Greedy linear value-approximation for factored Markov decision processes

Relu Patrascu
University of Waterloo
rpatrasc@cs.uwaterloo.ca

Pascal Poupart
University of Toronto
ppoupart@cs.toronto.edu

Dale Schuurmans
University of Waterloo
dale@cs.uwaterloo.ca

Craig Boutilier
University of Toronto
cebly@cs.toronto.edu

Carlos Guestrin
Stanford University
guestrin@stanford.edu

Abstract

Significant recent work has focused on using linear representations to approximate value functions for factored Markov decision processes (MDPs). Current research has adopted linear programming as an effective means to calculate approximations for a *given* set of basis functions, tackling very large MDPs as a result. However, a number of issues remain unresolved: *How accurate are the approximations produced by linear programs? How hard is it to produce better approximations?* and *Where do the basis functions come from?* To address these questions, we first investigate the complexity of minimizing the Bellman error of a linear value function approximation—showing that this is an inherently hard problem. Nevertheless, we provide a branch and bound method for calculating Bellman error and performing approximate policy iteration for general factored MDPs. These methods are more accurate than linear programming, but more expensive. We then consider linear programming itself and investigate methods for automatically constructing sets of basis functions that allow this approach to produce good approximations. The techniques we develop are guaranteed to reduce L_1 error, but can also empirically reduce Bellman error.

1 Introduction

Markov decision processes (MDPs) pose a problem at the heart of research on optimal control and reinforcement learning in stochastic environments. This is a well studied area and classical solution methods have been known for several decades. However, the standard algorithms—value-iteration, policy-iteration and linear programming—all produce optimal control policies for MDPs that are expressed in explicit form; that is, the policy, value function and state transition model are all represented in a tabular manner that enumerates the entire state space. This renders classical methods impractical for all but toy problems. The real goal is to achieve solution methods that scale up reasonably in the size of the *state description*, not the size of the state space itself (which is usually either exponential or infinite). Justifiably, most recent work in the area has concentrated on scaling-up solution techniques to handle realistic problems.

There are two basic premises to scaling-up: (1) exploiting structure in the MDP model (i.e. structure in the reward function and the state transition model); and (2) exploiting structure in an approximate representation of the optimal value function (or policy). Most credible attempts at scaling-up have exploited both types of structure. Even then, it is surprisingly hard to formulate an optimization method that can handle large state descriptions while reliably producing value functions or policies with small approximation errors.

Initial research on expressing MDPs compactly has considered representations such as decision trees or decision diagrams to represent the state transition model and reward function in a concise manner (Boutilier, Dearden, & Goldszmidt 2000; Boutilier, Dean, & Hanks 1999). Another recent approach has investigated using factored-table state transition and reward models, where the state variables exhibit limited dependence and have localized influence (Koller & Parr 1999). Both representational paradigms allow complex MDPs to be represented in a compact form. Unfortunately, neither ensures that the optimal value function (or policy) has a compact representation (Koller & Parr 1999). Thus, it appears that one still has to consider compact approximations to the value function (or policy) to make progress.

Numerous schemes have been investigated for representing approximate value functions and policies in a compact form, including: hierarchical decompositions (Dietterich 2000), decision trees and diagrams (Boutilier, Dearden, & Goldszmidt 2000) generalized linear functions (Koller & Parr 1999; 2000; Guestrin, Koller, & Parr 2001a), splines (Trick & Zin 1997), neural networks (Bertsekas & Tsitsiklis 1996), and products of experts (Sallans & Hinton 2000).

This paper focuses on using factored-table MDP representations in conjunction with *linear* value function approximators. Linear approximators have the advantage of providing universal approximation ability (given a sufficient basis) and allowing the widest array of computational techniques to be brought to bear. Moreover, linear approximations interact very well with factored MDP representations (Koller & Parr 1999; 2000). This combination allows approximation methods to be devised which can be feasibly applied to very large MDPs. Two recent examples of this are the approximate policy iteration technique of (Guestrin, Koller, & Parr 2001a) and the approximate linear programming approaches of (Guestrin, Koller, & Parr 2001b; Schuurmans & Patrascu 2001). Many researchers have begun to adopt linear programming as a particularly convenient method for generating value approximations for large MDPs (de Farias & Van Roy 2001). A few issues remain unclear, however. First, the extent to which linear programming sacrifices approximation accuracy for computational expedience is not well understood. Second, there is a need for a systematic method for improving approximation accuracy in cases where it is insufficient. Finally, previous research has not significantly addressed the question of where the basis functions come from in the first place. We attempt to address these three questions here.

2 Background

We will consider MDPs with finite state and action spaces and assume the goal of maximizing infinite horizon discounted reward. We assume states are represented by vectors $\mathbf{s} = s_1, ..., s_n$ of length n, where for simplicity we assume the state variables $s_1, ..., s_n$ are in $\{0, 1\}$; hence the total number of states is $N = 2^n$. We also assume that there is a small finite set of actions $A = \{a_1, ..., a_\ell\}$. An MDP is then defined by: (1) a state transition model $P(\mathbf{s}'|\mathbf{s}, a)$ which specifies the probability of the next state $\mathbf{s}'$ given the current state $\mathbf{s}$ and action a; (2) a reward function $R(\mathbf{s}, a)$ which specifies the immediate reward obtained by taking action a in state $\mathbf{s}$; and (3) a discount factor γ, $0 \le \gamma < 1$. The problem is to determine an optimal control policy $\pi^* : \mathbf{S} \rightarrow A$ that achieves maximum expected future discounted reward in every state.

To understand the classical solution methods it is useful to define some auxiliary concepts. For any policy π, the value function $V^\pi : \mathbf{S} \rightarrow I\!R$ denotes the expected future discounted reward achieved by policy π in each state $\mathbf{s}$. This V^π satisfies a fixed point relationship $V^\pi = B^\pi V^\pi$, where B^π operates on arbitrary functions v over the state space

$$(B^\pi v)(\,\mathbf{s}) = R(\mathbf{s}, \pi(\mathbf{s})) + \gamma \textstyle\sum_{\mathbf{s}'} P(\mathbf{s}'|\mathbf{s}, \pi(\mathbf{s}))v(\mathbf{s}')$$

Another important operator, B^a, is defined for each action

$$(B^a v)(\,\mathbf{s}) = R(\mathbf{s}, a) + \gamma \textstyle\sum_{\mathbf{s}'} P(\mathbf{s}'|\mathbf{s}, a)v(\mathbf{s}')$$

The action-value function $Q^\pi : \mathbf{S} \times A \rightarrow I\!R$ denotes the expected future discounted reward achieved by taking action a in state $\mathbf{s}$ and following policy π thereafter; which must satisfy $Q^\pi(\mathbf{s}, a) = (B^a V^\pi)(\mathbf{s})$. Given an arbitrary function v over states, the greedy policy $\pi_{gre}(v)$ with respect to v is given by $\pi_{gre}(v)(\mathbf{s}) = \arg\max_a (B^a v)(\mathbf{s})$. Finally, if we let π^* denote the optimal policy and V^* denote its value function, we have the relationship $V^* = B^* V^*$, where $(B^* v)(\mathbf{s}) = \max_a (B^a v)(\mathbf{s})$. If, in addition, we define $Q^*(\mathbf{s}, a) = B^a V^*$ then we also have $\pi^*(\mathbf{s}) = \pi_{gre}(V^*)(\mathbf{s}) = \arg\max_a Q^*(\mathbf{s}, a)$. Given these definitions, the three fundamental methods for calculating π^* can be formulated as:

Policy iteration: Start with some $\pi^{(0)}$. Iterate $\pi^{(i+1)} \leftarrow \pi_{gre}(V^{\pi^{(i)}})$ until $\pi^{(i+1)} = \pi^{(i)}$. Return $\pi^* = \pi^{(i+1)}$.

Value iteration: Start with $v^{(0)}$. Iterate $v^{(i+1)} \leftarrow B^* v^{(i)}$ until $\|v^{(i+1)} - v^{(i)}\|_\infty < tol$. Return $\pi^* = \pi_{gre}(v^{(i+1)})$.

Linear programming: Calculate $V^* = \arg\min_v \sum_{\mathbf{s}} v(\mathbf{s})$ s.t. $v(\mathbf{s}) \ge (B^a v)(\mathbf{s})$ for all a, $\mathbf{s}$. Return $\pi^* = \pi_{gre}(V^*)$.

All three methods can be shown to produce optimal policies for the given MDP (Bertsekas 1995a; Puterman 1994) even though they do so in very different ways. However, to scale up it is necessary to exploit substantial structure in the MDP while also adopting some form of approximation for the optimal value function and policy.

2.1 Factored MDPs and linear approximators

A *factored* MDP is one that can be represented compactly by an additive reward function and a factored state transition model (Koller & Parr 1999; 2000). Specifically, we assume the reward function decomposes as $R(\mathbf{s}, a) = \sum_{r=1}^m R_{a,r}(\mathbf{s}_{a,r})$ where each local reward function $R_{a,r}$ is defined on a small set of variables $\mathbf{s}_{a,r}$. We assume the state transition model $P(\mathbf{s}'|\mathbf{s}, a)$ can be represented by a set of dynamic Bayesian networks (DBNs) on state variables—one for each action—where each DBN defines a compact transition model on a directed bipartite graph connecting state variables in consecutive time steps. Let $\mathbf{s}_{a,i}$ denote the parents of successor variable s'_i in the DBN for action a. To allow efficient optimization we assume the parent set $\mathbf{s}_{a,i}$ contains a small number of state variables from the previous time step. (It is possible to allow parents from the same time step, but we omit this possibility for simplicity.) Given this model, the probability of a successor state $\mathbf{s}'$ given a predecessor state $\mathbf{s}$ and action a is given by the product $P(\mathbf{s}'|\mathbf{s}, a) = \prod_{i=1}^n P(s'_i|\mathbf{s}_{a,i}, a)$. The main benefit of this representation is that it allows large MDPs to be encoded concisely: if the functions $R_{a,r}(\mathbf{s}_{a,r})$ and $P(s'_i|\mathbf{s}_{a,i}, a)$ depend on a small number of variables, they can be represented by small tables and efficiently combined to determine $R(\mathbf{s}, a)$ and $P(\mathbf{s}'|\mathbf{s}, a)$. It turns out that factored MDP representations interact well with linear functions.

We will approximate value functions with linear functions of the form $v_{\mathbf{w}}(\mathbf{s}) = \sum_{j=1}^k w_j b_j(\mathbf{s}_j)$, where $b_1, ..., b_k$ are a fixed set of basis functions and $\mathbf{s}_j$ denotes the variables on which basis b_j depends. For this approach to be effective the variable sets have to be small and interact well with the MDP model. Combining linear functions with factored MDPs provides many opportunities for feasible approximation. The first significant benefit of combining linear approximation with factored MDPs is that the approximate *action-value* (Q) function can also be represented concisely. Specifically, defining $q_{\mathbf{w}}(\mathbf{s}, a) = (B^a v_{\mathbf{w}})(\mathbf{s})$ we have

$$q_{\mathbf{w}}(\mathbf{s}, a) = \textstyle\sum_{r=1}^m R_{a,r}(\mathbf{s}_{a,r}) + \sum_{j=1}^k w_j h_{a,j}(\mathbf{s}_{a,\mathbf{j}}) \quad (1)$$

where $h_{a,j}(\mathbf{s}_{a,\mathbf{j}}) = \gamma \sum_{\mathbf{s}'_j} P(\mathbf{s}'_j|a, \mathbf{s}_{a,\mathbf{j}}) b_j(\mathbf{s}'_j)$ and $\mathbf{s}_{a,\mathbf{j}} = \bigcup_{s'_i \in \mathbf{s}'_j} \mathbf{s}_{a,i}$. That is, $\mathbf{s}_{a,i}$ are the parent variables of s'_i, and $\mathbf{s}_{a,\mathbf{j}}$ is the union of the parent variables of $s'_i \in \mathbf{s}'_j$. Thus, $h_{a,j}$ expresses the fact that in a factored MDP the expected future value of one component of the approximation depends only on the current state variables $\mathbf{s}_{a,\mathbf{j}}$ that are direct parents of the variables $\mathbf{s}'_j$ in b_j. If the MDP is sparsely connected then the variable sets in q will not be much larger than those in v. The ability to represent the state-action value function in a compact linear form immediately provides a feasible implementation of the greedy policy for $v_{\mathbf{w}}$, since $\pi_{gre}(v_{\mathbf{w}})(\mathbf{s}) = \arg\max_a \; q_{\mathbf{w}}(\mathbf{s}, a)$ by definition of π_{gre}, and $q_{\mathbf{w}}(\mathbf{s}, a)$ is efficiently determinable for each $\mathbf{s}$ and a.

2.2 Approximate linear programming

The combination of factored MDPs and linear approximators allows one to devise an efficient linear programming approach to generating value function approximations on a given basis. We refer to this as "approximate linear programming," or ALP for short.

ALP 1: Calculate $\mathbf{w} = \arg\min_{\mathbf{w}} \sum_{\mathbf{s}} v_{\mathbf{w}}(\mathbf{s})$ subject to $v_{\mathbf{w}}(\mathbf{s}) - q_{\mathbf{w}}(\mathbf{s}, a) \geq 0$ for all $\mathbf{s}$, a. Return $q_{\mathbf{w}}$.

Although this approach has been known since (Schweitzer & Seidman 1985) it has only recently been exploited with factored MDPs and linear approximators—resulting in feasible techniques that can scale up to large problems (Guestrin, Koller, & Parr 2001a; Schuurmans & Patrascu 2001). To understand how scaling-up is facilitated, first note that the LP objective can be encoded compactly

$$\textstyle\sum_{\mathbf{s}} v_{\mathbf{w}}(\mathbf{s}) = \sum_{j=1}^{k} w_j y_j \text{ where } y_j = 2^{n-|\mathbf{s}_j|} \sum_{\mathbf{s}_j} b_j(\mathbf{s}_j) \tag{2}$$

Here the y_j components can be easily precomputed by enumerating assignments to the small sets of variables in basis functions. Second, even though there are an exponential number of constraints—one for each state-action pair—these constraints have a structured representation that allows them to be handled more efficiently than explicit enumeration. Note that the constraint $v_{\mathbf{w}}(\mathbf{s}) - q_{\mathbf{w}}(\mathbf{s}, a) \geq 0$ can be rewritten as $\mathbf{c}_{(\mathbf{s},a)} \cdot \mathbf{w} \geq r_{(\mathbf{s},a)}$ where

$$\begin{array}{rcll} c_{(\mathbf{s},a),j} & = & b_j(\mathbf{s}_j) - h_{a,j}(\mathbf{s}_{a,\mathbf{j}}) & \text{for } j = 1, ..., k \\ r_{(\mathbf{s},a)} & = & \sum_r R_{a,r}(\mathbf{s}_{a,r}) & \end{array} \tag{3}$$

(recall the definition of $h_{a,j}$ in (1) above). This allows one to rewrite the linear program in a more conventional form:

ALP 2: Calculate $\mathbf{w} = \arg\min_{\mathbf{w}} \mathbf{y} \cdot \mathbf{w}$ subject to $\mathbf{c}_{(\mathbf{s},a)} \cdot \mathbf{w} \geq r_{(\mathbf{s},a)}$ for all $\mathbf{s}$, a. Or in matrix notation: $\mathbf{w} = \arg\min_{\mathbf{w}} \; \mathbf{y}^\top \mathbf{w}$ subject to $C\mathbf{w} \geq \mathbf{r}$.

The easy part to solving this linear program is handling the objective $\mathbf{y}$. The hard part is handling the large number of constraints. However, this can be simplified by noting that one can check $\mathbf{w}$ for constraint violations by determining $\min_{\mathbf{s},a} (\mathbf{c}_{(\mathbf{s},a)} \cdot \mathbf{w} - r_{(\mathbf{s},a)})$. This can be calculated by conducting a search over state configurations, one for each action, each of which can be solved efficiently by solving a cost network. Overall, this observation allows one to either re-represent the entire set of constraints in a compact form (Guestrin, Koller, & Parr 2001a), or efficiently search for a most violated constraint in a constraint generation scheme (Schuurmans & Patrascu 2001). Either approach yields reasonable approximation methods for factored MDPs.

2.3 Approximation error

One important issue is to ascertain the approximation error of the solutions produced by linear programming. Here one can show that ALP calculates a weight vector $\mathbf{w}$ that minimizes the L_1 error between $v_{\mathbf{w}}$ and V^*, subject to the constraints imposed by the linear program (de Farias & Van Roy 2001): Recall that the L_1 error is given by

$$\textstyle\sum_{\mathbf{s}} |v_{\mathbf{w}}(\mathbf{s}) - V^*(\mathbf{s})| \tag{4}$$

and note that the linear program constraint implies $v_{\mathbf{w}} \geq V^*$ (Bertsekas 1995a). Then, $\sum_{\mathbf{s}} |v_{\mathbf{w}}(\mathbf{s}) - V^*(\mathbf{s})| = \sum_{\mathbf{s}} v_{\mathbf{w}}(\mathbf{s}) - V^*(\mathbf{s}) = \sum_{\mathbf{s}} v_{\mathbf{w}}(\mathbf{s}) - C$ for $C = \sum_{\mathbf{s}} V^*(\mathbf{s})$.

Unfortunately, the L_1 error is not the best objective to minimize in this context. Normally we are interested in achieving a small L_∞ error, which is given by $\max_{\mathbf{s}} |v_{\mathbf{w}}(\mathbf{s}) - V^*(\mathbf{s})|$. Although minimizing L_∞ error is the ultimate goal, there are no known techniques for minimizing this objective directly. However, progress can be made by considering the closely related *Bellman error*

$$\max_{\mathbf{s}} |v_{\mathbf{w}}(\mathbf{s}) - \max_a q_{\mathbf{w}}(\mathbf{s}, a)| \tag{5}$$

Although Bellman error is much more convenient to work with than L_∞ error, it is still much harder minimize than the L_1 objective $\sum_{\mathbf{s}} v_{\mathbf{w}}(\mathbf{s})$. However, reducing Bellman error remains an important research question because it is directly related to the L_∞ error by the well known inequality: L_∞ *error* $\leq \frac{\gamma}{1-\gamma}$ *Bellman error* (Williams & Baird 1993).

3 Minimizing Bellman error

The first contribution of this paper is to show that some progress can be made in attacking Bellman error within the linear value function approach. However, dealing with Bellman error poses significant challenges. The first observation is that given a *fixed* weight vector $\mathbf{w}$, simply determining the Bellman error of $v_{\mathbf{w}}$ is a hard problem.

Theorem 1 It is *co-NP-complete* to determine whether the Bellman error of $v_{\mathbf{w}}$ for given $\mathbf{w}$ is less than a given δ.[1]

Proof sketch. The idea is to show that the complementary problem of deciding whether Bellman error is at least δ is NP-complete. First, it is easy to show the problem is in NP because a witness state can be used to certify a large Bellman error, and this can be verified in polynomial time using the structured computation (1). Next, to show the problem is NP-hard one can use a simple reduction from 3SAT: Given a 3CNF formula, let the state variables correspond to the propositional variables. Construct a basis function b_j for each clause, such that b_j indicates whether the clause is satisfied by the state assignment. Set the rewards to zero and the transition model to identity for each action, and set $\gamma = 0$ and $\mathbf{w} = \mathbf{1}$. The Bellman error (5) for this setup becomes $\max_{\mathbf{s}} \sum_{j=1}^{k} b_j(\mathbf{s}_j)$. If k is the number of clauses, then the Bellman error will be k if and only if the original 3CNF formula is satisfiable. ▮

Of course, the real goal is not just to evaluate Bellman error, but to *minimize* Bellman error. This, however, appears to pose an even greater computational challenge.

Theorem 2 It is *NP-hard* to determine whether there exists a weight vector $\mathbf{w}$ such that $v_{\mathbf{w}}$ has Bellman error less than

[1] Although Theorems 1 and 2 do not directly follow from the results of (Lusena, Goldsmith, & Mundhenk 2001; Mundhenk *et al.* 2000), the proofs are straightforward.

δ for a given δ. The problem, however, remains in $NP^{co\text{-}NP}$. (We conjecture that it is complete for this harder class.)

Proof sketch. First, to establish that the problem is in $NP^{co\text{-}NP}$, one can note that an acceptable $\mathbf{w}$ can be given as a certificate of small Bellman error, and this can then be verified by consulting a co-NP oracle. Second, NP-hardness follows from a trivial reduction from 3SAT: Given a 3CNF formula, let the state variables correspond to the propositional variables, and construct a local reward function r_j for each clause that is the same for each action, where r_j is set to be the indicator function for satisfaction of clause j. Choose a single trivial basis function $b_0 = 0$. Set the transition model to be identity for each action and set $\gamma = 0$. The Bellman error (5) in this setup becomes $\max_{\mathbf{s}} \sum_{j=1}^{k} r_j(\mathbf{s}_j)$. If k is the number of clauses, then $\min_{\mathbf{w}} \max_{\mathbf{s}} \sum_{j=1}^{k} r_j(\mathbf{s}_j)$ yields value k if and only if the original 3CNF formula is satisfiable. ∎

Thus, dealing with Bellman error appears to involve tackling hard problems. Nevertheless, we can make some progress toward developing practical algorithms.

First, for the problem of calculating Bellman error, an effective branch and bound strategy can easily be derived. Note that the Bellman error (5) reduces to two searches

$$\min_{\mathbf{s}} \Big(v_{\mathbf{w}}(\mathbf{s}) - \max_{a} q_{\mathbf{w}}(\mathbf{s}, a) \Big)$$
$$\max_{\mathbf{s}} \Big(v_{\mathbf{w}}(\mathbf{s}) - \max_{a} q_{\mathbf{w}}(\mathbf{s}, a) \Big) \qquad (6)$$

The first search is easy because it is equivalent to $\min_a \min_{\mathbf{s}} v_{\mathbf{w}}(\mathbf{s}) - q_{\mathbf{w}}(\mathbf{s}, a)$ and can be calculated by solving a cost network for each action (similar to the ALP problem above). However, the second search is much harder because it involves a maxi-min problem: $\max_{\mathbf{s}} \min_a v_{\mathbf{w}}(\mathbf{s}) - q_{\mathbf{w}}(\mathbf{s}, a)$. Nevertheless, we can calculate this value by employing a branch and bound search over states for each action. The key observation is that (6) is equivalent to

$$\max_{a} \max_{\mathbf{s}:\mathbf{s} \in \pi_{\mathbf{w}}^{-1}(a)} v_{\mathbf{w}}(\mathbf{s}) - q_{\mathbf{w}}(\mathbf{s}, a)$$

That is, for each state the minimum action is determined by the policy $\pi_{\mathbf{w}}(\mathbf{s}) = \arg\min_a q_{\mathbf{w}}(\mathbf{s}, a)$ defined by $\mathbf{w}$, and therefore for each action a we can restrict the search over states to regions of the space where a is the action selected by $\pi_{\mathbf{w}}$. That is, for a given action, say a_1, we search for a constrained maximum

$$\max_{\mathbf{s}} v_{\mathbf{w}}(\mathbf{s}) - q_{\mathbf{w}}(\mathbf{s}, a_1) \text{ s.t. } \begin{array}{l} q_{\mathbf{w}}(\mathbf{s}, a_1) \geq q_{\mathbf{w}}(\mathbf{s}, a_2) \\ \vdots \\ q_{\mathbf{w}}(\mathbf{s}, a_1) \geq q_{\mathbf{w}}(\mathbf{s}, a_\ell), \end{array} \qquad (7)$$

and similarly for actions $a_2, ..., a_\ell$. An exhaustive search over the state space could determine this quantity for each action. However, we can implement a much more efficient branch and bound search by calculating upper bounds on the maximum using the Lagrangian:

$$\begin{aligned} L(\mathbf{s}, \boldsymbol{\mu}) \;=\; & v_{\mathbf{w}}(\mathbf{s}) - q_{\mathbf{w}}(\mathbf{s}, a_1) \;+\; \mu_2[q_{\mathbf{w}}(\mathbf{s}, a_1) - q_{\mathbf{w}}(\mathbf{s}, a_2)] \\ & \vdots \\ & +\; \mu_\ell[q_{\mathbf{w}}(\mathbf{s}, a_1) - q_{\mathbf{w}}(\mathbf{s}, a_\ell)] \end{aligned}$$

Cycle problem

$n =$	12	15	18	20	24	28	32
$N =$	4e4	3e5	3e6	1e7	2e8	3e9	4e10
Nodes	194	288	384	392	466	832	700
Time (s)	63	131	225	279	451	959	1086
B.Err.	8.1	9.6	12.3	13.8	16.7	19.5	22.4

3legs problem

$n =$	13	16	19	22	25	28
$N =$	8e4	7e5	5e6	4e7	3e8	3e9
Nodes	2150	802	6950	1327	18394	3124
Time (s)	525	291	3454	866	14639	2971
B.Err.	8.6	12.9	12.9	17.2	17.2	21.5

Table 1: Bellman results: singleton bases[2]

It is easy to show that $\max_{\mathbf{s}} L(\mathbf{s}, \boldsymbol{\mu})$ gives an upper bound on the constrained maximum (7) for any $\boldsymbol{\mu} \geq 0$. (The unconstrained maximum is clearly an upper bound on the constrained maximum, and if the constraints are satisfied we must again have an upper bound since $\boldsymbol{\mu}$ is nonnegative.) What is crucial about this upper bound is that it can be efficiently calculated for fixed $\boldsymbol{\mu}$ by solving a cost network over state variables (because it is just a weighted sum of functions that in principle share the same structure). By choosing good penalties $\boldsymbol{\mu}$ (and adapting them using standard subgradient optimization (Bertsekas 1995b)) we obtain an effective pruning strategy that can solve for the Bellman error in far less time than explicit enumeration. For example, Table 1 demonstrates results on two problems from (Guestrin, Koller, & Parr 2001a) where in one case the Bellman error is calculated by only expanding 700 search nodes even when the problem has 4 billion states and 33 actions.[2] Overall, this appears to be a practical algorithm, and we use it to calculate the Bellman error for all of the value approximations we produce below.

The second question—minimizing Bellman error—remains an open problem to the best of our knowledge, and we do not have an exact method. Nevertheless, a similar branch and bound search strategy can be used to implement an approximate policy iteration scheme.

API: Start with an arbitrary $\mathbf{w}^{(0)}$. Iterate $(\mathbf{w}^{(i+1)}, \delta) \leftarrow \arg\min_{(\mathbf{w},\delta)} \delta$ subject to $-\delta \leq v_{\mathbf{w}}(\mathbf{s}) - q_{\mathbf{w}}(\mathbf{s}, \pi_{\mathbf{w}^{(i)}}(\mathbf{s})) \leq \delta$ for all $\mathbf{s}$ and $\delta \geq 0$, until $\mathbf{w}^{(i+1)} \doteq \mathbf{w}^{(i)}$.

This procedure uses a linear program to recover the weight

[2] We conducted most of our experiments on the computer network problems described in (Guestrin, Koller, & Parr 2001a; 2001b). For these problems there is a directed network of computer systems $s_1, ..., s_n$ where each system is either up ($s_i = 1$) or down ($s_i = 0$). Systems can spontaneously go down with some probability at each step, but this probability is increased if an immediately preceding machine in the network is down. There are $n + 1$ actions: do nothing (the default) and reboot machine i. The reward in a state is simply the sum of systems that are up, with a bonus reward of 1 if system 1 (the server) is up. The transition probabilities of the state of a computer depend only on its previous state and the state(s) of any parent computers in the network. The two network topologies reported here are "cycle" and "three legs."

	ALP		API		
n	B.Err.	time (s)	B.Err.	time (s)	Iters.
cycle problem, singleton bases					
5	2.8	2	0.9	160	10
8	4.1	8	1.8	1,600	16
10	6.7	14	2.4	5,672	22
3legs problem, singleton bases					
4	1.8	1	0.6	383	7
7	4.0	3	1.0	697	14
10	3.9	9	1.8	16,357	19

Table 2: API versus ALP results[2]

vector $\mathbf{w}^{(i+1)}$ that minimizes the one step error in approximating the value of the current policy $\pi_{\mathbf{w}^{(i)}}$ defined by $\mathbf{w}^{(i)}$. Here a branch and bound search is used to generate constraints for this linear program. The overall procedure generalizes that of (Guestrin, Koller, & Parr 2001a), and produces the same solutions in cases where both apply. However, the new technique does not require the additional assumption of a "default action" nor an explicit representation of the intermediate policies (in their case, a decision list). The drawback is that one has to perform a branch and bound search instead of solving cost networks to generate the constraints. Figure 2 shows that the new API procedure is much more expensive than ALP, but clearly produces better approximations given the same basis. However, API does not fully minimize the Bellman error of the final weight vector. Instead it achieves a bounded approximation of the Bellman error of the optimal weight vector (Guestrin, Koller, & Parr 2001a). Overall, this method appears to be too costly to justify the modest gains in accuracy it offers.

4 Minimizing L_1 error

Attacking Bellman error directly with API may yield reasonable approximations, but comes at the expense of solving many linear programs (one per policy iteration) and performing branch and bound search. Clearly the direct ALP approach is much faster, but unfortunately produces noticeably worse Bellman error than API. This raises the question of whether one can improve the approximation error of ALP without resorting to branch and bound. A related question is understanding how the basis functions might be selected in the first place. To address both questions simultaneously we investigate strategies for *automatically* constructing sets of basis functions to achieve good approximation error. We will follow the generic algorithm of Figure 1.

The three unknowns in this procedure are (1) the procedure for generating candidate domains, (2) the procedure for constructing a basis function given a candidate domain, and (3) the procedure for scoring the potential contribution of a basis function to reducing approximation error.

4.1 Scoring candidate bases

The first issue we tackle is scoring the potential merit of a basis function b_{k+1} given a current basis and weight vector. Ideally, we would like to measure the new basis function's effect on Bellman error. However, we have seen that measuring Bellman error is a hard problem. Moreover, the ALP procedure does not minimize Bellman error, it only minimizes L_1 error. Therefore, we are really only in a position to conveniently evaluate a basis function's effect on L_1 error and hope that this leads to a reduction in Bellman error as a side effect. Our experimental results below show that this is generally the case, although it is clearly not always true. For now we focus on attacking L_1 error.

Greedy basis function selection Start with constant function $b_0(\emptyset)$ and initial solution w_0, and iterate:
Given a current basis $b_1(\mathbf{s}_1), ..., b_k(\mathbf{s}_k)$ and weights $\mathbf{w}$, note that each basis function b_j is defined on a subset of the state variables, $\mathbf{s}_j \subseteq \{s_1, ..., s_n\}$, which we refer to as its *domain*.
(1) Generate a set of candidate domains $\mathbf{s}'_1, ..., \mathbf{s}'_J$ from the current domains.
(2) For each candidate domain $\mathbf{s}'_j$ construct a basis function.
(3) For each basis function, score its ability to reduce approximation error. Add the best candidate to the basis, and re-solve the LP to calculate new weights on the entire basis.

Figure 1: Greedy basis function selection procedure

The first issue is determining whether a new basis function will allow any progress to be made in the linear program at all. To do this note that the dual of the linear program in **ALP 2** is

$$\max_{\boldsymbol{\lambda}} \ \boldsymbol{\lambda}^\top \mathbf{r} \ \text{ subject to } \ \boldsymbol{\lambda}^\top C \leq \mathbf{y}^\top, \boldsymbol{\lambda} \geq 0$$

(Technically we must restrict $\mathbf{w} \geq 0$, but this does not pose any insurmountable difficulties.) Imagine that we have restricted attention to some basis set $b_1, ..., b_k$ and solved the linear program with respect to this set (thus fixing $w_j = 0$ for $j > k$). Let $(\mathbf{w}, \boldsymbol{\lambda})$ be a solution pair to this linear program and let $B = \{i : \lambda_i > 0\}$ be the indices of the active primal constraints. Then by complementary slackness we must have $C_B \mathbf{w} = \mathbf{r}_B$ and hence $\mathbf{w} = C_B^{-1} \mathbf{r}_B$ where C_B is square and invertible. By a further application of complementary slackness we have $\boldsymbol{\lambda}_B^\top C_B = \mathbf{y}^\top$ and therefore $\boldsymbol{\lambda}_B^\top = \mathbf{y}^T C_B^{-1}$. Now, consider what happens if we wish to add a new basis function b_{k+1} to $\{b_1, ..., b_k\}$. This new basis function generates a new column of C which imposes a new constraint $\boldsymbol{\lambda}^\top \mathbf{c}_{:,k+1} \leq y_{k+1}$ on $\boldsymbol{\lambda}$. If the current $\boldsymbol{\lambda}$ is feasible for the new constraint, then $\boldsymbol{\lambda}$ is already a global solution to the dual problem involving this basis function, and we can make no further progress on improving the primal LP objective. This immediately yields an efficient test of whether b_{k+1} allows any progress to be made: If b_{k+1} generates a column $\mathbf{c}_{B,k+1}$ on the active constraints such that the dual constraint $\boldsymbol{\lambda}_B^\top \mathbf{c}_{B,k+1} > y_{k+1}$ is violated, then b_{k+1} allows some progress. On the other hand, if b_{k+1} satisfies the dual constraint it is useless. This provides an efficient test because the column $\mathbf{c}_{B,k+1}$ can easily be computed for b_{k+1} on the k active constraints in B.

Next to *quantify* the potential improvement of adding a basis function we consider plausible ways to score candidates. Note that any scoring function must strike a compromise between accuracy and computational cost that lies

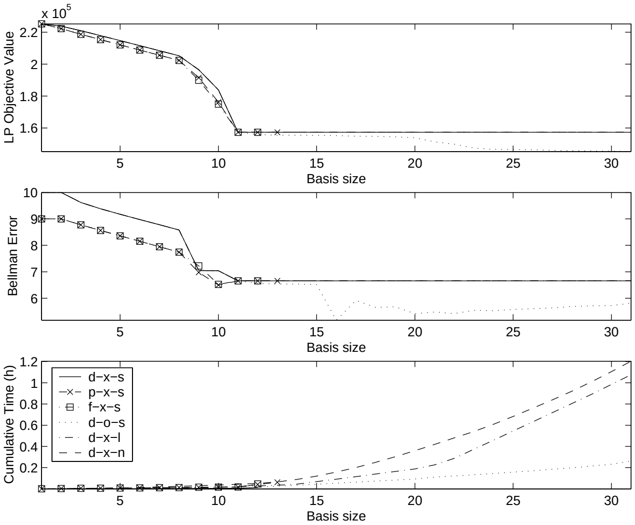

Figure 2: Basis function selection results on the cycle problem ($n = 10$).[2] Legend shows combinations of basis construction strategies. **Scores**: d = dual, p = partial, f = full. **Basis**: x = XOR, o = optimized. **Domain**: s = sequential, l = lattice, n = neighbor.

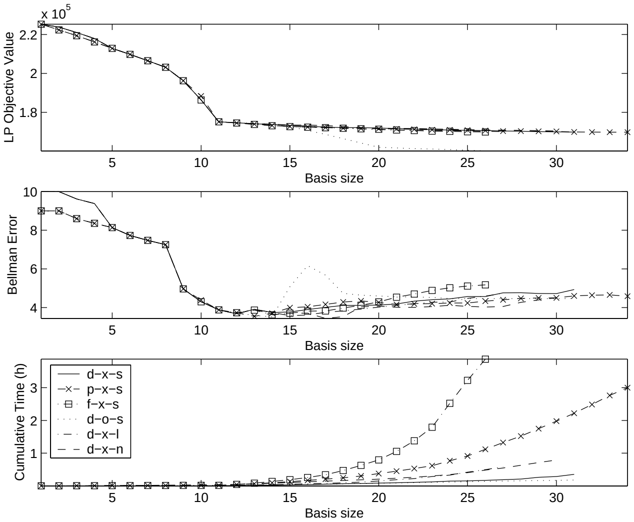

Figure 3: Basis function selection results on the 3legs problem ($n = 10$).[2] Legend shows combinations of basis construction strategies. **Scores**: d = dual, p = partial, f = full. **Basis**: x = XOR, o = optimized. **Domain**: s = sequential, l = lattice, n = neighbor.

between two extremes: The cheapest scoring function is just the degree of dual constraint violation $\boldsymbol{\lambda}_B^\top \mathbf{c}_{B,k+1} - y_{k+1}$, which we refer to as the *dual score*. In this case a larger dual constraint violation means that, to the first degree, the basis function will decrease the linear program objective faster than another basis function with a smaller degree of constraint violation. However, the dual score ignores the primal constraints and therefore may not always be predictive. Clearly, the most accurate, but most expensive scoring technique is simply to re-solve the entire linear program with the new basis function to determine its exact effect. We refer to this strategy as the *full LP score*. Our experimental results show that the full LP score does not yield noticeable improvements in approximation over the dual score and is much more expensive; see Figures 2 and 3. (We also experimented with an intermediate alternative, *partial LP score*, but it also yielded no noticeable benefit, so we omit a detailed description.) Since the dual score achieves comparable reductions to the more expensive scoring methods in our experiments, we concentrate solely on this cheap alternative below.

4.2 Constructing candidate bases

Next we turn to the problem of constructing the basis functions themselves. Assume a set of domain variables has already been selected. (Below we consider how to choose candidate domains.) Any new function we construct on the given set of variables must be nonlinear to ensure that it has a non-vacuous effect. Here one could consider a wide range of possible representations, including decision trees and decision diagrams, neural networks, etc. However, for simplicity we will just focus on table-based representations where a basis function is just represented as a lookup table over assignments to the given variables. A useful advantage of the table-based representation, particularly in the context of ALP, is that one can *optimize* the lookup table values themselves to yield a basis function b_{k+1} that maximizes the dual score. This can be done by solving a small auxiliary linear program. (Unfortunately we have to omit the derivation in this shortened paper.) The interesting thing about this approach is that it implicitly considers an infinite space of basis functions to select the best candidate. We refer to this approach as the *optimized basis*. We compared the optimized basis approach to a simpler method that just considered a fixed basis function for each domain. In particular we considered the *XOR basis* for two-valued state variables, $s_i \in \{0, 1\}$, which is defined by $b(s_i) = (-1)^{s_i}$, $b(s_i, s_j) = (-1)^{s_i}(-1)^{s_j}$, $b(s_1, .., s_k) = (-1)^{s_1} \cdots (-1)^{s_k}$, etc. Figures 2 and 3 show that the optimized basis yields noticeable benefits in reducing the linear program objective with little additional cost to the fixed XOR basis.

4.3 Selecting candidate domains

The final issue we consider is how to select the candidate domains on which to build new basis functions. A constraint we wish to maintain is to keep the domain sizes small so that the cost networks do not become unwieldy. We consider three approaches that differ in how tightly they control the growth in domain size. The most conservative approach, which we refer to as *sequential*, only considers a domain once every smaller sized domain has been used. That is, after the trivial constant basis, it then considers only singleton domains, and then considers pairs of state variables only once the singletons are exhausted, etc. A slightly less conservative approach, which we call *lattice*, allows a candidate domain to be considered if and only if all of its proper subsets have already been added to the current basis. Finally, the least conservative approach, *neighbors*, allows a candi-

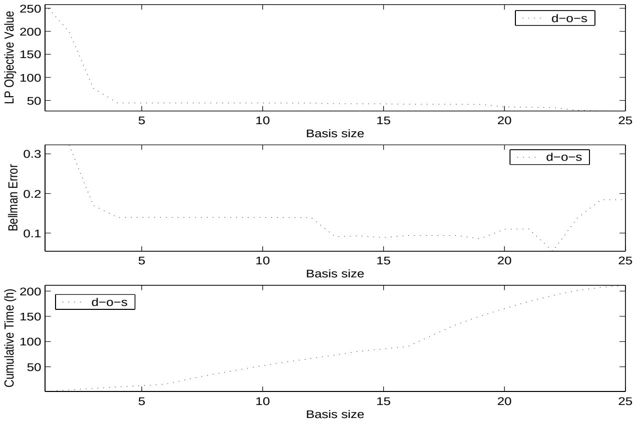

Figure 4: Basis function selection results on the resource allocation problem (2 resources, 4 tasks, 16 actions).[3]

date domain to be considered as soon as it can be constructed as the union of a current domain plus one new state variable. Figures 2 and 3 compare each of these three methods. Surprisingly, the least expensive sequential strategy matches the others in reducing the linear program objective.

4.4 Observations

Overall, the experiments exhibit interesting trends; see Figures 2–4.[3] The optimized basis approach is clearly the most effective at improving the linear program objective, although it shows a surprising "over-fitting" effect on Bellman error. Most of the remaining methods produce indistinguishable results, except in runtime. The results are somewhat promising in that the adaptive basis selection technique reduced the Bellman error of the fixed singleton basis used in the earlier experiments (6.7 to 5 for the cycle problem; 3.9 to 3.8 for the 3legs problem). Also, in every case substantial reductions were achieved in the linear programming objective; particularly using the optimized basis functions. However, these results still do not match that of the computationally much more expensive API method of Section 3. It seems apparent that for the ALP approach to achieve comparable Bellman error, one may have to add a substantial number of basis functions. Investigating the number of basis functions necessary to make ALP truly competitive with API in terms of Bellman error remains future work.

Acknowledgments

Research supported by NSERC, CITO and MITACS. Thanks to Ron Parr for his patient discussions.

References

Bertsekas, D., and Tsitsiklis, J. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Bertsekas, D. 1995a. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific.

Bertsekas, D. 1995b. *Nonlinear Optimization*. Athena Scientific.

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

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 2000. Stochastic dynamic programming with factored representations. *Artificial Intelligence*.

de Farias, D., and Van Roy, B. 2001. The linear programming approach to approximate dynamic programming.

Dietterich, T. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR* 13:227–303.

Guestrin, C.; Koller, D.; and Parr, R. 2001a. Max-norm projection for factored MDPs. In *Proceedings IJCAI*.

Guestrin, C.; Koller, D.; and Parr, R. 2001b. Multiagent planning with factored MDPs. In *Proceedings NIPS*.

Koller, D., and Parr, R. 1999. Computing factored value functions for policies in structured MDPs. In *Proceedings IJCAI*.

Koller, D., and Parr, R. 2000. Policy iteration for factored MDPs. In *Proceedings UAI*.

Lusena, C.; Goldsmith, J.; and Mundhenk, M. 2001. Non-approximability results for partially observable Markov decision processes. *JAIR* 14:83–103.

Mundhenk, M.; Goldsmith, J.; Lusena, C.; and Allender, E. 2000. Complexity of finite-horizon Markov decision processes. *JACM* 47(4):681–720.

Puterman, M. 1994. *Markov Decision Processes: Discrete Dynamic Programming*. Wiley.

Sallans, B., and Hinton, G. 2000. Using free energies to represent Q-values in a multiagent reinforcement learning task. In *Proceedings NIPS*.

Schuurmans, D., and Patrascu, R. 2001. Direct value-approximation for factored MDPs. In *Proceedings NIPS*.

Schweitzer, P., and Seidman, A. 1985. Generalized polynomial approximations in Markovian decision problems. *J. Math. Anal. and Appl.* 110:568–582.

Trick, M. A., and Zin, S. E. 1997. Spline approximations to value functions: A linear programming approach. *Macroeconomic Dynamics* 1:255–277.

Williams, R., and Baird, L. 1993. Tight performance bounds on greedy policies based on imperfect value functions. Technical report, Northeastern University.

[3] We conducted an additional experiment in a generic resource allocation domain, where the problem is to allocate resources to a number of heterogeneous tasks. In this case, the state of the MDP is described by n *task* and m *resource* binary variables. If task T_i, $i = 1, \ldots, n$, is not active at the current time step, then it activates at the next time step with a probability that need not be the same for each task. At every stage resource variable S_j, $j = 1, \ldots, m$, is replenished or depleted (if currently applied to a task) stochastically. Applying a number of resources to an active task may fail to bring the task to completion, thus acting in a simple noisy-or fashion. Completed tasks result in rewards which are summed together. Actions assign free resources to needy tasks at a cost linear in the number of assigned resources. There are as many actions as there are ways to assign resources to tasks.