ELSEVIER

Artificial Intelligence 125 (2001) 3–17

Artificial Intelligence

www.elsevier.com/locate/artint

A sufficiently fast algorithm for finding close to optimal clique trees

Ann Becker, Dan Geiger [*]

Computer Science Department, Technion, Haifa 32000, Israel

Received 18 June 1996

Abstract

We offer an algorithm that finds a clique tree such that the size of the largest clique is at most $(2\alpha + 1)k$ where k is the size of the largest clique in a clique tree in which this size is minimized and α is the approximation ratio of an α-approximation algorithm for the 3-way vertex cut problem. When $\alpha = 4/3$, our algorithm's complexity is $O(2^{4.67k} n \cdot poly(n))$ and it errs by a factor of 3.67 where $poly(n)$ is the running time of linear programming. This algorithm is extended to find clique trees in which the state space of the largest clique is bounded. When $k = O(\log n)$, our algorithm yields a polynomial inference algorithm for Bayesian networks. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Clique tree algorithm; Triangulation algorithm; Bayesian networks; 3-way vertex cut problem

1. Introduction

All exact inference algorithms for the computation of a posterior probability in general Bayesian networks have two conceptual phases. One phase handles operations on the graphical structure itself and the other performs probabilistic computations; The clique tree algorithm [16] requires us to first find a "good" clique tree and then perform probabilistic computations on the clique tree and the method of conditioning [17,18] requires us to find a "good" loop cutset and then perform a calculation using the loop cutset. In [5], we offered an algorithm that finds a loop cutset for which the logarithm of the state space is guaranteed to be within a constant factor off the optimal value. In this paper, we provide a similar optimization for the clique tree algorithm.

[*] Corresponding author.
E-mail addresses: anyuta@cs.technion.ac.il (A. Becker), dang@cs.technion.ac.il (D. Geiger).

We shall first restrict our discussion to networks for which all vertices have the same state space size and to the optimality criterion which we call cliquewidth. The *cliquewidth* of an undirected graph G is the size of the largest clique in a clique tree of G in which the size of the largest clique is minimized. A more common term is *treewidth* which is the cliquewidth minus 1.

Preceding this work, all methods in the AI and Statistics communities for finding a clique tree had no guarantee of performance and could perform rather poorly when presented with an appropriate example. One algorithm, due to Rose [21], is as follows: repeatedly, select a vertex v with minimum number of neighbors $N(v)$, delete v from the graph, and make a clique out of $N(v)$. The resulting sequence of cliques creates a clique tree. This greedy algorithm minimizes the size of each clique as it is being created. However, it could easily make a mistake at the first step that would lead it to a clique tree far off the optimal size. Another algorithm, investigated by Kjærulff [13], uses a technique known as simulating annealing which takes a long time to run and has no guarantees on the quality of the output.

Finding an optimal clique tree is NP-complete but for a graph with n vertices and a cliquewidth k there exists an $\mathrm{O}(n^{k+1})$ algorithm that finds an optimal clique tree [2]. This algorithm is not practical for the size of Bayesian networks dealt with in practice. Other algorithms for finding an optimal clique tree have a complexity of $\mathrm{O}(f(k)n)$ where $f(k)$ is a super-exponential function of k [7]. These algorithms are practical for cliques of size $k = 5$ at most. A more practical algorithm for constructing an optimal clique tree when the largest clique size is 4 is given in [4]. For larger values of k there is no algorithm known to date that can find the optimum clique tree quickly. The exponential dependency in k cannot be improved unless P = NP because finding an optimal clique tree for $k = \mathrm{O}(n)$ is NP-complete [2].

Kloks in his book *Treewidth* [14], which is devoted to finding clique trees in various graphs, gives a *polynomial* algorithm that finds a clique tree of a given graph G such that its largest clique size is at most a factor $12\Delta \log n$ off optimal where Δ is a large unspecified constant (see also, [6]). Kloks states that finding a polynomial algorithm that constructs a clique tree such that the size of the largest clique is a constant factor off optimal is a major open problem. The importance of this problem stems from the fact that many NP-complete problems on graphs can be solved polynomially if the input graph has a clique tree with fixed sized cliques and if such a clique tree can be found efficiently [1,3]. Some of these problems are: INDEPENDENT SET, DOMINATING SET, GRAPH K-COLORABILITY, HAMILTONIAN CIRCUIT and CONSTRAINT SATISFACTION PROBLEMS [10].

Robertson and Seymour [20], among other key results, were the first to present an algorithm that finds a clique tree of a given graph G such that its largest clique size is at most a constant factor off optimal (they actually used a slightly different concept termed *branchwidth*). Reed [19] presents Robertson and Seymour's algorithm in a more accessible form and shows that its output is always less than 4 times the cliquewidth and the complexity is $\mathrm{O}(k^2 3^{3k} n^2)$. Reed also gives an algorithm that errs by a factor of 5 and has a complexity $\mathrm{O}(k^2 3^{4k} n \log n)$. Lagergren [15] presents efficient parallel algorithms for this problem.

We offer an algorithm that finds a clique tree such that the size of the largest clique is at most $(2\alpha + 1)$ times the cliquewidth where α is the approximation ratio for any approximation algorithm for the 3-way vertex cut problem. When using a $\frac{4}{3}$-approximation algorithm for the 3-way vertex cut problem ($\alpha = \frac{4}{3}$) due to [11], our algorithm's complexity is $\mathrm{O}(2^{4.67k} n \cdot \mathit{poly}(n))$ and it errs by a factor of 3.67 where $\mathit{poly}(n)$ is the running time of linear programming.

When $k = \mathrm{O}(\log n)$, our algorithm, like previous ones, is polynomial. Consequently, it yields a polynomial inference algorithm for the class of Bayesian networks that have a logarithmic cliquewidth. Of course, one does not know a priori what is the cliquewidth of a given network and so a user must terminate the algorithm if the running time is too long, in which case, due to the complexity results reported herein, the running time of exact inference must be quite large as well.

In Section 3, we describe the algorithm and prove its performance guarantee. This algorithm is made as simple as possible to facilitate the proof. In Section 4, we describe several heuristics that improve the algorithm's average case performance. In Section 5, we describe the changes needed so that the algorithm takes into account vertices with different state space sizes. The modified algorithm guarantees that the logarithm of the size of the state space of the heaviest clique in the clique tree found is less than a constant factor off the optimal value. In Section 6 we describe experiments made using the graph Medianus I. In most instances our algorithm was superior to an enhanced greedy algorithm both in terms of the largest state space and in terms of the total state space. In Section 7 we discuss the extent to which our results can be improved.

2. The clique tree algorithm

The clique tree algorithm is currently the most practical inference method for Bayesian networks. In this section we provide the relevant highlights of the clique tree algorithm. For details, consult [12,16].

Definition. A directed acyclic graph (DAG) is a graph with no directed cycles. In a DAG, $\mathit{pa}(v)$ denotes the set of parents of a vertex v. A *Bayesian network* is a DAG such that with each vertex v we associate a finite set $D(v)$ called the state space of v and a probability distribution $P(v|\mathit{pa}(v))$. The joint distribution of V is given by $P(V) = \prod_{v \in V} P(v|\mathit{pa}(v))$.

The *updating problem* is to compute the posterior probability of every vertex given specific values to a subset of vertices.

The clique tree algorithm solves the updating problem as follows. For every vertex v, it connects every pair of v's parents and removes the direction of all edges in the graph. The resulting graph is undirected (called the moral graph). Then, the moral graph is triangulated; edges are added until every cycle of length more than three has a chord. These are called *fill-in edges*. Once the graph is triangulated (or chordal), a tree of cliques, called the *clique tree*, is constructed. The clique tree algorithm then loads all probabilities into the clique tree and performs the calculations on the new structure.

Definition. Let $G = (V, E)$ be a chordal graph. A *clique tree* of G is a tree $\mathcal{H}$ such that each maximal clique C of G is a node in $\mathcal{H}$, and for every vertex v of G, if we remove from $\mathcal{H}$ all nodes not containing v, the remaining nodes stay connected.

The single most important step of this algorithm is triangulation. There are many ways to add edges to a given graph until it becomes chordal. In particular, one can simply make a single large clique. However, the time for loading the probabilities and performing the calculations is proportional to the total state space given by $\sum_{C\in\mathcal{H}}\prod_{v\in C}|D(v)|$, which is dominated by the size of the largest clique if all vertices have the same state space size. For example, if a largest clique contains m vertices and if their state space is of size two, then the probability table for this clique is of size 2^m. The objective of triangulation is to find a clique tree such that the largest clique size is as small as possible. Sections 3 and 4 provide an approximation algorithm for this problem. In Section 5, we describe the changes needed in order to account for varying state space sizes.

3. The triangulation algorithm

A natural approach to triangulate a graph $G=(V,E)$ is to use a divide and conquer technique. In each iteration a minimum set of vertices X is found which removal from G splits G into two disconnected components having vertex sets A and B such that $A\cup B\cup X=V$. The set X is called a *minimum vertex cut*. The algorithm proceeds on the two smaller problems $G[X\cup A]$ and $G[X\cup B]$, the subgraphs induced from G by the vertex sets $X\cup A$ and $X\cup B$ respectively. Each subgraph is triangulated such that X becomes a clique in it.

While this approach yields a triangulated graph, the size of the cliques produced may grow up to an $\mathrm{O}(n)$ factor off their initial size if in each step one of the graphs shrinks only by a constant number of vertices and the vertex cut found in each step has many edges connecting it to previously found cuts. [1] We leave out the construction of such an example. Robertson and Seymour, Reed, and Kloks all provide clever modifications that prevent the initial clique X from growing beyond a constant factor off its initial size [14,19,20].

We provide an algorithm that is similar to previous ones except that rather than dividing the graph to two subproblems using a min-cut/max-flow algorithm, we divide it to three subproblems using an approximation algorithm for the 3-way vertex cut problem.

The 3-way vertex cut problem is defined as follows: given a weighted undirected graph and three vertices, find a set of vertices of minimum weight whose removal leaves each of the three vertices disconnected from the other two. This problem is known to be NP-hard [8], however, there are some α-approximation algorithms for it, that is, algorithms which guarantee that the weight of their output is no more than α times the weight of an optimal 3-way vertex cut. In the next section, we describe a simple 2-approximation algorithm. A more sophisticated polynomial $\frac{4}{3}$-approximation algorithm for the 3-way vertex cut problem is reported in [11] (actually, their algorithm is a $(2-\frac{2}{k})$-approximation algorithm that finds k-way vertex cuts).

Our algorithm produces a triangulated graph whose largest clique size is less than $(2\alpha+1)k$ where k is the cliquewidth of G and α is the ratio between the weight of the 3-way vertex cut found by the algorithm we use and the optimal 3-way vertex cut. For

[1] We thank Leonid Zosin for constructing such an example.

$\alpha = \frac{4}{3}$, obtained by using Garg et al.'s algorithm, our approximation algorithm yields a triangulation having a cliquewidth bounded by $3.67k$.

Definition. Let $G = (V, E)$ be a graph. A *decomposition* of G is a partition (X, A, B, C) of V, where A and B are non-empty sets, such that there are no edges between A, B and C.

Definition. Given an integer $k \geqslant 1$, a real number $\alpha \geqslant 1$, a graph $G = (V, E)$ such that $|V| \geqslant (2\alpha + 1)k$, and a subset of vertices $W \subseteq V$, a decomposition (X, A, B, C) of G is called a *W-decomposition* with respect to (k, α) if $|W| < (\alpha + 1)k$, $|X| \leqslant \alpha k$, $|(W \cap A) \cup X| < (\alpha + 1)k$, $|(W \cap B) \cup X| < (\alpha + 1)k$, and $|(W \cap C) \cup X| < (\alpha + 1)k$.

For example, suppose G is the chain $a - b - c - d - e$. The triplet $X = \{c\}$, $A = \{a, b\}$, $B = \{d, e\}$ and $C = \emptyset$ is a decomposition of G. Given $W = \{b, d\}$, this decomposition is a W-decomposition of G with respect to $k = 1$ and $\alpha = 2$. Given $W = \{b, c\}$, the triplet $X = \{d\}$, $A = \{a, b, c\}$, $B = \{e\}$ and $C = \emptyset$ is not a W-decomposition of G with respect to $(k = 1, \alpha = 2)$ because $|(W \cap A) \cup X| = 3$.

The triangulation algorithm is given in Fig. 1. In order to triangulate a graph G having a cliquewidth k we call *Triangulate*$(G, \emptyset, k)$. When the algorithm is called with $W = \emptyset$, the size of the second argument of *Triangulate* in each recursive call is (strictly) less than $(\alpha + 1)k$ because, when $|V| \geqslant (2\alpha + 1)k$, by definition of W-decompositions, the sets $W_A \cup X$, $W_B \cup X$, $W_C \cup X$ which are the arguments passed in the recursive calls, contain less than $(\alpha + 1)k$ vertices, respectively. Fig. 2 shows a graph and how it splits into three subgraphs in a recursive call of *Triangulate*. The set W serves to monitor the size of the largest clique in the subproblems in each recursive call.

```
ALGORITHM Triangulate($G, W, k$)

Input: An undirected graph $G(V, E)$, $W \subseteq V$, $k$.
Output: A triangulation of $G$ such that $W$
    is made a clique and such that the size
    of the largest clique $< (2\alpha + 1)k$ (Success)
    or, a valid statement that the cliquewidth
    of $G$ is larger than $k$ (Failure).

If $|V| < (2\alpha + 1)k$ then make a clique out of $G$
    else
    Find a $W$-decomposition
                $(X, A, B, C)$ of $G$ with respect to $(k, \alpha)$;
    If not found return "cliquewidth $> k$"
    $W_A \leftarrow W \cap A$, $W_B \leftarrow W \cap B$, $W_C \leftarrow W \cap C$;
    call Triangulate($G[A \cup X]$, $W_A \cup X$, $k$);
    call Triangulate($G[B \cup X]$, $W_B \cup X$, $k$);
    call Triangulate($G[C \cup X]$, $W_C \cup X$, $k$);
    make a clique of $G[W \cup X]$.
```

Fig. 1. The triangulation algorithm.

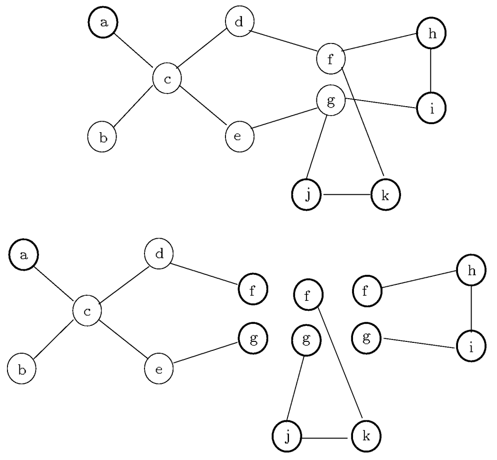

Fig. 2. An example of one level of a recursive call with $k = 3$ and $\alpha = 1$. Highlighted vertices are in W and $X = \{f, g\}$. The three graphs at the bottom are passed as arguments to the next level of the recursion.

The next two lemmas show that a W-decomposition must exist or the cliquewidth is greater than k, in which case the algorithm outputs correctly this fact. Consequently, a naive way to use this algorithm is to repeatedly call *Triangulate*$(G, \emptyset, k)$ starting with $k = 1$ and incrementing k by 1 whenever the algorithm fails to triangulate G. In the next section, we provide implementation details and a complexity analysis.

Lemma 1. *Given a graph $G(V, E)$ with a cliquewidth $\leqslant k$, $|V| \geqslant k + 2$, and a subset of vertices W, $|W| > 1$, there exists a decomposition (X, A, B, C) of G such that $|X| \leqslant k$, $|W \cap A| \leqslant \frac{1}{2}|W|$, $|W \cap B| \leqslant \frac{1}{2}|W|$ and $|W \cap C| \leqslant \frac{1}{2}|W|$.*

Proof. A constructive proof of similar claims is given in [14, Lemma 2.2.9]. Let $H(G)$ be a clique tree of G with a cliquewidth $\leqslant k$. Add edges until all cliques in this clique tree become of size k and all intersections of cliques become of size $k - 1$. Call the resulting clique tree $T(G)$. Now consider the following algorithm. Start with any clique X in $T(G)$. If there is no connected component in $G[V \setminus X]$ which has more than $\frac{1}{2}|W|$ vertices of W, then stop. Otherwise, let S be a component in $G[V \setminus X]$ which has more than $\frac{1}{2}|W|$ vertices of W. There exists a vertex y in S which has $k - 1$ neighbors in X in the graph $T(G)$ (viewed as a chordal graph). Let x be the vertex in X that is not a neighbor of y in $T(G)$. Define $Y = X \setminus \{x\} \cup \{y\}$. Note that Y is a clique and it has k vertices. The algorithm continues with Y.

To show that this algorithm terminates, we prove that in each step of the algorithm one of two conditions is met. Hereafter, the component which includes the largest part of W will be called the *main* component. The first condition is that the number of vertices in the main component decreases and the number of vertices of W in the main component does not increase. The second condition is that the number of vertices of W in the main component decreases.

Notice that $G[V \setminus Y]$ has two types of components. One type consist only of vertices in $S \setminus \{y\}$. If the main component of $G[V \setminus Y]$ is among these, the number of vertices is decreased and the number of vertices of W does not increase. The other type of components consist only of vertices of $\{x\} \cup V \setminus (S \cup X)$. The total number of vertices of W in this set is less than $\frac{1}{2}|W|$ because S contains more than half the vertices of W. Hence, in this case, the number of vertices from W in the main component decreases by one. Consequently, the algorithm terminates.

Suppose now that X is the final clique considered by this algorithm. If $G[V \setminus X]$ has two or more non empty components, then group them into three sets to form the desired decomposition. Otherwise, there is only one component in $G[V \setminus X]$. Consequently, the clique X is a leaf in the clique tree $T(G)$. Since V contains at least $k+2$ vertices, and there is only one component in $G[V \setminus X]$, there exists a unique clique Y that contains $k-1$ vertices of X and which is not a leaf in $T(G)$. The graph $G[V \setminus Y]$ has at least two connected components and each contains at most half the vertices of W (because $|W| > 1$). $\square$

Lemma 2. *Given an integer $k \geqslant 1$, a real number $\alpha \geqslant 1$, a graph $G(V, E)$ with $|V| \geqslant (2\alpha + 1)k$ and a subset of vertices W such that $|W| < (\alpha + 1)k$, there exists a W-decomposition (X, A, B, C) of G with respect to (k, α) or the cliquewidth of G is larger than k.*

Proof. Let G be a graph with a cliquewidth $\leqslant k$. If $|W| \leqslant 1$, then let X be any minimal vertex cut. If $|X| \leqslant k$, the resulting decomposition is a W-decomposition with respect to (k, α). Otherwise, the cliquewidth is larger than k.

Suppose $|W| > 1$. Let (X, A, B, C) be a decomposition of G with the properties guaranteed by Lemma 1. We will prove that (X, A, B, C) is also a W-decomposition with respect to (k, α). If it were not, then assume that $|(W \cap A) \cup X| \geqslant (\alpha + 1)k$, this inequality implies that $|W \cap A| \geqslant \alpha k$ because $|X| \leqslant k$. But according to Lemma 1, we have $|W| \geqslant 2|W \cap A|$. Consequently, $|W| \geqslant 2\alpha k$ in contradiction to its given size which is smaller than $(\alpha + 1)k$. Hence, if the cliquewidth of $G \leqslant k$, then there is a W-decomposition with respect to (k, α). Equivalently, if there isn't a W-decomposition with respect to (k, α), the cliquewidth must be larger than k. $\square$

Theorem 3. *If $G(V, E)$ is a graph with n vertices, $k \geqslant 1$ an integer, $\alpha \geqslant 1$ a real number, and W is a subset of V such that $|W| < (\alpha + 1)k$, then Triangulate(G, W, k) triangulates G such that the vertices of W form a clique and such that the size of a largest clique of the triangulated graph $< (2\alpha + 1)k$ or the algorithm correctly outputs that the cliquewidth of G is larger than k.*

Proof. If the algorithm outputs that the cliquewidth of G is larger than k, then this is a valid statement by Lemma 2. Assume the algorithm does not produce this output.

The algorithm always terminates because in every recursive call of *Triangulate* the graphs $G[A \cup X]$, $G[B \cup X]$ and $G[C \cup X]$ have less vertices than $G[A \cup B \cup C \cup X]$ since A and B are not empty.

Next, we show that the algorithm returns a triangulated graph. We prove this by induction using the recursive structure of the algorithm. Clearly the claim is true if $|V| <$

$(2\alpha+1)k$. Assume $|V| \geqslant (2\alpha+1)k$. By induction the recursive call *Triangulate*$(G[A\cup X]$, $W_A \cup X$, $k)$ returns a triangulation of $G[A \cup X]$, such that $W_A \cup X$ is a clique. Similarly, for B and C. The algorithm then makes a clique of $W \cup X$. Consequently, the graphs $G[A \cup W \cup X]$, $G[B \cup W \cup X]$ and $G[C \cup W \cup X]$ are triangulated as well. Since the intersection of these triangulated graphs is a clique, the union must also be triangulated.

It remains to show that the cliquewidth of the triangulated graph is less than $(2\alpha + 1)k$. This is clearly true if $|V| < (2\alpha + 1)k$. Hence assume $|V| \geqslant (2\alpha + 1)k$. Let M be a largest clique in the triangulated graph. There are two cases to consider. If M contains no vertex of $A \setminus W$, $B \setminus W$ and $C \setminus W$, then M contains only vertices of $W \cup X$. Consequently, $|M| = |W \cup X| \leqslant |W| + |X| < (\alpha + 1)k + \alpha k$, and the cliquewidth is less than $(2\alpha + 1)k$ as claimed. If M contains a vertex of $A \setminus W$, then it contains no vertex of $B \cup C$ because there are no edges between $A \setminus W$ and $B \cup C$. Hence M is a clique in the triangulation of $G[A \cup X]$. By induction we know that $|M| < (2\alpha + 1)k$. □

Note that Lemma 2 and Theorem 3 hold for every $\alpha \geqslant 1$. However, in order to find a W-decomposition with respect to (k,α) sufficiently fast (Lemma 2 only guarantees existence), we choose α to be the approximation factor of an algorithm for the 3-way vertex cut problem, an algorithm which we employ for finding a W-decomposition. We now give an algorithm that finds a W-decomposition with respect to (k,α) where α is chosen as just described. Then we will argue for correctness.

For every possible selection of four disjoint subsets W_A, W_B, W_C, W_X of W, such that $|W_A| \geqslant |W_B| \geqslant |W_C|$, we show how to check if there exists a W-decomposition (X, A, B, C) with respect to (k,α), such that $W_A \subseteq A$, $W_B \subseteq B$, $W_C \subseteq C$ and $W_X \subseteq X$. There are at most $4^{|W|}$ choices to divide W into four set of vertices W_A, W_B, W_C, W_X.

Let W_A, W_B, W_C, W_X be a particular selection. We consider two cases,

(1) $|W_A| < k$, and

(2) $|W_A| \geqslant k$.

Each case uses a different procedure.

Procedure I ($|W_A| < k$): Remove W_X from the graph, add three dummy vertices v_a, v_b and v_c each connected to all the vertices in W_A, W_B and W_C, respectively. Set the capacity of all vertices in $W_A \cup W_B \cup W_C$ to infinity and the capacity of all other vertices to one. Find an α-approximation 3-way vertex cut Y which splits v_a, v_b and v_c into three disconnected components. If Y has a finite weight, then, due to the capacities selected, it must split W_A,W_B and W_C to three disconnected components as well. Otherwise drop this choice from further consideration. Now let X be $Y \cup W_X$, A be the union of the connected components of $G[V \setminus X]$ such that $W_A \subseteq A$, B be the union of the connected components of $G[V \setminus X]$ such that $W_B \subseteq B$, and $C = V \setminus (A \cup B \cup X)$. If $|X| < (\alpha + 1)k - |W_A|$ and $|X| \leqslant \alpha k$ then output (X, A, B, C) as the desired W-decomposition with respect to (k,α) (because $|W_A| \geqslant |W_B| \geqslant |W_C|$).

Procedure II ($|W_A| \geqslant k$): Remove W_X from the graph, add a dummy vertex v_a that is connected to all the vertices in W_A, and add another dummy vertex v_{bc} that is connected to all vertices in W_B and W_C. Set the capacity of all vertices in $W_A \cup W_B \cup W_C$ to infinity and the capacity of all other vertices to one. Find a minimum vertex cut Y which splits v_a and v_{bc} into two disconnected components. If Y has a finite weight, then, it must split W_A and $W_B \cup W_C$ as well. Otherwise drop this choice from further consideration.

Finding a minimum vertex cut is done by any of the well known max-flow/min-cut algorithms. Now let X be $Y \cup W_X$, A be the union of the connected components of $G[V \setminus X]$ such that $W_A \subseteq A$, $B = V \setminus (A \cup X)$, and $C = \emptyset$. If $|X| < (\alpha+1)k - |W_A|$, $|X| < (\alpha+1)k - |W_B \cup W_C|$ and $|X| \leqslant \alpha k$ then output (X, A, B, C) as the desired W-decomposition with respect to (k, α).

Now we will show that if a W-decomposition with respect to (k,α) exists, as guaranteed by Lemma 2, then either procedure I or procedure II will find a W-decomposition with respect to (k,α) for some choice of W_A, W_B, W_C, W_X. Let (X', A', B', C') be a decomposition of G with the properties guaranteed by Lemma 1. Let $W_A = A' \cap W$, $W_B = B' \cap W$,$W_C = C' \cap W$ and $W_X = X' \cap W$, and assume without loss of generality that $|W_A| \geqslant |W_B| \geqslant |W_C|$. Procedure I for $|W_A| < k$, and procedure II for $|W_A| \geqslant k$ both generate for this choice of W_A, W_B, W_C, W_X, a decomposition (X, A, B, C). We now show that in either case this decomposition is a W-decomposition with respect to (k, α).

Case 1: $|W_A| < k$. The set of vertices $X' \setminus W_X$ is a 3-way vertex cut for the sets W_A, W_B, and W_C in the graph $G[V \setminus W_X]$. An α-approximation algorithm for the 3-way vertex cut problem outputs a cut Y, such that $|Y| \leqslant \alpha|X' \setminus W_X|$. Since $\alpha \geqslant 1$, we get $|Y \cup W_X| \leqslant \alpha|X'|$. Consequently, $|Y \cup W_X| \leqslant \alpha k$ because $|X'| \leqslant k$. Finally, $|W_A \cup (Y \cup W_X)| < k + \alpha k = (\alpha+1)k$. Thus all the conditions for a W-decomposition with respect to (k,α) are met.

Case 2: $|W_A| \geqslant k$. The set of vertices $X' \setminus W_X$ is a vertex cut for the sets W_A, $W_B \cup W_C$ in $G[V \setminus W_X]$. A max-flow/min-cut algorithm outputs a vertex cut Y such that $|Y| \leqslant |X' \setminus W_X|$. Consequently $|Y \cup W_X| \leqslant k$ because $|X'| \leqslant k$. Finally, since $|W| \geqslant 2|W_A|$ (Lemma 1), we get $|W_A \cup (Y \cup W_X)| < ((\alpha+1)/2)k + k \leqslant (\alpha+1)k$. Hence from $|W_B \cup W_C)| < \alpha k$ it follows that $|(W_B \cup W_C) \cup (Y \cup W_X)| < \alpha k + k = (\alpha+1)k$. Thus all the conditions for a W-decomposition with respect to (k,α) are met.

4. Implementation and complexity

The algorithm presented in the previous section can be improved substantially by three adjustments: processing the input of the algorithm, changing the termination condition of the recursion, and processing the output of the algorithm. We shall first describe these changes and demonstrate the algorithm on a simple example. Then, we provide further implementation details and analyze the algorithm's complexity.

The input graph of the algorithm may contain vertices such that all their neighbors are connected. A vertex v is called *simplicial* in G if its neighbors $N(v)$ form a clique. Before calling *Triangulate*, starting with the input graph G, we repeatedly remove every simplicial vertex from the current graph yielding a graph G'. The graph G' has a cliquewidth no larger than that of G. By triangulating G' by *Triangulate* (or any other triangulation algorithm) we triangulate G as well. Hence, this preprocessing step retains the validity of the algorithm. This step improves the running time complexity whenever simplicial vertices are found.

The termination condition of the recursion is that whenever $|V| < (2\alpha+1)k$ a clique is formed out of G. However, instead of a clique, it suffices to produce a clique tree of G in which W is a clique. This step is done by forming a clique of W and then completing it to a clique tree of G by any of the known greedy heuristics. The proof of Theorem 3 remains valid without any change. Consequently, the worst case approximation is not affected. However, in many instances the approximation is improved.

The output of the algorithm is a triangulated graph $T(G)$ which is not necessarily minimal. This means that some edges that were added (fill-in edges) might possibly be removed and the resulting graph remains triangulated. Kjærulff provides an algorithm that, given a triangulation of a graph G and an ordering on its vertices, produces a minimal triangulated graph [13]. We use Kjærulff's algorithm with an ordering that is determined as follows. First in the ordering are the simplicial vertices in the order in which they are removed from G. The order of the remaining vertices is determined recursively while running *Triangulate*; In each level of the recursion, the vertices in $X \setminus W$ follow those in $A \setminus W$, those in $B \setminus W$ and those in $C \setminus W$.

We now demonstrate the effects of these modifications on the graph depicted in Fig. 2 (assuming $W = \emptyset$). If simplicial vertices are removed, then the remaining graph does not contain the vertices a and b. The next phase, when $k = 3$ and $\alpha = 1$, creates three cliques: $\{c, d, e, f, g\}$, $\{f, g, h, i\}$ and $\{f, g, j, k\}$, in addition to $\{a, c\}$ and $\{b, c\}$ due to the simplicial vertices. Finally, applying Kjærulff's minimization algorithm removes the edges (f, i), (f, j), (c, f), (c, g), (d, g) yielding an optimal clique tree.

The total complexity of running *Triangulate* with a given k is the time it takes to find a W-decomposition times the number of nodes in the recursion tree which is at most n. The time it takes to verify whether a choice W_A, W_B, W_C, W_X can generate a W-decomposition with respect to $(k, \alpha = \frac{4}{3})$ takes *poly*(n) which is the time it takes to run Garg et al's $\frac{4}{3}$-approximation algorithm for the 3-way vertex cut problem. This polynom is quite high as it is the complexity linear programming. (A more practical algorithm, without a polynomial complexity guarantee, is the simplex algorithm.) Thus the complexity of running *Triangulate* with a given k is $O(4^{(1+\alpha)k} n \cdot poly(n))$ where $\alpha = \frac{4}{3}$, because there are at most $4^{|W|}$ choices and $|W| < (\alpha + 1)k$. Since, in the worst case, the algorithm is run for $i = 1$ up to the cliquewidth of G, the total running time is $O(\sum_{i=1}^{k} 4^{2.33i} n \cdot poly(n))$ which is $O(2^{4.67k} n \cdot poly(n))$. The size of the largest clique in the output is at most $2\alpha + 1 = 3.67$ times the cliquewidth.

In a simpler implementation we use a straightforward 2-approximation algorithm for finding a 3-way vertex cut; Find a minimum vertex cut between v_a and $\{v_b, v_c\}$, a minimum vertex cut between v_b and $\{v_a, v_c\}$ and a minimum vertex cut between v_c and $\{v_a, v_b\}$. The output vertex cut is the union of any two of the three vertex cuts. This output is clearly a 3-way cut and it is at most twice the optimal weight because each of the three cuts weighs less than an optimal 3-way vertex cut. Finding each vertex cut is done using a max-flow/min-cut algorithm which takes $O(kn^2 \log n)$. This algorithm for the 3-way vertex cut is analogous to the one described in [9] for the edge multiway cut. A more clever implementation using Reed's arguments can find an appropriate vertex cut in $O(k^2 n)$. Consequently, since $\alpha = 2$, the total complexity is $O(k^2 4^{3k} n^2)$ and the largest clique in the output is at most 5 times the cliquewidth.

In practice, our implemented algorithm, which uses the 2-approximation 3-way vertex cut, encountered a much smaller complexity. The set W is almost always less than $(1 + \alpha)k$ and in most cases it is less than k which implies that the complexity encountered is proportional to 2^{2k} rather than to $2^{4.67k}$. Furthermore, when a W-decomposition (X, A, B, C) exists, it is often the case that W consists of two subsets and the third is empty, in which case the algorithm for finding a 3-way vertex cut is not activated (as is the case in the graph of Section 6). In addition, instead of increasing k by one whenever *Triangulate* fails on the input k, we can increase it to the minimal value k^* for which a decomposition that was tested with respect to (k, α) was found to be a W-decomposition with respect to (k^*, α) $(k^* > k)$.

Finally, note that Theorem 3 provides only a worst case bound of $2\alpha + 1$ for the ratio between the size of the largest clique and the cliquewidth of the given graph. However, if for an integer k, *Triangulate* produces a triangulation having a largest clique of size l and the algorithm fails for $k - 1$ (it is run for $i = 1, \ldots, k$ until it succeeds), then the ratio l/k is an upper bound for the ratio between the output and the cliquewidth of G because the cliquewidth must be greater than $k - 1$. This bound is much tighter than $2\alpha + 1$ because it takes into account the given graph and the specific steps made by *Triangulate*. It is an instance-specific posteriori bound rather than a worst case a priori bound. The bound l/k is produced by the algorithm in order to inform the user about the quality of the clique tree found.

5. The weighted problem

It remains to describe the changes needed in order to account for different state spaces of each vertex. The weight $w(v)$ of a vertex v is the logarithm (base 2) of its state space size and the weight of a clique is the sum of the weights of its constituent vertices. Note that the weight of a vertex with a binary state space is 1 and the weight of other vertices is larger than 1. Our optimality criterion is now the weighted cliquewidth of G. The *weighted cliquewidth* of G is the weight of the heaviest clique in a clique tree of G in which the weight of the heaviest clique is minimized.

To minimize the heaviest clique, we modify the algorithm as follows. We find a *weighted W-decomposition* with respect to (m, α), namely, a decomposition (X, A, B, C) of $G = (V, E)$, where $w(V) \geqslant (2\alpha + 1)m$, such that the following holds: $w(W) < (\alpha + 1)m$, $w(X) \leqslant \alpha m$, $w((W \cap A) \cup X) < (\alpha + 1)m$, $w((W \cap B) \cup X) < (\alpha + 1)m$ and $w((W \cap C) \cup X) < (\alpha + 1)m$. As in the unweighted case, we recursively triangulate the graphs $G[A \cup X]$, $G[B \cup X]$, and $G[C \cup X]$ and make a clique out of $W \cup X$. Once the termination condition is met, namely, $w(V) < (2\alpha + 1)m$, we apply the following greedy algorithm which is called the *minimum weight heuristics*: repeatedly, select a vertex v which forms with its neighbors $N(v)$ a set of minimum weight, remove it from the current graph, and make $N(v)$ a clique. We call this modified algorithm *W-Triangulate*.

The following claim holds.

Theorem 4. *If G is a graph with n vertices, m and $\alpha \geqslant 1$ are real numbers, and W is a subset of V such that $w(W) < (\alpha + 1)m$, then W-Triangulate(G, W, m) triangulates G such that the vertices of W form a clique and such that the weight of a heaviest clique of the triangulated graph $< (2\alpha + 1)m$ or the algorithm correctly outputs that the weighted cliquewidth of G is larger than m.*

Proof. Theorem 3 and Lemmas 1 and 2 remain valid when the cardinality of sets is replaced with their weight and k is replaced with m. □

Theorem 4 states that in the clique tree found by *W-Triangulate* the weight of the heaviest clique is less than $2\alpha + 1$ times the weighted cliquewidth.

The complexity of *W-Triangulate* depends on the maximum size of W throughout the recursive calls which we denote by s. The complexity of *W-Triangulate* is $O(k^2 4^{3s} n^2)$ when the 2-approximation algorithm for the 3-way vertex cut problem is used. The heaviest clique in the resulting clique tree is at most 5 times the weighted cliquewidth. Since, $k \leqslant s \leqslant \min\{m, n\}$, this complexity is comparable to the complexity of inference on the resulting clique tree which is $O(2^{5m} n)$ and it is smaller than the complexity of inference if state spaces are sufficiently large.

6. Experimental results

Kjærulff has tested several heuristic algorithms for constructing clique trees for two graphs that were used for a medical application: Medianus I and Medianus II [13]. His experiments show that the minimum weight heuristics enhanced by removing redundant fill-in edges is superior to all other heuristics that were considered. We will compare *W-Triangulate* with this enhanced minimum weight heuristics on Medianus I (see Fig. 4). We use two optimality criteria for the comparison, the logarithm of the state space size of the heaviest clique denoted by M and the logarithm of the sum of the state space sizes of all cliques (the logarithm of the total state space) denoted by T. Criterion M is the one that served to develop *W-Triangulate* and T is the one that optimizes the construction of the probability tables for the resulting clique tree.

The two algorithms were run on Medianus I with state sizes randomly selected from the range 3 to 21 with an average of approximately 6 (as in [13]). One hundred random runs were made. In 68 runs our algorithm has outperformed the enhanced minimum weight heuristics in both optimality criteria. Fig. 3 shows that the averaged improvement of T was 0.64 and the maximum improvement was 2.37 which amounts to a reduction of storage by a factor of about 5. In the 16 instances in which the greedy method was more successful, the difference was at most 1.22. Of course, to obtain the best results one can simply run both algorithms.

When the state space size of each vertex was selected between 6 and 32 with an average of 13, we found two graphs in which T is approximately 30 using *W-Triangulate*

	W-Triangulate			Eq	Greedy		
	#	Δ_{ave}	Δ_{max}	#	#	Δ_{ave}	Δ_{max}
M	75	.62	2.35	16	9	.3	.93
T	73	.64	2.37	11	16	.42	1.22

Fig. 3. The results for 100 runs on Medianus I. The first line records the differences on the average and in the extreme case of the logarithm of the heaviest clique. In 16 instances the algorithms produced equal output. The second line records the same information regarding the logarithm of the total state space.

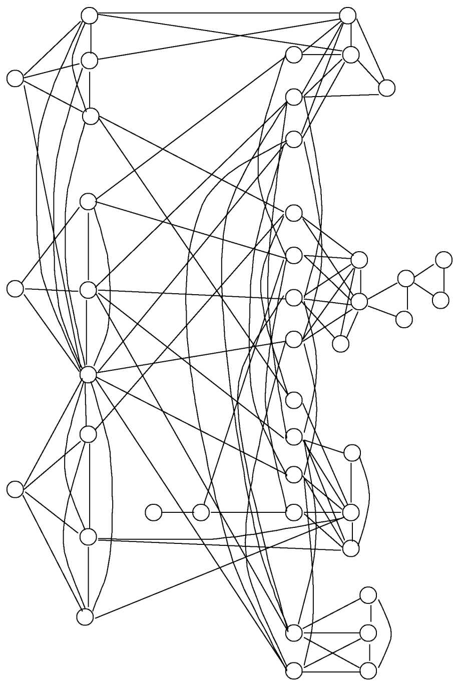

Fig. 4. The Medianus I graph.

and T is approximately 34.5 using the enhanced greedy algorithm which implies that instead of 1GB of memory which we need for storing the conditional probabilities, the greedy algorithm would have used over 20GB. In general, as the state spaces increase our algorithm becomes far better than the enhanced minimum weight heuristics.

Recall that the algorithm W-*Triangulate*$(G, \emptyset, k)$ is run with increasing values of k until a triangulation is found. We have recorded the number of vertices l in the largest clique (in size) of the clique tree found by W-*Triangulate*$(G, \emptyset, k)$ when it succeeded and compared it to the value of k. Let $\Delta = l - k$. The largest clique size found is of size l while the cliquewidth is larger than $k - 1$ (because the algorithm failed with $k - 1$ as an input). Then, Δ was 0 in 6 graphs (provably optimum size), 1 in 14 graphs (at most one vertex off optimum), 2 in 29 graphs, 3 in 48 graphs and 4 in 3 graphs. The worst case upper bound on the ratio between the size of the largest clique and the unknown cliquewidth was $l/k = 10/6$ rather than 3.67 which is guaranteed in theory. Indeed, one cannot hope to improve the clique tree too much on this graph.

We also collected some statistics on the running time complexity. We counted the number of partitions made each time a W-decomposition is constructed. The count for Medianus I was always far less than 4^k rather than 4^{3k} which is the worst case bound. The recursion depth was at most 3. The algorithm runs in less than a minute for most graph instances but occasionally it takes up to two minutes. On these examples Robertson and Seymour's algorithm runs faster and obtains identical results. Our algorithm is faster when k is large and n is small.

7. Discussion

We presented an algorithms that finds a clique tree in which the largest clique is no more than 3.67 times the cliquewidth. If the cliquewidth of G is of size $k = \mathrm{O}(\log n)$, then our approximation algorithm is polynomial since its complexity is $\mathrm{O}(2^{4.67k} n \cdot \mathit{poly}(n))$ where $\mathit{poly}(n)$ is the complexity of linear programming. It is well known that inference in an optimal clique tree with binary variables takes $\mathrm{O}(2^k n)$ which is polynomial for a logarithmic cliquewidth. Thus, inference done using the clique tree produced by our algorithm, as well as by Robertson and Seymour's algorithm, is guaranteed to be polynomial as well because if we err at most by a constant factor, the time of inference is at most the optimal time raised to some power and so inference stays polynomial. Note that the heuristic constructions of clique trees which do not guarantee a constant error bound are not polynomial.

Our results could possibly be improved by constructing an algorithm that finds an optimal clique tree with respect to the weighted cliquewidth with a complexity of optimal inference, i.e., $\mathrm{O}(2^k n)$, or errs by a factor smaller than 3.67. Our current algorithm, however, can yield at best an error factor of 3 if an efficient exact algorithm is found for the 3-way vertex cut problem for graphs with bounded cliquewidth. The existence of such an algorithm is hinted parenthetically in [9] but the dependency in k is possibly super exponential.

As a final comment, let us shed light on the common utterance used by the AI community, that "inference is easy in sparse graphs". Recall that if the cliquewidth is of size k, then the graph has no more than kn edges (see, e.g., Section 4). Hence, sparse graphs in the context of easy inference should mean that the cliquewidth is of size $\mathrm{O}(\log n)$, which allows a polynomial inference algorithm, and implies that there are no more than $\mathrm{O}(n \log n)$ edges in the graph.

Acknowledgements

We thank Seffi Naor for pointing us to the term treewidth, and for pointing us to [11]. We thank Leonid Zosin for showing us examples of poor clique trees produced by various greedy algorithms. We thank Linda van der Gaag and Hans Bodlaender for their helpful comments and for the references they provided us, in particular, Reed's paper. The conference version of this paper can be found in the *Proceedings of the 12th conference on Uncertainty in Artificial Intelligence*. This research was supported by the fund for promotion of research at the technion.

References

[1] S. Arnborg, Efficient algorithms for combinatorial problems on graphs with bounded decomposability, BIT 25 (1985) 2–23.

[2] S. Arnborg, D.G. Corneil, A. Proskurowski, Complexity of finding embeddings in a k-tree, SIAM J. Algebraic and Discrete Methods 8 (1987) 277–284.

[3] S. Arnborg, J. Lagergren, D. Seese, Easy problems for tree-decomposable graphs, J. Algorithms 12 (1991) 308–340.
[4] S. Arnborg, A. Proskurowski, Characterization and recognition of partial 3-trees, SIAM J. Algebraic and Discrete Methods 7 (1986) 305–314.
[5] A. Becker, D. Geiger, Optimization of Pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem, Artificial Intelligence 82 (1996) 1–22.
[6] H.L. Bodlaender, J.R. Gilbert, H. Hafsteinsson, T. Kloks, Approximating treewidth, pathwidth, and minimum elimination tree height, Graph-theoretic concepts in computer science, in: Proc. 17th International Workshop, WG'91, Fischbachau, Germany, Lecture Notes in Computer Science, Vol. 570, Springer, Berlin, 1991; See also: J. Algorithms 18 (1995) 238–255.
[7] H.L. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth, in: Proc. 25th ACM STOC, 1993, pp. 226–234.
[8] W.H. Cunningham, The optimal multiterminal cut problem, Discrete Mathematics and Theoretical Computer Science (DIMACS Series) 5 (1991) 105–120.
[9] E. Dahlhaus, D.S. Johnson, C.H. Papadimitriou, P.D. Seymour, M. Yannakakis, The complexity of multiway cuts, in: Proc. 24th Annual ACM STOC, 1994, pp. 241–251.
[10] R. Dechter, J. Pearl, Tree clustering for constraint networks, Artificial Intelligence 38 (1989) 353–366.
[11] N. Garg, V.V. Vazirani, M. Yannakakis, Multiway cuts in directed and node weighted graphs, in: Proc. Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, Lecture Notes in Computer Science, Vol. 820, Springer, Berlin, 1994, pp. 487–498.
[12] F.V. Jensen, S.L. Lauritzen, K.G. Olesen, Bayesian updating in causal probabilistic networks by local computations, Computational Statistics Quarterly 4 (1990) 269–282.
[13] U. Kjærulff, Triangulation of graph-algorithms giving small total state space, Technical Report R 90-09, Department of Mathematics and Computer Science, Aalborg University, Denmark, 1990.
[14] T. Kloks, Treewidth, Lecture Notes in Computer Science, Vol. 842, Springer, Berlin, 1994.
[15] J. Lagergren, Efficient parallel algorithms for graphs of bounded treewidth, J. Algorithms 20 (1996) 20–44.
[16] S.L. Lauritzen, D.J. Spiegelhalter, Local computations with probabilities on graphical structures and their application to expert systems (with discussion), J. Roy. Statist. Soc. B 50 (2) (1988) 157–224.
[17] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufmann, San Mateo, CA, 1988.
[18] J. Pearl, Fusion, propagation and structuring in belief networks, Artificial Intelligence 29 (3) (1986) 241–288.
[19] B. Reed, Finding approximate separators and computing treewidth quickly, in: Proc. 24th Annual Symposium on Theory of Computing, ACM Press, NY, 1992, pp. 221–228.
[20] N. Robertson, P.D. Seymour, Graph minors XIII. The disjoint paths problem, J. Combinatorial Theory B 63 (1995) 65–110.
[21] D. Rose, Triangulated graphs and the elimination process, J. Math. Anal. Appl. 32 (1974) 597–609.