Optimal and Approximate Stochastic Planning using Decision Diagrams

Jesse Hoey Robert St-Aubin Alan Hu Craig Boutilier

TR-00-05

Department of Computer Science
University of British Columbia
Vancouver, BC, V6T 1Z4, CANADA
{jhoey,staubin,ajh}@cs.ubc.ca, cebly@cs.toronto.edu

June 10, 2000

Abstract

Structured methods for solving factored Markov decision processes (MDPs) with large state spaces have been proposed recently to allow dynamic programming to be applied without the need for complete state enumeration. We present algebraic decision diagrams (ADDs) as efficient data structures for solving very large MDPs. We describe a new value iteration algorithm for exact dynamic programming, using an ADD input representation of the MDP. We extend this algorithm with an approximate version for generating near-optimal value functions and policies with much lower time and space requirements than the exact version. We demonstrate our methods on a class of large MDPs (up to 34 billion states). We show that significant gains can be had with our optimal value iteration algorithm when compared to tree-structured representations (with up to a twenty-fold reduction in the number of nodes required to represent optimal value functions). We then demonstrate our approximate algorithm and compare results with the optimal ones. Finally, we examine various variable reordering techniques and demonstrate their use within the context of our methods.

1 Introduction

Markov decision processes (MDPs) have become the semantic model of choice for decision theoretic planning (DTP) in the AI planning community. While classical computational methods for solving MDPs, such as value iteration and policy iteration [19], are often effective for small problems, typical AI planning problems fall prey to Bellman's *curse of dimensionality*: the size of the state space grows exponentially with the number of domain features. Thus, classical dynamic programming, which requires explicit enumeration of the state space, is typically infeasible for feature-based planning problems.

Considerable effort has been devoted to developing representational and computational methods for MDPs that obviate the need to enumerate the state space [6]. *Aggregation* methods do this by aggregating a set of states and treating the states within any aggregate state as if they were identical [3]. Within AI, *abstraction* techniques have been widely studied as a form of aggregation, where states are (implicitly) grouped by ignoring certain problem variables [14, 8, 12]. These methods automatically generate abstract MDPs by exploiting structured representations, such as probabilistic STRIPS rules [16] or *dynamic Bayesian network* (DBN) representations of actions [13, 8].

In this paper, we describe a dynamic abstraction method for solving MDPs using *algebraic decision diagrams* (ADDs) [1] to represent value functions and policies. ADDs are generalizations of ordered *binary decision diagrams* (BDDs) [10] that allow non-boolean labels at terminal nodes. This representational technique allows one to describe a value function (or policy) as a function of the variables describing the domain rather than in the classical "tabular" way. The decision graph used to represent this function is often extremely compact, implicitly grouping together states that agree on value at different points in the dynamic programming computation. As such, the number of expected value computations and maximizations required by dynamic programming are greatly reduced.

The algorithm described here derives from the *structured policy iteration (SPI)* algorithm of [8, 7, 5], where decision trees are used to represent value functions and policies. Given a DBN action representation (with decision trees used to represent conditional probability tables) and a decision tree representation of the reward function, SPI constructs value functions that preserve much of the DBN structure. Unfortunately, decision trees cannot compactly represent certain types of value functions, especially those that involve disjunctive value assessments. For instance, if the proposition $a \vee b \vee c$ describes a group of states that have a specific value, a decision tree must duplicate that value three times (and in SPI the value is computed three times). Furthermore, if the proposition describes not a single value, but rather identical subtrees involving other variables, the entire subtrees must be duplicated. Decision graphs offer the advantage that identical subtrees can be merged into one. As we demonstrate in this paper, this offers considerable computational advantages in certain natural classes of problems. In addition, highly optimized ADD manipulation software can be used in the implementation of value iteration.

Notwithstanding such advances, large MDPs often have prohibitive computational requirements (time and space) where the generation of optimal policies is concerned. Thus, to deal with resource limitations, one must relax the constraint of *optimality* while still maintaining reasonably decision quality. This can be accomplished by reducing the "level of detail" in the representation and ignoring certain problem features that have little impact on decisions. Portions of the state space which are very similar in value can be approximated by treating them as a single state. Approximations of this kind have been examined in the context of tree structured approaches [7], and here this research is extended by applying them to ADD structured solution methods.

Decision diagrams are ideally suited to the approximation task, as similar values can be easily merged throughout the diagram. Specifically, each element in some set of values can be replaced with a *range* of values which includes all members. This new ranged value can only occur once in an ADD, so all parents of the original values will share a common leaf in the newly created ADD. All parent nodes became isomorphic as a result of this merging, and themselves will be merged, with the disjunction created by this merging propagating upward in the ADD, until an ADD of minimal size is constructed. This stands in contrast to tree structured methods, in which only sibling nodes can be easily merged. Detecting further isomorphic structure which is distant in the tree is difficult, and is accomplished, for example, by finding variable orderings which bring the isomorphic structures closer together. However, finding such variable orderings is expensive, and is generally approached heuristically, for example, by using the *information gain criterion* explored in the decision tree literature [7, 20]. The ADD approach obviates the need for such variable reordering. Nevertheless, variable orders play an important role in the sizes of any boolean function represented as an ADD, and can greatly reduce the sizes of intermediate value functions, and hence increase the performance of optimal or approximate policy generation.

We develop two approximation methods for ADD-structured value functions, and apply them to the value diagrams generated during dynamic programming. The result is a near-optimal value function, which induces a near-optimal policy. Our goal is to find approximations that make dynamic programming feasible for large state spaces, while generating policies with values as close as possible to the optimal. Our results compare the two approximation methods using different resource bounds, and we show how the performance and errors change as functions of the approximation strength. We also compare three variable reordering techniques, highlighting one that yields large performance increases for both optimal and approximate policy generation.

The remainder of the paper is organized as follows. We provide a cursory review of MDPs and value iteration in Section 2. In Section 3, we review ADDs and describe our ADD representation of MDPs. In Section 4, we describe a conceptually straightforward version of SPUDD, a value iteration algorithm that uses an ADD value function representation, and describe the key differences with the SPI algorithm. We also describe several optimizations that reduce both the time and memory requirements of SPUDD. This is followed in Section 5 by a development of our approximation methods and in Section 5.4 by an examination of variable reordering techniques. Empirical results on a class of process planning examples are described in Section 6. We are able to solve some very large MDPs exactly (up to 34 billion states) and we show that the ADD value function representation is considerably smaller than the corresponding decision tree in most instances. This illustrates that natural problems have the type of disjunctive structure that can be exploited by decision graph representations. Section 6.2 demonstrates our approximate value iteration methods on the same class of problems, showing that significant computational resource gains can be acheived with only modest value sacrifices. We examine the effects of three variable reordering techniques in Section 6.5, and show that our methods can be made robust to variable order specifications with only slight computation time losses. We conclude in Section 7 with a discussion of future work in using ADDs for DTP.

2 Markov Decision Processes

We assume that the domain of interest can be modeled as a fully-observable MDP [2, 19] with a finite set of states $\mathcal{S}$ and actions $\mathcal{A}$. Actions induce stochastic state transitions, with $\Pr(s, a, t)$ denoting the probability with which state t is reached when action a is executed at state s. We also assume a real-valued reward function R, associating with each state s its immediate utility $R(s)$.[1]

A *stationary policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$ describes a particular course of action to be adopted by an agent, with $\pi(s)$ denoting the action to be taken in state s. We assume that the agent acts indefinitely (an infinite horizon). We compare different policies by adopting an *expected total discounted reward* as our optimality criterion wherein future rewards are discounted at a rate $0 \leq \beta < 1$, and the value of a policy is given by the expected total discounted reward accrued. The expected value $V_\pi(s)$ of a policy π at a given state s satisfies [19]:

$$V_\pi(s) = R(s) + \beta \sum_{t \in \mathcal{S}} Pr(s, \pi(s), t) \cdot V_\pi(t) \tag{1}$$

A policy π is *optimal* if $V_\pi \geq V_{\pi'}$ for all $s \in \mathcal{S}$ and policies π'. The *optimal value function* V^* is the value of any optimal policy.

Value iteration [2] is a simple iterative approximation algorithm for constructing optimal policies. It proceeds by constructing a series of n-stage-to-go value functions V^n. Setting $V^0 = R$, we define

$$V^{n+1}(s) = R(s) + \max_{a \in \mathcal{A}} \left\{ \beta \sum_{t \in \mathcal{S}} Pr(s, a, t) \cdot V^n(t) \right\} \tag{2}$$

The sequence of value functions V^n produced by value iteration converges linearly to the optimal value function V^*. For some finite n, the actions that maximize Equation 2 form an optimal policy, and V^n approximates its value. A commonly used stopping criterion specifies termination of the iteration procedure when

$$||V^{n+1} - V^n|| < \frac{\epsilon(1-\beta)}{2\beta} \tag{3}$$

(where $||X|| = \max\{|x| : x \in X\}$ denotes the supremum norm). This ensures that the resulting value function V^{n+1} is within $\frac{\epsilon}{2}$ of the optimal function V^* at any state, and that the resulting policy is ϵ-optimal [19].

[1] We ignore actions costs for ease of exposition. These impose no serious complications.

3 ADDs and MDPs

Algebraic decision diagrams (ADDs) [1] are a generalization of BDDs [10], a compact, efficiently manipulable data structure for representing boolean functions. These data structures have been used extensively in the VLSI CAD field and have enabled the solution of much larger problems than previously possible. In this section, we will describe these data structures and basic operations on them and show how they can be used for MDP representation.

3.1 Algebraic Decision Diagrams

A BDD represents a function $\mathcal{B}^n \rightarrow \mathcal{B}$ from n boolean variables to a boolean result. Bryant [10] introduced the BDD in its current form, although the general ideas have been around for quite some time (e.g., as branching programs in the theoretical computer science literature). Conceptually, we can construct the BDD for a boolean function as follows. First, build a decision tree for the desired function, obeying the restrictions that along any path from root to leaf, no variable appears more than once, and that along every path from root to leaf, the variables always appear in the same order. Next, apply the following two reduction rules as much as possible: (1) merge any duplicate (same label and same children) nodes; and (2) if both child pointers of a node point to the same child, delete the node because it is redundant (with the parents of the node now pointing directly to the child of the node). The resulting directed, acyclic graph is the BDD for the function.[2] In practice, BDDs are generated and manipulated in the fully-reduced form, without ever building the decision tree.

ADDs generalize BDDs to represent real-valued functions $\mathcal{B}^n \rightarrow \mathcal{R}$; thus, in an ADD, we have multiple terminal nodes labeled with numeric values. More formally, an ADD denotes a function as follows:

1. The function of a terminal node is the constant function $f() = c$, where c is the number labelling the terminal node.
2. The function of a nonterminal node labeled with boolean variable X_1 is given by

$$f(x_1 ... x_n) = x_1 \cdot f_{then}(x_2 \ldots x_n) + \overline{x_1} \cdot f_{else}(x_2 \ldots x_n)$$

   where boolean values x_i are viewed as 0 and 1, and f_{then} and f_{else} are the functions of the ADDs rooted at the *then* and *else* children of the node.

BDDs and ADDs have several useful properties. First, for a given variable ordering, each distinct function has a unique reduced representation. In addition, many common functions can be represented compactly because of isomorphic-subgraph sharing. Furthermore, efficient algorithms (e.g., depth-first search with a hash table to reuse previously computed results) exist for most common operations, such as addition, multiplication, and maximization. For example, Figure 1 shows a computation of the maximum of two ADDs. Finally, because BDDs and ADDs have been used extensively in other domains, very efficient implementations are readily available. As we will see, these properties make ADDs an ideal candidate to represent structured value functions in MDP solution algorithms.

[2] We are describing the most common variety of BDD. Numerous variations exist in the literature.

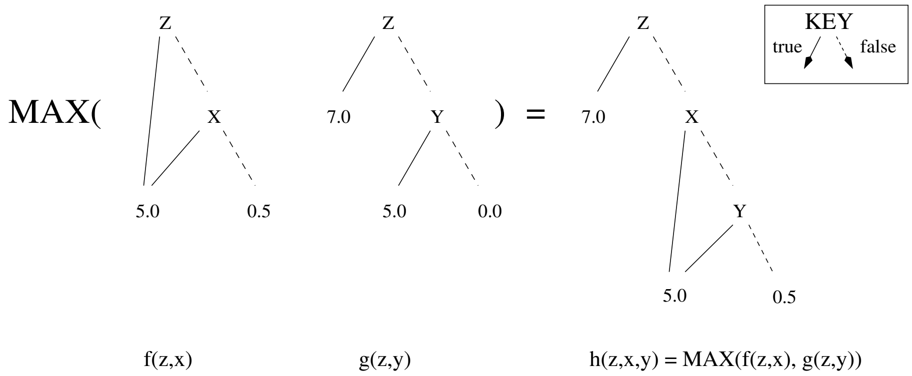


Figure 1: Simple ADD maximization example

3.2 ADD Representation of MDPs

We assume that the MDP state space is characterized by a set of variables $\mathbf{X} = \{X_1, \cdots, X_n\}$. Values of variable X_i will be denoted in lowercase (e.g., x_i). We assume each X_i is boolean, as required by the ADD formalism, though we discuss multi-valued variables in Section 5. Actions are often most naturally described as having an effect on specific variables under certain conditions, implicitly inducing state transitions. DBN action representations [13, 8] exploit this fact, specifying a local distribution over each variable describing the (probabilistic) impact an action has on that variable.

A DBN for action a requires two sets of variables, one set $\mathbf{X} = \{X_1, \cdots, X_n\}$ referring to the state of the system before action a has been executed, and $\mathbf{X}' = \{X'_1, \cdots, X'_n\}$ denoting the state after a has been executed. Directed arcs from variables in $\mathbf{X}$ to variables in $\mathbf{X}'$ indicate direct causal influence and have the usual semantics [17, 13].[3] The conditional probability table (CPT) for each post-action variable X'_i defines a conditional distribution $P^a_{X'_i}$ over X'_i—i.e., a's effect on X_i—for each instantiation of its parents. This can be viewed as a function $P^a_{X'_i}(X_1 \ldots X_n)$, but where the function value (distribution) depends only on those X_j that are parents of X'_i. No quantification is provided for pre-action variables X_i: since the process is fully observable, we need only use the DBN to predict state transitions. We require one DBN for each action $a \in \mathcal{A}$.

In order to illustrate our representation and algorithm, we introduce a simple adaptation of a process planning problem taken from [14]. The example involves a factory agent which has the task of connecting two objects A and B. Figure 2(a) illustrates our representation for the action *bolt*, where the two parts are bolted together. We see that whether the parts are successfully connected, $\mathbf{C}$, depends on a number of factors, but is independent of the state of variable $\mathbf{P}$ (*painted*). In contrast, whether part A is punched, $\mathbf{APU}$, after bolting depends only on whether it was punched before bolting.

[3] We ignore the possibility of arcs among post-action variables, disallowing correlations in action effects. See [5] for a treatment of dynamic programming when such correlations exist.

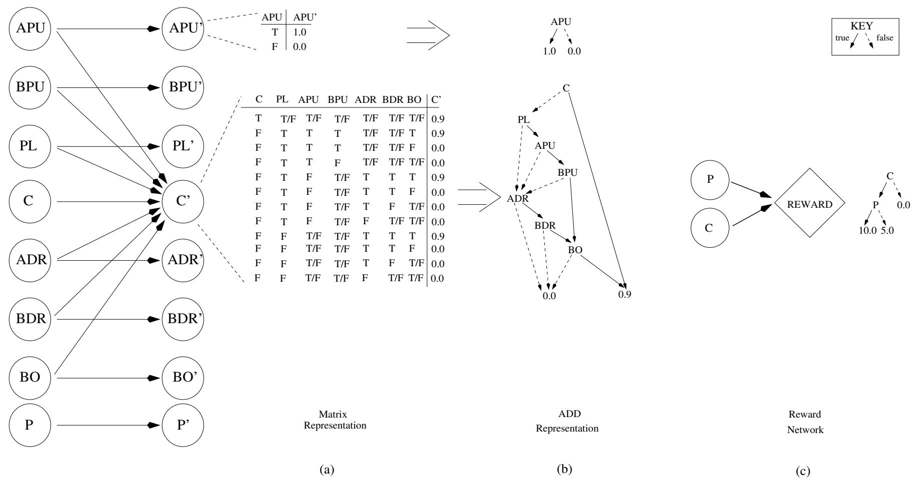


Figure 2: Small FACTORY example: (a) action network for action *bolt*; (b) ADD representation of CPTs (action diagrams); and (c) immediate reward network and ADD representation of the reward table.

Rather than the standard, locally exponential, tabular representation of conditional probability tables, we use ADDs to capture regularities in the CPTs (i.e., to represent the functions $P^a_{X'_i}(X_1 \ldots X_n)$). This type of representation exploits context-specific independence in the distributions [9], and is related to the use of tree representations [8] and rule representations [18] of CPTs in DBNs. Figure 2(b) illustrates the ADD representation of the CPT for two variables, $\mathbf{C}'$ and $\mathbf{APU}'$. While the distribution over $\mathbf{C}'$ is a function of its seven parent variables, this function exhibits considerable regularity, readily apparent by inspection of the table, which is exploited by the ADD. Specifically, the distribution over $\mathbf{C}'$ is given by the following formula:

$$
\begin{aligned}
P^{bolt}_{\mathbf{C}'}(\mathbf{C}, \mathbf{PL}, \mathbf{APU}, \mathbf{BPU}, \mathbf{ADR}, \mathbf{BDR}, \mathbf{BO}) = \\
[\mathbf{C} + \overline{\mathbf{C}}[(\mathbf{PL} \cdot \overline{\mathbf{APU}} + \overline{\mathbf{PL}}) \cdot \mathbf{ADR} \cdot \mathbf{BDR} \\
+ \mathbf{PL} \cdot \mathbf{APU} \cdot \mathbf{BPU}] \cdot \mathbf{BO}] \cdot 0.9
\end{aligned}
$$

(we ignore the zero entries). Similarly, the ADD for $\mathbf{APU}'$ corresponds to:

$$P^{bolt}_{\mathbf{APU}'}(\mathbf{APU}) = \mathbf{APU} \cdot 1.0$$

Reward functions can be represented similarly. Figure 2(c) shows the ADD representation of the reward function for this simple example: the agent is rewarded with 10 if the two objects are connected and painted, with a smaller reward of 5 when the two objects are connected but not painted, and is given no reward when the parts are not connected. The reward function, $R(X_1, \ldots, X_n)$, is simply

$$R(\mathbf{C}, \mathbf{P}) = \mathbf{C} \cdot \mathbf{P} \cdot 10.0 + \mathbf{C} \cdot \overline{\mathbf{P}} \cdot 5$$

This example action illustrates the type of structure that can be exploited by an ADD representation. Specifically, the CPT for $\mathbf{C}'$ clearly exhibits disjunctive structure, where a variety of distinct conditions *each* give rise to a specific probability of successfully connecting two parts. While this ADD has seven internal nodes and two leaves, a tree representation for the same CPT requires eleven internal nodes and twelve leaves. As we will see, this additional structure can be exploited in value iteration. Note also that the standard matrix representation of the CPT requires 128 parameters.

ADDs are often much more compact than trees when representing functions, but this is not always the case. The ordering requirement on ADDs means that certain functions can require an exponentially larger ADD representation than a well-chosen tree; similarly, ADDs can be exponentially smaller than trees. Our initial results suggest that such pathological examples are unlikely to arise in most problem domains (see Section 6), and that ADDs offer an advantage over decision trees.

4 Value Iteration using ADDs

In this section, we present an algorithm for optimal policy construction that avoids the explicit enumeration of the state space. SPUDD (stochastic planning using decision diagrams) implements classical value iteration, but uses ADDs to represent value functions and CPTs. It exploits the regularities in the action and reward networks, made explicit by the ADD representation described in the previous section, to discover regularities in the value functions it constructs. This often yields substantial savings in both space and computational time. We first introduce the algorithm in a conceptually clear way, and then describe certain optimizations.

OBDDs have been explored in previous work in AI planning [11], where universal plans (much like policies) are generated for nondeterministic domains. The motivation in that work, avoiding the combinatorial explosion associated with state space enumeration, is similar to ours; but the details of the algorithms, and how the representation is used to represent planning domains, is quite different.

4.1 The Basic SPUDD Algorithm

The SPUDD algorithm, shown in Figure 3, implements a form of value iteration, producing a sequence of value functions $V^0, V^1, \cdots$ until the termination condition is met. Each i stage-to-go value function is represented as an ADD denoted $V^i(X_1, \ldots, X_n)$. Since $V^0 = R$, the first value function has an obvious ADD representation. The key insight underlying SPUDD is to exploit the ADD structure of V^i and the MDP representation itself to discover the appropriate ADD structure for V^{i+1}. Expected value calculations and maximizations are then performed at each terminal node of the new ADD rather than at each state.

Given an ADD for V^i, Step 3 of SPUDD produces V^{i+1}. When computing V^{i+1}, the function V^i is viewed as representing values at *future* states, after a suitable action has been performed with $i + 1$ stages remaining. So variables in V^i are first replaced by their *primed*, or post-action, counterparts (Step 3(a)), referring to the state with i stages-to-go; this prevents them from being confused with unprimed variables that

1. **Set** $V^0 = R$ where R is the immediate reward diagram; set $i = 0$
2. Create *dual* action diagrams, $Q^a_{X'_i}(X'_i, X_1, \ldots X_n)$ for each $a \in \mathcal{A}$, and for each $X'_i \in \mathbf{X'}$
3. **Repeat** until $||V^{i+1} - V^i|| < \frac{\epsilon(1-\beta)}{2\beta}$
   (a) **Swap** all variables in X^i with *primed* versions to create X'^i
   (b) **For** all $\mathbf{a} \in \mathcal{A}$
   Set $temp = V'^i$
   For all *primed variables*, X'_j in V'^i
   $temp = temp * Q^a_{X'_j}$
   Set $temp =$ Sum the sub-diagrams of $temp$
   over the *primed* variable X'_j
   End For
   Multiply the result by discounting factor β
   and add R to obtain V^i_a
   End For
   (c) **Maximize** over all V^i_a's to create V^{i+1}.
   (d) **Increment** i
   End Repeat
4. Perform one more iteration and assign to each terminal node the actions a which contributed the value in the value ADD at that node; this yields the ϵ-*optimal policy* ADD, π^*. Note that terminal nodes which have the same values for multiple actions are assigned all possible actions in π^*.
5. Return the value diagram V^{i+1} and the optimal policy π^*.

Figure 3: SPUDD algorithm

refer to the state with $i+1$ stages-to-go. Figure 4(a) shows the zero stage-to-go primed value diagram, V'^0, for our simple example.

For each action a, we then compute an ADD representation of the function V^{i+1}_a, denoting the expected value of performing action a with $i+1$ stages to go given that V^i dictates i stage-to-go value. This requires several steps, described below. First, we note that the ADD-represented functions $P^a_{X'_i}$, taken from the action network for a, give the (conditional) probabilities that variables X'_i are made *true* by action a. To fit within the ADD framework, we introduce the *negative action diagrams*

$$\overline{P^a_{X'_i}}(X_1, \ldots, X_n) = (1 - P^a_{X'_i}(X_1, \ldots, X_n))$$

which gives the probability that a will make X'_i false. We then define the *dual action diagrams* $Q^a_{X'_i}$ as the ADD rooted at X'_i, whose *true* branch is the action diagram $P^a_{X'_i}$ and whose *false* branch is the negative action diagram $\overline{P^a_{X'_i}}$:

$$\begin{aligned} Q^a_{X'_i}(X'_i; X_1, \ldots X_n) &= X'_i \cdot P^a_{X'_i}(X_1, \ldots X_n) + \\ & \quad \overline{X'_i} \cdot \overline{P^a_{X'_i}}(X_1, \ldots X_n)) \end{aligned} \tag{4}$$

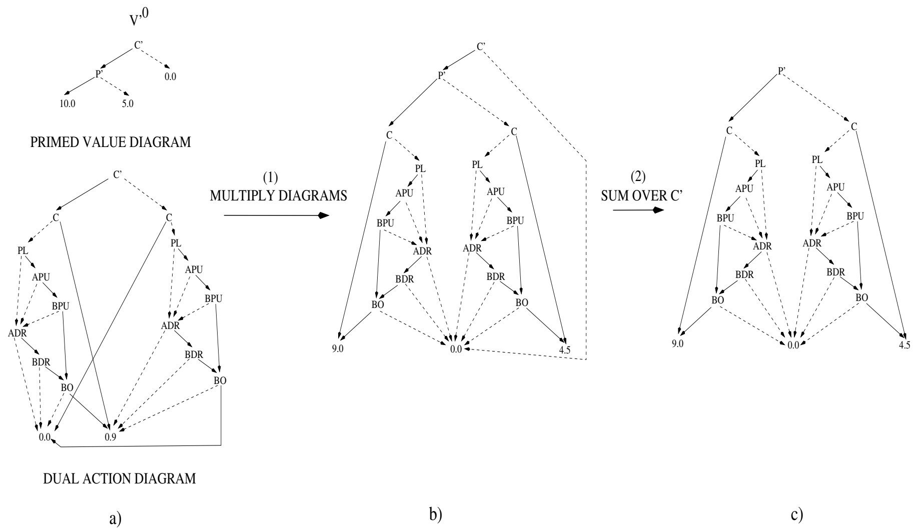


Figure 4: First Bellman backup for the *Value Iteration using ADDs* algorithm. (a) *0-stage-to-go primed* value diagram, and dual action diagram for variable $\mathbf{C}'$, $Q^a_{C'}$ (b) Intermediate result after multiplying V'^0 with $Q^a_{C'}$. (c) Intermediate result after quantifying over $\mathbf{C}'$.

Intuitively, $Q^a_{X'_i}(x'_i; x_1, \ldots x_n)$ denotes $P(X'_i = x'_i | X_1 = x_1, \cdots, X_n = x_n)$ (under action a). Figure 4(a) shows the dual action diagram for the variable **C'** from the example in Figure 2(b).

In order to generate V_a^{i+1}, we must, for each state s, combine the i stage-to-go value for each state t with the probability of reaching t from s. We do this by multiplying, in turn, the dual action diagrams for each variable X'_j by V'^i and then eliminating X'_j by summing over its values in the resultant ADD. More precisely, by multiplying $Q^a_{X'_j}$ by V'^i, we obtain a function $f(X'_1, \cdots, X'_n, X_1, \cdots X_n)$ where

$$\begin{aligned} f(x'_1, \cdots, x'_n, x_1, \cdots x_n) &= \\ &V'^i(x'_1, \cdots, x'_n) P(x'_j | x_1, \ldots x_n) \end{aligned}$$

(assuming transitions induced by action a). This intermediate calculation is illustrated in Figure 4(b), where the dual diagram for variable $\mathbf{C}'$ is the first to be multiplied by V'^0. Note that $\mathbf{C}'$ lies at the root of this ADD. Once this function f is obtained, we can eliminate dependence of future value on the specific value of X'_j by taking an expectation over both of its truth values. This is done by summing the left and right subgraphs of the ADD for f, leaving us with the function

$$\begin{aligned} g(X'_1, \cdots, X'_{j-1}, X'_{j+1}, \cdots, X'_n, X_1, \ldots X_n) &= \\ &\sum_{x'_j} V'^i(X'_1, \cdots, x'_j, \cdots, X'_n) P(x'_j | X_1, \ldots X_n) \end{aligned}$$

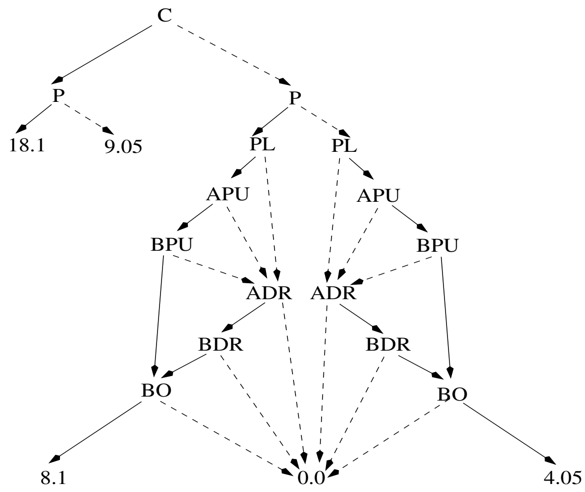


Figure 5: Resulting *1-stage-to-go* value diagram for action *bolt*, V^1_{bolt}.

This is illustrated in Figure 4(c), where the variable $\mathbf{C}'$ is eliminated. This ADD denotes the expected *future* value (or 0 stage-to-go value) as a function of the parents of $\mathbf{C}'$ with 1 stage-to-go and all post-action variables except $\mathbf{C}'$ with 0 stages-to-go.

This process is repeated for each post-action variable X'_j that occurs in the ADD for V'^i: we first multiply $Q^a_{X'_j}$ into the intermediate value ADD, then eliminate that variable by taking an expectation over its values. Once all primed variables have been eliminated, we are left with a function

$$
\begin{aligned}
&h(X_1, \cdots, X_n) = \\
&\quad \sum_{x'_1, \cdots, x'_n} V'^i(x'_1, \cdots, x'_n) P(x'_1 | X_1, \ldots X_n) \cdots \\
&\quad P(x'_n | X_1, \ldots X_n)
\end{aligned}
$$

By the independence assumptions embodied in the action network, this is precisely the expected future value of performing action a. By adding the reward ADD R to this function, we obtain an ADD representation of V^{i+1}_a. Figure 5 shows the result for our simple example. The remaining primed variable $\mathbf{P}'$ in Figure 4(c) has been removed, producing $V^1_{\mathbf{bolt}}$ using a discount factor of 0.9. Finally, we take the maximum over all actions to produce the V^{i+1} diagram. Given ADDs for each V^{i+1}_a, this requires simply that one construct the ADD representing $\max_{a \in \mathcal{A}} V^{i+1}_a$.

The stopping criterion in Equation 3 is implemented by comparing each pair of successive ADDs, V^{i+1} and V^i. Once the value function has converged, the ϵ-optimal policy, or *policy ADD*, is extracted by performing one further dynamic programming backup, and assigning to each terminal node the actions which produced the maximizing value. Since each terminal node represents some state set of states S, the set of actions thus determined are each optimal for any $s \in S$.

4.2 Optimizations

1. **Set** $BIGADD$ = user-specified limit for size of graphs
   Set $temp$ = ADD constant 1; $k = 1; \quad m = 0 \quad size = 0$
2. **While** $k <$ number of variables
   $j = k; i_j = k$
   While $size < BIGADD$
   Set $temp = temp * Q^a_{v'_k}$
   $size =$ no. of internal nodes in $temp$
   $k = k + 1$
   End While
   $P^a(v'_j, \ldots, v'_{k-1}, v_1, \ldots, v_n) = temp$
   $m = m + 1$
   End While
3. **Repeat** until $||V^{i+1} - V^i|| < \frac{\epsilon(1-\beta)}{2\beta}$

   $\vdots$

   (c) **For** all $\mathbf{a} \in \mathcal{A}$
   Set $last\_level = m$
   Call pRew(V'^n,P^a,0,1)

procedure **pRew** ($value$,$action$,var,$next\_level$)

If $var > i_{next\_level}$
If $var > i_{last\_level}$
$result = value$; $last\_level = last\_level - 1$
Else
$temp$ = **pRew** ($value$,$action$,var,$next\_level + 1$)
$temp = temp * act$
$result =$ sum all sub-diagrams of $temp$ over *primed* variables, v'_j $j > i_{last\_level}$
else
$tempT$ = **prRew**(then($value$), $action$,$level + 1$,$next\_level$)
$tempE$ = **prRew**(else($value$), $action$,$level + 1$,$next\_level$)
$result$ = tree rooted at $v'_{i_{level}}$ with *then,else* branches: $tempT$,$tempE$, resp.
return result

Figure 6: Modified SPUDD algorithm

The algorithm as described in the last section, and as shown in Figure 3, suffers from certain practical difficulties which make it necessary to introduce various optimizations in order to improve efficiency with respect to both space and time. The problems arise in Step 3(b) when V'^i is multiplied by the dual action diagrams Q^a. Since there are potentially n primed variables in the ADD for V'^i and n unprimed variables in the ADD for Q^a, there is an intermediate step in which a diagram is created with (potentially) up to $2n$ variables. Although this will not be the case in general, it was deemed necessary to modify the method in order to deal with the possibility of this problem arising. Furthermore, a large computational overhead is introduced by re-calculating the joint probability distributions over the primed variables at each iteration. In this section, we first discuss optimizations for dealing with space, followed by a method for optimizing computation time.

The increase in the diagram size during Step 3(b) of the algorithm can be countered by approaching the multiplications and sums slightly differently. Instead of blindly multiplying the V'^i by the dual action diagram for the variable at the root of V'^i, we can traverse the ADD for V'^i to the level of the last variable in the ADD ordering, then multiply and sum the sub-diagrams rooted at this variable by the corresponding dual diagram. This process will only remove the dependency of the V'^i on a primed variable for a given branch, and will therefore only introduce a single diagram of n unprimed variables at a leaf node of V'^i. By recursively carrying out this procedure using the structure of the ADD for V'^i, the intermediate stages never grow too large. Essentially, the additional unprimed variables are introduced only at specific points in the ADD and the corresponding primed variable immediately eliminated—this is much like the tree-structured dynamic programming algorithm of [8].

Unfortunately, this method requires a great deal of unnecessary, repeated computation. Since the action diagrams for a given problem do not change during the generation of a policy, the joint probability distribution $Pr(s, a, t)$ from Equation 2 could be pre-computed. In our case, this means we could take the product of all dual action diagrams for a given action a, as shown in Equation 5 below, prior to a specific value iteration. We refer to this product diagram, P^a, as the *complete* action diagram for action a:

$$P^a(X'_1, \ldots, X'_n, X_1, \ldots, X_n) = \prod_{i=1}^{n} Q^a_{X'_i}(X'_i; X_1, \ldots, X_n) \qquad (5)$$

The resulting function P^a provides a representation of the state transition probabilities for action a. This explicit P^a function could then be multiplied by the V'^n during Step 3 of the algorithm, and then primed variables eliminated. Although this may lead to a substantial savings in computation time, it will again generate diagrams with up to $2n$ variables.

As a compromise, we implemented a method where the space-time trade-off can be addressed explicitly. A "tuning knob" enables the user to find a middle ground between the two methods mentioned above. We accomplish this by pre-computing only subsets of the complete action diagram. That is, we break the large diagram up into a few smaller pieces. The set of variables $(X_1, \ldots, X_n)$ is divided into m subsets, preserving the total ordering (e.g., $[X_1, \ldots, X_{i_1}]$, $[X_{i_1+1}, \ldots, X_{i_2}]$, $\ldots$, $[X_{i_m}, \ldots X_n]$). Then, the complete action diagrams are pre-computed for each of these subsets (e.g.: $(P^a(X'_{i_j}, \ldots, X'_{i_{j+1}}, v_1, \ldots, X_n))$. Step 3(b) of the algorithm must be modified as shown in Figure 6. The primed value diagram V'^i is traversed to the top of the second level (i_1+1), and the procedure is carried out recursively on each sub-diagram rooted at variables X'_{i_1+1}. If a level is reached with no variables below it, then the sub-diagram rooted at each variable X'_{i_m} of V'^i is multiplied with the corresponding subset of the complete action diagram, $P^a(X'_{i_m}, \ldots, X'_n, X_1, \ldots, X_n)$, and summed over primed variables X'_k, $k > i_m$. In this way, the diagrams are kept small by making sure that enough elimination occurs to balance the effects of multiplying by complete action diagrams. The space and time requirements can then be controlled by the number of subsets the complete action diagrams are broken into. In theory, the more subsets, the smaller the space requirements and the larger the time requirements. Although we have been able to produce substantial changes in the space and time requirements of the algorithm using this tuning knob, its effects are still unclear. At present, we choose the m subsets of variables by simply building the complete action diagrams according to some variable ordering until they reach a user-defined size limit, at which point we start on the next subset. We note that this space-time tradeoff bears some resemblance to the space-time tradeoffs that arise in probabilistic inference algorithms like variable elimination [15].

This revised procedure (Figure 6) has a small inefficiency, as our results in the next section will show. Since we are pre-computing subsets of the complete action diagrams, any variables which are included in the domain, but are not relevant to its solution, will be included in these pre-computed diagrams. This will increase the size of the intermediate representations and will add overhead in computation time. It is important to be able to discard them, and to only compute the policy over variables that are relevant to the value function and policy [8]. A possible way to deal with these types of variables in our algorithm would be to progressively build the complete action diagrams during the iterative procedure. In this way, only the variables relevant to the domain would be added.

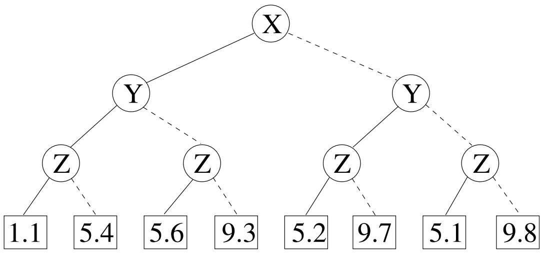
5 Approximating Value Functions

While structured solution techniques offer many advantages, the exact solution of MDPs in this way can only work if there are "few" distinct values in a value function. Even if a DBN representation shows little dependence among variables from one stage to another, the influence of variables tends to "bleed" through a DBN over time, and many variables become relevant to predicting value. Thus, even using structured methods, we must often relax the optimality constraint and generate only approximate value functions, from which will hopefully arise near optimal policies. It is generally the case that many of the values distinguished by DP are similar. Replacing such values with intermediate values leads to size reduction, while not significantly affecting the precision of the value diagrams.

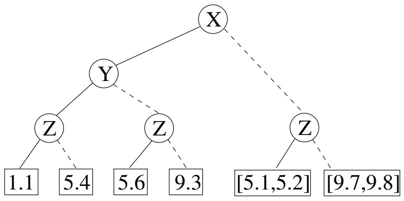
The remainder of this section will discuss the methods we have developed for approximating value functions, including an examination of variable reordering techniques for ADDs. It will close by presenting value iteration with such approximated value diagrams.

5.1 Decision Diagrams and Approximation

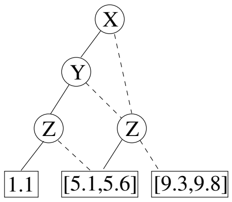
Consider the value diagram shown in Figure 7(a), which has eight distinct values as shown. The value of each state s is represented as a pair $[l, u]$, where the lower, l, and upper, u, bounds on the values are both represented. The *span* of a state, s, is given by $span(s)$=$u - l$. Point values are represented by setting u=l, and have zero *span*. Now suppose that the diagram in Figure 7(a) exceeds resource limits, and a reduction in size is necessary to continue the value iteration process. If we choose to no longer distinguish values which are within 0.1 of each other, the diagram in Figure 7(b) results,



(a)



(b) 0.1



(c) 0.5

Figure 7: Approximation applied to (a) original value diagram with errors of (b) 0.1 and (c) 0.5.

which saves two internal nodes. A further reduction (to 0.5) gives the diagram shown in Figure 7(c). The states which had proximal values have been merged, where merging a set of states $s_1, s_2, \ldots, s_n$ with values $[l_1, u_1], \ldots, [l_n, u_n]$, results in an aggregate state, t, with a *ranged* value $[\min(l_1, \ldots, l_n), \max(u_1, \ldots, u_n)]$. The estimate of the true value of the states represented by this range is then given by the mid-point of the range, since the mid-point has a minimal error of $span(t)/2$. We refer to the *combined span* of the states $s_1, \ldots, s_n$ as the *span* of the pair that would result from merging all the states, $cspan(s_1, \ldots, s_n) = \max(u_1, \ldots, u_n) - \min(l_1, \ldots, l_n)$. The span of V is the maximum of all spans in the value diagram, $span(V) = \max_{s \in \mathcal{S}} span(s)$, and therefore the maximum error in V is simply $span(V)/2$ [7]. The *extent* of a value diagram V is the *combined span* of the portion of the state space which it represents, $extent(V) = cspan(s|s \in V)$. For example, the span of the diagram in Figure 7(c) is 0.5, whereas its extent is 8.7.

If a tree-structured representation was being used [7], the original value function shown in Figure 7(a) cannot be reduced from its original form simply.[4]

Consider the value tree shown in Figure 8(a). In order to reduce this tree to its smaller form shown on the right (Figure 8(c)), we can, if using a tree representation, examine the structure and notice that the subtrees of the root node are identical. Then, we can simply discard the root node (or connect its parent to one of its children in the more general case), and discard one of the child subtrees, as shown in Figure 8(b-1). However, detecting isomorphic subtrees is a difficult task, and is generally not

[4]In fact, the tree in Figure 7(b) is in its most reduced form using a tree-structured representation. There is no other variable ordering which would result in a smaller tree

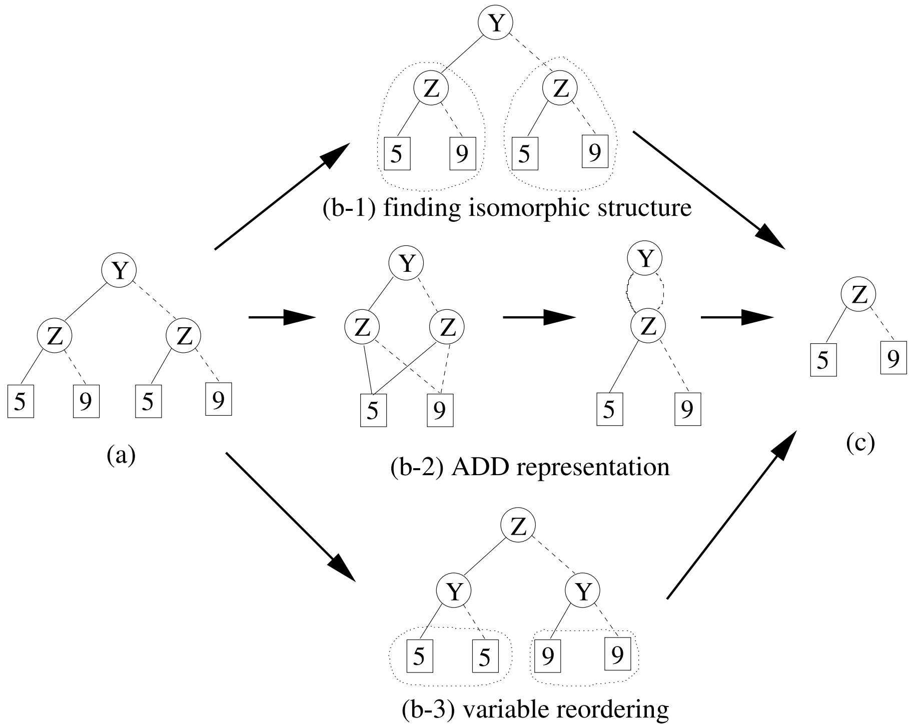


Figure 8: Methods of reducing the sizes of value functions (a) initial value tree (c) final reduced form (b-1) isomorphic structure is found by exhaustive search, (b-2) the ADD representation incrementally adjusts the structure, (b-3) variable reordering enables merging in a simple step.

attempted in practice. An ADD representation, on the other hand, makes this task significantly easier, as shown in Figure 8(b-2).

Further reduction requires examining the structure of the tree and locating the isomorphic subtrees which can be merged, a task which is rendered significantly easier with the use of ADDs. Structure in an ADD is merged incrementally, starting with the leaf nodes, and propagating up the diagram. Thus, only one level of the diagram is considered at a time, and the cost of the operation is reduced [10]. An alternative when using tree structured representations is to look for changes in variable ordering which will bring isomorphic structures closer together in the tree, thus enabling their merging in a simple step (Figure 8(b-3)) [7, 23]. However, finding an optimal ordering is also a difficult task, and heuristic methods are typically used: these do not guarantee that isomorphic structures will be found [20].

ADD-structured value functions can be leveraged by approximation techniques because approximations can always be performed directly without pre-processing techniques such as variable reordering. Of course, variable reordering can still play an important computational role in ADD-structured methods, but are not needed for *dis-*

covering approximations. One could attempt to reorder the variables in the ADD in Figure 7(c), which in this case yields no smaller diagrams, but which in general may further reduce the size. We will not encounter the situation of a necessary variable reordering step using ADDs, as is the case using trees.

When generating approximate value functions and policies for an MDP, available resources might dictate that ADDs be kept below some fixed size; in contrast, decision quality might require errors below some fixed value. In these cases one is looking for the most accurate or the smallest diagram, respectively, for the specified bounds. These two views will be referred to as *max_size* and *max_error* modes, respectively. The following section examines two methods for approximating value diagrams under these two restrictions.

5.2 Methods

Our objective is a reduction in the size of an ranged value ADD by replacing some leaves with larger ranges. If a maximum error bound is specified by the user ($\mathtt{m}ax\_error$ mode), then the goal is to replace all leaves which have combined spans less than this error bound by a single leaf. If a maximum size bound is specified ($\mathtt{m}ax\_size$ mode), then the goal is to merge leaves with minimum combined spans until the size falls below the bound. We have examined two approximation methods, which we refer to as the *all-pairs* and *round-off* methods.

The *all-pairs* method, when used in *max_error* mode, examines sets of leaves in a value diagram V, and finds those which have combined spans less than some limit, as shown in Figure 9(a). It starts with a leaf $[l, u]$ in V, and finds the set of all leaves $[l_i, u_i]$ such that the combined span of $[l_i, u_i]$ with $[l, u]$ is less than the specified error, `max_error`, and merges all leaves in this set. Repeating this process until no more merges are possible gives the desired result. When used in *max_size* mode, as shown in Figure 9(b), the *all-pairs* method examines all pairs of leaves in V, merging the pair with the smallest combined span. This is repeated until the size of the diagram drops below `max_size`.

The *all-pairs* method must make $O(n^2)$ and $O(n^3)$ calls to **merge** in *max_error* and *max_size* mode, respectively. A **merge** step involves propagating all new structure up through the diagram, and thus will examine all nodes in the diagram in the worst case, leading to an $O(n)$ complexity [10]. Thus, the *all-pairs* method is expensive, but will find the smallest size diagram for a given error bound, or the smallest error for a given size bound.

The *round-off* method, shown in Figure 10 exploits the fact that simply reducing the precision of the values at the leaves of an ADD merges the similar values. That is, by dividing every value in V by some round-off error, ($\mathtt{r}oe$), and then rounding to integer values, all sets of leaves in V with combined spans of less than $\mathtt{r}oe$ will be merged

A ranged value $[l, u]$ will be rounded to $[\mathtt{f}loor(l), \mathtt{c}eil(u)]$, but only if both l and u fall in the same interval ($\mathtt{f}loor(l) = \mathtt{f}loor(u)$). This ensures that all newly created ranged values have spans smaller than the round-off error. The rounding-off step creates a value diagram V', in which all rounded values have spans of 1, and in which similar leaves are merged. The (reduced) structure of V' must then be applied to the original value function, V, by replacing each leaf in V' with the combined span of

$LEAF\_SET \leftarrow$ all leaves in V
while $LEAF\_SET$ not empty
    $x \leftarrow$ remove element from $LEAF\_SET$
    for all $j \in LEAF\_SET$
        if $\mathtt{c}span(x, j) < \mathtt{max\_error}$
            $X \leftarrow \mathbf{merge}(X, j)$ (also get merged in V)
            remove j from $LEAF\_SET$
        end if
    end for
end while

(a)

$PAIR\_SET \leftarrow$ all pairs of leaves in V
while $\mathtt{s}ize(V) < \mathtt{max\_size}$
    $X \leftarrow \mathbf{merge}(i, j)$,
        for $(i, j) = argmin_{(i,j) \in PAIR\_SET} \{\mathtt{c}span(i, j)\}$
    remove all pairs involving i or j from $PAIR\_SET$
    add new pairs $(X, i)\ \forall\ i \in PAIR\_SET$ to $PAIR\_SET$
end while

(b)

Figure 9: All-pairs method for (a) error bound given by $\mathtt{m}ax\_error$ and (b) size bound given by $\mathtt{m}ax\_size$. $\mathbf{merge}(i, j)$ indicates that leaves i and j become a single leaf, and this new structure gets propagated into the ADD. $\mathtt{c}span$ computes the combined span of its arguments

```
procedure roundOff(V)
    for each [l, u] ∈ leaves of V
        if floor(l) == floor(u)
            l = floor(l)
            u = ceil(l)
        end if
    end for
end procedure
```

Figure 10: Round-off method of the round-off error ($\mathtt{r}oe$).

all corresponding leaves in V. This method is correct in that it will not merge pairs of values with combined spans greater than the specified error bound. However, it is not complete as it will not merge all pairs with combined spans less than the specified bound, since the boundaries defined by the rounding process will separate values which are closer than the round-off error. For example, if rounding the two leaves $[l_1, u_1] = [0.8, 0.9]$ and $[l_2, u_2] = [1.1, 1.2]$ to within an error of $\mathtt{r}oe = 0.5$, the pairs become $[l_1, u_1] = [1.6, 1.8]$ and $[l_2, u_2] = [2.2, 2.4]$ after division by $\mathtt{r}oe$, and get rounded to $[l_1, u_1] = [1.0, 2.0]$ and $[l_2, u_2] = [2.0, 3.0]$, different intervals which will not get merged, whereas the original leaves have a combined span of only 0.4.

To approximate the value diagram V in *max_error* mode using the *round-off* method, we simply set

$$V = \mathtt{roundOff}(V/\mathtt{max\_error}),$$

where the rounding procedure `roundOff` is as described previously and is shown in Figure 10. In *max_size* mode, we can simply start with $\mathtt{r}oe = 0$ and increment until the size of the rounded diagram drops below `max_size`. A slight extension is to binary search through $\mathtt{r}oe$ values (starting with $\mathtt{r}oe = extent(V)$), which yields a more precise value of the final round-off error, and is not dependent on the increment size as is the linear search method. We used the binary search method in our experiments.

The `roundOff` method must make $O(n^2)$ calls to **merge**. Further, a binary search through values of $\mathtt{r}oe$ will take $O(log(\delta))$ steps, where δ is the *extent* of the diagram. Thus, the *round-off* method appears more efficient than the *all-pairs* method. However, we have carefully implemented the *all-pairs* method by pre-sorting the leaves according to their minimum values, l. This leads a gain in efficiency in the search for similar leaves (the *argmin* search in Figure 9(b)), which is not matched in the *round-off* method. Therefore, we expect that the worst case $O(n^3)$ behaviour will seldom occur in the *all-pairs* method.

5.3 Value Iteration with Approximate Value Functions

Approximate value iteration simply means applying an approximation technique to the *n-stage to go* value function generated at each iteration of Eq. 2. In *max_size* mode, the approximation is applied only if the diagram exceeds the size bound, whereas in *max_error* mode, the approximation is applied at every iteration to find the smallest possible value diagram for the specified error bound. The multiplication of the *ranged* value function with the *complete* action diagrams, and the maximization over all actions proceeds as described in [7]. The multiplication is applied to all u and l labels of each range separately, and the maximization over actions is performed by taking the maximum of all u labels for a state as the new u label, and the maximum of all l labels for a state as the new l label.

We adopt a conservative approach to our convergence criterion, and stop whenever the ranges of two consecutive value functions indicate that the stopping criterion (Eq. 3) *might* hold. That is, corresponding *ranged* values in two value functions V^i and V^{i+1} are considered converged if they overlap or lie within a given tolerance. This stopping criterion guarantees convergence [7].

A non-ranged value function is obtained from the ranged *n-stage to go* value function by simply assigning each value to be the mid-point of the corresponding range. This mid-point value function, V, differs in value at each state by at most $span(V)/2$ from the optimal *n-stage* value function V^n. However, this error norm cannot be used to compare the errors with different pruning strengths, since the number of iterations to convergence, and hence the scale of values in the approximate value function, will be different. Hence, it is useful to derive an error norm which can be used regardless of the number of iterations to convergence of approximate value iteration. We compare the *i-stage* error with the total *extent* of the diagram, which will quantify the error values according to the value scale. This norm, referred to as the *a -error*= $span(V)/(2 * extent(V))$, represents the normalized maximum approximation error in a diagram. Therefore, a small pruning error arising early in the value iteration will yield the same *a-error* as a larger pruning error occurring closer to convergence, when the values have increased to close to their optimal values. We also need to compare the approximate value function with the optimal value function V^* generated using Eq. 2. However, this comparison cannot be made directly since the number of iterations to convergence, and hence the scale of the values may be different. The comparison can be made by normalizing the values in both the approximate and optimal value functions by their extent, then summing the squared error of the difference between the two, and taking the average over all leaves of the value diagram. We will refer to this norm as the *ASSE*.

5.4 Variable Reordering

As previously mentioned, variable reordering can have a significant effect on the size of an ADD, but finding the variable ordering which gives rise to the smallest ADD for a boolean function is co-NP-complete [10]. We examine three reordering methods. The first two are standard for reordering variables in BDDs, Rudell's sifting algorithm and random reordering [21, 22]. The sifting algorithm considers each variable in turn. A variable is moved up and down in the order so that it takes all positions. The best position is identified and the variable is returned to that position. Random reordering simply picks pairs of variables at random and swaps them in the order. The swap is performed by a series of swaps of adjacent variables. The best order among those obtained by the series of swaps is retained [22]. The number of pairs chosen for swapping is equal to the number of variables in existence.

The last reordering method we consider arises in the decision tree induction literature, and is related to the *information gain criterion* [20]. Given a value diagram V with extent δ, each variable x is considered in turn. The value diagram is restricted first with $x = true$, and the extent δ_t and the number of leaves n_t are calculated for the restricted ADD. Similar values δ_f and n_f are found for the $x = false$ restriction. If we collapsed the entire ADD into a single node, assuming a uniform distribution over values in the resulting range gives us the entropy for the entire ADD:

$$E = \int p(v)log(p(v))dv = log(\delta), \tag{6}$$

and represents our degree of uncertainty about the values in the diagram. Splitting the values with the variable x results in two new value diagrams, with average entropy

$$E_x = \log(\delta_t)\frac{n_t}{n} + \log(\delta_f)\frac{n_f}{n},$$

where $n = n_t + n_f$. The $\log(\delta_t)$ and $\log(\delta_f)$ factors are simply the entropy of the two resulting subdiagrams, as given by Eq. 6, after variable x is abstracted. The gain in information (decrease in entropy) by splitting the values using the variable x is $\Delta E_x = E - E_x$. These information gain values are used to rank the variables, and the resulting order is applied to the diagram. This method will be referred to as the *minimum span method*.

In general, it is usually advantageous to keep variables which are strongly correlated close in the variable orderings. The most strongly related variables in our algorithms are the pairs of primed (*post-action*) and unprimed (*pre-action*) variables. These pairs are thus initially always placed together in the ordering for all our experiments, but the *sifting* and *random* reordering methods are allowed to break these pairs up.

Whereas ordering of the variables in a decision tree proceeds recursively on each subtree after the best variable is chosen for the root, the minimum span method simply ranks the variables in a single pass. It would be possible to apply a recursive method to the value ADDs, but, due to the requirement of a total order, the optimal orderings in the subtrees would have to be combined in some way. It is not clear at this time that there would be an advantage to the additional computational load of the recursion.

The sifting and random reordering methods perform variable swaps and measure the effects of the swaps on the combined sizes of all ADDs in use. On the other hand, the minimum span method only considers a single value diagram. Furthermore, the minimum span method only considers the *post-action* ($\mathbf{X}$) variables, whereas the sifting and random methods also consider the *pre-action* ($\mathbf{X}'$) variables. Clearly, the minimum span method has not yet been optimized for our application, and we expect the sifting and random methods to find better overall orderings.

6 Data and Results

The procedures described above was implemented using the *CUDD* package [22], a library of *C* routines which provides support for manipulation of ADDs. Experimental results described in this section were all obtained using a dual-processor *SUN SPARC*

Ultra 60 running at 300Mhz with 1 Gb of RAM, with only a single processor being used. The SPUDD algorithm was tested on three different types of examples, each type having MDP instances with different numbers of variables, hence a wide variety of state space sizes. The first example class consists of various adaptations of a process planning problem taken from [14]. The second and third example classes consist of synthetic problems taken from [8, 4]. These are designed to test best- and worst-case behavior of SPUDD.[5]

The first example class consists of process planning problems taken from [14], involving a factory agent which must paint two objects and connect them. The objects must be smoothed, shaped and polished and possibly drilled before painting, each of which actions require a number of tools which are possibly available. Various painting and connection methods are represented, each having an effect on the quality of the job, and each requiring tools. The final product is rewarded according to what kind of quality is needed. Rewards range from 0 to 10 and a discounting factor of 0.9 was used throughout.

The examples used here, unlike the one described in Section 3, were not designed with any structure in mind which could be taken advantage of by an ADD representation. In the original problem specification, three ternary variables were used to represent painting quality of each object (*good, poor* or *false*), and the connection quality (*good, bad* or *false*). However, as discussed above, ADDs can only represent binary variables, so that each ternary variable was expanded into two binary ones. For example, the variable *connected*, describing the type of connection between the two objects, was represented by boolean variables *connected* and *connected_well*. This expansion enlarges the state space by a factor of $4/3$ for each ternary variable so expanded (by introducing unreachable states). A number of FACTORY examples were devised, with state space sizes ranging from 55 thousand to 34 billion.

Our results are presented in three parts. The first are those using optimal value iteration as described in Section 4. The second are those using the approximate version described in Section 5. The last describes results of variable reordering experiments.

6.1 Optimal Value Iteration Results

Optimal policies were generated using SPUDD and a *structured policy iteration* (SPI) implementation for comparison purposes [8]. Results, displayed in Table 1, are presented for SPUDD running on six FACTORY examples, and for SPI running on five. SPI was not run on examples larger than *f3* with 25 boolean variables, because its estimated time and space requirements exceeded available capacity. SPI implements modified policy iteration using trees to represent CPTs and intermediate value and policy functions. SPI, however, does allow multi-valued variables—so versions of each example were tested in SPI using both ternary variables, and their binary expansion. Table 1 shows the number of ternary variables in each example, along with the total number of variables. The state space sizes of each FACTORY example are shown for both the original and the binary-expansion formulations. SPUDD was only run on the

[5] Data for these problems can be found at the Web page: `www.cs.ubc.ca/spider/staubin/Spudd/index.html`.

Ex.	State space size vars (ternary/total)	states	SPUDD - Value time (sec)	internal nodes	leaves	equiv. tree leaves	SPI - Value time (sec)	internal nodes	leaves	ratio tree:ADD nodes
f	3/14	0.55×10^5	-	-	-	-	961	6786	7882	5.6
	0/17	1.31×10^5	14	1220	246	7375	1081	9444	9445	7.7
f0	3/16	2.21×10^5	-	-	-	-	3125	16446	18991	10.3
	0/19	5.24×10^5	19	1597	246	1339	2657	21226	21227	13.3
f1	3/18	8.84×10^5	-	-	-	-	8794	32281	38103	10.4
	0/21	2.10×10^6	42	3101	327	46348	10067	44420	44421	14.3
f2	3/19	1.77×10^6	-	-	-	-	8856	32281	38103	10.4
	0/22	4.19×10^6	44	3101	327	46348	9991	44420	44421	14.3
f3	4/21	1.06×10^7	-	-	-	-	54987	140591	170010	15.3
	0/25	3.36×10^7	127	9215	357	238803	56531	186799	186800	20.3
f4	4/24	6.37×10^7	-	-	-	-	-	-	-	-
	0/28	2.68×10^8	287	22169	526	707890	-	-	-	-
f5	0/31	2.15×10^9	706	43675	1465	1520000	-	-	-	-
f6	0/35	3.44×10^{10}	3676	162182	4054	17100000	-	-	-	-

Table 1: Results for FACTORY examples.

binary-expanded versions.

The examples labelled *f1* and *f2* differ only by a single binary variable, which is not affected by any action in the domain, and which does not itself affect any other variables. Hence, the number of internal nodes resulting in Table 1 are identical for the two examples. This variable was added in order to show how structured representations like SPUDD and SPI can effectively discard variables which do not affect the problem at hand, as discussed in Section 4.2.

Running times are shown for SPUDD and SPI. However, the algorithms do not lend themselves easily to comparisons of running times, since implementation details cloud the results; so running times will not be discussed further here. The SPI results are shown in order to compare the sizes of the final value function representations, which give an indication of complexity for policy generation algorithms. However, a question arises about the variable orderings when comparing such numbers, as mentioned in Section 3. The variable ordering for SPUDD is chosen prior to runtime and remains the same during the entire process. No special techniques were used to choose the ordering, although it may be argued that good orderings could be gleaned from the MDP specification. Variable orderings within the branches of the tree structure in the SPI algorithm are determined primarily by the choice of ordering in the reward function and action descriptions [8]. Again, no special techniques were used to choose the variable ordering in SPI. Finding the optimal variable orderings in either case is a difficult problem, and we assume here that neither algorithm has an advantage in this regard. Dynamic reordering algorithms are available in CUDD, and have been implemented but not yet fully tested in SPUDD (see below).

In order to compare representation sizes, we compare the number of internal nodes in the value function representations only. This is most important when doing dynamic programming back-up steps and is a large factor in determining both running time and space requirements. Furthermore, we compare numbers from SPUDD using binary representations with numbers from SPI using binary/ternary representations in order not to disadvantage SPI, which can make use of ternary variables. We also compare both implementations using only binary variables. The *equivalent tree leaves* column in Table 1 gives the number of leaves of the totally ordered binary tree (and hence the number of internal nodes) that results in expanding the value ADD generated by SPUDD. These numbers give the size of a tree that would be generated if a total ordering was imposed. Comparing these numbers with the numbers generated by SPI give an indication of the savings that occur due to the relaxation of the total ordering constraint. The rightmost column in Table 1 shows the ratio of the number of internal nodes in the tree representation to the number in the ADD representation. We see that reductions of up to 15 times are possible, when comparing only binary representations to binary/ternary representations, and reductions of over 20 times when comparing the same binary representations. These space savings also showed up in the amount of memory used. For example, the *f3* example took 691Mb of memory using SPI, and only 148Mb using SPUDD. The *f4* example took 378Mb of space using SPUDD.

The BIGADD limit (see Figure 6) was set to 10000 for all the examples. This limit broke up the complete action diagrams into $m = 2$ or 3 pieces, with typically 6000-10000 nodes in the first and second and under 1000 nodes in the third if it existed. In the large examples (*f2, 3 and 4*), it was not possible (with 1Gb of RAM) to generate the full complete action diagram ($m = 1$), and running times became too large when BIGADD was set to 1. The functionality of this "tuning knob" was not fully investigated, but is an interesting avenue for future exploration.

For comparison purposes, *flat* (unstructured) value iteration was run on both the *f* and *f0* examples. The times taken for these problems were 895 and 4579 seconds, respectively. For the larger problems, memory limitations precluded completion of the flat algorithm.

In order to examine the worst-case behaviour, we tested SPUDD on a series of examples, drawn from [8, 4], in which every state has a unique value; hence, the ADD representing the value function will have a number of terminal nodes exponential in the number of state variables. The problem EXPON involves n ordered propositions and n actions, one for each proposition. Each action makes its corresponding proposition true, but causes all propositions lower in the order to become false. A reward is given only if all variables are true. The problem is representable in $O(n^2)$ space using ADDs; but the optimal policy winds through the entire state space like a binary counter. This problem causes worst-case behaviour for SPUDD because all 2^n states have different values. SPUDD was tested on the EXPON example with $6, 8, 10$ and 12 variables, leading to state spaces with sizes $64, 256, 1024$ and 4096, respectively. The initial reward and the discounting factor in these examples must be scaled to accommodate the 2^n-step look-ahead for the largest problem (12 variables), and were set to 10^{16}

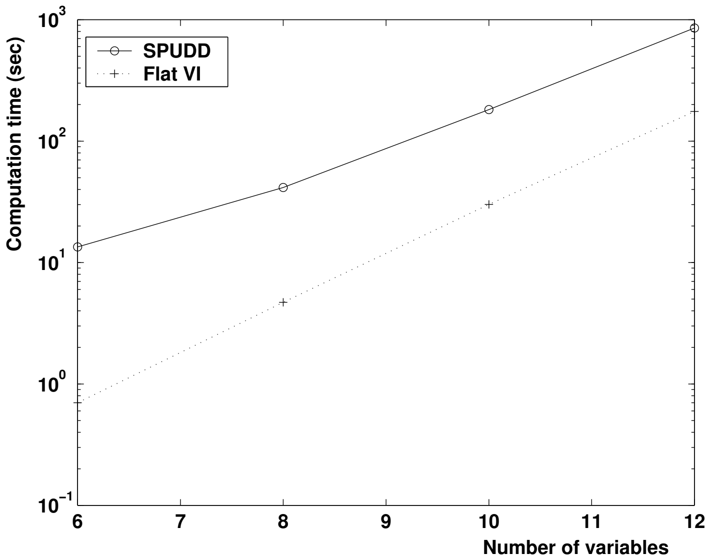


Figure 11: EXPON example: worst case behavior for SPUDD.

and 0.99, respectively.[6] Figure 11 compares the running times of SPUDD and (flat) value iteration plotted (in log scale) as a function of the number of variables. Running times for both algorithms exhibit exponential growth with the number of variables, as expected.[7] It is not surprising that flat value iteration performs better in this type of problem since there is absolutely no structure that can be exploited by SPUDD. However, the overhead involved with creating ADDs is not overly severe, and tends to diminish as the problems grow larger. With $n = 12$, SPUDD takes less than 10 times longer than value iteration.

One can similarly construct a "best-case" series of examples, where the value function grows linearly in the number of problem variables. The problem LINEAR involves n variables and has $n + 1$ distinct values. The MDP can be represented in $O(n^2)$ space using ADDs and the optimal value function can be represented in $O(n)$ space with an ADD (see [4] for further details).[8] Hence, the inherent structure of such a problem can easily be exploited. As seen in Figure 12, SPUDD clearly takes advantage of the structure in the problem, as its running time increases linearly with the number of variables, compared to an exponential increase in running time associated with flat value iteration.

[6] Since the value obtained at the state furthest from the goal is the goal reward discounted by the number of system states (since each must be visited along the way), the goal reward must be set very high to ensure that the value at this state is not (practically) zero.

[7] The running times are especially large due to the nature of the problem which requires a large number of iterations of alue iteration to converge.

[8] Of course, *best-case* behavior for SPUDD involves a problem in which all variables are irrelevant to the value function. This problem represents a "best case" in which all variables are required in the prediction of state value.

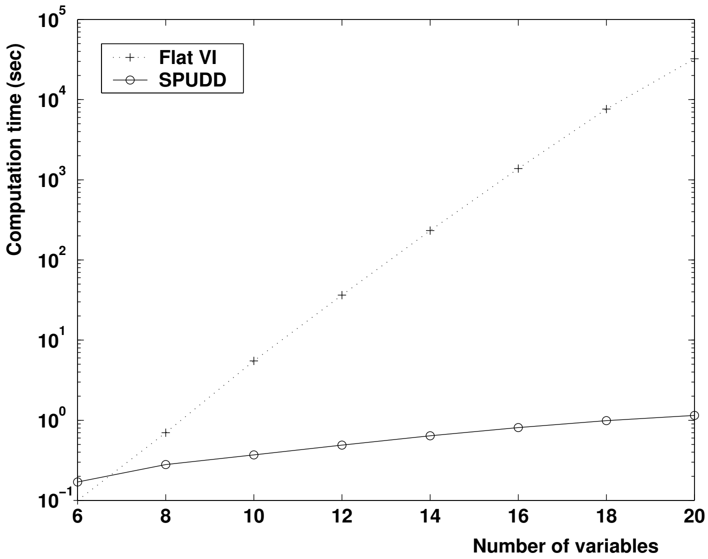


Figure 12: LINEAR example: best-case behavior for SPUDD.

6.2 Approximation Results

All experiments in this section were performed on problem domains where the variable ordering was the one selected implicitly by the constructors of the domains. [9]

In this section we compare optimal value iteration using ADDs with approximate value iteration using different pruning percentages p in *max_error* and *max_size* modes. In order to avoid overly aggressive pruning in the early stages of the value iterations when in *max_error* mode, we need to take into account the size of the value function at every iteration. Therefore, we use a sliding tolerance specified as $p \sum_{i=0}^{n} \beta^i span(R)$ where R is the initial reward diagram, β is the discount factor introduced earlier and n is the iteration number.

We illustrate running time, value function size (internal nodes and leaf nodes), number of iterations, and the *normalized ASSE*. The *NASSE* is the ratio of the *ASSE* for the approximate value function with the *ASSE* for a trivial policy. Thus, a near-optimal policy would have a very small *NASSE* while a poor policy would result in a large *NASSE*. It is important to note that the *all-pairs* method in *max_error* mode, the pruning strength is an upper bound for the approximation error. Indeed, the approximation algorithm guarantees to return the value diagram of smallest size for which no range will exceed the specified tolerance. This is very useful as we have tight control over the error bound of any approximated value function, and thus also over the quality of the approximate policy. On the other hand, the *round-off* algorithm depends on the boundaries defined by the rounding process, and so the approximation error is not as tightly bounded by the pruning percentage.

[9] Experiments showed that conclusions in this section are independent of variable order.

Value Function	pruning (%)	time (s)	iter	nodes (int)	leaves	a-error (%)	NASSE (%)
Optimal	0	287.1	44	22169	526	0.0	0.0
Allpair	1	671.6	44	17108	117	0.9	0.6
	2	667.5	44	15960	77	1.9	1.0
	3	136.6	15	15230	58	3.0	7.4
	4	82.3	12	14510	48	4.0	11.7
	5	45.2	10	11208	38	4.9	14.7
	6	30.6	9	8969	28	5.9	17.3
	7	22.0	8	8704	26	6.9	23.2
	8	11.1	7	5974	22	8.0	26.4
	9	5.1	6	3627	17	8.9	29.1
Roundoff	1	1034.5	44	18842	305	0.4	0.2
	2	1047.9	44	19034	320	0.9	0.6
	3	511.2	24	19847	323	1.4	2.6
	4	617.3	28	20027	320	1.9	1.6
	5	267.5	15	20590	332	2.7	9.1
	6	173.6	14	18824	306	3.3	10.6
	7	153.5	12	19541	284	4.2	13.5
	8	116.8	11	17777	275	5.0	14.4
	9	125.4	12	16948	253	5.4	13.2

Table 2: Comparing optimal with approximate value iteration in *max_error* mode on a domain with 28 boolean variables.

In the following two sub-sections we present results obtained in *max_error* and *max_size* modes using both the *all-pairs* and the *round-off* methods.

6.3 *max_error* mode

Table 2 shows results obtained using both approximation methods in *max_error mode*. Contrary to what we expected, *round-off* is generally slower then *all-pairs* for a given pruning percentage. However, as previously discussed, the pruning percentage in the *round-off* method is not a tight bound on the approximation error, and therefore one must compare running times for results with similar *a-errors*. In this case, the times are more similar, although the *round-off* method still appears almost twice as slow as the *all-pairs* method. The NASSE is also generally smaller for the *round-off* method.

The effect of approximation on the performance of the value iteration algorithm is threefold. First, the approximation itself introduces an overhead which depends on the size of the value function being approximated. This effect can be seen in Table 2 at low pruning strengths ($1-2\%$), where the running time is increased from that taken by optimal value iteration. Second, the ranges in the value function reduce the number of iterations needed to attain convergence, as can be seen in Table 2 for pruning strengths greater than 3%. However, for the lower pruning strengths, this effect is not observed. This can be explained by the fact that a small number of states with values much greater (or much lower) than that of the rest of the state space may never be approximated. Therefore, to converge, this portion of the state space would require the same number of iterations as in the optimal case.[10]

The third effect of approximation is to reduce the size of the value functions, thus

[10] We are currently looking into alleviating this effect in order to increase convergence speed for low pruning strengths.

Value Function	pruning (%)	time (s)	iter	nodes (int)	leaves	a-error (%)	NASSE (%)
Optimal	0	287.1	44	22169	526	0.0	0.0
Allpair	2.5	308.2	44	21551	513	0.0	0.0
	5	308.9	44	21052	503	0.0	0.0
	10	321.7	44	19949	476	0.0	0.0
	20	677.3	44	17842	186	0.4	0.2
	30	450.5	32	15794	63	2.5	2.2
	40	159.3	16	13499	38	3.1	7.1
	50	98.4	13	10943	27	4.7	10.2
	60	66.8	11	8990	25	5.3	13.8
	70	45.8	10	6694	14	8.7	18.5
	80	24.0	8	4227	13	9.5	24.1
Roundoff	2.5	466.9	44	21766	515	0.0	0.0
	5	85.8	13	65	10	37.9	64.7
	10	67.7	12	165	19	38.3	61.7
	20	421.9	26	17316	283	3.1	2.3
	30	206.8	17	15625	199	3.8	5.9
	40	172.6	15	13448	156	5.4	8.4
	50	147.6	15	11005	152	11.4	19.7
	60	80.9	11	8730	94	10.6	22.4
	70	61.2	10	6525	75	13.6	29.4
	80	38.3	9	4204	76	22.7	43.8

Table 3: Comparing optimal with approximate value iteration in *max_size* mode on a domain with 28 boolean variables.

reducing the per iteration computation time during value iteration. This effect is clearly seen at pruning strengths greater than 3% for the *all-pairs* method and 4% for the *round-off* method, where it overtakes the cost of approximation, and generates significant time and space savings. Speed-ups of 6 and 9 fold are obtained for pruning strengths of 5% and 6% respectively. Furthermore, fewer than 30 leaf nodes represent the entire state space, while the value functions errors are less than 17%. This confirms our initial hypothesis that many values within a given domain are very similar and thus, replacing such values with ranges drastically reduces the size of the resulting diagram without significantly affecting the quality of the resulting policy. Pruning at more than 7% has a larger error, and takes a very short time to converge.

The tradeoff between the resource bounds which need to be satisfied, and the error bounds which can be tolerated is application dependent. Figure 13 demonstrates this tradeoff for our examples by plotting the average of the normalized computational time and the ASSE against the pruning strength. Minimizing this function acheives optimally low error and fast computation together. The time and error are equally weighted in Figure 13, but applications can choose appropriate weights to find optimal pruning strengths in a similar fashion.

The tradeoff between computation time and decision error is generally application dependent. In the domain presented here, assuming that time and error are equally weighted, we find optimality with pruning strengths on the order of 6%. However, more solid conclusions could be drawn using more sophisticated trade-off functions.

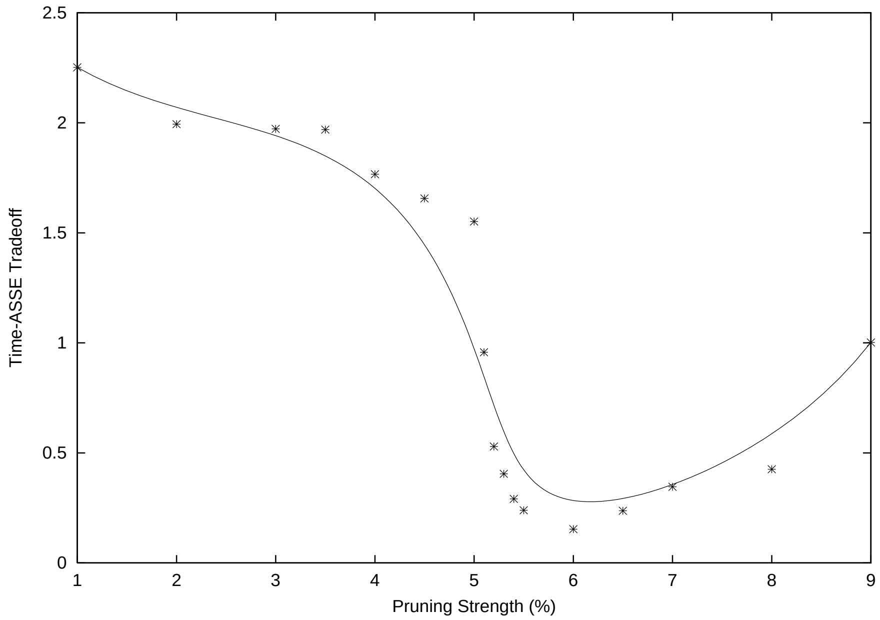


Figure 13: Smoothed curve of the ASSE and Computation time tradeof as a function of the pruning strength for a domain with 28 boolean variables.

6.4 *max_size* mode

In this experiment (Table 3), one should note that pruning percentages represent the diminution in size imposed on the value function. For example, a pruning percentage of 10 will restrict the approximate value function to be at least 10% smaller than the optimal value function.

For the *all-pairs* method, the results are very similar as those presented in the previous subsection. However, as one can see, the results obtained with the *round-off* method are non-monotonic. That is, for increasing pruning strength, the NASSE is not always increasing. In particular, the results for 5% and 10% pruning are very peculiar in that they converge very quickly, generate large errors and small diagrams. At the present time, we can only attribute this effect to a particularity of the problem domains in combination with the round-off boundaries as discussion previously. Further experiments need to be performed in order to more fully characterize such behaviour.

6.5 Variable reordering

Results in the previous section were all generated using the "intuitive" variable ordering for the problem at hand. It is probable that such an ordering is close to optimal, since the specification of the problem is more likely to put the most important variables first. However, such orderings may not always be obvious, and the effects of a poor ordering on the resources required for policy generation can be extreme. Therefore,

Ex.	Optim. size	post-shuffle	Avg. post-reorder SFT	RND	MNS	Reorder time (s) SFT	RND	MNS
f	1200	2840	788	1138	1152	1.2	1.3	.8
f0	1597	4642	945	1577	1603	2.3	2.9	1.3
f1	3101	6542	1661	3099	3107	4.5	6.2	3.3
f2	3101	7813	1854	3604	3107	5.7	9.6	2.9
f3	9215	20124	2185	6743	9221	14.2	17.5	10.0
f4	22169	42584	4620	10426	22171	48.8	39.4	27.9
f5	43675	98277	11653	33518	43675	115.4	147.7	79.9
f6	161318	442342	29029	90701	161318	539.5	638.6	432.2

Table 4: Reordering techniques applied to value diagrams. All values are averages over 10 trials. MNS = *minspan*, SFT = *sifting* and RND = *random*

to characterize the reordering methods discussed in Section 5.4, we ran three sets of experiments. We first examine a single reordering attempt on value diagrams of various sizes using the three reordering methods. We characterize the effects of reordering on value diagram size as well as the time for reordering attempts. In the second and third experiments, we examine the effects of reordering when applied during optimal and during approximate policy generation.

Starting with the final (optimal) value diagrams found using the “intuitive” ordering for various problem sizes, we randomly shuffle the variable orders and then attempt reordering using the *sifting*, *random* and *minspan* methods. We record the final value diagram sizes and the times for single reordering attempts, as shown in Table 4. All results are averages of ten trials. Figure 14 shows the resulting sizes after reordering plotted against the original value function size (with the intuitive ordering). The figure shows the *sifting* method to be a clear winner, resulting in sizes which are five times smaller than the original (intuitive ordering) ones. The *random* reordering method only reduces the sizes by a factor of two, while the *minspan* method finds diagrams of nearly equal size to those with the intuitive ordering. As can be seen in Table 4, though, all methods clearly reduce the sizes from those using a randomly shuffled ordering. Figure 15 compares running times of the three reordering methods for a single reordering attempt. In this case, we see that the *minspan* method is the fastest, followed by the *sifting* and *random* methods. Overall, and as we will demonstrate in the following, the sifting method appears to be the optimal performer, losing little in computation time to the *minspan* method, and outperforming both *random* and *minspan* methods in sizes generated.

We next examine the effectiveness of these variable reordering techniques for value iteration, making no *a-priori* assumptions about variable orders. We initially randomly shuffle the variable orders and then run optimal value iteration. We run with no reordering attempts, and with the *sifting*, *random* and *minspan* methods applied at each iteration. The final value diagram sizes are compared with those found using the “intuitive” ordering in Figure 16. In the absence of any reordering, diagrams produced with randomly shuffled variable orders are up to 3 times larger than those produced with the intuitive (unshuffled) order. The minimum span reordering method, starting from a randomly shuffled order, finds orders which are equivalent to the intuitive one, producing value diagrams with nearly identical size, an effect which appeared similarily in Table 4. The sifting and random reordering methods find orders which reduce

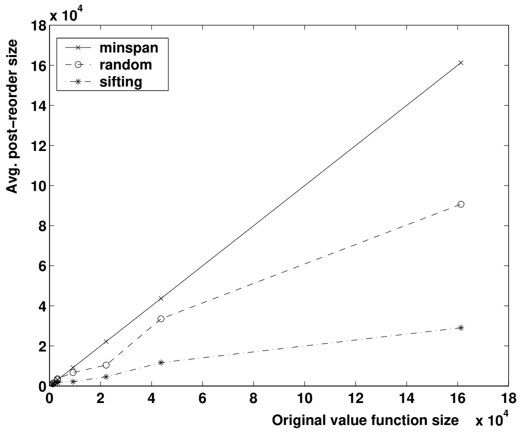


Figure 14: Sizes of value functions after reordering methods applied once, plotted as a function of the original value function sizes.

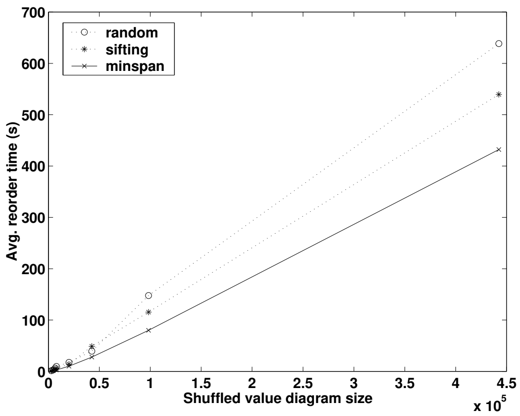


Figure 15: Times for reordering plotted as a function of the shuffled value function sizes.

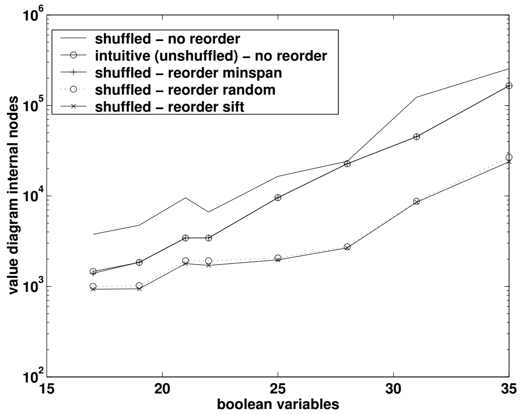


Figure 16: Sizes of final value diagrams plotted as a function of the problem domain size, using optimal value iteration

the sizes further by up to a factor of 5. Similar results are obtained with approximate value iteration with a pruning strength of 3% applied to a range of problem domain sizes (Figure 17).

Reordering attempts take time, but on the other hand, dynamic programming is faster with smaller diagrams. We examine this tradeoff in Figure 18, which compares the time taken for optimal value iteration when using the "intuitive" variable ordering with that when starting from a randomly shuffled ordering and applying the sifting reordering method. Results are shown for no reordering (labeled *reorder never* in Figure 18), for sifting applied only once at the beginnning (labeled *reorder once prior* in Figure 18), for sifting applied only for the first 5 iterations of VI (labeled *reorder first 5* in Figure 18), and for sifting applied for all iterations of VI (labeled *reorder all* in Figure 18). Finally, Figure 18 shows the time taken for optimal value iteration using a variable ordering generated by running approximate value iteration (*all-pairs* 10% error pruning) with sifting applied at each iteration (labeled *reorder from approx* in Figure 18). Randomly shuffled orders cause VI to take up to 3 times longer than when using the "intuitive" order. Most other reordering methods we examined made VI run in time similar to that when using the "intuitive" ordering. However, reordering only for the first five iterations of VI outperformed the "intuitive" orders by up to a factor of 4, and the randomly shuffled order by up to a factor of 5.

Overall, we find that applying sifting reordering for the first five iterations of our algorithm is the best performer. It is clear from our experiments that reordering methods solve the problem of specifying a good variable ordering, with little or no computational expense from those generated by hand.

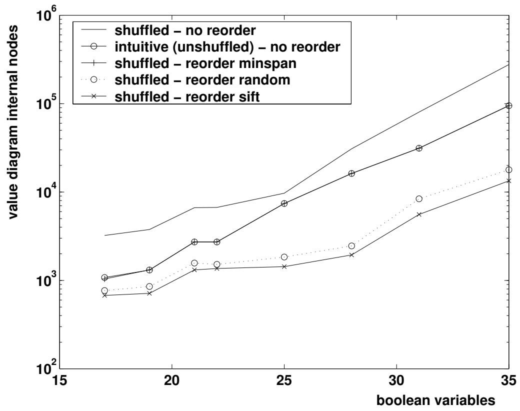


Figure 17: Sizes of final value diagrams plotted as a function of the problem domain size, using approximate value iteration with 3% error pruning.

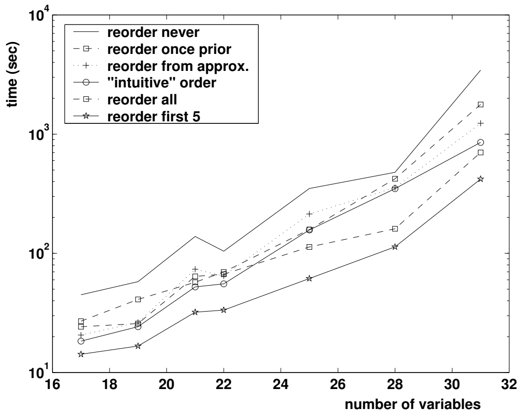


Figure 18: Time for optimal value iteration using various reordering methods (see text)
.

7 Concluding Remarks

In this paper, we described SPUDD, an implementation of value iteration, for solving MDPs using ADDs. The ADD representation captures some regularities in system dynamics, reward and value, thus yielding a simple and efficient representation of the planning problem. By using such a compact representation, we are able to solve certain types of problems that cannot be dealt with using current techniques, including explicit matrix and decision tree methods. Though the technique described in this paper has not yet been tested extensively on realistic domains, our preliminary results are encouraging.

We have also examined two methods for approximate dynamics programming for MDPs using ADDs. ADDs were found to be ideally suited to this task. The results we present have clearly shown their applicability on a range of MDPs with up to 34 billion states.

Investigations into the use of variable reordering during value iteration have also proved fruitful, and yield large improvements in running times. We have shown that our reordering methods are capable of equalling or bettering the "intuitive" orderings, when starting from randomly shuffled orders. This implies independence from the specification of good *a-priori* variable orders, a task usually performed by human experts only.

One drawback of using ADDs is the requirement that variables be boolean. Any (finite-valued) non-boolean variable can be split into a number of boolean variables, generally in a way that preserves at least some of the structure of the original problem (see above), though it often makes the new state space larger than the original. Conceptually, there is no difficulty in allowing ADDs to deal with multi-valued variables (all algorithms and canonicity results carry over easily). However, for domains with relatively few multi-valued variables, SPUDD does not appear to be handicapped by the requirement of variable splitting.

Future directions for this project will involve a closer look at some of the tuning parameters of our algorithms, and application of our methods to realistic problem domains currently being used.

Acknowledgements

Thanks to Richard Dearden for helpful comments and for providing both his SPI code and example descriptions for comparison purposes. St-Aubin was supported by NSERC. Hu was supported by NSERC. Boutilier was supported by NSERC Research Grant OGP0121843 and IRIS-III Project "Dealing with Actions."

References

[1] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *International Conference on Computer-Aided Design*, pages 188–191. IEEE, 1993.

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

[3] D. P. Bertsekas and D. A. Castanon. Adaptive aggregation for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, 34:589–598, 1989.

[4] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. manuscript, 1999.

[5] Craig Boutilier. Correlated action effects in decision theoretic regression. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 30–37, Providence, RI, 1997.

[6] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 1998. To appear.

[7] Craig Boutilier and Richard Dearden. Approximating value trees in structured dynamic programming. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 54–62, Bari, Italy, 1996.

[8] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1104–1111, Montreal, 1995.

[9] Craig Boutilier, Nir Friedman, Moisés Goldszmidt, and Daphne Koller. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 115–123, Portland, OR, 1996.

[10] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[11] Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Automatic obdd-based generation of universal plans in non-deterministic domains. In *AAAI-98*, 1998.

[12] Thomas Dean and Robert Givan. Model minimization in markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 106–111, Providence, 1997.

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

[14] Richard Dearden and Craig Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 89:219–283, 1997.

[15] Rina Dechter. Topological parameters for time-space tradeoff. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 220–227, Portland, OR, 1996.

[16] Steve Hanks and Drew V. McDermott. Modeling a dynamic and uncertain world i: Symbolic and probabilistic reasoning about change. *Artificial Intelligence*, 1994.

[17] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, 1988.

[18] David Poole. Exploiting the rule structure for decision making within the independent choice logic. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 454–463, Montreal, 1995.

[19] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley, New York, NY., 1994.

[20] J.Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[21] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. ICCAD-93*, pages 42–47, Santa Clara, CA, November 1993.

[22] Fabio Somenzi. CUDD: CU decision diagram package. Available from `ftp://vlsi.colorado.edu/pub/`, 1998.

[23] Paul E. Utgoff. Decision tree induction based on efficient tree restructuring. Technical Report 95-18, University of Massachusetts, March 1995.