Compiling Bayesian Networks Using Variable Elimination *

Mark Chavira and **Adnan Darwiche**
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095-1596
{chavira,darwiche}@cs.ucla.edu

Abstract

Compiling Bayesian networks has proven an effective approach for inference that can utilize both global and local network structure. In this paper, we define a new method of compiling based on variable elimination (VE) and Algebraic Decision Diagrams (ADDs). The approach is important for the following reasons. First, it exploits local structure much more effectively than previous techniques based on VE. Second, the approach allows *any* of the many VE variants to compute answers to multiple queries simultaneously. Third, the approach makes a large body of research into more structured representations of factors relevant in many more circumstances than it has been previously. Finally, experimental results demonstrate that VE can exploit local structure as effectively as state–of–the–art algorithms based on conditioning on the networks considered, and can sometimes lead to much faster compilation times.

1 Introduction

Variable elimination [Zhang and Poole, 1996; Dechter, 1996] (VE) is a well–known algorithm for answering probabilistic queries with respect to a Bayesian network. The algorithm runs in time and space exponential in the treewidth of the network. Advantages of VE include its generality and simplicity. In this paper, we consider two aspects of VE. The first is how to effectively utilize local structure in the form of determinism [Jensen and Andersen, 1990] and context–specific independence [Boutilier *et al.*, 1996] to perform inference more efficiently. The second consideration is how to answer multiple queries simultaneously. For example, given particular evidence, we would like to be able to simultaneously compute probability of evidence and a posterior marginal on each network variable. And we would like to be able to repeat this process for many different evidence sets.

Many proposals have been extended to utilize local structure in the context of VE. Although the standard version of the algorithm uses tables to represent factors, the algorithm can sometimes run more efficiently by using a more structured representation like ADDs [R.I. Bahar *et al.*, 1993], affine ADDs [Sanner and McAllester, 2005], representations composed of confactors [Poole and Zhang, 2003], sparse representations [Larkin and Dechter, 2003], and a collection of others. Such a representation makes use of local structure to skip certain arithmetic operations and to represent factors more compactly. However, the effectiveness of these approaches has been sometimes limited in practice, because the overhead they incur may very well outweigh any gains.

There has been less work to answer multiple queries simultaneously using VE as is done by the jointree algorithm [Jensen *et al.*, 1990; Lauritzen and Spiegelhalter, 1988]; but see [Cozman, 2000] for an exception. Theoretical results in [Darwiche, 2000] demonstrate that by keeping a *trace*, one can use VE as a basis for compilation. This algorithm, which we will refer to as *tabular compilation*, will compile a Bayesian network offline into an arithmetic circuit (AC). Tabular compilation runs in time and space exponential in treewidth; the resulting AC has size exponential in treewidth; and once offline compilation is complete, the AC can be used online to answer many queries simultaneously in time linear in its size. However, the applicability of this approach is limited, because tabular compilation provides no practical advantage over jointree.

An important observation is that when compiling using VE, we need not use tables, but can instead use one of the more structured representations of factors. This observation is seemingly innocent, but it has two critical implications. First, the usual cost incurred when using structured representations gets pushed into the offline phase, where it can be afforded. Second, the usual advantage realized when using more structured representations means that the size of the resulting AC will not necessarily be exponential in treewidth. The smaller AC translates directly into more efficient online inference, where efficiency matters most. We see that the proposed approach is important for four key reasons. First, it exploits local structure much more effectively than previous techniques based on VE. Second, the approach allows *any* of the many VE variants to compute answers to multiple queries simultaneously. Third, the approach makes a large body of research into more structured representations of factors relevant in many more circumstances than it has been previously. Finally, experimental results demonstrate that on the consid-

*This work has been partially supported by Air Force grant #FA9550-05-1-0075-P00002 and JPL/NASA grant #1272258.

ered networks, VE can exploit local structure as effectively as state–of–the–art algorithms based on conditioning as applied to a logical encoding of the networks, and can sometimes lead to much faster compilation times.

To demonstrate these ideas, we propose a new method of compiling based on VE and algebraic decision diagrams (ADDs). We will refer to this method of compiling as *ADD compilation*. It is well known that ADDs can represent the initial conditional probability distributions (factors) of a Bayesian network more compactly than tables. However, it is not clear whether ADDs will retain this advantage when producing intermediate factors during the elimination process, especially in the process of compilation which involves no evidence. Note that the more evidence we have, the smaller the ADDs for intermediate factors.

Experimental results will demonstrate several important points with respect to ADD compilation. The first point deals with AC sizes. ADD *elimination* (no compilation) can outperform tabular *elimination* (no compilation) when there are massive amounts of local structure, but can dramatically underperform tabular elimination otherwise. However, ADD *compilation* produces ACs that are much smaller than those produced by tabular *compilation*, even in many cases where there are lesser amounts of local structure. Second, because of the smaller AC size, online inference time is capable of outperforming jointree by orders of magnitude. Finally, ADD compilation can be much faster in some cases than a state–of–the–art compilation technique, which reduces the problem to logical inference [Darwiche, 2002].

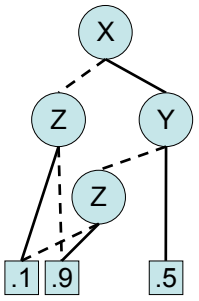
This paper is organized as follows. We begin in Section 2 by reviewing VE, elimination using ADDs, and compilation into ACs. Next, Section 3 shows how a trace can be kept during elimination employing ADDs to compile a network into an AC. We then discuss in Section 4 implications for compiling using structured representations of factors. We present experimental results in Section 5 and conclude in Section 6.

2 Background

In this section, we briefly review VE, elimination using ADDs, and compilation into ACs.

2.1 Variable Elimination

Variable elimination is a standard algorithm for computing probability of evidence with respect to a given a Bayesian network [Zhang and Poole, 1996; Dechter, 1996]. For space reasons, we mention only a few points with regard to this algorithm. First, the algorithm acts on a set of *factors*. Each factor involves a set of variables and maps instantiations of those variables to real–numbers. The initial set of factors are the network's conditional probability distributions (usually tables). Elimination is driven by an ordering on the variables called an *elimination order*. During the algorithm, two *factor operations* are performed many times: factors are *multiplied* and a variable is *summed out* of a factor. These factor operations reduce to performing many multiplication and addition operations on real–numbers. Although tables are the most common representation for factors, any representation that supports multiplication and summing–out can be used.



X	Y	Z	$f(.)$
x_1	y_1	z_1	0.9
x_1	y_1	z_2	0.1
x_1	y_2	z_1	0.9
x_1	y_2	z_2	0.1
x_2	y_1	z_1	0.1
x_2	y_1	z_2	0.9
x_2	y_2	z_1	0.5
x_2	y_2	z_2	0.5

Figure 1: An ADD over variables X, Y, Z and the corresponding table it represents. Dotted edges point to low–children and solid edges point to high–children.

2.2 Elimination using ADDs

An ADD is a graph representation of a function that maps instantiations of Boolean variables to real–numbers [R.I. Bahar *et al.*, 1993]. Figure 1 depicts an ADD and the corresponding table (function) it represents. In the worst case, an ADD has the same space complexity as a table representing the same function. Moreover, the factor operations of multiplication and summing–out are implemented on ADDs in polynomial time. There are four points that are important with respect to ADDs. First, because multiplication and summing–out are available for ADDs, we can use ADDs during VE instead of tables. Second, because ADDs can leverage local structure, an ADD can be much smaller than the corresponding table. Third, for the same reason, when applied to ADDs, the factor operations of multiplication and summing–out can result in far fewer arithmetic operations on numbers than the same factor operations acting on tables. Finally, the constants involved in using ADDs are much larger than those involved in using tables, so much so that elimination using ADDs will often take much longer than elimination using tables, even when the ADD performs fewer arithmetic operations on numbers, and so much so that we may run out of memory when using ADDs, even in cases where tabular elimination does not. We will discuss later how we can deal with many–valued variables within an ADD.

2.3 Compiling into ACs

The notion of using arithmetic circuits (ACs) to perform probabilistic inference was introduced in [Darwiche, 2000; 2003]. With each Bayesian network, we associate a corresponding multi-linear function (MLF) that computes the probability of evidence. For example, the network in Figure 2—in which variables A and B are Boolean, and C has three values—induces the following MLF:

$$\lambda_{a_1}\lambda_{b_1}\lambda_{c_1}\theta_{a_1}\theta_{b_1}\theta_{c_1|a_1,b_1} + \lambda_{a_1}\lambda_{b_1}\lambda_{c_2}\theta_{a_1}\theta_{b_1}\theta_{c_2|a_1,b_1} + \ldots$$
$$\lambda_{a_2}\lambda_{b_2}\lambda_{c_2}\theta_{a_2}\theta_{b_2}\theta_{c_2|a_2,b_2} + \lambda_{a_2}\lambda_{b_2}\lambda_{c_3}\theta_{a_2}\theta_{b_2}\theta_{c_3|a_2,b_2}$$

The terms in the MLF are in one-to-one correspondence with the rows of the network's joint distribution. Assume that all *indicator variables* λ_x have value 1 and all *parameter variables* $\theta_{x|\mathbf{u}}$ have value $\Pr(x|\mathbf{u})$. Each term will then be a

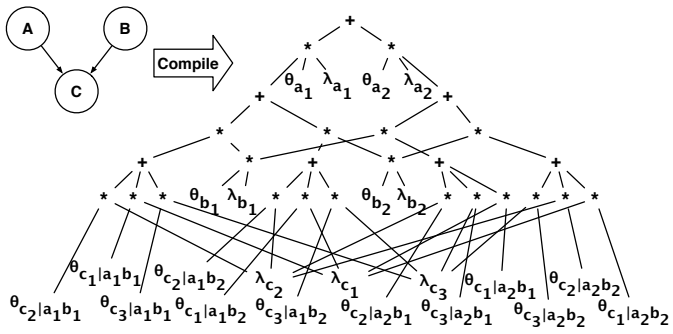


Figure 2: A Bayesian network and a corresponding AC.

product of probabilities which evaluates to the probability of the corresponding row from the joint. The MLF will add all probabilities from the joint, for a sum of 1.0. To compute the probability $\Pr(\mathbf{e})$ of evidence $\mathbf{e}$, we need a way to exclude certain terms from the sum. This removal of terms is accomplished by carefully setting certain indicators to 0 instead of 1, according to the evidence.

The fact that a network's MLF computes the probability of evidence is interesting, but the network MLF has exponential size. However, if we can factor the MLF into something small enough to fit within memory, then we can compute $\Pr(\mathbf{e})$ in time that is linear in the size of the factorization. The factorization will take the form of an AC, which is a rooted DAG (directed acyclic graph), where an internal node represents the sum or product of its children, and a leaf represents a constant or variable. In this context, those variables will be indicator and parameter variables. An example AC is depicted in Figure 2. We refer to this process of producing an AC from a network as *compiling* the network.

Once we have an AC for a network, we can compute $\Pr(\mathbf{e})$ by assigning appropriate values to leaves and then computing a value for each internal node in bottom-up fashion. The value for the root is then the answer to the query. We can also compute a posterior marginal for each variable in the network by performing a second downward pass [Darwiche, 2003]. Hence, many queries can be computed simultaneously in time linear in the size of the AC. Another main point is that this process may then be repeated for as many evidence sets as desired, without recompiling.

3 ADD Compilation

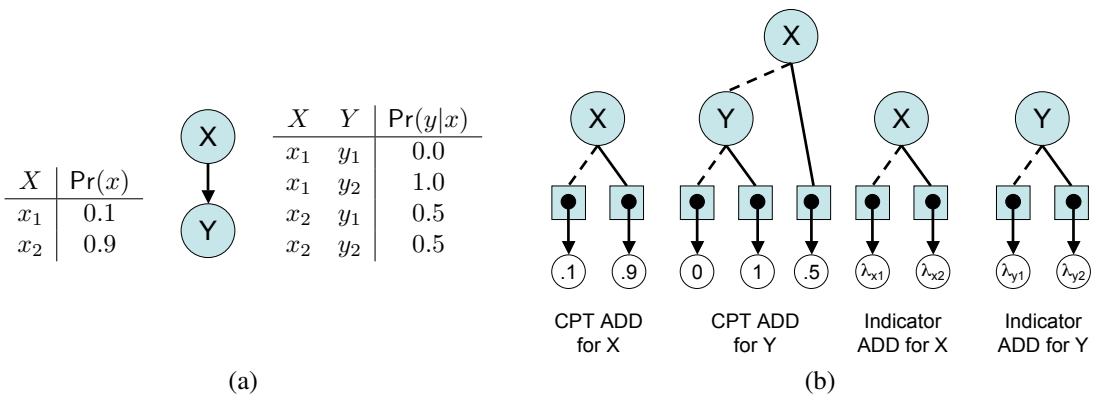
We have seen how ADDs can be used in place of tables during VE and how compilation produces an AC from a network. In this section, we combine the two methods to compile an AC using a VE algorithm that employs ADDs to represent factors.

The core technique behind our approach is to work with ADDs whose sinks point to ACs (their roots) instead of pointing to constants. We will refer to these ADDs as *symbolic ADDs*. Given a Bayesian network, we convert each conditional probability table (CPT) into a symbolic ADD called a *CPT ADD*. To do so, we first convert the CPT into a normal ADD and then replace each constant n within a sink by a pointer to an AC consisting of a single node labeled with constant n. For each variable X in the network, we also construct an *indicator ADD*, which acts as a placeholder for evidence to be entered during the online inference phase. Assuming that variable X is binary with values x_1 and x_2, the indicator ADD for X will consist of one internal node, labeled with variable X, and having two sink children. The child corresponding to $x_1(x_2)$ will be a sink labeled with a pointer to a single–node AC that is itself labeled with indicator $\lambda_{x_1}(\lambda_{x_2})$. Figure 3(a) shows a simple Bayesian network and Figure 3(b) shows the corresponding symbolic ADDs.

Algorithm 1 $Multiply(\alpha_1 : SymADD, \alpha_2 : SymADD)$: returns $SymADD$.

1: swap α_1 and α_2 **if** $Pos(\alpha_1) > Pos(\alpha_2)$
2: **if** $cache(\alpha_1, \alpha_2) \neq null$ **then**
3: return $cache(\alpha_1, \alpha_2)$
4: **else if** α_1 and α_2 are leaf nodes **then**
5: $\alpha \leftarrow$ new ADD leaf node
6: $Ac(\alpha) \leftarrow$ new AC $*$ node with children $Ac(\alpha_1)$ and $Ac(\alpha_2)$
7: **else if** $Pos(\alpha_1) = Pos(\alpha_2)$ **then**
8: $\alpha \leftarrow$ new internal node
9: $Var(\alpha) \leftarrow Var(\alpha_1)$
10: $Lo(\alpha) \leftarrow Multiply(Lo(\alpha_1), Lo(\alpha_2))$
11: $Hi(\alpha) \leftarrow Multiply(Hi(\alpha_1), Hi(\alpha_2))$
12: **else**
13: $\alpha \leftarrow$ new internal node
14: $Var(\alpha) \leftarrow Var(\alpha_1)$
15: $Lo(\alpha) \leftarrow Multiply(Lo(\alpha_1), \alpha_2)$
16: $Hi(\alpha) \leftarrow Multiply(Hi(\alpha_1), \alpha_2)$
17: **end if**
18: $cache(\alpha_1, \alpha_2) \leftarrow \alpha$
19: return α

Working with symbolic ADDs requires a modification to the standard ADD operations, but in a minimal way. We first point out that ADD operations are typically implemented recursively, reducing an operation over two ADDs into operations over smaller ADDs, until we reach boundary conditions: ADDs that correspond to sinks. It is these boundary conditions that need to be modified in order to produce symbolic ADD operations. For example, Algorithm 1 specifies how to multiply two symbolic ADDs. When applied to an internal ADD node, the functions Pos, Lo, and Hi return the position of the variable labeling the node in the ADD variable order, the low–child (corresponding to $false$), and the high–child (corresponding to $true$), respectively. When applied to an ADD sink, the functions Pos and Ac return ∞ and a pointer to the AC that labels the sink, respectively. Finally, $cache$ is the standard computed table used in ADD operations. The main observation is that the algorithm is identical to that for multiplying normal ADDs (see [R.I. Bahar *et al.*, 1993]), except for Line 6. When multiplying normal ADDs, this line would label the newly created sink with a constant that is the product of the constants labeling α_1 and α_2. When multiplying symbolic ADDs, we instead label the new sink with a pointer to an AC that represents an analogous multiplication. This AC will consist of a new multiply node having children that are the ACs pointed to by α_1 and α_2. The algorithm for summing–out a variable from a symbolic ADD is constructed from the algorithm for summing–out a variable from a normal



X	Pr(x)
x_1	0.1
x_2	0.9

X	Y	Pr($y\|x$)
x_1	y_1	0.0
x_1	y_2	1.0
x_2	y_1	0.5
x_2	y_2	0.5

Figure 3: (a) A simple Bayesian network and (b) symbolic ADDs representing the network.

Algorithm 2 $Compile(N$: Bayesian Network): returns AC.

1: $\Psi \leftarrow$ the set of indicator and CPT ADDs for N
2: $\pi \leftarrow$ ordering on the variables ($\pi(i)$ is ith variable in order)
3: **for** i in $1 \ldots$ number of variables in N **do**
4: $\quad P \leftarrow \{\alpha \in \Psi : \alpha$ mentions variable $\pi(i)\}$
5: $\quad \Psi \leftarrow (\Psi - P) \cup \{\Sigma_{\pi(i)} \prod_{\alpha \in P} \alpha\}$
6: **end for**
7: $\beta \leftarrow \prod_{\alpha \in \Psi} \alpha$
8: return $Ac(\beta)$

ADD in an analogous way, by modifying the code that labels newly created sinks. In this case, instead of labeling with a constant representing a sum, we instead label with a pointer to an new addition node.

We now describe the compilation process in Algorithm 2. Line 1 begins by representing the Bayesian network using symbolic ADDs as described previously. We then generate an ordering on the variables using a minfill heuristic on Line 2. Lines 3–7 perform variable elimination in the standard way, using the multiply and sum–out operations of symbolic ADDs. Afterward, we are left with a trivial symbolic ADD: a sink labeled with a pointer to an AC. This AC is a factorization of the network MLF and a compilation of the given network! Each internal node of the AC is labeled with a multiplication or addition operation, and each leaf is labeled with a constant or an indicator variable. The AC represents a history or *trace* of all arithmetic operations performed during the elimination process.

For the above procedure to scale and produce the results we report later, it has to be augmented by a number of key techniques that we describe next.

ADD variable order: ADDs require a fixed variable order which is independent of the elimination order we have discussed. For that, we use the reverse order used in the elimination process. This ensures that when a variable is eliminated from an ADD, it appears at the bottom of the ADD. The sum–out operation on ADDs is known to be much more efficient when the summed out variable appears at the bottom.

Unique table: We maintain a cache, called a unique table in the ADD literature, to ensure that we do not generate redundant AC nodes. Each entry is an AC node that has been constructed, indexed the node's label and its children. Before we ever construct a new AC node, we first consult the unique table to see if a similar node has been constructed before. If this is the case, we simply point to the existing node instead of constructing a duplicate node. As is standard practice, we also use a unique table for the ADD nodes.

Multi–valued variables: When a network variable X has more than two values, we define its ADD variables as follows. For each value x_i of X, we create ADD variable V_{x_i}. We translate instantiations of X in the straightforward way. For example, we translate $X = x_2$ as $V_{x_2} =$ true and $V_{x_i} =$ false for $i \neq 2$. Moreover, we construct an ADD over V_{x_i} which enforces the constraint that exactly one V_{x_i} can be true. We multiply this ADD into every CPT and indicator ADD that mentions X. Finally, when eliminating (summing–out) a multi–valued variable X, we sum out all corresponding ADD variables V_{x_i} simultaneously.

Converting a CPT into an ADD: The straightforward way to construct the ADD of a CPT is to construct an ADD for each row of the CPT and then add the resulting ADDs. This method was extremely inefficient for large CPTs (over a thousand parameters). Instead, we construct a tree–structured ADD whose terminals correspond to the CPT rows, and then use the standard reduce operation of ADDs to produce a DAG structure.

Simplifications: In certain cases, simplifications are possible. For example, when multiplying two ADD nodes, if one of the nodes α is a sink labeled with a pointer to an AC node which which is itself labeled constant 0, then we can simply return α without doing more work. Similar simplifications exist for constant 1 and for the summing–out operation.

Figure 4 depicts the elimination process when applied to the symbolic ADDs of Figure 3(b) using elimination order Y, X. The first step is to multiply the set of symbolic ADDs involving Y, which consists of Y's CPT ADD and Y's indicator ADD. Figure 4(a) shows the result. Observe that each ADD sink points to an AC that represents the multiplication of two ACs, one pointed to by a sink from Y's CPT ADD and one pointed to by a sink from Y's indicator ADD. Also observe that when one of the ACs involved in the multiplication

is a sink labeled with 0 or 1, the resulting AC is simplified. Figure 4(b) next shows the result of summing–out Y from the symbolic ADD in Figure 4(a). Here, each ADD node labeled with Y and its children have been replaced with a sink which points to an addition node. A simplification has also occurred. Figure 4(c) shows the result of multiplying symbolic ADDs involving variable X in two steps: first Figure 4(b) by X's indicator ADD and then the result by X's CPT ADD. Finally, Figure 4(d) shows the result of summing–out X from Figure 4(c). At this point, since there is only one remaining symbolic ADD, we have our compilation.

4 Implications

There has been much research into using alternatives to tabular representations of factors in VE. The motivation for such research is that tabular elimination makes use only of global network structure (its topology). It may therefore miss opportunities afforded by local structure to make inference more efficient. Structured representations of factors typically exploit local structure to lessen the space requirements required for inference and to skip arithmetic operations on numbers (multiplications and additions) required for inference. In some cases, structured representations of factors have been successful in performing inference when tabular elimination runs out of memory, and in some cases, structured representations allow for faster inference than tabular elimination. However, the time and space overhead involved in using more structured representations of factors may very well outweigh the benefits. This is particularly true when there are not excessive amounts of local structure present in the network. Table 1 demonstrates the performance of elimination (no compilation) using both tables and ADDs applied to the set of networks we will be considering later in the paper. All ADD operations were performed using the publicly available CUDD ADD package (http://vlsi.colorado.edu/∼fabio/CUDD). We see that in only two cases does ADD elimination outperform tabular elimination. Both of the networks involved, bm-5-3 and mm-3-8-3, contain massive amounts of determinism. In the remaining cases, ADD elimination is actually slower than tabular elimination, often by an order of magnitude or more. The large number of examples where structured representations of factors are slower than tabular elimination have limited their applicability.

The main observation behind this paper is that even when elimination using a more structured representation of factors is slower than tabular elimination, structured representations become very powerful in the context of compilation. Compilation is an offline process, and so extra time can be afforded. In fact, the combination of structured representations with compilation is ideal, since the very purpose of compilation is to spend time offline in an effort to identify opportunities for savings that can be used repeatedly during online inference. Hence, in the context of compilation, the disadvantages of using structured representations are much less severe (virtually disappearing) and the advantages are retained.[1] Although

[1]Note that the use of more structured representations of factors is usually asymptotically no worse than the use of tabular representations, which is particularly true for ADDs.

Network	Tabular Time (ms)	ADD Time (ms)	Improvement
alarm	31	360	0.086
barley	307	14,049	0.022
bm-5-3	4,892	658	7.435
diabetes	949	33,220	0.029
hailfinder	48	515	0.093
link	1,688	2,658	0.635
mm-3-8-3	2,166	843	2.569
mildew	72	92,602	0.001
munin1	155	1,255	0.124
munin2	204	3,170	0.064
munin3	350	5,049	0.069
munin4	406	4,361	0.093
pathfinder	51	5,213	0.010
pigs	69	597	0.116
st-3-2	186	362	0.514
tcc4f	29	153	0.190
water	76	1,015	0.075

Table 1: Time to perform VE (no compilation) using tables and using ADDs.

overhead is incurred during the offline phase, each arithmetic operation saved makes the size of the resulting AC smaller, yielding more efficient online inference. These observations make research into structured representations of factors applicable in many more circumstances than it has been previously. Another major advantage of adding compilation to elimination is that one can compute online answers to many queries simultaneously and very quickly as is done by the jointree algorithm; see [Darwiche, 2003] for details.

5 Experimental Results

In this section, we present experimental results of applying ADD compilation and compare ADD compilation to tabular compilation, Ace compilation, and jointree. All ADD operations were performed using the CUDD ADD package (http://vlsi.colorado.edu/∼fabio/CUDD), modified to work with symbolic ADDs. Ace (http://reasoning.cs.ucla.edu/ace) is a state–of–the–art compiler that encodes Bayesian networks as logical knowledge bases and then applies conditioning to the result. Ace has been shown to be extremely effective on networks having large amounts of local structure [Chavira *et al.*, May 2006; 2005], able to compile many networks with treewidths in excess of one–hundred. Ace was also shown in [Chavira and Darwiche, 2005] to perform online inference orders of magnitude more efficiently than jointree when applied to networks having treewidth small enough for jointree to work and having lesser amounts of local structure. On these networks, Ace online inference is very efficient, but some compile times are in excess of one thousand seconds. The networks with which we experimented are a superset of these networks from [Chavira and Darwiche, 2005].

Experiments ran on a 2GHz Core Duo processor with 2GB of memory, using the networks in Table 2.[2] To make the experiments as fair as possible, for each network, we first

[2]Because Ace was not available for the test platform, Ace compilation was carried out on a 1.6GHz Pentium M with 2GB of memory.

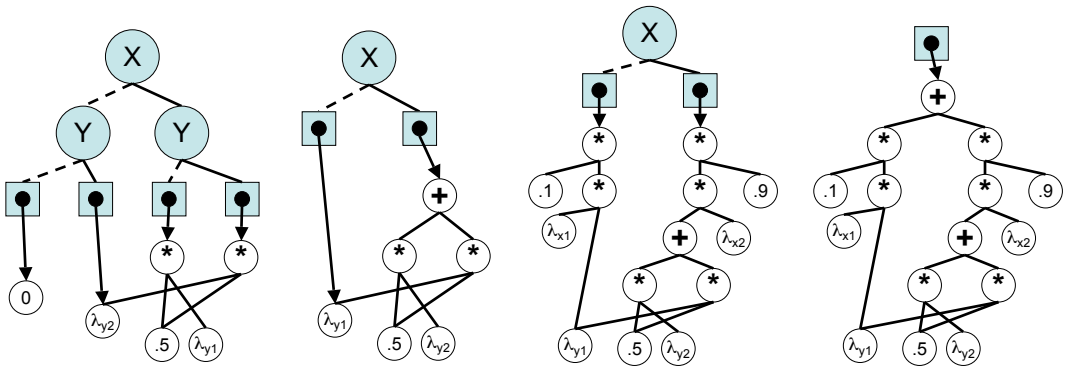


Figure 4: Symbolic ADDs produced after (a) multiplying factors involving Y, (b) summing–out Y, (c) multiplying factors involving X, and (d) summing–out X.

generated an elimination order π, and all experiments were performed with respect to π. The second column in Table 2 shows the maximum cluster size of a jointree generated from π, which gives an indication of difficulty with respect to structure–based methods. *We compiled each network using jointree, Ace 1.2, ADD VE, and tabular VE, in each case driven by* π. We evaluate a compilation algorithm based on compilation time, compilation size, and online inference time. We first discuss compilation time, which includes reading the network, compiling, and writing the AC to disk.

Columns 3–5 compare Ace compilation time to ADD compilation time. We see that on these networks, neither algorithm dominates the other. Our experiments show that Ace compilation often ran faster than ADD compilation in the following cases: when the networks had very low treewdith (alarm, hailfinder, tcc4f) and when the networks had massive amounts of determinism (bm-5-3, mm-3-8-3, and st-3-2). We also found Ace compilation times to be significantly lower on networks (not shown) having very high treewidth (e.g., the blockmap and mastermind networks from [Chavira *et al.*, May 2006]), many of which ADD compilation could not handle. *However, ADD compilation times are significantly more efficient on precisely those networks on which Ace has trouble.* In particular, networks with treewidths in the range 14-30 having lesser amounts of local structure (barley, diabetes, link, mildew, and the munin networks). On these types of networks, Ace compilation can require thousands of seconds and ADD compilation can run an order of magnitude faster. Perhaps the most striking example is barley, on which Ace required over two hours while ADD compilation required a little over two minutes. Ace compilation also ran out of memory on link, which compiled successfully using ADDs.

We next discuss compilation sizes, which we measure using the number of edges of the AC produced. The AC size is important, because it determines the time and space requirements for online inference, which is linear in this size. The AC size also provides the main indicator for the extent to which local structure was exploited. AC sizes produced using ADDs and using Ace turned out to be very close, showing that both methods seem to exploit local structure to a similar extent. Columns 6–8 compare sizes of ACs that would be produced using tabular compilation to those produced using ADD compilation (the size of the AC embedded in a jointree [Park and Darwiche, 2004] would be about equal to the size of the AC produced by tabular compilation). *ADD compilation sizes can be multiple orders of magnitude smaller than those produced by tabular compilation.* The difference is most pronounced on networks having massive amounts of local structure (bm-5-3, mm-3-8-3, st-3-2), but is also striking in some other cases. For example, on munin1, munin4, pathfinder, and water, ADD compilation sizes were over 40, 11, 17, and 93 times smaller, respectively. We see here the massive advantage of utilizing more structured representations of factors when compiling via VE.

Because ACs produced using Ace were about the same size as those produced using ADDs, online inference times for the two compilation algorithms were also very similar. The last three columns in Table 2 compare online inference using ADD compilations and jointree (tabular compilation online times would be about equal to jointree online times). For each network, we generated 16 sets of evidence. For each evidence set, we then computed probability of evidence and a posterior marginal on each network variable. Note that answering these queries would be very difficult using standard VE (without compilation), because of the large number of queries involved. Each reported number is the sum of times for all 16 evidence sets. Here we see an online time advantage to ADD compilation about equal to its large space advantage over tabular compilation, which makes sense, since neither tabular compilation nor jointree make use of local structure. *Once again we see the massive advantage that can be achieved by utilizing structured representations of factors during compilation.*

6 Conclusion

In this paper, we combined three ideas: variable elimination, compilation, and structured representations of factors. The result is an algorithm that retains the advantages of these approaches while minimizing their shortcomings. The approach is important for the following reasons: (1) it utilizes local

Table 2: Comparing Ace compilation times to ADD compilation times, tabular compilation size to ADD compilation size, and jointree online inference time to ADD online inference time. Imp. stands for factor of improvement.

Network	Max	Offline Compile Time (s)			AC Edge Count			Online Inference Time (ms)		
	Clust.	Ace	ADD-VE	Imp.	Tabular-VE	ADD-VE	Imp.	Jointree	ADD-VE	Imp.
alarm	7.2	0.3	3.9	0.1	3,534	3,030	1.2	166	32	5.2
barley	22.8	8,190.2	122.8	66.7	66,467,777	24,653,744	2.7	65,226	35,209	1.9
bm-5-3	19.0	0.8	6.0	0.1	75,591,750	14,836	5095.2	89,593	83	1079.4
diabetes	17.2	1,710.0	110.3	15.5	34,728,957	17,219,042	2.0	29,316	20,421	1.4
hailfinder	11.7	0.7	1.2	0.5	72,755	25,992	2.8	245	70	3.5
link	21.0	-	699.7	-	127,262,777	89,097,450	1.4	223,542	175,769	1.3
mildew	21.4	3,125.2	218.9	14.3	16,094,592	3,352,330	4.8	10,077	4,522	2.2
mm-3-8-3	19.0	1.5	11.9	0.1	36,635,566	108,428	337.9	34,001	198	171.7
munin1	26.2	1,005.1	316.7	3.2	1,260,407,123	31,409,970	40.1	669,915	37,451	17.9
munin2	18.9	198.4	31.7	6.3	20,295,426	5,662,218	3.6	17,857	7,180	2.5
munin3	17.3	188.4	17.6	10.7	16,987,088	3,503,242	4.8	13,351	4,945	2.7
munin4	19.6	205.0	37.8	5.4	76,028,532	6,869,760	11.1	42,754	8,683	4.9
pathfinder	15.0	4.9	5.8	0.9	796,588	44,468	17.9	1,332	102	13.1
pigs	17.4	23.1	10.0	2.3	4,925,388	2,558,680	1.9	3,020	2,814	1.1
st-3-2	21.0	0.5	2.4	0.2	19,374,934	22,070	877.9	17,536	82	213.9
tcc4f	10.0	0.9	1.1	0.8	33,408	22,612	1.5	281	73	3.8
water	20.8	3.0	20.7	0.1	15,996,054	170,428	93.9	16,676	251	66.4

structure more effectively than previous approaches based on VE; (2) it allows any variant of VE to answer multiple queries simultaneously; (3) it makes a large body of research into structured representations of factors more relevant than it had been previously; and (4) it demonstrates that variable elimination can utilize local structure as effectively as state–of–the–art approaches based on conditioning (applied to a logical encoding of the Bayesian network), and can sometimes lead to much faster compilation times.

References

[Boutilier *et al.*, 1996] Craig Boutilier, Nir Friedman, Moisés Goldszmidt, and Daphne Koller. Context–specific independence in Bayesian networks. In *UAI*, pages 115–123, 1996.

[Chavira and Darwiche, 2005] Mark Chavira and Adnan Darwiche. Compiling Bayesian networks with local structure. In *IJCAI*, pages 1306–1312, 2005.

[Chavira *et al.*, 2005] Mark Chavira, David Allen, and Adnan Darwiche. Exploiting evidence in probabilistic inference. In *UAI*, pages 112–119, 2005.

[Chavira *et al.*, May 2006] Mark Chavira, Adnan Darwiche, and Manfred Jaeger. Compiling relational Bayesian networks for exact inference. *IJAR*, 42(1–2):4–20, May 2006.

[Cozman, 2000] Fabio Gagliardi Cozman. Generalizing Variable Elimination in Bayesian Networks. In *Workshop on Probabilistic Reasoning in Artificial Intelligence*, Atibaia, Brazil, November 2000.

[Darwiche, 2000] Adnan Darwiche. A differential approach to inference in Bayesian networks. In *UAI*, pages 123–132, 2000.

[Darwiche, 2002] Adnan Darwiche. A logical approach to factoring belief networks. In *Proceedings of KR*, pages 409–420, 2002.

[Darwiche, 2003] Adnan Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM*, 50(3):280–305, 2003.

[Dechter, 1996] Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *UAI*, pages 211–219, 1996.

[Jensen and Andersen, 1990] Frank Jensen and Stig K. Andersen. Approximations in Bayesian belief universes for knowledge based systems. In *UAI*, pages 162–169, Cambridge, MA, July 1990.

[Jensen *et al.*, 1990] F. V. Jensen, S.L. Lauritzen, and K.G. Olesen. Bayesian updating in recursive graphical models by local computation. *Computational Statistics Quarterly*, 4:269–282, 1990.

[Larkin and Dechter, 2003] David Larkin and Rina Dechter. Bayesian inference in the presence of determinism. In C. M. Bishop and B. J. Frey (eds), editors, *AI-STAT*, Key West, FL., 2003.

[Lauritzen and Spiegelhalter, 1988] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of Royal Statistics Society, Series B*, 50(2):157–224, 1988.

[Park and Darwiche, 2004] James Park and Adnan Darwiche. A differential semantics for jointree algorithms. *Artificial Intelligence*, 156:197–216, 2004.

[Poole and Zhang, 2003] D. Poole and N.L. Zhang. Exploiting contextual independence in probabilistic inference. *Journal of Artificial Intelligence*, 18:263–313, 2003.

[R.I. Bahar *et al.*, 1993] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, 1993. IEEE Computer Society Press.

[Sanner and McAllester, 2005] Scott Sanner and David A. McAllester. Affine algebraic decision diagrams (aadds) and their application to structured probabilistic inference. In *IJCAI*, pages 1384–1390, 2005.

[Zhang and Poole, 1996] Nevin Lianwen Zhang and David Poole. Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.