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 Affine 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 efficiently finding 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 efficiently 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 specified 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 manageable-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 efficient 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 efficiency 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 efficiency 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 Affine 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 ADDs. 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 difficult problems on which APRICODD runs out of memory.

\(^1\)“MADCAP” contains “MDP”, “ADP”, and “AD”=A\(^2\)D\(^2\)=AADD.
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 re-convergent 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 efficient representation of functions with context-specific 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.

Defining 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 finding this canonical representation for binary decision diagrams (BDDs) that can be easily generalized to ADDs.

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

One additional benefit of the use of ADDs is that they can be efficiently pruned to reduce their size in exchange for some approximation error. For compression of an ADD \( F \) within \( \epsilon \) error, the operation \( \text{ApproxADD}(F, \epsilon) \) (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 \( \epsilon \). The old values are then replaced with these new values creating a new (minimally reduced) ADD. An illustrative example of \( \text{ApproxADD}(F, \epsilon) \) for ADDs is provided in Figure 2.

2.2 Affine ADDs (AADDs)

To address the limitations of ADDs, we review the affine 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 define AADDs with the following BNF:

\[
G ::= c + b F \\
F ::= 0 \mid \text{if}(F_{\text{var}}, c_h + b_h F_h, c_l + b_l F_l) \begin{cases} 
\text{true} & (\text{high}) \\
\text{false} & (\text{low})
\end{cases}
\]

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 \((0, 1]\). \( F_{\text{var}} \) 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 \( F_{\text{var}} \) 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 \( G \) grammar will be called normalized AADDs; expressions in the \( G \) grammar will be called generalized AADDs.\(^2\)

Let \( \text{Val}(\cdot, \rho) \) be the value of AADD \( G \) under variable value assignment \( \rho \). This valuation can be defined recursively as follows where examples of affine transforms in the expression below are shown on the branches of Figure 1(b):

\[
\text{Val}(G, \rho) = c + b \cdot \text{Val}(F, \rho)
\]

\[
\text{Val}(F, \rho) = \begin{cases} 
0 & F = 0 \\
\text{true} : c_h + b_h \cdot \text{Val}(F_h, \rho) & F \neq 0 \land \rho(F_{\text{var}}) = true \\
\text{false} : c_l + b_l \cdot \text{Val}(F_l, \rho) & F \neq 0 \land \rho(F_{\text{var}}) = false
\end{cases}
\]

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:

\[
\text{Val}(F, \rho) = 0 + 3(0.333 + 0.667(0.5 + 0.5(1 + 0))) \approx 3
\]

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, subtraction, multiplication, division, division,

\(^2\)Since normalized AADDs in grammar \( F \) are restricted to the range \([0, 1]\), we need the top-level positive affine transform of generalized ADDs in grammar \( G \) to allow for the representation of functions with arbitrary range.
Figure 2: Compression of the ADD $\sum_{i=1}^{3} 2^i x_i + \sum_{j=1}^{4} \sum_{j=0.01x_j}$ within precision 0.1. Dotted lines are the low (false) branch and solid lines are the high (true) branch.

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

```
input : (c, b, F) : Offset, multiplier, and node id

begin
    // Check for terminal node
    if F = 0 or (F visited and FMaxRange > range) then
        return;
    // Initialize error budget for current node
    $F^\epsilon$ := $c$;
    // Update max range for current node
    FMaxRange := max(FMaxRange, range);
    // Recurse on both branches of F with updated range
    MarkRange((F.c_l, F.b_l, F.F_l), range - b_l);
    MarkRange((F.c_h, F.b_h, F.F_h), range - b_h);
end
```

3. APPROXIMATION WITH ADDS

We now turn to one of the major contributions of the paper — a method for efficiently finding compact approximations of ADDs within an $\epsilon$ error budget.

Whereas it was fairly simple to approximate ADDs with $\epsilon$ 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 affine 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 $\sum_{i=1}^{4} \sum_{j=0.01x_j}$ 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 first 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 $F_{\text{MaxRange}}$, which should be initialized to zero before running the algorithm). Incidentally, MarkRange also

sets the $F^\epsilon$ property of each node $F$, which indicates how much of the $\epsilon$ 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((c, b, F), $\epsilon$) operation for AADDs that we will formally define shortly.

Merge Nodes: The first approximation procedure we might want to do is illustrated in Figure 3. Here we have two nodes $F_1$ and $F_2$ and 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 affine transforms on their low and high branches. If they do not have the same children, we note that if their grandchildren were first merged, they might then have the same children. These nodes and affine transforms would be identical except for an asymmetrical noise term $\sum_{i=1}^{4} \sum_{j=0.01x_j}$. 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 affine transform for the low branch of $F_1$ is used for the low branch of $F_2$ and likewise when the affine transform for the high branch of $F_1$ is used for the high branch of $F_2$:

$$\text{error} := \max(F_{1\text{MaxRange}}^\epsilon, F_{2\text{MaxRange}}^\epsilon) \cdot \max(|F_1.c_l - F_2.c_l| + |F_1.b_l - F_2.b_l|, |F_1.c_h - F_2.c_h| + |F_1.b_h - F_2.b_h|)$$

If $\text{error} < \epsilon$ 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 $F_2^\epsilon := F_2 - \text{error}$. Clearly, the maximum merge error is just the error of the affine 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...
We now formally define 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 finds all the nodes than can be merged taking into account the merge error. After that it replaces all references to merged nodes with the notation used in the function?

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. The procedure takes any unvisited node and finds all the nodes than can be merged taking into account the merge error. After that it replaces all references to merged nodes with the notation used in the function?

#### 4. MDPS AND DYNAMIC PROGRAMMING

Having defined ADDs, AADDS and efficient approximation methods for each, we now proceed to our originally stated goal: efficient 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 Process (MDP) [11], states will be represented by vectors \( \bar{x} \) 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 defined by: (1) a state transition model \( P(\bar{x}'|\bar{x}, a) \) which specifies the probability of the next state \( \bar{x}' \) given the current state \( \bar{x} \) and action \( a \); (2) a reward function \( R(\bar{x}, a) \) which specifies the immediate reward obtained by taking action \( a \) in state \( \bar{x} \); and (3) a discount factor \( \gamma \), \( 0 \leq \gamma < 1 \). A policy \( \pi \) specifies the action \( \pi(\bar{x}) \) to take in each state \( \bar{x} \). Our goal is to find a policy that maximizes the value function, defined using the infinite horizon, discounted reward criterion: 

\[
V(\pi) = E^\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t | \bar{x} \right],
\]

where \( r_t \) is the reward obtained at time \( t \) (starting in state \( \bar{x} \)).

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) \( P(x'_t|\bar{x}_t, a) \) where each next state variable \( x'_t \) is only dependent upon the action
Algorithm 3: ApproxAADD(AADD = (c, h, F), ε)

begin
MarkRange(AADD, h, ε);
foreach variable level from bottom to top in AADD do
  foreach node Fi in a level do
    if Fi existed then
      continue;
      F1 = min(F1fj, F1h);
    foreach node Fi in a level do
      if Fi existed then
        continue;
      F2 = min(F2fi, F2h);
      if F1f = F2f and F1h = F2h then
        FmergeErr = compute using
        Eq (1) with F1, F2;
        err = min(F1, F2) - FmergeErr;
      if err > 0 then
        insert F2 in mergeList;
  if size(mergeList) > 0 then
    foreach node F2 in mergeList do
      F1MaxRange =
      F2 = min(F1MaxRange, F2);
      replace references to F2 with F1;
      Flist = true;
end

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

4.2 Dynamic Programming (DP)

Value iteration [2] is a simple dynamic programming algorithm for constructing optimal policies. We first define a backup operator B\alpha for action a as follows:

(B\alpha V)(x) = \gamma \sum_{x'} P(x'|x, a) V(x')

If \pi denotes the optimal policy and V\pi its value function, then V\pi(x) = \max_{a \in A} \{\sum_{i=1}^{m} R_i(x, a) + (B\alpha V\pi)(x)\}.

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

V^{t+1}(x) = \max_{a \in A} \{\sum_{i=1}^{m} R_i(x, a) + (B\alpha V^t)(x)\}

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

To further structure the value iteration algorithm in a factored MDP, we can represent the reward constituents R_i(x, a), transition conditional probability tables (CPTs) and value function V\pi as ADDs or AADDs. We call CPT\pi the decision diagram representation for P(x'|x, 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 first 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 ApproxAADD(\pi, \epsilon) or ApproxAADD(\pi, \epsilon) operations to approximate the structured value function V\pi as an ADD or AADD with up to \epsilon 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{1}{1-\gamma} due to geometric discounting of future value and therefore error. AVI with ADDs was first 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 define 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^t of ADDs that are represented as (A)ADDs. First it creates the (A)ADD representation of all DBN CPTs in the MDP and initializes the first value function to 0. The loop is repeated until a maximum number of iterations or until a Bellman error BE = max_h\max_a[V^{t+1}(x) - V^{t-1}(x)] termination condition (BE < tol) is met. At each iteration the Regress algorithm is called and V^t is updated with the max over all Q^t_{DD,D} for each action a computed by Regress(V^{t-1}_{DD,D}, 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 ApproxAADD 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^t_{DD,D}, i.e., it regresses V^{t-1}_{DD,D} through action a that provides the values Q^t_{DD,D} that could be obtained if executing a and acting so as to obtain V^{t-1}_{DD,D} thereafter. During regression we “prime” the variables by converting each X_i to X_i' (since the V^{t-1}_{DD,D} is now part of the “next” state) and the CPTs for action a are multiplied in and summed out. 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 difficulty 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 AADD computations used in MADCAP.

The first 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

\footnote{For ADDs, we assume there are no synchronic arcs among variables X_i, X_j for i \neq j in the DBN. If synchronic arcs are present, the algorithm can be modified to multiply in all relevant CPTs. For AADDs we do this pre-multiplication by default as described below.}
functions containing both the current state variables, \( \vec{x} \), and the next state variables, \( \vec{z} \). 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 ADD can compactly represent multiplicative structure, this same blowup is not incurred for premultiplying CPTs represented as ADDs 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 define 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

\[ F = \text{if}(F^{\text{var}}, c_0 + b_0 F_h, c_1 + b_1 F_l). \]

Then we can recurse on the two children of \( F \);

\[ c_0 + b_0 F_h = M(\mathcal{M}, c_0 + b_0 F_h), \]
\[ c_1 + b_1 F_l = M(\mathcal{M}, c_1 + b_1 F_l). \]

If \( F^{\text{var}} \in \mathcal{M} \) then we merge the result of the recursive calls using a modified summation operator, \( \otimes \);

\[ c_r + b_r F_r = \text{normalize}(\text{if}(F^{\text{var}}, c_0 + b_0 F_h, c_1 + b_1 F_l)). \]

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

If a variable in \( \mathcal{M} \) doesn’t appear in a branch of the tree, this algorithm will not apply our modified summation, \( \otimes \), to that branch. Hence the algorithm only works if the marginalization function, \( \otimes \), is idempotent, i.e. \( \otimes(F, F) = F \). 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 recominded 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 difficult factored MDPs, SYSADMIN and TRAFFIC, both displaying a variety of context-specific, additive, and multiplicative structure. Specifically, 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-specific 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 ADDs and thus hopefully problems where MADCAP excels.

To evaluate the efficacy of ADDs and our ADD 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 \( \text{Conn}(c_j, c_i) \) denote a connection from \( c_j \) to \( c_i \). We have \( n \) actions: \( \text{reboot}(c_1, \ldots, \text{reboot}(c_n) \).

The CPTs in the transition DBN have the following form:

\[
P(x_i' = 1|\vec{x}, a) = \begin{cases} 
\frac{a - \text{reboot}(c_i)}{a \neq \text{reboot}(c_i)} : 1 \\
\frac{a - \text{reboot}(c_i)}{a \neq \text{reboot}(c_i)} : (0.05 + 0.9 x_i) \\
1 & \left[ x_i \neq 0 \land \text{Conn}(c_j, c_i) \right] + 1 \\
1 & \left[ \neg x_i \lor \neg \text{Conn}(c_j, c_i) \right] + 1
\end{cases}
\]

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
computers with incoming connections that are also currently running. The reward is the sum of computers that are running at any time step: \( R(\bar{x}, a) = \sum_{i=1}^{n} x_i \). 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 traffic intersection control. While this is not meant to be an accurate large-scale traffic model over long stretches of road, it should still approximately model near-saturation traffic flow conditions at intersections where speeds are limited by queueing and traffic turn delays.

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

Our basic traffic 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 define a state variable \( t_j \in \{\text{turn}, \text{no-turn}\} \) indicating whether a car in \( x_j \) will intend to turn into oncoming traffic 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 traffic light state and the opposing traffic cell states \( t_a \) and \( x_a \) (two opposing right-turning cars may safely turn though and this is allowed by conditioning on \( t_a \)).

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

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: \( R(\bar{x}, a) = \sum_{i=1}^{n} \mathbb{1}[x_i = U] \). We use discount factor \( \gamma = 0.9 \). An optimal policy in this domain will adjust the lights based on traffic flow in all directions to minimize the number of occupied cells (i.e., congestion) in the traffic network.

### 6.2 Experiments

We first 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 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 specifies approximately \( 2^{28} \) 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 \emph{a priori} bound of 10% error without exceeding memory limits (EML). While MADCAP also cannot \emph{exactly} solve this problem without exceeding memory limits, we note that it can find an approximate solution with no more than 10% error, indicating that MADCAP is capable of approximately solving problems within fixed 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 efficiently 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 fixed 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.

**REFERENCES**


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


