International Journal of Approximate Reasoning 53 (2012) 1350–1366

Contents lists available at SciVerse ScienceDirect

International Journal of Approximate Reasoning

journal homepage: www.elsevier.com/locate/ijar

All roads lead to Rome—New search methods for the optimal triangulation problem[☆]

Thorsten J. Ottosen[a], Jiří Vomlel[b,*]

[a] Department of Computer Science, Aalborg University, DK-9220 Aalborg, Denmark
[b] Institute of Information Theory and Automation of the AS CR, 182 08 Prague, The Czech Republic

ARTICLE INFO

Article history:
Available online 28 June 2012

Keywords:
Bayesian networks
Optimal triangulation
Probabilistic inference
Cliques in a graph

ABSTRACT

To perform efficient inference in Bayesian networks by means of a Junction Tree method, the network graph needs to be triangulated. The quality of this triangulation largely determines the efficiency of the subsequent inference, but the triangulation problem is unfortunately NP-hard. It is common for existing methods to use the treewidth criterion for optimality of a triangulation. However, this criterion may lead to a somewhat harder inference problem than the total table size criterion. We therefore investigate new methods for depth-first search and best-first search for finding optimal total table size triangulations. The search methods are made faster by efficient dynamic maintenance of the cliques of a graph. This problem was investigated by Stix, and in this paper we derive a new simple method based on the Bron–Kerbosch algorithm that compares favourably to Stix' approach. The new approach is generic in the sense that it can be used with other algorithms than just Bron–Kerbosch. The algorithms for finding optimal triangulations are mainly supposed to be off-line methods, but they may form the basis for efficient any-time heuristics. Furthermore, the methods make it possible to evaluate the quality of heuristics precisely and allow us to discover parts of the search space that are most important to direct randomized sampling to.

1. Introduction

Bayesian networks [1] have over the last decades shown that they are among the most important AI techniques and have enjoyed wide-spread use in industry as well as in academia. However, to perform efficient *exact* inference in a Bayesian network, the underlying network often needs to be compiled into a special data structure named a Junction Tree. The creation of an efficient Junction Tree requires that we find a very cost efficient triangulation of the graph of the Bayesian network [2]. It is this triangulation problem that we address in this paper.

Triangulation algorithms aim at minimizing different criteria. The most common are the fill-in, the treewidth and the total table size criteria. The *fill-in criterion* requires the triangulated graph to have the minimum total number of fill-in edges. The *treewidth* of a graph is the size of the largest clique minus one, and the *treewidth criterion* requires the triangulated graph to have minimum treewidth. The *total table size criteria* requires the triangulated graph to have the minimum total table size. Commonly seen triangulation heuristics include min-fill, min-width and min-weight which all greedily pick the next vertex to eliminate based on a local (myopic) score. In *min-fill* a vertex is chosen if its elimination leads to the fewest fill-in edges; in *min-width* a vertex is chosen if it has the fewest number of neighbours; in *min-weight* a vertex v is chosen if the product of the cardinalities of the state spaces of variables corresponding to the vertex and its neigbours in the graph is minimal [3].

[☆] J. Vomlel was supported by the Ministry of Education of the Czech Republic through Grants 1M0572 and 2C06019 and by the Czech Science Foundation through Grants ICC/08/E010 and 201/09/1891.

[*] Corresponding author. Postal address: ÚTIA AV ČR, Pod vodárenskou věží 4, 182 08, Prague 8, Czech Republic. Tel.: +420 266 052 398; fax: +420 286 890 378.
E-mail addresses: thorsten.ottosen@dezide.com (T.J. Ottosen), vomlel@utia.cas.cz (J. Vomlel).

0888-613X/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ijar.2012.06.006

We consider the problem of finding optimal total table size triangulations of Bayesian networks. We solve this problem by searching the space of all possible triangulations. This search is carried out by trying all possible elimination orders and choosing one of those that have a minimal total table size. Of all commonly-used optimality criteria, the total table size yields the most exact bound of the memory and time requirement of the probabilistic inference. However, finding optimal triangulations is difficult. The decision problems of determining if a graph has treewidth less or equal to a given constant k, determining if a graph has a triangulation with a fill-in of size less or equal to k, and determining if a graph has a triangulation with total table size less or equal to k are all NP-complete. The optimization problems of finding a triangulation with minimal treewidth, fill-in, or total table size are all NP-hard. See [4] for the minimum fill-in problem and [5] for triangulation with minimal total table size.

There are several issues that motivate an investigation of this problem. Since the problem is NP-hard, we cannot expect the problem to be solvable in a reasonable amount of time for large networks. However, triangulation can always be performed off-line on specialized servers and saved for use by the inference algorithms. This is important as intractability or simply poor performance is a major obstacle to more wide-spread adoption of Bayesian networks and decision graphs in statistics, engineering and other sciences. Furthermore, efficient off-line algorithms allow us to evaluate the quality of on-line methods which can otherwise only be compared to other on-line methods. An off-line method, on the other hand, can effectively answer whether the subsequent inference can ever be tractable. Finally, off-line methods can often be turned into good any-time heuristics, for example in the spirit of [6].

Previous research on triangulation has also used best-first search [7] and depth-first search [8], however, the optimality criteria is the treewidth of the graph and so the found triangulation is (in the best case) only guaranteed to be within a factor of n (n being the number of vertices of the graph) from the optimal total table size triangulation—this factor could mean the difference between an intractable and a tractable inference. With the treewidth optimality criterion, one can continuously apply the preprocessing rules of [9], but for the total table size criterion we can (so far) only remove simplicial vertices which makes this problem considerably harder. The seminal idea of divide-and-conquer triangulation using decomposable subgraphs dates back to Tarjan [10]. Leimer refines this approach such that the generated subgraphs are not themselves decomposable (i.e., they are maximal prime subgraphs and this unique decomposition is denoted a maximal prime subgraph decomposition) [11]. Basically, this means that the problem of triangulating a graph G is no more difficult than triangulating the largest maximal prime subgraph of G. This decomposition is exploited in [12].

In [13] a dynamic programming algorithm is given based on decompositions by minimal separators, and again the optimality criterion is treewidth. As noted by its authors, the method may be adopted to yield an optimal total table size triangulation as well. Finally, an overview of triangulation approaches is given in [14].

This paper is an extended version of two papers [15,16] presented at The Fifth European Workshop on Probabilistic Graphical Models (PGM 2010) in Helsinki.

2. Preliminaries

We shall use the following notation and definitions. $G = (V, E)$ is an *undirected graph* with *vertices* V (also written as $\mathcal{V}(G)$) and *edges* E (also written as $\mathcal{E}(G)$). For a set of edges F, $\mathcal{V}(F)$ is the set of vertices $\{u, v : \{u, v\} \in F\}$. For $W \subseteq V$, $G[W]$ is the *subgraph induced by* W. Two vertices u and v are *connected* in G if there is an edge between them. A graph G is *complete* if all pairs of vertices $\{u, v\}$ ($u \neq v$) are connected in G. A set of vertices $W \subseteq V$ is *complete in* G if $G[W]$ is a complete graph. The *neighbours* $\text{nb}(v, G)$ of a vertex $v \in V$ is the set $W \subseteq V$ such that each $u \in W$ is connected to v. The *family* $\text{fa}(v, G)$ of a vertex v is the set $\text{nb}(v, G) \cup \{v\}$, and the neighbours and the family of a set of vertices is defined similarly.

If W is complete and no complete set U exists such that $W \subset U$, then W is a *clique*. (Remark: note that any complete set is sometimes called a clique; then what we defined as a clique is called a maximal clique.) The *set of all cliques* of a graph is denoted $\mathcal{C}(G)$ and the set of all cliques that intersects with a set of vertices W is denoted $\mathcal{C}(W, G)$. For a single vertex v we also allow the notation $\mathcal{C}(v, G)$. If $G' = (V, E \cup F)$ is the graph resulting from adding a set of new edges F to G, then $\mathcal{RC}(G, G') = \mathcal{C}(G) \setminus \mathcal{C}(G')$ is the set of *removed cliques*, and $\mathcal{NC}(G, G') = \mathcal{C}(G') \setminus \mathcal{C}(G)$ is the set of *new cliques*. Figure 1 illustrates these concepts. Finally, a complete set of vertices C in G' is called a *clique candidate* for G'.

The *table size* of a clique C is given by $\text{ts}(C) = \prod_{v \in C} |\text{sp}(v)|$ where $\text{sp}(v)$ denotes the state space the variable corresponding to v in the Bayesian network.[1] Finally, the *total table size* of a graph G is given by $\text{tts}(G) = \sum_{C \in \mathcal{C}(G)} \text{ts}(C)$.

An undirected graph is *triangulated* (or chordal) if every cycle of length greater than 3 has a chord (that is, an edge connecting two non-adjacent vertices on the cycle). For example, in Fig. 1 the graph on the left is not triangulated whereas the graph on the right is triangulated. A *triangulation* of G is a set of edges T such that $T \cap E = \emptyset$ and the graph $H = (V, E \cup T)$ is triangulated. T is a *minimal triangulation* if no $T' \subset T$ exists such that $H' = (V, E \cup T')$ is triangulated. We denote the set of all minimal triangulations of a graph G for $\mathcal{T}(G)$.

The *elimination* of a vertex $v \in V$ of $G = (V, E)$ is the process of removing v from G and making $\text{nb}(v, G)$ a complete set by adding missing edges between nodes in $\text{nb}(v, G)$. The added edges are called *fill-in edges*. For example, in Fig. 2 (left), eliminating the vertex 6 induces the two fill-in edges shown with dotted edges in the adjacent graph. The elimination of a

[1] We assume $\text{sp}(v) \geq 2$ for all variables v since variables with $\text{sp}(v) < 2$ can be omitted from the Bayesian network.

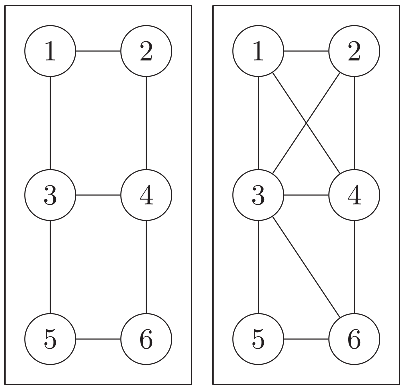

Fig. 1. Left: The initial graph $G = (V, E)$. Right: The updated graph G'. We have $\mathcal{C}(G) = E$ and $\mathcal{C}(G') = \{\{1, 2, 3, 4\}, \{3, 5, 6\}, \{3, 4, 6\}\}$. So in this example we have $\mathcal{RC}(G, G') = \mathcal{C}(G)$ and $\mathcal{NC}(G, G') = \mathcal{C}(G')$.

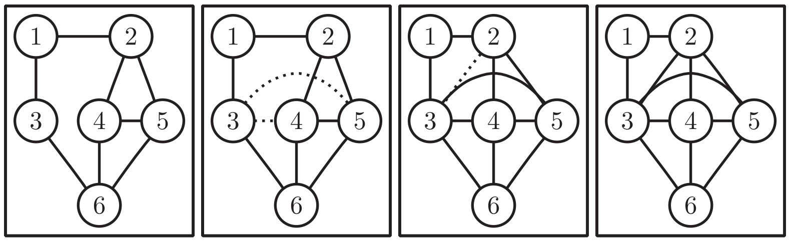

Fig. 2. Example of the fill-in edges and partially triangulated graphs induced by an elimination order that starts with the sequence $\langle 6, 4\rangle$: the dotted edges are fill-in edges. Left: the initial graph. Middle left: the fill-in edges induced by eliminating vertex 6. Middle right: the fill-in edges induced by eliminating vertex 4. Right: the final triangulated graph.

vertex induces a new graph $H = (V, E \cup F)[V \setminus \{v\}]$ where F is the set of fill-in edges added to G to make $\mathrm{nb}(v, G)$ a complete set. If $F = \emptyset$, then v is a *simplicial vertex*.

An *elimination order* of $G = (V, E)$ is a bijection $f : V \to \{1, 2, \ldots, |V|\}$ prescribing an order for eliminating all vertices of G. If a graph $G = (V, E)$ is not triangulated, then we may triangulate it using any given elimination order f by considering the nodes in V in the order defined by f and sequentially adding fill-in edges to G so that after considering node v the set

$$B(v) = \{w \in \mathrm{nb}(v, G) : f(w) > f(v)\}$$

induces a complete subgraph in the extended graph. The union of all the fill-in edges is a triangulation of G.

3. Searching for a triangulation of minimal total table size

In this section we shall discuss how we may search among all minimal triangulations and be guaranteed that a triangulation of the minimal total table size is always included in the set of minimal triangulations. Furthermore, we prove that the total table size of a graph must be monotonely increasing when edges are added to the graph. If these facts were not the case, the methods presented in Section 5 could not be admissible.

The following lemma [17, Lemma 1] relates minimal triangulations with elimination orders.

Lemma 1. *Let T be a minimal triangulation of G. Then there exists an elimination order f of G such that the union of all the fill-in edges is T.*

In this way each minimal triangulation T of G corresponds to at least one elimination order, and we may explore the space $\mathcal{T}(G)$ by investigating all possible elimination orders.

By adding edges we can never lower the total table size. We state this formally in Lemma 3. This lemma is essentially Lemma 4 in [18] but for completeness (and necessity) we provide a more general statement that is valid for any graph (instead of graphs that are already triangulated). In the proof we use Lemma 2.

Lemma 2. *By removing an edge $\{u, v\}$ from a clique $C \supseteq \{u, v\}$ we create at most two new cliques $\{u\} \cup W$ and $\{v\} \cup W$ where $W = C \setminus \{u, v\}$.*

Proof. Obviously C is not a complete set anymore. Sets $\{u\} \cup W$ and $\{v\} \cup W$ are complete and they are cliques if there is not any other complete set containing them in the graph. There are no other subsets of C that would not be subsets of either $\{u\} \cup W$ or $\{v\} \cup W$. □

Lemma 3. *Assume an undirected graph $G = (V, E)$ with $|\mathrm{sp}(v)| \geq 2$ for all $v \in V$. By adding an edge to G the value of the total table size criteria cannot decrease.*

Proof. By adding an edge $\{u, v\}$ we can either:

- join two cliques $C_1 = \{u\} \cup W$ and $C_2 = \{v\} \cup W$ into one clique $C = \{u, v\} \cup W$,
- create a new clique that replaces one original clique, either clique $\{u\} \cup W$ or $\{v\} \cup W$ replaces $W \in \mathcal{C}(G)$, or
- create a new clique without removing any.

Since removing and adding edges are symmetric operations it follows from Lemma 2 that no other cases are possible. In the first case we can write:

$$\begin{aligned}
&\mathrm{ts}(\{u, v\} \cup W)\\
&\quad = |\mathrm{sp}(u)| \cdot |\mathrm{sp}(v)| \cdot \prod_{w \in W} |\mathrm{sp}(w)|\\
&\quad = \frac{1}{2}|\mathrm{sp}(u)| \cdot |\mathrm{sp}(v)| \cdot \prod_{w \in W} |\mathrm{sp}(w)| \ + \ \frac{1}{2}|\mathrm{sp}(u)| \cdot |\mathrm{sp}(v)| \cdot \prod_{w \in W} |\mathrm{sp}(w)|\\
&\quad \geq |\mathrm{sp}(v)| \cdot \prod_{w \in W} |\mathrm{sp}(w)| \ + \ |\mathrm{sp}(u)| \cdot \prod_{w \in W} |\mathrm{sp}(w)|\\
&\quad = \mathrm{ts}(\{u\} \cup W) \ + \ \mathrm{ts}(\{v\} \cup W).
\end{aligned}$$

In the second case:

$$\mathrm{ts}(\{w\} \cup C) = |\mathrm{sp}(w)| \cdot \prod_{x \in C} |\mathrm{sp}(x)| \ > \ \mathrm{ts}(C),$$

where w is either u or v. Clearly, also in the third case the total table size must increase. □

A consequence of Lemma 3 is that in order to find a triangulation minimizing the total table size criteria, it is sufficient to search among minimal triangulations [18, Lemma 5]:

Lemma 4. *Assume an undirected graph $G = (V, E)$ with $|\mathrm{sp}(v)| \geq 2$ for all $v \in V$. Then there exists a minimal triangulation that also minimizes the total table size criteria.*

It follows from Lemma 1 that this can be accomplished by investigating all possible elimination orders, which is also the assertion of Theorem 4 in [18].

4. The search space for triangulation algorithms

Our goal is to explore the space $\mathcal{T}(G)$ encoding all possible minimal ways to triangulate a graph G in. To do this, we generate a search graph dynamically (on-demand) where each node corresponds to a subset of V being eliminated from G, and where each edge is labeled with the particular vertex that has been eliminated. (Note that we exclusively use the term “node” for vertices in the search graph whereas the term “vertex” is used exclusively for vertices in the undirected graph being triangulated.) In the *start node s* no vertices have been eliminated, and in a *goal node t* all vertices have been eliminated and the graph G has been triangulated.

Since we are seeking optimal total table size triangulations we also need to associate this quantity with each node. By definition, the total table size is easy to compute if we know the cliques of the partially triangulated graph, and therefore we also need to associate this graph with each node. Below we give a small example of a path in the search space—in Section 6 we shall explain the algorithms in detail.

Example 1. Consider the graphs in Fig. 2 and assume that all variables (in the original Bayesian network) are binary. The graph associated with the start node s would be the graph on the left and this graph has a total table size of $2^2+2^2+2^2+2^3+2^3 = 28$.

The graph associated with a successor node m of s (corresponding to the elimination of vertex 6) would correspond to the graph in the middle (left) (including dotted edges) with total table size $2^2 + 2^2 + 2^3 + 2^4 = 32$.

And the successor node of m (corresponding to the elimination of vertex 4) would be associated with graph on the right, which is also a goal node, with total table size $2^3 + 2^4 + 2^4 = 40$. Note that when introducing fill-in edges, we must not add edges to vertices that have already been eliminated—this is why this step does not add the edge $\{2, 6\}$ even though the vertices are both neighbours of vertex 4.

By adding edges we can never lower the total table size (cf. Lemma 3). As a consequence, the total table size of a node is never higher than the total table size of its successor node(s). This implies that the total table size associated with any non-goal node n is a lower-bound on the total table size of any goal node that may be discovered from n. This property guarantees that the algorithms in Section 6 are admissible.

5. Maintaining cliques of a dynamic graph

In this section we consider the problem of maintaining the set of cliques of a dynamic undirected graph. The graph is dynamic in the sense that edges can be removed and added, but the set of vertices is invariant. When we add a new set of edges, we call the problem *incremental*, and when we remove a set of edges, we call the problem *decremental*. Finding all the cliques of a static graph is a hard problem: determining whether there is a clique of size k in a graph is NP-complete [19] and listing all the cliques may require exponential time as there exists graphs with exponentially many cliques [20]—albeit it is solvable in polynomial time for many classes of graphs. However, in this work we shall consider the initial set of cliques for given (several well-known algorithms exists for this purpose).

Previous research has been motivated by fuzzy clustering [21], but we have another application in mind. Specifically, our motivation is to find optimal triangulations of Bayesian networks with respect to the total table size criterion by using a best-first or depth-first search. This requires a lower bound on the total table size for which we may use the total table size of the current partially triangulated graph. In turn, this requires that we know the cliques of the current graph.

Finally, there also seem to be a growing interest in detecting cliques using graph theoretical methods in areas like computational biochemistry, genomics and bioinformatics [22]. Some of these applications even ask for dynamic updates [23].

5.1. Stix' approach to clique maintenance

Stix observed that it was somewhat expensive to recompute all cliques of a graph given that the graph had only changed slightly. Therefore Stix derived the approach explained below and showed that it did indeed out-perform a full recomputation scheme [21].

Stix' approach works by adding (removing) one edge at a time. To add (remove) a set of edges, the technique is simply applied once for each edge. The technique (both for incremental and decremental problems) may be summarized as follows:

(1) Let $G = (V, E)$ be an undirected graph, and let $G' = (V, E \cup \{\{v, w\}\})$.
(2) Initially let $\mathcal{C} = \mathcal{C}(G)$.
(3) Generate a set of clique candidates $\mathcal{K}$ for the updated graph G'.
(4) Add/remove a candidate $C \in \mathcal{K}$ to/from $\mathcal{C}$ depending on whether it is a clique in G'.
(5) In the end, $\mathcal{C}$ equals $\mathcal{C}(G')$.

The check in step (4) is shown in Algorithm 1 where we have improved Stix' approach by only considering the neighbours $\mathrm{nb}(C, G')$ of a clique candidate C (notice that this algorithm should be called with the updated graph G' as its second argument).

Stix' algorithm is based on the following theorem:

Theorem 1 [21]. *Let $G = (V, E)$ be an undirected graph, and let $G' = (V, E \cup \{\{v, w\}\})$ be the graph after adding the edge $\{v, w\}$. Then*

1. *All cliques of $\mathcal{C}(G)$ that do not contain v or w are in $\mathcal{C}(G')$.*
2. *For all $A \in \mathcal{C}(v, G)$ and for all $B \in \mathcal{C}(w, G)$ we have*
   (a) *$L = A \cap B \cup \{v, w\}$ is a clique candidate for G'.*
   (b) *$|A \setminus B| = 1 \implies A \notin \mathcal{C}(G')$; otherwise A is a clique candidate for G'.*
   (c) *$|B \setminus A| = 1 \implies B \notin \mathcal{C}(G')$; otherwise B is a clique candidate for G'.*
3. *The set $\mathcal{C}(G')$ is fully determined by statement (1) and by inspecting all the clique candidates of statement (2).*

Stix' algorithm with several improvements is presented as Algorithm 2. Notice that the first part of condition $2(b)$ and $2(c)$ is not checked in Algorithm 2. We conducted experiments with these conditions being checked, but found it to be about 40% slower. Furthermore, we accumulate clique candidates in a set to reduce the number of calls to IsClique$(\cdot)$.

Next we illustrate how Stix' algorithm works in a small example.

Algorithm 1 . Verifying a complete set C is a clique (improved version). In Stix' original version, line 3 iterated over all vertices in $V \setminus C$

```
1: function IsClique($C$, $G$)
2:     Input: A non-empty, complete set of vertices $C$, and a graph $G$.
3:     for all $v \in \mathrm{nb}(C, G)$ do
4:         if $C \subseteq \mathrm{nb}(v, G)$ then return false
5:         end if
6:     end for
7:     return true
8: end function
```

Algorithm 2. Incremental clique maintenance by single-edge updates (improved version). In Stix' original version the set $\mathcal{K}$ of clique candidates was not used; instead the call to IsClique$(\cdot)$ was applied immediately after constructing a clique candidate

```
1: function EdgeBasedUpdate($\mathcal{C}$, $G$, $F$)
2:     Input: A graph $G = (V, E)$, the set of cliques $\mathcal{C}$ of $G$, and the set of new edges $F$.
3:     for all $\{u, v\} \in F$ do
4:         Let $G' = (V, \mathcal{E}(G) \cup \{\{u, v\}\})$
5:         Set $\mathcal{K} = \emptyset$
6:         for all $A \in \mathcal{C}(u, G)$ do                ▷ Generate clique candidates by Theorem 1
7:             for all $B \in \mathcal{C}(v, G)$ do
8:                 Let $C = A \cap B \cup \{u, v\}$
9:                 Set $\mathcal{K} = \mathcal{K} \cup \{C\}$
10:            end for
11:        end for
12:        for all $K \in \mathcal{C}(u, G) \cup \mathcal{C}(v, G)$ do
13:            if !IsClique($K$, $G'$) then Set $\mathcal{C} = \mathcal{C} \setminus \{K\}$
14:            end if
15:        end for
16:        for all $K \in \mathcal{K}$ do
17:            if IsClique($K$, $G'$) then Set $\mathcal{C} = \mathcal{C} \cup \{K\}$
18:            end if
19:        end for
20:        Set $G = G'$
21:    end for
22:    return $\mathcal{C}$
23: end function
```

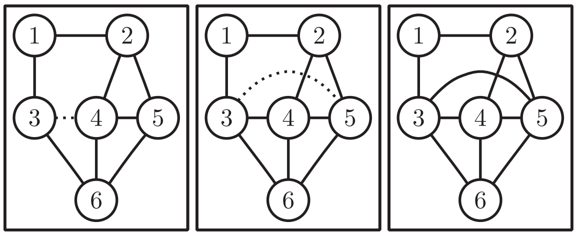

Fig. 3. The sequence of graphs considered by Stix' algorithm (Algorithm 2) when adding the set of edges {{3, 4}, {3.5}}. Left: The initial graph—a dotted edge indicates it is about to be added to the graph. Middle: the graph after the first edge has been added. Right: The final graph.

Example 2. Consider the graph in Fig. 3 on the left which we want to update with the set of edges $\{\{3, 4\}, \{3, 5\}\}$. When adding the edge $\{3, 4\}$, line 7-13 in Stix' algorithm combines the two sets of cliques $\mathcal{C}(3, G) = \{\{1, 3\}, \{3, 6\}\}$ and $\mathcal{C}(4, G) = \{\{2, 4, 5\}, \{4, 5, 6\}\}$. The resulting set of clique candidates is then $\mathcal{K} = \{\{3, 4\}, \{3, 4\}, \{3, 4\}, \{3, 4, 6\}\}$. Then follows a series of calls to `IsClique(·)` which determines that the clique $\{3, 6\}$ must be removed from and the clique $\{3, 4, 6\}$ must be added to $\mathcal{C}$.

In the second iteration we add the edge $\{3, 5\}$ and get the set of candidates $\mathcal{K} = \{\{3, 5\}, \{3, 5\}, \{3, 4, 5\}, \{3, 4, 5, 6\}\}$ and determine that the cliques $\{3, 4, 6\}$ and $\{4, 5, 6\}$ must be removed and the clique $\{3, 4, 5, 6\}$ must be added.

The above example shows that there are two potential performance problems with Stix' approach when adding multiple edges:

1. Many duplicate clique candidates are generated and existing cliques are combined multiple times, and
2. A great number of calls to `IsClique(·)` are needed to prune candidates and remove old cliques.

To overcome these problems, one might try to generalize Stix' theoretical results to account for a larger set of edges being added at one time. However, it turns out that such an approach suffers even more from the problems above. In the following we shall therefore present a radically different approach that overcomes both problems, that is, an approach where (a) fewer clique candidates are generated, and (b) no maximality check is needed to determine the relevance of such clique candidates.

5.2. Clique maintenance by local search

The general idea behind this method is simple: instead of running the Bron–Kerbosch algorithm [24] (or a similar algorithm for finding cliques) on the whole graph, run it on a smaller subgraph where all the new cliques appear and existing cliques disappear. Then simply update the set of cliques based on the vertices of the subgraph and the newly found cliques. Algorithm 3 is the modified Bron–Kerbosch algorithm which by using a pivot can reduce the search space (to get the original algorithm simply exchange the iteration in line 6 with "**for all** $v \in P$ **do**"). In our implementation we pick the pivot deterministically as the first vertex in P because this is very easy (for alternative pivot selection strategies see [25,26]). We also tested the techniques of [27,28], but found that the Bron–Kerbosch algorithm with pivot was much faster.

Algorithm 3. The Bron–Kerbosch algorithm with pivot. The algorithm returns the set of cliques in the graph $G = (V, E)$ when called with the arguments $(G, \emptyset, V, \emptyset)$

```
1:  function BKWithPivot($G, R, P, X$)
2:      if $P = \emptyset$ and $X = \emptyset$ then return $\{R\}$
3:      else
4:          Let $\mathcal{C} = \emptyset$
5:          Let $u$ = SelectPivot($P, X, G$)
6:          for all $v \in P \setminus \mathrm{nb}(u, G)$ do
7:              Set $P = P \setminus \{v\}$
8:              Let $R_{new} = R \cup \{v\}$
9:              Let $P_{new} = P \cap \mathrm{nb}(v, G)$
10:             Let $X_{new} = X \cap \mathrm{nb}(v, G)$
11:             Let $\mathcal{K}$ = BKWithPivot($G, R_{new}, P_{new}, X_{new}$)
12:             Set $X = X \cup \{v\}$
13:             Set $\mathcal{C} = \mathcal{C} \cup \mathcal{K}$
14:         end for
15:         Return $\mathcal{C}$
16:     end if
17: end function
```

To find all cliques of a graph G, the algorithm should be called with the argument tuple $(G, \emptyset, \mathcal{V}(G), \emptyset)$. However, an important observation is that the algorithm can also search a subgraph $G[W]$ for cliques by simply passing the arguments $(G, \emptyset, W, \emptyset)$. It is this ability that our new clique maintenance algorithm takes advantage of. Our new algorithm for dynamic clique maintenance is presented in Algorithm 4, and explained in the next example.

Example 3. Consider again Fig. 3. We immediately update the graph $G = (V, E)$ with the set of new edges $\{\{3, 4\}, \{3, 5\}\}$ (line 6). The set U becomes $\{3, 4, 5\}$ and $\mathrm{fa}(U, G')$ actually equals V and so we will run Bron–Kerbosch on the whole graph (of course, for larger graphs this is rarely the case). Then we iterate through the existing cliques and remove those that intersect with U (line 9-13)—this step only leaves the clique $\{1, 2\}$ in $\mathcal{C}$. Then we add all the cliques found in the subgraph $G'[\mathrm{fa}(U, G')]$ (line 14-18) if and only if they intersect with U—in this case only $\{1, 2\}$ is not added. We can observe that the clique $\{2, 4, 5\}$ is both removed and added again by the algorithm.

As the above example explains, our algorithm sometimes removes and adds the same clique. This is usually not a problem in practice, as comparison of cliques is much faster than calling `IsClique(·)`. The correctness of the algorithm follows from the results below.

Lemma 5. *Let $G = (V, E)$ be an undirected graph, and let $G' = (V, E \cup F)$ be the graph resulting from adding a set of new edges F to G. Let $U = \mathcal{V}(F)$. If $C \in \mathcal{NC}(G, G')$, then $C \subseteq \mathrm{fa}(U, G')$.*

Proof. Since C is a new clique, it must contain at least two vertices from U. Since C is complete all vertices $v \in C \setminus U$ must be connected to some vertex in U. Hence v is a neighbour of U. □

Lemma 6. *Let G and G' be given as in Lemma 5. Then $C \in \mathcal{RC}(G, G')$ if and only if there exists $K \in \mathcal{NC}(G, G')$ such that $C \subset K$.*

Algorithm 4. Incremental clique maintenance by local search

```
1: function SetBasedUpdate($\mathcal{C}$, $G$, $F$)
2:     Input: A graph $G = (V, E)$, the set of cliques $\mathcal{C}$ of $G$, and the set of new edges $F$.
3:     Let $U = \mathcal{V}(F)$
4:     Let $G' = (V, E \cup F)$
5:     Let $\mathcal{C}^{new} = $ BKWithPivot($G'$, $\emptyset$, fa($U$, $G'$), $\emptyset$)
6:     for all $C \in \mathcal{C}$ do                                  ▷ Remove old cliques
7:         if $C \cap U \neq \emptyset$ then Set $\mathcal{C} = \mathcal{C} \setminus \{C\}$
8:         end if
9:     end for
10:    for all $C \in \mathcal{C}^{new}$ do                            ▷ Add new cliques
11:        if $C \cap U \neq \emptyset$ then Set $\mathcal{C} = \mathcal{C} \cup \{C\}$
12:        end if
13:    end for
14:    return $\mathcal{C}$
15: end function
```

Proof. By Lemma 5, all new cliques $K \subseteq \text{fa}(U, G')$. Since a newly formed clique K is the only way to remove an existing clique C from $\mathcal{C}(G')$, the result follows. $\square$

Theorem 2. *Let G, G', F and U be given as in Lemma 5. The cliques of $\mathcal{C}(G')$ can be found by removing the cliques from $\mathcal{C}(G)$ that intersect with U and adding cliques of $G'[\text{fa}(U, G')]$ that intersect with U.*

Proof. We first show we add all new cliques by considering just $G'[\text{fa}(U, G')]$. By Lemma 5, this subgraph contains all the new cliques. Furthermore, any new clique C must intersect with U; otherwise it could not contain a new edge. Therefore all the new cliques are added.

We remove all relevant cliques if $C \in \mathcal{RC}(G, G')$ implies $C \cap U \neq \emptyset$. So let $C \in \mathcal{RC}(G, G')$. Assume $C \cap U = \emptyset$; then for each $v \in \text{nb}(C, G) \cap U$, $C \not\subseteq \text{nb}(v, G)$ (otherwise C could not be a clique in G). But then no new clique can cover C which is a contradiction to Lemma 6. Hence $C \cap U \neq \emptyset$, and therefore we remove all relevant cliques.

Last we consider that we might also remove a clique $C \in \mathcal{C}(G) \cap \mathcal{C}(G')$. Since $C \cap U \neq \emptyset$, then $C \subseteq \text{fa}(U, G')$, and C will be added again when we add cliques from $G'[\text{fa}(U, G')]$ that intersect with U. $\square$

Remark. Theorem 2 also implies that we can apply further pruning based on the set U in Algorithm 3. In particular, the for-loop in line 7 can be skipped if $(R \cup P) \cap U = \emptyset$. We have not implemented this pruning, however.

Remark. Stix derives a second approach to deal with the decremental problem. A nice property of our approach is that it works almost unaltered for this case (simply remove the set F of edges from the graph instead of adding them).

Remark. If Algorithm 4 is to be used with large sparse graphs, then it becomes a problem that in lines 8–11 we need to scan all existing cliques instead of just those in $\text{fa}(U, G)$. In this case, an implementation should maintain a list of references of cliques for each vertex. In each list should be put a reference to a clique if that clique contains the vertex, and we then only need to scan the lists induced by U. We have not implemented this optimization either.

We report the results of experimental comparisons of algorithms for dynamic clique maintenance in Section 7.1.

6. Optimal total table size triangulation algorithms

We have now shown how we may efficiently compute the total table size for each successor m of a node n in the search space $\mathcal{T}(G)$, and we have furthermore established that the total table size for a node n is a lower-bound of any possible triangulation associated with the set of goal nodes reachable from n. Given this, we may use standard algorithms like best-first search and depth-first search to explore the search space and at the same time be guaranteed that the algorithms terminate with an optimal solution.

Best-first search is an algorithm that successively expands nodes with the shortest distance to the start node until a goal node has a shorter or equally short path than all non-goal nodes. The benefit of the best-first strategy is that we may avoid exploring paths that are far from the optimal path. The disadvantage of a best-first strategy is that the algorithm must keep track of a *frontier* (or fringe) or nodes that still needs to be explored. *Depth-first search*, on the other hand, explores all paths in a depth-first manner and therefore uses only $\Theta(|V|)$ memory for a graph $G = (V, E)$. However, depth-first search is typically forced to explore more paths than best-first search.

To compute a cost for each path in the search graph, we associate the following with each node n:

1. $H = (V, E \cup F)$: the original graph with all fill-in edges F accumulated along the path to n from the start node s.
2. R: the remaining graph $H[V \setminus W]$ where W are the vertices of G eliminated along the path from s to n.
3. $\mathcal{C}$: the set of cliques for H, $\mathcal{C}(H)$.
4. tts: the total table size for the graph H.
5. $\mathcal{L}$: a list of vertices describing the elimination order.

To maintain tts(H) efficiently we need $\mathcal{C}(H)$ which in turn requires H, and we saw how this can be done in Section 5. The graph R makes it easy to determine if the graph H is triangulated and may be computed on demand to reduce memory requirements.

In the worst case, the complexity of any best-first search method is $O(\beta(|V|) \cdot |V|!)$ (where $\beta(\cdot)$ is a function that describes the per-node overhead) because we must try each possible elimination order. However, it is well known that the remaining graph $H[V \setminus W]$ is the same no matter what order the vertices in W have been eliminated in, and so we can use *coalescing* of nodes and thus reduce the worst case complexity to $O(\beta(|V|) \cdot 2^{|V|})$ [29].

For depth-first search the complexity is often thought to remain at $\Theta(\gamma(|V|) \cdot |V|!)$, however, at the expense of memory we may also apply coalescing for pruning purposes. Hence, depth-first search can be made to run in $O(\gamma(|V|) \cdot |V|!)$ time using $O(2^{|V|})$ memory, but the hidden constants will be much smaller in connection with memory consumption compared to best-first search.

Both $\gamma(|V|)$ and $\beta(|V|)$ take at least $O(|V|^3)$ time as they are dominated by the removal of simplicial vertices (the lookup into the coalescing map takes $O(|V|)$ time due to the computation of the hash-key, and the priority queue look-up for best-first search may take $O(|V|)$ time since the queue may become exponentially large). Getting a more precise bound on the two functions is difficult as the complexity of maintaining the cliques and total table size depends very much on the graph being triangulated (of course, if the graph has an exponential number of cliques, updating the cliques will dominate the complexity; in Section 7.2 we shall see that this is often the case even though the number of cliques is far from exponential).

In Algorithm 5 we describe the basic best-first search with coalescing, and depth-first search with coalescing and pruning based on the currently best path is described in Algorithm 6. We establish an initial solution via a greedy min-fill heuristic. The idea of using an initial upper-bound (and improving it during the search) to prune non-optimal paths is well-known, see e.g., [30]. The procedure `EliminateVertex(·)` simply eliminates a vertex from the remaining graph R and updates the cliques and total table size of the partially triangulated graph H (see Section 5). The procedure `EliminateSimplicial(·)` removes all simplicial vertices from the remaining graph. It is this procedure that has $O(|V|^3)$ complexity.

Algorithm 5. Best-first search with coalescing for finding an optimal triangulation of the graph G with respect to the total table size criterion

```
1:  function TriangulationByBFS($G$)
2:      Let $s = (G, G, \mathcal{C}(G), \text{tts}(G), \langle\rangle)$
3:      EliminateSimplicial($s$)
4:      if $|\mathcal{V}(s.R)| = 0$ then return $s$
5:      end if
6:      Let $map = \emptyset$                                        ▷ Coalescing map
7:      Let $\mathcal{O} = \{s\}$                                     ▷ The open set
8:      while $\mathcal{O} \neq \emptyset$ do
9:          Let $n = \arg\min_{x \in \mathcal{O}} x.\text{tts}$        ▷ Select node to expand
10:         if $|\mathcal{V}(n.R)| = 0$ then return $n$           ▷ Return optimal goal node
11:         end if
12:         Set $\mathcal{O} = \mathcal{O} \setminus \{n\}$
13:         for all $v \in \mathcal{V}(n.R)$ do
14:             Let $m = $ Copy($n$)
15:             EliminateVertex($m, v$)                  ▷ Update graph, cliques and tts
16:             EliminateSimplicial($m$)                 ▷ Remove simplicial vertices
17:             if $map[m.R].\text{tts} \leq m.\text{tts}$ then continue
18:             end if
19:             Set $\mathcal{O} = \mathcal{O} \setminus \{map[m.R]\}$
20:             Set $map[m.R] = m$
21:             Set $\mathcal{O} = \mathcal{O} \cup \{m\}$
22:         end for
23:     end while
24: end function
```

Algorithm 6. Depth-first search with coalescing and upper-bound pruning for finding an optimal triangulation of the graph G with respect to the total table size criterion. In line 6 we establish an initial solution via a greedy min-fill heuristic.

```
1: function TriangulationByDFS($G$)
2:     Let $s = (G, G, \mathcal{C}(G), \text{tts}(G), \langle\rangle)$
3:     EliminateSimplicial($s$)
4:     if $|\mathcal{V}(s.R)| = 0$ then return $s$
5:     end if
6:     Let $best$ = MinFill($s$)                              ▷ Best path
7:     Let $map = \emptyset$                                  ▷ Coalescing map
8:     ExpandNode($s$, $best$, $map$)                         ▷ Start recursive call
9:     return $best$
10: end function
11: procedure ExpandNode($n$, &$best$,&map)
12:     for all $v \in \mathcal{V}(n.R)$ do
13:         Let $m$ = Copy($n$)
14:         EliminateVertex($m$, $v$)                          ▷ Update graph, cliques and tts
15:         EliminateSimplicial($m$)                           ▷ Remove simplicial vertices
16:         if $|\mathcal{V}(m.R)| = 0$ then                   ▷ Found a goal node
17:             if $m.\text{tts} < best.\text{tts}$ then Set $best = m$
18:             end if
19:         else                                               ▷ Not a goal node
20:             if $m.\text{tts} \geq best.\text{tts}$ then continue
21:             end if
22:             if $map[m.R].\text{tts} \leq m.\text{tts}$ then continue
23:             end if
24:             Set $map[m.R] = m$
25:             ExpandNode($m$, $best$, $map$)
26:         end if
27:     end for
28: end procedure
```

7. Results

7.1. Dynamic clique maintenance

In this section we shall compare approaches for dynamic clique maintenance. For that purpose we have used a set of public Bayesian networks.[2] We did not moralize the initial directed network even though this might lead to a slightly more accurate test. By taking the underlying graph we have got seven real-world undirected graphs.[3] For each graph we generated 10 more by successively adding 5% of the missing (undirected) edges at random (in total 77 graphs). In particular, a random edge was generated by picking its endpoints randomly, and if the edge was missing in the graph it was added; otherwise we continued until a missing edge was generated. We have then performed two tests on this dataset:

1. *Test 1.* For each graph in the dataset, add all the missing edges in isolation. The set of cliques is updated after an edge is added. Then the edge is removed, and the next edge is added etc.
2. *Test 2.* For each graph in the dataset (after all initial simplicial vertices are removed), triangulate the graph by making all vertices simplicial in some random order. The set of cliques is updated after each vertex is made simplicial.

These two test scenarios were chosen because we believe that they show the worst-case performance (the scenario of Test 1), and the expected speedup for our triangulation problem (the scenario of Test 2). For each graph we then ran the one-edge test (Test 1) 2000 times and simplicial-vertices test (Test 2) 3000 times (with different random order of vertices for each run of Test 2) and saved the mean time. We have then plotted the mean time of the improved Stix' approach divided by the mean time of the new approach (we call this the "saving ratio").

In Figs. 4 and 5 we have collected the results of the two tests. The total time saving ratio for various graph densities are summarized in Table 1.

We can see that even for Test 1, the new method always performs better, and there is a slight tendency for the saving ratio to rise with the size of the graph. There seems to be no clear connection between the density of the graph and the saving ratio.

[2] http://compbio.cs.huji.ac.il/Repository/networks.html.
[3] The seven graphs are: Insurance, Water, Mildew, Alarm, Barley, Hailfinder, and Carpo.

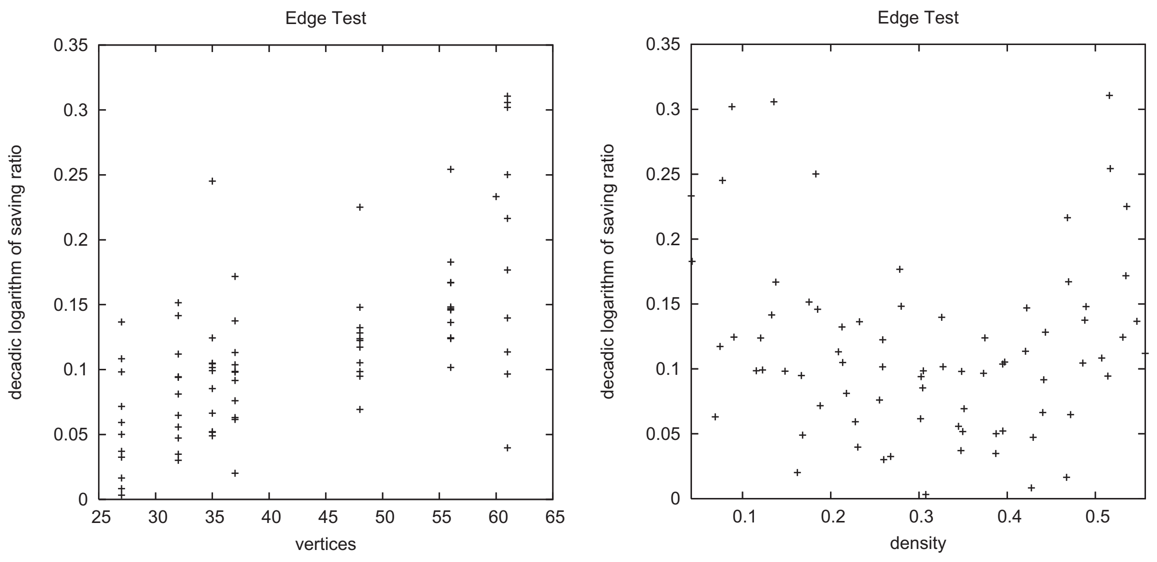

Fig. 4. Results for Test 1: updating the set of cliques after adding a single edge. Each cross represents the mean time of our method divided by the mean time for the improved Stix method on a particular graph. All points are above zero, which indicates that the new approach was always faster (notice the logarithmic *y*-axis).

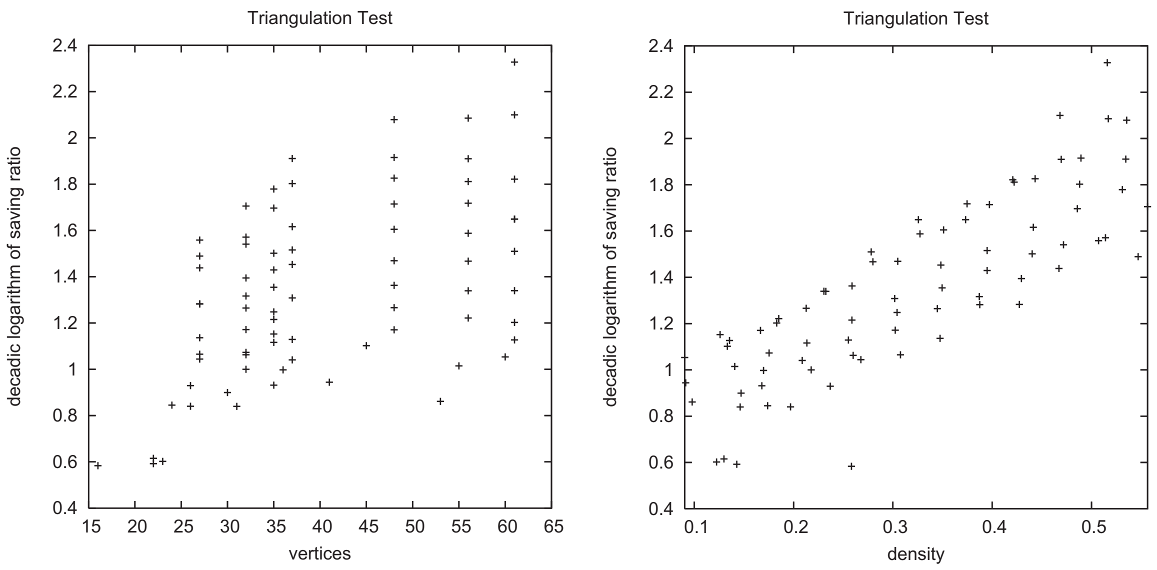

Fig. 5. Results for Test 2: updating the set of cliques after adding fill-in edges. Each cross represents the mean time of our method divided by the mean time for the improved Stix method on a particular graph. All points are above zero, which indicates that the new approach was always faster (notice the logarithmic *y*-axis).

For Test 2 the new method is significantly better, especially for more dense graphs. There also seems to be a clear connection between the density of the graph and performance, with the saving ratio increasing as the density increases. We suspect that this is because denser graphs have a tendency to require more fill-in edges per non-simplicial vertex (on average) than sparser graphs thereby leading to more iterations for Stix' approach.

Referring to Table 1, then we can make several observations. First, the improved version of Stix' algorithm is between 1.5 and 7 times faster than the original algorithm. Secondly, Stix' original algorithm often performs worse than the Bron–Kerbosch algorithm; we are therefore unable to reproduce the results reported by Stix (Table 1 in [21]) on our set of graphs. We speculate that Stix might not have used a pivot in the Bron–Kerbosch algorithm. Third, and perhaps a little surprising, we see that the Bron–Kerbosch algorithm in Test 2 basically performs equally to our new algorithm for densities above 0.2 and almost as good for densities in the range $[0.1, 2)$. Luckily, it is easy to modify Algorithm 4 to fallback dynamically to the Bron–Kerbosch algorithm when $\text{fa}(U, G')$ becomes too large (say, $|\text{fa}(U, G')| > |V|/2$). In this manner, one can combine the strengths of both algorithms (we have not implemented this optimization).

Table 1
Total time saving ratio for different graph densities and for different methods: original Stix' method (origS), improved Stix' method (imprS), Bron–Kerbosch algorithm on the whole graph (BK), and the new approach (localBK). A value above one indicates that the second method was faster in terms of total running time for all graphs of the specified density.

Density δ	origS/BK	origS/imprS	imprS/localBK	BK/localBK
Test 1 results				
$\delta \in [0, 0.1)$	0.43	1.58	1.58	5.85
$\delta \in [0.1, 0.2)$	0.57	1.78	1.51	4.75
$\delta \in [0.2, 0.3)$	0.83	2.54	1.32	4.04
$\delta \in [0.3, 0.4)$	1.53	3.94	1.27	3.27
$\delta \in [0.4, 0.5)$	2.45	6.04	1.44	3.56
$\delta \in [0.5, 0.6)$	5.15	6.94	1.85	2.50

	imprS/localBK	BK/localBK
Test 2 results		
$\delta \in [0, 0.1)$	9.415	1.303
$\delta \in [0.1, 0.2)$	13.596	1.076
$\delta \in [0.2, 0.3)$	24.137	1.016
$\delta \in [0.3, 0.4)$	40.987	0.998
$\delta \in [0.4, 0.5)$	82.285	1.005
$\delta \in [0.5, 0.6)$	153.930	1.000

Since our new method is the fastest algorithm overall, we have used it internally in all of the algorithms that are compared in the next section.[4]

7.2. Optimal triangulation

In this section we describe experiments with the methods for optimal triangulation minimizing the total table size criteria. We also report results of experiments with several heuristics derived from these.

For the purpose of triangulation experiments we have in addition to the 77 graphs (from Test 1 and Test 2) generated 50 random graphs with varying size and density. These graphs are special bipartite graphs that result from the application of rank-one decomposition to BN2O networks—see [31] for details. These graphs have three characteristics: (1) they consist solely of binary nodes, (2) they need not be moralized after using the rank-one decomposition, and (3) bottom and top layers are not connected. Thereby the initial graph is sparser than usual which gives triangulation algorithms more freedom (in terms of choosing fill-in edges) when searching for a triangulation. For some of these graphs this freedom causes the simple triangulation heuristics to terminate with triangulations that are far from optimal.

We have performed two different tests on this dataset:

1. *Test 3.* A comparison of depth-first search and best-first search, and
2. *Test 4.* A comparison between heuristic methods.

In both Tests 3 and 4 we do not perform any initial maximal prime subgraph decomposition [13] leading to harder problem for the algorithms. For Test 4 we have implemented the following heuristic methods:

(a) *Limited-branching depth-first search*. This means that we only expand the n successors of a node which have the lowest total table size. For example, "limited-branch-5" expands at most five successors per node.
(b) *Limited memory best-first search*. Here we limit the size of the open set $\mathcal{O}$ to some fixed value n by removing the worst nodes when the set is considered full. For example, "limited-mem-10k" has at most 10.000 nodes in its open set.
(c) The min-width, min-fill, and min-weight heuristics implemented so that a successor node is generated for all ties instead of breaking ties randomly. These algorithms therefore perform a global search over all possible min-width, min-fill, and min-weight induced triangulations, respectively. We refer to these algorithms as *min-width**, *min-fill**, and *min-weight**, respectively.

The results from Test 3 are given in Fig. 6. It appears that the performance of depth-first search and best-first search is comparable; best-first search is slightly better. In Table 2 we give summary statistics for this test. We computed the p-value of the Wilcoxon two-sample test of the null hypothesis that the distribution of depth-first time minus best-first time is symmetric about 0 with the alternative hypothesis being that the values for depth-first search are greater. The p-value is 0.01962, which means that the null hypothesis is rejected, that is, differences are significant at the 5% level (in favor of best-first search being faster).

To see the importance of the efficiency of clique maintenance for the problem of finding an optimal triangulation, we measured the percentage of the total processing time depth-first search and best-first search spent on clique maintenance.

[4] All experiments in Sections 7.1 and 7.2 were conducted on a Core i7-2700k using only one 3.5 GHz core at a time. The implementation is written in C++ and comprises about 20, 000 lines of code.

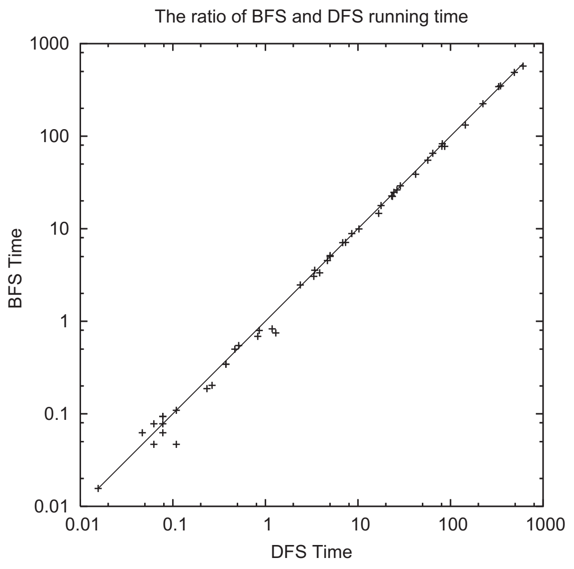

Fig. 6. Comparison of the running time of best-first search and depth-first search. Both algorithms terminate with an optimal triangulation. Each cross represents a pair of running times of best-first search and depth-first search for a graph (values above the line indicate that depth-first search was faster).

Table 2
Summary statistics for exhaustive search algorithms. Each row summarizes the mean time for graphs with *n* vertices. The value in parenthesis in the first column indicates the number of graphs of that size.

Vertices (no. of graphs)	Mean time of DFS	Mean time of BFS
20 (10)	0.43s	0.31s
30 (20)	13.14s	12.11s
40 (10)	39.79s	38.63s
>40 (10)	212.18s	209.80s
Fastest	42%	62%

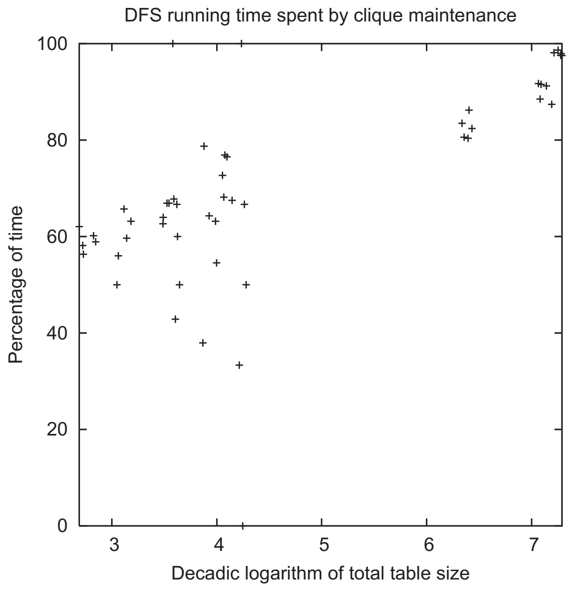

Fig. 7. The percentage of running time spent by depth-first search on clique maintenance. Each cross corresponds to computations on one bipartite graph of a BN2O network.

We report this percentage in Figs. 7 and 8. We can see that in the graphs with large total table size most running time is actually spent by clique maintenance.

The results from Test 4 are given in the Table 3. Here we have run the heuristics on the 50 graphs (corresponding to BN2O networks) from Test 3 and on 77 graphs (corresponding to Bayesian networks from the repository) from Tests 1 and 2. In the

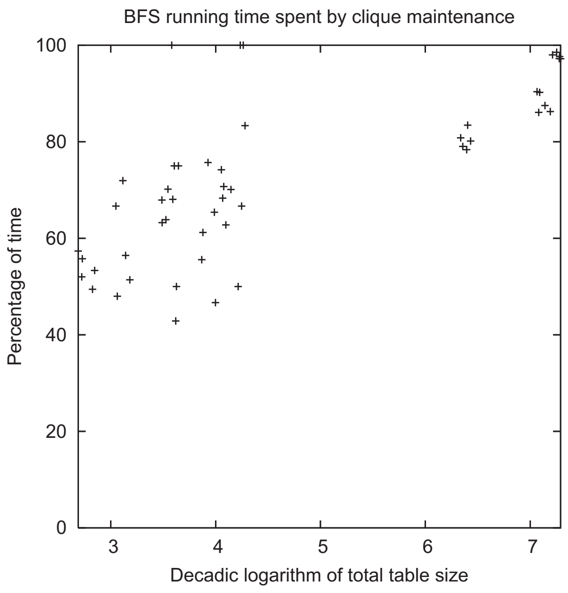

Fig. 8. The percentage of running time spent by best first search on clique maintenance. Each cross corresponds to computations on one bipartite graph of a BN2O network.

Table 3
Results for heuristic algorithms. The first column describes the percentage of graphs that were triangulated optimally, and the second column contains the maximum percent-wise deviation from the total table size of an optimal triangulation. The third column indicates total time for triangulating all graphs. Note that min-width* is equal to min-weight* for BN2O models.

	% Optimal	Max %-wise deviation	Total time (s)
BN2O models (50 models)			
min-width*	82	23916	0.62
min-fill*	84	1322	0.45
lim.-br.-2 DFS	94	53	35.90
lim.-br.-3 DFS	98	27	85.70
lim.-br.-4 DFS	100	0	163.22
lim.-mem-100 BFS	96	2	772.84
lim.-mem-1k BFS	100	0	2115.32
Bayesian networks from the BN repository (21 models)			
min-weight*	0	2372	0.05
min-width*	5	1408	0.33
min-fill*	10	931	0.20
lim.-br.-2 DFS	19	115	3.46
lim.-br.-3 DFS	62	76	25.94
lim.-br.-4 DFS	86	6	178.82
lim.-mem-100 BFS	38	261	24.01
lim.-mem-1k BFS	67	29	170.86
DFS	100	0	78737.33

results are included only 21 of these models for which all tested heuristics and depth-first search returned results in less than one day (notice that no initial subgraph decomposition was done).

From the results for BN2O networks we can conclude that min-fill* and min-width* are quite often good heuristics, but that their induced search spaces are too small to avoid triangulations that are far from optimal. The new heuristics seem to avoid this pitfall. Even though the heuristics from Table 3 are not near as fast as normal greedy heuristics, they do provide some significant insights. In particular, the results tell us which part of the search graph that is most beneficial to explore and which parts we may well leave uninvestigated. This means that we can direct the sampling of randomized algorithms (e.g., simulated annealing) to these parts of the search space where an optimal or near-optimal solution is very likely to exist.

First we shall discuss the results of the BN2O networks; notice that such models only have binary variables. Therefore min-width actually corresponds to the commonly used min-weight heuristic. The graph where min-width* found an exceedingly poor triangulation has 40 vertices and a density around 0.36. The total table size for min-width* was 595, 634, 176, and this means that no stochastic (breaking ties randomly) min-width heuristic can yield a triangulation that requires below some 2.4 GB of memory (assuming 4 bytes for a `float`). Contrast this with the optimal triangulation which leads to a memory requirement of only 10 MB. Similarly to min-width*, in our experiments there was a different graph where min-fill* found an somewhat poor triangulation. It has also 40 vertices and its density is 0.36. The optimal triangulation uses less than 10 MB of memory whereas the one found by min-fill* uses more than 120 MB of memory.

If we look at the results in the second part of Table 3, we see that min-weight* is actually the worst heuristic. This must be considered somewhat surprising as min-weight (with breaking ties randomly) is generally considered a good heuristic; however, the running time does suggest that the search space for min-weight* is somewhat smaller than for min-fill* and min-width* for our test graphs, which might be a consequence of a smaller number of ties to choose from at each step. Furthermore, notice that our results do not say anything about the cardinality of the best total table size found by these heuristics. It may be that min-fill* attains its best value only in a single elimination order, and that all other elimination orders lead to a total table size that is worse than most elimination orders investigated by min-weight* (if one wanted to investigate this, best-first search may be modified for such a purpose). We would also like to highlight a few of the results from the table. The graph where min-weight*, min-width*, and min-fill* found an exceedingly poor triangulation has 32 vertices and a density around 0.39. The total table size for min-weight* was $1,913,187,406,080$, and this means that no stochastic (breaking ties randomly) min-weight heuristic can yield a triangulation that requires below some 7700 GB of memory. Contrast this with the optimal triangulation which leads to a memory requirement of 310 GB.

7.3. *Discussion*

The fact that depth-first search came out nearly as fast as best-first search (sometimes even faster) must be considered a surprise. We believe that the main reason for this is that the pruning via the coalescing map turns out to work quite well—this pruning is the direct cause of the change in complexity from $\Theta(\gamma(|V|) \cdot |V|!)$ to $O(\gamma(|V|) \cdot |V|!)$. The experiments indicate that best-first search actually runs in $O(\gamma(|V|) \cdot 2^{|V|})$ time. Secondly, it is worth mentioning that depth-first search only needs very few (otherwise expensive) free-store allocations.

The results of the comparison between depth-first search and best-first search should not be taken as a universal truth. There are still improvements that can be made to both methods which may change the results in both directions.

For depth-first search we can think of two improvements. First, to further improve the pruning by the coalescing map, then we should consider a hybrid best-first-depth-first scheme where we explore the most promising paths earlier. Secondly, it should be possible to *guarantee* an $O(\gamma(|V|) \cdot 2^{|V|})$ complexity bound by storing (in the coalescing map) the total table size increase from the current node to the goal node. This total table size increase can be put into the coalescing map when depth-first search returns from a recursive call.

For best-first search the most obvious enhancement would be to investigate special cache- and virtual memory friendly data structures for the priority queue. This could have large impact on performance. Secondly, [32] uses a depth-first search from a frontier-node to establish a lower-bound for the node in question. Such a technique can reduce the space requirements for best-first-search exponentially in the look-ahead depth employed.

Having the above discussion in mind, we still believe that depth-first search performs surprisingly well, and that it is worth reconsidering for the minimum treewidth criterion. In other words, the work of [8] would become very interesting when combined with coalescing.

The experiments also reveal that many graphs are very difficult to triangulate optimally using our methods. For the BN20 networks, we could also generate instances that were too hard to solve in reasonable time. It appears to us that there is no direct connection between the size and density of the graph and how hard it may be to triangulate with the technique presented in this paper. Rather, a hard instance is characterized by having a narrow distribution of the total table size over the different elimination orders. This forces the exhaustive search algorithms to explore almost the whole search space. The method of Shoikhet and Geiger [13] uses decomposition based on minimal separators, and the well-known Hugin tool builds on this idea while adopting it to minimize total table size instead of treewidth [33]. Each minimal separator is used to split the graph into so-called fragments which must all be optimally triangulated. These methods are therefore sensitive to number of minimal separators rather than the narrowness of the distribution over total table sizes. In this sense we believe that our method complements existing techniques. We leave it as further research to compare these techniques to ours, and since these techniques also need to find optimal triangulations of fragments, it would be interesting to see if the two approaches may be integrated.

8. Conclusion

This paper contains several contributions. First, we have described a new method for maintaining the cliques of a dynamic graph and furthermore described simple ways to improve Stix' method. The new method works by employing a local search for cliques, and this local search can in principle be done by any existing clique search algorithm. The new method is both simpler and more generic than previous methods, and experiments show that the new method performs better than the improved Stix method, especially when adding a set of fill-in edges. The second best algorithm when adding fill-in edges was a full recomputation via the Bron–Kerbosch algorithm with pivot, but for graph densities below 0.1, the new method clearly wins. It follows that for large and sparse graphs, the new method can become significantly faster (especially when considering the two non-implemented optimizations about per-vertex clique lists and fallback on Bron–Kerbosch when fa(U,G) becomes large). We also found that we were unable to reproduce the results reported by Stix on our test graphs, that is, Stix' method was slower than Bron–Kerbosch in many cases.

Secondly, we have described new methods for finding optimal total table size triangulations of undirected graphs. Experiments revealed that the methods rely heavily on efficient dynamic maintenance of the cliques of the partially triangulated graphs. These triangulation methods are mainly supposed to be used off-line, but they may also be transformed into any-time heuristics.

Third, experiments show that depth-first search is almost as fast as best-first search which was quite unexpected. The main reason is that we use pruning based on a coalescing map which lowers the time complexity from $\Theta(\gamma(n) \cdot n!)$ to $O(\gamma(n) \cdot n!)$ (n being the number of vertices in the graph and $\gamma(n)$ being the per-node overhead). From the experiments we can infer that this pruning is so effective that depth-first search actually runs in $O(\gamma(n) \cdot 2^n)$ time. Therefore we believe that it will be beneficial to reconsider depth-first search for triangulation with the minimum treewidth criterion [8] while employing coalescing.

Fourth, we have examined the best-case quality of common heuristic algorithms on two different sets of graphs. The experiments show that these heuristics will never be able to guarantee good triangulations on all types of graphs, for example, on one model the min-width (and min-weight) heuristic would (in the best case) return a triangulation that requires at least 2.4 GB of memory whereas the optimal solution requires only 10 MB. This shows that off-line triangulation methods can be very beneficial in some cases.

Fifth, the results from heuristics with branching tell us which part of the search space is most beneficial to explore and which parts we may well leave uninvestigated. In this way we can direct the sampling of randomized algorithms to these parts of the search space where an optimal or near-optimal solution is very likely to exist.

Acknowledgement

The authors would like to thank Frank Jensen and the anonymous reviewers of PGM 2010 and of the IJAR paper for their helpful comments. T. Ottosen would like to thank Finn V. Jensen and ÚTIA for hosting him during Spring 2009.

References

[1] J. Pearl, Bayesian networks: a model of self-activated memory for evidential reasoning, in: Proceedings, Cognitive Science Society, UC Irvine, 1985, pp. 329–334.
[2] F.V. Jensen, F. Jensen, Optimal junction trees, in: In UAI, Morgan Kaufmann, 1994, pp. 360–366.
[3] U. Kjærulff, Triangulation of graphs—algorithms giving small total state space, Tech. Rep. R-90-09, Aalborg University, March 1990.
[4] M. Yannakakis, Computing the minimum fill-in is NP-complete, SIAM Journal on Algebraic and Discrete Methods 2 (1) (1981) 77–79.
[5] W. Wen, Optimal decomposition of belief networks, in: Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence (UAI-90), Elsevier Science, New York, NY, 1990, pp. 209–224.
[6] F.T. Ramos, F.G. Cozman, Anytime anyspace probabilistic inference, International Journal of Approximate Reasoning 38 (1) (2005) 53–80.
[7] P.A. Dow, R.E. Korf, Best-first search for treewidth, in: AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence, AAAI Press, 2007, pp. 1146–1151.
[8] V. Gogate, R. Dechter, A complete anytime algorithm for treewidth, in: Proceedings of the Proceedings of the Twentieth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-04), AUAI Press, Arlington, Virginia, 2004, pp. 201–208.
[9] H.L. Bodlaender, A.M. Koster, F. van den Eijkhof, Preprocessing rules for triangulation of probabilistic networks, Computational Intelligence 21 (2005) 286–305.
[10] R.E. Tarjan, Decomposition by clique separators, Discrete Mathematics 55 (2) (1985) 221–232.
[11] H.-G. Leimer, Optimal decomposition by clique separators, Discrete Mathematics 113 (1–3) (1993) 99–123.
[12] M.J. Flores, J.A. Gámez, Triangulation of Bayesian networks by retriangulation, International Journal of Intelligent Systems 18 (2003) 153–164.
[13] K. Shoikhet, D. Geiger, A practical algorithm for finding optimal triangulations, in: AAAI'97: Proceedings of the 14th national conference on Artificial intelligence, AAAI Press, 1997, pp. 185–190.
[14] M.J. Flores, J.A. Gámez, A review on distinct methods and approaches to perform triangulation for Bayesian networks, Advances in Probabilistic Graphical Models (2007) 127–152.
[15] T.J. Ottosen, J. Vomlel, All roads lead to rome—new search methods for optimal triangulations, in: P. Myllymäki, T. Roos, T. Jaakkola (Eds.), Proceedings of the Fifth European Workshop on Probabilistic Graphical Models (PGM-2010), 2010, pp. 201–208.
[16] T.J. Ottosen, J. Vomlel, Honour thy neighbour—clique maintenance in dynamic graphs, in: P. Myllymäki, T. Roos, T. Jaakkola (Eds.), Proceedings of the Fifth European Workshop on Probabilistic Graphical Models (PGM-2010), 2010, pp. 209–216.
[17] T. Ohtsuki, L.K. Cheung, T. Fujisawa, Minimal triangulation of a graph and optimal pivoting order in a sparse matrix, Journal of Mathematical Analysis and Applications 54 (1976) 622–633.
[18] C.D. Bartels, J.A. Bilmes, Creating non-minimal triangulations for use in inference in mixed stochastic/deterministic graphical models, Machine Learning (2011) 1–41.
[19] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, 1972, pp. 85–103.
[20] J.W. Moon, L. Moser, On cliques in graphs, Israel Journal of Mathematics 3 (1965) 23–28.
[21] V. Stix, Finding all maximal cliques in dynamic graphs, Computational Optimization and Applications 27 (2) (2004) 173–186.
[22] S. Butenko, W.E. Wilhelm, Clique-detection models in computational biochemistry and genomics, European Journal of Operational Research 173 (2005) 1–17.
[23] A. Chateau, P. Riou, E. Rivals, Approximate common intervals in multiple genome comparison, in: F.-X. Wu, M.J. Zaki, S. Morishita, Y. Pan, S. Wong, A. Christianson, X. Hu (Eds.), BIBM, IEEE, 2011, pp. 131–134.
[24] C. Bron, J. Kerbosch, Algorithm 457: Finding all cliques of an undirected graph, Communication of ACM 16 (9) (1973) 575–577.
[25] F. Cazals, C. Karande, A note on the problem of reporting maximal cliques, Theoretical Computer Science 407 (1–3) (2008) 564–568.
[26] I. Koch, Enumerating all connected maximal common subgraphs in two graphs, Theoretical Computer Science 250 (1–2) (2001) 1–30.
[27] E. Harley, A. Bonner, N. Goodman, Uniform integration of genome mapping data using intersection graphs, Bioinformatics 17 (6) (2001) 487–494.

[28] E. Tomita, A. Tanaka, H. Takahashi, The worst-case time complexity for generating all maximal cliques and computational experiments, Theoretical Computer Science 363 (1) (2006) 28–42.
[29] A. Darwiche, Modelling and Reasoning with Bayesian Networks, Cambridge University Press, 2009.
[30] V. Stix, Target-oriented branch and bound method for global optimization, Journal of Global Optimization 26 (2003) 261–277.
[31] P. Savicky, J. Vomlel, Triangulation heuristics for BN2O networks, in: C. Sossai, G. Chemello (Eds.), Proceedings of the 10th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, Springer, 2009, pp. 566–577.
[32] R. Stern, T. Kulberis, A. Felner, R. Holte, Using lookaheads with optimal best-first search, in: AAAI Proceedings, 2010.
[33] Hugin, Hugin API Reference Manual, Hugin Expert A/S, seventh ed., 2012.