Using Recursive Decomposition to Construct Elimination Orders, Jointrees, and Dtrees

Adnan Darwiche and Mark Hopkins

Computer Science Department
University of California
Los Angeles, CA 90095
{darwiche,mhopkins}@cs.ucla.edu

Abstract. Darwiche has recently proposed a graphical model for driving conditioning algorithms, called a dtree, which specifies a recursive decomposition of a directed acyclic graph (DAG) into its families. A main property of a dtree is its width, and it was shown previously how to convert a DAG elimination order of width w into a dtree of width $\leq w$. The importance of this conversion is that any algorithm for constructing low-width elimination orders can be directly used for constructing low-width dtrees. We propose in this paper a more direct method for constructing dtrees based on hypergraph partitioning. This new method turns out to be quite competitive with existing methods in minimizing width. We also present methods for converting a dtree of width w into elimination orders and jointrees of no greater width. This leads to a new class of algorithms for generating elimination orders and jointrees (via recursive decomposition).

1 Introduction

Darwiche has recently proposed a graphical model, called a dtree, which specifies a recursive decomposition of a directed acyclic graph (DAG) into its families. The main application of dtrees is in driving a class of divide-and-conquer algorithms, called recursive conditioning, which can be used for anyspace probabilistic and logical reasoning [5, 3, 4]. Formally, a dtree is a full binary tree with its leaves corresponding to the DAG families (nodes and their parents). Figure 1 depicts a DAG and two corresponding dtrees.

The quality of a dtree is measured by a number of parameters. The main property of a dtree is its width. For example, if we have a belief network with n variables, and if we can construct a dtree of width w for the network, then we can answer probabilistic queries in $O(n \exp(w))$ space and time. A dtree has other important properties though. For example, if the height of a dtree is h, then we can reason about the network in $O(n)$ space and $O(n \exp(hw))$ time. Therefore, constructing dtrees with minimal width and height is quite important.

Existing methods for constructing dtrees for a DAG focus on initially constructing a good elimination order for the DAG. It was previously shown how to convert an elimination order of width w for DAG G into a dtree of width

$\leq w$ for the same DAG [5], implying that any algorithm for constructing low-width elimination orders is immediately an algorithm for constructing low-width dtrees. It was also shown that any dtree can be balanced in $O(n \log n)$ time, giving it a $O(\log n)$ height, while only increasing its width by a constant factor [5]. Therefore, to construct a dtree for linear-space reasoning, one can compute an elimination order of small width, convert it to a dtree of no greater width, and then balance the dtree to minimize its height.

We report in this paper on a new method for constructing balanced dtrees. The method is based on hypergraph partitioning, a well-studied problem with applications to many areas, including VLSI design, efficient storage of databases on disk, and data mining [10]—the goal here is to partition a hypergraph into equally-sized parts, while minimizing the edges which cross from one part to another. Specifically, we show how the process of constructing balanced dtrees for a DAG can be reduced to the process of recursively partitioning a hypergraph based on the DAG.

Although the proposed method does not directly attempt to minimize the dtree width, our experimental results show that from a width standpoint, it generates dtrees that are competitive with those produced from elimination orders based on the min-fill heuristic. Furthermore, the generated dtrees are superior when considering other properties such as height.

A key point is that our algorithm for constructing dtrees has a much broader applicability, since any algorithm for producing low-width dtrees is immediately a good algorithm for producing low-width jointrees and elimination orders. It was shown previously that any dtree for a DAG can be immediately converted into a jointree for that DAG [5]. Therefore, our new method for constructing dtrees is immediately a method for constructing jointrees with similar properties, including width. We also show in this paper that each dtree of width w naturally determines a partial elimination order. Moreover, each (total) elimination order which is consistent with this partial order is guaranteed to have a width no greater than w. The implication of these results is that any method for recursively decomposing a DAG into a dtree can be used to produce elimination orders and jointrees for that DAG, with interesting guarantees on their qualities.

This paper is structured as follows. We start in Section 2 by reviewing dtrees and their applications. We then introduce the problem of hypergraph partitioning in Section 3, where we show how it can be used to obtain balanced dtrees. We then show in Section 4 how to convert dtrees of a certain width into elimination orders and jointrees of no greater width. We next present our experimental results in Section 5 and finally close with some concluding remarks in Section 6.

2 Dtrees

A dtree is a full binary tree which induces a recursive decomposition on a directed acyclic graph. A dtree is used to drive divide–and–conquer algorithms, such as the algorithm of recursive conditioning for inference in Bayesian networks [5]. The following is the formal definition of a dtree.

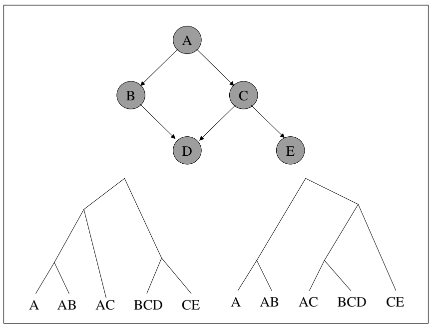

Fig. 1. A directed acyclic graph and two corresponding dtrees.

Definition 1. *A dtree T for a DAG G is a full binary tree, the leaves of which correspond to the families of G. Specifically, if t is a leaf node which corresponds to family F, we have* $\mathsf{vars}(t) \stackrel{def}{=} F$.

Figure 1 depicts two dtrees for the DAG shown in the same figure. Examine the first dtree. The top level specifies a partition of the DAG families into two sets: $\{A, AB, AC\}$ and $\{BCD, CE\}$. The left subtree specifies a partition of families $\{A, AB, AC\}$, while the right subtree specifies a partition of families $\{BCD, CE\}$ (unique in this case).

We will use t_l and t_r to denote the left child and right child of node t in a dtree. Following standard conventions on binary trees, we will often not distinguish between a node and the dtree rooted at that node. Finally, for internal nodes t, we define $\mathsf{vars}(t)$ as $\mathsf{vars}(t_l) \cup \mathsf{vars}(t_r)$.

To partition a set of families Σ into two independent sets Σ_l and Σ_r, we must delete enough DAG edges so that the families in Σ_l and Σ_r will no longer share variables. An edge is deleted by setting the node at its tail. Therefore, one measure of dtree quality is how many DAG nodes we have to set in order to induce the decomposition specified by the dtree.

Definition 2. *[5] The cutset of internal node t in a dtree,* $\mathsf{cutset}(t)$, *is defined as* $\mathsf{vars}(t^l) \cap \mathsf{vars}(t^r) - \mathsf{acutset}(t)$, *where* $\mathsf{acutset}(t)$ *is the union of cutsets associated with ancestors of t.*

That is, the cutset of dtree node t contains the DAG nodes which must be set in order to remove any shared variables between the families under t_l and those under t_r.

Figure 2 depicts Algorithm RC that uses a dtree for computing probabilities with respect to a Bayesian network [5]. The main feature of the algorithm is that it takes space which is linear in the network size. Here, we associate each conditional probability table (CPT) with a leaf node t, where $\mathsf{vars}(t)$ are the CPT variables. To compute the probability of evidence $\mathbf{e}$, all we have to do is record the instantiation $\mathbf{e}$, and then call RC(t), where t is the root of given dtree.

```
                    Algorithm RC

RC(dtreenode t)
   if t is a leaf node,
   then return lookup(t)
   else float p←0
        for each instantiation c of uninstantiated variables in cutset(t) do
            record instantiation c
            p←p+RC(t^l)RC(t^r)
            un-record instantiation c
        return p
```

Fig. 2. Pseudocode for linear-space recursive conditioning. *lookup*(t) returns the probability of the instantiation recorded on the CPT associated with leaf node t.

As evidenced by Algorithm RC, the quality of a dtree is dictated by the size of its cutsets since algorithms based on dtrees are exponential in the size of such cutsets. The height of the dtree is also important. For example, Algorithm RC takes $O(n \exp(hc))$ time, where h is the dtree height, c is the size of its largest cutset, and n is the number of its nodes.[1]

There are other measures of dtree quality, which become relevant when one is willing to use more space in order to reduce running time. In particular, one can use the technique of memoization from dynamic programming to cache some of the results of RC [5].

Definition 3. *[5] The* <u>*context*</u> *of node* t *in a dtree,* $\mathsf{context}(t)$*, is defined as* $\mathsf{vars}(t) \cap \mathsf{acutset}(t)$.

Specifically, at each node t in the dtree, one could cache the result of RC(t) indexed by the instantiation of $\mathsf{context}(t)$. If another call is made to node t under the same context instantiation, it can be retrieved from the cache instead of invoking a recursive call. This memoization scheme will then require each node t in the dtree to maintain a cache of size $\exp(c)$, where c is the size of $\mathsf{context}(t)$. The total space required is then $O(n \exp(w_c))$, where w_c is the size of the maximal context in the dtree (known as the *context width*).

A final measure of dtree quality is its width:

Definition 4. *[5] The* <u>*cluster*</u> *of node* t *in a dtree is defined as follows:*

$$\mathsf{cluster}(t) = \begin{cases} \mathsf{vars}(t), & \textit{if } t \textit{ is leaf;} \\ \mathsf{cutset}(t) \cup \mathsf{context}(t), & \textit{otherwise.} \end{cases}$$

The <u>*width*</u> *of a dtree is defined as the size of its largest cluster minus 1.*

If one can afford the space required by memoization, then one can reason in $O(n \exp(w))$ time and $O(n \exp(w_c))$ space, where w is the dtree width, w_c is its

[1] The number of dtree nodes is always twice (minus one) the number of DAG nodes.

context width, and n is the number of its nodes. In general though, one can use as much space as one can afford, while still being able to predict the average running time under the chosen amount of space [5].

Therefore, if one will not use memoization, then one is interested in minimizing the height and cutsets of a dtree. But if memoization is to be used, then one needs to minimize the width and context width of a dtree. If one is to use partial memoization, then the situation is a bit more complicated especially that there is a tension between the previous properties of a dtree. For example, the technique proposed in [5] for balancing a dtree does that at the expense of increasing its width and context width.

We discuss in Section 3 a different class of algorithms for constructing dtrees based on hypergraph partitioning. This class of algorithms attempts to minimize the height and cutsets of dtrees. Yet, we shall present experimental results in Section 5 showing that it produces very competitive dtrees from the standpoint of width and context width, at least when compared with dtrees constructed based on elimination orders. Given that one can easily convert a dtree of width w into an elimination order or jointree of width $\leq w$, the proposed method has implications on the construction of elimination orders and jointrees. This is discussed in Section 4.

3 Dtree Construction as Hypergraph Partitioning

Previous methods for constructing low-width dtrees have focused on using existing heuristics to generate low-width elimination orders, then converting these elimination orders to dtrees [5]. An alternative approach is to generate the dtrees directly. The technique we now present uses hypergraph partitioning as a tool for directly generating low-width dtrees.

A *hypergraph* is a generalization of a graph, such that an edge is permitted to connect an arbitrary number of vertices, rather than exactly two. The edges of a hypergraph are referred to as *hyperedges*. The problem of *hypergraph partitioning* is to find a way to split the vertices of a hypergraph into k approximately equal parts, such that the number of hyperedges connecting vertices in different parts is minimized [10].

The problem of hypergraph partitioning is well-studied, as it applies to many fields, including VLSI design, efficient storage of databases on disk, and data mining [10]. Since solving the problem optimally is at least NP-hard [8], much energy has been devoted to developing approximation algorithms for hypergraph partitioning. A paper by Alpert and Khang [1] surveys a variety of the approaches taken to this problem.

For our purposes, we used hMeTiS, a hypergraph partitioning package distributed by the University of Minnesota [11]. Loosely speaking, hMeTiS collapses vertices and hyperedges of the original hypergraph to produce a smaller, aggregated hypergraph, then uses various specialized algorithms to partition the smaller hypergraph. After doing this, it uses specialized algorithms to construct a partition for the original, refined hypergraph using the partition for the smaller,

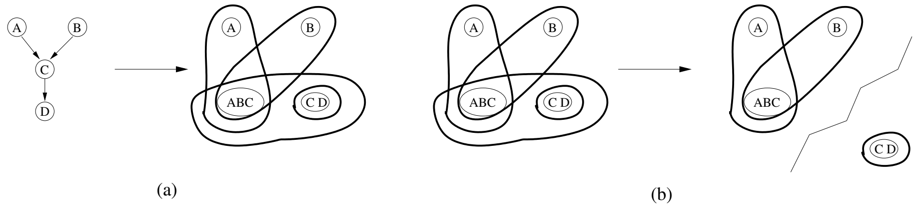

Fig. 3. (a) From a DAG to a hypergraph. (b) An example bipartitioning of the hypergraph into two subgraphs.

aggregated hypergraph. Experimental results have shown that the partitions produced by hMeTiS are consistently better than those produced by other popular algorithms [11]. In addition, hMeTiS is between one and two orders of magnitude faster than other algorithms [11]. One of the useful features of hMeTiS is that the user can specify how balanced the partition will be. Concretely, the user can specify that each part must contain no less than $X\%$ of the vertices.

Generating a dtree for a DAG using hypergraph partitioning is fairly straightforward. The first step is to express the DAG G as a hypergraph H:

- For each family F in DAG G, we add a node N_F to H.
- For each variable V in DAG G, we add a hyperedge to H which connects all nodes N_F such that $V \in F$.

An example of this is depicted in Figure 3(a).

Notice that any full binary tree whose leaves correspond to the vertices of H is a dtree for our DAG. This observation allows us to design a simple algorithm using hypergraph partitioning to produce a dtree. Figure 4 shows the pseudocode for this algorithm. Essentially, it partitions the hypergraph into two sets of vertices, then recursively generates dtrees for each set, and finally combines the resulting dtrees into a new dtree (whose left child is the dtree for the first set, and whose right child is the dtree for the second set).

HGR2BDT attempts to minimize the cutset at each node in the dtree. To see this, observe that every time we partition the hypergraph H into H_l and H_r, we attempt to minimize the number of hyperedges that span the partitions H_l and H_r. By construction, these hyperedges correspond to DAG variables that are shared by families (vertices) on both sides of the cut. Hence by attempting to minimize the number of hyperedges that span our hypergraph cut, we are actually attempting to minimize the number of variables that are shared by the left and right subtrees of the dtree that we are producing! Notice that we do not make any direct attempt to minimize the width of the dtree. However, we shall see in Section 5 that local cutset minimization is actually a good heuristic for developing low-width dtrees.

An advantage to this approach is that it also produces balanced dtrees, in the sense that for any node in the dtree, the ratio of the number of leaves in its left subtree to the number of leaves in its right subtree is bounded. This is a direct consequence of the fact that hMeTiS computes balanced hypergraph partitions.

Algorithm HGR2BDT

```
HGR2BDT(hypergraph H)
dtreenode t
if H has only one vertex N_F,
      then vars(t) ← F
            t_l ← NULL
            t_r ← NULL
      else partition H into two subgraphs H_l and H_r
            t_l ← HGR2BDT(H_l)
            t_r ← HGR2BDT(H_r)
            vars(t) ← vars(t_l) ∪ vars(t_r)
return t
```

Fig. 4. Pseudocode for producing dtrees using hypergraph partitioning.

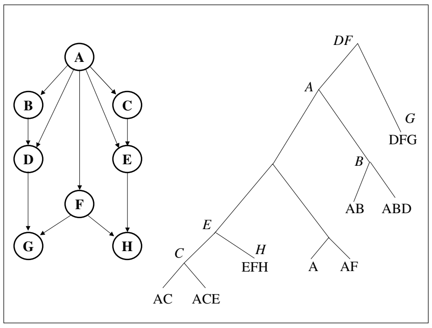

Fig. 5. Converting a dtree into an elimination order.

Thus the algorithm computes dtrees that have height of $O(\log n)$, where n is the number of nodes in the given DAG.

4 From Dtrees to Elimination Orders and Jointrees

In this section, we discuss width-preserving transformations from dtrees to elimination orders, and from dtrees to jointrees. The implication of such transformations is that any algorithm for constructing low-width dtrees is immediately an algorithm for constructing low-width elimination orders and jointrees. We will begin our discussion by reviewing the concept of an elimination order.

An *elimination order* of an undirected graph G is an ordering $\pi(1), \pi(2), \ldots$ of the nodes in G. One of the simplest ways for defining the *width* w of order π is constructively. Simply eliminate nodes $\pi(1), \pi(2), \ldots, \pi(n)$ from G in that order, connecting all neighbors of a node before eliminating it. The maximum number of neighbors that any eliminated node has is then the width of order

π. The width of an elimination order with respect to a DAG G is defined as its width with respect to the moral graph of G–that is, the graph which results from connecting all parents of each node, and then dropping the directionality of edges.

Elimination orders are the basis of an important class of algorithms, known as variable elimination algorithms [6, 14]. They are also the basis for constructing jointrees, which drive clustering algorithms [9, 13]. In both cases, the complexity of algorithms is exponential only in the width of given elimination order w. Hence, generating low-width elimination orders is critical for the efficiency of these algorithms.

An algorithm is presented in [5] for converting an elimination order of width w into a dtree of width $\leq w$. The method allows one to capitalize on algorithms for constructing low-width elimination orders in order to construct low-width dtrees. Here, we present a result which allows us to do the opposite. Specifically, we show how a dtree of width w can be used to induce elimination orders of width $\leq w$. In fact, we show that each dtree specifies a partial elimination order, and any total order consistent with it is guaranteed to have no greater width.

Definition 5. *Let T be a dtree for DAG G. We say that node v of G is eliminated at node t of T precisely when $v \in \mathsf{cluster}(t) - \mathsf{context}(t)$.*

Note that for an internal node t, $\mathsf{cluster}(t) - \mathsf{context}(t)$ is precisely $\mathsf{cutset}(t)$ [5]. Figure 5 depicts a dtree and the DAG nodes eliminated at each of its nodes.

Theorem 1. *Let T be a dtree for DAG G. Then every DAG node is eliminated at some unique dtree node in T.*

This allows us to define a partial elimination order, where for each DAG nodes v and u, we have $v < u$ iff the dtree node at which v is eliminated is a descendant of the dtree node at which u is eliminated.

In the dtree of Figure 5, we have $C < E < A < D, F$. We also have $H < E$, $B < A$ and $G < D, F$. Any total elimination order consistent with these constraints is guaranteed to have no greater width than that of the dtree.

Theorem 2. *Let T be a dtree of width w for DAG G and let π be a total elimination order for G which is consistent with the partial elimination order defined by T. The width of π is then $\leq w$.*

The following two orders are consistent with the dtree in Figure 5: $< C, H, E, B, A, G, D, F >$ and $< H, C, B, E, G, A, F, D >$. Each of these elimination orders has width 2. It is easy to generate an elimination order which is consistent with a given dtree through a post-order traversal of the dtree.

Therefore, if we have an algorithm for constructing low-width dtrees, then we immediately have an algorithm for constructing low-width elimination orders.

A similar result exists for converting a dtree of width w into a jointree of the same width [5]. We review the result here as it allows us to put the experimental results of Section 5 in broader perspective. We start with the formal definition of a jointree.

A *jointree* for DAG G is a pair (T, C), where T is a tree and C labels each node in T with a subset of nodes in G such that

1. Each family of DAG G is contained in some label $C(v)$.
2. For every three nodes v, u and w in T, if w is on the path connecting v and u, then $C(v) \cap C(u) \subseteq C(w)$.

Each label $C(v)$ is called a *cluster,* and the *width* of a jointree is defined as the size of its largest cluster minus one. Another important aspect of a jointree is its separators: for each edges (u, v) in the jointree, one defines the *separator* as $C(u) \cap C(v)$. The running time of algorithms based on jointrees is exponential in the width. Their space complexity, however, can be only exponential in the size of the separators.

It is shown in [5] that if T is a dtree for a DAG G, and if C is a function that maps each node in dtree T to its cluster (as defined in Definition 4), then (T, C) is a jointree for DAG G. Moreover, the contexts of the dtree correspond to the separators of the induced jointree. This means that one can easily convert a dtree into a jointree of the same width. It also means that if the dtree have small contexts, then the jointree will have small separators. Finally, if the dtree is balanced, then the jointree it induces will be also balanced in the following sense. We can choose a jointree node (call it the root) so that the distance from the root to any jointree leaf is $O(\log n)$, where n is the number of DAG nodes. This property is quite desirable for jointree algorithms as it reduces the amount of message propagation needed when the evidence on a small number of variables changes.

5 Experimental Results

We compare experimentally in this section two methods for constructing dtrees: one based on elimination orders and another based on hypergraph partitioning. The first method generates unbalanced dtrees, while the second generates balanced ones. As long as the two methods are comparable with regards to the width of dtrees they generate, we will prefer balanced dtrees. There are many heuristics for generating low-width elimination orders [12], but it is well accepted that the min-fill heuristic is among the best. This is the one we use in our experiments.

To build dtrees for our set of benchmark suites with HGR2BDT, we implemented HGR2BDT in C++ using the Standard Template Library, as well as the hMeTiS hypergraph partitioning package from the University of Minnesota [11]. Recall that hMeTiS allows the user to specify how balanced each partition will be. We varied this parameter such that hMeTiS could produce bipartitions of maximum ratio 51-49, 60-40, 70-30, 80-20, and 90-10. For example, for ratio 60-40, the larger part of the bipartition could be comprised of at most 60% of the vertices of the original hypergraph. Since hMeTiS is also nondeterministic, we ran 5 trials at each balance setting, and then took the best dtree (in terms of width) from the 25 total trials. This is the dtree that we report in our results.

Table 1. Statistics for ISCAS'85 Benchmark Circuits.

Circuit	hgr2bdt			Min-fill					
				Unbalanced			Balanced		
	Width	Context Width	Height	Width	Context Width	Height	Width	Context Width	Height
c432	27	23	11	27	23	16	27	23	12
c499	22	19	13	24	25	47	31	25	13
c880	23	22	13	25	24	42	29	25	16
c1355	22	19	24	24	25	49	31	25	16
c1908	44	32	13	50	43	23	51	46	16
c2670	33	29	22	37	32	39	37	29	19
c3540	74	61	15	97	81	73	97	81	19
c5315	52	49	16	45	44	79	53	51	19
c6288	46	38	35	53	43	48	53	43	19
c7552	42	35	17	48	37	41	51	42	21

Table 2. Results for Suite of Belief Networks.

Network	hgr2bdt		Min-fill			
			Unbalanced		Balanced	
	Width	Height	Width	Height	Width	Height
barley	7	10	7	19	8	9
diabetes	7	11	4	53	9	14
link	16	17	15	33	19	17
mildew	5	7	4	13	7	8
munin1	11	10	11	31	12	12
munin2	9	13	7	47	9	17
munin3	8	16	7	35	10	17
munin4	9	13	8	37	10	18
pigs	11	11	10	38	14	14
water	10	7	10	12	10	9

Our first suite of DAGs is obtained from the ISCAS'85 benchmark circuits [2]. These circuits have been studied by El Fattah and Dechter in [7], wherein elimination orders were generated using several well-known heuristics. We found that min-fill produced better orders than any of the heuristics surveyed in [7]. Hence we used min-fill to construct elimination orders for these circuits, then constructed dtrees based on these orders in the manner described by [5]. We also constructed dtrees using HGR2BDT. The results are reported in Table 1. A third class of dtrees is also reported, which results from balancing the first class using a technique described in [5].

A number of observations are in order here. First, if all we care about is generating low-width elimination orders, then constructing a dtree using HGR2BDT and extracting an elimination order from it is almost always (much) better than using the min-fill heuristic. A particularly dramatic example of this is c3540, for which HGR2BDT was able to produce an elimination order of width 74. By contrast, min-fill produced an elimination order of width 97, while the best heuristic surveyed by El Fattah and Dechter in [7] produced an elimination order of width 114. Interestingly enough, these dtrees not only lead to better elimination orders, but are also balanced and tend to have smaller contexts. Therefore, HGR2BDT appears to be favorable for constructing dtrees and jointrees as well, for which other properties (beyond width) are of interest.

Our second class of DAGs is obtained from belief networks posted at http://www.cs.huji.ac.il/labs/compbio/Repository/; see Table 2. For these networks, the min-fill heuristic (without balancing) did better overall than either of the two methods that generate balanced dtrees. So we are paying a price here for

Table 3. Results for Randomly Generated DAGs.

Number of nodes	Edge prob.	Version	hgr2bdt			Min-fill					
						Unbalanced			Balanced		
			Width	Context Width	Height	Width	Context Width	Height	Width	Context Width	Height
200	.015	1	22	20	10	22	21	37	22	21	13
		2	34	28	10	32	31	42	33	31	13
		3	28	23	11	28	26	35	31	27	13
		4	29	25	10	29	28	42	30	28	13
		5	29	26	10	27	26	38	29	26	13
300	.008	1	29	25	11	31	30	47	31	30	14
		2	24	21	11	24	23	37	25	23	13
		3	33	29	11	33	33	50	35	32	14
		4	29	25	11	31	30	40	31	30	15
		5	30	27	11	31	30	49	32	31	14
400	.005	1	21	19	11	21	20	50	23	20	14
		2	20	18	11	20	19	45	21	20	15
		3	17	16	11	15	15	42	18	19	15
		4	18	16	11	18	19	42	21	19	15
		5	22	19	11	24	23	44	24	23	15
500	.004	1	16	15	11	16	14	39	16	14	15
		2	21	20	14	22	21	44	24	22	15
		3	23	21	12	24	22	51	24	22	14
		4	26	23	11	25	23	48	28	22	15
		5	23	22	11	23	22	32	23	22	16

balance, although it does not seem to be too high.[2] The highest price appears to be for network *diabetes,* which has 413 nodes and whose dtree height went from 53 to 11. What is clear though is that generating balanced dtrees using HGR2BDT appears to be superior to generating dtrees using an elimination order and then balancing them.

Our third suite of DAGs is generated randomly according to the given probabilities of edges; see Table 3. For this suite, the use of HGR2BDT for generating dtrees, jointrees and elimination orders seems to produce the best results overall, considering width, context width and height.

It is worth noting that the execution time of HGR2BDT is reasonable. For the largest network in our testing set, c7552 (a network with 7230 vertices), HGR2BDT takes approximately 5 minutes to produce a dtree on a Pentium II 266. For most of the smaller networks, the execution time of HGR2BDT is only a matter of seconds.

6 Conclusion

This paper rests on two contributions, one theoretical and another practical. Theoretically, we have shown how methods for recursively decomposing DAGs can be used to construct elimination orders, dtrees and jointrees. Practically, we have proposed and evaluated the use of a state-of-the-art system for hypergraph partitioning to recursively decompose DAGs and, hence, to construct elimination orders, dtrees and jointrees. The new method appears to be different from current tradition in automated reasoning, where elimination orders are the basis of constructing various graphical models. There are many heuristics

[2] We are treating these networks as having variables with equal cardinalities, which is a simplifying assumption. The situation is more complex if we decide to take variable cardinalities into account.

for generating low-width elimination orders, and it is customary for automated reasoning systems to give the user a choice of which one to use since even a small reduction in width can have a drastic practical effect. Our experimental results suggest that the construction of graphical models based on hypergraph partitioning should clearly be considered as one of these choices, whether one is interested in elimination orders, jointrees, or dtrees.

References

1. Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning. *Integration, the VLSI Journal*, 19(1–81), 1995.
2. F. Beglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN. In *Proceedings of the IEEE symposium on Circuits and Systems*, 1985. http://www.cbl.ncsu.edu/www/CBL_Docs/iscas85.html.
3. Adnan Darwiche. Compiling knowledge into decomposable negation normal form. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 284–289, 1999.
4. Adnan Darwiche. Utilizing device behavior in structure–based diagnosis. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1096–1101, 1999.
5. Adnan Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, February, 2001.
6. Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 211–219, 1996.
7. Yousri El Fattah and Rina Dechter. An evaluation of structural paramters for probabilistic reasoning: Results on benchmark circuits. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 244–251, 1996.
8. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, CA, 1979.
9. 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.
10. George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. *IEEE Transactions on VLSI Systems*, 1998.
11. George Karypis and Vipin Kumar. *hMeTiS: A Hypergraph Partitioning Package*, 1998.
12. U. Kjaerulff. Triangulation of graphs—algorithms giving small total state space. Technical Report R-90-09, Department of Mathematics and Computer Science, University of Aalborg, Denmark, 1990.
13. 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.
14. Nevin Lianwen Zhang and David Poole. Exploiting causal independence in bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.

A Proofs

Proof of Theorem 1

Follows since:

- All the cutsets of a dtree are disjoint, hence, a variable cannot be eliminated at two different internal nodes.
  By definition of a cutset, all cutsets on the same path from root to leaf are disjoint. Now suppose that two cutsets contain the same variable X and are descendants of node t in a dtree. By definition of cutset, either the cutset of t or the cutset of an ancestor of t must contain X. This is a contradiction.
- A variable X is eliminated at a leaf node t only if $X \in \mathsf{vars}(t)$ and $X \notin \mathsf{vars}(t')$ for any other leaf node t'–otherwise, $X \in \mathsf{context}(t)$ and will not be eliminated at t. Hence, if a variable is eliminated at a leaf node, then it cannot be eliminated at any other leaf node. Morever, by the previous property and definition of cutset, X cannot appear in any dtree cutset. Hence, it cannot be eliminated at an internal node either. □

Proof of Theorem 2

Given a decomposition tree T, which is annotated by eliminated variables at each node, let us perform a variable elimination process where variables are eliminated at the children of a node before they are eliminated at the node itself:

- When eliminating variables at a leaf node t, we sum-out these variables (in any order) from the table at t and store the result at t. Note that if a variable X is eliminated at a leaf node t, then X appears only at that leaf node.
- When eliminating variables at a non-leaf node t, we multiply the tables at the children of t; sum-out the variables (in any order) from the multiplication; and store the result at t. Note that if $X \in \mathsf{cutset}(t)$, then X must appear in the tables stored at the children of t.[3] Hence, when eliminating X, the two tables must be multiplied together. When $\mathsf{cutset}(t)$ is empty though, there is no need to multiply the two tables but we will multiply them without loss of generality.

We can prove that the table stored at a node t during the elimination process is defined over $\mathsf{context}(t)$. This clearly follows for tables stored at each leaf node. For a table stored at a non-leaf node, it follows since $\mathsf{context}(t_l) \cup \mathsf{context}(t_r) = \mathsf{cluster}(t)$ and, hence, $(\mathsf{context}(t_l) \cup \mathsf{context}(t_r)) - \mathsf{cutset}(t) = \mathsf{context}(t)$ [5].

Now, to prove that the width of any of the elimination orders above is $\leq w$, we need to prove that no table will have more than $w + 1$ variables in the above elimination process. This follows since $\mathsf{context}(t_l) \cup \mathsf{context}(t_r) = \mathsf{cluster}(t)$. Therefore, at no point will the elimination scheme have a table with more than

[3] By definition of a cutset, X must appear in some table of dtree t_l and in some table of dtree t_r. Moreover, by Theorem 1, X could not have been eliminated earlier.

$|\,\mathsf{cluster}(t)\,|$ variables, for some node t in the dtree. Note, however, that we may construct a table over variables $\mathsf{cluster}(t)$ when we can afford not to (that is, when $\mathsf{cutset}(t) = \emptyset$). Therefore, the width of the elimination order may actually be less than w. Here is an example: DAG $5 \rightarrow 0 \rightarrow 1, 5 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6$. The elimination order $6, 4, 1, 0, 5, 2, 3$ has width 1. Yet, it is compatible with a dtree of width 2: $((46) * (34)) * (((23) * (5)) * ((25) * ((05) * (01))))$. $\square$