Approximate Dynamic Programming with Affine ADDs

Scott Sanner
NICTA and ANU
Canberra, Australia
ssanner@nicta.com.au

William Uther
NICTA and UNSW
Sydney, Australia
William.Uther@nicta.com.au

Karina Valdivia Delgado
University of Sao Paulo
Sao Paulo, Brazil
kvd@ime.usp.br

ABSTRACT

The Affin ADD (AADD) is an extension of the Algebraic Decision Diagram (ADD) that compactly represents *context-specific*, *additive* and *multiplicative* structure in functions from a discrete domain to a real-valued range. In this paper, we introduce a novel algorithm for efficientl findin AADD approximations that we use to develop the MADCAP algorithm for AADD-based structured approximate dynamic programming (ADP) with factored MDPs. MADCAP requires less time and space to achieve comparable or better approximate solutions than the current state-of-the-art ADD-based ADP algorithm of APRICODD and can provide approximate solutions for problems with context-specific additive and multiplicative structure on which APRICODD runs out of memory.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Dynamic programming; Plan execution, formation, and generation

General Terms

Algorithms

Keywords

Planning, Markov Decision Processes, Approximate Dynamic Programming

1. INTRODUCTION

Many single agent planning tasks in fully observed state spaces with stochastic action outcomes can be formalized and solved within the Markov Decision Process (MDP) framework. While traditional approaches to solving MDPs focused on exact enumerated state solutions to MDPs, this approach has proven impractical for large-scale decision-theoretic planning tasks where the number of distinct states in a model can easily exceed the limits of primary and secondary storage on modern computers. In recent years, there has been a great deal of research aimed at exploiting structure in order to compactly represent and efficientl solve MDPs [3].

One common way to exploit MDP structure is to describe it using a propositionally factored model that exploits various independences in the reward and transition functions [3]. While many MDPs can be compactly specifie in this way, this structure does not always translate to compactness in the solution. Thus, exploitation of structure must often be used in conjunction with approximate solution methods that guarantee the resulting representation remains manageably-sized.

One class of approximate solution methods for propositionally factored MDPs is provided by the linear-value approximation framework [6]. In this setting, an MDP value function is approximated as a linear combination of basis functions. While this approach is computationally efficien and appealing, it must be provided with an appropriate basis function set. To this end, methods have been proposed for generating basis functions for varying classes of problems; see [8] for an excellent review of recent approaches along with a proposal of new techniques. For certain methods of basis function generation discussed in [8], there do exist guarantees of error improvement when adding new basis functions; however, there are no corresponding guarantees on the efficien y of the resulting factored MDP computation for these same approaches. On the other hand, there are two approaches that deal directly with basis function generation and efficien y in a factored MDP setting [10, 9], but they do not provide *a priori* guarantees on the error obtained using these basis functions.

An alternate solution method for factored MDPs is provided by approximate structured dynamic programming (DP), e.g., using algebraic decision diagrams (ADDs) [1] in a structured value iteration algorithm such as SPUDD [7], which can be extended with ADD approximation techniques as in APRICODD [13]. ADDs permit the exploitation of *context-specific independence (CSI)* [4] and redundant structure in the solution of factored MDPs. In the APRICODD approach, steps in the dynamic programming solution to MDPs are interleaved with approximation steps that reduce the representational complexity of the ADD representation in exchange for bounded approximation error in the solution. This approach offers two advantages: (1) The structure of the approximated value function is automatically derived, and (2) the error of the approximate solution can be *a priori* bounded.

The main drawback to APRICODD vs. linear-value approximation techniques is that ADDs only exploit CSI whereas linear-value approximations can exploit both CSI and additive structure. However, the Affin ADD (AADD) has been used to extend the SPUDD algorithm to exploit CSI, additive, *and* multiplicative structure [12]. To this end, the major contribution of this paper is MADCAP[1]: an approximate dynamic programming algorithm based on a novel technique we introduce for compactly approximating AADDs. We empirically show that MADCAP yields faster running times, better compression and lower error rates than state-of-the-art ADD-based APRICODD, and can approximately solve some difficul problems on which APRICODD runs out of memory.

[1]"MADCAP" contains "MDP", "ADP", and "AD"=A^2D^2=AADD.

Cite as: Approximate Dynamic Programming with Affin ADDs, Scott Sanner, William Uther and Karina Valdivia Delgado, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. 1349-1356

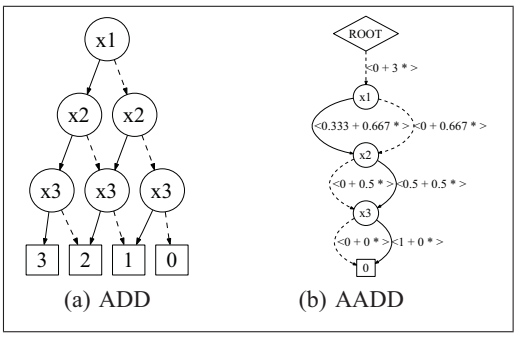

Figure 1: (a) ADD and (b) AADD diagrams for $\sum_{i=1}^{3} x_i$. Circles show variables, branches show variable assignments (solid $= true$ (high), dotted $= false$ (low)), squares show valuations.

2. DECISION DIAGRAMS

We begin by introducing ADDs and AADDs that are used in the structured approximate MDP solution algorithms in this paper.

2.1 Algebraic Decision Diagrams (ADDs)

Algebraic decision diagrams (ADDs) [1] provide a compact way to represent and perform operations on functions from a factored boolean domain to a real-valued range (i.e., $\{0,1\}^n \rightarrow \mathbb{R}$). They rely on two main principles to do this:

1. ADDs represent a function $\mathbb{B}^n \rightarrow \mathbb{R}$ as a directed acyclic graph (DAG) – essentially a decision tree with reconvergent branches and real-valued terminal nodes.

2. ADDs enforce a strict variable ordering on the decisions from the root to the terminal node, enabling a minimal, canonical diagram to be produced for a given function. Thus, two identical functions will always have identical ADD representations under the same variable ordering.

ADDs often provide an efficien representation of functions with context-specifi independence [4] and redundant structure. For example, the function $\sum_{i=1}^{3} x_i$ ($x_i \in \{0,1\}$) represented in Figure 1(a) as an ADD exploits the redundant structure of sub-diagrams in a DAG to avoid an exponential tree representation.

Definin a *reduced* ADD to be the minimally-sized ordered decision diagram representation of a function under a given variable ordering, Bryant [5] provides a proof that this is a unique canonical representation and provides a *Reduce* algorithm for findin this canonical representation for binary decision diagrams (BDDs) that can be easily generalized to ADDs.

Unary operations such as $\min$, and $\max$, and marginalization over variables as well as binary operations such as addition, substraction, multiplication, division, $\min$, and $\max$ can be performed efficientl on ADDs [1].

One additional benefi of the use of ADDs is that they can be efficientl pruned to reduce their size in exchange for some approximation error. For compression of an ADD F within ϵ error, the operation $\texttt{ApproxADD}(F, \epsilon)$ (Algorithm 1) can be performed by collecting all leaves of the ADD and determining which can be merged to form new values without approximating more than ϵ. The old values are then replaced with these new values creating a new (minimally reduced) ADD. An illustrative example of $\texttt{ApproxADD}(F, \epsilon)$ for ADDs is provided in Figure 2.

2.2 Affine ADDs (AADDs)

To address the limitations of ADDs, we review the affin extension to the ADD (AADD) that is capable of canonically and compactly representing context-specific additive, *and* multiplicative structure in functions from $\{0,1\}^n \rightarrow \mathbb{R}$ [12]. An example of an AADD representing the function $\sum_{i=1}^{3} x_i$ ($x_i \in \{0,1\}$) is provided in Figure 1(b) where we see that it compactly exploits additive structure in comparison to its ADD counterpart in (a).

In the following, we draw on the presentation and results from [12]. We formally defin AADDs with the following BNF:

$$\begin{aligned} G &::= c + bF \\ F &::= 0 \mid if(F^{var}, \underbrace{c_h + b_h F_h}_{true \text{ (high)}}, \underbrace{c_l + b_l F_l}_{false \text{ (low)}}) \end{aligned}$$

Here, c_h and c_l are real (or floating-point constants in the closed interval $[0,1]$, b_h and b_l are real constants in the half-open interval $(0,1]$, F^{var} is a boolean variable associated with F, and F_l and F_h are non-terminals of type F. We also impose the constraints:

1. The variable F^{var} does not appear in F_h or F_l.
2. $\min(c_h, c_l) = 0$
3. $\max(c_h + b_h, c_l + b_l) = 1$
4. If $F_h = 0$ then $b_h = 0$ and $c_h > 0$. Similarly for F_l.
5. In the grammar for G, we require that if $F = 0$ then $b = 0$, otherwise $b > 0$.

Expressions in the F grammar will be called normalized AADDs; expressions in the G grammar will be called generalized AADDs.[2]

Let $Val(\cdot, \rho)$ be the value of AADD $\cdot$ under variable value assignment ρ. This valuation can be define recursively as follows where examples of affin transforms in the expression below are shown on the branches of Figure 1(b):

$$Val(G, \rho) = c + b \cdot Val(F, \rho)$$

$$Val(F, \rho) = \begin{cases} F = 0: & 0 \\ F \neq 0 \wedge \rho(F^{var}) = true: & c_h + b_h \cdot Val(F_h, \rho) \\ F \neq 0 \wedge \rho(F^{var}) = false: & c_l + b_l \cdot Val(F_l, \rho) \end{cases}$$

Let G be the example AADD provided in Figure 1(b). Then we evaluate the case where $\rho = (x_1 = 1, x_2 = 1, x_3 = 1)$ by the following recursively evaluated expression:

$$Val(F, \rho) = 0 + 3(0.333 + 0.667(0.5 + 0.5(1 + 0))) \approx 3$$

Under a given variable ordering, generalized AADDs are canonical, i.e., two identical functions will always have identical AADD representations. Like ADDs, there is an AADD *Reduce* algorithm that will produce this minimal canonical representation [12].

The unary operations of $\min$, $\max$, and marginalization and binary operations of addition, substraction, multiplication, division,

Algorithm 1: $\texttt{ApproxADD}(DD, \epsilon)$

begin
  $leaves_{old}$=collectLeavesADD (DD);
  $\{leaves_{old} \rightarrow leaves_{new}\}$
    =mergeLeaves $(leaves_{old}, \epsilon)$;
  return createNewDD
    $(DD, \{leaves_{old} \rightarrow leaves_{new}\})$;
end

[2]Since normalized AADDs in grammar F are restricted to the range $[0,1]$, we need the top-level positive affin transform of generalized AADDs in grammar G to allow for the representation of functions with arbitrary range.

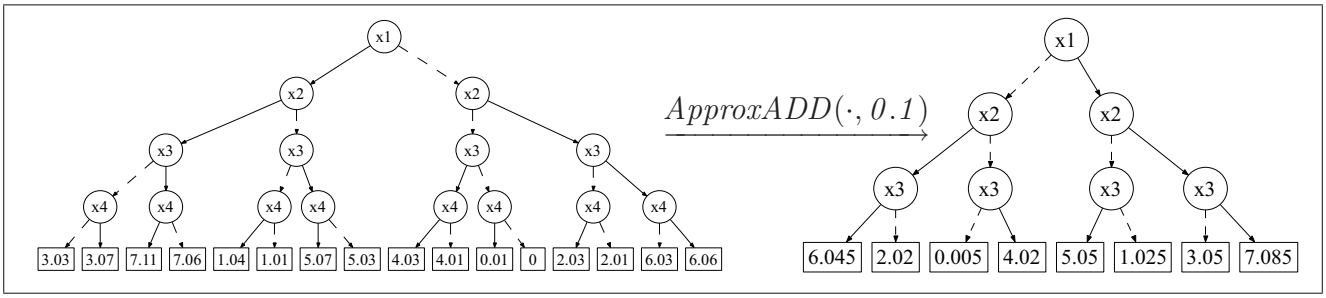

Figure 2: Compression of the ADD $\sum_{i=1}^{3} 2^i x_i + \sum_{i=1}^{4}\sum_{j=i}^{4} 0.01 x_i x_j$ within precision 0.1. Dotted lines are the low ($false$) branch and solid lines are the high ($true$) branch.

Algorithm 2: `MarkRange`$(\langle c, b, F\rangle, range, \epsilon)$

input : $\langle c, b, F\rangle$: Offset, multiplier, and node id
begin
  // Check for terminal node
  if F = 0 **or** (F *visited* **and** $F^{MaxRange} > range$) **then**
    return ;
  // Initializes error budget for current node
  $F^{\epsilon} := \epsilon$;
  // Update max range for current node
  $F^{MaxRange} := \max(F^{MaxRange}, range)$;
  // Recurse on both branches of F with updated range
  `MarkRange`$(\langle F.c_l, F.b_l, F.F_l\rangle, range \cdot b_l)$;
  `MarkRange`$(\langle F.c_h, F.b_h, F.F_h\rangle, range \cdot b_h)$;
end

min, and max can all be performed efficientl on AADDs while exploiting CSI, additive, and multiplicative structure in these operations. In the case of binary operations for AADDs, one can obtain an exponential reduction in time and space complexity over the same operations applied to ADDs and they will never perform more than a constant times worse than ADDs in time and space [12].

3. APPROXIMATION WITH AADDS

We now turn to one of the major contributions of the paper — a method for efficientl findin compact approximations of AADDs within an ϵ error budget.

Whereas it was fairly simple to approximate ADDs with ϵ error as shown in Figure 2, it is less straightforward for AADDs. The problem is that the only leaf value is 0 and that all of the value structure is stored internally in the edge-based affin transforms.

To see how we might approximate an AADD, it is best to view an example. If we jump ahead and examine Figure 4, we note that the "noisy" AADD on the left is simply the function $\sum_{i=1}^{3} 2^i x_i$ with pairwise noise factors $\sum_{i=1}^{4}\sum_{j=i}^{4} 0.01 x_i x_j$ added in. On the right hand side, we see the compressed version of this AADD representing a compact approximation of $\sum_{i=1}^{3} 2^i x_i$ without the additional pairwise noise terms that lead to branching in the AADD, since this structure can be merged or pruned within an $\epsilon = 0.1$ error budget.

How do we obtain this compressed AADD on the right hand side of Figure 4? It turns out that there are two basic operations that will allow us to recover it. However, we must firs execute `MarkRange` (Algorithm 2) on the AADD we wish to compress in order to determine the maximum contribution of any node F to the overall value (we store this value in $F^{MaxRange}$, which should be initialized to zero before running the algorithm). Incidentally, `MarkRange` also sets the F^{ϵ} property of each node F, which indicates how much of the ϵ error is left to use in potentially approximating node F. Once we've done this, we can perform the following two operations leading to a `ApproxAADD`$(\langle c, b, F\rangle, \epsilon)$ operation for AADDs that we will formally defin shortly.

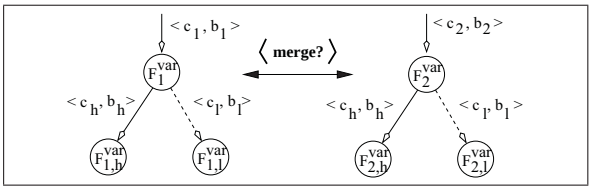

Figure 3: Two AADD nodes F_1 and F_2 (where $F_1^{var} = F_2^{var}$) with the notation used in the merging procedure. Here, we want to ask whether these two nodes can be merged while incurring less than ϵ error impact on the function?

Merge Nodes: The firs approximation procedure we might want to do is illustrated in Figure 3. Here we have two nodes F_1 and F_2 and we need to determine whether to merge them.

To see why we would want to do this, note that in Figure 4 there are many nodes that have the same variable tests, same children, and nearly identical affin transforms on their low and high branches. If they do not have the same children, we note that if their grandchildren were firs merged, they might then have the same children. These nodes and affin transforms would be identical except for an asymmetrical noise term $\sum_{i=1}^{4}\sum_{j=i}^{4} 0.01 x_i x_j$. However, we note that we can remove this noise in many cases by merging these nearly identical nodes while controlling the amount of error induced by this approximation.

To potentially merge nodes with identical high and low children, we must calculate the maximum error incurred in the function when the affin transform for the low branch of F_1 is used for the low branch of F_2 and likewise when the affin transform for the high branch of F_1 is used for the high branch of F_2:

$$\begin{aligned} error := \max(&F_1^{MaxRange}, F_2^{MaxRange}) \\ &\cdot \max(|F_1.c_l - F_2.c_l| + |F_1.b_l - F_2.b_l|, \\ &\quad |F_1.c_h - F_2.c_h| + |F_1.b_h - F_2.b_h|) \end{aligned} \tag{1}$$

If $error < \epsilon$ then we can perform a node merge where we simply replace F_1 with F_2 and update our error budget for F_2 as $F_2^{\epsilon} := F_2^{\epsilon} - error$. Clearly, the maximum merge error is just the error of the affin transform approximation multiplied by the $MaxRange$ of this node since all nodes are normalized $[0, 1]$. A slightly more complex procedure could replace both nodes with an averaged version, but this has subtle implications for AADD normalization that

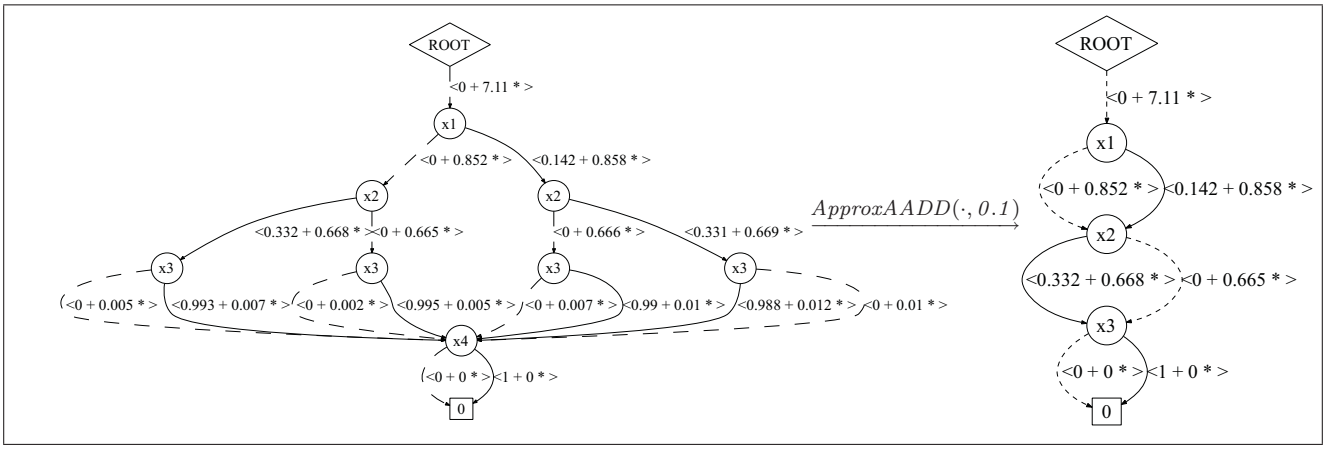

Figure 4: Compression of the AADD $\sum_{i=1}^{3} 2^i x_i + \sum_{i=1}^{4}\sum_{j=i}^{4} 0.01 x_i x_j$ within precision 0.1. Dotted lines are the low ($false$) branch and solid lines are the high ($true$) branch.

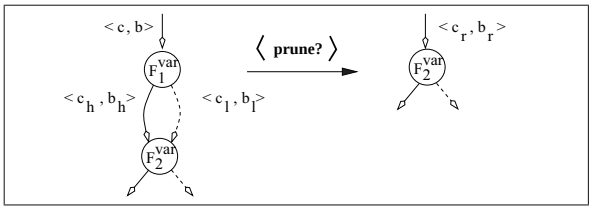

Figure 5: An AADD node F_1 with the notation used in the pruning procedure. Here, we want to ask whether F_1 can be completely pruned while incurring less than ϵ error impact on the function?

complicate the algorithm and affect its efficien y.

Prune Nodes: The second approximation procedure we might want to do is to remove a node entirely and replace it with its child in the case that it has the same child on its low and high branches as shown for F_1 in Figure 5.

To see why we would want to do this operation, note that in Figure 4, the variable x_4 has little impact (quantitatively, 0.04 or less) on the overall value of the AADD; with an allowable error budget $\epsilon = 0.1$, it can be removed entirely. This removal *cannot* be done by merging nodes, it requires pruning nodes. The error analysis for such pruning determines the error incurred if the decision for F_1 is removed:

$$error := F^{MaxRange}(|F_1.c_h - F_1.c_l| + |F_1.b_h - F_1.b_l|)/2$$

If $error < \epsilon$, we assign $c_r := c + b \cdot \frac{F_1.c_l + F_1.c_h}{2}$, $b_r := b \cdot \frac{F_1.b_h + F_1.b_l}{2}$, replace F_1 with F_2 and reduce our error budget for F_2 by $F_2^\epsilon := F_2^\epsilon - error$. Again, since all nodes are normalized $[0, 1]$, we only lose the error induced by deviation of the two affin transforms from their average multiplied by the $MaxRange$ for the node being pruned.

However, it turns out that in a greedy approximation procedure, pruning nodes can often use up most of the error budget early on in the approximation, thereby preventing the merging of nodes in later operations that could potentially save more space with less error cost. As a consequence, while we see the potential value of node pruning in Figure 5, we note that it has led to poor performance in practice so we opt not to use it here. Nonetheless, we mention it here simply because it may be useful in future work if its aggressive error consumption could somehow be controlled better (e.g., placing a lower error budget on prune operations).

Algorithm: We now formally defin the algorithm that performs AADD compression.

`ApproxAADD` (Algorithm 3) merges the nodes of the AADD by starting at the bottom level nodes and making its way up to the root nodes. By doing this, we ensure that as many child nodes are merged as possible so that merging can be performed on their parent nodes. The procedure takes any unvisited node and find all the nodes than can be merged taking into account the merge error. After that it replaces all references to merged nodes with F_1 and updates the F_1^ϵ to reflec its decreased error budget.

4. MDPS AND DYNAMIC PROGRAMMING

Having define ADDs, AADDs and efficien approximation methods for each, we now proceed to our originally stated goal: efficien approximate structured dynamic programming algorithms for factored MDPs. In this section and the next, we explain how ADD and AADD compression operations can be used to approximate structured representations of the value function to yield approximate structured dynamic programming.

4.1 Factored Representation

In the factored version of a Markov Decision Processs (MDP) [11], states will be represented by vectors $\vec{x}$ of length n, where for simplicity we assume the state variables $x_1, \ldots, x_n$ have domain $\{0, 1\}$; hence the total number of states is $N = 2^n$. We also assume a set of actions $A = \{a_1, \ldots, a_n\}$. An MDP is define by: (1) a state transition model $P(\vec{x}'|\vec{x}, a)$ which specifie the probability of the next state $\vec{x}'$ given the current state $\vec{x}$ and action a; (2) a reward function $R(\vec{x}, a)$ which specifie the immediate reward obtained by taking action a in state $\vec{x}$; and (3) a discount factor γ, $0 \leq \gamma < 1$. A policy π specifie the action $\pi(\vec{x})$ to take in each state $\vec{x}$. Our goal is to fin a policy that maximizes the value function, define using the infinit horizon, discounted reward criterion: $V^\pi(\vec{x}) = E_\pi[\sum_{t=0}^{\infty} \gamma^t \cdot r^t | \vec{x}]$, where r^t is the reward obtained at time t (starting in state $\vec{x}$).

Many MDPs often have a natural structure that can be exploited in the form of a factored MDP [3]. For example, the transition function can be factored as a dynamic Bayes net (DBN) $P(x_i'|\vec{x}_i, a)$ where each next state variable x_i' is only dependent upon the action

Algorithm 3: `ApproxAADD`($AADD = \langle c, b, F \rangle, \epsilon$)

```
begin
    MarkRange(AADD,b,ε);
    foreach variable level from bottom to top in AADD do
        foreach node F_1 in a level do
            if F_1^visited then
                continue;
            F_1^ε = min(F_{1,l}^ε, F_{1,h}^ε) ;
            foreach node F_2 in a level do
                if F_2^visited then
                    continue;
                F_2^visited = true;
                F_2^ε = min(F_{2,l}^ε, F_{2,h}^ε);
                if F_{1,l} = F_{2,l} and F_{1,h} = F_{2,h} then
                    F_2^mergeErr = compute using
                                   Eq (1) with F_1, F_2;
                    err = min(F_1^ε, F_2^ε) − F_2^mergeErr;
                    if err > 0 then
                        insert F_2 in mergeList;
            if size(mergeList) > 0 then
                foreach node F_2 in mergeList do
                    F_1^MaxRange =
                        max(F_1^MaxRange, F_2^MaxRange);
                    F_1^ε = min(F_1^ε, F_2^ε);
                    replace references to F_2 with F_1;
                    F_2^visited = true;
end
```

a and its direct parents $\vec{x}_i$ in the DBN. Then the transition model can be compactly specifie as $P(\vec{x}'|\vec{x}, a) = \prod_{i=1}^{n} P(x'_i|\vec{x}_i, a)$. The reward may be factored additively as $R(\vec{x}, a) = \sum_{i=1}^{m} R_i(\vec{x}, a)$.

4.2 Dynamic Programming (DP)

Value iteration [2] is a simple dynamic programming algorithm for constructing optimal policies. We firs defin a *backup* operator B^a for action a as follows:

$$(B^a V)(\vec{x}) = \gamma \sum_{\vec{x}'} \prod_{i=1}^{n} P(x'_i|\vec{x}_i, a) V(\vec{x}') \qquad (2)$$

If π^* denotes the optimal policy and V^* its value function, then $V^*(\vec{x}) = \max_{a \in A} \left\{ \sum_{r=1}^{m} R_i(\vec{x}_r, a) + (B^a V^*)(\vec{x}) \right\}$.

Value iteration proceeds by constructing a series of t-stage-to-go value functions V^t. Setting V^0 to arbitrary values, we defin

$$V^{t+1}(\vec{x}) = \max_{a \in A} \left\{ \sum_{r=1}^{m} R_i(\vec{x}_r, a) + (B^a V^t)(\vec{x}) \right\} \qquad (3)$$

The sequence of value functions V^t produced by value iteration converges linearly to the optimal value function V^*.

To further structure the value iteration algorithm in a factored MDP, we can represent the reward constituents $R_i(\vec{x}, a)$, transition conditional probability tables (CPTs) and value function $V(\vec{x})$ as ADDs or AADDs. We call $CPT_a^{x'_i}$ the decision diagram representation for $P(x'_i|\vec{x}_i, a)$. Having done this, we note that all operations in the value iteration equation (3) can be performed directly on these ADDs or AADDs. This idea was firs introduced in SPUDD [7] and later extended to AADDs [12].

5. APPROXIMATE DP

Approximate value iteration (AVI) is an approximate dynamic programming variant of the value iteration algorithm with the additional step that after each Bellman backup, the value function may be projected onto a more compact representation while inducing some error in this projection step.

To use AVI in conjunction with ADDs or AADDs, we simply use the `ApproxADD`(V, ϵ) or `ApproxAADD`(V, ϵ) operations to approximate the structured value function V represented as an ADD or AADD with up to ϵ error on each step if the size of V grows too large. If the MDP is discounted with $\gamma < 1$, we can bound the total induced error as $\frac{\epsilon}{1-\gamma}$ due to geometric discounting of future value and therefore error. AVI with ADDs was firs introduced in APRICODD [13] and we contribute the extension to AADDs by using the AADD approximation techniques that we introduced in Section 3.

5.1 APRICODD and MADCAP Algorithms

We now formally defin the algorithm that performs approximate value iteration with decision diagrams (APRICODD if using ADDs, and MADCAP if using AADDs).

`Solve` (Algorithm 4) constructs a series of t-stage-to-go value functions V_{DD}^t that are represented as (A)ADDs. First it creates the (A)ADD representation of all DBN CPTs in the MDP and initialize the firs value function to 0. The loop is repeated until a maximum number of iterations or until a Bellman error $BE = max_{\vec{x}}|V^t(\vec{x}) - V^{t-1}(\vec{x})|$ termination condition ($BE < tol$) is met. At each iteration the `Regress` algorithm is called and V_{DD}^t is updated with the max over all Q_{DD}^t for each action a computed by `Regress`(V_{DD}^{t-1}, a). After this, BE is computed and tested for termination. If the algorithm does not terminate, then we approximate the (A)ADD up to $\delta \cdot Vmax$ via the `ApproxADD` or `ApproxAADD` procedure calls. By making the approximation error sensitive to $Vmax$ we prevent over-aggressive value approximation in the initial stages of AVI when values are relatively small as suggested in APRICODD [13]. If $\delta = 0$ and ADDs are used, this algorithm reduces to SPUDD [7].

`Regress` (Algorithm 5) computes Q_{DD}^t, i.e, it regresses V_{DD}^{t-1} through action a that provides the values Q_{DD}^t that could be obtained if executing a and acting so as to obtain V_{DD}^{t-1} thereafter. During regression we "prime" the variables by converting each X_i to X'_i (since the V_{DD}^i is now part of the "next" state) and the CPTs for action a are multiplied in and summed out.[3] Finally, the future value is discounted and the reward (A)ADD is added in to complete the regression.

5.2 Reducing Numerical Error in MADCAP

One observed difficult with AADDs is that the constant multiplication, subtraction, and division required to maintain normalized nodes and caches may lead to substantial accumulations of numerical precision errors [12]. In this section, we contribute new techniques to improve the numerical stability of of AADD computations used in MADCAP.

The firs improvement over previous MDP algorithms based on AADDs is to adjust the variable ordering in the AADD. When solving an MDP using decision diagrams it is necessary to represent

[3]For ADDs, we assume there are no synchronic arcs among variables X'_i, X'_j for $i \neq j$ in the DBN. If sychronic arcs are present, the algorithm can be modifie to multiply in all relevant CPTs. For AADDs we do this pre-multiplication by default as described below.

Algorithm 4: `Solve`(MDP, tol, $maxIter$, δ)

begin
  Create (A)ADD CPTs $CPT_a^{x_i'}$ for MDP;
  $V_{DD}^0 = 0$;
  $Vmax = \max(R_{DD})$;
  $t = 0$;
  while $i < maxIter$ **do**
    $t = t + 1$;
    $V_{DD}^t = -\infty$;
    foreach $a \in A$ **do**
      Q_{DD}^t=`Regress`$(V_{DD}^{t-1}, a, \delta \cdot Vmax)$;
      V_{DD}^t=max(V_{DD}^t, Q_{DD}^t);
    $Diff_{DD} = V_{DD}^t \ominus V_{DD}^{t-1}$;
    $BE = \max(\max(Diff_{DD}), -\min(Diff_{DD}))$;
    if $BE < tol$ **then**
      break;
    $V_{DD}^t =$ `Approx(A)ADD` $(V_{DD}^t, \delta \cdot Vmax)$;
    $Vmax = \max(R_{DD}) + \gamma Vmax$;
  return V_{DD}^t ;
end

Algorithm 5: `Regress`(V_{DD}, a, ϵ)

begin
  $Q_{DD} =$ `convertToPrimes`(V_{DD});
  for *all* X_i' *in* Q_{DD} **do**
    $Q_{DD} = Q_{DD} \otimes CPT_a^{x_i'}$;
    $Q_{DD} = \sum_{x_i' \in X_i'} Q_{DD}$;
  return $R_{DD} \oplus (\gamma \otimes Q_{DD})$;
end

functions containing both the current state variables, $\vec{x}$, and the next state variables, $\vec{x}'$. We found that interleaving the current and next state variables was the most effective ordering; $x_1, x_1', x_2, x_2', \ldots$. Choosing the ordering of the state variables themselves is also important, but problem dependent.

The second optimization builds on the original SPUDD work [7] that notes computation time can be accelerated if the transition function conditional probability tables (CPTs) are premultiplied CPTs (since they will have to be multiplied together at some point). However, when transition effects are exogenous and independent (i.e., not correlated with the agent's action or each other), ADDs can not exploit the factored multiplicative structure in the joint transition model, thus leading to a representational blowup that makes premultiplying CPTs impractical in these cases. Because the AADD can compactly represent multiplicative structure, this same blowup is not incurred for premultiplying CPTs represented as AADDs and thus the premultiplication optimization can be practically used in all factored MDPs on which we have experimented.

The third optimization introduced is to implement the simultaneous marginalization of multiple variables. We defin a recursive function, $M(\mathcal{M}, c + bF)$, which takes a set of variables to marginalize, $\mathcal{M}$, and a de-normalized diagram to work with, $c + bF$, and returns a de-normalized diagram containing the result. If F is a constant node then we can return $c + bF$ with no change. Otherwise, let

$$F = \text{if}(F^{var}, c_h + b_h F_h, c_l + b_l F_l).$$

Then we can recurse on the two children of F;

$$c_h' + b_h' F_h' = M(\mathcal{M}, c_h + b_h F_h), \text{ and}$$
$$c_l' + b_l' F_l' = M(\mathcal{M}, c_l + b_l F_l).$$

If $F^{var} \in \mathcal{M}$ then we merge the result of the recursive calls using a modifie summation operator, $\odot$;

$$c_r + b_r F_r = \odot(c_h' + b_h' F_h', c_l' + b_l' F_l'),$$

and otherwise we recombine them using F^{var};

$$c_r + b_r F_r = normalize(\text{if}(F^{var}, c_h' + b_h' F_h', c_l' + b_l' F_l')).$$

Finally, we combine that result with our original de-normalisation data and return $c + b(c_r + b_r F_r) = (c + bc_r) + (bb_r)F_r$.

If a variable in $\mathcal{M}$ doesn't appear in a branch of the tree, this algorithm will not apply our modifie summation, $\odot$, to that branch. Hence the algorithm only works if the marginalization function, $\odot$, is idempotent, i.e. $\odot(F, F) = F$. As summation is not idempotent, we use average within the recursive calls and then multiply by $2^{|\mathcal{M}|}$ at the end.

Internally this function was designed to work with normalized diagrams as much as possible – the de-normalization data at one level, the constants c and b, have no effect on any of the recursive calls and are simply recombined with the result at the end of the function. This substantially reduces the effect of rounding errors at one level on other levels of the diagram.

6. EMPIRICAL RESULTS

In this section we report on a variety of experiments applying APRICODD and MADCAP to two very difficul factored MDPs, SysAdmin and Traffic, both displaying a variety of *context-specific*, *additive*, and *multiplicative structure*. Specificall , both problems involve additive structure in their reward functions and SysAdmin also contains additive structure in its DBN CPTs. Additionally, both problems have independent exogenous effects that act on each of their state variables leading to context-specifi independence in their DBN CPTs and multiplicative structure in their joint transition functions. These three types of structure are hard for ADDs and thus APRICODD to jointly exploit, but more natural to exploit with AADDs and thus hopefully problems where MADCAP excels.

To evaluate the effica y of AADDs and our AADD approximation approach in the MADCAP algorithm, we compare to the state-of-the-art ADD-based approximate dynamic programming algorithm of APRICODD introduced with MADCAP in Section 5.

6.1 Evaluation Domains

SysAdmin Factored MDP: In the SysAdmin problem [6], we have n computers $c_1, \ldots, c_n$ connected via a directed graph topology (c.f. Fig. 6). Let variable x_i denote whether computer c_i is up and running (1) or not (0). Let $Conn(c_j, c_i)$ denote a connection from c_j to c_i. We have n actions: $reboot(c_1), \ldots, reboot(c_n)$. The CPTs in the transition DBN have the following form:

$$P(x_i' = 1 | \vec{x}_i, a) = \begin{cases} a = reboot(c_i) : 1 \\ a \neq reboot(c_i) : (0.05 + 0.9 x_i) \\ \quad \cdot \frac{|\{x_j | j \neq i \wedge x_j = 1 \wedge Conn(c_j, c_i)\}| + 1}{|\{x_j | j \neq i \wedge Conn(c_j, c_i)\}| + 1} \end{cases}$$

If a computer is not rebooted then its probability of running in the next time step depends on its current status and the proportion of

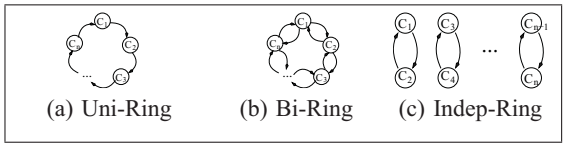

Figure 6: Diagrams of the three example SYSADMIN connection topologies that we will focus on in this paper.

computers with incoming connections that are also currently running. The reward is the sum of computers that are running at any time step: $R(\vec{x}, a) = \sum_{i=1}^{n} x_i$. We use discount factor $\gamma = 0.9$ An optimal policy in this problem will reboot the computer that has the most impact on the expected future discounted reward given the network configuration

TRAFFIC Factored MDP: This problem represents traffi intersection control. While this is not meant to be an accurate large-scale traffi model over long stretches of road, it should still approximately model near-saturation traffi fl w conditions at intersections where speeds are limited by queueing and traffi turn delays.

A diagram of the model is shown in Figure 7. The traffi state is given by $\vec{x} = (x_1, \ldots, x_n)$ where $\vec{x} \in \{O, U\}^n$ indicates for each traffi cell x_i ($1 \leq i \leq n$) if it is occupied O or unoccupied U.

Our basic traffi model for *intermediate road cells* is that a car will move forward into the next cell as long as it is unoccupied, otherwise it stops in its current cell and waits.

For each *intersection road cell* x_j (i.e., leading into an intersection), we defin a state variable $t_j \in \{turn, no\text{-}turn\}$ indicating whether a car in x_i will intend to turn into oncoming traffi or not. The state variable t_j is drawn randomly with probability $p_t = 0.5$ that a car will turn when a new car arrives. When determining the update for x_j, we note that it can always go straight or turn left on a green, but whether it can cross the opposing lane to make a right turn depends on the opposing traffi light state and the opposing traffi cell states t_o and x_o (two opposing right-turning cars may safely turn though and this is allowed by conditioning on t_o).

We refer to a boundary traffi cell x_k as a *feeder road cell* since new cars are introduced at these points. We assume that when the cell is not occupied, new cars arrive on a time step with probability $p_a = 0.2$.

Finally, we have state variables $\vec{c}$ encoding the current state of the light cycle. In Figure 7, we have $\vec{c} = (c_1, c_2, c_3, c_4)$, where one may interpret each binary c_i as indicating whether the intended light is green (or not). We allow all light configuration where exactly one c_i is green or two opposing c_i are green. The action set is simply to remain in the same light configuratio state $\vec{c}$ or to advance to the next light configuratio in a predefine sequence: $A = \{advance, no\text{-}change\}$.

The above dynamics can be compactly represented in the transition DBN of a factored MDP. The reward is the sum of road cells that are *unoccupied* at any time step: $R(\vec{x}, a) = \sum_{i=1}^{n} \mathbb{I}[x_i = U]$. We use discount factor $\gamma = 0.9$. An optimal policy in this domain will adjust the lights based on traffi fl w in all directions to minimize the number of occupied cells (i.e., congestion) in the traffi network.

6.2 Experiments

We firs begin with a variety of results for SYSADMIN problems. In Figure 8, we show the time and space required to obtain an approximate solution with no more than 5% error as the number of computers in the problem is increased. Here we note that MADCAP consistently outperforms APRICODD in both time and space requirements. In Figure 9, we show the amount of space that both

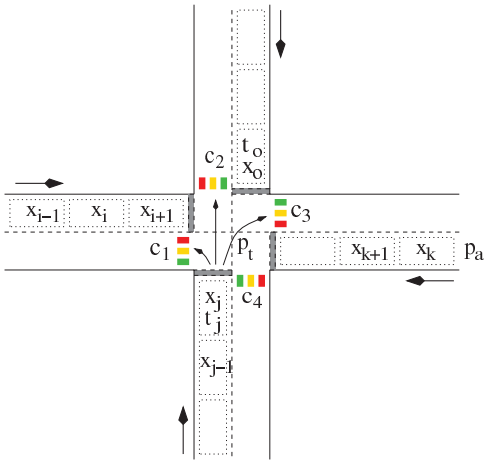

Figure 7: Diagram showing a 4-way single-lane intersection including variables used in our state description. Note that we do not model road cells that exit the intersection as we assume that cars freely exit the boundaries of the model once they have passed through the intersection.

algorithms require to achieve various true errors[4] as the approximation error is varied between 0% and 30%. Here we see that the AADD-based MADCAP is able to consistently achieve lower errors in its solution using less space than APRICODD.

Next we move onto some large TRAFFIC problems. Here we note that the largest problem has 2^{24} states and the joint transition function specifie approximately 2^{48} transition probabilities — almost all non-zero. In Figure 10, we note that while MADCAP outperforms APRICODD on the smaller of the two problems, APRICODD cannot even approximately solve the larger problem with an *a priori* bound of 10% error without exceeding memory limits (*EML*). While MADCAP also cannot *exactly* solve this problem without exceeding memory limits, we note that it can fin an approximate solution with no more than 10% error, indicating that MADCAP is capable of approximately solving problems within fi ed error bounds that APRICODD cannot.

7. CONCLUDING REMARKS

We contributed MADCAP: an approximate dynamic programming algorithm for factored MDPs based on a novel technique we introduced for efficientl and compactly approximating AADDs. In addition, we provided various enhancements for improving the numerical stability of MADCAP. We showed that the MADCAP algorithm yields faster running times, better compression and lower error rates than state-of-the-art ADD-based APRICODD on problems with *context-specific*, *additive* and *multiplicative* structure, and can approximately solve problems within fi ed error bounds on which APRICODD runs out of memory.

Acknowledgements

NICTA is funded by the Australian Government's Backing Australia's Ability and ICT Centre of Excellence programs.

8. REFERENCES

[1] R. I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their applications. In *IEEE /ACM International Conf. on CAD*, pages 428–432, 1993.

[4]True errors are the percent max error of each approximate solution $\hat{V}(s)$ w.r.t. $V^*(s)$: $\max_s |V^*(s) - \hat{V}(s)| / \max_t V^*(t)$.

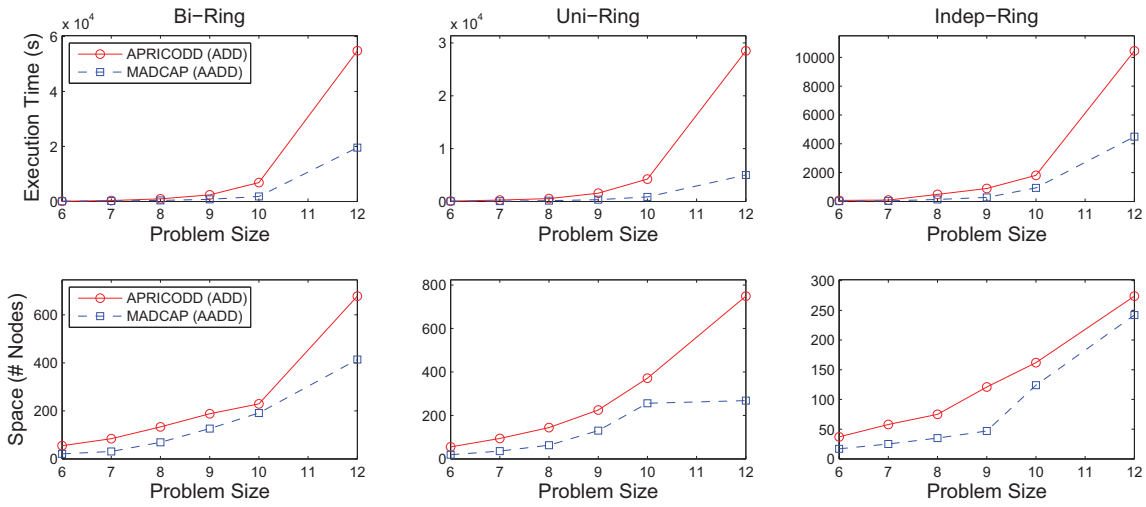

Figure 8: Time and space performance for three different SysAdmin problems as the level of pruning precision is held constant at $\delta = 0.05$ and the number of computers in SysAdmin increases. $\delta = 0.05$ enforces that the true approximation error is within $\pm$ 5% of the maximum value achievable. Results were taken after 100 iterations of approximate dynamic programming.

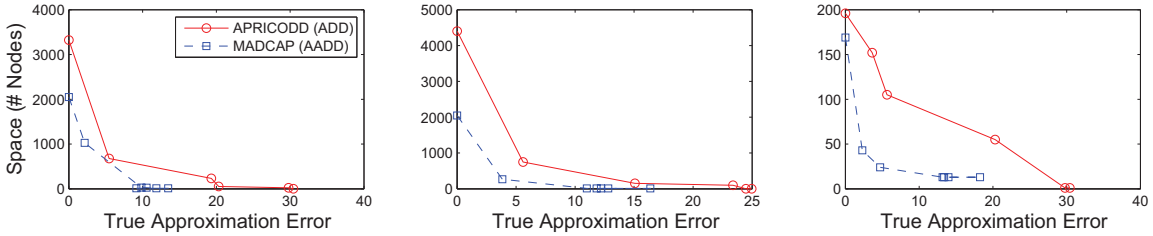

Figure 9: Space performance for three different SysAdmin problems vs. the percent of true approximation error (TAE) as the number of computers is held constant at 12 and the pruning precision varies from $\delta = 0.0, \ldots, 0.3$. When $\delta = 0$, this enforces that TAE = 0%, thus yielding the *exact* solution. Results were taken after 100 iterations of approximate dynamic programming.

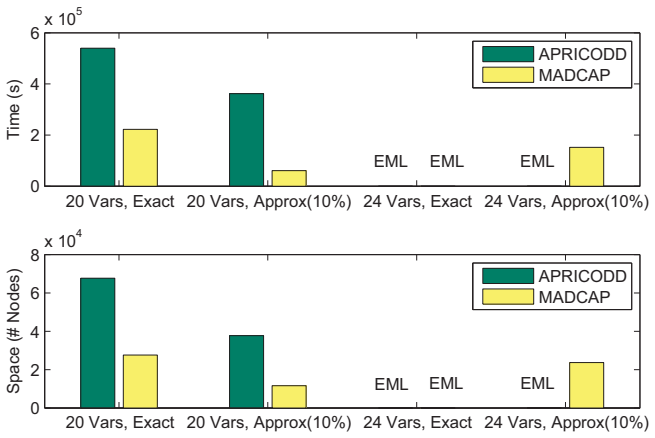

Figure 10: Experiments on large Traffic MDPs with 20 variables (2^{20} states) and 24 variables (2^{24} states). *EML* denotes that the algorithm exceeded its memory limit (1.4 Gb for these results). We note here that MADCAP was capable of finding an approximate value function within 10% of the optimal solution on the larger Traffic problem while APRICODD ran out of memory with this same error bound.

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

[3] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research (JAIR)*, 11:1–94, 1999.

[4] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specifi independence in Bayesian networks. In *Uncertainty in Artificial Intelligence (UAI-96)*, pages 115–123, 1996.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, C-35(8), Aug. 1986.

[6] C. Guestrin, D. Koller, R. Parr, and S. Venktaraman. Efficien solution methods for factored MDPs. *Journal of Artificial Intelligence Research (JAIR)*, 19:399–468, 2003.

[7] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Uncertainty in Artificial Intelligence (UAI-99)*, pages 279–288, 1999.

[8] R. Parr, L. Li, G. Taylor, C. Painter-Wakefield and M. L. Littman. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *ICML-08*, pages 752–759, 2008.

[9] R. Patrascu, P. Poupart, D. Schuurmans, C. Boutilier, and C. Guestrin. Greedy linear value-approximation for factored Markov decision processes. In *(AAAI-02)*, pages 285–291.

[10] P. Poupart, C. Boutilier, R. Patrascu, and D. Schuurmans. Piecewise linear value function approximation for factored MDPs. In *(AAAI-02)*, pages 292–299.

[11] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

[12] S. Sanner and D. McAllester. Affin algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *IJCAI 2005*, 2005.

[13] R. St-Aubin, J. Hoey, and C. Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *(NIPS-00)*, 2000.