

Efficient ADD Operations for Point-Based Algorithms

Guy Shani

shanigu@cs.bgu.ac.il
Ben-Gurion University
Beer-Sheva, Israel

Pascal Poupart

ppoupart@cs.uwaterloo.ca
University of Waterloo
Ontario, Canada

Ronen I. Brafman

brafman@cs.bgu.ac.il
Ben-Gurion University
Beer-Sheva, Israel

Solomon E. Shimony

shimony@cs.bgu.ac.il
Ben-Gurion University
Beer-Sheva, Israel

Abstract

During the past few years, point-based POMDP solvers have gradually scaled up to handle medium sized domains through better selection of the set of points and efficient backup methods. Point-based research has focused on flat, explicit representation of the state space, yet in many realistic domains a factored representation is more appropriate. The latter have exponentially large state-spaces, and current methods are unlikely to handle models of reasonable size. Thus, adapting point-based methods to factored representations by modeling propositional state spaces better, e.g. by using Algebraic Decision Diagrams (ADDs) is needed. While a straightforward ADD-based implementation can effectively tackle large factored POMDPs, we propose several techniques to further improve scalability. In particular, we show how ADDs can be used successfully in factored domains that exhibit reasonable locality. Our algorithms are several orders of magnitude faster than current point-based algorithms used with flat representations.

Introduction

Partially Observable Markov Decision Processes (POMDPs) are widely used to model agents acting in a stochastic environment with partial observability. Exact solution algorithms for POMDPs can handle small state spaces only, but approximate solution methods, and in particular, the family of point-based methods (Pineau, Gordon, & Thrun 2003), generates good, approximate policies for larger domains.

In many cases, it is natural to describe the state of the environment within a POMDP via a set of *state variables*, and the effects of actions in terms of their effects on these variables. Dynamic Bayesian Networks (DBNs) with conditional probability tables (CPTs) in the form of decision-trees or graphs are often used to represent these effects compactly. This representation is often referred to as a *factored* representation. The state space of such models is exponential in the number of variables, and quickly grows outside the reach of methods that operate on an explicit, flat representation, including point-based methods. To address this problem, researchers suggested the use of Algebraic Decision Diagram (ADDs) (Boutilier & Poole 1996; Hansen & Feng 2000; Poupart 2005; Hoey *et al.* 2007). ADDs represent functions $f : \{T/F\}^n \rightarrow \mathbb{R}$ and can therefore compactly represent

the environment dynamics. ADDs support fast function sum and product operations. Consequently, all basic POMDP operations, such as the computation of the next belief state, can be implemented efficiently on ADDs. When each action affects a relatively small number of variables, this representation can be very compact.

ADD-based algorithms for MDPs/POMDPs are not new. In the context of fully observable Markov Decision Processes (MDPs), ADD-based algorithms scale-up better than algorithms that operate on flat representations (St-Aubin, Hoey, & Boutilier 2000). POMDPs provide a considerable challenge — instead of a single ADD used to describe a single MDP value function, in POMDPs we need to maintain ADDs for multiple value functions (α -vectors) and belief states (Hansen & Feng 2000). The number of ADD operations is therefore much more significant in POMDPs. The Symbolic Perseus (SP) algorithm (Poupart 2005) used ADDs to implement the Perseus algorithm (Spaan & Vlassis 2005), using several techniques and approximations that we discuss in this paper.

This paper provides the first thorough description and evaluation of the various possible choices when using ADDs in point-based algorithms. Even though ADDs were considered in the past, no previous research has properly evaluated their usability on multiple POMDP domains, and identified the possibilities and difficulties. We explain how ADD operations can be improved in various ways, and also explain various approximation techniques that can be employed. Our operations are applicable for any point-based algorithm.

In addition, we take a special look at domains that possess effect locality, i.e., the property that each action affects only a small number of variables. Many standard test domains exhibit this property, and other authors attempted to utilize it (Guestrin, Koller, & Parr 2001). This paper explains how backup and τ operations used in point-based methods can be efficiently implemented for such domains. We show experimentally that in domains exhibiting reasonable effect locality, our ADD-based point-based algorithms are several orders of magnitude faster than the best existing point-based algorithms, considerably pushing the boundary of existing POMDP solutions algorithms.

Background

MDPs and POMDPs

A Markov Decision Process (MDP) is a tuple $\langle S, A, tr, R \rangle$ where S is a set of world states, A is a set of actions, $tr(s, a, s')$ is the probability of transitioning from state s to state s' using action a , and $R(s, a)$ defines the reward for executing action a in state s .

A Partially Observable Markov Decision Process (POMDP) is a tuple $\langle S, A, tr, R, \Omega, O, b_0 \rangle$ where S, A, tr, R compose an MDP, Ω is a set of observations and $O(a, s, o)$ is the probability of observing o after executing a and reaching state s . The agent maintains a *belief-state* — a vector b of probabilities such that $b(s)$ is the probability that the agent is currently at state s . b_0 defines the initial belief state — the agent belief over its initial state.

The transition from belief state b to belief state b' using action a is deterministic given an observation o and defines the τ transition function. We denote $b' = \tau(b, a, o)$ where:

$$b'(s') = \frac{O(a, s', o) \sum_s b(s) tr(s, a, s')}{pr(o|b, a)} \quad (1)$$

$$pr(o|b, a) = \sum_s b(s) \sum_{s'} tr(s, a, s') O(a, s', o) \quad (2)$$

In many cases the belief state b has many zero value entries. Implementing belief states using data structures that allow iterating only over the non-zero entries of the belief state is therefore much faster in practice.

In many MDP and POMDP examples the agent should either reach some state (called the goal state) where it receives a reward, or collect rewards that can be found in a very small subset of the state space. Other problems provide different rewards in each state.

Value Functions for POMDPs

It is well known that the value function V for the belief-space MDP can be represented as a finite collection of $|S|$ -dimensional vectors known as α vectors. Thus, V is both piecewise linear and convex (Smallwood & Sondik 1973). A policy over the belief space is defined by associating an action a to each vector α , so that $\alpha \cdot b$ represents the value of taking a in belief state b and following the policy afterwards. It is therefore standard practice to compute a value function — a set V of α vectors. The policy π_V is derivable using:

$$\pi_V(b) = \operatorname{argmax}_{a: \alpha_a \in V} \alpha_a \cdot b \quad (3)$$

where $\alpha_a \cdot b$ is the inner product (or dot product) of vectors:

$$\alpha \cdot b = \sum_s \alpha(s) b(s) \quad (4)$$

The value function can be iteratively computed

$$V_{n+1}(b) = \max_a [b \cdot r_a + \gamma \sum_o pr(o|a, b) V_n(\tau(b, a, o))] \quad (5)$$

where $r_a(s) = R(s, a)$. The computation of the next value function $V_{n+1}(b)$ out of the current V_n (Equation 5) is

known as a *backup* step, and can be efficiently implemented (Pineau, Gordon, & Thrun 2003) by:

$$\operatorname{backup}(b) = \operatorname{argmax}_{g_a^b: a \in A} b \cdot g_a^b \quad (6)$$

$$g_a^b = r_a + \gamma \sum_o \operatorname{argmax}_{g_{a,o}^\alpha: \alpha \in V} b \cdot g_{a,o}^\alpha \quad (7)$$

$$g_{a,o}^\alpha(s) = \sum_{s'} O(a, s', o) tr(s, a, s') \alpha^i(s') \quad (8)$$

The $g_{a,o}^\alpha$ computation (Equation 8), which we call the *atomic backup* of an α -vector, does not depend on the belief state b or the number of vectors in V . Like the τ function computation (Equation 1) it has a complexity of $O(|S|^2)$, but as most α -vectors are not sparse the atomic backup operation in practice is slower than τ .

Point Based Value Iteration

Updating V over the entire belief space, hence computing an optimal policy is computationally hard (PSPACE-hard). Approximation schemes attempt to decrease the computation complexity.

A possible approximation is to compute an optimal value function over a subset of the belief space (Pineau, Gordon, & Thrun 2003). An optimal value function for a subset of the belief space is only an approximation of a full solution, but will hopefully generalize well to other belief states. Point-based algorithms choose a subset B of the belief space, reachable from the initial belief state, and compute a value function only over these belief points using point-based backups (Equation 6).

Point-based algorithms differ on two main scales — the selection of B , the set of belief points and the order by which point-based backups are performed.

Pineau and Thrun (Pineau, Gordon, & Thrun 2003) expand B iteratively, choosing at each step successor belief points that are as far as possible from B , attempting to cover the space of reachable beliefs as best as possible. When computing the value function they go over the belief states in an arbitrary order, creating an α -vector for each belief state.

In the Perseus algorithm, Spaan and Vlassis (Spaan & Vlassis 2005) collect a set of belief points through a random walk over the belief space. When updating the value function, they randomly select at each step a belief state b from B that was not yet improved and compute an α -vector for b . Afterwards, all belief states whose value has been improved by the new α -vector are removed from B . Once B has been emptied all original belief points are returned and a new iteration is started.

A different approach is taken by the trial-based algorithms — HSVI (Smith & Simmons 2005) and FSVI (Shani, Brafman, & Shimony 2007). These algorithms execute traversals through the belief space, computing backups in reversed order. These algorithms currently scale up well because they do not constantly maintain a large set of belief points, only the current belief space traversal. HSVI has convergence guarantees but its traversal heuristic is time consuming. FSVI computes good traversals in belief space rapidly but cannot guarantee convergence.

All point-based algorithms above use a number of basic operations such as backups, g -operations (Equation 8), the τ

function (Equation 1), and inner products between α -vectors and belief states (Equation 4). An efficient implementation for these operations will speed up all point based algorithms.

Factored POMDPs

Traditionally, the MDP/POMDP state space is defined by a set of states, which we call a flat representation. For many problems, however, it is natural to define a set of state variables $X = \{X_1, \dots, X_n\}$ such that each state $s = \langle x_1, \dots, x_n \rangle$ is an assignment for the state variables (Boutilier & Poole 1996). The transition function $tr(s, a, s')$ is replaced by the distribution $pr(X'_i|X, a)$ ¹.

It is also possible to define factored actions (Guestrin, Koller, & Parr 2001) and observations, but we currently choose a flat representation for actions and observations. Without loss of generality, we use only binary variables to simplify the exposition (i.e. limit the possible values of a state variable to T/F). To represent a 'real' state variable that requires n distinct values, such as a coordinate on an $n \times n$ grid, we use $\log(n)$ artificial boolean state variables, jointly representing a value in the required range.

Consider, for example, the network administration problem (Poupart 2005), where a set of machines are connected through some network. Each machine may fail with some probability, which is increased when a neighbor machine is down. The task of the network administrator is to maximize the number of working machines, using restart operations over a single machine. The administrator must ping a machine to receive noisy status information (up/down).

This problem can be described using a state variable for each machine $X = \{M_1, \dots, M_n\}$. When machine i is currently working (up) we set $M_i = T$.

A factored representation becomes compact when dependencies between state variables are limited. Formally, for each state variable X_i and action a we define X_i^a to be the minimal set of variables s.t. $pr(X'_i|a, X_i^a) = pr(X'_i|a, X)$. Smaller X_i^a sets result in a compact description of the $pr(X'_i|a, X_i^a)$ distribution. For example, if the network topology is a clique, then for each i and a $X_i^a = X$ and the problem does not factor well. If the topology is a ring, then $|X_i^a| = 3$ — machine i and its two neighbors — the problem has a compact factored representation.

Boutilier and Poole (Boutilier & Poole 1996) suggested representing factored POMDPs using decision trees, where nodes are labeled by state variables and edges are labeled by possible values for the state variables (T/F). The decision tree leaves contain real numbers, representing probabilities or values.

Algebraic Decision Diagrams (ADDs)

An Algebraic Decision Diagram (R.I. Bahar *et al.* 1993) is an extension of Binary Decision Diagrams, that can be used to compactly represent decision trees. A decision tree can have many identical subtrees, and an ADD unifies these subtrees, resulting in a rooted DAG rather than a tree (Figure 1). The ADD representation becomes more compact as the problem becomes more structured.

¹We follow Boutilier and Poole in denoting the pre-action variables by X and the post-action variables by X' .

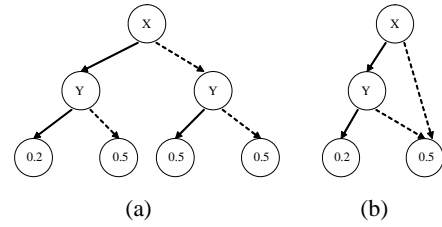


Figure 1: A decision tree and an ADD of the same function. All variables have binary values. A solid line indicates a 0 value (False) and a dashed line indicates a value of 1 (True).

As ADDs define functions $f : \{T/F\}^n \rightarrow \mathfrak{R}$, function *product* (denoted \otimes), function *sum* (denoted \oplus) and *scalar product*, or inner product, (denoted \odot) can be computed efficiently by iterating over the ADDs and caching visited nodes (Bryant 1986).

Given a variable ordering, an ADD has a unique minimal representation. The *reduce* operation takes an ADD and reduces it to its minimal form. Another operation specialized for ADDs is the *variable elimination*, also known as existential abstraction (denoted \sum_{X_i}). This operation eliminates a given variable X_i from the ADD by replacing each occurrence of X_i by the sum of its children. Let A be an ADD, and Y and Z variables, where Z does not appear in A . The *translate* operation (denoted $A(Y \rightarrow Z)$), replaces each occurrence of variable Y by variable Z in A . Let A be an ADD and $\langle x_1, \dots, x_n \rangle$ is an assignment to the variables in A . We denote the value A assigns to this specific assignment by $A(x_1, \dots, x_n)$. This value is computed by traversing the path specified by the assignment until a leaf is reached.

Point-Based Value Iteration with ADDs

An ADD-based point-based algorithm uses ADDs to represent belief states and α -vectors, and must provide an ADD-based implementation of the basic operations used: belief state backup (Equations 6 to 8), belief state update (Equation 1) and inner product of a belief state and an α -vector (Equation 4).

ADD-based Operations

To compute both τ and backups the *Complete Action Diagram* P_a^o (Boutilier & Poole 1996; Hansen & Feng 2000) is created, for each action a and observation o .

$$P_a^o(x_1, \dots, x_n, x'_1, \dots, x'_n) = \quad (9)$$

$$tr(\langle x_1, \dots, x_n \rangle, a, \langle x'_1, \dots, x'_n \rangle) O(a, \langle x'_1, \dots, x'_n \rangle, o)$$

ADDs are also used for the immediate reward function:

$$R_a(x_1, \dots, x_n) = R(\langle x_1, \dots, x_n \rangle, a) \quad (10)$$

The set of all P_a^o and R_a ADDs are a sufficient description of the system dynamics.

Belief update: Using the complete action diagram and replacing Equation 1, the next belief state is computed:

$$b' = \left(\sum_{X_1, \dots, X_n} P_a^o \otimes b \right) (X' \rightarrow X) \quad (11)$$

where the \otimes operation is ADD product and the sum is an existential abstraction over the pre-action state variables. A scalar product over the values in the resulting ADD by $1/p(o|a, b)$ is now needed. The posterior probability $pr(o|a, b)$ can be computed by eliminating all variables in the resulting ADD, but in the next section we offer a faster method.

After eliminating all pre-action variables, the remaining ADD is over X' — the post-action variables. Next, the variables are translated back into pre-action variables, replacing each occurrence of any X'_i with X_i .

Backup: To replace Equation 8 for computing $g_{a,\alpha}^o$ with an ADD implementation a similar approach is used:

$$\alpha' = \sum_{X'_1, \dots, X'_n} P_a^o \otimes \alpha(X \rightarrow X') \quad (12)$$

The translation here of the former α -vector (ADD) is needed because Equation 8 requires the value of the post-action state s' . All other operations used for backup such as summing and scalar products are done over the ADDs.

Inner product: Inner product operations provide a challenge. A simple implementation that follows the algorithm structure of Bryant (1986), will compare poorly to a flat representation. This is because it is easy to implement inner products very efficiently using sparse (non-zero entries only) vector representations of a belief state and α -vector. We explain in the next section how to efficiently implement inner products.

Compressing ADDs

St-Aubin et al. (2000) show how the structure in an ADD can be exploited to create approximations that influence the size of the ADD. We can reduce the number of leaves, and hence the number of nodes, by grouping together values that are ϵ far, thus creating smaller ADDs with little loss of accuracy. This operation can be easily done for ADDs representing α -vectors. To ensure that the α -vectors remain a lower bound over the value function we always replace a value v with a smaller value in the range $[v - \epsilon, v]$.

The same process is applicable to belief states. However, for belief states, where values can be substantially smaller than α -vector values, we use a much smaller ϵ . Also, we make sure that a non-zero value is never transformed into a zero value in the process.

Scaling Up ADD Operations

In flat POMDPs, belief states and α -vectors are efficiently implemented as vectors containing only the non-zero entries. ADD representation can be beneficial when belief states and/or α -vectors contains significant structure. In the worst case, the ADD reduces to a complete decision tree, and the amount of storage space is twice that of a flat vector.

The straight-forward replacement of vectors with ADDs discussed in the previous section provides some benefits in structured domains, but much leverage can be gained when using several modifications. We describe below a set of improvements that leads to substantially better performance. We provide details on some exact improvements and also on some approximations.

Efficient Inner Product Operations

The inner product operation between a belief state and an α -vector is executed many times by point-based algorithms. Within the augmented backup process described below and when attempting to find the best α -vector for a belief state during policy execution, many inner products are used.

A possible implementation of inner-products can handle the ADDs as if they were fully specified decision trees, traversing the two trees together, multiplying leaf scalars, and summing up the results. Most ADD operations such as product and sum are implemented likewise, and can be augmented by caching paths that were already traversed. Another, even more costly alternative, is to first compute the product of the two ADDs and then eliminate all variables through existential abstraction, resulting in a single leaf containing the sum of the scalars in the ADD.

As the result of an inner product is not an ADD but a scalar, we suggest a more efficient solution, that does not traverse the entire ADD structure. We first add to each ADD vertex a scalar, specifying the sum of the subtrees below it. As in an ADD an efficient representation may not specify some variables, we need to be careful when computing the sum such that missing variables are still taken into consideration. For example, in the ADD in Figure 1, the value sum of the Y variable is $0.2 + 0.5 = 0.7$, but when we compute the value sum of the X variable it is $0.7 + 0.5 \times 2 = 1.7$ because the right subtree of X contains two instances of the 0.5 value (as we can see specified in the fully specified decision tree) represented by a single node. Computing the sum of values is done within the reduce operation.

Using the computed sum at each node, we can avoid traversing all the nodes of the ADDs (Algorithm 1²).

Algorithm 1 InnerProduct(v_1, v_2)

```

1: if  $v_1$  is a leaf then
2:   return  $value(v_1) \times valueSum(v_2)$ 
3: if  $v_2$  is a leaf then
4:   return  $value(v_2) \times valueSum(v_1)$ 
5: if  $root(v_1) < root(v_2)$  then
6:   return  $InnerProduct(leftChild(v_1), v_2) +$ 
            $InnerProduct(rightChild(v_1), v_2)$ 
7: if  $root(v_1) > root(v_2)$  then
8:   return  $InnerProduct(v_1, leftChild(v_2)) +$ 
            $InnerProduct(v_1, rightChild(v_2))$ 
9: else
10:  return  $InnerProduct(leftChild(v_1), leftChild(v_2)) +$ 
            $InnerProduct(rightChild(v_1), rightChild(v_2))$ 

```

The recursion terminates whenever the traversal over one of the ADDs reaches a leaf. Hence, computation time is the product of the sizes of the two ADDs in the worst case (just like regular ADD products), but when the structure of the ADDs is considerably different, significant improvement is gained.

Let us trace the algorithm execution over the two ADDs in Figure 2. We begin the algorithm in the root of both ADDs (labeled by variable X), and first descend left to their False children. In A_1 we are now at the variable labeled

²Details omitted for a more readable pseudo-code.

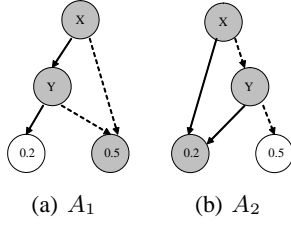


Figure 2: During an inner product operation, only nodes colored in grey are visited.

Y , but in A_2 we have reached a leaf. As such, there is no farther need to visit the children of Y in A_1 and we can immediately compute the value. The sum of the children of Y is $0.2 + 0.5 = 0.7$. The value of the leaf in A_2 is 0.2 . The result is therefore $0.2 \times 0.7 = 0.14$. The straight forward computation of the left subtree would have been $0.2 \times 0.2 + 0.5 \times 0.2 = 0.14$. When computing the inner products of the A_1 and A_2 , only the grey nodes are visited.

Adding the sum of its children to each node has another helpful aspect. When computing a belief state update (Equation 1) we need to normalize the resulting belief state. The normalizing value — $pr(o|a, b)$ — is the sum of values of the ADD. While it is possible to receive this value by an existential abstraction over all the variables of the ADD, the value sum of the root is the sum of all values of the ADD.

Relevant Variables

The size of P_a^o — the complete action diagram — influences the computation speed of all operations. Variable ordering provides a substantial reduction but a farther reduction can be achieved by ignoring some state variables.

In many problems each action affects only a small subset of the state variables of the domain. For example, in the RockSample domain, when a robot takes a *move* action, the state of the rocks (Good/Bad) does not change. We denote by X_R^a the set of state variables affected by action a , and X_{-R}^a , the set of state variables unchanged by a :

$$X_R^a = \{X_i : pr(X_i' \neq x | X_i = x, a) > 0\} \quad (13)$$

$$X_{-R}^a = \{X_i : pr(X_i' = x | X_i = x, a) = 1\} \quad (14)$$

$$pr(o|X', a) = pr(o|X_{-R}^a, a) \quad (15)$$

The last equation specifies that the observation received after action a depends only on the state variables in X_{-R}^a .

Now, we can simplify Equation 11:

$$\sum_{X_1, \dots, X_n} P_a^o \otimes b = \sum_{X_1, \dots, X_n} P_{R_a}^o \otimes P_{-R_a} \otimes b \quad (16)$$

where $P_{R_a}^o$ is an ADD restricted to the relevant variables and P_{-R_a} is restricted to the irrelevant variables. P_{-R_a} contains only variables that are unaffected by action a and also do not affect the probability of receiving an observation afterwards, and hence does not depend on the observation o .

Computing this equation using two ADD product operations is inefficient, as pre-computing the complete action diagram is done once where Equation 16 is executed over each τ operation. However, $P_{-R_a} \otimes b$ can be computed without an ADD product. For each $X_i \in X_{-R}^a$,

$pr(X_i' = x | X_i = x, a) = 1$, i.e. an action does not change the variable value. Hence, the product reduces to a translation operation, replacing each $X_i \in X_{-R}^a$ in b by X_i' :

$$\sum_{X_1, \dots, X_n} P_a^o \otimes b = \sum_{X_i \in X_{-R}^a} P_{R_a}^o \otimes b(X_{-R}^a \rightarrow X_{-R}^a) \quad (17)$$

In fact, P_{-R_a} is unneeded and thus not computed. An equivalent change is done to Equation 12, required for backup computations.

$$\sum_{X_i', \dots, X_n'} P_a^o \otimes \alpha(X \rightarrow X') = \sum_{X_i' \in X_{-R}^a} P_{R_a}^o \otimes \alpha(X_{-R}^a \rightarrow X_{-R}^a) \quad (18)$$

Instead of the complete action diagrams we hence maintain partial action diagrams only over the relevant variables. This results in smaller (in some cases, much smaller) ADDs, and in rapid computations of backups and τ operations.

Beliefs As Products of Marginals

The complete action diagrams (and even the partial action diagrams from the previous section) are, in the worst case, exponential in the number of state variables. Also, if a belief state is represented with a single ADD, its size might be exponential in the number of state variables. Unless several states have the same probability, the ADD will be a full tree. However, Boyen and Koller (1998) noticed that many of the correlations between state variables are weak and suggested that an approximation that breaks some of the weak correlations could be reasonable.

We define a set of components $C = \{C_0, \dots, C_m\}$ such that $C_i = \{X_{i_0}, \dots, X_{i_k}\}$ and $C_i \cap C_j = \emptyset$. Variables within a component are highly correlated while variables from different components are weakly correlated. A joint belief over all the variables is approximated as a product of marginal beliefs over the different components — $\tilde{b} = \prod_{i=0..m} \tilde{b}_i$, where b_i is an ADD representing the belief over component C_i . We maintain $T_{a,i}$ — the transition probabilities, and $O_{a,o,i}$ — the observation probabilities, for each C_i .

Belief update over this approximated belief state is done by multiplying the transition and observation probabilities, and eliminating all variables except for a single component:

$$\tilde{b}_i = \left(\sum_{X \cup X' / C_i'} \prod_i O_{a,o,i} \tilde{b}_i T_{a,i} \right) (X' \rightarrow X) \quad (19)$$

This approach trades the exact representation of belief states for a smaller representation.

Variable Elimination In Equation 19, if the product of the transition and observation ADDs is computed before any variable is eliminated, an exponentially large ADD will be generated, defeating the purpose of approximating beliefs by a compact product of marginals. However, since belief updates and point-based backups essentially correspond to sum-product operations, it is natural to consider the variable elimination algorithm (Zhang & Poole 1994). We can interleave the products and summations so that the intermediate ADDs remain small. The key to variable elimination is to select a good ordering for the elimination of the variables. While finding the best ordering is NP-hard, heuristics often

perform well in practice. As an example, a heuristic that greedily eliminates the variable that leads to the smallest intermediate ADD often yields good results.

The efficiency of variable elimination can also be improved by considering only the relevant variables (and their conditional probability distributions).

It is interesting to compare belief updates with complete action diagrams to variable elimination. The complete action diagram can be pre-computed, which saves the multiplication of all transition and observation probabilities. In contrast, it is not possible to pre-compute any product with variable elimination due to the interleaving of products and sums. But, the intermediate ADDs can be much smaller than the complete action diagram. Both approaches can be further improved by considering only relevant variables. We can also tradeoff accuracy for an additional speedup in variable elimination when approximating beliefs by a product of marginals.

Factored Expectation Another operation that must be re-defined is the inner product of a belief state and an α -vector:

$$\alpha \cdot \tilde{b} = \alpha \odot \prod_i \tilde{b}_i \quad (20)$$

It is possible to speed up the products $\alpha \odot \prod_i \tilde{b}_i$ too. We traverse the α vector ADD in such a way that the values stored in the leafs are multiplied by the product of the marginals corresponding to each path. Less than linear time can be achieved by pruning paths as soon as a marginal probability of 0 is encountered. We call the resulting algorithm Factored Expectation (Algorithm 2).

Algorithm 2 FE(v_α, \tilde{b})

```

1: if isLeaf( $v_\alpha$ ) then
2:   return value( $v_\alpha$ )
3: else
4:   result  $\leftarrow$  0,  $X_i \leftarrow$  variable( $v_\alpha$ )
5:   if  $pr(X_i = 0|\tilde{b}) > 0$  then
6:     result $+$  =  $pr(X_i = 0|\tilde{b}) * FE(leftChild(v_\alpha), \tilde{b})$ 
7:   if  $pr(X_i = 1|\tilde{b}) > 0$  then
8:     result $+$  =  $pr(X_i = 1|\tilde{b}) * FE(rightChild(v_\alpha), \tilde{b})$ 
9:   return result

```

Efficient Backup Operations

Measuring the CPU time it takes to execute atomic backup operations and comparing it to τ operations it is apparent that τ operations are considerably faster. We believe that this is because α -vectors rapidly become less structured, causing their ADDs to grow in size while, in the examples we use, belief states are highly structured and remain so even after executing many actions from the initial belief point b_0 . For example, in the Network Administration problem, immediately after a restart action, half of the states (states where the restarted machine is down) have zero probability. In the RockSample domain, where actions results are stochastic, only states that correspond to a single location over the board have non-zero probability. However, α -vectors for both domains rarely have any zero value states.

Therefore, computing the value of a belief state in the next value function by Equation 5, which is done by computing many belief updates, is faster than computing the backup operation (Equation 6) that computes many atomic backups.

Some of the computed atomic backups will be later used to construct the new α -vector in Equation 7. However, most of the $g_{a,o}^\alpha$ computed by the atomic backup operation are dominated by others and will not participate in the newly created α -vector. If we knew the action a and the vectors α_i that maximize Equation 7 and Equation 6 we could avoid computing dominated $g_{a,o}^\alpha$. Indeed, the action and α -vectors that maximize these equations are exactly the same as those that maximize Equation 5, which suggests a more efficient implementation for the backup process.

Algorithm 3 uses Equation 5 to compute which $g_{a,o}^\alpha$ are combined to yield the resulting α -vector and only then computes and combines them, thus minimizing the needed number of atomic backup computations.

Algorithm 3 Backup(b, V)

```

1:  $a^* \leftarrow nil, v^* \leftarrow -inf$ 
2: for each  $o \in \omega$  do  $\alpha_o^* \leftarrow nil$ 
3: for all  $a \in A$  do
4:    $Q_a(b) \leftarrow 0$ 
5:   for all  $o \in \omega$  do
6:      $b' \leftarrow \tau(b, a, o)$ 
7:      $\alpha_o \leftarrow \operatorname{argmax}_{\alpha \in V} \alpha \cdot b'$ 
8:      $Q_a(b) \leftarrow Q_a(b) + \alpha_o \cdot b'$ 
9:   if  $Q_a(b) > v^*$  then
10:     $a^* \leftarrow a, v^* \leftarrow Q_a(b)$ 
11:   for each  $o \in \omega$  do  $\alpha_o^* \leftarrow \alpha_o$ 
12: return  $r_{a^*} + \gamma \sum_o g_{a_o^*, o}^{\alpha_o^*}$ 

```

As all $g_{a,o}^\alpha$ computations are replaced with τ computations the complexity of the backup process does not change. Nevertheless, the actual CPU time can be reduced considerably.

Experimental Results

We tested our approach on a number of domains with increasing size. Our goal is to evaluate the ADD-based approach and determine whether it allows us to scale up better to larger domains. We also evaluate the influence of effect locality on ADD operations.

Tables 1, 2 and 3 present a detailed comparison of the running time of flat and ADD-based implementations of the basic operations used by point-based algorithms: τ function, atomic backup operations, inner products, and the full backup process. We also provide total running time. Our tests use the FSVI algorithm (Shani, Brafman, & Shimony 2007) which is both very simple and highly efficient. However, our ADD operations can be used by any point-based algorithm. Table 1 also contains results for the product of marginals approximation (denoted Marginals).

Our flat representation is also as efficient as possible. All belief points and α -vectors maintain only non-zero entries and all iterations are done only over these entries. All operations that iterate over the set of states $S_{a,s} = \{s' : tr(s, a, s') \neq 0\}$ for some state s are implemented in $O(|S_{a,s}|)$, avoiding zero transitions.

Rocks	6	8	10	12	14
$ S $	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Average time for $g_{a,o}^\alpha$ computation in milliseconds (10^{-3})					
Flat	4993	77904	×	×	×
Factored	4	23	74	432	1321
Average time for τ computation in microseconds (10^{-6})					
Flat	13952	281953	1455786	×	×
Factored	315	375	585	701	1115
Marginals	1980	2676	3554	4462	5009
Average time for inner product in microseconds (10^{-6})					
Flat	9	38	×	×	×
Factored	8	11	14	14	65
Marginals	14	38	97	230	1354
Average time for backup computation in milliseconds (10^{-3})					
Flat	10705	168074	×	×	×
Factored	33	112	126	246	2836
Marginals	123	278	640	2339	44943
Average belief state size (number of ADD vertices)					
Factored	10	11	12	14	15
Marginals	22	24	25	34	32
Average α -vector size (number of ADD vertices)					
	389	909	973	1515	5682
Total time until convergence in seconds					
Flat	501	35883	×	×	×
Factored	1	5	17	33	2258
Marginals	14	113	214	370	4774
Final Average Discounted Reward (ADR) - 200 trials					
	12.52	15.69	18.64	21.09	25.01

Table 1: Average time on the rock sample problem with an 8×8 board and an increasing number of rocks.

Benchmark Domains

RockSample - In the RockSample domain (Smith & Simmons 2005) a rover is scanning an n over m board containing k rocks using a long range sensor.

The factored representation has state variables for X and Y coordinates and a state variable for each rock. There are 4 move actions, k sensor actions and a single sample action. There are two observations — 'good' and 'bad'.

Network Administrator - In this domain a network administrator should maintain a network of n machines arranged in a ring topology (Poupart 2005). Each machine can be either 'up' or 'down'. The administrator can ping a machine, checking whether the machine is up or down. The factored representation has n variables specifying the machine state. The agent can 'ping' or 'restart' each machine ($2n$ actions) or execute a 'no-op' action. There are 2 observations — 'up and 'down'.

Logistics - In the logistics domain the agent must distribute a set of n packages between m cities. Each package starts in a random city and has a predefined destination city. To move packages between cities the packages must be loaded onto a truck. The agent has k trucks it can use.

Each action can have stochastic effects — loading and unloading a package may fail with a probability of 0.1. Loading fails always if the specified truck and package are not at the same city. Driving the truck to a city may fail with a probability of 0.25, causing the truck to end in some other city with uniform probability. Each action returns an observation of 'success' or 'fail' with 0.9 accuracy. The agent can

Machines	6	7	8	9	10	11	12
$ S $	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Average time for $g_{a,o}^\alpha$ computation in milliseconds (10^{-3})							
Flat	0.5	2	8	34	142	477	1827
Factored	1	1.9	5.1	14.9	36.7	110	227
Average time for τ computation in milliseconds (10^{-3})							
Flat	0.33	1.1	4.4	16.9	76.8	218	990
Factored	6.3	10.8	15.1	27.4	75.5	149	527
Average time for product in microseconds (10^{-6})							
Flat	34.1	36.1	45.6	66.2	93	172	373
Factored	76.1	131	226	305	481	647	820
Average time for backup computation in seconds							
Flat	0.11	0.68	10.8	81.4	452	1508	×
Factored	0.8	3.8	15.4	52.8	89.1	362	599
Total time till convergence in seconds $\times 10^3$							
Flat	0.11	0.65	2.4	10.2	25.1	317	×
Factored	0.22	1.06	3.5	11.3	24.7	110	480

Table 2: Average time for operations on the Network Administration problem with a ring configuration and different numbers of machines.

also 'ping' a package or a truck to find its location. When pinging a truck the result is the truck location and when pinging a package the result is the truck it is on, or the city it is at if it is currently unloaded. Results have a 0.8 accuracy. Driving the trucks between cities costs 1 and all other operations cost 0.1. When delivering a package to its final destination the agent receives a reward of 10.

For each truck there is a variable with m possible values (cities) and for each package there is a variable with $m + k$ possible values (cities and trucks). There are $n \times k$ load actions, loading a specific package to a specific truck, and n unload actions. There are $n + k$ ping actions. There are at most $m + k$ possible observations.

Packages	2	3	4	5	6
$ S $	2^{10}	2^{13}	2^{16}	2^{19}	2^{22}
Average time for τ computation in milliseconds (10^{-3})					
Flat	37	1539	36024	×	×
Factored	2.3	22.3	13.4	82.1	94.3
Average time for $g_{a,o}^\alpha$ computation in milliseconds (10^{-3})					
Flat	94	7393	537895	×	×
Factored	1.6	17.1	15.4	14.9	35
Inner product computation in microseconds (10^{-6})					
Flat	62.8	254.1	1188	×	×
Factored	95.5	855.2	487.9	3151	5428
Average time for backup computation in seconds					
Flat	2.8	80.3	5831	×	×
Factored	1.5	2.2	3.0	7.6	9.2
Total time till convergence in seconds					
Flat	203	15448	×	×	×
Factored	36	91	650	1559	5832

Table 3: Average time for the Logistics domain with 4 cities, 2 trucks and increasing number of packages.

Results

We compare our ADD based operations, using all the improvements explained above over all domains. In the tables

below, Flat refers to representing belief states and α -vectors as mapping from states to values. In both belief states and α -vectors only non-zero entries are maintained. Factored denotes the ADD based representation.

	ADD size for Move actions	ADD size for Check actions	Average time for G computation
7 Rocks - $ S = 2^{13}$			
XX'	21790	107412	125
Mixed	51	1712	73
Relevant	18	494	4
8 Rocks - $ S = 2^{14}$			
XX'	44164	219506	330
Mixed	51	1994	145
Relevant	18	505	13
9 Rocks - $ S = 2^{15}$			
XX'	86536	446238	855
Mixed	54	2280	331
Relevant	18	525	44

Table 4: Influence of variable ordering and relevant variables over ADD size and computation time in the RockSample problem with an 8X8 board and increasing number of rocks. XX' — first all pre-action variables and then all post-action variables. Mixed — X'_i specified immediately after X_i . Relevant — using only relevant variables in a mixed order.

Over all benchmarks the flat executions (which were the slowest) were stopped after a 48 hours timeout, resulting in an \times symbol in the tables below. Over all benchmarks the flat representation and the ADD representation resulted in identical ADRs. The approximate belief representation worked well for all domains: no approximation was introduced for Rock Sample, while the approximations introduced for Network and Logistics still allowed us to find equally good policies.

Rocks	7	8	9
g-based	24,873	35,285	71,076
τ -based	216	290	408

Table 5: CPU time for a backup operation comparing g-based vs. τ -based backups over the RockSample 8×8 domain with relevant variables and increasing number of rocks.

The best results are presented in the Logistics domain as it displays the maximal action effect locality. The size of the relevant action diagrams is therefore compact and all point based operations are computed very rapidly.

Table 4 shows the influence of variable ordering (St-Aubin, Hoey, & Boutilier 2000) and the elimination of irrelevant variables over ADD size and operation time for actions with many irrelevant variables (move actions in RockSample) and actions with a small number of irrelevant variables (check actions). Table 5 shows the advantage of τ -based backups compared to the standard g -based backup process used by point-based algorithms.

It is evident that in the RockSample and Logistics domain, ADD operations do not exhibit the exponential growth of operation time given the number of state variables. The inner

product operation benefits the most from our improvements. Belief states remain structured and thus compact, and the size of the α -vectors does not have a considerable effect on the inner product. This is very important, as point based algorithms execute a large number of inner products.

Conclusion and Future Work

This paper explains how ADDs can be used in point-based algorithms over factored POMDPs. We thoroughly outline the various approaches to scaling up, including efficient ADD operations, belief approximation through a product of marginals and the efficient point based backups.

We experimented with a range of benchmark domains with different properties. Our experimental results show that in domains that present action effect locality, the ADD-based algorithm is orders of magnitude better, and is able to scale up to substantially larger models. In the network domain, where effect locality does not hold (any machine can fail following each action), the ADD-based algorithm does not provide significant improvements, although it still scales up better than the flat algorithm.

References

- Boutilier, C., and Poole, D. 1996. Computing optimal policies for partially observable decision processes using compact representations. In *AAAI-96*, 1168–1175.
- Boyer, X., and Koller, D. 1998. Tractable inference for complex stochastic processes. In *UAI '98*, 33–42.
- Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.
- Guestrin, C.; Koller, D.; and Parr, R. 2001. Solving factored pomdps with linear value functions. In *IJCAI workshop on Planning under Uncertainty and Incomplete Information*.
- Hansen, E. A., and Feng, Z. 2000. Dynamic programming for POMDPs using a factored state representation. In *Artificial Intelligence Planning Systems*, 130–139.
- Hoey, J.; von Bertoldi, A.; Poupart, P.; and Mihailidis, A. 2007. Assisting persons with dementia during handwashing using a partially observable markov decision process. In *International Conference on Vision Systems (ICVS)*.
- Pineau, J.; Gordon, G.; and Thrun, S. 2003. Point-based value iteration: An anytime algorithm for POMDPs. In *IJCAI*.
- Poupart, P. 2005. *Exploiting Structure to Efficiently Solve Large Scale POMDPs*. Ph.D. Dissertation, University of Toronto.
- R.I. Bahar; E.A. Frohm; C.M. Gaona; G.D. Hachtel; E. Macii; and A. Pardo. 1993. Algebraic Decision Diagrams and Their Applications. In *International Conference on CAD*, 188–191.
- Shani, G.; Brafman, R.; and Shimony, S. 2007. Forward search value iteration for pomdps. In *IJCAI-07*.
- Smallwood, R., and Sondik, E. 1973. The optimal control of partially observable processes over a finite horizon. *OR* 21.
- Smith, T., and Simmons, R. 2005. Point-based pomdp algorithms: Improved analysis and implementation. In *UAI 2005*.
- Spaan, M. T. J., and Vlassis, N. 2005. Perseus: Randomized point-based value iteration for POMDPs. *JAIR* 24:195–220.
- St-Aubin, R.; Hoey, J.; and Boutilier, C. 2000. APRICODD: Approximate policy construction using decision diagrams. In *NIPS*.
- Zhang, N. L., and Poole, D. 1994. A simple approach to bayesian network computations. In *Canadian AI*, 171–178.