Affine Algebraic Decision Diagrams (AADDs) and their Application to Structured Probabilistic Inference

Scott Sanner
Department of Computer Science
University of Toronto
Toronto, ON M5S 3H5, CANADA
ssanner@cs.toronto.edu

David McAllester
TTI at Chicago
1427 East 60th Street
Chicago, IL 60637, USA
mcallester@tti-c.org

Abstract

We propose an affine extension to ADDs (AADD) capable of compactly representing context-specific, additive, and multiplicative structure. We show that the AADD has worst-case time and space performance within a multiplicative constant of that of ADDs, but that it can be linear in the number of variables in cases where ADDs are exponential in the number of variables. We provide an empirical comparison of tabular, ADD, and AADD representations used in standard Bayes net and MDP inference algorithms and conclude that the AADD performs at least as well as the other two representations, and often yields an exponential performance improvement over both when additive or multiplicative structure can be exploited. These results suggest that the AADD is likely to yield exponential time and space improvements for a variety of probabilistic inference algorithms that currently use tables or ADDs.

1 Introduction

Algebraic decision diagrams (ADDs) [1] provide an efficient means for representing and performing arithmetic operations on functions from a factored boolean domain to a real-valued range (i.e., $\mathbb{B}^n \rightarrow \mathbb{R}$). 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 – 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.

As shown in Figure 1, ADDs often provide an efficient representation of functions with context-specific independence [2], such as functions whose structure is conjunctive (1a) or disjunctive (1b) in nature. Thus,

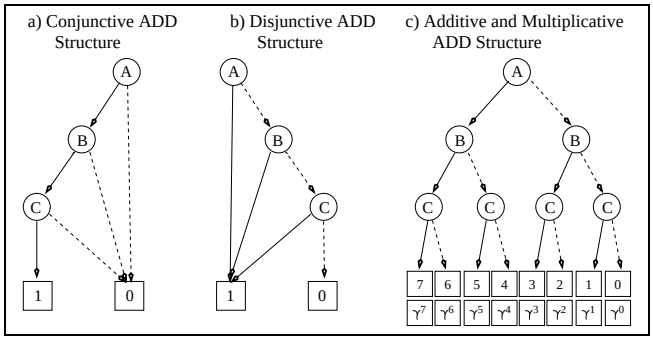

Figure 1: Some example ADDs showing a) conjunctive structure ($f = if(a \wedge b \wedge c, 1, 0)$, b) disjunctive structure ($f = if(a \vee b \vee c, 1, 0)$), and c) additive ($f = 4a + 2b + c$) and multiplicative ($f = \gamma^{4a+2b+c}$) structure (top and bottom sets of terminal values, respectively). The high (true) edge is solid, the low (false) edge is dotted.

ADDs can offer exponential space savings over a fully enumerated tabular representation. However, the compactness of ADDs does not extend to the case of additive or multiplicative independence, as demonstrated by the exponentially large representations when this structure is present (1c). Unfortunately such structure often occurs in probabilistic and decision-theoretic reasoning domains, potentially leading to exponential running times and space requirements for inference in these domains.

2 Affine Algebraic Decision Diagrams

To address the limitations of ADDs, we introduce an affine extension to the ADD (AADD) that is capable of canonically and compactly representing context-specific, additive, and multiplicative structure in functions from $\mathbb{B}^n \rightarrow \mathbb{R}$. We define AADDs with the following BNF:

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

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 following constraints:

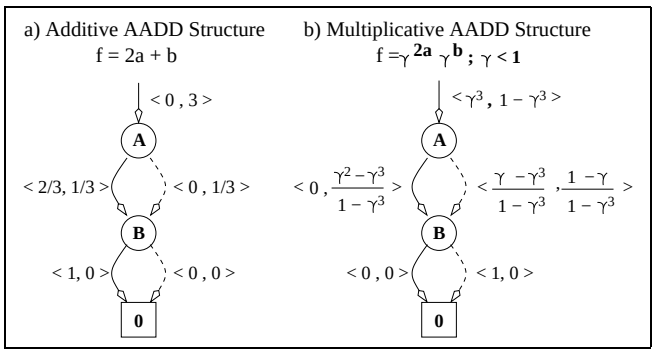

Figure 2: Portions of the ADDs from figure 1c expressed as generalized AADDs. The edge weights are given as $\langle c, b \rangle$.

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 grammar for F will be called normalized AADDs and expressions in the grammar for G will be called generalized AADDs.[1]

Let $Val(F, \rho)$ be the value of AADD F under variable value assignment ρ. This can be defined recursively by the following equation:

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

Lemma 2.1. *For any normalized AADD F we have that $Val(F, \rho)$ is in the interval $[0, 1]$, $\min_\rho Val(F, \rho) = 0$, and if $F \neq 0$ then $\max_\rho Val(F, \rho) = 1$.*

We say that F satisfies a given variable ordering if $F = 0$ or F is of the form $if(F^{var}, c_h + b_h F_h, c_l + b_l F_l)$ where F^{var} does not occur in F_h or F_l and F^{var} is the earliest variable under the given ordering occuring in F. We say that a generalized AADD of form $c + bF$ satisfies the order if F satisfies the order.

Lemma 2.2. *Fix a variable ordering. For any non-constant function g mapping $\mathbb{B}^n \longrightarrow \mathbb{R}$, there exists a unique generalized AADD G satisfying the given variable ordering such that for all $\rho \in \mathbb{B}^n$ we have $g(\rho) = Val(G, \rho)$.*

Both lemmas can be proved by straightforward induction on n. The second lemma shows that under a given variable ordering, generalized AADDs are canonical, i.e., two identical functions will always have identical AADD representations.

As an example of two AADDs with compact additive and multiplicative structure, Figure 2 shows portions of the exponentially sized ADDs from Figure 1c represented by generalized AADDs of linear size.

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

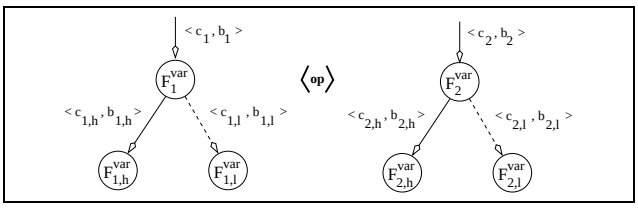

Figure 3: Two AADD nodes F_1 and F_2 and a binary operation *op* with the corresponding notation used in this paper.

We let *op* denote a binary operator on AADDs with possible operations being addition, substraction, multiplication, division, min, and max denoted respectively as $\oplus$, $\ominus$, $\otimes$, $\oslash$, $\min(\cdot, \cdot)$, and $\max(\cdot, \cdot)$.

3 Related Work

There has been much related work in the formal verification literature that has attempted to tackle additive and multiplicative structure in representation of functions from $\mathbb{B}^n \to \mathbb{B}^m$. These include *BMDs, K*BMDs, EVBDDs, FEVBDDs, HDDs, and *PHDDs [3]. While space limitations prevent us from discussing all of the differences between AADDs and these data structures, we note two major differences: 1) These data structures have different canonical forms relying on GCD factorization and do not satisfy the invariant property of AADDs that internal nodes have range $[0, 1]$. 2) The fact that these data structures have integer terminals ($\mathbb{B}^m$) requires a rational or direct floating-point representation of values in $\mathbb{R}$. In our experience, this renders them unusable for probabilistic inference due to considerable numerical precision error and computational difficulties.

4 Algorithms

We now define AADD algorithms that are analogs of those given by Bryant [4][2] except that they are extended to propagate the affine transform of the edge weights on recursion and to compute the normalization of the resulting node on return.

All algorithms rely on the helper function *GetGNode* given in Algorithm 1 that takes an unnormalized AADD node of the form $if(v, c_h + b_h F_h, c_l + b_l F_l)$ and returns the cached, generalized AADD node of the form $\langle c_r + b_r F_r \rangle$.[3]

4.1 Reduce

The *Reduce* algorithm given in Algorithm 2 takes an arbitrary ordered AADD, normalizes and caches the internal nodes, and returns the corresponding generalized AADD. One nice property of the *Reduce* algorithm is that one does not need to prespecify the structure that

[2]Bryant gives algorithms for binary decision diagrams (BDDs), but they are essentially identical to those for ADDs.

[3]In the algorithms we use the representation $\langle c, b, F \rangle$, while in the text we often use the equivalent notation $\langle c + bF \rangle$ to make the node semantics more clear.

Algorithm 1: $GetGNode(v, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle) \longrightarrow \langle c_r, b_r, F_r\rangle$

input : $v, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle$: Var, offset, mult, and node id for high/low branches
output : $\langle c_r, b_r, F_r\rangle$: Return values for offset, multiplier, and canonical node id
begin
  // If branches redundant, return child
  if $(c_l = c_h \wedge b_l = b_h \wedge F_l = F_h)$ **then**
    return $\langle c_l, b_l, F_l\rangle$;
  // Non-redundant so compute canonical form
  $r_{min} := \min(c_l, c_h)$; $r_{max} := \max(c_l + b_l, c_h + b_h)$;
  $r_{range} := r_{max} - r_{min}$;
  $c_l := (c_l - r_{min})/r_{range}$; $c_h := (c_h - r_{min})/r_{range}$;
  $b_l := b_l/r_{range}$; $b_h := b_h/r_{range}$;
  if $(\langle v, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle\rangle \rightarrow id$ *is not in node cache)* **then**
    $id :=$ currently unallocated id;
    insert $\langle v, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle\rangle \rightarrow id$ in cache;
  // Return the cached, canonical node
  return $\langle r_{min}, r_{range}, id\rangle$;
end

Algorithm 2: $Reduce(\langle c, b, F\rangle) \longrightarrow \langle c_r, b_r, F_r\rangle$

input : $\langle c, b, F\rangle$: Offset, multiplier, and node id
output : $\langle c_r, b_r, F_r\rangle$: Return values for offset, multiplier, and node id
begin
  // Check for terminal node
  if *(F = 0)* **then**
    return $\langle c, 0, 0\rangle$;
  // Check reduce cache
  if $(F \rightarrow \langle c_r, b_r, F_r\rangle$ *is not in reduce cache)* **then**
    // Not in cache, so recurse
    $\langle c_h, b_h, F_h\rangle := Reduce(c_h, b_h, F_h)$;
    $\langle c_l, b_l, F_l\rangle := Reduce(c_l, b_l, F_l)$;
    // Retrieve canonical form
    $\langle c_r, b_r, F_r\rangle := GetGNode(F^{var}, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle)$;
    // Put in cache
    insert $F \rightarrow \langle c_r, b_r, F_r\rangle$ in reduce cache;
  // Return canonical reduced node
  return $\langle c + b \cdot c_r, b \cdot b_r, F_r\rangle$;
end

the AADD should exploit. If the represented function contains context-specific, additive, or multiplicative independence, the *Reduce* algorithm will compactly represent this structure uniquely and automatically w.r.t. the variable ordering as guaranteed by previous lemmas.

4.2 Apply

The *Apply* routine given in Algorithm 3 takes two generalized AADD operands and an operation as given in Figure 3 and produces the resulting generalized AADD. The control flow of the algorithm is straightforward: We first check whether we can compute the result immediately, otherwise we normalize the operands to a canonical form and check if we can reuse the result of a previously cached computation. If we can do neither of these, we then choose a variable to branch on and recursively call the *Apply* routine for each instantiation of the variable. We cover these steps in-depth in the following sections.

Terminal computation

The function *ComputeResult* given in the *top half* of Table 1, determines if the result of a computation can be immediately computed without recursion. The first entry in this table is required for proper termination of the algorithm as it computes the result of an operation applied to two terminal 0 nodes. However, the other entries denote a number of pruning optimizations that immediately return a node without recursion. For example, given the operation $\langle 3 + 4F_1\rangle \oplus \langle 5 + 6F_1\rangle$, we can immediately return the result $\langle 8 + 10F_1\rangle$.

Recursive computation

If a call to *Apply* is unable to immediately compute a result or reuse a previously cached computation, we must recursively compute the result. For this we have two cases (the third case where both operands are 0 terminal nodes having been taken care of in the previous section):

F_1 or F_2 is a 0 terminal node, or $F_1^{var} \neq F_2^{var}$: We assume the operation is commutative and reorder the operands so that F_1 is the 0 node or the operand whose variable comes *later* in the variable ordering.[4] Then we propagate the affine transform to each of F_2's branches and compute the operation applied separately to F_1 and *each* of F_2's high and low branches. We then build an if statement conditional on F_2^{var} and normalize it to obtain the generalized AADD node $\langle c_r, b_r, F_r\rangle$ for the result:

$$\langle c_h, b_h, F_h\rangle = Apply(\langle c_1, b_1, F_1\rangle, \langle c_2 + b_2 c_{2,h}, b_2 b_{2,h}, F_{2,h}\rangle, op)$$
$$\langle c_l, b_l, F_l\rangle = Apply(\langle c_1, b_1, F_1\rangle, \langle c_2 + b_2 c_{2,l}, b_2 b_{2,l}, F_{2,l}\rangle, op)$$
$$\langle c_r, b_r, F_r\rangle = GetGNode(F_2^{var}, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle)$$

F_1 and F_2 are non-terminal nodes and $F_1^{var} = F_2^{var}$: Since the variables for each operand match, we know the result $\langle c_r, b_r, F_r\rangle$ is simply a generalized if statement branching on F_1^{var} $(= F_2^{var})$ with the true case being the operator applied to the high branches of F_1 and F_2 and likewise for the false case and the low branches:

$$\langle c_h, b_h, F_h\rangle = Apply(\langle c_1 + b_1 c_{1,h}, b_1 b_{1,h}, F_{1,h}\rangle, \langle c_2 + b_2 c_{2,h}, b_2 b_{2,h}, F_{2,h}\rangle, op)$$
$$\langle c_l, b_l, F_l\rangle = Apply(\langle c_1 + b_1 c_{1,l}, b_1 b_{1,l}, F_{1,l}\rangle \langle c_2 + b_2 c_{2,l}, b_2 b_{2,l}, F_{2,l}\rangle, op)$$
$$\langle c_r, b_r, F_r\rangle = GetGNode(F_1^{var}, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle)$$

[4] We note that the first case prohibits the use of the non-commutative $\ominus$ and $\oslash$ operations. However, a simple solution would be to recursively descend on either F_1 or F_2 rather than assuming commutativity and swapping operands to ensure descent on F_2. To accommodate general non-commutative operations, we have used this alternate approach in our specification of the *Apply* routine given in Algorithm 3.

Canonical caching
If the AADD *Apply* algorithm were to compute and cache the results of applying an operation directly to the operands, the algorithm would provably have the same time complexity as the ADD *Apply* algorithm. Yet, if we were to compute $\langle 0+1F_1\rangle \oplus \langle 0+2F_2\rangle$ and cache the result $\langle c_r + b_r F_r\rangle$, we could compute $\langle 5+2F_1\rangle \oplus \langle 4+4F_2\rangle = \langle (2c_r+9)+2b_r F_r\rangle$ without recursion.

This suggests a canonical caching scheme that normalizes all cache entries to increase the chance of a cache hit. The actual result can then be easily computed from the cached result by reversing the normalization. This ensures optimal reuse of the *Apply* operations cache and can lead to an exponential reduction in running time over the non-canonical caching version.

We introduce two additional functions to perform this caching: $GetNormCacheKey$ to compute the canonical cache key, and $ModifyResult$ to reverse the normalization in order to compute the actual result. These algorithms are summarized in the *bottom half* of Table 1.

4.3 Other Operations

While space limitations prevent us from covering all of the operations that can be performed (efficiently) on AADDs, we briefly summarize some of them here:

min and max computation: The min and max of a generalized AADD node $\langle c + bF\rangle$ are respectively c and $c+b$ due to $[0,1]$ normalization of F.

Restriction: The restriction of a variable x_i in a function to either *true* or *false* (i.e. $F|_{x_i=T/F}$) can be computed by returning the proper branch of nodes containing that test variable during the *Reduce* operation.

Sum out/marginalization: A variable x_i can be summed (or marginalized) out of a function F simply by computing the sum of the restricted functions (i.e. $F|_{x_i=T} \oplus F|_{x_i=F}$).

Negation/reciprocation: While it may seem that negation of a generalized AADD node $\langle c + bF\rangle$ would be as simple as $\langle -c + -bF\rangle$, we note that this violates our normalization scheme which requires $b > 0$. Consequently, negation must be performed explicitly as $0 \ominus \langle c + bF\rangle$. Likewise, reciprocation (i.e., $\frac{1}{\langle c+bF\rangle}$) must be performed explicitly as $1 \oslash \langle c + bF\rangle$.

Variable reordering: A generalization of Rudell's ADD variable reordering algorithm [5] that recomputes edge weights of reordered nodes can be applied to AADDs without loss of efficiency.

4.4 Cache Implementation

If one were to use a naive cache implementation that relied on exact floating-point values for hashing and equality testing, one would find that many nodes which *should* be the same under exact computation often turn out to have offsets or multipliers differing by $\pm 1e$-15 due to numerical precision error. This can result in an exponential explosion of nodes if not controlled. Consequently, it is better to use a hashing scheme that considers equality within some range of numerical precision error ϵ. While it is difficult to guarantee such an exact property for an *efficient* hashing scheme, we next outline an approximate approach that we have found to work both efficiently and nearly optimally in practice.

The node cache used in *GetGNode* and the operation result cache used in *Apply* both use cache keys containing four floating-point values (i.e., the offsets and multipliers

Algorithm 3: $Apply(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op) \longrightarrow \langle c_r, b_r, F_r\rangle$

input : $\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op$: Nodes and op
output : $\langle c_r, b_r, F_r\rangle$: Generalized node to return
begin
  // Check if result can be immediately computed
  if *($ComputeResult(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op) \rightarrow \langle c_r, b_r, F_r\rangle$ is not null)* **then**
    return $\langle c_r, b_r, F_r\rangle$;
  // Get normalized key and check apply cache
  $\langle\langle c'_1, b'_1\rangle, \langle c'_2, b'_2\rangle\rangle :=$
    $GetNormCacheKey(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op)$;
  if *($\langle\langle c'_1, b'_1, F_1\rangle, \langle c'_2, b'_2, F_2\rangle, op\rangle \rightarrow \langle c_r, b_r, F_r\rangle$ is not in apply cache)* **then**
    // Not terminal, so recurse
    if *(F_1 is a non-terminal node)* **then**
      if *(F_2 is a non-terminal node)* **then**
        if *(F_1^{var} comes before F_2^{var})* **then**
          $var := F_1^{var}$;
        else
          $var := F_2^{var}$;
      else
        $var := F_1^{var}$;
    else
      $var := F_2^{var}$;
    // Propagate affine transform to branches
    if *(F_1 is non-terminal $\wedge$ $var = F_1^{var}$)* **then**
      $F_l^{v1} := F_{1,l}$; $F_h^{v1} := F_{1,h}$;
      $c_l^{v1} := c'_1 + b'_1 \cdot c_{1,l}$; $c_h^{v1} := c'_1 + b'_1 \cdot c_{1,h}$;
      $b_l^{v1} := b'_1 \cdot b_{1,l}$; $b_h^{v1} := b'_1 \cdot b_{1,h}$;
    else
      $F_{l/h}^{v1} := F_1$; $c_{l/h}^{v1} := c'_1$; $b_{l/h}^{v1} := b'_1$;
    if *(F_2 is non-terminal $\wedge$ $var = F_2^{var}$)* **then**
      $F_l^{v2} := F_{2,l}$; $F_h^{v2} := F_{2,h}$;
      $c_l^{v2} := c'_2 + b'_2 \cdot c_{2,l}$; $c_h^{v2} := c'_2 + b'_2 \cdot c_{2,h}$;
      $b_l^{v2} := b'_2 \cdot b_{2,l}$; $b_h^{v2} := b'_1 \cdot b_{2,h}$;
    else
      $F_{l/h}^{v2} := F_2$; $c_{l/h}^{v2} := c'_2$; $b_{l/h}^{v2} := b'_2$;
    // Recurse and get cached result
    $\langle c_l, b_l, F_l\rangle := Apply(\langle c_l^{v1}, b_l^{v1}, F_l^{v1}\rangle, \langle c_l^{v2}, b_l^{v2}, F_l^{v2}\rangle, op)$;
    $\langle c_h, b_h, F_h\rangle := Apply(\langle c_h^{v1}, b_h^{v1}, F_h^{v1}\rangle, \langle c_h^{v2}, b_h^{v2}, F_h^{v2}\rangle, op)$;
    $\langle c_r, b_r, F_r\rangle := GetGNode(var, \langle c_h, b_h, F_h\rangle, \langle c_l, b_l, F_l\rangle)$;
    // Put result in apply cache and return
    insert $\langle c'_1, b'_1, F_1, c'_2, b'_2, F_2, op\rangle \rightarrow \langle c_r, b_r, F_r\rangle$ into apply cache;
  return $ModifyResult(\langle c_r, b_r, F_r\rangle)$;
end

$ComputeResult(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op) \longrightarrow \langle c_r, b_r, F_r\rangle$	
Operation and Conditions	**Return Value**
$\langle c_1 + b_1F_1\rangle \ \langle op\rangle \ \langle c_2 + b_2F_2\rangle$; $F_1 = F_2 = 0$	$\langle (c_1 \ \langle op\rangle \ c_2) + 0 \cdot 0\rangle$
$\max(\langle c_1 + b_1F_1\rangle, \langle c_2 + b_2F_2\rangle)$; $c_1 + b_1 \leq c_2$	$\langle c_2 + b_2F_2\rangle$
$\max(\langle c_1 + b_1F_1\rangle, \langle c_2 + b_2F_2\rangle)$; $c_2 + b_2 \leq c_1$	$\langle c_1 + b_1F_1\rangle$
$\langle c_1 + b_1F_1\rangle \oplus \langle c_2 + b_2F_2\rangle$; $F_1 = F_2$	$\langle (c_1 + c_2) + (b_1 + b_2)F_1\rangle$
$\max(\langle c_1 + b_1F_1\rangle, \langle c_2 + b_2F_1\rangle)$; $F_1 = F_2$, $(c_1 \geq c_2 \wedge b_1 \geq b_2) \vee (c_2 \geq c_1 \wedge b_2 \geq b_1)$	$c_1 \geq c_2 \wedge b_1 \geq b_2 : \langle c_1 + b_1F_1\rangle$ $c_2 \geq c_1 \wedge b_2 \geq b_1 : \langle c_2 + b_2F_1\rangle$
Note: for all max operations above, return opposite for min	
$\langle c_1 + b_1F_1\rangle \ \langle op\rangle \ \langle c_2 + b_2F_2\rangle$; $F_2 = 0, op \in \{\oplus, \ominus\}$	$\langle (c_1 \ \langle op\rangle \ c_2) + b_1F_1\rangle$
$\langle c_1 + b_1F_1\rangle \ \langle op\rangle \ \langle c_2 + b_2F_2\rangle$; $F_2 = 0, c_2 \geq 0, op \in \{\otimes, \oslash\}$	$\langle (c_1 \ \langle op\rangle \ c_2) + (b_1 \ \langle op\rangle \ c_2)F_1\rangle$
Note: above two operations can be modified to handle $F_1 = 0$ when $op \in \{\oplus, \otimes\}$	
other	*null*

$GetNormCacheKey(\langle c_1, b_1, F_1\rangle, \langle c_2, b_2, F_2\rangle, op) \longrightarrow \langle\langle c_1', b_1'\rangle\langle c_2', b_2'\rangle\rangle$ **and** $ModifyResult(\langle c_r, b_r, F_r\rangle) \longrightarrow \langle c_r', b_r', F_r'\rangle$		
Operation and Conditions	**Normalized Cache Key and Computation**	**Result Modification**
$\langle c_1 + b_1F_1\rangle \oplus \langle c_2 + b_2F_2\rangle$; $F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle 0 + 1F_1\rangle \oplus \langle 0 + (b_2/b_1)F_2\rangle$	$\langle (c_1 + c_2 + b_1c_r) + b_1b_rF_r\rangle$
$\langle c_1 + b_1F_1\rangle \ominus \langle c_2 + b_2F_2\rangle$; $F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle 0 + 1F_1\rangle \ominus \langle 0 + (b_2/b_1)F_2\rangle$	$\langle (c_1 - c_2 + b_1c_r) + b_1b_rF_r\rangle$
$\langle c_1 + b_1F_1\rangle \otimes \langle c_2 + b_2F_2\rangle$; $F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle (c_1/b_1) + F_1\rangle \otimes \langle (c_2/b_2) + F_2\rangle$	$\langle b_1b_2c_r + b_1b_2b_rF_r\rangle$
$\langle c_1 + b_1F_1\rangle \oslash \langle c_2 + b_2F_2\rangle$; $F_1 \neq 0$	$\langle c_r + b_rF_r\rangle = \langle (c_1/b_1) + F_1\rangle \oslash \langle (c_2/b_2) + F_2\rangle$	$\langle (b_1/b_2)c_r + (b_1/b_2)b_rF_r\rangle$
$\max(\langle c_1 + b_1F_1\rangle, \langle c_2 + b_2F_2\rangle)$; $F_1 \neq 0$, Note: same for min	$\langle c_r + b_rF_r\rangle = \max(\langle 0 + 1F_1\rangle, \langle (c_2 - c_1)/b_1 + (b_2/b_1)F_2\rangle)$	$\langle (c_1 + b_1c_r) + b_1b_rF_r\rangle$
any $\langle op\rangle$ not matching above: $\langle c_1 + b_1F_1\rangle \ \langle op\rangle \ \langle c_2 + b_2F_2\rangle$	$\langle c_r + b_rF_r\rangle = \langle c_1 + b_1F_1\rangle \ \langle op\rangle \ \langle c_2 + b_2F_2\rangle$	$\langle c_r + b_rF_r\rangle$

Table 1: Input and output summaries of the *ComputeResult*, *GetNormCacheKey*, and *ModifyResult* routines.

for two AADD nodes). If we consider this 4-tuple of floating-point values to be a point in Euclidean space, then we can measure the error between two 4-tuples $\langle u_1, u_2, u_3, u_4\rangle$ and $\langle v_1, v_2, v_3, v_4\rangle$ as the L_2 (Euclidean) distance between these points. As a consequence of the triangle inequality for four dimensions, we know that $\sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + (u_3 - v_3)^2 + (u_4 - v_4)^2} \leq \epsilon$ $\Rightarrow |\sqrt{{u_1}^2 + {u_2}^2 + {u_3}^2 + {u_4}^2} - \sqrt{{v_1}^2 + {v_2}^2 + {v_3}^2 + {v_4}^2}| \leq \epsilon$. This is shown graphically for two dimensions in Figure 4.

Based on this inequality, we can use the following approximate hashing scheme for $\langle v_1, v_2, v_3, v_4\rangle$: Compute the L_2 distance between $\langle v_1, v_2, v_3, v_4\rangle$ and the origin; For hashing, extract only the bits from the floating-point base representing a fractional portion greater than ϵ and use this as the integer representation of the hash key (we are effectively discretizing the distances into buckets of width ϵ); For equality, test that the true L_2 metric between $\langle u_1, u_2, u_3, u_4\rangle$ and $\langle v_1, v_2, v_3, v_4\rangle$ is less than ϵ.

While this hashing scheme does not guarantee that all 4-tuples having distance from the origin $\langle 0, 0, 0, 0\rangle$ within ϵ hash to the same bucket (some 4-tuples within ϵ could fall over bucket boundaries), we found that with bucket width $\epsilon = 1e\text{-}9$ and numerical precision error generally less than $1e\text{-}13$, there was only a small probability of two nodes equal within numerical precision straddling a bucket boundary. For the empirical results described in Section 6, this hashing scheme was sufficient to prevent any uncontrollable cases of numerical precision error.

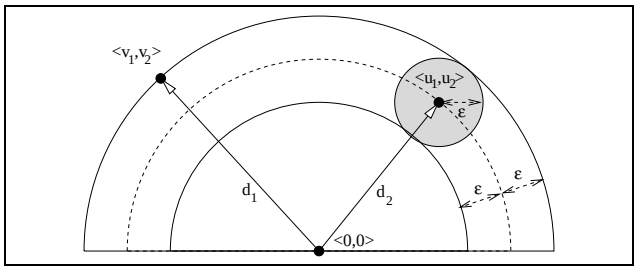

Figure 4: A 2D visualization of the hashing scheme we use. All points within ϵ of $\langle u_1, u_2\rangle$ (the dark circle) lie within the ring having outer and inner radius $\sqrt{{u_1}^2 + {u_2}^2} \pm \epsilon$. Thus, a hashing scheme which hashes all points within the ring to the *same* bucket guarantees that all points within ϵ of $\langle u_1, u_2\rangle$ also hash to the same bucket. We can use squared distances to avoid a $\sqrt{}$, thus the hash key for $\langle v_1, v_2\rangle$ can be efficiently computed as the squared distance from $\langle v_1, v_2\rangle$ to the origin.

5 Theoretical Results

Here we present two fundamental results for AADDs:

Theorem 5.1. *The time and space performance of Reduce and Apply for AADDs is within a multiplicative constant of that of ADDs in the worst case.*

Proof Sketch. Under the same variable ordering, an ADD is equivalent to a non-canonical AADD with fixed edge weights $c = 0, b = 1$. Thus the ADD *Reduce* and *Apply* algorithms can be seen as analogs of the AADD algorithms without the additional constant-time

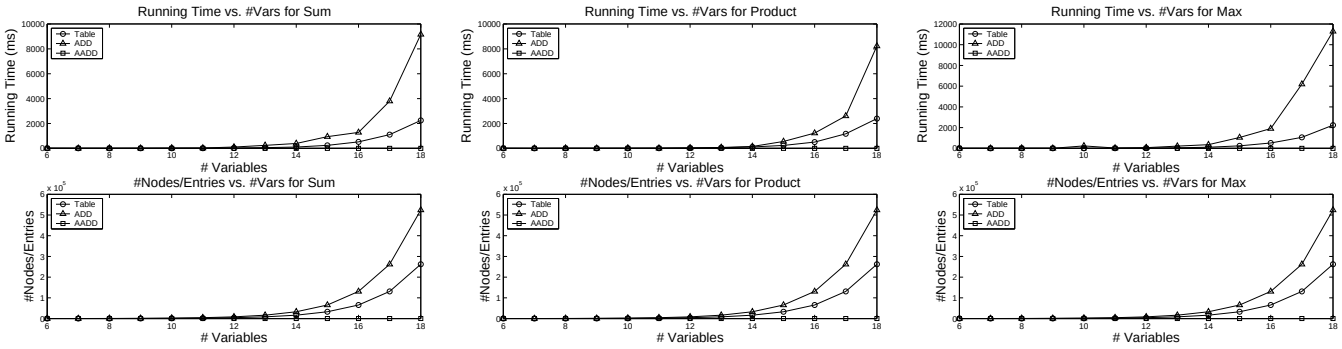

Figure 5: Comparison of *Apply* operation running time (top) and table entries/nodes (bottom) for tables, ADDs and AADDs. Left to Right: $(\sum_i 2^i x_i) \oplus (\sum_i 2^i x_i)$, $(\gamma^{\sum_i 2^i x_i}) \otimes (\gamma^{\sum_i 2^i x_i})$, $\max(\sum_i 2^i x_i, \sum_i 2^i x_i)$. Note the linear time/space for AADDs.

and constant-space overhead of normalization. Since normalization can only increase the number of *Reduce* and *Apply* cache hits and reduce the number of cached nodes, it is clear that an AADD must generate equal or fewer *Reduce* and *Apply* calls and have equal or fewer cached nodes than the corresponding ADD. This allows us to conclude that in the worst case where the AADD generates as many *Reduce* and *Apply* calls and cache hits as the ADD, the AADD is still within a multiplicative constant of the time and space required by the ADD.

Theorem 5.2. *There exist functions F_1 and F_2 and an operator op such that the running time and space performance of Apply(F_1, F_2, op) for AADDs can be linear in the number of variables when the corresponding ADD operations are exponential in the number of variables.*

Proof Sketch. Two functions and *Apply* operation examples where this holds true are $\sum_{i=1}^n 2^i x_i \oplus \sum_{i=1}^n 2^i x_i$ and $\prod_{i=1}^n \gamma^{2^i x_i} \otimes \prod_{i=1}^n \gamma^{2^i x_i}$. (Examples of these operands as ADDs and AADDs were given in Figures 1c and 2.) Because these computations result in a number of terminal values exponential in n, the ADD operations must require time and space exponential in n. On the other hand, it is known that the operands can be represented in linear-sized AADDs. Due to this structure, the *Apply* algorithm will begin by recursing on the high branch of both operands to depth n. Then, at each step as it returns and recurses down the low branch, the respective additive and multiplicative differences will be normalized out due to the canonical caching scheme, thus yielding cache hits for all low branches. This results in n cached nodes and $2n$ *Apply* calls for the AADD operations.

6 Empirical Results

6.1 Basic Operations

Figure 5 demonstrates the relative time and space performance of tables, ADDs, and AADDs on a number of basic operations. These verify the exponential to linear space and time reductions proved in Theorem 5.2.

6.2 Bayes Nets

For Bayes nets (BNs), we simply evaluate the variable elimination algorithm [6] (under a fixed, greedy tree-width minimizing variable ordering) with the conditional probability tables (CPTs) P_j and corresponding operations replaced with those for tables, ADDs, and AADDs:

$$\sum_{x_i \notin Query} \prod_{P_1 \dots P_j} P_1(x_1|Parents(x_1)) \cdots P_j(x_j|Parents(x_j))$$

Table 2 shows the total number of table entries/nodes required to represent the original network and the total running time of 100 random queries (each consisting of one query variable and one evidence variable) for a number of publicly available BNs (*http://www.cs.huji.ac.il/labs/compbio/Repository*) and two *noisy-or* and *noisy-max* CPTs with 15 parent nodes.

Note that the intermediate probability tables were often too large for the tables or ADDs, but not the AADDs, indicating that the AADD was able to exploit additive or multiplicative structure in these cases. Also, the AADD yields an exponential to linear reduction on the *Noisy-Or-15* problem and a considerable computational speedup on the *Noisy-Max-15* problem by exploiting the additive and multiplicative structure inherent in these special CPTs [7].

6.3 Markov Decision Processes

For MDPs, we simply evaluate the value iteration algorithm [8] using a tabular representation and its extension for decision diagrams [9] to factored MDPs from the *SysAdmin* domain [10].[5] Here we simply substitute tables, ADDs, and AADDs for the reward function R, value function V, and transition model dynamics P_i in the following factored MDP value iteration update:

$$V^{t+1}(s_1, \dots, s_n) = R(s_1, \dots, s_n) + \gamma \max_a \sum_{s'_1 \dots s'_n} \left[\prod_{P_1 \dots P_n} P_i(s'_i|Parents(s'_i), a) \right] V^t(s'_1, \dots, s'_n)$$

Figure 6 shows the relative performance of value iteration until convergence within 0.01 of the optimal value

[5] In these domains, there is a network of computers whose operational status (i.e., running or crashed) at each time step is a stochastic function of the previous status of computers with incoming connections. At each time step the task of the system administrator is to choose the computer to reboot in order to maximize the sum of operational computers over a discounted reward horizon.

Bayes Net	Table		ADD		AADD	
	# Table Entries	Running Time	# ADD Nodes	Running Time	# AADD Nodes	Running Time
Alarm	1,192	2.97 s	689	2.42 s	405	1.26 s
Barley	470,294	*EML**	139,856	*EML**	60,809	207 m
Carpo	636	0.58	955	0.57 s	360	0.49 s
Hailfinder	9045	26.4 s	4511	9.6 s	2538	2.7 s
Insurance	2104	278 s	1596	116 s	775	37 s
Noisy-Or-15	65566	27.5 s	125356	50.2 s	1066	0.7 s
Noisy-Max-15	131102	33.4 s	202148	42.5 s	40994	5.8 s

Table 2: Number of table entries/nodes in the original network and variable elimination running times using tabular, ADD, and AADD representations for inference in various Bayes nets. **EML* denotes that a query exceeded the 1Gb memory limit.

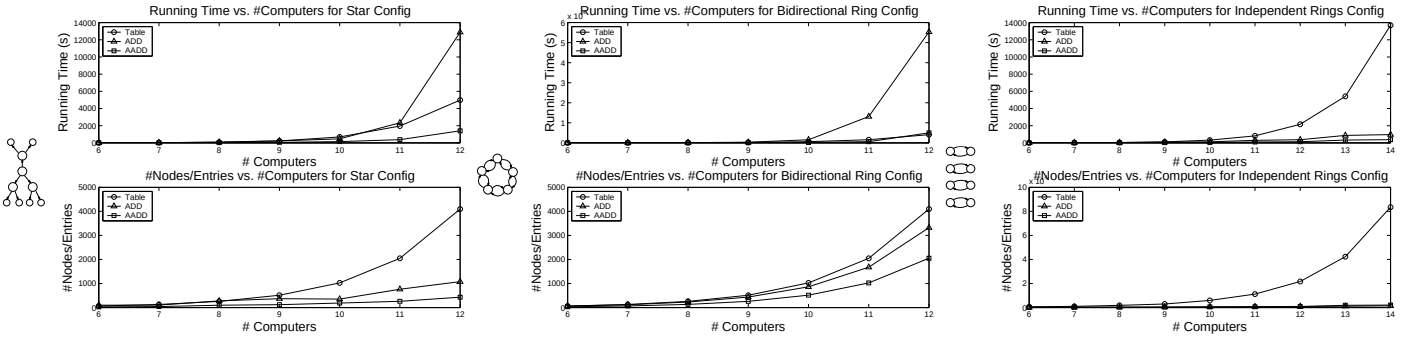

Figure 6: MDP value iteration running times (top) and number of entries/nodes (bottom) in the final value function using tabular, ADD, and AADD representations for various network configurations in the *SysAdmin* problem.

for networks in a star, bidirectional, and independent ring configuration. While the reward and transition dynamics in the *SysAdmin* problem have considerable additive structure, we note that the exponential size of the AADD (as for all representations) indicates that little additive structure survives in the *exact* value function. Nonetheless, the AADD-based algorithm still manages to take considerable advantage of the additive structure during computations and thus performs comparably or exponentially better than ADDs and tables.

7 Concluding Remarks

We have presented the AADD and have proved that its worst-case time and space performance are within a multiplicative constant of that of ADDs, but can be linear in the number of variables in cases where ADDs are exponential in the number of variables. And we have provided an empirical comparison of tabular, ADD, and AADD representations used in Bayes net and MDP inference algorithms, concluding that AADDs perform at least as well as the other two representations, and often yield an exponential time and space improvement over both.

In conclusion, we should emphasize that we have not set out to show that variable elimination and value iteration with AADDs are the best Bayes net and MDP inference algorithms available – many other approaches propose to handle similar structure efficiently via special-purpose problem-structure or algorithm modifications. Rather, we intended to show that transparently substituting AADDs in two diverse probabilistic inference algorithms that used tables or ADDs could yield exponential performance improvements over both by exploiting context-specific, additive, and multiplicative structure in the underlying representation. These results suggest that the AADD is likely to yield exponential time and space improvements for a variety of probabilistic inference algorithms that currently use tables or ADDs, or at the very least, a gross simplification of these algorithms.

References

[1] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, F. Somenzi: Algebraic Decision Diagrams and Their Applications. ICCAD. (1993)

[2] Boutilier, C., Friedman, N., Goldszmidt, M., Koller, D.: Context-specific independence in Bayesian networks. UAI. (1996)

[3] Drechsler, R., Sieling, D.: BDDs in theory and practice. Software Tools for Technology Transfer. (2001)

[4] Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Computers. (1986)

[5] Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. ICCAD. (1993)

[6] Zhang, N.L., Poole, D.: A simple approach to bayesian network computations. CCAI. (1994)

[7] Takikawa, M., D'Ambrosio, B.: Multiplicative factorization of noisy-max. UAI. (1999)

[8] Bellman, R.E.: Dynamic Programming. Princeton University Press, Princeton, NJ (1957)

[9] Hoey, J., St-Aubin, R., Hu, A., Boutilier, C.: SPUDD: Stochastic planning using decision diagrams. UAI. (1999)

[10] Guestrin, C., Koller, D., Parr, R.: Max-norm projections for factored MDPs. IJCAI. (2001)