^π exactly, but in general, we need to estimate J^π by using simulation. In particular, given that the system is in state i, to be able to find the action chosen by policy π', we try each action $u \in \mathcal{U}(i)$ one by one. For each action u, we simulate the transition of the system from state i to the next state. From that point on, we follow the actions chosen by policy π. Accumulating the discounted cost incurred along the way yields a sample of $\sum_{j\in\mathcal{S}} p_{ij}(u)\,[g(i,u,j) + \alpha\, J^\pi(j)]$. By simulating multiple trajectories, we can use a sample average to estimate $\sum_{j\in\mathcal{S}} p_{ij}(u)\,[g(i,u,j) + \alpha\, J^\pi(j)]$. Once we estimate this quantity for all $u \in \mathcal{U}(i)$, the action that yields the smallest value for $\sum_{j\in\mathcal{S}} p_{ij}(u)\,[g(i,u,j) + \alpha\, J^\pi(j)]$ is the action chosen by policy π' when the system is in state i. This approach is naturally subject to simulation error, but it often performs well in practice and it can significantly improve the performance of the original policy π. It is also worthwhile to emphasize that the original policy π can be an arbitrary policy, including policies driven by value function approximations or heuristics.

There are numerous applications of rollout policies, where one can significantly improve the performance of the original policy. Tesauro and Galperin (1996) use rollout policies to strengthen several strategies for playing backgammon. Bertsekas, Tsitsiklis and Wu (1997) view certain combinatorial optimization problems as sequential decision processes and improve the performance of several heuristics by using rollout policies. Bertsekas and Castanon (1999) provide applications on stochastic scheduling problems and Secomandi (2001) focuses on vehicle routing problems. Yan, Diaconis, Rusmevichientong and Van Roy (2005) generate strategies for playing solitaire by using rollout policies.

7 State Aggregation

An intuitive idea to deal with the large number of states is to use state aggregation, where we partition the state space into a number of subsets and assume that the value function is constant over each partition. To this end, we use $\{\mathcal{X}_\ell : \ell = 1, \ldots, L\}$ to denote the partition of the states such that $\cup_{\ell=1}^L \mathcal{X}_\ell = \mathcal{S}$ and $\mathcal{X}_\ell \cap \mathcal{X}_{\ell'} = \varnothing$ for all $\ell \neq \ell'$. We assign the value r_ℓ to the value function over the partition $\mathcal{X}_\ell$. The interesting question is whether we can develop an algorithm that finds a good set of values for $\{r_\ell : \ell = 1, \ldots, L\}$. Using $\mathbf{1}(\cdot)$ to denote the indicator function, one approach is to generate a sequence of states $\{i_0, i_1, \ldots\}$ so that if we have $i_k \in \mathcal{X}_\ell$, then we update r_ℓ by using the stochastic approximation iteration

$$r_\ell \leftarrow (1-\gamma)\, r_\ell + \gamma \min_{u\in\mathcal{U}(i_k)} \sum_{j\in\mathcal{S}} p_{ij}(u) \big[g(i_k,u,j) + \alpha \sum_{l=1}^L \mathbf{1}(j \in \mathcal{X}_l)\, r_l\big] \tag{22}$$

immediately after observing state i_k. One can develop convergence results for the updating procedure in (22) under fairly general assumptions on how the sequence of states $\{i_0, i_1, \ldots\}$ are sampled. For example, Bertsekas and Tsitsiklis (1996) sample a subset $\mathcal{X}_\ell$ uniformly over $\{\mathcal{X}_\ell : \ell = 1, \ldots, L\}$. Given that the subset $\mathcal{X}_\ell$ is sampled, they sample $i_k \in \mathcal{X}_\ell$ according to the probabilities $\{q_\ell(i) : i \in \mathcal{X}_\ell\}$. The probabilities $\{q_\ell(i) : i \in \mathcal{X}_\ell\}$ can be arbitrary except for the fact that they add up to one.

We can expect state aggregation to work well as long as the value function is relatively constant within each of the subsets $\{\mathcal{X}_\ell : \ell = 1, \ldots, L\}$. We let $\epsilon_\ell = \max_{i,j \in \mathcal{X}_\ell} |J^*(i) - J^*(j)|$ to capture how much the value function fluctuates within the subset $\mathcal{X}_\ell$. In this case, letting $\epsilon = \max_{\ell=1,\ldots,L} \epsilon_\ell$, Bertsekas and Tsitsiklis (1996) show that the updating procedure in (22) converges to a point $\hat{r} = \{\hat{r}_\ell : \ell = 1, \ldots, L\}$ that satisfies $|J^*(i) - \hat{r}_\ell| \leq \epsilon/(1-\alpha)$ for all $i \in \mathcal{X}_\ell$ and $\ell = 1, \ldots, L$. Therefore, we estimate the value function with an approximation error of $\epsilon/(1-\alpha)$. We note that the limiting point of the updating procedure can depend on the choice of the probabilities $\{q_\ell(i) : i \in \mathcal{X}_\ell\}$, but the choice of these probabilities ultimately does not affect the convergence result and the fact that the limiting point provides an approximation error of $\epsilon/(1-\alpha)$.

We note that a parameterized approximation architecture such as the one in (6) is actually adequate to represent state aggregation. In particular, we can define the feature functions $\{\phi_\ell(\cdot) : \ell = 1, \ldots, L\}$ such that $\phi_\ell(i) = 1$ whenever $i \in \mathcal{X}_\ell$ and $\phi_\ell(i) = 0$ otherwise. In this case, the adjustable parameter r_ℓ in the parameterized approximation architecture $\sum_{\ell=1}^{L} r_\ell \, \phi_\ell(\cdot)$ captures the value assigned to the value function over the partition $\mathcal{X}_\ell$. Bean, Birge and Smith (1987) analyze state aggregation for shortest path problems. Tsitsiklis and Van Roy (1996) and Van Roy (2006) provide convergence results for updating procedures similar to the one in (22). Singh, Jaakkola and Jordan (1995) pursue the idea of soft state aggregation, where each state belongs to a subset $\mathcal{X}_\ell$ with a particular probability.

8 Conclusions

In this chapter, we described a number of approaches for alleviating the computational difficulties associated with dynamic programming. Methods such as the temporal difference learning and Q-learning algorithm avoid transition probability matrices by using simulated trajectories of the system to estimate expectations. There are variants of these methods that use parameterized approximation architectures and the goal of these variants is to address the difficulty associated with storing the value function. The linear programming approach solves a large linear program to fit a parameterized approximation architecture to the value function. Approximate policy iteration uses a combination of regression and simulation in an effort to obtain a sequence of policies that improve on each other. Rollout policies start with an arbitrary policy and improve the performance of this policy with relatively small amount of work. State aggregation builds on the intuitive notion of partitioning the state space into a number of subsets and assuming that the value function is constant over each partition.

Due to the introductory nature of this chapter and the limited space, our coverage of computational dynamic programming approaches has not been exhaustive. One important point is that if we have a good approximation $\tilde{J}(\cdot, r)$ to the value function, then to be able to make the decisions by using this

value function approximation, we need to plug the value function approximation in the right side of the optimality equation in (1) and solve the resulting optimization problem. If the number of admissible actions in each state is small, then we can solve the resulting optimization problem by checking the objective function value provided by each action one by one, but this approach becomes problematic when the number of admissible actions is large. In problems where the action is itself a high dimensional vector, it may be possible to choose the feature functions $\{\phi_\ell(\cdot) : \ell = 1, \ldots, L\}$ such that the optimization problem in question decomposes by each component of the action vector. Another way to deal with the resulting optimization problem is to choose the feature functions such that the value function approximation $\tilde{J}(\cdot, r) = \sum_{\ell=1}^{L} r_\ell \, \phi_\ell(\cdot)$ ends up having a special structure and the resulting optimization problem can be solved by using standard optimization tools. For example, if the state and the action take values in the Euclidean space, then we may use linear or piecewise linear functions of the state as the feature functions and it may be possible to solve the resulting optimization problem by using linear or integer programming tools. Godfrey and Powell (2002*a*), Godfrey and Powell (2002*b*), Topaloglu and Powell (2006), Schenk and Klabjan (2008) and Simao, Day, George, Gifford, Nienow and Powell (2009) use linear and piecewise linear value function approximations in numerous dynamic programs that arise from the freight transportation setting. By using such value function approximations, they are able to deal with states and actions that are themselves vectors with hundreds of dimensions.

If the state and the action take values in the Euclidean space, then another useful approach is to decompose the Markov decision problem in such a way that one can obtain approximations to the value function by concentrating on each component of the state separately. Adelman and Mersereau (2008) use the term "weakly coupled dynamic program" to refer to dynamic programs that would decompose if a few linking constraints did not couple the decisions acting on different components of the vector valued state. They use Lagrangian relaxation to relax these complicating constraints. They propose methods to choose a good set of Lagrange multipliers and show that their Lagrangian relaxation idea provides lower bounds on the value function. Karmarkar (1981), Cheung and Powell (1996), Castanon (1997) and Topaloglu (2009) apply Lagrangian relaxation to dynamic programs arising from the inventory allocation, fleet management, sensor management and revenue management settings.

Our development in this chapter assumes that the feature functions $\{\phi_\ell(\cdot) : \ell = 1, \ldots, L\}$ are fixed and $\{\phi_\ell(i) : \ell = 1, \ldots, L\}$ are able to capture the important features of state i from the perspective of estimating the discounted cost starting from this state. The construction of the feature functions is traditionally left to the model builder and this task may require quite a bit of insight into the problem at hand and a considerable amount of trial and error. There is some recent work on developing automated algorithms to construct the feature functions and this is an area that can be beneficial to all of the methods that we describe in this chapter. Klabjan and Adelman (2007) use a special class of piecewise linear functions as possible candidates for the feature functions. They show that it is possible to approximate any function with arbitrary precision by using enough number of functions from their special class. Veatch (2009) considers control problems in the queueing network setting and describes an approach for iteratively adding feature functions from a predefined set.

References

Adelman, D. (2007), 'Dynamic bid-prices in revenue management', *Operations Research* **55**(4), 647–661.

Adelman, D. and Mersereau, A. J. (2008), 'Relaxations of weakly coupled stochastic dynamic programs', *Operations Research* **56**(3), 712–727.

Barto, A. G., Bradtke, S. J. and Singh, S. P. (1995), 'Learning to act using real-time dynamic programming', *Artificial Intelligence* **72**, 81–138.

Bean, J., Birge, J. and Smith, R. (1987), 'Aggregation in dynamic programming', *Operations Research* **35**, 215–220.

Bellman, R. (1957), *Dynamic Programming*, Princeton University Press, Princeton.

Benveniste, A., Metivier, M. and Priouret, P. (1991), *Adaptive Algorithms and Stochastic Approximations*, Springer.

Bertsekas, D. (2010), Dynamic programming and optimal control, 3rd Edition, Volume II, Chapter 6, Approximate dynamic programming, Technical report, MIT, Cambridge, MA.
Available at `http://web.mit.edu/dimitrib/www/dpchapter.pdf`.

Bertsekas, D. and Castanon, D. (1999), 'Rollout algorithms for stochastic scheduling problems', *J. Heuristics* **5**, 89–108.

Bertsekas, D. P. (2001), *Dynamic Programming and Optimal Control*, Athena Scientific, Belmont, MA.

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

Bertsekas, D. and Shreve, S. (1978), *Stochastic Optimal Control: The Discrete Time Case*, Academic Press, New York.

Bertsekas, D., Tsitsiklis, J. and Wu, C. (1997), 'Rollout algorithms for combinatorial optimization', *Journal of Heuristics* **3**(3), 245–262.

Blackwell, D. (1965), 'Discounted dynamic programming', *Ann. Math. Stat.* **36**, 226–235.

Castanon, D. A. (1997), Approximate dynamic programming for sensor management, *in* '*P*roceedings of the 36th Conference on Decision & Control', San Diego, CA.

Cheung, R. K. and Powell, W. B. (1996), 'An algorithm for multistage dynamic networks with random arc capacities, with an application to dynamic fleet management', *Operations Research* **44**(6), 951–963.

de Farias, D. P. and Van Roy, B. (2003*a*), Approximate linear programming for average-cost dynamic programming, *in* 'Advances in Neural Information Processing Systems', Vol. 15, MIT Press.

de Farias, D. P. and Van Roy, B. (2003*b*), 'The linear programming approach to approximate dynamic programming', *Operations Research* **51**(6), 850–865.

de Farias, D. P. and Van Roy, B. (2004), 'On constraint sampling in the linear programming approach to approximate dynamic programming', *Mathematics of Operations Research* **29**(3), 462–478.

de Farias, D. P. and Van Roy, B. (2006), 'A cost-shaping linear program for average-cost approximate dynamic programming with performance guarantees', *Mathematics of Operations Research* **31**(3), 597–620.

de Farias, D. P. and Weber, T. (2008), Choosing the cost vector of the linear programming approach to approximate dynamic programming, *in* 'Proceedings of the 47th IEEE Conference on Decision and Control', pp. 67 –72.

Desai, V. V., Farias, V. F. and Moallemi, C. C. (2009), Approximate dynamic programming via a smoothed linear program, Technical report, MIT, Sloan School of Management.

Godfrey, G. A. and Powell, W. B. (2002*a*), 'An adaptive, dynamic programming algorithm for stochastic resource allocation problems I: Single period travel times', *Transportation Science* **36**(1), 21–39.

Godfrey, G. A. and Powell, W. B. (2002*b*), 'An adaptive, dynamic programming algorithm for stochastic resource allocation problems II: Multi-period travel times', *Transportation Science* **36**(1), 40–54.

Howard, R. A. (1960), *Dynamic Programming and Markov Processes*, MIT Press, Cambridge, MA.

Karmarkar, U. S. (1981), 'The multiperiod multilocation inventory problems', *Operations Research* **29**, 215–228.

Klabjan, D. and Adelman, D. (2007), 'An infinite-dimensional linear programming algorithm for deterministic semi-Markov decision processes on Borel spaces', *Mathematics of Operations Research* **32**(3), 528–550.

Kunnumkal, S. and Topaloglu, H. (2008), 'Exploiting the structural properties of the underlying Markov decision problem in the Q-learning algorithm', *INFORMS Journal on Computing* **20**(2), 288–301.

Kunnumkal, S. and Topaloglu, H. (2009), 'Stochastic approximation algorithms and max-norm "projections"', *The ACM Transactions on Modeling and Computer Simulation* (to appear).

Kushner, H. J. and Clark, D. S. (1978), *Stochastic Approximation Methods for Constrained and Unconstrained Systems*, Springer-Verlang, Berlin.

Kushner, H. J. and Yin, G. G. (2003), *Stochastic Approximation and Recursive Algorithms and Applications*, Springer, New York.

Powell, W. B. (2007), *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, John Wiley & Sons, Hoboken, NJ.

Puterman, M. L. (1994), *Markov Decision Processes*, John Wiley and Sons, Inc., New York.

Schenk, L. and Klabjan, D. (2008), 'Intra market optimization for express package carriers', *Transportation Science* **42**, 530–545.

Schweitzer, P. and Seidmann, A. (1985), 'Generalized polynomial approximations in Markovian decision processes', *Journal of Mathematical Analysis and Applications* **110**, 568–582.

Secomandi, N. (2001), 'A rollout policy for the vehicle routing problem with stochastic demands', *Operations Research* **49**(5), 796–802.

Sennott, L. I. (1989), 'Average cost optimal stationary policies in infinite state Markov decision processes with unbounded costs', *Operations Research* **37**, 626–633.

Shapley, L. (1953), 'Stochastic games', *Proc. Nat. Acad. Sci. U.S.A* **39**, 1095–1100.

Si, J., Barto, A. G., Powell, W. B. and Wunsch II, D., eds (2004), *Handbook of Learning and Approximate Dynamic Programming*, Wiley-Interscience, Piscataway, NJ.

Simao, H. P., Day, J., George, A. P., Gifford, T., Nienow, J. and Powell, W. B. (2009), 'An approximate dynamic programming algorithm for large-scale fleet management: A case application', *Transportation Science* **43**(2), 178–197.

Singh, S. P., Jaakkola, T. and Jordan, M. I. (1995), Reinforcement learning with soft state aggregation, *in* 'Advances in Neural Information Processing Systems 7', MIT Press, pp. 361–368.

Sutton, R. S. (1984), Temporal Credit Assignment in Reinforcement Learning, PhD thesis, University of Massachusetts, Amherts, MA.

Sutton, R. S. (1988), 'Learning to predict by the methods of temporal differences', *Machine Learning* **3**, 9–44.

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

Tesauro, G. and Galperin, G. (1996), 'On-line policy improvement using Monte-Carlo search', *Advances in Neural Information Processing Systems* **9**, 1068–1074.

Topaloglu, H. (2009), 'Using Lagrangian relaxation to compute capacity-dependent bid-prices in network revenue management', *Operations Research* **57**(3), 637–649.

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

Tsitsiklis, J. N. (1994), 'Asynchronous stochastic approximation and Q-learning', *Machine Learning* **16**, 185–202.

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

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

Tsitsiklis, J. and Van Roy, B. (2001), 'Regression methods for pricing complex American-style options', *IEEE Transactions on Neural Networks* **12**(4), 694–703.

Van Roy, B. (2006), 'Performance loss bounds for approximate value iteration with state aggregation', *Mathematics of Operations Research* **31**(2), 234–244.

Veatch, M. (2009), Adaptive simulation/LP methods for queueing network control, *in* 'INFORMS Conference, San Diego, CA'.

Veatch, M. H. and Walker, N. (2008), Approximate linear programming for network control: Column generation and subproblems, Technical report, Gordon College, Department of Mathematics.

Watkins, C. J. C. H. (1989), Learning from Delayed Rewards, PhD thesis, Cambridge University, Cambridge, England.

Watkins, C. J. C. H. and Dayan, P. (1992), 'Q-learning', *Machine Learning* **8**, 279–292.

Yan, X., Diaconis, P., Rusmevichientong, P. and Van Roy, B. (2005), Solitaire: Man versus machine, *in* 'Advances in Neural Information Processing Systems 17', pp. 1553–1560.