Empirical Evaluation of AND/OR Multivalued Decision Diagrams for Compilation and Inference

William Lam and Rina Dechter

Donald Bren School of Information and Computer Sciences
University of California, Irvine, CA 92697, USA
`{willmlam,dechter}@ics.uci.edu`

Abstract. AND/OR Multi-valued Decision Diagrams (AOMDD) were shown provide a more compact representation of discrete-domain real-valued functions compared to other decision diagram variants [1]. We show the performance of AOMDDs on compilation and inference tasks in graphical models. We introduce the elimination operator to AOMDDs, which in conjunction with the combination operator introduced in previous work, yields a full bucket elimination (BE) scheme using AOMDDs as an alternative function representation to tables. For compilation, we show that we can achieve a more compact AOMDD compared to previous work. For inference, we show that we are able to solve instances that do not fit in main memory when using tables.

Keywords: graphical models, compilation, counting queries, determinism, context-specific independence

1 Introduction

AND/OR Multi-valued Decision Diagrams (AOMDDs) combine the two frameworks of AND/OR search spaces and Multi-valued Decision Diagrams (MDDs) to create a framework that compactly represents discrete-domain functions such as those in discrete graphical models [1]. The AND/OR search space is a more compact search space for search-based inference algorithms in graphical models compared to OR search spaces. For problems with decomposition into subproblems, the AND/OR search space captures this. Decision diagrams are generally used to represent functions compactly. [2]

The key algorithm for combining AOMDDs, *apply*, first introduced in [3] was never implemented before. We show empirical results of the bottom-up compilation scheme introduced in that work. Our work also extends upon previous work by introducing the elimination operator to AOMDDs. With these two operators in place (apply and elimination), this yields the full bucket elimination scheme using AOMDDs as an alternative function representation to tables. We provide the first empirical results demonstrating the algorithm and contrasting its performance with the BE algorithm using tables.

Similar work is presented in [4], where an algebraic decision diagram (ADD) structure is considered. In [5], ADDs are extended with affine transformations to capture additive and multiplicative structures in graphical models. However, AND structure is still not exploited in these alternative decision diagram variants and they are restricted to variables with binary domains.

2 Background

We start with presenting preliminaries by defining graphical models and counting queries.

Definition 1 ***(graphical model)*** *A graphical model is a tuple $\mathcal{R} = \langle X, D, F, \otimes \rangle$, where $X = \{X_1, ..., X_n\}$ is a set of variables, $D = \{D_1, ..., D_n\}$ is the set of the respective finite domains of the variables in X, $F = \{f_1, ..., f_r\}$ is a set of real-valued functions defined over a subset of variables $S_i \subseteq X$, and $\otimes$ is a combination operator $(i.e. \prod, \sum, \bowtie)$ The graphical model represents a global function computed by $\otimes_{i=1}^{r} f_i$*

Definition 2 ***(counting query)*** *For a CSP, the number of solutions is the number of assignments which do not violate any constraints. For a weighted CSP, the weighted solution count is the sum of the costs of all solutions such that no constraint is violated (having a cost of 0). For graphical models representing probability distributions, this is likelihood/ partition function computation.*

For counting queries, the combination operator is $\prod$. The task is then to find $\sum \prod_{i=1}^{r} f_i$.

Definition 3 ***(induced width/treewidth)*** *- An ordered graph is a pair (G, d), where G is the primal graph and $d = X_(1), ..., X_(n)$ is an ordering of the nodes, where $X(i)$ denotes the i^{th} node in the ordering. The width of a node is the number of neighbors that a node has that precedes it in the ordering. The induced width of the ordered graph $w^*(d)$ is the width maximum width among all nodes after processing the nodes from last to first such that when a node is processed, all of its preceding neighbors are connected. The induced width w^* of a graph is the minimal induced width over all possible orderings.*

2.1 Bucket Elimination

It can be seen that computing the global function directly results in a function with $\prod_{i=1}^{n} |D_i|$ entries, which is not feasible to compute with a large enough n. Therefore we require a way to compute a query without resorting to computing the global function, which is to separate the elimination operator into parts that eliminate over single variables and push them into the combination operation.

To formalize this process, bucket elimination is the standard framework for performing exact queries on a graphical model [6]. In BE, we construct a tree structure which guides how messages are passed based on a variable ordering.

This data structure captures the process of separating the elimination operation. The largest message generated has a scope size of $w(d)$, the induced width with respect to ordering d. The result is an algorithm with a time and space complexity of $O(nk^{w*(d)})$, where n is the number of variables, k is the maximum domain size over all variables.

2.2 AND/OR Search

AND/OR search spaces allow us to use the underlying structure of the graphical model to save on work over the standard search techniques. AND nodes represent the decomposition of the problem into independent subproblems, not captured by the standard OR search space. We first need to find a *pseudo tree* of the primal graph to define the structure of the *AND/OR search tree* for a graphical model.

Definition 4 ***(pseudo tree)*** *A pseudo tree $\mathcal{T}$ of a graph is a rooted tree with the same nodes and that each arc in the induced graph is a backarc in $\mathcal{T}$.*

Definition 5 ***(AND/OR search tree)***
An AND/OR search tree uses the structure of a pseudo tree $\mathcal{T}$. It contains: (1) OR nodes that correspond to variables in the network labeled X_i and (2) AND nodes labeled $\langle X_i, x_i \rangle$, corresponding to the values X_i can take on. Finally, each AND node has child OR nodes corresponding to the children of variable X_i in $\mathcal{T}$.

Some subtrees may actually correspond to identical subproblems, so we can merge them, yielding the *context minimal AND/OR search graph.* This captures the fact that the assignment to variables along a path are sometimes irrelevant, allowing us to take advantage of this to achieve as smaller search space.

2.3 Decision Diagrams

A *decision diagram* is a directed acyclic graph representing a function that exploits problem structure, creating a compact canonical representation. The main work that led to its many variants is based upon the *reduced ordered binary decision diagram (OBDD)*[2].

Definition 6 ***(binary decision diagram (BDD))*** *A BDD is a directed acyclic graph representing a function $f : \mathbb{B}^N \Rightarrow \mathbb{B}$. The graph has two terminal nodes* 0 *and* 1 *that represent the right-hand side of the mapping. Each internal node then corresponds to one of the N boolean variables on the left-hand side of the mapping each with two pointers to either other internal nodes or terminal nodes.*

Much like search graphs, BDDs are subject to a variable ordering. We can similarly merge isomorphic subgraphs for compactness. In addition we can remove *redundant* nodes, which are defined as nodes whose children are identical.

Applying these two properties yields the reduced ordered binary decision diagram (OBDD) [2].

Since OBDDs only represent functions of the form $f : \mathbb{B}^N \Rightarrow \mathbb{B}$, there are variants which extend to functions whose variables have arbitrary sized domains, such as multi-valued decision diagrams (MDD) [7]. Additionally, there are also variants which extend the output to real-values, such as algebraic decision diagrams (ADD) [8]. For the various extensions of decision diagrams, we can perform arbitrary binary operations between two diagrams via the *apply* operator, detailed in [2] for OBDDs. The variants have similar algorithms for performing *apply* adapted for its needs.

Weighted Decision Diagrams MDDs are sufficient for representing constraints in a CSP, as MDDs encode functions of the form $f : \mathbb{X}^N \Rightarrow \mathbb{B}$ which encode set membership over the domain $\mathbb{X}^N$.

However, we are interested in more general classes of graphical models, such as Markov networks, Bayesian networks, and weighted CSPs. To do this, decision diagrams need to represent functions of the form $f : \mathbb{X}^N \Rightarrow \mathbb{R}$.

A possible extension is to place weights on the arcs. The function's value can then be evaluated by taking the combination (product or sum, depending on the problem) over the weights encountered on a path representing a full assignment.

With this extension, we must also consider if the weights are equal when we wish to merge isomorphic subgraphs. This alone allows us rescale the any weight and the weights on adjacent levels of the diagram accordingly and have a decision diagram representing the same function. Therefore, to ensure canonicity, we normalize the weights of all arcs out of a node to 1, collecting all the normalization factors at the root of the diagram [1].

2.4 AOMDD

By augmenting context-minimal AND/OR search graphs to remove redundant nodes, we attain AOMDDs (or equivalently, augmenting weighted MDDs with AND nodes.) The basic unit of an AOMDD is the meta-node:

Definition 7 *(meta-node) A meta-node u in an AOMDD is either: (1) a terminal node labeled with 0 or 1, or (2) a nonterminal node grouping an OR node labeled with a variable X and its k AND children representing each assignment to X. Each AND node stores a set of pointers to other meta-nodes which are labeled with variables that are descendants of X in the pseudo-tree. Additionally, the AND node stores a weight for weighted graphical models.*

Rather than being subject to a variable ordering as in most decision diagram variants, the structure of an AOMDD is subject to a pseudo tree. For a function in any graphical model, the structure is guided by an embeddable pseudo tree.

Definition 8 *(embeddable pseudo tree) A pseudo tree $\mathcal{T}_1$ is embeddable in a pseudo tree $\mathcal{T}$ if the set of variables of $\mathcal{T}_1$ is a subset of the set of variables of $\mathcal{T}$ and have the same ancestor-descendant relationships.*

It is also important to know the embedded pseudo tree due to redundancy reduction. For instance, if a variable is always redundant in the function, it will not appear in the AOMDD structure. However, when performing an operation to sum over that variable, it needs to be made clear that the variable exists in the function that the AOMDD represents.

A	B	C	f
0	0	0	0
0	0	1	1
0	1	0	0.5
0	1	1	0.5
1	0	0	0.2
1	0	1	0.8
1	1	0	0.2
1	1	1	0.8

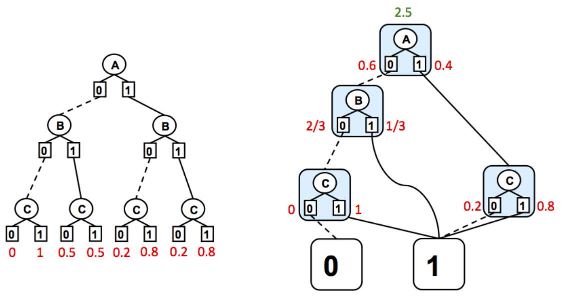


Fig. 1. Comparison of the table representation, AND/OR tree, and AOMDD. Unlabeled AND nodes in the tree are assumed to have a weight of 1. Using the AND/OR tree as a starting point, the last two C nodes are merged for isomorphism, which in turn makes the last B node redundant and is therefore removed. The second C node is redundant from the start and is removed.

There are two problem structural properties which AOMDDs mainly exploit. The first is determinism, which shows up naturally in WCSPs as hard constraints with a cost of 0 in the functions. The other property is context-specific independence [9]. We provide an alternative definition here for functions rather than conditional distributions.

Definition 9 ***(context-specific independence)***
For any function $f : \mathbb{X}_1 \times ... \times \mathbb{X}_N : \mathbb{R}$, context-specific independence exists in f iff $\exists$ an assignment A to some subset of the domain such that $f(A, \cdot)$ is constant ($\cdot$ denotes the arguments to the unassigned variables).

This has similarities to the idea behind context-minimality, in that case we require all assignments to some subset to satisfy the same condition. Essentially, context-specific independence tells us that we can merge subgraphs at a finer granularity. In an AOMDD structure, context-specific independence is captured by the removal of redundant nodes. Figure 1 demonstrates the AOMDD data structure on a small table.

It is easy to see that the number of nodes in an AOMDD is bounded by the number of entries in the table. In the worst case, the diagram is a full tree over the domains of the variables. Therefore, given k is the largest domain size and N is the number of variables in the function, an AOMDD has $\sum_{i=1}^{N} k^{i-1}$ metanodes nodes in this case, giving us $O(k^N)$.

3 Algorithms

In this work, we include the reduction rule by redundancy and use AOMDDs as an alternative to a tabular representation of the functions and messages in bucket elimination. To perform this, we require a method of applying the combination operator to AOMDDs and the elimination operators. For AOMDDs, it is presented here for the first time.

3.1 Apply

The main operation to perform the combination of two AOMDDs is the *apply* operator. It is stated that the runtime of apply is quadratic in the size of the input AOMDDs. We omit the full details of the algorithm for space issues and refer the reader to previous work [1].

3.2 Elimination

In this section, we will introduce the elimination operator for AOMDDs and discuss the differences and restrictions of the operation when compared to decision diagrams without AND decomposition.

In the case of AOMDDs, the AND/OR structure requires extra care to be taken when performing any operation different from the one chosen for problem decomposition (multiplication by default). This is due to the fact that a full assignment is represented by a subtree of the diagram rather than a path. Therefore, this means that we cannot simply perform an apply with a different operator arbitrarily; doing so may switch the order of operations.

This requires us to impose a restriction on the elimination operator such that the variable being eliminated must the bottom-most node of the embedded pseudo tree that has not already been eliminated. This guarantees that we do not run into an AND decomposition structure when eliminating a variable. At the same time, this also means that only the ancestor variables' metanodes of the eliminated variable will need to be changed.

From another viewpoint, since AOMDDs further compress a function representation by taking advantage of decomposition in the pseudo tree, operations on it are bound by the same rules as variable elimination. Namely, once we consider a fixed variable ordering, eliminating a variable whose children are not yet eliminated would induce edges in the induced graph between all of its neighbors. For the AOMDD, this would suggest that any decomposition in the embedded pseudo tree would need to be removed by converting to a chain structure. However, doing so would make AOMDDs incompatible for operations with each other as well not leveraging the computations made on decomposing the function structure.

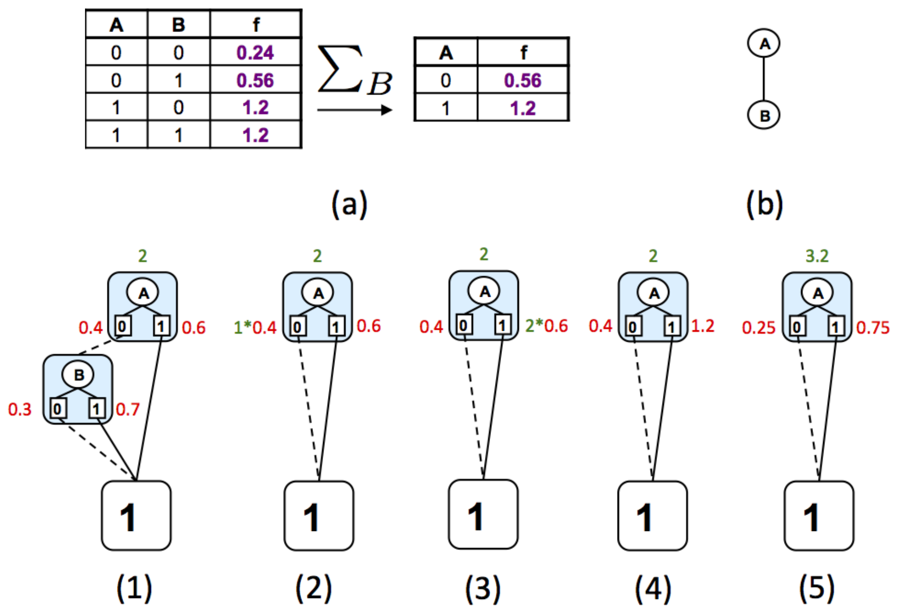
The pseudo code is given in Algorithm 1. A basic description of the algorithm is as follows. From the embedded pseudo tree of the AOMDD, we create a list of *relevant* variables by tracing a path from the leaf node representing the elimination variable to the root of the tree. This creates a direct path from the

Algorithm 1 ELIMINATE($f, eVar, \Downarrow$: elimination operator : $(\sum, \max, \min)$)

```
Input:
    AOMDD $f$ (with embedded pseudo tree $\mathcal{T}$ containing $eVar$ as a leaf)
Output:
    AOMDD $f'$ representing $\Downarrow_{eVar} f$ (with embedded pseudo tree $\mathcal{T}'$)
 1: if $f = \mathbf{0}$ or $f = \mathbf{1}$ then
 2:    Remove $eVar$ from $\mathcal{T}$
 3:    if $\Downarrow = \sum$ then
 4:       return $|eVar| * f$
 5:    else
 6:       return $f$
 7:    end if
 8: end if
 9: $R \leftarrow$ set of variables corresponding to nodes on the path from node $eVar$ to
    the root of $\mathcal{T}$
10: $bfsOrder \leftarrow$ metanodes of AOMDD $f$ in a BFS ordering corresponding to
    variables in $R$
11: $receivedWeight = \emptyset$
12: for $m \in$ reversed($bfsOrder$) do
13:    if var($m$) = elimVar then
14:       $weight \leftarrow \Downarrow (m.weights)$ // apply operator on AND node weights
15:       scale each parent's weight by $weight$
16:       insert each parent into $receivedWeight$
17:       change all references to $m$ to terminal $\delta(weight! = 0)$
18:    else
19:       if $\Downarrow = \sum$ then
20:          for $child \in m.children$ do
21:             if $child \notin receivedWeight$ then
22:                continue
23:             else if none of $child.children \in R$ then
24:                scale $child.weight$ by $|eVar|$
25:             end if
26:          end for
27:       end if
28:       if $m$ is redundant then
29:          $weight \leftarrow m.children[0].weight$
30:          scale each parent's weight by weight of any AND node (identical)
31:          change all references to $m$ to terminal $\delta(weight! = 0)$
32:       else
33:          Normalize $m$ and scale each parent's weight by the normalizer
34:       end if
35:    end if
36: end for
```

root of the AOMDD down to the elimination variable without the need to explore other branches of the AOMDD. We then create a reverse BFS ordering based on list of relevant variables. If the node is an elimination variable, we eliminate the node by performing the necessary operator and promote the weight to the parent. Otherwise, we normalize the node and pass on the normalization constant to the parent.

One caveat to note is that the metanode for a variable we are eliminating may not be present in the decision diagram due to the reduction rules. This is an issue when the elimination operator is summation. We must compensate for any missed metanodes, which we can identify if we see an ancestor of the elimination variable connected to a terminal metanode. Since nodes would be missing only if it were redundant, we can simply multiply the weight of that ancestor metanode by the domain size of the elimination variable. However, in the process after eliminating a node, the intermediate structure looks identical to that of when the input diagram does not have the node due to redundancy reduction. Therefore, we keep track of which nodes already received a weight from a child node to distinguish between these two. Lines 19-27 in the pseudocode serve this purpose.



A	B	f
0	0	0.24
0	1	0.56
1	0	1.2
1	1	1.2

$\sum_B \longrightarrow$

A	f
0	0.56
1	1.2

Fig. 2. Example of elimination on AOMDDs. The state of the AOMDD is shown through the process.

We demonstrate the algorithm on a small example, shown in Figure 3.2. The function tables above (a) the AOMDDs demonstrate the operation performed in a standard representation. We are interested in summing out variable B. The embedded pseudo tree (b) is used to determine the set of relevant variables, which in this case is $\{A, B\}$.

We begin with the AOMDD shown at (1), which represents the same function as the input table. Visiting the relevant nodes in a reverse BFS order, we visit the metanode B first, eliminate it, and propagate its result up to the parent AND node in metanode A, shown in (2). At (3), we are left with only metanode A. Checking the 0 AND node, since it received a weight from something, we are done with it. Checking the 1 AND node, since it has not received a weight, we multiply it by the domain size of B, which is 2 in this case. The result is now shown at (4). Finally, we normalize the AND node weights of metanode A and propagate its normalization term up to the root, yielding the resulting AOMDD in (5), which represents the same function as the output table.

In the cases of maximization and minimization, we do not encounter the same problem since these operators choose one from the set of values, which has no effect on functions where all the output values are identical. Conditioning can also be considered a form of elimination and also does not suffer from the issues that summation encounters for the same reasons.

3.3 Compilation

To compile a graphical model into an AOMDD, there are two methods. The first, presented in [3] is similar to bucket elimination [10]. We use the scheduling of bucket elimination, but do not eliminate variables as we work our way up the tree structure.

Since AOMDDs are based on the more specific pseudo-tree (as opposed to a variable ordering in MDDs), [3] also defines a new *apply* operator that combines two AOMDDs provided their structure is based on pseudo trees that can be embedded in some pseudo tree.

The second method performs an AND/OR search and the trace is used to build the AOMDD [1] by also performing the necessary reductions to achieve the canonical AOMDD.

The size of a AOMDD can be bounded by the induced width/treewidth of the graph, yielding $O(nk^{w*(d)})$. In practice, the size is smaller since the bound assumes the worst case where no problem structure can be exploited. For compilation of AOMDDs we use a process identical to BE, except we do not eliminate any variables in the process.

3.4 Inference

By eliminating variables in the BE compilation process, this yields a full BE algorithm for inference using AOMDDs as a function representation (AOMDD-BE). The complexity remains the same as standard BE, as in the worst case, the AOMDD has as many AND nodes as the number of entries in the table. However, for some problem structures, the AOMDD size can be far smaller than the table size.

4 Experiments

For all tables in this section, for each problem instance, we report number of variables (n), induced width (w), height of the pseudo tree (h), maximum domain size (k), time, and memory usage. The algorithms were implemented in C++ (64-bit) and the experiments were run on 2.6 GHz machines with 24GB of RAM.

4.1 Compilation

Previous work in [1] compiles AOMDDs by using the trace of an execution of AND/OR search. However, in their implementation, the redundancy rule is not applied in [1], so those AOMDDs are not as compact as possible. In this work, we compile the AOMDDs via a bucket elimination schedule. We also apply the redundancy rule by doing so on every metanode that is generated. This way, memory is freed as soon as possible at the finest granularity when operating on AOMDDs.

We evaluate the BE based compilation algorithm with redundancy reduction (BE-AOMDD+R) and the top-down compilation scheme which traces the context minimal AO search, using boolean constraint propagation to detect determinism (AOMDD-BCP).

We ran the both compilation algorithms on the ISCAS problems within the UAI 2006 evaluation set and only BE-AOMDD+R on the protein side-chain prediction networks.

name	n	w	h	k	# functions	time (s) [BE-AOMDD+R] [AOMDD-BCP]	CM OR	Metanodes [BE-AOMDD+R] [AOMDD-BCP]	Memory Usage (MB)	Compiled AOMDD mem (MB)
BN_42	850	20	50	2	879	10 36	5623680	**25841** 95963	405.21	8.12
BN_43	850	21	50	2	881	73 647	22731586	**148184** 629027	2132.53	46.37
BN_45	850	21	56	2	875	17 142	15778481	**122763** 260917	646.25	34.44

Table 1. Compilation results on UAI 2006 benchmarks (ISCAS circuits). Note that many instances are not shown here, which BE-AOMDD+R fails to compile due to memory limitations.

UAI 2006 ISCAS networks. In table 1, the CM OR column and Metanodes column show the differences between the size of the context minimal AO graph and the AOMDD. Within the Metanode column of a particular instance, the sizes of the AOMDDs generated by BE-AOMDD+R and AOMDD-BCP are contrasted.

We can see that there are significant savings yielded from applying the redundancy rule. However, there are many instances of the UAI 2006 set not explicitly shown in the table for which BE-AOMDD fails to compile without running out of memory which AOMDD-BCP is able to. One possible reason for this is that AOMDD-BCP preprocesses the network for determinism and prunes out branches of the search graph before they are generated. Our implementation of BE-AOMDD performs no preprocessing on the input problem. This can be additionally attributed to the difference in a bottom-up vs. a top-down approach [11], where it is possible that for these particular problem instances and variable orderings, determinism is encountered earlier by exploring top-down. The actual AOMDD generated by AOMDD-BCP has some redundancy reduction due to pruning in the AO search graph. Given enough time and memory resources, we expect BE-AOMDD to always yield a smaller compiled AOMDD than AOMDD-BCP, due to stronger redundancy reduction which removes any nodes whose corresponding variables' assignments do not affect the value of any extension of the assignment by descendant variables in the pseudo tree.

name	n	w	h	k	# functions	time (s)	CM OR	Metanodes [BE-AOMDD+R]	Max Memory Usage (MB)	Compiled AOMDD memory (MB)
pdb1fna	75	6	18	81	218	136	1983522	56377	467.61	44.44
pdb1j8e	39	6	12	81	119	294	2714323	258198	950.33	238.32
pdb1pef	17	6	11	81	55	430	4123288	342367	4499.79	772.83
pdb1rb9	42	7	14	81	128	1127	13370233	1163424	3789.48	1751.98
pdb2igd	50	6	19	81	146	1295	33711674	451081	3396.36	1132.93

Table 2. Compilation results on protein networks using BE-AOMDD+R.

Protein networks. Table 2 shows the results. We look at the same quantities as with the previous set of instances, but do not compare between the two compilation algorithms.

We can observe that these problems have low treewidth, but high domain sizes. The AOMDDs are able to capture the constantness of a function with respect to different assignments to the function input domain. Although we did not run AOMDD-BCP on these instances, but we speculate that due the large number of values, those problems would run out of time.

4.2 Inference

The following evaluates the AOMDD-BE algorithm, which is the same as bucket elimination, but uses AOMDDs to represent all functions. We ran experiments on the UAI 2006 evaluation problems, mastermind instances, and genetic linkage analysis networks, available at `http://graphmod.ics.uci.edu`. In each table, we compare the time and memory usages of standard BE vs. AOMDD-BE. Times reported as "OOM" indicate that the algorithm exceeded our memory bound. Results on memory usage are based on the usage of the cache storing nodes of the AOMDDs. For instance where BE runs out of memory, we simulated the its execution by only passing information about scope sizes to compute the memory usage.

problem	n	w	h	k	time (s) [BE]	time (s) [AOMDD-BE]	Mem (MB) [BE]	Mem (MB) [AOMDD-BE]
BN_22	2425	5	575	91	1	13	**26.93**	581.27
BN_24	1819	5	381	91	1	23	**24.07**	977.52
BN_28	24	5	9	10	1	13	**1.79**	568.36
BN_30	1156	48	179	2	OOM	38	1.50E+10	**245.93**
BN_32	1444	56	219	2	OOM	4384	4.45E+12	**3006.08**
BN_34	1444	55	220	2	OOM	145	2.30E+12	**515.45**
BN_36	1444	56	210	2	OOM	7792	3.51E+12	**2629.44**
BN_40	1444	55	235	2	OOM	91	1.82E+12	**322.76**
BN_42	880	23	54	2	21	2	314.04	**21.62**
BN_43	880	22	53	2	10	2	153.66	**11.83**
BN_46	499	22	49	2	18	<1	248.97	**1.99**
BN_49	661	44	59	2	OOM	1188	7.83E+08	**2991.78**
BN_51	661	44	61	2	OOM	3433	1.17E+09	**2274.11**
BN_53	561	48	95	2	OOM	4063	8.43E+09	**3303.48**
BN_61	667	44	61	2	OOM	17	9.46E+08	**235.72**
BN_65	440	61	95	2	OOM	1062	Overflow*	**2843.65**
BN_67	440	61	99	2	OOM	9893	Overflow*	**1270.54**
BN_78	54	13	24	2	<1	<1	**0.51**	29.82
BN_84	360	20	24	2	4	22	**24.76**	546.21
BN_86	422	22	40	2	26	73	**179.44**	1084.59
BN_92	422	22	33	2	26	23	**187.43**	433.65

Table 3. UAI 2006 benchmarks.(* The size in MB could not be stored within a double precision number representation.)

UAI 2006 and Mastermind benchmarks. Results are presented in Tables 3 and 4. In columns 5 and 6, we see the runtimes for BE and AOMDD-BE, while the last two columns show the the memory usages of BE and AOMDD-BE.

We see that our scheme is able to solve some problems which do not fit in standard main memory. These problems have structures that AOMDDs exploit well. Namely, the functions of these problems have many zero values that can be represented easily by AOMDDs. In addition, AOMDDs are able to take advantage of functions that have many values that are the same, but not necessarily zero. Such functions are present in a number of the instances on which it outperforms BE based on memory usage. However, there must be a significant amount of compression before we get any memory savings. Namely, as each node contains information to capture the structure of the problem, it means that much more memory is used when representing a function which has many different values.

name	n	w	h	k	time (s) [BE]	time (s) [AOMDD-BE]	Mem (MB [BE]	Mem (MB) [AOMDD-BE]
03_08_03-0000	1220	18	53	2	4	5	**48.23**	154.45
03_08_03-0001	1220	18	54	2	4	4	**53.63**	105.21
03_08_03-0006	1220	18	41	2	2	2	44.38	**41.70**
03_08_03-0007	1220	18	52	2	2	1	46.40	**21.64**
03_08_04-0000	2288	29	79	2	OOM	643	49865.56	**4187.84**
03_08_04-0001	2288	28	76	2	OOM	293	39769.34	**2610.66**
03_08_05-0006	3692	37	101	2	OOM	6	24847465.64	**39.04**
03_08_05-0007	3692	37	80	2	OOM	9	25456599.73	**120.19**
04_08_03-0001	1418	22	58	2	46	54	**638.49**	907.49
04_08_03-0002	1418	22	55	2	41	33	**621.16**	786.86
04_08_03-0006	1418	22	59	2	46	11	621.98	**270.47**
04_08_03-0007	1418	22	52	2	36	2	617.11	**47.70**
04_08_04-0006	2616	35	88	2	OOM	17	4675522.59	**371.95**
04_08_04-0007	2616	35	93	2	OOM	17	3467438.50	**349.37**
05_08_03-0000	1616	26	57	2	690	1064	9092.90	**5742.06**
05_08_03-0001	1616	26	67	2	758	759	8853.24	**3919.56**
06_08_03-0006	1814	29	73	2	OOM	135	92849.60	**1150.94**
06_08_03-0007	1814	29	66	2	OOM	27	76976.00	**588.10**
06_08_03-0008	1814	29	64	2	OOM	24	85068.34	**523.36**
06_08_03-0010	1814	29	66	2	OOM	13	93097.25	**253.99**
10_08_03-0006	2606	43	92	2	OOM	654	2085939395.05	**673.54**

Table 4. Mastermind instances.

We generally see that for lower treewidth networks, standard BE is sufficient and has better runtime, however, it is unable to solve problems with higher treewidth simply due to lack of memory.

name	n	w	h	k	time (s) [BE]	time (s) [AOMDD-BE]	Mem (MB) [BE]	Mem (MB) [AOMDD-BE]
pedigree1	334	15	61	4	2	14	**23.61**	210.09
pedigree9	1118	25	137	7	550	5301	7499.77	**4030.34**
pedigree18	1184	19	102	5	7	200	**136.13**	959.28
pedigree20	437	21	58	5	131	291	1393.90	**1030.66**
pedigree23	402	20	58	5	19	52	**241.57**	532.46
pedigree25	1289	23	86	5	146	1284	**2037.69**	2999.84
pedigree30	1289	20	102	5	13	307	**220.63**	1044.76
pedigree33	798	24	116	4	347	883	4277.26	**1368.42**
pedigree37	1032	20	62	5	OOM	3535	251109.68	**7992.43**
pedigree38	724	16	67	5	OOM	2201	172249.65	**6253.16**
pedigree39	1272	20	83	5	46	400	**772.20**	1555.68
pedigree44	811	24	79	4	516	3795	6153.63	**4782.29**

Table 5. Pedigree networks. Instances not shown here (7,13,19,31,34,40,41,42,50,51) run out of memory with both algorithms.

Pedigree networks. We also ran experiments on genetic linkage analysis networks (known as pedigree), for which the partition function value of many of them were not known before the work in [12], which makes use of hard disk to push the memory restrictions of solving a problem.

The results are shown in table 5. As with the previous set or problem instances, timing results are shown in columns 5 and 6 while memory usage is shown in columns 7 and 8.

Our results are less promising on these networks. There are only two instances which AOMDD-BE manages to perform very well on, which standard BE would require about 30 times the amount of memory. For the rest that AOMDD-BE managed to solve, a large number of problems were solvable by standard BE with a shorter amount of time and less memory. Even with those where AOMDD-BE uses less memory, the runtime is often much worse, due to overhead in maintaining the properties of a canonical AOMDD. We can attribute these results this set of problems having overall less determinism and context-specific independence. However, these results also demonstrate the use of decision diagrams on non-binary networks for inference, when compared to related work using ADDs [4, 13].

5 Conclusion

For many hard problems (such as the pedigree networks), the overhead of using the minimal AOMDD structure for function representation actually results in worse performance in terms of both time and space. On other problems, such as the ISCAS networks in the UAI 2006 evaluation set and the mastermind instances, our scheme shows good performance despite having high treewidth. We demonstrated results reinforcing the potential of using AOMDDs for the classic BE algorithm. Future work would include comparing with related techniques for exploiting determinism and context-specific independence such as ACE [14].

Other related work includes using ADDs for approximate inference [13], where the size of the diagram in bucket elimination is bounded by merging nodes until the diagram is with in a specified bound, subject to minimizing a metric, such as KL divergence between the function represented by the compacted diagram and the original function. It would be interesting to apply similar schemes to AOMDDs.

References

1. Mateescu, R., Dechter, R., Marinescu, R.: AND/OR multi-valued decision diagrams (AOMDDs) for graphical models. Journal of Artificial Intelligence Research **33**(1) (2008) 465–519
2. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8) (1986)
3. Mateescu, R., Dechter, R.: Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In: Principles and Practice of Constraint Programming (CP 2006). (2006) 10.1007/11889205_25.
4. Chavira, M., Darwiche, A.: Compiling bayesian networks using variable elimination. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07). (2007) 2443–2449

5. Sanner, S., McAllester, D.: Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). Volume 19. (2005) 1384
6. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113**(1) (1999) 41–85
7. Kam, T., Villa, T., Brayton, R., Sangiovanni-Vincentelli, A.: Multi-valued decision diagrams: theory and applications. Multiple-Valued Logic **4**(1) (1998) 9–62
8. Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F.: Algebric decision diagrams and their applications. Formal methods in system design **10**(2) (1997) 171–206
9. Boutilier, C., Friedman, N., Goldszmidt, M., Koller, D.: Context-specific independence in bayesian networks. In: Proc. of the 12th International Conference on Uncertainty in Artificial Intelligence (UAI-96). (1996) 115–123
10. Dechter, R.: Constraint processing. Morgan Kaufmann (2003)
11. Mateescu, R., Dechter, R.: The relationship between and/or search and variable elimination. In: Proceedings of the Twenty First Conference on Uncertainty in Artificial Intelligence (UAI05). (2005) 380–387
12. Kask, K., Dechter, R., Gelfand, A.: BEEM: Bucket elimination with external memory. In: Proc. of the 26th Annual Conference on Uncertainty in Artificial Intelligence (UAI-10). (2010) 268–276
13. Gogate, V., Domingos, P.: Approximation by quantization. In: Proc. of the 27th Annual Conference on Uncertainty in Artificial Intelligence (UAI-11). (2011) 247–255
14. Chavira, M., Darwiche, A.: Compiling bayesian networks with local structure. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). Volume 19. (2005) 1306