Linear Algebra in Very High-Dimension Vector Spaces: Algorithms and Data Structures for Implementing Exact and Approximate Solution Methods

Kee-Eung Kim Thomas Dean and Samuel E. Hazlehurst kek,tld,seh,@cs.brown.edu

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-00-02
March 2000

Linear Algebra in Very High-Dimension Vector Spaces: Algorithms and Data Structures for Implementing Exact and Approximate Solution Methods

Kee-Eung Kim Thomas Dean Samuel E. Hazlehurst
Department of Computer Science, Brown University
Providence, RI 02912-1910, U.S.A.
{kek, tld, seh}@cs.brown.edu

Abstract

In this paper, we show how certain problems can be recast in terms of equations involving matrices and vectors that are represented *implicitly*. These matrices and vectors have large explicit representations in the sense that it is practically impossible to even enumerate their indices. Instead, the indices are expressed using a set of index variables that describe the distinguishing features of the problem domain. The matrices and vectors are represented in factored form using compact, easily computed functions of the index variables. The entries of these matrices and vectors are reconstituted on an as-needed basis, often in such a way that computations can be performed on large sets of indices with no more effort than that required for a single index and its corresponding entry. We show how elementary vector and matrix operations can be performed on these implicitly represented matrices and vectors (*e.g.*, scalar and dot products, raising a matrix to a power) and how these basic operations can then be combined to perform more complex calculations (*e.g.*, solving a system of equations). Our motivating examples involve solving Markov decision processes and planning under uncertainty, but the methods described in this paper apply to a wide range of problems.

1 Introduction

There is a large class of numerical optimization problems that can be described in terms of equations involving vectors and matrices. One example that we use throughout this short overview is the problem of finding an optimal or near-optimal policy for a Markov decision problem (MDP). MDPs constitute a stochastic generalization of the deterministic propositional planning problems studied in artificial intelligence [Fikes and Nilsson, 1971]. The objective functions and solution methods for MDPs are often characterized in terms of matrix-vector equations [Puterman, 1994].

Consider calculating $\vec{v}$ defined as

$$\vec{v} = A\vec{u} \tag{1}$$

in which A is an $n \times n$ matrix and $\vec{u}$ and $\vec{v}$ are vectors of length n. The matrix A might be the adjacency matrix for a graph representing a database relation or the state-transition matrix for a planning problem and $\vec{u}$ might be the specification of vertices corresponding to a query or the initial-state distribution. What if (as is often the case in problems of interest to artificial intelligence) $n = k^m$ for some m and $k \geq 2$? Well, for one thing, if m is large, we probably can't represent Equation 1 explicitly by allocating space for each entry of A, $\vec{u}$, and $\vec{v}$. But what if A, $\vec{u}$, and $\vec{v}$ are "symmetrical" or "regular" in some sense? For example, sparse matrix representations exploit certain types of symmetries for problems in which $O(n)$ is acceptable (and hence m is relatively small) but $O(n^2)$ is not. In this paper, we consider a different class of regularities for problems in which m is large and $O(n)$ time or space is out of the question.

In many of the optimization problems encountered in artificial intelligence, including the problem of finding optimal policies for MDPs, the matrices and vectors can be quite large. In particular, it is often the case that the number of indices (and hence the size of a matrix, say, if represented as simple table) is exponential in the size of the problem description.

For example, in the case of factored MDPs [Boutilier *et al.*, 1999], if we were to allocate space for each entry, the sizes of the state-transition matrix and the reward vector would be exponential in the number of state variables used to describe the domain. In many cases, however, there are representations for these large matrices and vectors that allow us to encode these objects in a compact and tractable form.

Let's suppose that our matrices and vectors correspond to functions and relations on a domain Ω. Suppose further that we can describe each element of Ω in terms of a set of m features of the domain denoted $Z_1, \ldots, Z_m$ where each feature Z_i can take on values from Ω_{Z_i} and $\Omega = \prod_{i=1}^{m} \Omega_{Z_i}$.

Let $\vec{z} = (z_1, \ldots, z_m)$ be an element of Ω and let $I(\vec{z})$ be the integer index for $\vec{z}$. We refer to $\{Z_1, \ldots, Z_m\}$ as the set of *index variables* for Ω.

Consider representing a function $f : \Omega \times \Omega \rightarrow \Re$ as a matrix $A = \{a_{ij}\}$ and suppose that there exists a partition P of $\Omega \times \Omega$ such that for each block $B \in P$ there exists $a_B \in \Re$ such that for all $(\vec{z}, \vec{z}') \in B$ if $I(\vec{z}) = i$ and $I(\vec{z}') = j$ then $a_{ij} = f(\vec{z}, \vec{z}') = a_B$.

If $|P|$, the number of blocks in P, is small, say polynomial in m, and it is possible to represent each block as a compact, easily computed function of the $\{Z_i\}$, then at least we can represent A compactly and compute the value of any a_{ij} efficiently. This is the sort of regularity that we are concerned with in this paper. In particular, we're interested in answering (at least partially) the following questions:

- How would this sort of regularity help in terms of writing down A, $\vec{u}$, and $\vec{v}$ and in terms of calculating $A\vec{u}$ or solving for $\vec{u}$ in $A\vec{u} = \vec{v}$ given A and $\vec{v}$?
- Are there interesting cases of matrices and vectors for which small partitions of the sort described above exist and such that can we find them (*e.g.*, elicit the compact representations and requisite numbers from experts) and exploit their regularities?
- If we can carry out calculations like $A\vec{u}$, why not $A^T\vec{u}$ (where T denotes matrix transpose), and raising A to a power, *etc.*?

Our initial motivation for this work was to extend iterative methods based on matrix and vector operations to solve large Markov decision processes. In the following, we address the general problem of defining elementary matrix and vector operations involving very high dimensional vector spaces, but we draw on examples that arise in solving Markov decision processes with very large, high-dimensional state spaces.[1]

2 Exploiting Structural Regularities

Consider representing a function $f : \Omega \to \Re$ as a vector $\vec{u}$ and assume that there exists a partition P of $\Omega = (\Omega_{Z_1}, \ldots, \Omega_{Z_m})$ such that for each block (*i.e.*, equivalence class) $B \in P$

$$\exists u_B \in \Re, \forall \vec{z} \in B, \text{ if } I(\vec{z}) = i \text{ then } u_i = f(\vec{z}) = u_B \tag{2}$$

where we denote by u_B the value common to any index $I(\vec{z})$ such that $\vec{z} \in B$. Assume further that for the matrix $A = \{a_{ij}\}$ which represents a function $g : \Omega \times \Omega \to \Re$ and each pair of blocks $(B_1, B_2) \in P \times P$

$$\begin{aligned} &\exists a_{B_1B_2} \in \Re, \forall \vec{z_1} \in B_1, \forall \vec{z_2} \in B_2, \\ &\text{if } I(\vec{z_1}) = i \text{ and } I(\vec{z_2}) = j \text{ then } a_{ij} = g(\vec{z_1}, \vec{z_2}) = a_{B_1B_2} \end{aligned} \tag{3}$$

where we denote by $a_{B_1B_2}$ the value common to any pair of indices $(I(\vec{z_1}), I(\vec{z_2}))$ such that $(\vec{z_1}, \vec{z_2}) \in (B_1, B_2)$. Under these assumptions we can compute $\vec{v} = A\vec{u}$ as follows

$$\forall B \in P, v_B = \sum_{B' \in P} |B'| a_{BB'} u_{B'}$$

so that $v_i = v_B$ for all $i = I(\vec{z})$ such that $\vec{z} \in B$. The running time in this case is $O(|P|^2)$ which is acceptable if $|P|$ is small. In some sense, the size of the partition provides a bound on the intrinsic complexity of the problem defined in the domain.

It will turn out that other elementary matrix-vector operations, *e.g.*, scalar and dot products, sums and differences, can also be performed in time polynomial in $|P|$. In some cases, we can even achieve polynomial performance for more complicated operations. For example, we can calculate $A^k\vec{u}$ given A, $\vec{u}$, and an integer k in $O(k|P|^2)$ time. We can also solve for $\vec{u}$ in $A\vec{u} = \vec{v}$ given A and $\vec{v}$ using an iterative scheme in which each iteration takes $O(|P|^2)$ and convergence is independent of n. The trick is that we need to carve up the domain finely enough to ensure that the operands, A and $\vec{u}$ in this case, are uniform on P (in the sense of satisfying Equation 2 and Equation 3). If A satisfies Equation 3 for the partition P', and $\vec{v}$ satisfies Equation 2 for a different partition P'', then, in the worst case, the coarsest uniform partition (in the sense of being the partition with the fewest blocks that satisfies both Equation 2 and Equation 3) will be of size $O(|P'||P''|)$.

Unfortunately, not all vector equations can be so easily computed. For example, given our previously stated initial motivation for this work, an equation that is of particular interest

[1] By "very large, high-dimensional state spaces" we mean state spaces whose size is exponential in m; by "very high-dimension vector spaces" we mean vector spaces with a number of dimensions exponential in m.

to us is the vector version of the Bellman equation for a Markov decision process

$$\vec{v} = \max_{\alpha} \{\vec{r}_\alpha + \gamma A_\alpha \vec{v}\} \tag{4}$$

in which $\vec{v}$ is the vector representing the optimal value function, γ is a scalar-valued discount rate, $\vec{r}_\alpha$ is the vector representing the reward function with respect to action α, A_α is the state-transition probability matrix with respect to action α, and the max over actions is on a state-by-state basis (or block-by-block basis in our case) [Puterman, 1994]. In the worst-case, solving this system of equations using the methods described above (essentially generalizing on the work of [Boutilier *et al.*, 1995] and [Dean and Givan, 1997]) takes time proportional to the product of the sizes of the representations for $\vec{r}_\alpha$ and A_α, *i.e.*, exponential in the number of actions.

Simplifying somewhat, if we have an equation involving h distinct terms (matrices and vectors) each of which satisfies some variant of Equation 2 or Equation 3 for some partition P_i, then, in the worst case, solving the equation will require $O(\prod_{i=1}^{h} |P_i|)$. To avoid this potential blowup, we can take advantage of cases where the P_i agree on blocks or perform approximations that involve combining blocks in partitions whose associated values are approximately the same and then doing additional bookkeeping to keep careful track of the resulting errors. One particular technique in the context of Markov decision processes can be found in [Boutilier and Dearden, 1996].

In the remainder of this paper, we describe procedures implementing elementary operations on matrices and vectors, applications using these elementary operations to solve problems posed in terms of linear algebra, and the use of a machine learning technique (tree pruning) to avoid potential blowups by trading space for accuracy.

3 Representing Very Large Matrices and Vectors

To illustrate our main points, we borrow the representation for a toy robot planning problem presented in [Boutilier *et al.*, 1995]. Let R, U, W, HC, WC denote our index variables (which we assume to be boolean in order to keep the examples simple) representing, respectively, the weather outside being rainy, the robot having an umbrella, the robot being wet, the robot holding coffee, and the robot's boss wanting coffee. The details of this problem are not critical for this paper but it will be somewhat easier to follow the examples with these distinctive variable names. Let $Z = (R, U, W, HC, WC)$ denote a vector of variables ranging $\Omega = \{0, 1\}^5$.

In this paper, we use decision trees as our basic data structure for encoding functions, *i.e.*, matrices and vectors. There are other methods for compactly representing functions, *e.g.*, rule-based methods that are often used in artificial intelligence to achieve compact representations [Boutilier *et al.*, 1999], and generalized versions of binary decision diagrams that are used in a number of applications including VLSI design [Hoey *et al.*, 1999].

As an example, we can represent the vector $\vec{v} = (v_1, \ldots, v_{32})$ corresponding to the reward function for a particular action in the robot planning problem as shown in Figure 1. In this example, $I(0, 0, 1, 0, 1) = 5$ and $v_5 = 4$. Note that each leaf in the tree represents a *set* of

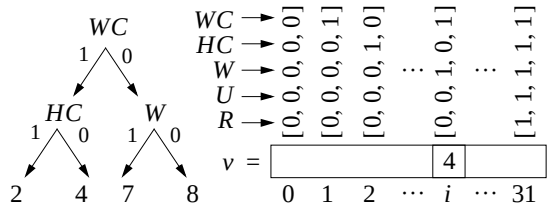

Figure 1: A vector represented as a decision tree

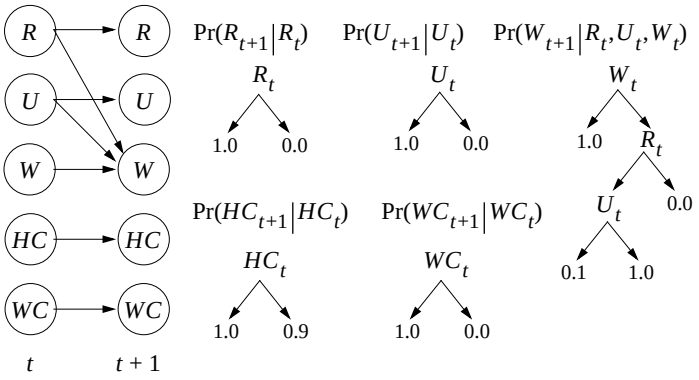

Figure 2: Compact representation for a robot action

indices for the vector. This basic idea of representing the values corresponding to sets of indices is important for compactly representing large vectors and matrices.

The techniques that we consider in this paper work by compactly representing sets of indices whose corresponding entries in a matrix or vector have the same value. Two indices are said to be *equivalent with respect to value* if their corresponding entries are the same. This equivalence relation induces a partition on the set of all indices. Ideally, we would only want to allocate an amount of space polynomial in the number of index variables for each block in this partition — this amount of space would have to suffice for both the value of the entry (common to all of the entries) and for the representation of the block. Fortunately, in some cases, we can represent large blocks of indices quite compactly, *e.g.*, we might interpret the formula $WC \wedge \overline{HC}$ as representing the set of all indices in which WC is assigned 1 and HC is assigned 0. Note that in Figure 1 we have achieved economy of representation by exploiting independence involving the index variables, *e.g.*, if WC is 1, then $f(x)$ is independent of the value of W given a value for HC.

We can represent large matrices in a similar manner. The state-transition probabilities for the action of our robot going outside for a cup of coffee can be specified as a *two-stage temporal Bayesian network* [Dean and Kanazawa, 1989] as shown in Figure 2. This network provides an encoding of the $2^n \times 2^n$ state-transition matrix. Each of the conditional probability tables, one for each index variable, is encoded as a tree as suggested in [Boutilier *et al.*, 1995]. The probability of ending up in one state having started from another is determined by the product of the values found in the trees shown in Figure 3, where, if $i = I(1,1,1,0,0) = 28$ and $j = I(1,1,0,0,0) = 24$, then $a_{ij} = \Pr[R_{t+1} \wedge U_{t+1} \wedge W_{t+1} \wedge \overline{HC}_{t+1} \wedge \overline{WC}_{t+1} | R_t \wedge U_t \wedge \overline{W}_t \wedge \overline{HC}_t \wedge \overline{WC}_t] = 0.01$.

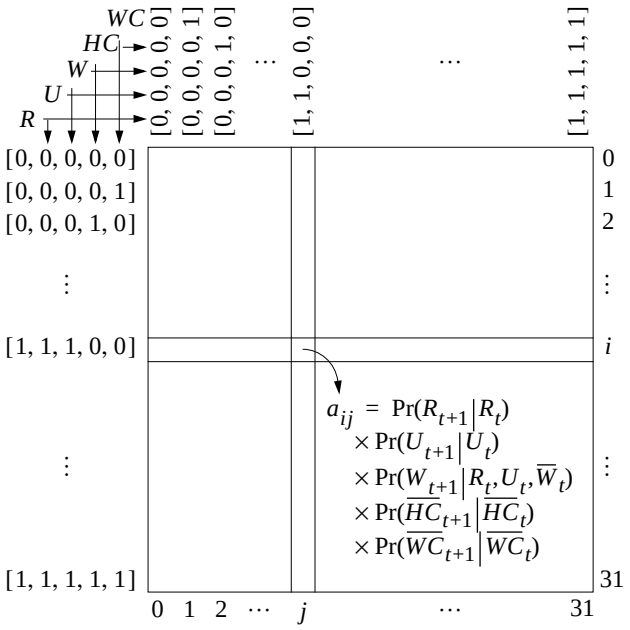

Figure 3: Transition matrix for a robot action

Let $A = \{a_{ij}\}$ so that a_{ij} can be represented as a function of the index variables

$$f(Z_1, \ldots, Z_m; Z'_1, \ldots, Z'_m)$$

where $i = I(Z_1, \ldots, Z_m)$ and $j = I(Z'_1, \ldots, Z'_m)$. In general, if we wish to represent a linear transformation from $\Re^{2^m}$ to $\Re^{2^m}$ as a $2^m \times 2^m$ matrix, we assume that the corresponding function on $\Omega \times \Omega$ can be factored as a simple combination (*e.g.*, sum or product) of functions defined on much smaller spaces. Specifically, by analogy with Bayesian networks we assume that f can be rewritten as $\odot_k f_k(Z_k; Y_k)$ where $\odot$ is a binary operator and Y_k denotes a (hopefully small) subset of the $\{Z'_i\}$.[2] In the example above, $\odot$ is scalar multiplication but other operations can used in other applications (*e.g.*, maximum in the case of computing the transitive closure of a large implicitly specified graph represented as an adjacency matrix, and sum in the case of representing additive, multi-attribute value functions). In the following, we assume that each of the f_k is represented as a tree.

4 Elementary Structured Operations

One of the most basic operations on trees will be to *graft* one tree onto another by attaching a copy of the second to all of the leaves of the first then simplifying the resulting tree by removing redundant or useless branches. In terms of partitions, each tree represents a partition and grafting one tree onto another is equivalent to intersecting the two partitions.

[2]To be completely general, we would have to represent each f_k in the form $f_k(Z_k; U_k; W_k)$ where $U_k \subseteq \{Z_1, \ldots, Z_m\} \setminus Z_k$ and $W_k \subseteq \{Z'_1, \ldots, Z'_m\}$ — U_k is a subset of feature variables used for index i excluding Z_k, and W_k is a subset of feature variables used for index j. Using the less expressive representation simplifies our presentation without losing important details.

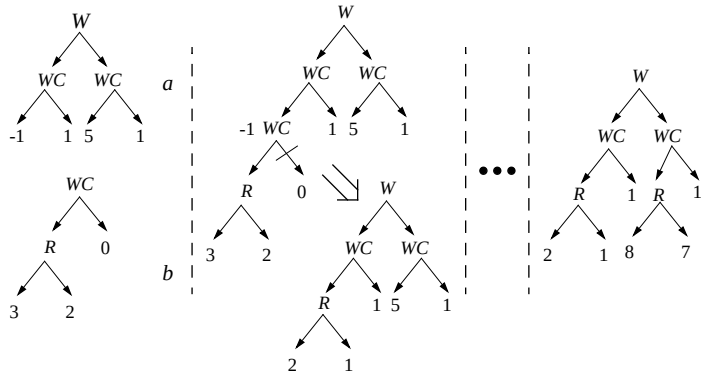

Figure 4: Adding two vectors represented as trees

The following pseudocode for the function TreeAttach implements this operation minus the steps for simplification.

```
func TreeAttach(upper : leaf, lower : leaf, f : func)
    return f(upper, lower)

func TreeAttach(upper : leaf, lower : tree, f : func)
    let newresult :=
        foreach children child of root(lower) do
            TreeAttach(upper, child, f)
    return newresult

func TreeAttach(upper : tree, lower : leaf or tree, f : func)
    let newresult :=
        foreach children child of root(upper)
            TreeAttach(child, lower, f)
    return newresult
```

Using the function TreeAttach we can easily implement vector addition, subtraction, and inner product. Vector addition is illustrated in Figure 4 with an example from the robot domain. Our implementation takes as its inspiration the algorithm described in Boutilier *et al.* [1995] for performing "structured" value iteration to solve large Markov decision processes. Here we adapt their basic methods for manipulating trees representing conditional probability tables to implement many of the basic vector and matrix operations familiar in linear algebra.

The term "structured" refers to the fact that in order to represent the transition matrices and value functions economically Boutilier *et al.* had to exploit structure in the form of regularities in the domain. Following their lead, we refer to both our elementary vector-vector and matrix-vector operations and the composite methods based on them as *structured methods*.

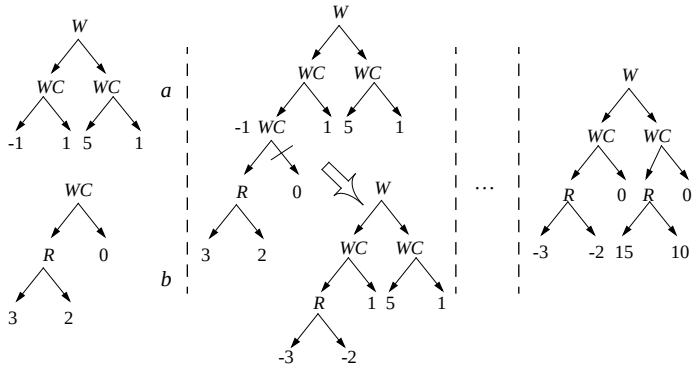

Figure 5: Building intermediate tree for taking inner product of two vectors represented as trees.

func StructuredPlus(domain : **desc**, a : **leaf** or **tree**, b : **leaf** or **tree**)
    return TreeAttach(a, b, $+$)

func StructuredSubtract(domain : **desc**, a : **leaf** or **tree**, b : **leaf** or **tree**)
    return TreeAttach(a, b, $-$)

Inner product is handled similarly. We describe how the operation is done by using the example in Figure 5 in the framework of the coffee robot 2TBN (Figure 2). When grafting one tree to the other, instead of taking the sum as in vector addition, we multiply the item stored in the grafted node to each item in the leaf of the grafting tree. For the example in Figure 5, we multiply -1 stored in WC to each item in the leaf of b. We keep doing this for each leaf node of a, until we get the resulting tree on the rightmost side.

The final step is to take the sum of the items in the leaves of the tree, taking into account that the leaves represent blocks of indices. Since we know the index variables defining the problem, and we know the description of the block corresponding to each leaf node, we can calculate the number of individual indices for each leaf node. In the coffee robot domain example (Figure 2), we had 5 boolean index variables. Thus, $a^T b = 2^2(-3) + 2^2(-2) + 2^3(1) + 2^2(15) + 2^2(10) + 2^3(1) = 96$.

We note that given two vectors a and b with the number of leaves n_a and n_b respectively, the inner product can be done in $O(n_a n_b)$ time. Shown below is the algorithm for the inner product operation.

func StructuredInnerProduct(domain : **desc**, a, b : **leaf** or **tree**)
    let result := TreeAttach(a, b, $\times$)
    foreach leaf leaf **in** result **do**
        leaf = leaf $\times$ Arity(IndexVars(domain) - Path(result, leaf))
    let sum := 0
    foreach leaf leaf **in** result **do**
        sum = sum + leaf
    return sum

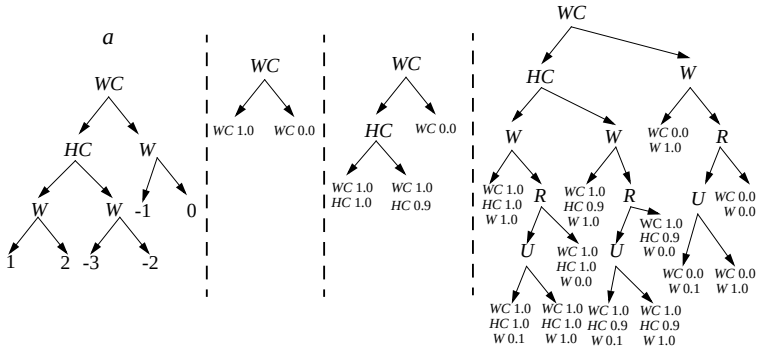

Figure 6: Intermediate steps in multiplying the transpose of a matrix with a vector

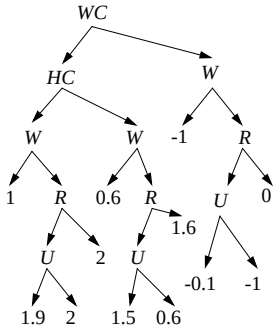

Figure 7: Vector resulting from Figure 6

Another important structured operation is the multiplication of the transpose of a matrix and a vector. In the case in which the matrix corresponds to the state-transition matrix for a Markov decision process and the vector to a reward or value function, this operation amounts to performing a Bellman backup [Puterman, 1994]. Figures 6 and 7 illustrate some of the steps in the application of the following procedure with the vector $\vec{a}$ as shown in Figure 6 and the matrix as described in Figures 2 and 3.

```
StructuredTransposeMultiply({f_i} : list of tree, a : tree)
    let result := EmptyTree()
    foreach index variable Z_k in a do
        foreach leaf leaf in result do
            leaf := TreeAttach(leaf, f_k, Append)
        result := Simplify(result)
    foreach leaf leaf in result do
        leaf := Collapse(leaf, a)
    return result
```

The first loop of StructuredTransposeMultiply constructs the rightmost tree in Figure 6. Simplify eliminates leaves that correspond to empty sets of indices, *i.e.*, leaves at the end of paths corresponding to contradictory assignments to the index variables, and removes redundant branches. The last loop in StructuredTransposeMultiply constructs the final output

using the intermediate result obtained from the first loop; here is how Collapse calculates the value stored at a leaf.

Let $\{Z_{k_1}, \ldots, Z_{k_l}\}$ be the set of index variables used in the tree representation of $\vec{a}$; $\vec{a}$ induces a partition P on the indices and hence on Ω. Associated with each block $B \in P$ is an assignment to the $\{Z_{k_i}\}$; let $z_{k_i,B}$ be the value assigned to Z_{k_i} by B and a_B be the common value assigned to all the indices in B. Let $\{f_{k_1}(Z_{k_1}; Y_{k_1}), \ldots, f_{k_l}(Z_{k_l}; Y_{k_l})\}$ be the set of functions represented as trees associated with the $\{Z_{k_i}\}$, and without loss of generality let $\Omega_{Z_{k_i}} = \{0, \ldots, |\Omega_{Z_{k_i}}| - 1\}$. Finally, let $y_{k_i,h}$ be the (vector) assignment of values to Y_{k_i} in $f_{k_i}(Z_{k_i}; Y_{k_i})$ determined by the path from the root of the tree to the h-th leaf of the tree resulting from the first loop. Just prior to the final loop, the h-th leaf contains the following information:

$$\begin{gathered}
[f_{k_1}(0; y_{k_1,h}), \ldots, f_{k_1}(|\Omega_{k_1}| - 1; y_{k_1,h})] \\
\vdots \\
[f_{k_l}(0; y_{k_l,h}), \ldots, f_{k_l}(|\Omega_{k_l}| - 1; y_{k_l,h})]
\end{gathered}$$

The procedure Collapse takes a leaf and replaces the above information with a single number: $\sum_{B \in P} \prod_{i=1}^{l} f_{k_i}(z_{k_i,B}; y_{k_i,h}) a_B$.[3]

Finally, we define a structured operator for the multiplication of a matrix and a vector. Given that we are multiplying matrix M with vector $\vec{a}$, the i-th component of the vector obtained by multiplying matrix M and vector a is given as

$$\begin{aligned}
(M\vec{a})_i &= \sum_j M_{ij} a_j \\
&= \sum_j f(I^{-1}(i), I^{-1}(j)) a_j \\
&= \sum_j \prod_k f_k(I^{-1}(i)_k, I^{-1}(j)) a_j.
\end{aligned}$$

The algorithm for carrying out above equation is another tree-manipulation algorithm. Again, we explain the algorithm in terms of the 2TBN in Figure 2 with the example shown in Figure 8. Assume that the vector we are given as input is the leftmost tree in Figure 8. We want to calculate how the next time-step index variables will be affected after the vector is multiplied by the transition probability matrix. We set an ordering for the next time-step index variables; let's say we have $[W, U, R, HC, WC]$. We denote next time-step index variables as primed variables, $[W', U', R', HC', WC']$. We choose the first index variable in the ordering, W', and attach its conditional probability tree to each leaf of the vector tree to obtain the tree shown in the second column of Figure 8.

Eventually we will have to add two subtrees of W by the definition of matrix-vector multiplication. Since no other index variable depends on W, we can safely add the two subtrees resulting in the tree shown in the third column; if some other index variable depends

[3]Note that for the leaves in Figure 6, given that the index variables are binary and the functions are probability distributions, we need only indicate a value for Z_k, say p, since the value for $\overline{Z}_k$ is $1 - p$.

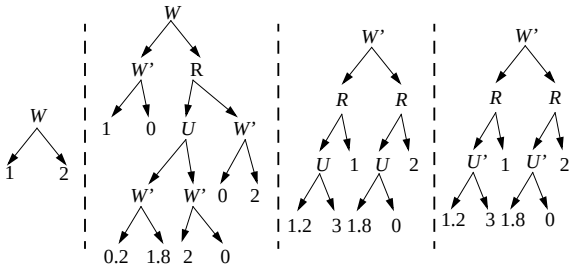

Figure 8: Left multiplication of the transition probability matrix with a vector (Step 1).

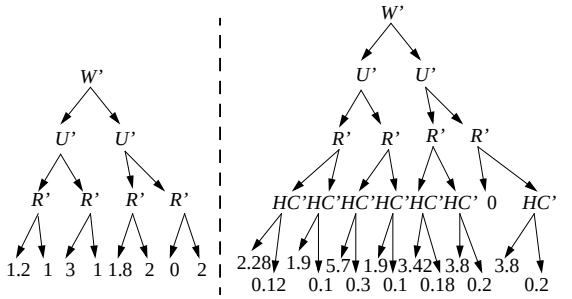

Figure 9: Multiplication of the transition probability matrix with a vector (Step 2).

on W, we can't simply add subtrees since in adding subtrees we are taking advantage of the fact $(xy + zy) = (x + z)y$. In this case, we select U' and repeat the same operation. The result is shown in the last column of Figure 8.

The left column in Figure 9 shows the tree after the conditional probability tree for R' is grafted. Exploiting conditional independence we perform some simplifications. HC' is then grafted and the result is shown in the right side of Figure 9. Since the last index variable WC' does not depend on any other variable and it is persistent, we can skip grafting its conditional probability tree.

The final result is shown in Figure 10 where we eliminate the primes since every index variable in the tree is the next-time step index variable, and the next time step has become the current time step.

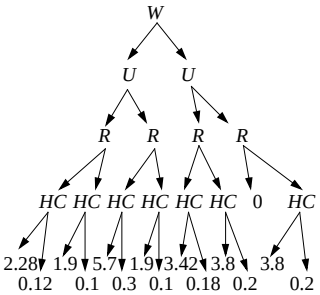

Figure 10: Left multiplication of the transition probability matrix with a vector (Step 3).

The following is a high-level description of the algorithm for implicit matrix-vector multiplication:

```
func StructuredMultiply(domain : desc, a : leaf or tree)
    let result := a
    foreach k = 1, ... , n do
        result := TreeAttach(result, domain[k], ×)
        foreach node node in result do
            if the subtrees of node can be added then
                replace node by StructuredPlus of subtrees
    return result
```

5 Approximating Large Matrices and Vectors

We mentioned that the intermediate or final values of a calculation involving structured operations could explode such that the size of the representation of the coarsest uniform partition with respect to the terms involved in the calculation could grow to be of a size exponential in the total number of terms. This is not a problem in the case of calculating $[A\vec{u}](i)$ or $[A^k\vec{u}](i)$, but can, for example, lead to problems in solving Equation 4.

To avoid such combinatorial explosions, we use approximation methods to make sure that the sizes of intermediate results remain tractable. These methods involve combining blocks in partitions whose associated values are *approximately* the same. Specifically, we apply standard pruning technique similar to those used in learning decision trees [Quinlan, 1992]. The basic idea when applied to elementary structured operations on trees is that whenever the variance of the values in the leaves in a particular subtree is less than some threshold, we replace that subtree by a leaf containing the average. Since some indices are treated the same whose entries are in fact different, these methods introduce errors into calculations. Our methods are heuristic as it is easily shown that finding the smallest representation that achieves a given error is NP-hard [Dean *et al.*, 1997]. However, by keeping track of bounds on the values of the entries for indices in each block of the partitions (the partitions are implicit in the tree data structures), we can report bounds on the error in the final results.

Whenever we deal with numerical calculations that introduce errors, we have to address issues of numerical stability, convergence, and precision. In Section 6, we consider several applications of our structured methods accompanied by appropriate numerical analyses. In particular, we show how to make use of traditional methods for analyzing the consequences of floating point underflow and overflow [Wilkinson, 1963] to study convergence and precision in the use of structured operations. We also look at some ideas from learning theory and consider how they can be used to analyze the result of applying pruning techniques to tree representations of vectors and matrices.

In preparation for the analyses of Section 6, the following two subsections provide some very basic background on modern error analysis and a quick introduction to some useful ideas from machine learning and function approximation theory.

5.1 Error Analysis from Numerical Techniques

A great deal of attention in numerical analysis has been focused on solving systems of linear equations. Since not all real numbers can be represented exactly (or compactly) on computers, researchers in numerical analysis must carefully consider the effect of rounding errors in each algorithm proposed for solving systems of linear equations. The basic theory makes use of the *condition number* of a matrix. Given a matrix norm $\|\cdot\|$ (refer to [Kincaid and Cheney, 1991] for detailed introduction), the condition number of matrix A is defined as

$$\kappa(A) \equiv \|A\|\,\|A^{-1}\|.$$

The meaning of this quantity is revealed by the following question: *Given a system of linear equations $Ax = b$, suppose that by perturbation, b becomes $\tilde{b}$. If $\tilde{x}$ satisfies $A\tilde{x} = \tilde{b}$, how much do x and $\tilde{x}$ differ in terms of norm $\|\cdot\|$?* Assuming that A^{-1} exists, we have

$$\begin{aligned}\|x - \tilde{x}\| = \|A^{-1}b - A^{-1}\tilde{b}\| &\le \|A^{-1}\|\|b - \tilde{b}\| = \|A^{-1}\|\|Ax\|\frac{\|b - \tilde{b}\|}{\|b\|} \\ &\le \|A^{-1}\|\|A\|\|x\|\frac{\|b - \tilde{b}\|}{\|b\|}\end{aligned}$$

which leads to

$$\frac{\|x - \tilde{x}\|}{\|x\|} \le \|A^{-1}\|\|A\|\frac{\|b - \tilde{b}\|}{\|b\|} = \kappa(A)\frac{\|b - \tilde{b}\|}{\|b\|}\ .$$

From the above inequality, we see that $\kappa(A)$ serves as an upper bound on the difference between the solutions, x and $\tilde{x}$, with respect to perturbations in the right-hand side of the equation $Ax = b$. If $\kappa(A)$ is large, small perturbations lead to big differences in the solutions. Often, the perturbations are formulated in terms of a matrix E (particularly for those methods that calculate A^{-1} directly), so that we are solving

$$(A + E)\tilde{x} = b$$

such that $E\tilde{x} = b - \tilde{b}$. The advantage of using a matrix E is that in some cases, we can easily denote the error incurred by inaccurate arithmetic operations. For example, when we use the Gaussian elimination method to solve a system of equations, where we compute L and U matrices, we set

$$LU \equiv A + E\ .$$

If every element of U is in $[-1, 1]$, then every element of E is guaranteed to be between $[-\frac{1}{2}\epsilon, \frac{1}{2}\epsilon]$, where ϵ denotes the machine precision [Wilkinson, 1963]. The relative difference in the solutions can be also derived using E:

$$\frac{\|x - \tilde{x}\|}{\|x\|} \le \|I - (A + E)^{-1}A\| \le \frac{\|E\|\|A^{-1}\|}{1 - \|E\|\|A^{-1}\|}$$

Note that $\kappa(A)$ denotes a property of the matrix A which is independent of the accuracy of the arithmetic operations, whereas E denotes the error incurred during the method, which is dependent on *both* $\kappa(A)$ and the accuracy of the arithmetic operations. Although the standard use of E deals with machine precision [Wilkinson, 1963], the basic approach can easily be extended to analyze the errors that result from pruning.

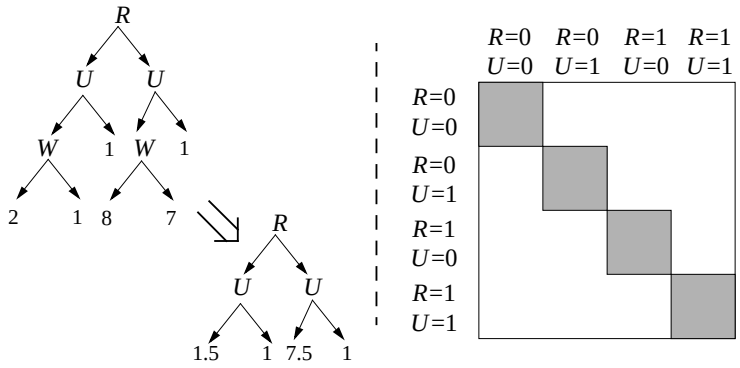

Figure 11: An example illustrating the pruning process and the resulting averager matrix. The example follows the domain given in Figure 2. The four diagonal blocks are 8×8.

5.2 Averagers and Stable Approximations

Another way to capture certain types of approximation is developed in the work of Gordon [1995]. Gordon's method is especially well suited for our analysis given that our approximation method, pruning, can be seen as a particular case of an *averager*. Given an exact vector $\vec{x}$, the approximate vector $\vec{x}'$ is calculated by the following formula:

$$\vec{x}'(i) = [M_\beta \vec{x}](i) \equiv \sum_j \beta_{ij} \vec{x}(j)$$

We define an averager to be a matrix M_β consisting of the elements $\{\beta_{ij}\}$ corresponding to nonnegative reals and satisfying $\sum_j \beta_{ij} = 1$ for all j. To see that our pruning method is an averager, we refer to the example shown in Figure 11. In this example, the non-zero elements are gathered along the diagonal. The ordering of the index variables in the matrix and those in the tree are chosen to be the same in this example. In general, we won't find such a nice diagonal decomposition; however, any approximation carried out by pruning has its associated averager matrix M_β.

Gordon [1995] shows that under certain conditions the use of an averager results in an approximation which is *stable* in the sense that it converges to an approximate solution whose error with respect to the exact solution is nicely bounded.

6 Applications and Analysis

There is a large class of numerical algorithms that are described and implemented in terms of elementary operations on matrices and vectors. With the structured operations for implicit matrices and vectors defined in the previous sections, we can extend many of these algorithms to handle large implicit matrices and vectors. In this section, we provide examples of these extensions and analyze the resulting algorithms.

We note that some algorithms such as Gaussian elimination cannot easily be implemented using the data structures and basic operations that we develop in this paper. In particular

some operations common in manipulating explicit matrices, *e.g.*, exchanging columns or rows, are made difficult by tree representations. We do not consider extending some popular algorithms that are known for being efficient and numerically stable, *e.g.*, those involving LU decompositions, because of the difficulty of implementing such operations. Instead, we focus on algorithms based on the elementary operations for explicit matrices and vectors that we extended to implicit matrices and vectors in Section 4.

6.1 Solving Systems of Linear Equations

Solving systems of linear equations is the most fundamental problem in linear algebra and an often required subroutine in many numerical procedures. Algorithms for solving systems of linear equations are used in a wide range of problems in AI, OR, economics, and scientific computing.

There are many algorithms suggested for solving systems of linear equations, *e.g.*, Gaussian elimination method, matrix decomposition methods, and iterative methods. However, we concentrate on iterative methods that only require matrix-vector multiplication and vector-vector addition operations, since we want to solve large problems encoded implicitly. We begin with one of the simplest iterative algorithms used in the solution of Markov decision processes.

6.1.1 Policy Evaluation using Bellman Backup

Suppose we are given an MDP $M = (Q, A, P, \vec{r})$, a discount rate γ, and a policy $\pi : Q \to A$ and we wish to evaluate the policy. The *value of the policy* π, denoted $\vec{v^\pi}$, is defined as the vector of scalars whose kth component is the discounted cumulative reward for starting in state k and henceforth acting according to the policy π. This vector is defined by the following system of equations represented in matrix form.

$$(I - \gamma(P^\pi)^T)\vec{v^\pi} = \vec{r}.$$

where P^π is the transition probability matrix corresponding to the policy π. It can easily be shown that $\vec{v^\pi}$ satisfies

$$\vec{v^\pi} = \vec{r} + \gamma(P^\pi)^T\vec{v^\pi} \tag{5}$$

and that $\vec{v^\pi}$ is the unique fixed point of the following iterative method

$$\vec{v}^\pi_{i+1} = \vec{r} + \gamma(P^\pi)^T\vec{v}^\pi_i$$

due to Bellman [1957]. Note if P^π and $\vec{r}$ are represented as trees and the matrix and vector operations are defined as in Section 4, then the computational cost of each iteration is polynomial in the product of the sizes of P^π and $\vec{r}$. The following pseudo-code is the Bellman backup procedure using structured linear algebra operators:

func StructuredBellmanBackup(domain : **desc**, r : **leaf** or **tree**, v : **leaf** or **tree**)
    let result := StructuredTransposeMultiply(domain, $\gamma \cdot v$)
    return StructuredPlus(result, r)

By the way, in the earlier sections, we did not describe how to multiply vector represented as a tree by a constant. This is done by multiplying every leaf item with the constant.

6.1.1.1 Error analysis using rounding error technique We carry out the analysis of the Bellman backup as follows. By using approximate operators, *i.e.*, structured linear algebra operators with pruning, the Bellman backup can be viewed as

$$\vec{v}^{\pi}_{i+1} = \vec{r} + \gamma[(P^{\pi})^T + E]\vec{v^{\pi}_i} + \vec{e}$$

where matrix E and vector $\vec{e}$ denote the errors incurred during matrix-vector multiplication and vector-vector addition, respectively. Since the true solution satisfies Equation (5), we have

$$\|\vec{v}^{\pi}_{i+1} - \vec{v}^{\pi}\| \leq \gamma\|(P^{\pi})^T\|\|\vec{v^{\pi}_i} - \vec{v}^{\pi}\| + \|\gamma E\vec{v}^{\pi}_i + \vec{e}\|$$

which leads to

$$\|\vec{v}^{\pi}_{\infty} - \vec{v}^{\pi}\| \leq \frac{\max_i \|\gamma E\vec{v}^{\pi}_i + \vec{e}\|}{1 - \gamma\|(P^{\pi})^T\|}$$

This inequality indicates that the approximate solution $\vec{v}^{\pi}_{\infty}$ is within $\frac{\max_i \|\gamma E\vec{v}^{\pi}_i + \vec{e}\|}{1-\gamma\|(P^{\pi})^T\|}$ of the exact solution $\vec{v}^{\pi}$.

Above analysis provides an alternative way to derive the error bound reported in [Dearden and Boutilier, 1 for evaluating policies for MDPs. Their error bound is defined in terms of abstraction error due to pruning ($\vec{e}$ in our analysis) and inexact transition probabilities (E in our analysis).

6.1.2 Richardson's Method

Bellman backup method in the previous section is a special case of more general method called Richardson's method applied to calculating the value function of an MDP. We return to the general problem of solving a system of linear equations. Assume that we want to solve systems of linear equations given in a matrix-vector form

$$A\vec{u} = \vec{v}.$$

Many iterative methods are defined in terms of a *splitting* of A such that $A = M - N$. Using this splitting, the above equation is rewritten in an equivalent form:

$$M\vec{u} = N\vec{u} + \vec{v}.$$

This equation readily suggests an iterative process, defined by

$$M\vec{u}^{(k+1)} = N\vec{u}^{(k)} + \vec{v}. \tag{6}$$

Iterative methods are based on the following fundamental theorem:

Theorem 1 *If $\|M^{-1}N\| < 1$ for some subordinate matrix norm, then the sequence produced by Equation 6 converges to the solution of $A\vec{u} = \vec{v}$ for any initial vector $\vec{u}^{(0)}$.* □

The simplest of the iterative methods is the Richardson method where $M \equiv I$ and $N \equiv I - A$ [Kincaid and Cheney, 1991]. If we know that $\|I - A\| < 1$, we can apply the following algorithm for the implicitly represented matrix A and vector $\vec{v}$ (the Max is over the absolute value of the entries stored in the leaves of the tree).

```
func StructuredRichardson({f_i} : list of tree, b : tree, ε : real)
  let old := 0, new := b
  while Max(|StructuredSubtract(new,old)|) > ε do
    old := new
    new := StructuredSubtract(new,
             StructuredMultiply({f_i}, new))
    new := StructuredPlus(new, b)
  return new
```

The simplicity of the above specification should give you some appreciation of the potential power of applying structured operators. In a similar manner, we can implement variants of iterative methods such as Jacobi, Gauss-Seidel, and successive over relaxation using elementary structured operations and some special operators to handle the splittings. We can also implement structured versions of the bi-conjugate gradient descent method, which falls into another category of algorithms for solving linear equations.

6.1.2.1 Error analysis using rounding error technique To apply rounding error analysis to Richardson's method, we simply replace $\gamma(P^\pi)^T$ in the previous analysis by $I - A$. Thus, for the error associated with approximate structured Richardson's method defined by

$$\vec{u}_{i+1} = \vec{u}_i - (A + E)\vec{u}_i + \vec{v} + \vec{e},$$

we have following inequality for the error associated with the approximate structured Richardson's method:

$$\|\vec{u}_\infty - \vec{u}\| \leq \frac{\max_i \|E\vec{u}_i + \vec{e}\|}{1 - \|I - A\|}$$

The result computed by the approximate Richardson's method, $\vec{u}_\infty$, will remain close to $\vec{u}$, the true solution, as long as $\|I - A\| < 1$. The error bound depends on E and $\vec{e}$, as well as $\|I - A\|$. This comes from the fact that the overall error depends on how much the errors are propagated from one iteration to the next — smaller $\|I - A\|$ means a smaller rate of propagation.

6.1.3 Acceleration Techniques

Equation 6 shows that

$$\|\vec{u}_{i+1} - \vec{u}\| \leq \|M^{-1}N\|\|\vec{u}_i - \vec{u}\|,$$

This means that the speed of convergence is determined by $\|M^{-1}N\|$. It is natural to try to find the best M and N such that $\|M^{-1}N\|$ is minimal. Many *acceleration techniques* attempt to do just this.

The following is a general acceleration technique called the *extrapolation method* that can be applied to any iterative formula derived from Equation 6:

$$\vec{u}^{(k+1)} = M^{-1}N\vec{u}^{(k)} + M^{-1}\vec{v}.$$

We introduce an acceleration parameter $\lambda \neq 0$ to the above equation to obtain

$$\begin{aligned}\vec{u}^{(k+1)} &= \lambda(M^{-1}N\vec{u}^{(k)} + M^{-1}\vec{v}) + (1-\lambda)\vec{u}^{(k)} \\ &= G_\lambda \vec{u}^{(k)} + \lambda\vec{c}\end{aligned}$$

where $G_\lambda \equiv \lambda M^{-1}N + (1-\lambda)I$ and $\vec{c} \equiv M^{-1}\vec{v}$. If the only information available about the eigenvalues of $M^{-1}N$ is that they lie in the interval $[a, b]$, then the best choice for λ is $2/(2-a-b)$ [Kincaid and Cheney, 1991]. We calculate the interval $[a, b]$ as a preprocessing step using structured versions of the power and inverse power methods (see Section 6.2) to determine λ for our experiments.

Algorithms for solving a system of linear equations are used in Markov decision processes for a number of purposes. In this paper, we concentrate on applying equation-solving techniques to calculate the value function for a particular policy π, a mapping from states to actions. The value function for π is the vector $\vec{v}_\pi$ that satisfies the equation

$$(I - \gamma T_\pi)\vec{v}_\pi = \vec{r}_\pi,$$

where $\vec{v}_\pi(i)$ denotes the expected discounted cumulative reward starting in state i and acting according to π for the infinite horizon case with discount rate γ. We assume that T_π, the state-transition probability matrix for π, and $\vec{r}_\pi$, the reward vector for π are specified in structured matrix representation. The iterative formula for the Richardson method reduces to

$$\vec{v}_\pi^{(k+1)} = \gamma T_\pi \vec{v}_\pi^{(k)} + \vec{r}_\pi,$$

whereas the extrapolation method is specified by

$$\vec{v}_\pi^{(k+1)} = \lambda\gamma T_\pi \vec{v}_\pi^{(k)} + (1-\lambda)\vec{v}_\pi^{(k)} + \lambda\vec{r}_\pi.$$

6.1.3.1 Error analysis in terms of averagers The asymptotic behavior of $\vec{v}_\pi^{(k)}$ in the presence of approximate operators can be analyzed in the context of MDPs. We can show

that a certain class of approximations achieves *stable approximation* [Gordon, 1995]. Define the Bellman backup operator T_π as a mapping from $\vec{v}_\pi^{(k)}$ to $\vec{v}_\pi^{(k+1)}$ such that

$$\vec{v}_\pi^{(k+1)}(i) = [T_\pi \vec{v}_\pi^{(k)}](i) \equiv \vec{r}(i) + \gamma \sum_j p_{ij} \vec{v}_\pi^{(k)}(j).$$

It can be shown that the combined mapping $T_\pi \circ M_\beta$ is a stable approximation in the sense that

Theorem 2 (stable approximation) *Let the iteration T_π converges to vector $\vec{v}^*$, and let vector $\vec{v}^+$ be any fixed point of M_β, i.e., $\vec{v}^+ = M_\beta \vec{v}^+$. Suppose that $\|\vec{v}^+ - \vec{v}^*\|_\infty = \epsilon$. Then, the iteration of $T_\pi \circ M_\beta$ converges to the vector $\vec{v}_0$ so that $\|\vec{v}_0 - \vec{v}^*\| \leq \frac{2\gamma\epsilon}{1-\gamma}$.*

Similar theoretical results can be also found in [Tsitsiklis and Van Roy, 1996].

The key idea of establishing stable approximation for approximate structured operators is that the tree pruning procedure is a special case of the averager assuming that (1) the variables appear in a fixed order when we follow the paths from root to leaves and (2) the pruning criterion is changed so that leaves at depth less than, say, h never gets pruned, whereas leaves at depth at least h always gets pruned, *i.e.*, the depth of the whole tree is always at most h.

When the above assumptions are satisfied, we can formulate the approximate Bellman backup as

$$\vec{v}_\pi^{(k+1)} = A(\vec{r} + \gamma A P_\pi \vec{v}_\pi^{(k)})$$

where matrix A represents the averager induced by pruning after each structured operator. Using the fact that $\vec{v}^* = \vec{r} + \gamma P_\pi \vec{v}^*$, we have the following inequality.

$$\begin{aligned}
\|\vec{v}_\pi^{(k+1)} - A\vec{v}_\pi^*\| &= \|\gamma A^2 P \vec{v}_\pi^{(k-1)} - \gamma A P \vec{v}_\pi^*\| \\
&= \|\gamma A^2 P \vec{v}_\pi^{(k-1)} - \gamma A^3 P \vec{v}_\pi^* + \gamma A^3 P \vec{v}_\pi^* - \gamma A P \vec{v}_\pi^*\| \\
&\leq \|\gamma A^2 P\| \|\vec{v}_\pi^{(k)} - A\vec{v}_\pi^*\| + \|A^2 - I\| \|\gamma A P \vec{v}_\pi^*\|
\end{aligned}$$

We immediately notice that $\|\vec{v}_\pi^{(k+1)} - A\vec{v}_\pi^*\|$ is bounded for all k due to the fact that $\|\gamma A^2 P\| < 1$. Thus,

$$\|\vec{v}_\pi^{\infty} - A\vec{v}_\pi^*\| \leq \frac{\|A^2 - I\| \|\gamma A P \vec{v}_\pi^*\|}{1 - \|\gamma A^2 P\|}.$$

Similar analysis can be carried out under the extrapolation method. However, there is no convergence guarantee for the approximate extrapolation method with $\lambda > 1$.

6.1.4 Experiments

As a simple experiment, we compare the performance of the structured Richardson method with the structured version of the acceleration technique both of which were described in

ϵ	γ	Richardson method		Extrapolation method	
		# of iterations	Run time (sec)	# of iterations	Run time (sec)
0.1	0.8	31	37.02	11	13.48
0.1	0.9	72	87.97	23	29.63
0.1	0.95	160	197.66	49	64.57
0.05	0.8	34	40.80	12	14.76
0.05	0.9	78	95.43	25	32.13
0.05	0.95	174	215.05	53	69.76

Figure 12: Experimental results with exact operations.

Section 6.1. The task was to estimate the value of always executing the action stay in Boutilier and Dearden [1996]'s Coffee Robot domain. Figures 12 and 13 show the number of iterations and execution time for the Richardson method and the extrapolation method with varying target absolute error ϵ and discount rate γ. The stopping criteria for the Richardson method is given by

$$\|\vec{v}_\pi^{(k+1)} - \vec{v}_\pi^{(k)}\| \leq \epsilon \frac{1-\gamma}{2\gamma}$$

and the extrapolation method by

$$\|\vec{v}_\pi^{(k+1)} - \vec{v}_\pi^{(k)}\| \leq \epsilon \frac{1-[1-\lambda(1-\gamma)]}{2[1-\lambda(1-\gamma)]}.$$

The latter inequality is derived from the fact that $\|G_\lambda\| \leq 1-\lambda(1-\gamma)$ for our particular choice of λ.

For the Richardson method, one matrix-vector multiplication, one vector-vector addition, and one scalar-vector multiplication are performed at each iteration. For the extrapolation method, one matrix-vector multiplication, two vector-vector additions, and three scalar-vector multiplications are performed per iteration. This explains why the extrapolation method takes slightly longer running time per iteration than the Richardson method. However, we can see that the information about the largest and smallest eigenvalues allows us to apply the extrapolation method in *structured domains*, which is a clear advantage.

Figure 13 shows the results for the same set of problems as in Figure 12 but in this case using the Richardson and extrapolation methods implemented with the pruning techniques. After each exact elementary structured operation, TreePrune takes the (possibly large) tree as an input and examines the leaves of the subtree rooted at each internal node. TreePrune goes through each node in a recursively bottom-up fashion. If the variance of the leaves is smaller than some threshold value given as input, the algorithm contracts the whole subtree to a single leaf node containing the average. Otherwise, it proceeds to the parent node without any changes:

```
func TreePrune(T : tree, θ : real)
  let result :=
    foreach children child of T do
```

ϵ	γ	Richardson method		
		# of iterations	Run time (sec)	Rel err
0.1	0.8	30	7.49	0.22
0.1	0.9	71	17.91	0.17
0.1	0.95	158	40.16	0.14
0.05	0.8	34	8.54	0.22
0.05	0.9	77	19.57	0.17
0.05	0.95	171	43.48	0.14
ϵ	γ	Extrapolation method		
		# of iterations	Run time (sec)	Rel err
0.1	0.8	11	2.89	0.26
0.1	0.9	22	5.99	0.21
0.1	0.95	48	13.39	0.16
0.05	0.8	12	3.17	0.26
0.05	0.9	24	6.61	0.21
0.05	0.95	52	14.54	0.16

Figure 13: Experimental results with approximate operations. "Rel err" denotes the relative error incurred by approximating the vector. The threshold used for tree pruning algorithm heuristic was set to 1.0 throughout the experiments on the Richardson method and 5.0 on the extrapolation method.

```
      TreePrune(child, θ);
  let average := Average of leaf items in result;
  let variance := Variance of leaf items in result;
  if variance < θ do
    return average;
  else return T;
```

Figure 13 also lists the relative error for the approximations. The relative error is defined as $\|(\vec{v}_\pi - \vec{v}'_\pi)/\vec{v}_\pi\|_\infty$ where $\vec{v}_\pi$ and $\vec{v}'_\pi$ represent, respectively, the exact solution and the approximate value functions.

The effectiveness of this pruning technique is sensitive to the threshold (at least in the domain we looked at). In Figure 14, we show the number of leaves in the tree representing the value function produced as output and the relative error achieved by this tree. Note that the number of leaves in trees produced as output is sensitive to the threshold; moreover, the relative error is different among trees with the same number of leaves. There are several reasons for this variation [Boutilier and Dearden, 1996, Singh and Yee, 1994]. In some cases, there are trees that are different in shape but have the same number of leaves. In other cases, there are trees with same shape but different values at the leaves due to different prunings at intermediate steps.

θ	Richardson method		Extrapolation method	
	# of leaves	Rel err	# of leaves	Rel err
0.0	30	0.003	30	0.003
0.1	30	0.003	30	0.003
0.5	30	0.157	30	0.087
1.0	6	0.218	30	0.104
2.0	6	0.519	30	0.173
3.0	6	0.519	6	0.261
4.0	1	0.885	6	0.261
5.0	1	0.878	6	0.261

Figure 14: Experimental results with approximate operations. θ is the threshold used for pruning. "Rel err" denotes the relative error incurred by approximating the vector. ϵ and γ were set 0.1 and 0.8, respectively.

6.2 Calculating Eigensystems

The eigensystem (eigenvalues and eigenvectors) of a matrix reveal a lot of useful information about problems described in terms of the matrix. For example, in a Markov chain, the second largest eigenvalue provides a measure of the *mixing rate* of the process: how fast a random walk converges to the steady state distribution. Also, if none of the eigenvalues is -1, then the chain is called *ergodic* and is thus known to have several desirable properties.

In this section, we show how to calculate the eigensystem of a large matrix using structured linear algebra operators. We also show its application to compuate singular values of a large matrix.

6.2.1 Power Method

There are a lot of numerical algorithms for calculating the eigensystem of a matrix. In the following, we illustrate a structured version of the *power method*, which is an iterative method for finding largest eigenvalue and its corresponding eigenvector. The power method utilizes the following property:

Theorem 3 (Power Method) *For an arbitrary matrix M and an initial vector $x^{(0)}$, the vector*

$$x^{(k)} = Mx^{(k-1)}$$

aligns with the eigenvector u_1 corresponding to the largest eigenvalue λ_1. Moreover,

$$r^{(k)} = \frac{\|x^{(k+1)}\|}{\|x^{(k)}\|}$$

approaches to the eigenvalue λ_1.

The following pseudo-code is the structured version of the power method:

```
func StructuredPowerMethod(domain : desc, b : leaf or tree, ε : real)
    let oldx := b, newx := 0, α := 0
    while Max(|StructuredSubtract(domain, newx, oldx)|) > ε do
        oldx := newx
        newx := StructuredMultiply(domain, oldx)
        α := √StructuredInnerProduct(domain, newx, newx)
        newx := StructuredTimes(domain, 1/α, newx)
    return {α, newx}
```

At the termination of the program, α provides an estimate of the largest eigenvalue and an estimate of the associated eigenvector is stored at newx for any trial vector b. Variants of the power method such as the *inverse power method* to calculate the smallest eigenvalue and its eigenvector, and the *shifted power method* to get the eigenvalue farthest from a specific value given as the input, *etc.*, are implementable using structured operations. For a detailed description of numerical issues concerning the classical power method and its variants, refer to [Press *et al.*, 1993].

We can extend the power method so that it makes the set of trial vectors converge to the dominant set of eigenvalues/eigenvectors of the matrix. This is called the *simultaneous iteration method*. It works for an arbitrary matrix but it solves m systems of linear equations in each iteration when finding m dominant eigenvalues/eigenvectors. Structured versions of the simultaneous iteration method can be derived from the classical version in a straightforward way. Detailed analysis of the classical version of simultaneous iteration method can be found in [Jennings and McKeown, 1992].

6.2.2 Singular Value Decomposition

A singular value decomposition (SVD) of a matrix also reveals useful information about a matrix. SVD is applied in various tasks such as linear regression and principal component analysis. We start with the basic singular value decomposition theorem:

Theorem 4 (SVD) *An arbitrary (stochastic) $n \times n$ matrix M can be factored as*

$$M = PDQ$$

where P is an $n \times m$ unitary matrix, D is an $m \times m$ diagonal matrix, and Q is an $m \times n$ unitary matrix.

The non-zero diagonal entries in D are called *singular values* of matrix M. They are exactly the square root of eigenvalues of the matrix $M^T M$. This property shows that to calculate singular value decomposition on the matrix M, we first calculate eigenvalues/eigenvectors of symmetric matrix $M^T M$. We extend the power method described in the previous section to calculate all eigenvalues/eigenvectors of $M^T M$.

Since symmetric matrices have nonnegative eigenvalues, let's say $M^T M$ has eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq 0$. We denote u_i as the eigenvector corresponding to the eigenvalue

λ_i, $i = 1, \ldots, n$. A symmetric matrix has another property [Strang, 1980] that there exists a set of orthonormal eigenvectors u_i's such that

$$M^T M = \lambda_1 u_1 u_1^T + \cdots + \lambda_n u_n u_n^T.$$

The power method calculates λ_1 and u_1. Suppose that we reapply the power method to

$$M^T M - \lambda_1 u_1 u_1^T.$$

This will yield the eigenvalue λ_2 and associated eigenvector u_2'. If $\lambda_2 = \lambda_1$, there is no guarantee that u_2' is perpendicular to u_1, we need to find u_2 that is both the eigenvector corresponding to λ_2 and perpendicular to u_1. This can be done by projection:

$$\begin{aligned} v_2 &= u_2' - \frac{u_1^T u_2'}{u_1^T u_1} u_1, \\ u_2 &= v_2 / \|v_2\|. \end{aligned}$$

We can easily verify that $M^T M u_2 = \lambda_2 u_2$. In general, if we know eigenvalues/eigenvectors $\{\lambda_i, u_i | i = 1, \ldots, k\}$, we can calculate λ_{k+1} and u_{k+1}' by applying the power method to

$$M^T M - \lambda_1 u_1 u_1^T - \cdots - \lambda_k u_k u_k^T. \tag{7}$$

¿From that, we orthnormalize u_{k+1}' with respect to $u_1, \ldots, u_k$:

$$\begin{aligned} v_{k+1} &= u_{k+1}' - \frac{u_1^T u_{k+1}'}{u_1^T u_1} u_1 - \cdots - \frac{u_k^T u_{k+1}'}{u_k^T u_k} u_k, \\ u_{k+1} &= v_{k+1} / \|v_{k+1}\|. \end{aligned} \tag{8}$$

Suppose that we have computed m non-zero eigenvalue/eigenvector pairs of $M^T M$:

$$\{(\lambda_i, u_i) | i = 1, \ldots, m\}.$$

We are ready to build matrices P, D and Q that form singular value decomposition of M:

$$\begin{aligned} D &= \mathrm{diag}(\sqrt{\lambda_1}, \ldots, \sqrt{\lambda_m}), \\ P &= \begin{bmatrix} — & u_1 & — \\ & \vdots & \\ — & u_m & — \end{bmatrix}, \\ Q &= \begin{bmatrix} | & & | \\ \frac{1}{\sqrt{\lambda_1}} M u_1 & \cdots & \frac{1}{\sqrt{\lambda_n}} M u_m \\ | & & | \end{bmatrix}. \end{aligned}$$

Instead of calculating all of m non-zero singular values, we can compute some ($m' < m$) of them, building $m' \times m'$ matrix D, $n \times m'$ matrix P and $m' \times n$ matrix Q. This is one way of approximating the matrix M [Press *et al.*, 1993]. The following is the structured version of singular value decomposition algorithm which calculates m' singular values:

```
func StructuredSVD(domain : desc, m' : integer, ε : real)
    let eigensystem := {}, P := {}, D := {}, Q := {}
    do i = 1, ..., m'
        let eigenpair := StructuredPowerMethod(domain, eigensystem,
                                 StructuredRandomVector(), ε)
        eigenpair := StructuredOrthonormalize(domain, eigensystem,
                                 eigenpair)
        eigensystem := Append(eigensystem, eigenpair)
    do i = 1, ..., m'
        D := Append(D, sqrt(eigensystem[i][1]))
        Q := Append(Q, eigensystem[i][2])
        P := Append(P, StructuredTimes(domain, 1/D[i],
                                 StructuredMultiply(domain, eigensystem[i][2])))
    return {P, D, Q}
```

Note that we implicitly extended StructuredPowerMethod to calculate the i-th eigenvalue-eigenvector pair in the way shown in Equation 7. StructuredOrthonormalize orthonormalizes the i-th eigenvector with respect to those eigenvectors already calculated according to Equation 8.

6.2.3 Experiments

In this section, we summarize some experiments to illustrate the structured version of power method. The task was to find the dominant eigenvalue-eigenvector pair for a structured stochastic transition matrix. Figure 15 describes the result for three stochastic transition matrices corresponding to different actions in the coffee robot domain. Since it is known that the dominant eigenvalue of a stochastic transition matrix is 1, we define relative error in the eigenvalue as $1-\lambda$, where λ is the eigenvalue calculated by the (approximate) power method. For each $\theta = 0.00, 0.01, 0.05, 0.10$, we get possibly different eigenvectors. Thus, in the last four columns of Figure 15, we show the inner products between eigenvectors. Except for $\theta = 0.1$, the eigenvectors were close to the true eigenvector. In the case of go1 with $\theta = 0.05$, the obtained eigenvector was indeed very close to the true eigenvector — it just converged to some eigenvector different from those with $\theta = 0.00, 0.01$. Note that multiple eigenvectors can share the same eigenvalue.

7 Related Work

Press *et al.* [1993] describe a wide range of numerical methods. To the extent that this body of work is concerned with exploiting structure, the emphasis is on sparse matrix methods. For a good introduction to iterative methods for solving systems of linear equations, see [Kincaid and Cheney, 1991]. The pioneering work by Wilkinson [1963] gives another perspective in analyzing direct methods such as Gaussian elimination in the presence of approximate operators.

action	θ	T	# of leaves	rel err in eigenvalue	$\vec{x} \cdot \vec{x}_\theta$			
					0.00	0.01	0.05	0.1
stay	0.00	1128.22	400	0.0258	1.0	0.996	0.959	0.282
	0.01	44.58	12	0.0160		1.0	0.975	0.259
	0.05	14.46	7	0.0000			1.0	0.224
	0.10	5.24	1	0.9975				1.0
go1	0.00	1124.7	400	0.0258	1.0	0.982	0.503	0.282
	0.01	29.83	9	0.0154		1.0	0.493	0.258
	0.05	8.9	5	0.0000			1.0	0.447
	0.10	4.86	1	0.9975				1.0
go2	0.00	1142.96	400	0.0258	1.0	0.996	0.959	0.282
	0.01	44.02	12	0.0160		1.0	0.975	0.259
	0.05	14.13	7	0.0000			1.0	0.224
	0.10	5.15	1	0.9975				1.0

Figure 15: Results of the structured power method on Coffee Robot domain. For the transition probability matrices corresponding to actions stay, go1, and go2, we calculated the dominant eigenvalue/eigenvector for each action. For varying degree of approximation (θ), we ran power method for 20 iterations. T is the running time in terms of seconds. Note that the number of leaves for the eigenvector decreases dramatically as we increased θ, thus sacrificing the accuracy of each structured linear operator. The method to calculate the relative errors in eigenvalues and compare the eigenvectors can be found in the text.

Work on Markov decision processes dates back to the 1950's and there are several excellent up-to-date references available that summarize the research carried out in operations research and adaptive control [Puterman, 1994, Bertsekas, 1987]. Much of the emphasis in these fields was on the use of iterative, dynamic programming methods for solving discrete problems (or discretized versions of continuous problems); unfortunately, since these methods required summing over the state and actions spaces, they did not scale to most of the problems encountered in practice [Littman *et al.*, 1995].

Dean and Kanazawa [1989] and Tatman and Shachter [1990] introduced the idea of structured Markov processes and suggested how they might be used for representing and solving planning and control problems with very large state and action spaces [Dean and Wellman, 1991]. In recent years, a great deal of progress has been made exploring structured versions of earlier, unstructured algorithms [Boutilier *et al.*, 1995, Dean *et al.*, 1997]. Tsitsiklis and Van Roy [1996] also discuss approximation methods for solving factored MDPs. The approach described in this paper was inspired in part by the structured methods presented in [Boutilier *et al.*, 1995] and the subsequent analysis of the underlying reduced models in Dean *et al.* [1997]. Boutilier *et al.* [1999] provide a recent survey of structured methods for representing and solving very large MDPs.

The work on solving MDPs is also closely related to work on reinforcement learning algorithms. A survey by Kaelbling et al. [1996] and a book by Sutton and Barto [1998] provide excellent introductions to reinforcement learning and its relationship to MDPs. The book by Bertsekas and Tsitsiklis [1996] connects the traditional work on dynamic programming,

research on reinforcement learning, and work on function approximation and machine learning. The most relevant aspects of this work concern the convergence and error analysis of reinforcement learning techniques which make use of function approximation schemes. Gordon [1995] provides a good overview of this work as does the text by Bertsekas and Tsitsiklis [1996].

8 Conclusion

This paper describes procedures for implementing linear algebra operators that apply to matrices and vectors represented in factored form. These procedures can be used to extend numerical methods to handle very high dimensional vector spaces. In this paper, we emphasized factored representations based on trees, thereby generalizing on the sort of graphical models popular in artificial intelligence and operations research.

In terms of future work, we are particularly excited about the prospects for using machine learning techniques based on sampling to find small, approximate representations. In terms of the general approach, we are currently pursuing answers to the following question: *Given fixed computational resources, what are the best points in the calculations to apply approximation?* We've been considering a number of possibilities: We could "cache out" the result of each each elementary operation by applying TreePrune to obtain an intermediate result. Alternatively, we might perform a number of exact binary operations, monitoring the size of the intermediate results, and then, when the intermediate results threaten to become too large, apply pruning to obtain a compact, approximate intermediate result.

We are also working on methods that involve sampling from the indices of large matrices and vectors and then using machine learning techniques to the sample to induce a compact model of the intermediate results of the binary operations. In analogy to the idea in the previous paragraph of performing sequences of binary operations before caching out the results, we might also sample computational threads or trees corresponding to partial calculations from sequences of binary operations. In this case, the question of how long or complicated these sequences are allowed to become may be resolved in some cases by an appeal to uniform sampling theory. For the task of solving factored Markov decision processes, the resulting algorithm would be a hybrid of the approximate structured value iteration of Boutilier and Dearden [1996] and the sparse sampling planner of Kearns *et al.* [1999].

While our initial motivation was to expedite the solution of Markov decision processes, in this paper we have attempted to push the generality of our methods as far as possible. This exercise serves to (a) increase the class of problems that can be solved using structured methods and (b) increase the class of methods that can be extended and brought to bear on the problems that motivated us in the first place. One of the most important advantages of our approach over earlier, more specialized approaches is that we can now easily take advantage of a wealth of knowledge from numerical analysis and related disciplines in devising new algorithms.

By keeping track of bounds on the values of the entries for indices in each block of the partitions, we can report bounds on the error in the final results. Standard pruning techniques

can be applied to elementary structured operations to produce approximate structured operations. In much the same way as finite-precision arithmetic introduces errors in computer implementations of numerical methods, approximate operations on matrices and vectors introduce errors and thus require careful analysis with respect to convergence and precision. Under reasonable assumptions, we can guarantee that the resulting approximations converge and the error (the difference between result from the algorithm using the approximate structured operators and the true answer) is bounded.

The contributions of this work include data structures for representing very large matrices and vectors, a set of procedures that operate on these data structures, and a set of analytical methods that enable us to apply a wide range of numerical methods based on linear algebra directly to the solution of combinatorial optimization problems involving very large matrices and vectors.

Acknowledgments

We thank Craig Boutilier for numerous discussions and insights as well as sharing the Coffee Robot domain with us.

References

[Bellman, 1957] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Bertsekas and Tsitsiklis, 1996] Dimitris P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, Massachusetts, 1996.

[Bertsekas, 1987] Dimitri P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, N.J., 1987.

[Boutilier and Dearden, 1996] Craig Boutilier and Richard Dearden. Approximating value trees in structured dynamic programming. In Lorenza Saitta, editor, *Proceedings ICML-96*, 1996.

[Boutilier *et al.*, 1995] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Exploiting structure in policy construction. In *Proceedings IJCAI-99*, pages 1104–1111, 1995.

[Boutilier *et al.*, 1999] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[Dean and Givan, 1997] Thomas Dean and Robert Givan. Model minimization in Markov decision processes. In *Proceedings AAAI-97*. AAAI, 1997.

[Dean and Kanazawa, 1989] Thomas Dean and Keiji Kanazawa. A Model for Reasoning about Persistence and Causation. *Computational Intelligence*, pages 143–150, 1989.

[Dean and Wellman, 1991] Thomas Dean and Michael Wellman. *Planning and Control*. Morgan Kaufmann, San Francisco, California, 1991.

[Dean *et al.*, 1997] Thomas Dean, Robert Givan, and Sonia Leach. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)*, pages 124–131, San Francisco, CA, 1997. Morgan Kaufmann Publishers.

[Dearden and Boutilier, 1997] Richard Dearden and Craig Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 1997.

[Fikes and Nilsson, 1971] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Gordon, 1995] Geoffrey J. Gordon. Stable function approximation in dynamic programming. Technical Report CMU-CS-103, School of Computer Science, Carnegie Mellon University, 1995.

[Hoey *et al.*, 1999] Jesse Hoey, Robert St.Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings UAI-99*, 1999.

[Jennings and McKeown, 1992] Alan Jennings and J. J. McKeown. *Matrix Computation*. John Wiley & Sons, 1992.

[Kaelbling *et al.*, 1996] Leslie P. Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[Kearns *et al.*, 1999] Michael Kearns, Yishay Mansour, and Andrew Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. In *Proceedings IJCAI-99*, 1999.

[Kincaid and Cheney, 1991] David Kincaid and Ward Cheney. *Numerical Analysis*. Brooks/Cole Publishing Company, 1991.

[Littman *et al.*, 1995] Michael Littman, Thomas Dean, and Leslie Kaelbling. On the complexity of solving Markov decision problems. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 394–402, 1995.

[Press *et al.*, 1993] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, 1993.

[Puterman, 1994] Martin L. Puterman. *Markov Decision Processes*. John Wiley & Sons, New York, 1994.

[Quinlan, 1992] J. Ross Quinlan. *C4.5 : Programs for Machine Learning*. Morgan Kaufmann, 1992.

[Singh and Yee, 1994] Satinder P. Singh and Richard C. Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16:227–233, 1994.

[Strang, 1980] Gilbert Strang. *Linear Algebra and Its Applications*. Academic Press, 1980.

[Sutton and Barto, 1998] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.

[Tatman and Shachter, 1990] Joseph A. Tatman and Ross D. Shachter. Dynamic programming and influence diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(2):365–379, 1990.

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

[Wilkinson, 1963] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1963.