International Joint Conference on Artificial Intelligence (IJCAI-01), vol. 1, pp. 673 - 680, Seattle, Washington, August 2001.

Max-norm Projections for Factored MDPs

Carlos Guestrin
Computer Science Dept.
Stanford University
guestrin@cs.stanford.edu

Daphne Koller
Computer Science Dept.
Stanford University
koller@cs.stanford.edu

Ronald Parr
Computer Science Dept.
Duke University
parr@cs.duke.edu

Abstract

Markov Decision Processes (MDPs) provide a coherent mathematical framework for planning under uncertainty. However, exact MDP solution algorithms require the manipulation of a *value function*, which specifies a value for each state in the system. Most real-world MDPs are too large for such a representation to be feasible, preventing the use of exact MDP algorithms. Various approximate solution algorithms have been proposed, many of which use a linear combination of basis functions as a compact approximation to the value function. Almost all of these algorithms use an approximation based on the (weighted) $\mathcal{L}_2$-norm (Euclidean distance); this approach prevents the application of standard convergence results for MDP algorithms, all of which are based on max-norm. This paper makes two contributions. First, it presents the first approximate MDP solution algorithms — both value and policy iteration — that use max-norm projection, thereby directly optimizing the quantity required to obtain the best error bounds. Second, it shows how these algorithms can be applied efficiently in the context of *factored MDPs*, where the transition model is specified using a dynamic Bayesian network.

1 Introduction

Over the last few years, *Markov Decision Processes (MDPs)* have been used as the basic semantics for optimal planning for decision theoretic agents in stochastic environments. In the MDP framework, the system is modeled via a set of states which evolve stochastically. The key problem with this representation is that, in virtually any real-life domain, the state space is quite large. However, many large MDPs have significant internal structure, and can be modeled compactly if the structure is exploited in the representation.

Factored MDPs [Boutilier *et al.*, 1999] are one approach to representing large, structured MDPs compactly. In this framework, a state is implicitly described by an assignment to some set of *state variables*. A *dynamic Bayesian network (DBN)* [Dean and Kanazawa, 1989] can then allow a compact representation of the transition model, by exploiting the fact that the transition of a variable often depends only on a small number of other variables. Furthermore, the momentary rewards can often also be decomposed as a sum of rewards related to individual variables or small clusters of variables.

Even when a large MDP can be represented compactly, e.g., in a factored way, solving it exactly is still intractable: Exact MDP solution algorithms require the manipulation of a value function, whose representation is linear in the number of states, which is exponential in the number of state variables. One approach is to approximate the solution using an approximate value function with a compact representation. A common choice is the use of *linear* value functions as an approximation — value functions that are a linear combination of basis functions.

This paper makes a twofold contribution. First, we provide a new approach for approximately solving MDPs using a linear value function. Previous approaches to linear function approximation typically have utilized a least squares ($\mathcal{L}_2$-norm) approximation to the value function. Least squares approximations are incompatible with most convergence analyses for MDPs, which are based on max-norm. We provide the first MDP solution algorithms — both value iteration and policy iteration — that use a linear max-norm projection to approximate the value function, thereby directly optimizing the quantity required to obtain the best error bounds.

Second, we show how to exploit the structure of the problem in order to apply this technique to factored MDPs. Our work builds on the ideas of Koller and Parr [1999; 2000], by using *factored (linear) value functions*, where each basis function is restricted to some small subset of the domain variables. We show that, for a factored MDP and factored value functions, various key operations can be implemented in closed form without enumerating the entire state space. Thus, our max-norm algorithms can be implemented efficiently, even though the size of the state space grows exponentially in the number of variables.

2 Markov decision processes

A *Markov Decision Process (MDP)* is defined as a 4-tuple (S, A, R, P) where: S is a finite set of $|S|$ states; A is a set of actions; R is a *reward function* $R : S \times A \mapsto \mathbb{R}$, such that $R(s, a)$ represents the reward obtained by the agent in state s after taking action a; and P is a *Markovian transition model* where $P(s' \mid s, a)$ represents the probability of going from state s to state s' with action a.

We will be assuming that the MDP has an infinite horizon and that future rewards are discounted exponentially with a discount factor $\gamma \in [0, 1)$. A stationary policy π for an MDP is a mapping $\pi : S \mapsto A$, where $\pi(s)$ is the action the agent takes at state s. The policy is associated with a *value function* $\mathcal{V}_\pi \in \mathbb{R}^{|S|}$, where $\mathcal{V}_\pi(s)$ is the discounted cumulative value that the agent gets if it starts at state s. The value function for a fixed policy is the fixed point of a set of equations that define the value of a state in terms of the value of its possible successor states. More formally, we define:

Definition 2.1 *The* DP operator, $\mathcal{T}_\pi$, *for a fixed stationary policy* π *is:*

$$\mathcal{T}_\pi \mathcal{V}(s) = R(s, \pi(s)) + \gamma \Sigma_{s'} P(s' | s, \pi(s)) \mathcal{V}(s').$$

$\mathcal{V}_\pi$ *is the fixed point of* $\mathcal{T}_\pi$: $\mathcal{V}_\pi = \mathcal{T}_\pi \mathcal{V}_\pi$. ∎

The optimal value function $\mathcal{V}^*$ is also defined by a set of equations. In this case, the value of a state must be the maximal value achievable by any action at that state. More precisely, we define:

Definition 2.2 *The* Bellman operator, $\mathcal{T}^*$*, is:*

$$\mathcal{T}^*\mathcal{V}(s) = \max_a [R(s,a) + \gamma \Sigma_{s'} P(s' \mid s,a)\mathcal{V}(s')].$$

$\mathcal{V}^*$ *is the fixed point of* $\mathcal{T}^*$*:* $\mathcal{V}^* = \mathcal{T}^*\mathcal{V}^*$. ∎

For any value function $\mathcal{V}$, we can define the policy obtained by acting greedily relative to $\mathcal{V}$. In other words, at each state, we take the action that maximizes the one-step utility, assuming that $\mathcal{V}$ represents our long-term utility achieved at the next state. More precisely, we define

$$\mathit{Greedy}(\mathcal{V})(s) = \arg\max_a [R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\mathcal{V}(s')].$$

The greedy policy relative to the optimal value function $\mathcal{V}^*$ is the optimal policy $\pi^* = \mathit{Greedy}(\mathcal{V}^*)$. There are several algorithms to compute the optimal policy, we will focus on the two most used: value iteration and policy iteration.

Value iteration relies on the fact that the Bellman operator is a *contraction* — it is guaranteed to reduce the *max-norm* ($\mathcal{L}_\infty$) distance between any pair of value functions by a factor of at least γ. This property guarantees that the Bellman operator has a unique fixed point $\mathcal{V}^*$ [Puterman, 1994]. Value iteration exploits this property, approaching the fixed point through successive applications of the Bellman operator: $\mathcal{V}^{(t+1)} = \mathcal{T}^*\mathcal{V}^{(t)}$. After a finite number of iterations, the greedy policy $\mathit{Greedy}(\mathcal{V}^{(t)})$ will be the optimal policy.

Policy iteration iterates over policies. Each iteration consists of two phases. *Value determination* computes, for a policy $\pi^{(t)}$, the value function $\mathcal{V}_{\pi^{(t)}}$, by finding the fixed point of: $\mathcal{T}_{\pi^{(t)}}\mathcal{V}_{\pi^{(t)}} = \mathcal{V}_{\pi^{(t)}}$. *Policy improvement* defines the next policy as $\pi^{(t+1)}(s) = \mathit{Greedy}(\mathcal{V}_{\pi^{(t)}})$. It can be shown that this process converges to the optimal policy.

3 Solving MDPs with max-norm projections

In many domains, the state space is very large, and we need to perform our computations using approximate value functions. A very popular choice is to approximate a value function using *linear regression*. Here, we define our space of allowable value functions $\mathcal{V} \in \mathcal{H} \subseteq \mathbb{R}^{|S|}$ via a set of *basis functions* $H = \{h_1, \ldots, h_k\}$. A *linear value function* over H is a function $\mathcal{V}$ that can be written as $\mathcal{V}(s) = \sum_{j=1}^k w_j h_j(s)$ for some coefficients $\mathbf{w} = (w_1, \ldots, w_k)'$. We define $\mathcal{H}$ to be the linear subspace of $\mathbb{R}^{|S|}$ spanned by the basis functions H. It is useful to define an $|S| \times k$ matrix A whose columns are the k basis functions, viewed as vectors. Our approximate value function is then represented by $A\mathbf{w}$.

Linear value functions: The idea of using linear value functions for dynamic programming was proposed, initially, by Bellman et al. [1963] and has been further explored recently [Tsitsiklis and Van Roy, 1996; Koller and Parr, 1999; 2000]. The basic idea is as follows: in the solution algorithms, whether value or policy iteration, we use only value functions within $\mathcal{H}$. Whenever the algorithm takes a step that results in a value function $\mathcal{V}$ that is outside this space, we *project* the result back into the space by finding the value function within the space which is close to $\mathcal{V}$. More precisely:

Definition 3.1 *A* projection operator Π *is a mapping* $\Pi : \mathbb{R}^{|S|} \rightarrow \mathcal{H}$. Π *is said to be a* projection w.r.t. a norm $\|\cdot\|$ *if:* $\Pi\mathcal{V} = A\mathbf{w}^*$ *such that* $\mathbf{w}^* \in \arg\min_{\mathbf{w}} \|A\mathbf{w} - \mathcal{V}\|$. ∎

Unfortunately, these existing algorithms all suffer from a problem that we might call "norm incompatibility." When computing the projection, they all utilize the standard projection operator with respect to $\mathcal{L}_2$ norm or a *weighted* $\mathcal{L}_2$ norm. On the other hand, most of the convergence and error analyses for MDP algorithms utilize max-norm. This incompatibility has made it difficult to provide error guarantees.

In this section, we propose a new approach that addresses the issue of norm compatibility. Our key idea is the use of a projection operator in $\mathcal{L}_\infty$ norm. This problem has been studied in the optimization literature as the problem of finding the Chebyshev solution to an overdetermined linear system of equations [Cheney, 1982]. The problem is defined as finding $\mathbf{w}^*$ such that:

$$\mathbf{w}^* \in \arg\min_{\mathbf{w}} \|C\mathbf{w} - \mathbf{b}\|_\infty . \qquad (1)$$

We will use an algorithm due to Stiefel [1960], that solves this problem by linear programming:

$$\begin{array}{ll} \text{Variables:} & w_1, \ldots, w_k, \phi\,; \\ \text{Minimize:} & \phi\,; \\ \text{Subject to:} & \phi \geq \sum_{j=1}^k c_{ij} w_j - b_i \quad \text{and} \\ & \phi \geq b_i - \sum_{j=1}^k c_{ij} w_j, \quad i = 1\ldots|S|\,. \end{array} \qquad (2)$$

For the solution $(\mathbf{w}^*, \phi^*)$ of this linear program, $\mathbf{w}^*$ is the solution of Eq. (1) and ϕ is the $\mathcal{L}_\infty$ projection error. Note that this LP only has $k+1$ variables. However, there are $2 \cdot |S|$ constraints, which makes it impractical for large state spaces. In the remainder of this section, we will discuss how this projection addresses the norm incompatibility problem. In Section 4, we will show that, in factored MDPs, all the $2 \cdot |S|$ constraints can be represented efficiently, leading to a tractable algorithm.

Approximate Value iteration: The basic idea of approximate value iteration is quite simple. We define an $\mathcal{L}_\infty$ projection operator Π_∞ that takes a value function $\mathcal{V}$ and finds $\mathbf{w}$ that minimizes $\|A\mathbf{w} - \mathcal{V}\|_\infty$. This is an instance of Eq. (1) and can be solved by Eq. (2). The algorithm alternates applications of the Bellman operator $\mathcal{T}^*$ and projection steps Π_∞:

$$\overline{\mathcal{V}}^{(t+1)} = \mathcal{T}^* A\mathbf{w}^{(t)}; \quad \text{and} \quad A\mathbf{w}^{(t+1)} = \Pi_\infty \overline{\mathcal{V}}^{(t+1)}.$$

In standard value iteration, we only need to perform the first step. However, $\overline{\mathcal{V}}^{(t+1)}$ may not be in $\mathcal{H}$, so we need to add the second step, the projection step, to the process. We can analyze this process, bounding the overall error between our approximate value function $A\mathbf{w}^{(t)}$ and the optimal value function $\mathcal{V}^*$. This analysis shows that the overall error depends on the single-step max-norm projection errors $\phi^{(t+1)} = \left\| A\mathbf{w}^{(t+1)} - \overline{\mathcal{V}}^{(t+1)} \right\|_\infty$. Thus, using the max-norm projection, we can minimize these projection errors directly, we omit the analysis for lack of space. Note that, in approximate value iteration, the error introduced by successive approximations may grow unboundedly. As we will show, this cannot happen in approximate policy iteration.

Approximate Policy iteration: As we discussed, policy iteration is composed of two steps: value determination and policy improvement. Our algorithm performs the policy improvement step exactly. In the value determination step, the value function is approximated through a linear combination of basis functions. Consider the value determination for a policy $\pi^{(t)}$. Define $R_{\pi^{(t)}}(s) = R(s, \pi^{(t)}(s))$, and $P_{\pi^{(t)}}(s' \mid s) = P(s' \mid s, a = \pi^{(t)}(s))$. We can now rewrite the value determination step in terms of matrices and vectors. If we view $\mathcal{V}_{\pi^{(t)}}$ and $R_{\pi^{(t)}}$ as $|S|$-vectors, and $P_{\pi^{(t)}}$ as an $|S| \times |S|$ matrix, we have the equations: $\mathcal{V}_{\pi^{(t)}} = R_{\pi^{(t)}} + \gamma P_{\pi^{(t)}} \mathcal{V}_{\pi^{(t)}}$. This is a system of linear equations with one equation for each state, which can only be solved exactly for small $|S|$. Our goal is to provide an approximate solution, within $\mathcal{H}$. More precisely, we want to find:

$$\begin{aligned}\mathbf{w}^{(t)} &= \arg\min_{\mathbf{w}} \|A\mathbf{w} - (R_{\pi^{(t)}} + \gamma P_{\pi^{(t)}} A\mathbf{w})\|_\infty ;\\ &= \arg\min_{\mathbf{w}} \left\|\left(A - \gamma P_{\pi^{(t)}} A\right)\mathbf{w}^{(t)} - R_{\pi^{(t)}}\right\|_\infty .\end{aligned}$$

This minimization is another instance of an $\mathcal{L}_\infty$ projection (Eq. (1)), where $C = (A - \gamma P_{\pi^{(t)}} A)$ and $\mathbf{b} = R_{\pi^{(t)}}$, and can be solved using the linear program of Eq. (2). Thus, our approximate policy iteration alternates between two steps:

$$\begin{aligned}\mathbf{w}^{(t)} &= \arg\min_{\mathbf{w}} \|A\mathbf{w} - (R_{\pi^{(t)}} + \gamma P_{\pi^{(t)}} A\mathbf{w})\|_\infty ;\\ \pi^{(t+1)} &= \mathit{Greedy}(A\mathbf{w}^{(t)}).\end{aligned}$$

For the analysis, we define the projection error for the policy iteration case, i.e., the error resulting from the approximate value determination step: $\beta^{(t)} = \left\|A\mathbf{w}^{(t)} - \left(R_{\pi^{(t)}} + \gamma P_{\pi^{(t)}} A\mathbf{w}^{(t)}\right)\right\|_\infty$. Unlike in approximate value iteration, the error is bounded in this case:

Lemma 3.2 *There exists a constant $\beta_P < \infty$ such that $\beta_P \geq \beta^{(t)}$ for all iterations t of the algorithm.* ∎

Finally, we define *discounted accumulated projection error* as $\overline{\beta}^{(t)} = \beta^{(t)} + \gamma\overline{\beta}^{(t-1)}$; $\overline{\beta}^{(0)} = 0$. Lemma 3.2 implies that the accumulated error remains bounded: $\overline{\beta}^{(t)} \leq \frac{\beta_P(1-\gamma^t)}{1-\gamma}$.

We can now bound the error in the value function resulting from approximate policy iteration:

Theorem 3.3 *In the approximate policy iteration algorithm, the distance between our approximate value function at iteration t and the optimal value function is bounded by:*

$$\left\|A\mathbf{w}^{(t)} - \mathcal{V}^*\right\|_\infty \leq \gamma^t \left\|A\mathbf{w}^{(0)} - \mathcal{V}^*\right\|_\infty + \frac{2\gamma\overline{\beta}^{(t)}}{(1-\gamma)^2}. \blacksquare$$

In words, the difference between our approximation at iteration t and the optimal value function is bounded by the sum of two terms. The first term is present in standard policy iteration and goes to zero exponentially fast. The second is the discounted accumulated projection error and, as the theorem shows, is bounded. This second term can be minimized by choosing $\mathbf{w}^{(t)}$ as the one that minimizes $\left\|A\mathbf{w}^{(t)} - \left(R_{\pi^{(t)}} + \gamma P_{\pi^{(t)}} A\mathbf{w}^{(t)}\right)\right\|_\infty$. Therefore, by performing max-norm projections, we can make the bound on the theorem as tight as possible.

4 Solving factored MDPs

4.1 Factored MDPs

Our presentation of factored MDPs follows that of [Koller and Parr, 2000]. In a factored MDP, the set of states is described via a set of random variables $\mathbf{X} = \{X_1, \ldots, X_n\}$, where each X_i takes on values in some finite domain $\mathrm{Dom}(X_i)$. A state $\mathbf{x}$ defines a value $x_i \in \mathrm{Dom}(X_i)$ for each variable X_i. We define a state transition model τ using a *dynamic Bayesian network (DBN)* [Dean and Kanazawa, 1989]. Let X_i denote the variable X_i at the current time and X'_i the variable at the next step. The *transition graph* of a DBN is a two-layer directed acyclic graph G_τ whose nodes are $\{X_1, \ldots, X_n, X'_1, \ldots, X'_n\}$. We denote the parents of X'_i in the graph by $\mathit{Parents}_\tau(X'_i)$. For simplicity of exposition, we assume that $\mathit{Parents}_\tau(X'_i) \subseteq \mathbf{X}$; i.e., all arcs in the DBN are between variables in consecutive time slices. (This assumption can be relaxed, but our algorithm becomes somewhat more complex.) Each node X'_i is associated with a *conditional probability distribution (CPD)* $P_\tau(X'_i \mid \mathit{Parents}_\tau(X'_i))$. The transition probability $P_\tau(\mathbf{x}' \mid \mathbf{x})$ is then defined to be $\prod_i P_\tau(x'_i \mid \mathbf{u}_i)$, where $\mathbf{u}_i$ is the value in $\mathbf{x}$ of the variables in $\mathit{Parents}_\tau(X'_i)$.

Consider, for example, the problem of optimizing the behavior of a system administrator maintaining a network of n computers. Each machine is connected to some subset of other machines. In one simple network, we might connect the machines in a ring, with machine i connected to machines $i+1$ and $i-1$. (In this example, we assume addition and subtraction are performed modulo n). Each machine is associated with a binary random variable F_i, representing whether it has failed. The parents of F'_i are F_i, F_{i-1}, F_{i+1}. The CPD of F'_i is such that if $F_i = \mathit{true}$, then $F'_i = \mathit{true}$ with high probability; i.e., failures tend to persist. If $F_i = \mathit{false}$, then F'_i is a *noisy or* of its two other parents; i.e., a failure in either of its neighbors can independently cause machine i to fail.

We can define the transition dynamics of an MDP by defining a separate DBN model $\tau_a = \langle G_a, P_a\rangle$ for each action a. However, in many cases, different actions have very similar transition dynamics, only differing in their effect on some small set of variables. In particular, in many cases a variable has a default evolution model, which only changes if an action affects it directly [Boutilier *et al.*, 1999]. Therefore, as in [Koller and Parr, 2000], we use the notion of a *default transition model* $\tau_d = \langle G_d, P_d\rangle$. For each action a, we define $\mathit{Effects}[a] \subseteq \mathbf{X}'$ to be the variables in the next state whose local probability model is different from τ_d, i.e., those variables X'_i such that $P_a(X'_i \mid \mathit{Parents}_a(X'_i)) \neq P_d(X'_i \mid \mathit{Parents}_d(X'_i))$.

In our system administrator example, we have an action a_i for rebooting each one of the machines, and a default action d for doing nothing. The transition model described above corresponds to the "do nothing" action, which is also the default transition model. The transition model for a_i is different from d only in the transition model for the variable F'_i, which is now $F'_i = \mathit{true}$ with some small fixed probability, regardless of the status of the neighboring machines.

Finally, we need to provide a compact representation of the reward function. We assume that the reward function is factored additively into a set of localized reward functions, each of which only depends on a small set of variables.

Definition 4.1 *A function f is* restricted *to a domain $\mathbf{C} \subseteq \mathbf{X}$ if $f : \mathrm{Dom}(\mathbf{C}) \mapsto \mathbb{R}$. If f is restricted to $\mathbf{Y}$ and $\mathbf{Y} \subset \mathbf{Z}$, we will use $f(\mathbf{z})$ as shorthand for $f(\mathbf{y})$ where $\mathbf{y}$ is the part of the instantiation $\mathbf{z}$ that corresponds to variables in $\mathbf{Y}$.*

Let $R_1, \ldots, R_r$ be a set of functions, where each R_i is restricted to variable cluster $\mathbf{W}_i \subset \{X_1, \ldots, X_n\}$. The reward function for state $\mathbf{x}$ is defined to be $\sum_{i=1}^{r} R_i(\mathbf{x}) \in \mathbb{R}$. In our example, we might have a reward function associated with each machine i, which depends on F_i.

Factorization may allow us to represent large MDPs very compactly. However, we must still address the problem of solving these MDPs. Solution algorithms rely on the ability to represent value functions and policies, the representations which requires the same number of parameters as the size of the space. One might be tempted to believe that factored transition dynamics and rewards would result in a factored value function, which can thereby be represented compactly. Unfortunately, even in factored MDPs, the value function rarely has any internal structure [Koller and Parr, 1999].

Koller and Parr [1999] suggest that there are many domains where our value function might be "close" to structured, i.e., well-approximated using a linear combination of functions each of which refers only to a small number of variables. More precisely, they define a value function to be a *factored (linear) value function* if it is a linear function over the basis $h_1, \ldots, h_k$, where each h_i is restricted to some subset of variables $\mathbf{C}_i$. In our example, we might have basis functions whose domains are pairs of neighboring machines, e.g., one basis function which is an indicator function for each of the four combinations of values for the pair of failure variables.

As shown by Koller and Parr [1999; 2000], factored value functions provide the key to performing efficient computations over the exponential-sized state sets that we have in factored MDPs. The key insight is that restricted domain functions (including our basis functions) allow certain basic operations to be implemented very efficiently. In the remainder of this section, we will show that this key insight also applies in the context of our algorithm.

4.2 Factored Max-norm Projection

The key computational step in both of our algorithms is the solution of Eq. (1) using the linear program in Eq. (2). In our setting, the vectors $\mathbf{b}$ and $C\mathbf{w}$ are vectors in $\mathbb{R}^{|S|}$, where S is our state space. In the case of factored MDPs, our state space is a set of vectors $\mathbf{x}$ which are assignments to the state variables $\mathbf{X} = \{X_1, \ldots, X_n\}$. We can view both $C\mathbf{w}$ and $\mathbf{b}$ as functions of these state variables, and hence also their difference. Thus, we can define a function $F^{\mathbf{w}}(X_1, \ldots, X_n)$ such that $F^{\mathbf{w}}(\mathbf{x}_i) = ((C\mathbf{w})_i - b_i)$. Note that we have executed a representation shift; we are viewing $F^{\mathbf{w}}$ as a function of $\mathbf{X}$, which is parameterized by $\mathbf{w}$.

The size of the state space is exponential in the number of variables. Hence, our goal in this section is to optimize Eq. (1) without explicitly considering each of the exponentially many states. The key is to use the fact that $F^{\mathbf{w}}$ has a factored representation. More precisely, $C\mathbf{w}$ has the form $\sum_j w_j f'_j(\mathbf{Z}_j)$, where $\mathbf{Z}_j$ is a subset of $\mathbf{X}$. For example, we might have $f'_1(X_1, X_2)$ which takes value 1 in states where $X_1 =$ *true* and $X_2 =$ *false* and 0 otherwise. Similarly, the vector $\mathbf{b}$ in our case is also a sum of restricted domain functions. Thus, we can express $F^{\mathbf{w}}$ as a sum $\sum_j f_j^{\mathbf{w}}(\mathbf{Z}_j)$, where $f_j^{\mathbf{w}}$ may or may not depend on $\mathbf{w}$. In the future, we sometimes drop the superscript $\mathbf{w}$ when it is clear from context.

We tackle the problem of a factored solution to the LP in Eq. (2) in two steps.

Maximizing over the state space: First, assume that $\mathbf{b}$ and $C\mathbf{w}$ are given, and that our goal is simply to compute $\max_i((C\mathbf{w})_i - b_i) = \max_{\mathbf{x}} F(\mathbf{x})$, i.e., to find the state $\mathbf{x}$ over which F is maximized. Recall that $F = \sum_{j=1}^{m} f_j(\mathbf{Z}_j)$. We can maximize such a function F using *non-serial dynamic programming* [Bertele and Brioschi, 1972] or *cost networks* [Dechter, 1999]. The idea is virtually identical to variable elimination in a Bayesian network. We review this construction here, as it is a key component in our solution LP.

Our goal is to compute
$$\max_{x_1, \ldots, x_n} \sum_j f_j(\mathbf{Z}_j[\mathbf{x}]),$$
where $\mathbf{Z}_j[\mathbf{x}]$ is the instantiation of the variables in $\mathbf{Z}_j$ in the assignment $\mathbf{x}$. The key idea is, rather than summing all functions and then doing the maximization, we maximize over variables one at a time. When maximizing over x_l, only summands involving x_l participate in the maximization. For example, assume
$$F = f_1(x_1, x_2) + f_2(x_1, x_3) + f_3(x_2, x_4) + f_4(x_3, x_4).$$
We therefore wish to compute:
$$\max_{x_1, x_2, x_3, x_4} f_1(x_1, x_2) + f_2(x_1, x_3) + f_3(x_2, x_4) + f_4(x_3, x_4).$$
We can first compute the maximum over x_4; the functions f_1 and f_2 are irrelevant, so we can push them out. We get
$$\max_{x_1, x_2, x_3} f_1(x_1, x_2) + f_2(x_1, x_3) + \max_{x_4}[f_3(x_2, x_4) + f_4(x_3, x_4)].$$
The result of the internal maximization depends on the values of x_2, x_3; i.e., we can introduce a new function $e_1(X_2, X_3)$ whose value at the point x_2, x_3 is the value of the internal $\max$ expression. Our problem now reduces to computing
$$\max_{x_1, x_2, x_3} f_1(x_1, x_2) + f_2(x_1, x_3) + e_1(x_2, x_3),$$
having one fewer variable. Next, we eliminate another variable, say X_3, with the resulting expression reducing to:
$$\max_{x_1, x_2} f_1(x_1, x_2) + e_2(x_1, x_2),$$
$$\text{where} \quad e_2(x_1, x_2) = \max_{x_3}[f_2(x_1, x_3) + e_1(x_2, x_3)].$$
Finally, we define
$$e_3 = \max_{x_1, x_2} f_1(x_1, x_2) + e_2(x_1, x_2)$$
The result at this point is a number, which is the desired maximum over $x_1, \ldots, x_4$.

In general, the variable elimination algorithm maintains a set $\mathcal{F}$ of functions, which initially $\{f_1, \ldots, f_m\}$. The algorithm then repeats the following steps:

1. Select an uneliminated variable X_l;
2. Take all $e_1, \ldots, e_L \in \mathcal{F}$ whose domain contains X_l.
3. Define a new function $e = \max_{x_l} \sum_j e_j$ and introduce it into $\mathcal{F}$. The domain of e is $\cup_{j=1}^{L} \mathrm{Dom}[e_j] - \{X_l\}$.

The computational cost of this algorithm is linear in the number of new "function values" introduced in the elimination process. More precisely, consider the computation of a new function e whose domain is $\mathbf{Z}$. To compute this function, we need to compute $|\mathrm{Dom}[\mathbf{Z}]|$ different values. The cost of the algorithm is linear in the overall number of these values, introduced throughout the algorithm. As shown in [Dechter, 1999], this cost is exponential in the induced width of the undirected graph defined over the variables $X_1, \ldots, X_n$, with an edge between X_l and X_m if they appear together in one of the original functions f_j.

Factored LP: Now, consider our original problem of minimizing $\|C\mathbf{w} - \mathbf{b}\|_\infty = \max_{\mathbf{x}} |F^{\mathbf{w}}(\mathbf{x})|$ over $\mathbf{w}$. As in Eq. (2), we want to construct a linear program which performs this optimization. However, we want a compact LP, that avoids an explicit enumeration of the constraints for the exponentially many states. The first key insight is that we can replace the entire set of constraints — $\phi \geq (C\mathbf{w})_i - b_i$ for all states i — by the equivalent constraint $\phi \geq \max_i((C\mathbf{w})_i - b_i)$, or equivalently, $\phi \geq \max_{\mathbf{x}} F^{\mathbf{w}}(\mathbf{x})$. The second key insight is that this new constraint can be implemented using a construction that follows the structure of variable elimination in cost networks. (An identical construction applies to the complementary constraints: $\phi \geq b_i - (C\mathbf{w})_i$.)

Consider any function e used within $\mathcal{F}$ (including the original f_i's), and let $\mathbf{Z}$ be its domain. For any assignment $\mathbf{z}$ to $\mathbf{Z}$, we introduce a variable into the linear program whose value represents $u^e_{\mathbf{z}}$. For the initial functions $f^{\mathbf{w}}_i$, we include the constraint that $u^{f_i}_{\mathbf{z}} = f^{\mathbf{w}}_i(\mathbf{z})$. As $f^{\mathbf{w}}_i$ is linear in $\mathbf{w}$, this constraint is linear in the LP variables. Now, consider a new function e introduced into $\mathcal{F}$ by eliminating a variable X_l. Let $e_1, \ldots, e_L$ be the functions extracted from $\mathcal{F}$, and let $\mathbf{Z}$ be the domain of the resulting e. We introduce a set of constraints:

$$u^e_{\mathbf{z}} \geq \sum_{j=1}^{L} u^{e_j}_{(\mathbf{z},x_l)[\mathbf{Z}_j]} \quad \forall x_l,$$

where $\mathbf{Z}_j$ is the domain of e_j and $(\mathbf{z}, x_l)[\mathbf{Z}_j]$ denotes the value of the instantiation $(\mathbf{z}, x_l)$ restricted to $\mathbf{Z}_j$. Let e_n be the last function generated in the elimination, and recall that its domain is empty. Hence, we have only a single variable u^{e_n}. We introduce the additional constraint $\phi \geq u^{e_n}$.

To understand this construction, consider our simple example above, and assume we want to express the fact that $\phi \geq \max_{\mathbf{x}} F^{\mathbf{w}}(\mathbf{x})$. We first introduce a set of variables $u^{f_1}_{x_1,x_2}$ for every instantiation of values x_1, x_2 to the variables X_1, X_2. Thus, if X_1 and X_2 are both binary, we would have four such variables. We would then introduce a constraint defining the value of $u^{f_1}_{x_1,x_2}$ appropriately. For example, for our f_1 above, we would have $u^{f_1}_{t,t} = 0$ and $u^{f_1}_{t,f} = w_1$. We would have similar variables and constraints for each f_j and each value $\mathbf{z}$ in $\mathbf{Z}_j$. Note that each of the constraints is a simple equality constraint involving numerical constants and perhaps the weight variables $\mathbf{w}$.

Next, we introduce variables for each of the intermediate expressions. For example, we would have a set of variables $u^{e_1}_{x_2,x_3}$; for each of them, we would have a set of constraints

$$u^{e_1}_{x_2,x_3} \geq u^{f_3}_{x_2,x_4} + u^{f_4}_{x_3,x_4}$$

one for each value x_4 of X_4. We would have a similar set of constraint for $u^{e_2}_{x_1,x_2}$ in terms of $u^{f_2}_{x_1,x_3}$ and $u^{e_1}_{x_2,x_3}$. Note that each constraint is a simple linear inequality.

It is easy to show that minimizing ϕ "drives down" the value of each variable $u^e_{\mathbf{z}}$, so that

$$u^e_{\mathbf{z}} = \max_{x_i} \sum_{j=1}^{\ell} u^{e_j}_{(\mathbf{z},x_i)[\mathbf{Z}_j]}.$$

We can then prove, by induction, that u^{e_n} must be equal to $\max_{\mathbf{x}} \sum_j f^{\mathbf{w}}_j(\mathbf{x})$. Our constraints on ϕ ensure that it is greater than this value, which is the maximum of $\sum_j f^{\mathbf{w}}_j(\mathbf{x})$ over the entire state space. The LP, subject to those constraints, will minimize ϕ, guaranteeing that we find the vector $\mathbf{w}$ that achieves the lowest value for this expression.

Returning to our original formulation, we have that $\sum_j f^{\mathbf{w}}_j$ is $C\mathbf{w} - \mathbf{b}$ in one set of constraints and $\mathbf{b} - C\mathbf{w}$ in the other. Hence our new set of constraints is equivalent to the original set: $\phi \geq C\mathbf{w} - \mathbf{b}$ and $\phi \geq \mathbf{b} - C\mathbf{w}$. Minimizing ϕ finds the $\mathbf{w}$ that minimizes the $\mathcal{L}_\infty$ norm, as required.

4.3 Factored solution algorithms

The factored max-norm projection algorithm described previously is the key to applying our max-norm solution algorithms in the context of factored MDPs.

Approximate value iteration: Let us begin by considering the value iteration algorithm. As described above, the algorithm repeatedly applies two steps. It first applies the Bellman operator to obtain $\overline{\mathcal{V}}^{(t+1)} = \mathcal{T}^* A\mathbf{w}^{(t)}$. Let $\pi^{(t)}$ be the stationary policy $\pi^{(t)} = \mathit{Greedy}(A\mathbf{w}^{(t)})$. Note that $\pi^{(t)}$ corresponds to the maximizing policy used in the Bellman operator at iteration t, i.e., $\mathcal{T}_{\pi^{(t)}} = \mathcal{T}^*$ for iteration t. Thus, we can compute $\overline{\mathcal{V}}^{(t+1)} = \mathcal{T}^* A\mathbf{w}^{(t)}$ by first computing $\pi^{(t)}$ and then performing a *backprojection* operation for this fixed policy, i.e., $\overline{\mathcal{V}}^{(t+1)} = R_{\pi^{(t)}} + \gamma P_{\pi^{(t)}} A\mathbf{w}^{(t)}$. Assume, for the moment, that $P_{\pi^{(t)}}$ is a factored transition model; we discuss the computation of the greedy policy and the resulting transition model below. As discussed by Koller and Parr [1999], the backprojection operation can be performed efficiently if the transition model and the value function are both factored appropriately. Furthermore, the resulting function $\overline{\mathcal{V}}^{(t+1)}$ is also factored, although the factors involve larger domains.

To recap their construction briefly, let f be a restricted domain function with domain $\mathbf{Y}$; our goal is to compute $P_\tau f$. We define the *back-projection of* $\mathbf{Y}$ *through* τ as the set of parents of $\mathbf{Y}'$ in the transition graph G_τ; $\Gamma_\tau(\mathbf{Y}') = \cup_{Y' \in \mathbf{Y}'} \mathit{Parents}_\tau(Y')$. It is easy to show that: $(P_\tau f)(\mathbf{x}) = \sum_{\mathbf{y}'} P_\tau(\mathbf{y}' \mid \mathbf{z}) f(\mathbf{y}')$, where $\mathbf{z}$ is the value of $\Gamma_\tau(\mathbf{Y})$ in $\mathbf{x}$. Thus, we see that $(P_\tau f)$ is a function whose domain is restricted to $\Gamma_\tau(\mathbf{Y})$. Note that the cost of the computation depends linearly on $|\mathrm{Dom}(\Gamma_\tau(\mathbf{Y}))|$, which depends on $\mathbf{Y}$ (the domain of f) and on the complexity of the process dynamics.

Therefore, $\overline{\mathcal{V}}^{(t+1)}$ is composed of the sum of two factored functions: the reward function $R_{\pi^{(t)}}$ which is assumed to be factored, and the backprojected basis functions $P_{\pi^{(t)}} A$ which are also factored, as we have just shown. Finally, the second step in approximate value iteration is to compute the projection $A\mathbf{w}^{(t+1)} = \Pi_\infty \overline{\mathcal{V}}^{(t+1)}$, i.e., find $\mathbf{w}^{(t+1)}$ that minimizes $\left\| A\mathbf{w}^{(t+1)} - \overline{\mathcal{V}}^{(t+1)} \right\|_\infty$. As both $\overline{\mathcal{V}}^{(t+1)}$ and $A\mathbf{w}^{(t+1)}$ are factored, we can perform this computation using the factored LP discussed in the previous section.

Approximate policy iteration: Policy iteration also iterates through two steps. The policy improvement step simply computes the greedy policy relative to $\pi^{(t)}$. We discuss this step below. The approximate value determination step computes:

$$\arg\min_{\mathbf{w}} \left\| (A - \gamma P_{\pi^{(t)}} A)\, \mathbf{w}^{(t)} - R_{\pi^{(t)}} \right\|_\infty .$$

Again, assuming that $P_{\pi^{(t)}}$ is factored, we can conclude that $C = (A - \gamma P_{\pi^{(t)}} A)$ is also a matrix whose columns correspond to restricted-domain functions. The target $\mathbf{b} = R_{\pi^{(t)}}$ corresponds to the reward function, which is assumed to be factored. Thus, we can again apply our factored LP.

In our discussion so far, we assumed that we have some mechanism for computing the greedy policy *Greedy*$(A\mathbf{w}^{(t)})$, and that this policy has a compact representation and a factored transition model. As shown in [Koller and Parr, 2000], the greedy policy relative to a factored value function has the form of a *decision list*. More precisely, the policy can be written in the form $\langle \mathbf{t}_1, a_1\rangle, \langle \mathbf{t}_2, a_2\rangle, \ldots, \langle \mathbf{t}_L, a_L\rangle$, where each $\mathbf{t}_i$ is an assignment of values to some small subset $\mathbf{T}_i$ of variables, and each a_i is an action. The optimal action to take in state $\mathbf{x}$ is the action a_j corresponding to the first event $\mathbf{t}_j$ in the list with which $\mathbf{x}$ is consistent.

Koller and Parr show the greedy policy can be represented compactly using such a decision list, and provide an efficient algorithm for computing it. Unfortunately, as they discuss, the resulting transition model is usually not factored. Thus, we cannot simply apply our factored LP construction, as suggested above. However, we can adapt it to deal with this issue.

The basic idea is to introduce cost networks corresponding to each branch in the decision list. Let S_i be the set of states $\mathbf{x}$ for which $\mathbf{t}_i$ is the first event in the decision list for which $\mathbf{x}$ is consistent. Recall that our LP construction defines a set of constraints that imply that $\phi \geq ((C\mathbf{w})_i - b_i)$ for each state i. Instead, we will have a separate set of constraints for the states in each subset S_i. For each state in S_i, we know that action a_i is taken. Hence, we can apply our construction above using P_{a_i} — a transition model which is factored by assumption — in place of the non-factored $P_{\pi^{(t)}}$.

The only issue is to guarantee that the cost network constraints derived from this transition model are applied only to states in S_i. Specifically, we must guarantee that they are applied only to states consistent with $\mathbf{t}_i$, but not to states that are consistent with some $\mathbf{t}_j$ for $j < i$. To guarantee the first condition, we simply instantiate the variables in $\mathbf{T}_i$ to take the values specified in $\mathbf{t}_i$. That is, our cost network now considers only the variables in $\{X_1, \ldots, X_n\} - \mathbf{T}_i$, and computes the maximum only over the states consistent with $\mathbf{T}_i = \mathbf{t}_i$. To guarantee the second condition, we ensure that we do not impose any constraints on states associated with previous decisions. This is achieved by adding indicators $\mathcal{I}_j$ for each previous decision $\mathbf{t}_j$, with weight $-\infty$. These will cause the constraints associated with $\mathbf{t}_i$ to be trivially satisfied by states in S_j for $j < i$. Note that each of these indicators is a restricted domain function of $\mathbf{T}_j$ and can be handled in the same fashion as all other terms in the factored LP. Thus, for a decision list of size L, our factored LP contains constraints from $2L$ cost networks.

It is instructive to compare our max-norm policy iteration algorithm with $\mathcal{L}_2$-projection policy iteration algorithm of Koller and Parr [2000] in terms of computational costs per iteration and implementation complexity. Computing the $\mathcal{L}_2$ projection requires (among other things) a series of dot product operations between basis functions and backprojected basis functions $\langle h_i \bullet P_\pi h_j\rangle$. These expressions are easy to compute if P_π refers to the transition model of a particular action a. However, if the policy π is represented as a decision list, as is the result of the policy improvement step, then this step becomes much more complicated. In particular, for every point in the decision list, for every pair of basis functions i and j, and for each assignment to the variables in $\mathrm{Dom}(h_i) \cup \mathrm{Dom}(P_a h_j)$, it requires the solution of a counting problem which is $\sharp P$-complete in general. Although, Koller and Parr show that this computation can be performed using a Bayesian Network (BN) inference, the algorithm still requires a BN inference for each one of those assignments at each point of the decision list. This makes the algorithm very difficult to implement efficiently in practice.

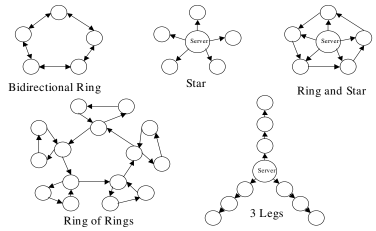


Figure 1: Network topologies tested.

The max-norm projection, on the other hand, relies on solving a linear program at every iteration. The size of the linear program depends on the cost networks generated. As we discuss, two cost networks are needed for each point in the decision list. The complexity of each of these cost networks is approximately the same as one of the BN inferences in the counting problem for the $\mathcal{L}_2$ projection. Overall, for each point in the decision list, we have a total of two of these "inferences", as opposed to one for each assignment of $\mathrm{Dom}(h_i) \cup \mathrm{Dom}(P_a h_j)$ for every pair of basis functions i and j. Thus, the max-norm policy iteration algorithm is substantially less complex computationally than the approach based on $\mathcal{L}_2$-projection. Furthermore, the use of linear programming allows us to rely on existing LP packages (such as CPLEX), which are very highly optimized.

5 Experimental Results

The factored representation of a value function is most appropriate in certain types of systems: Systems that involve many variables, but where the strong interactions between the variables are fairly sparse, so that the decoupling of the influence between variables does not induce an unacceptable loss in accuracy. As argued by Herbert Simon [1981] in "Architecture of Complexity", many complex systems have a "nearly decomposable, hierarchic structure", with the subsystems of such systems interacting only weakly between them. We selected to try our algorithm on a problem that we believe characterizes this type of structure.

The problem relates to a system administrator who has to maintain a network of computers; we experimented with various network architectures, shown in Fig. 1. Machines fail randomly, and a faulty machine increases the probability that its neighboring machines will fail. At every time step, the SysAdmin can go to one machine and reboot it, causing it to be working in the next time step with high probability. Each machine receives a reward of 1 when working (except in the ring, where one machine receives a reward of 2, to introduce

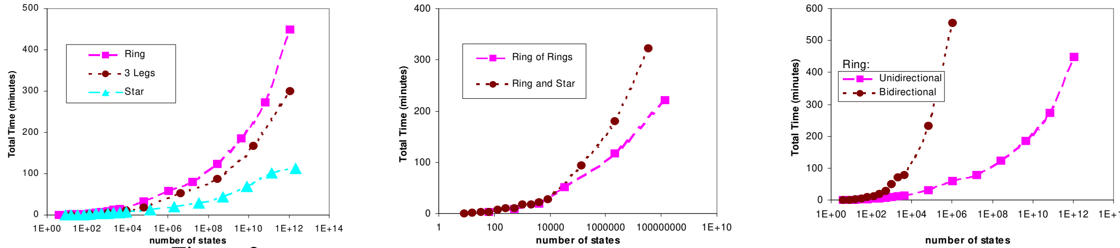
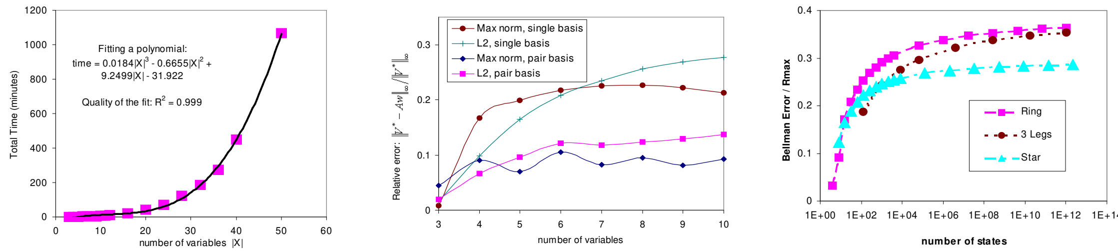


Figure 2: Running times for policy iteration on variants of the SysAdmin problem.



(a) **(b)** **(c)**

Figure 3: (a) Fitting a polynomial to the running time of the unidirectional ring; (b) Relative error to optimal value function $\mathcal{V}^*$ and comparison to $\mathcal{L}_2$ projection; (c) For large models, measuring Bellman error after convergence.

some asymmetry), a zero reward is given to faulty machines, and the discount factor is $\gamma = 0.95$. Note that the additive structure of the reward function makes it unsuitable for the tree-structured representation used by Boutilier *et al.* [1999].

The basis functions included independent indicators for each machine, with value 1 if it is working and zero otherwise (each, a restricted domain function of a single variable), and the constant basis, whose value is 1 for all states.

We implemented the factored policy and value iteration algorithms in Matlab, using CPLEX as the LP solver. In our experiments, most of the time was spent in Matlab generating the LP constraints; CPLEX is a remarkably efficient and reliable solver, allowing even large LPs to be solved very quickly. We present only results for policy iteration. The time per iteration was about equal for policy and value iteration, but policy iteration converged in many fewer iterations, only about 4 or 5 iterations in the models we tested.

To evaluate the complexity of the algorithm, tests were performed with increasing number of states, that is, increasing number of machines on the network. Fig. 2 shows the running time for increasing problem sizes, for various architectures. The simplest one is the "Star", where the backprojection of each basis function has domain restricted to two variables and the largest factor in the cost network has domain restricted to two variables. The most difficult one was the "Bidirectional Ring", where factors contain five variables.

Note, the number of states is growing exponentially (indicated by the log scale in Fig. 2), but running times increase only logarithmically in the number of states, i.e., polynomially in the number of variables. This is illustrated by Fig. 3(a), where we fit a 3rd order polynomial to the running times for the "unidirectional ring". Note, the problem size grows quadratically with the number of variables: adding a machine to the network also adds the possible action of fixing that machine. For this problem, the computation cost of our algorithm empirically grows approximately as $O\left((|\mathbf{X}| \cdot |A|)^{1.5}\right)$.

For further evaluation, we measured the error in our approximate value function relative to the true optimal value function $\mathcal{V}^*$. Note that it is only possible to compute $\mathcal{V}^*$ for small problems; in our case, we were only able to go up to 10 machines. For comparison, we also evaluated the error in the approximate value function produced by the $\mathcal{L}_2$-projection algorithm of Koller and Parr. As we discussed above, the $\mathcal{L}_2$ projections in factored MDPs by Koller and Parr [2000] are difficult and time consuming; hence, we were only able to compare the two algorithms for smaller problems, where an equivalent $\mathcal{L}_2$-projection can be implemented using an explicit state space formulation. Results for both algorithms are presented in Fig. 3(b), showing the relative error of the approximate solutions to the true value function for increasing problem sizes. The results indicate that, for larger problems, the max-norm formulation generates a better approximation of the true optimal value function $\mathcal{V}^*$ than the $\mathcal{L}_2$-projection. Here, we used two types of basis functions: the same single variable basis, and pairwise basis, which also includes indicators for neighboring pairs of machines. As expected, pairwise basis generated better approximations.

For these small problems, we can also compare the actual value of the policy generated by our algorithm to the value of the optimal policy. Here, the value of the policy generated by our algorithm is much closer to the value of the optimal policy than the error implied by the difference between our approximate value function and $\mathcal{V}^*$. For example, for the "Star" architecture with one server and up to 6 clients, our approximation with single variable basis functions had relative error of 12%, but the policy we generated had the same value as the optimal policy. In this case, the same was true for the policy generated by the $\mathcal{L}_2$ projection. In an "Unidirectional Ring" with 8 machines and pairwise basis, the relative error between our approximation and $\mathcal{V}^*$ was about 10%, but the resulting policy only had a 6% loss over the optimal policy. For the same problem, the $\mathcal{L}_2$ approximation has a value function error of 12%, and a true policy loss was 9%. In other words, both methods induce policies that have lower errors than the errors in the approximate value function (at least for small problems). However, our algorithm continues to outperform the $\mathcal{L}_2$ algorithm, even relative to actual policy loss.

For the large models, we can no longer compute the correct value function, so we cannot evaluate our results by computing $\|\mathcal{V}^* - Aw\|_\infty$. However, the *Bellman error*, defined as $\mathrm{BellmanErr}(\mathcal{V}) = \|\mathcal{T}^*\mathcal{V} - \mathcal{V}\|_\infty$, can be used to provide a bound: $\|\mathcal{V}^* - Aw\|_\infty \leq \frac{\mathrm{BellmanErr}(Aw)}{1-\gamma}$ [Williams and Baird, 1993]. Thus, we use the Bellman error to evaluate our answers for larger models. Fig. 3(c) shows that the Bellman error increases very slowly with the number of states.[1]

Finally, we can look at the actual policies generated in our experiments. First, we noted that these tended to be short, the length of the final decision list policy grew approximately linearly with the number of machines. Furthermore, the policy itself is often fairly intuitive. In the "Ring and Star" architecture, for example, the decision list says: If the server is faulty, fix the server; else, if another machine is faulty, fix it.

6 Conclusions

In this paper, we presented new algorithms for approximate value and policy iteration. Unlike previous approaches, our algorithms directly minimize the $\mathcal{L}_\infty$ error, and therefore have better theoretical performance guarantees than algorithms that optimize the $\mathcal{L}_2$-norm.

We have shown that the $\mathcal{L}_\infty$ error can be minimized using linear programming, and provided an approach for representing this LP compactly for factored MDPs, allowing these algorithms to be applied efficiently even for MDPs with exponentially large state spaces. Our algorithms are more efficient and substantially easier to implement than previous algorithms based on the $\mathcal{L}_2$-projection.

We have presented results on a structured problem, representing a simplified version of a maintenance task. Our results show that our methods scale effectively to very large factored MDPs, and that our approach can exploit problem-specific structure to efficiently generate approximate solutions to complex problems.

The success of our algorithm depends on our ability to capture the most important structure in the value function using a linear factored approximation. This ability, in turn, depends on the choice of the basis functions and on the properties of the domain. The algorithm currently requires the designer to specify the factored basis functions. This is a limitation compared to other algorithms (e.g., [Dearden and Boutilier, 1997]), which are fully automated. However, our experiments indicate that a few simple rules are often quite successful in designing a basis. First, we ensure that the reward function is representable by our basis. A simple basis that, in addition, contains a separate set of indicators for each variable is often successful. We can also add indicators over pairs of variables; most simply, we can choose these according to the DBN transition model, where an indicator is added between variables X_i and each one of the variables in *Parents*(X_i), thus representing one-step influences. This procedure can be extended, adding more basis to represent more influences as required. Thus, the structure of the DBN gives us indications of how to choose the basis functions. Other sources of prior knowledge can also be included for further specifying the basis.

The quality of our approximation also depends strongly on the structure of the domain. As we discussed above, our approximation is most successful in systems where variables are tightly coupled to only a small number of other variables. We believe that, for systems of this type, an approximation as a factored linear value function can be very accurate.

There are many possible extensions to this work. In particular, we hope to extend our algorithms to utilize other types of structure in the representation. One interesting direction involves factored action models, where multiple actions are taken simultaneously. Another involves the use of asymmetries (context-specificity) in the value function, as in the work of Dearden and Boutilier [1997], providing a complementary source of structure to the factorization used in our work.

Acknowledgments: We are very grateful to Dirk Ormoneit and Uri Lerner for many useful discussions. This work was supported by the ONR under the MURI program "Decision Making Under Uncertainty" and by the Sloan Foundation.

References

[Bellman *et al.*, 1963] R. Bellman, R. Kalaba, and B. Kotkin. Polynomial approximation – a new computational technique in dynamic programming. *Math. Comp.*, 17(8):155–161, 1963.

[Bertele and Brioschi, 1972] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, New York, 1972.

[Boutilier *et al.*, 1999] C. Boutilier, T. Dean, and S. Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 1999.

[Cheney, 1982] E. W. Cheney. *Approximation Theory*. Chelsea Publishing Co., New York, NY, 2nd edition, 1982.

[Dean and Kanazawa, 1989] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[Dearden and Boutilier, 1997] R. Dearden and C. Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 89(1):219–283, 1997.

[Dechter, 1999] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artif. Intel.*, 113(1–2):41–85, 1999.

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

[Koller and Parr, 2000] D. Koller and R. Parr. Policy iteration for factored mdps. In *Proc. of Uncertainty in AI (UAI-00)*, 2000.

[Puterman, 1994] M. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. Wiley, New York, 1994.

[Simon, 1981] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, Massachusetts, second edition, 1981.

[Stiefel, 1960] E. Stiefel. Note on Jordan elimination, linear programming and Tchebycheff approximation. *Numerische Mathematik*, 2:1 – 17, 1960.

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

[Williams and Baird, 1993] R. Williams and L. Baird. Tight performance bounds on greedy policies based on imperfect value functions. Tech. Report, Northeastern Univ., Massachusetts, 1993.

[1] Note that computing the Bellman error involves a maximization over the state space. Thus, the complexity of this computation grows exponentially with the number of variables. However, this maximization can be performed by a cost network using the same construction as in our max-norm projection.