A Practical Relaxation of Constant-Factor Treewidth Approximation Algorithms

Mark Hopkins
Dept. of Computer Science
University of California, Los Angeles
Los Angeles, CA, 90095, USA
mhopkins@cs.ucla.edu

Adnan Darwiche
Dept. of Computer Science
University of California, Los Angeles
Los Angeles, CA, 90095, USA
darwiche@cs.ucla.edu

Abstract

Algorithms that triangulate a graph such that the resulting triangulation is at most a constant-factor from the treewidth are important theoretically, but perform poorly in practice. In this paper, we show that a successful heuristic method for triangulating graphs (based on a structure known as a dtree) can be viewed as a relaxation of a well-known family of constant-factor approximation algorithms. In doing so, we provide a theoretical underpinning for this heuristic, and show how these theoretically elegant approximation algorithms can be effectively applied in practice.

1 Introduction

Given an undirected graph G and a triangulation for G of bounded width, there are a wide variety of graph operations that can be performed in polynomial time. Applications of interest to artificial intelligence researchers include constraint satisfaction problems (Dechter and Pearl, 1989) and Bayesian network inference (Lauritzen and Spiegelhalter, 1988; Jensen et al., 1990; Dechter, 1996; Darwiche, 2001). Hence, it is desirable to find algorithms that for a given graph G, return a graph triangulation with width as close as possible to the treewidth of G, i.e. the width of the best existing triangulation.

Since finding an optimal graph triangulation is an NP-hard problem (Arnborg et al., 1987), much attention has been given to providing algorithms that triangulate a graph such that the width of the triangulation is guaranteed to be within some (not-necessarily constant) factor of the treewidth. Although polynomial-time algorithms have been presented that approximate the treewidth k of a graph with n vertices to within a factor of $lg(k)$ (Amir, 2001) and $lg(n)$ (Kloks, 1994; Bodlaender et al., 1995), it is still an open question as to whether a polynomial-time algorithm exists that approximates the treewidth to within a constant factor. Existing algorithms by (Robertson and Seymour, 1995; Reed, 1992; Becker and Geiger, 1996; Amir, 2001; Kloks, 1994) run in time exponential in the treewidth.

In this paper, we show that a recent heuristic algorithm that provides good approximations to treewidth (but no guarantees) can be viewed as a relaxation of the family of constant-factor approximation algorithms presented in (Robertson and Seymour, 1995; Reed, 1992; Becker and Geiger, 1996; Amir, 2001; Kloks, 1994). This algorithm has been shown to be both fast and competitive with the best known algorithms for generating low-width graph triangulations (Darwiche and Hopkins, 2001).

The algorithm is presented in terms of a graphical data structure known as a decomposition tree (dtree). Every dtree has a width associated with it, and can be transformed in polynomial time into a graph triangulation of equivalent or lesser width. Thus, any algorithm for constructing low-width dtrees is immediately an algorithm for constructing low-width graph triangulations. Hence, after reviewing in Sections 2 and 3 the basics of graph triangulation, treewidth, and the existing algorithms that approximate treewidth to within a constant factor, we proceed in Section 4 by defining dtrees in terms of undirected graphs (previously, it was defined in terms of directed acyclic graphs). We then review our heuristic algorithm in Section 5 and show in Section 6 that it can be viewed as a relaxation of the exponential-time approximation algorithms. This perspective gives us an alternate formulation of these algorithms in terms of dtrees that allows us to gain further insight and intuition. Moreover, it demonstrates how the theoretical ideas behind these exponential-time algorithms can be exploited for practical usage.

2 Graph Triangulations and Treewidth

In this section, we review the central concepts of a graph triangulation and treewidth. First, we need some basic graph terminology. Unless stated otherwise, we will use the term *graph* to denote an undirected graph $G = (V, E)$, where V is a finite set of vertices, and E is a finite set of edges (we disallow the possibility of self-loops and multi-edges). The *vertex set* of a graph G is denoted $V(G)$, while the *edge set* is denoted $E(G)$. A pair of vertices $u, v \in V(G)$ are said to be *adjacent* if $(u, v) \in E(G)$. A graph G is *complete* if every pair of vertices of $V(G)$ are adjacent.

A *path* between two vertices $v_0, v_j \in V(G)$ is a sequence of distinct vertices $[v_0, v_1, ..., v_{j-1}]$ of $V(G)$ such that for $1 \leq i \leq j$, $(v_{i-1}, v_i) \in E(G)$. A *cycle* is a path from a vertex v back to v.

We now have enough terminology to discuss graph triangulations and define treewidth. Given an undirected graph, we *triangulate* it by adding edges such that we eliminate all chordless cycles of length greater than 3. That is to say, we ensure that for every cycle of length greater than 3, there exists some edge between two non-adjacent vertices of the cycle. A graph satisfying this property is said to be *triangulated.* Given an undirected graph $G = (V, E)$, we define a *triangulation* of G as a supergraph $G' = (V, E')$ of G such that G' is triangulated. We can define the *width* of a triangulated graph as the size of its largest clique, minus one. Furthermore, for an undirected graph G, we can define its *treewidth* as the minimum width over all possible triangulations of G.

3 Approximation Algorithms

Optimal algorithms for computing graph triangulations are generally very restricted in the types of graphs for which they can compute triangulations in a reasonable time frame (Arnborg et al., 1987; Shoikhet and Geiger, 1997; Bodlaender, 1993). As a result, there has been an emphasis on designing *approximation algorithms* that triangulate a graph G such that the width of the resulting graph is guaranteed to be within some (not-necessarily constant) factor of the treewidth of G. It is currently unknown whether a constant-factor approximation can be found in polynomial time. Existing algorithms by (Robertson and Seymour, 1995; Reed, 1992; Becker and Geiger, 1996; Amir, 2001; Kloks, 1994) run in time exponential in the treewidth.

All of these algorithms are variants on the constant-factor approximation algorithm presented by (Robertson and Seymour, 1995), which provides a factor-4 approximation. In this section we describe one formulation of this algorithm, similar to the formulations presented in (Reed, 1992) and (Amir, 2001).

It will help to introduce some more graph terminology. The subgraph of G induced by a subset S of the vertices of G, denoted $G[S]$, is the graph over S consisting of the edges of G that connect two vertices of S. Formally, $G[S] \overset{def}{=} (S, \{(u, v) \in E(G) | u, v \in S\})$. A *clique* of G is a subset S of the vertices of G such that $G[S]$ is a complete graph. For a subset S of the vertices of G, $G \backslash S$ is defined as the graph that results from removing S from G, along with all edges incident to vertices in S. Formally, $G \backslash S \overset{def}{=} (V(G) \backslash S, \{(u, v) \in E(G) | u, v \in (V(G) \backslash S)\})$.

A graph is *connected* if there exists a path between every pair of vertices of G and *disconnected* otherwise. A *vertex separator* (also called simply *separator*) is a set S of vertices

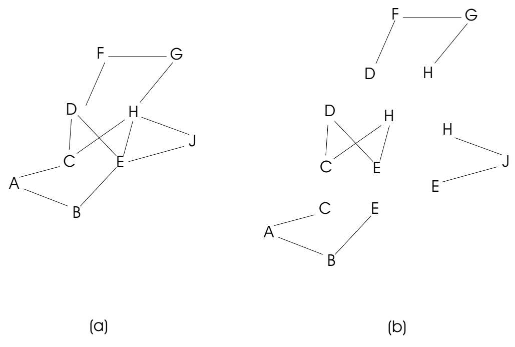


Figure 1: (a) An example graph with separator $CDEH$. (b) The edge partition of the graph with respect to separator $CDEH$.

of G such that $G\backslash S$ is disconnected. If S is a maximal set of vertices of G such that $G[S]$ is connected, then we refer to $G[S]$ as a *connected component* of G.

The *edge subgraph* of G induced by a subset R of the edges of G, denoted $G[R]$, is defined as the graph consisting of the set of edges R, along with the vertices that the edges of R are incident to. Formally, $G[R] \overset{def}{=} (\{v \in V(G) | (\exists u \in V(G))((u,v) \in R)\}, R)$.

Each vertex separator induces a unique partition of the graph edges, which we define next; see Figure 1. Suppose S is a vertex separator of graph G, and let C be a connected component of $G\backslash S$. Let R be the subset of $E(G)$ containing edges between two nodes in C and between a node in S and a node in C. We will refer to $G[R]$ as an *edge component* of graph G with respect to separator S. Clearly for each connected component of $G\backslash S$, there exists a unique corresponding edge component of G with respect to S. We define $(C_0, C_1, ..., C_k)$ as the *edge partition* of graph G with respect to separator S, where $C_0 = G[S]$, and the $C_i, 1 \leq i \leq k$ are the edge components of G with respect to S. Notice that the edge partition is unique, and that the collection of edge subgraphs form a partition of the edges of G (i.e. every edge of G appears in exactly one of the graphs C_i). We show an edge partition for an example graph in Figure 1. We can further define a *2-way edge partition* with respect to separator S by arbitrarily merging the components of the edge partition into two graphs, which together make up a partition of the edges of the original graph. Note that there are many possible 2-way edge partitions for a given separator.

Algorithm FACTOR4

```
FACTOR4(graph $G = (V, E)$, set $W \subseteq V$, int $k$)
    01. If $|V| \leq 4k + 1$, then make a clique of $G$
        and return.
    02. Find a vertex separator $X$ in $G$ such that
        $|X| \leq k + 1$, and a 2-way edge partition
        $(G_0 = (V_0, E_0), G_1 = (V_1, E_1))$ of $G$ with
        respect to $X$ such that neither $V_0\backslash X$ nor
        $V_1\backslash X$ contains more than 2/3 of the
        variables in $W$. If these cannot be found,
        then return "treewidth greater than $k$."
    03. Call FACTOR4($G_0$, $(V_0 \cap W) \cup X$, $k$).
    04. Call FACTOR4($G_1$, $(V_1 \cap W) \cup X$, $k$).
    05. Add edges between vertices of $W \cup X$ so
        that $G[W \cup X]$ is a complete graph.
```

Figure 2: Pseudocode for a 4-factor treewidth approximation algorithm. The algorithm modifies the original input graph at all levels of recursion.

The algorithm that approximates treewidth to within a factor of 4 is given in Figure 2. Calling FACTOR4(G,$\emptyset$,k) returns either a valid answer that the treewidth of G is greater than k, or a triangulation of G with width at most $4k + 1$. This is proven in (Robertson and Seymour, 1995; Reed, 1992). The key to this algorithm is step 2. Given that the treewidth of a graph G is k (or less), it is always possible to find a vertex separator X of size $k + 1$ (or less) such that the connected components of $G\backslash X$ contain at most 2/3 of the vertices of any given subset of the vertex set. Hence we can see that the size of W will never exceed $3k + 1$ on any recursive call $(2/3 * (3k + 1) + (k + 1) < 3k + 2)$. Thus the maximum clique size in the resulting triangulated graph will be $(3k+1)+(k+1) = 4k+2$, which implies that the algorithm will always return a triangulation of width at most $4k + 1$.

The main computational drawback of this algorithm lies in the search for an appropriate vertex separator. All existing algorithms thus far use subroutines that run in time exponential in the treewidth of the graph.

4 Dtrees

We now turn our attention to a successful heuristic algorithm presented in (Darwiche and Hopkins, 2001) that is much faster than the approximation algorithms of the previous section, and also produces far better graph triangulations in practice (Darwiche and Hopkins, 2001; Amir, 2001). This algorithm was originally conceived in the context of a structure called a decomposition tree (dtree), and this is how we will introduce it. Later, we will provide a firm theoretical foundation for this heuristic by showing how it can be viewed as a relaxed version of the approximation algorithm of the previous section.

A *dtree* (decomposition tree) is a full binary tree which induces a recursive decomposition of the edges of an undirected graph. Dtrees are used to drive divide–and–conquer algorithms, such as the algorithm of recursive conditioning for inference in Bayesian networks (Darwiche, 2001). Previously, dtrees had only been defined in terms of directed acyclic graphs (DAGs). Here, we provide an analogous definition for undirected graphs.

Definition 1 *A dtree T for an undirected graph G is a full binary tree, the leaves of which correspond to the edges of G. If t is a leaf node in dtree T which corresponds to edge (v, w) of G, we define* $\mathsf{vars}(t) \overset{def}{=} \{v, w\}$.

Figure 3 depicts two dtrees for the graph shown in the same figure. Examine the second dtree. The root node specifies a partition of the graph edges into two sets: $\{AB, AC\}$ and $\{BC, CD, BD, CE\}$. The left subtree specifies a partition of edges $\{AB, AC\}$ (in this case, unique), while the right subtree specifies a partition of edges $\{BC, CD, BD, CE\}$.

We will use t_l and t_r to denote the left child and right child of node t in a dtree, respectively. Following standard conventions on binary trees, we will often not distinguish between a node and the dtree rooted at that node. Notice that the dtree rooted at any internal node represents a subset of the edges of the original graph. This can alternatively be viewed as a graph containing precisely these edges. We will denote the edge subgraph represented by node t of a dtree as $G(t)$.

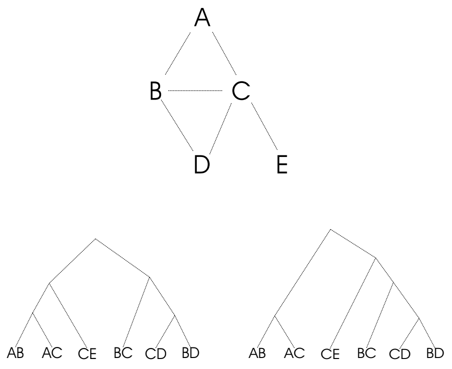


Figure 3: A graph and two corresponding dtrees.

Let us give a flavor of the algorithmic utility of dtrees. We first extend the definition for the *variables* of a dtree node to include internal nodes. For an internal node t of a dtree, $\mathsf{vars}(t) \overset{def}{=} \mathsf{vars}(t_l) \cup \mathsf{vars}(t_r)$.

Next we introduce the important concept of a *context*. Given a subset of edges R of $E(G)$, we refer to the *subgraph boundary* of $G[R]$ as the set of vertices shared between $G[R]$ and $G[E(G) \backslash R]$. We can naturally consider the subgraph boundary of $G(t)$ for any node t of a dtree. Notice that this gives us the set of variables that are shared between the graph represented by node t and the rest of the original graph. We refer to this as the *context* of t. In practical terms, it allows an algorithm to deduce which variables from the overall computation are particularly relevant to the subcomputation represented by t.

Furthermore, each internal dtree node t gives a recipe for splitting $G(t)$ into two subgraphs. We refer to those variables that are shared between these two subgraphs (that are not already in the context of t) as the *cutset* of t. Formally, the *cutset* of t is defined as $\mathsf{cutset}(t) \overset{def}{=} \mathsf{vars}(t_l) \cap \mathsf{vars}(t_r) - \mathsf{context}(t)$.

It is convenient to give a new name to the union of cutsets and contexts, which summarizes those variables that are "relevant" at a

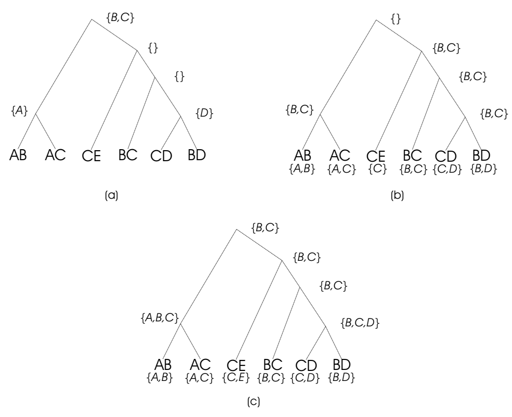


Figure 4: (a) A dtree and its cutsets (in italic). (b) A dtree with its contexts (in italic). (c) A dtree with its clusters (in italic).

given dtree node. We define

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

The *width* of a dtree is defined as the size of its largest cluster minus 1. Similarly, the *context width* is the size of its largest context, while the *cutset width* is the size of its largest cutset.

Figure 4(a) shows a dtree and its corresponding cutsets. Figure 4(b) shows the dtree contexts and Figure 4(c) shows its clusters.

The ways in which cutsets and contexts are used by conditioning algorithms are outside the scope of this paper, but we refer the reader to (Darwiche, 2001; Darwiche, 1999a; Darwiche, 1999b) for details. Here, we only focus on the significance of these sets from a complexity viewpoint. Specifically, we can use the algorithm of recursive conditioning given in (Darwiche, 2001) to compute the probability of some variable instantiation using running time exponential in the width of the dtree (under space exponential in the context width). Alternatively, we can do this computation using running time exponential in the height multiplied by the cutset width (under linear space). In fact, the above complexity results represent two extremes on a time-space tradeoff spectrum. In general, we can use any amount of space we have available, and still be able to predict the average running time of recursive conditioning (Darwiche, 2001).

(Hopkins and Darwiche, 2002) demonstrates polynomial-time transformations between dtrees and graph triangulations that preserve width. In other words, for a given graph G, it is possible to convert a triangulation of G with width w into a dtree of G with width at most w, and vice versa.

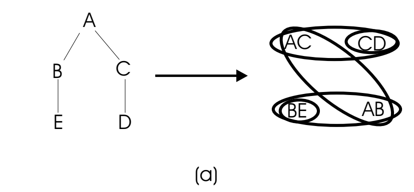
Hence there is a dual motivation for providing algorithms that construct dtrees directly. Firstly, when we are interested in properties of the dtree like height, context width, and cutset width, we have no control over such properties when constructing other data structures (like graph triangulations and elimination orders), since they are not classically defined for such structures. Thus, when we are interested in optimizing such properties, it is to our advantage to construct dtrees directly. Secondly, due to the polynomial-time transformations discussed above, any algorithm for constructing dtrees of low width are immediately good algorithms for providing low-width graph triangulations. In the next section, we detail our heuristic method for constructing dtrees.

5 Dtree Construction as Hypergraph Partitioning

The structure of dtrees lend themselves to a heuristic method based on recursive decomposition of the graph. 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 (Karypis et al., 1998).

For our purposes, we used hMeTiS, a hypergraph partitioning package distributed by the University of Minnesota (Karypis and Kumar, 1998). A more detailed account of hyper-



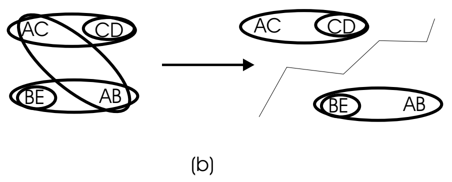


Figure 5: (a) From the undirected graph to a hypergraph. (b) An example bipartitioning of the hypergraph into two subgraphs.

graph partitioning and the various algorithmic approaches can be found in (Darwiche and Hopkins, 2001).

Generating a dtree for an undirected graph using hypergraph partitioning is fairly straightforward. The first step is to express the graph $G = (V, E)$ as a hypergraph H:

- For each edge $e \in E$, we add a node N_e to H.
- For each vertex $v \in V$, we add a hyperedge to H which connects all nodes N_e such that v is an endpoint of e.

An example of this is depicted in Figure 5(a).

Notice that any full binary tree whose leaves correspond to the edges of H is a dtree for our graph. This observation allows us to design a simple recursive algorithm using hypergraph partitioning to produce a dtree. Figure 6 shows the pseudocode for this algorithm. HGR2BDT starts by creating a dtree node t at Line 01. Lines 02-05 correspond to the base case where hypergraph H contains a single vertex N_e (corresponding to edge $e = (u, v)$) and, hence, leads to a unique dtree which contains the single leaf node t with $\mathsf{vars}(t) \leftarrow \{u, v\}$. Lines 06-08 correspond to the recursive step where hypergraph H has more than a single vertex. Here, we partition the hypergraph H into two subgraphs H_l and H_r, then recursively generate dtrees

Algorithm HGR2BDT

```
HGR2BDT(hypergraph $H$)
  01. create dtreenode $t$
  02. if $H$ has only one vertex $N_F$,
  03.    then vars($t$) $\leftarrow F$
  04.         $t_l \leftarrow$ NULL
  05.         $t_r \leftarrow$ NULL
  06.    else partition $H$ into subgraphs $H_l$ and $H_r$
  07.         $t_l \leftarrow$ HGR2BDT($H_l$)
  08.         $t_r \leftarrow$ HGR2BDT($H_r$)
  09. return $t$
```

Figure 6: Pseudocode for producing dtrees using hypergraph partitioning.

HGR2BDT(H_l) and HGR2BDT(H_r) for these subgraphs, and finally set these dtrees as the children of dtree node t.

HGR2BDT attempts to minimize the cutset of each node t it constructs at Line 01. 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 graph vertices that are shared by edges in H_l and those in H_r (which have not already been cut by previous partitions). Hence by attempting to minimize the number of hyperedges that span the partitions H_l and H_r, we are actually attempting to minimize the cutset associated with dtree node t. Notice that we do not make any direct attempt to minimize the width of the dtree. However, we shall see in the following section that cutset minimization is a good heuristic for dtree width minimization.

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

In (Darwiche and Hopkins, 2001), empirical results show that HGR2BDT performs quite favorably versus one of the best known heuris-

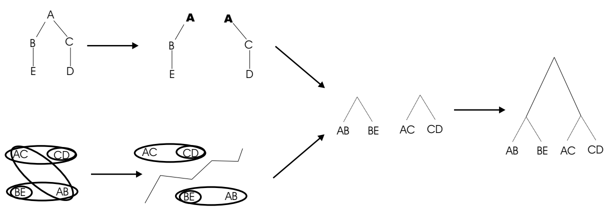


Figure 7: hgr2bdt from the perspective of vertex separators (above) and from the perspective of hypergraphs (below).

tic methods for graph triangulation, namely the min-fill heuristic for constructing elimination orders. Hence, in approaching this problem from the perspective of dtrees, we have managed to obtain one of the best known methods in practice for triangulating a graph. However, up until now, this method was lacking a solid theoretical basis. In the next section, we show how HGR2BDT can be viewed as a relaxed version of the theoretically sound but empirically impractical approximation algorithm detailed in Section 3.

6 Relationship to Approximation Algorithms

Consider the process of building a dtree for an undirected graph $G = (V, E)$ by finding a vertex separator S, choosing a 2-way edge partition (G_0, G_1) of G with respect to S, and constructing dtrees for G_0 and G_1.

After doing this, we then return a dtree node whose children are the two dtrees just constructed. Notice that this is exactly the process of HGR2BDT. Finding a vertex separator corresponds precisely to finding a hypergraph cut, whereas G_0 and G_1 are the two smaller hypergraphs that result after the hypergraph partition. In Figure 7, we show the HGR2BDT from the perspective of hypergraphs (as it is defined) and from the perspective of vertex separators (outlined above).

During this process, any given dtree node represents a subgraph $G' = (V', E')$ of the original graph and has a context $C \subseteq V'$ associated with it. Assume two things about the process of finding a vertex separator for this subgraph. Firstly, suppose that our algorithm always finds a vertex separator of size $k+1$ or less, where k is the treewidth of the original graph. Moreover, suppose that our algorithm always finds a vertex separator of size $k+1$ or less such that each of G_1 and G_2 contain at most 2/3 of the vertices of the context. Such a separator always exists – this is proven, as we noted in Section 3, in (Robertson and Seymour, 1995).

We can show that the preceeding algorithm is guaranteed to return a dtree whose width is at most $4k + 1$. Quite simply, suppose that a given node has context of size at most $3k + 1$. Thus, the context of any of its children contains at most 2/3 of the variables in its own context (i.e. 2/3 of $3k + 1 = 2k$), plus the variables in its cutset (which is at most $k + 1$). Since the context of the root node contains zero variables (and hence has context less than $3k + 1$), the context of any node in the dtree can have at most $3k + 1$ variables. Since we also know that the cutset of any node is at most $k + 1$, we can conclude that the maximum cluster size of any node is $4k + 2$ (hence width is at most $4k + 1$).

In fact, this algorithm described above is precisely the 4-factor approximation algorithm in (Robertson and Seymour, 1995), (Reed, 1992), and (Amir, 2001), when discussed in the language of dtrees. Notice that the set W in FACTOR4 can be naturally thought of as the context of the current dtree node, while the set X corresponds to the cutset. Similar algorithms proposed in (Becker and Geiger, 1996) and (Amir, 2001) can be analogously expressed. Thus, HGR2BDT can be viewed as a relaxed, heuristic version of these approximation algorithms, wherein we do not guarantee to find cutset width less than or equal to $k + 1$, nor guarantee that the context of each dtree node will have at most 2/3 of the context variables of its parent. Rather, HGR2BDT merely attempts to minimize cutset and balance the dtree (hence distributing the context variables), with no hard guarantees. Yet, as the experimental results show, this turns out to be a very effective method of dtree construction in practice.

7 Conclusion

Through this paper, we have achieved two important results. From the vantage point of the existing treewidth approximation algorithms, we have given an alternative formulation that provides a very clear and straightforward intuition of how the algorithms work. From the vantage point of our heuristic for constructing dtrees, we have established a solid theoretical foundation for what has proven to be an extremely practical and successful method of graph triangulation.

References

Eyal Amir. 2001. Efficient approximation for triangulation of minimum treewidth. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann.

Stefan Arnborg, Derek G. Corneil, and Andrej Proskurowski. 1987. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. and Discr. Meth.*, 8:277–284.

Ann Becker and Dan Geiger. 1996. A sufficiently fast algorithm for finding close to optimal junction trees. In *Proc. UAI '96*, pages 81–89. Morgan Kaufmann.

Hans L. Bodlaender, J.R. Gilbert, H. Hafsteinsson, and Ton Kloks. 1995. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *J. of Algorithms*, 18(2):238–255.

Hans. L. Bodlaender. 1993. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 226–234.

Adnan Darwiche and Mark Hopkins. 2001. Using recursive decomposition to construct elimination orders, jointrees, and dtrees. In *In Proceedings of the Sixth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU)*, pages 180–191.

Adnan Darwiche. 1999a. Compiling knowledge into decomposable negation normal form. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 284–289.

Adnan Darwiche. 1999b. Utilizing device behavior in structure–based diagnosis. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1096–1101.

Adnan Darwiche. 2001. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41.

Rina Dechter and Judea Pearl. 1989. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366.

Rina Dechter. 1996. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 211–219.

Mark Hopkins and Adnan Darwiche. 2002. Graph triangulation from the perspective of decomposition trees. Technical Report D–133, Computer Science Department, UCLA, Los Angeles, CA 90095.

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

George Karypis and Vipin Kumar. 1998. Hmetis: A hypergraph partitioning package. *Available at http://www.cs.umn.edu/ karypis.*

George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1998. Multilevel hypergraph partitioning: Applications in vlsi domain. *IEEE Transactions on VLSI Systems.*

Ton Kloks. 1994. *Treewidth.* Springer-Verlag, Lecture Notes in Computer Science.

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

Bruce A. Reed. 1992. Finding approximate separators and computing tree width quickly. In *Proc. 24th STOC*, pages 221–228. ACM Press.

N. Robertson and P.D. Seymour. 1995. Graph minors xiii: The disjoint paths problem. *J. Comb. Theory, Series B*, 63:65–110.

Kirill Shoikhet and Dan Geiger. 1997. A practical algorithm for finding optimal triangulations. In *Proc. AAAI '97*, pages 185–190. Morgan Kaufmann.