Simulation-Based Function Selection in Approximate Dynamic Programming

Jonathan R. Senning and Michael H. Veatch
Department of Mathematics and Computer Science
Gordon College
mike.veatch@gordon.edu jonathan.senning@gordon.edu

Abstract

Many approximate dynamic programming approaches replace the value function of a dynamic program with a lower-dimensional function, then use some method to fit the coefficients in the approximation. The resulting accuracy is governed by the basis functions, or features, used. We propose a method for selecting basis functions from a large set of candidate functions by minimizing a form of Bellman error on a simulated trajectory. An efficient algorithm is proposed to solve the resulting least squares problem. The method is tested in combination with approximate linear programming on average cost queueing network problems. It performs much better than random selection of functions.

September 18, 2012

I. Introduction

Approximate dynamic programming (ADP) has been used on a variety of large Markov Decision Process (MDPs). An important question in these methods is how to choose the approximation architecture when approximating the value function. Indeed, theoretical guarantees have been established for some algorithms that emphasize the importance of the approximation architecture. We consider approximations that are linear combinations of a set of basis functions, or features. Successful applications of ADP generally use basis functions that are chosen for the specific problem structure, e.g., [1], [2], [3], [4]. However, selecting the basis functions can require extensive trial and error, even if the problem has a well-understood structure.

This paper proposes a new method to select basis functions that combines problem-specific knowledge and simulation. The method is described for average cost MDPs. Any ADP method, including approximate value iteration, approximate policy iteration, and temporal differences, could be used once the basis functions are selected. For numerical tests, we use ALP and queueing network control problems. The algorithm takes as input a large set of candidate basis functions. It also requires an initial policy. The system is simulated using this policy. Sets of basis functions are evaluated by computing a best fit to Bellman's equation for that policy over the simulated states. This least squares form of Bellman error is used as a surrogate for accuracy. Each candidate basis function is evaluated by computing the reduction obtained by adding the function to the current set of functions. The function with the largest reduction is then added to the approximation architecture. After one or more functions are added in this fashion, the expanded ALP is solved to obtain a new policy. The new policy is simulated and the process is repeated.

Each iteration requires solving a least squares problem for each candidate function and solving the ALP. We show that by using QR decomposition the least squares problem for each additional function can be solved in a number of steps that is linear in the number of simulated states. Solving the ALP becomes more difficult as functions are added to the approximation; however, using a warm start can accelerate these solutions.

The basis function selection method, when combined with ALP using constraint sampling, is practical for problems with large state spaces. However, the ALP is not practical for problems with a large number of actions, since they are enumerated in the ALP constraints. The algorithm is presented assuming that there are a small number of possible transitions out of each state, so that expectations in the dynamic programming equations can be computed. Additional simulation could be done to estimate these expectations. Thus, of the three curses of dimensionality described in [5], our algorithm addresses the first and could be extended to address the third.

The main contribution of this paper is an algorithm to efficiently select basis functions from a large predefined set for use in ADP. We give heuristic justifications for the algorithm. If there is an exact candidate function, in the sense that adding it to the approximation allows the differential cost for the current policy to be fit exactly on the sampled states, then it will be selected by the algorithm. In numerical tests on queueing networks problems, the algorithm selects functions that improve accuracy much more than randomly selected functions. We measure accuracy by

the error in average cost found by the ALP (compared to optimal). The algorithm closes this gap ?? to ??% more than randomly selected functions. The approximation architectures found by the algorithm are also better than the best approximations found by extensive testing in [6]. The test problems have at most four buffers and can be truncated to a few million states. Up to 800?? candidate functions are used. The simulation could be used on significantly larger problems. However, on larger queueing networks the ALP solution time limits the number of functions that can be added to the approximation.

We do not report performance of the policies found. Unfortunately, the ALP can lead to policies with arbitrarily bad performance. Using a discounted cost formulation or the modified ALPs in [7], [4], and [8] allows for performance guarantees and could overcome this limitation.

The extensive ADP literature is surveyed in [9, Chapter 6] and [5]. The design of approximation architectures is discussed in [9, Section 6.1]. Several approaches to automate the generation of basis functions have been proposed. Non-parametric, kernel methods are used in [10] for least squares policy evaluation and in [11] for ALP. Neighborhood component analysis is used to map into a low-dimensional space, based on the Bellman error, in [12]. The ALP approach was originally proposed in [13]. For discounted MDPs on a finite state space, [14] shows that the quality of the cost-to-go function approximation obtained by the ALP is proportional, in a certain sense, to the quality of the best possible approximation using these basis functions. The quality of the approximation does not degrade with problem size. A similar bound is obtained on performance of the policy implied by the ALP. They use constraint sampling, which is shown to be probabilistically accurate in [15]. These results are extended in [4] to a Lagrangian form of the ALP that is potentially more accurate. Some accuracy guarantees for average cost problems are given in [7] and [8].

The next section formulates the MDP and ALP. Section 3 introduces a form of sampled Bellman error and an associated least squares problem. Section 4 solves the least squares problem using QR decomposition. The algorithm is summarized in Section 5 and numerical results are presented in Section 6, with some details deferred to the Appendix. Section 7 concludes.

II. Average cost MDPs and approximate linear programming

Consider a discrete time MDP x_t with a finite state space $\mathcal{S}$ and a finite set of actions $\mathcal{A}(x)$ available in state x. Under a stationary Markov policy $u(x)$, the state process is a Markov chain

with transition probability matrix $P_u = \big[p_{u(x)}(x,y)\big]$. A nonnegative cost $g(x,a)$ is incurred when action a is taken in state x. For shorthand, use the notation

$$(P_u h)(x) = \sum_{y \in \mathcal{S}} p_{u(x)}(x,y) h(y).$$

The average cost under a stationary policy u is

$$\lambda_u = \limsup_{T \to \infty} \frac{1}{T} E_{x,u} \sum_{t=0}^{T-1} g_u(x_t).$$

Here $E_{x,u}$ denotes expectation given the initial state $x_0 = x$ and policy u. Assume that for each u this value is independent of x. This assumption essentially imposes a unichain structure on the resulting Markov chains; see Bertsekas [9, Section 4.2]. Bellman's equation for this problem is

$$\min_{u(x) \in \mathcal{A}(x)} g(x,u(x)) + (P_u h)(x) - h(x) = \lambda. \tag{1}$$

Note that if (λ, h) solves (1), then so does $(\lambda, h + k\mathbf{1})$ for any additive constant k. One choice of this constant is made by the bias function, defined for policy u as

$$h_u(x) = \liminf_{T \to \infty} E_{x,u} \sum_{t=0}^{T} [g(x_t) - \lambda_u].$$

Another choice is the differential cost $h_u(x) - h_u(0)$, which is zero in some reference state $x = 0$. There exists an optimal stationary policy with average cost λ^* and bias h^* that satisfy (1); see, e.g., Puterman [16, Theorem 8.4.3].

Bellman's equation is equivalent to the following linear program (see [9, Section 4.3.3]):

$$\begin{aligned} &\max_{\lambda, h} \ \lambda \\ &\text{s.t.} \ \ g(x,a) + \sum_{y \in \mathcal{S}} p_a(x,y) h(y) - h(x) \geq \lambda \ \ \text{for all } a \in \mathcal{A}(x) \text{ for all } x \in S. \end{aligned} \tag{2}$$

To create a more tractable LP, the differential cost is approximated by

$$(\Phi r)(x) = \sum_{k=1}^{K} r_k \phi_k(x) \tag{3}$$

using some small set of basis functions ϕ_k and variables r_k. Substituting into (2) gives the approximate linear program

$$\text{(ALP)} \quad \max_{\lambda, r} \ \lambda$$
$$\text{s.t.} \ \ g(x,a) + \sum_{y \in \mathcal{S}} p_a(x,y)(\Phi r)(y) - (\Phi r)(x) \geq \lambda \mathbf{1} \ \ \text{for all } a \in \mathcal{A}(x) \text{ for all } x \in S.$$

(ALP) is feasible and, since it is equivalent to (2) with the constraint $J = \Phi r$ added, its objective function is bounded above. Let (λ_{ALP}, r_{ALP}) be an optimal solution. For any (λ, h) feasible for (ALP), λ is a lower bound. In particular, $\lambda_{ALP} \leq \lambda^*$; see Puterman [16, Theorem 8.4.1].

An h-greedy policy is

$$u_h(x) \in \arg\min_{a \in \mathcal{A}(x)} \left\{ g(x,a) + \alpha \sum_y p_a(x,y) h(y) \right\}. \tag{4}$$

In particular, the policy associated with (ALP) is a Φr_{ALP}-greedy policy u_{ALP}.

III. Sampled Bellman error

We would like to assess the quality of an approximation architecture without solving (ALP), so that we can select among many possible architectures. Given any (λ, h), Bellman error is defined as

$$B^{\lambda,h}(x) = \min_{u(x) \in \mathcal{A}(x)} g(x, u(x)) - \lambda + \left(P_u h\right)(x) - h(x). \tag{5}$$

If (λ, h) satisfy the constraints (ALP) then

1) $B^{\lambda,h}(x) \geq 0$ is the minimum slack of constraints for that x.
2) $E_{u_h}(B^{\lambda,h}) = \lambda_{u_h} - \lambda$, i.e., expected Bellman error with respect to the h-greedy policy is the suboptimality of that policy.

Thus, Bellman error should be related in some fashion to the accuracy of the ALP. To allow for two-sided error, we will use squared Bellman error over a sample path $x_0, \ldots, x_{T-1}$ and use $h = \Phi r$. Define

$$\widetilde{f}(\overline{r}) = \sum_{t=0}^{T-1} B^{\lambda, \Phi r}(x_t)^2 \tag{6}$$

where $\overline{r} = (\lambda, r)$. Then $\min_{\overline{r}} \widetilde{f}(\overline{r})$ can be used to measure the quality of the approximation architecture Φ. Let r^* be a value of r that minimizes (6).

A more tractable measure is obtained by eliminating the minimum over u. This is equivalent to assuming that u_{ALP} is also Φr^*-greedy. The result is the least squares problem $\min_{\overline{r}} f(\overline{r})$ where

$$f(\overline{r}) = \sum_{t=0}^{T-1} (g(x_t, u_{ALP}(x_t)) - \lambda + \left(P_{u_{ALP}} \Phi r\right)(x_t) - \Phi r\,(x_t))^2. \tag{7}$$

Consider adding one basis function, ϕ_0, so that $h = \Phi r + r_0 \phi_0$. The least squares problem for this architecture is

$$\min_{\overline{r}, r_0} f(\overline{r}, r_0). \tag{8}$$

We will solve (8) for each candidate function ϕ_0 in order to measure its contribution to the approximation architecture.

As a heuristic justification for (8), first note its large sample behavior. Let $B_u^{\lambda,h}$ denote Bellman error as in (5) but for the fixed policy u. Because the Markov chain for any policy is unichain, it is ergodic and

$$\lim_{T\to\infty} \frac{1}{T} f(\overline{r}) = E_u\left(B_u^{\lambda,\Phi r}(x)^2\right)$$

with probability one. By restricting $\overline{r}$ to a compact set, this convergence can be made uniform over $\overline{r}$ so that

$$\lim_{T\to\infty} \frac{1}{T} \min_{\overline{r}} f(\overline{r}) = \min_{\overline{r}} E_u\left(B_u^{\lambda,\Phi r}(x)^2\right)$$

with probability one. Thus, for a sufficiently long simulation, (8) will select the same function as if the stationary distribution was used.

Second, let h_{ALP} be the differential cost for policy u_{ALP} and suppose that there is a function ϕ_0 such that $h_{ALP}(x_t) = (\Phi r + r_0\phi_0)(x_t)$ for $t = 0, ..., T-1$ for some (r, r_0). Then the minimum in (8) is zero and this function, or some other function with the same property, will be selected by the least squares procedure. Furthermore, consider (ALP) with just the constraints for states $x_0, \ldots, x_{T-1}$ and use the augmented approximation $\Phi r + r_0\phi_0$. In this version of (ALP), λ_u and the values of (r, r_0) that minimize (8) are feasible. Thus, if there is a basis function that provides a perfect approximation on the sampled states for the current policy, such a function will be chosen. In the idealized setting where the policy has converged to optimal and the set of sampled states is sufficiently rich, λ^* will be optimal for (ALP), i.e., the exact optimal average cost will be found. Although we are interested in problems where no perfect approximation is likely, this behavior suggests that if there are functions that improve λ_{ALP}, they will be selected, particularly after a good policy has been found.

IV. QR DECOMPOSITION FOR THE LEAST SQUARES PROBLEM

This section shows that the least squares problem (8) can be solved efficiently for a large number of candidate basis functions using QR decomposition. Corresponding to (8) is an overdetermined linear system. Even though the systems are overdetermined and have least-squares solutions, many of them will have a rank-deficient matrix and so do not have a unique solution. The QR decomposition provides an efficient method to solve these systems, producing the least-squares solution with minimum norm [17], [18]. The key idea is that because these systems differ by a single matrix column, corresponding to the candidate basis function, the majority of the work in the decomposition need be done only once. See [19] and chapter 12 in [20].

For the least squares problem (7), the linear system is $A\overline{r} = b$, where $b_t = -g(x_t, u_{ALP}(x_t))$ for $t = 0, \ldots, T-1$ and A is a $T \times (K+1)$ matrix. The rows of A are numbered from 0 to $T-1$ so that they are indexed by t. The column for the variable λ contains -1 and the column for the variable r_k contains the entry $[(P_{u_{ALP}} - I)\phi_k](x_t)$ in row t. Recall that T is the length of the sample path and K is the number of basis functions in the current approximation. Permute the columns of A so that $s = \text{rank}(A)$ linearly independent columns come first (for notational simplicity, assume this is true of A and number these columns 1 to s). The decomposition yields $A = QR$ where Q is an orthogonal matrix and

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix},$$

with R_{11} $s \times s$, nonsingular, and upper triangular. Then $R = Q^T A$.

Now add the function ϕ_0, by inserting the column a, corresponding to ϕ_0, after the linearly independent columns of A. Let $A_{i:j}$ be the matrix formed by columns i through j of A. The augmented matrix is

$$\widehat{A} = [A_{1:s}\ a\ A_{s+1:K+1}],$$

the variables are $\widehat{r} = (r_{1:s},\ r_0,\ r_{s+1:K+1})^T$, and the linear system corresponding to (8) is

$$\widehat{A}\widehat{r} = b. \tag{9}$$

Observe that

$$Q^T\widehat{A} = \left[Q^T A_{1:s}\ Q^T a\ Q^T A_{s+1:K+1}\right] = \left[\begin{bmatrix} R_{11} \\ 0 \end{bmatrix}\ Q^T a_j\ \begin{bmatrix} R_{12} \\ 0 \end{bmatrix}\right].$$

The entries below the first $s+1$ rows in the column $Q^T a$ can be eliminated with a Givens rotation represented by the orthogonal matrix G:

$$GQ^T\widehat{A} = \left[\begin{bmatrix} R_{11} \\ 0 \end{bmatrix} \; GQ^T a \; \begin{bmatrix} R_{12} \\ 0 \end{bmatrix}\right]. \tag{10}$$

This matrix is upper triangular and GQ^T is orthogonal as both Q and G are orthogonal. Thus, $\widehat{A} = \widehat{Q}\widehat{R}$ where $\widehat{Q} = QG^T$ and $\widehat{R}$ is the matrix in (10).

Finally, as in [17], an orthogonal matrix Z and an upper triangular $(s+1)\times(s+1)$ matrix T_{11} can be found such that

$$\widehat{R} = \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} Z.$$

Once this is done (9) can written as

$$QG^T \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} Z\hat{r} = b.$$

The minimum norm solution is

$$\widehat{r} = Z^T \begin{bmatrix} T_{11}^{-1} & 0 \\ 0 & 0 \end{bmatrix} GQ^T b.$$

In our application $T \gg K$. The QR factorization operation count in this case is $O(K^2T)$. Only the Givens rotation, T_{11}^{-1}, and $\widehat{r}$ must be computed for each candidate function ϕ_0, requiring only $O(K^3)$ operations. Since K, the number of basis functions in the current approximation, is incremented at each iteration, the algorithm is faster in the early iterations and slows down as it progresses.

V. Basis function selection algorithm

In this section we describe an algorithm using the Bellman error criterion to iteratively expand the approximation architecture. At each step there is a set F of basis functions in the approximation architecture and a set C of additional candidate basis functions.

Initialization. Choose the candidate functions and divide them into the initial approximation F and the other candidates C. Choose a distribution π on S, the number of simulations N, the simulation length T, an initial policy u, a number of functions to add per iteration m, and a number of iterations.

Iteration.

1) Generate N sample paths of length T: sample x_0 from the distribution π, simulate x_1, $x_2, \ldots, x_{T-1}$ using policy u.
2) For each function $j \in C$, solve the least squares problem $\min f(\overline{r}, r_j)$.
3) Add the function j with smallest minimum in step 2 to the approximation: $F = F \cup \{j\}$ and $C = C \backslash \{j\}$.
4) Repeat steps 2 and 3 until m functions have been added.
5) Update the policy u using the approximation architecture F.

The least squares coefficients (r^*, r_j^*) from step 2 could be used to obtain a greedy policy in step 5. However, using a more involved ADP method seems likely to produce a better policy. We solve (ALP) to find the policy, once with the initial F and in step 5 at each iteration. Solving (ALP) also provides the average cost lower bound λ_{ALP} which could be used in a stopping criterion.

VI. Numerical results

The algorithm was tested on several queueing network control examples from the literature. In all of these examples, the state is $x = (x_1, ..., x_n) \in Z_+^n$, where x_i is the number of jobs in buffer, or class, i. Each server can serve a class among some set of classes or idle. Preemption is allowed. Interarrival and service times are independent and exponentially distributed with rates α_i and μ_i at class i. The cost rate is $c^T x$ for some vector $c > 0$. The problems are converted to discrete time by uniformization.

First, the algorithm was tested on a single queue. There are two actions, idle or serving, and it is optimal to serve whenever a job is present. For this problem, h^* is quadratic. Using a linear function as the initial approximation, the algorithm correctly chose the quadratic term out of 12?? functions in the first iteration, resulting in a perfect fit to h^*. Parameters for the other examples are listed in Table I, where the maximum of the server utilizations is also shown. The network topologies for the first two examples are shown in Figures 1 and 2. For the series line, jobs arrive at class 1 and move to class 2, (and 3 and 4) before exiting. Several types of candidate basis functions were used, as described in the Appendix. The initial approximation F consisted of general quadratic functions except as noted.

	α	μ	c	$\rho_{\max}$
3-class reentrant line	0.1429	0.6, 0.16, 0.25	1, 1, 1	0.89
Rybko-Stolyar network	0.0672, 0.0672	0.12, 0.12, 0.28, 0.28	1, 1, 1, 1	0.8
2-class series line	1	1.5, 1.25	1, 2	0.8
4-class series line	0.8	1.3, 1.2, 1.1, 1.0	1, 4/3, 5/3, 2	0.8

TABLE I

PARAMETERS FOR THE EXAMPLES

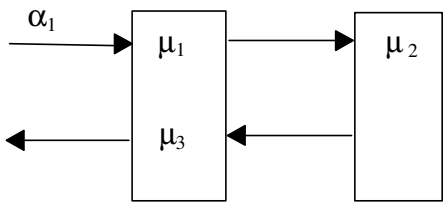

Fig. 1. Three-class reentrant line.

The optimal average cost was found using value iteration on a truncated state space. The ALP was solved using the truncation $x \leq (400,\ 400)$ for the two-class series, $x \leq (20,\ 40,\ 20)$ for the three-class reentrant line, constraint sampling with $100,000$ constraints for the four-class series line, and sampling with $500,000$ constriants for the Rybko-Stolyar network. All examples used $N = 500$ sample paths, each of length $T = 100$, for a total of $50,000$ simulated states. The initial state distribution was an independent geometric distribution for each class with

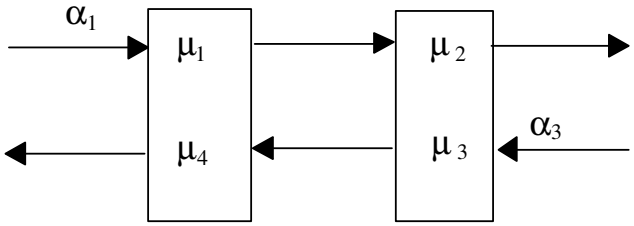

Fig. 2. The Rybko-Stolyar network.

parameter 0.8. This parameter and N determine the amount of exploration of the state space, beyond states visited by the greedy policy. In our tests some exploration helped, particularly in the early iterations of the algorithm, but most values gave similar results. Only $m = 1$ function was added at each iteration of the algorithm. Although this increases the amount of time spent solving ALPs, it provides the best assessment of how well the algorithm can select functions. The ALP was solved using CPLEX version 10 on a 64-bit, 3.4 GHz processor with 6 GB of RAM. ??update or omit??

The algorithm for selecting candidate functions was compared with randomly selecting functons. Eight runs of each were made and the results averaged. Figures **??** and **??** show the variability between runs for the two-class series line, starting with only linear functions. The vertical axis, λ/λ^*, shows the accuracy of the ALP lower bound on optimal average cost. Random function selection has more variability, but the average of eight runs appears to be reasonably accurate for both graphs. Figure **??** shows these averages and the result of solving multiple ALPs at each iteration, one with each candidate function added, and choosing the function that gives the largest average cost. This result is what the algorithm would achieve if the Bellman error was a perfect surrogate. ??need full ALP?? The algorithm graph rises quickly, showing that it chooses important functions in the early iterations, but does not rise quite as fast as the graph when the best functions are chosen.

Figures **??**, **??**, and **??** show similar results for the three-class, four-classs series, and Rybko-Stolyar networks. Although these examples are too large to find the best function to add, the graphs show that the algorithm doe smuch better than random functions selection. In Figure **??** and **??**, the graphs eventually meet because all functions were added; in Figure **??** the run was stoppped at 114 of 194 functions. The erratic behavior in **??** is due to numerical issues in solving the larger ALPs; we did not investigate further because the smaller approximations are of more interest. The algorithm does not do quite as well on the Rybko-Stolyar network, which is more difficult to approximate.

Results also depend on the candidate approximating functions. The functions used are described below; see the Appendix for definitions.

The functions were identified in [21] by comparing the accuracy of several sets of functions; our algorithm identifies smaller sets that have nearly the same accuracy. We also tested the algorithm on larger sets of functions, created by combining the types used above and varying

Head	Head	Head
entry	entry	entry
entry	entry	entry
entry	entry	entry

TABLE II
RUN TIMES

their parameters. Figures **??** and **??** show that the algorithm again does much better than random. However, these graphs do not climb as steeply as those with fewer candidate functions. When more functions are present, most of which are not very useful, the algorithm does not find as good of functions.

Run times using the algorithm ranged from ?? for the three-class reentrant line with 92 iterations (92 candidate functions) to ?? for the Rybko-Stolyar network with ?? iterations (?? candidate functions) ??RScombined2?? Even though 10,000 states were simulated at each iteration, most of the time in large runs is spent solving the ALP (?? and ??% for these two runs, respectively).

.

VII. CONCLUSION

TBD

APPENDIX: BASIS FUNCTIONS

The following basis functions are used for the queueing network examples. Recall that n is the number of classes, or buffers.

(i) *Quadratic functions.* A general quadratic requires $n(n+1)/2+n$ functions.

(ii) *Exponential functions.* There are n^2+n funcitons of the form

$$\begin{aligned}\phi_i(x) &= \beta_i^{x_i}\\ \phi_{ij}(x) &= x_j\beta^{x_i}\end{aligned}$$

where $\beta_i < 1$.

(iii) *Rational functions*. Higher-order terms, e.g., $x_1x_2x_3$ or $x_1^2x_2^2$, would be a natural choice. However, the functions need to be bounded by a quadratic, so we divide by a suitable

polynomial of degree two less than the numerator. Also, instead of using different powers, i.e., 1, x_i, x_i^2, we use $f_{i,1}(x_i) = x_i^2$, $f_{i,2}(x_i) = x_i(N_i - x_i)^+$, $f_{i,3}(x_i) = [(N_i - x_i)^+]^2$, where $x^+ = \max\{x, 0\}$. Note that $f_{i,2}$ and $f_{i,3}$ are zero beyond N_i. The three choices for $f_{i,j}$ lead to 3^n rational functions

$$\phi(x) = \frac{\prod_{i=1}^n f_{i,j(i)}(x_i)}{(1 + \sum_{i=1}^n x_i/N_i)^{2(n-1)}}, \quad j(i) = 1, 2, 3.$$

(iv) *Piece-wise quadratics.* Instead of a uniform grid, define rectangular regions that collect states into sets that grow exponentially as queue length grows. Let $S(y) = \{x : 2^{y_i - 1} - \frac{1}{2} \le x_i < 2^{y_i},\ i = 1, \ldots, n\}$ for all $y \in Z_+^n$ such that $z \neq 0$ and $0 \le y_i < M$ for some M. These $n^M - 1$ sets cover the states (excluding the origin) with $x_i < 2^{M-1}$, dividing each x_i into the sets $\{0\}, \{1\}, \{2, 3\}, \{4, 5, 6, 7\}$, etc. A general quadratic (or linear) function is used in each $S(y)$. The number of basis functions is roughly $n^M n^2/2$. However, if $y_i = 0$ or 1, then the linear and quadratic terms involving x_i are not needed because there is only one value of x_i in $S(y)$ and if $y_i = 2$, then the linear term involving x_i is not needed. Also, we remove the upper bound when $y_i = M - 1$, so that the sets cover all of Z_+^n.

(v) *Functions of one or two variables*. Consider a differential cost approximation of the form

$$h(x) = \sum_{i=1}^n h_i(x_i) \tag{11}$$

using functions of one variable. To parameterize h_i, use the indicator functions $\phi_j(x_i) = 1_{\{x_i = j\}}$ and let

$$h_i(x_i) = \sum_{j=0}^{M-1} r_{ij}\phi_j(x_i) + r_{i0}\phi_0(x_i). \tag{12}$$

Note that $h_i(x)$ is arbitrary for $x < M$ and has the form ϕ_0 for $x \ge M$. This parameterization is used in [3] with $\phi_0(x_i) = x_i \ln x_i$, which we include in our tests. The number of functions in (11) is linear in the number of classes, so that it is suitable for large problems. We use an ad hoc modification of (11) to improve accuracy: replace the indicator functions ϕ_j with

$$\phi_j(x) = 1_{\{x_i = j\}} \left(1 + \gamma |x_{(i)}|^2\right) B^{|x_{(i)}|}$$

where $|x_{(i)}| = \sum_{k \neq i} x_k$ and $\gamma \geq 0$, $B \leq 1$ are parameters to be chosen. Setting $\gamma = 0$ and $B = 1$ recovers the indicator functions. Finally, we consider functions of the two variables x_i and $\sum_{j=i+1}^{n} x_j$, known as the echelon inventory, for $i = 1, ..., n-1$.

ACKNOWLEDGEMENTS

Our students Stephen Rizzo, Lauren Berger, Chris Pfohl, and Annie Hsieh contributed ideas and software to this project.

REFERENCES

[1] D. Adelman, "A price-directed approach to stochastic inventory/routing," *Operations Research*, vol. 52, no. 4, pp. 499–514, 2004.

[2] H. Topaloglu and S. Kunnumkal, "Approximate dynamic programming methods for an inventory allocation problem under uncertainty," *Naval Research Logistics*, vol. 53, pp. 822–841, 2006.

[3] C. Moallemi, S. Kumar, and B. V. Roy, "Approximate and data-driven dynamic programming for queueing networks," 2006, working paper, Graduate School of Business, Columbia University.

[4] V. Desai, V. Farias, and C. Moallemi, "Approximate dynamic programming via a smoothed linear program," 2009, working paper, Graduate School of Business, Columbia University.

[5] W. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley, 2007.

[6] M. Veatch, "Approximate linear programming for networks: Average cost bounds," 2010, working paper, Gordon College, Dept. of Math. Available at http://faculty.gordon.edu/ns/mc/mike-veatch.

[7] D. de Farias and B. V. Roy, "A cost-shaping linear program for average-cost approximate dynamic programming with performance guarantees," *Math. Oper. Res.*, vol. 31, no. 3, pp. 597–620, 2006.

[8] M. Veatch, "Approximate linear programming for average cost MDPs," 2011, working paper, Gordon College, Dept. of Math. Available at http://faculty.gordon.edu/ns/mc/mike-veatch.

[9] D. Bertsekas, *Dynamic Programming and Optimal Control*, 3rd ed. Belmont, MA: Athena Scientific, 2007, vol. 2.

[10] T. Jung and D. Polani, "Kernelizing lspe(?)," in *Proc. of the 2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE, 2007, pp. 338–345.

[11] N. Bhat, V. Farias, and C. Moallemi, "Non-parametric approximate dynamic programming via the kernel method," 2012, working paper, Columbia University.

[12] P. W. Keller, S. Mannor, and D. Precup, "Automatic basis function construction for approximate dynamic programming and reinforcement learning," in *Proceedings of the 23rd international conference on Machine learning*, ser. ICML '06. ACM, 2006, pp. 449–456.

[13] P. Schweitzer and A. Seidmann, "Generalized polynomial approximations in Markovian decision processes," *J. of Mathematical Analysis and Applications*, vol. 110, pp. 568–582, 1985.

[14] D. de Farias and B. V. Roy, "The linear programming approach to approximate dynamic programming," *Oper. Res.*, vol. 51, no. 6, pp. 850–865, 2003.

[15] ——, "On constraint sampling for the linear programming approach to approximate dynamic programming," *Math. Oper. Res.*, vol. 29, no. 3, pp. 462–478, 2004.

[16] M. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York: John Wiley and Sons, Inc., 1994.

[17] L. Foster and R. Kommu, "Algorithm 853: An efficient algorithm for solving rank-deficient least squares problems," *ACM Trans. Math. Softw.*, vol. 32, pp. 157–165, 2006.

[18] G. Quintana-Ortí, X. Sun, and C. H. Bischof, "A BLAS-3 version of the QR factorization with column pivoting," *SIAM J. Sci. Comput.*, vol. 19, pp. 1486–1494, 1998.

[19] S. Hammarling and C. Lucas, "Updating the QR factorization and the least squares problem," pp. 1–72, 2008.

[20] G. Golub and C. Loan, *Matrix computations*, ser. Johns Hopkins studies in the mathematical sciences. Johns Hopkins University Press, 1996.

[21] M. Veatch, "Approximate dynamic programming for networks: Fluid models and constraint reduction," May 2009, working paper, Gordon College, Dept. of Math. Available at faculty.gordon.edu/ns/mc/Mike-Veatch.