All Roads Lead To Rome—New Search Methods for Optimal Triangulation

Thorsten J. Ottosen
Department of Computer Science, Aalborg University, Denmark
nesotto@cs.aau.dk

Jiří Vomlel
Institute of Information Theory and Automation of the AS CR, The Czech Republic
vomlel@utia.cas.cz

Abstract

To perform efficient inference in Bayesian networks, 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. The algorithms 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.

1 Introduction

We consider the problem of finding optimal cost 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: computing a minimum fill-in is NP-complete (Yannakakis, 1981) and finding a triangulation with minimal total table size is NP-hard (Wen, 1990).

There are several issues that motivates 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 is tractable. Finally, off-line methods can often be turned into good any-time heuristics.

Previous research on triangulation has also used best-first search (Dow and Korf, 2007) and depth-first search (Gogate and Dechter, 2004), 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 cri-

Pp. 209–217 in *Proceedings of the Fifth European Workshop on Probabilistic Graphical Models (PGM-2010)*, edited by Petri Myllymäki, Teemu Roos and Tommi Jaakkola. HIIT Publications 2010–2.

terion, one can continuously apply the preprocessing rules of (Bodlaender et al., 2005), 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 triangulating using decomposable subgraphs dates back to (Tarjan, 1985). 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) (Leimer, 1993). 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 (Flores and Gámez, 2003).

In (Shoikhet and Geiger, 1997) 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 (Flores and Gámez, 2007).

2 Preliminaries

We shall use the following notation and definitions. $G = (V, E)$ is an *undirected graph* with *vertices* $V = \mathcal{V}(G)$ and *edges* $E = \mathcal{E}(G)$. For a set of edges F, $\mathcal{V}(F)$ is the set of vertices $\{u, v : \{u, v\} \in F\}$. An undirected graph is *triangulated* (or chordal) if every cycle of length greater than 3 has a chord. For example, in Figure 1 the graph on the left is not triangulated whereas the graph on the right is triangulated. For $W \subseteq V$, $G[W]$ is the *subgraph induced by* W. 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. We denote the set of all triangulations of a graph G for $\mathcal{T}(G)$.

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* $nb(v, G)$ of a vertex $v \in V$ is the set $W \subseteq V$ such that each $u \in W$ is connected to v. The *family* $fa(v, G)$ of a vertex v is the set $nb(v, G) \cup \{v\}$, and the neighbours and family of a set of vertices is defined similarly. The *elimination* of a vertex $v \in V$ of $G = (V, E)$ is the process of removing v from G and making $nb(v, G)$ a complete set. This process induces a new graph $H = (V \setminus \{v\}, E \cup F)$ where F is the set of *fill-in edges*. For example, in Figure 1 (left), eliminating the vertex 6 induces the two fill-in edges shown with dotted edges in the adjacent graph. 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 all vertices are eliminated in G according to an elimination order f, the union of all the fill-in edges produced induces a triangulation of G. In this way each 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.

Given $G = (V, E)$, a set of vertices $C \subseteq V$ is a *clique* if it is a maximal complete set and $\mathcal{C}(G)$ is the set of all cliques in G. The *table size* of a clique C is given by $ts(C) = \prod_{v \in C} |sp(v)|$ where $sp(v)$ denotes the state space of the variable corresponding to v in the Bayesian network. Finally, the *total table size* of a graph H is given by $tts(H) = \sum_{C \in \mathcal{C}(H)} ts(C)$.

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 and min-width which both greedily pick the next vertex to eliminate based on a local 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.

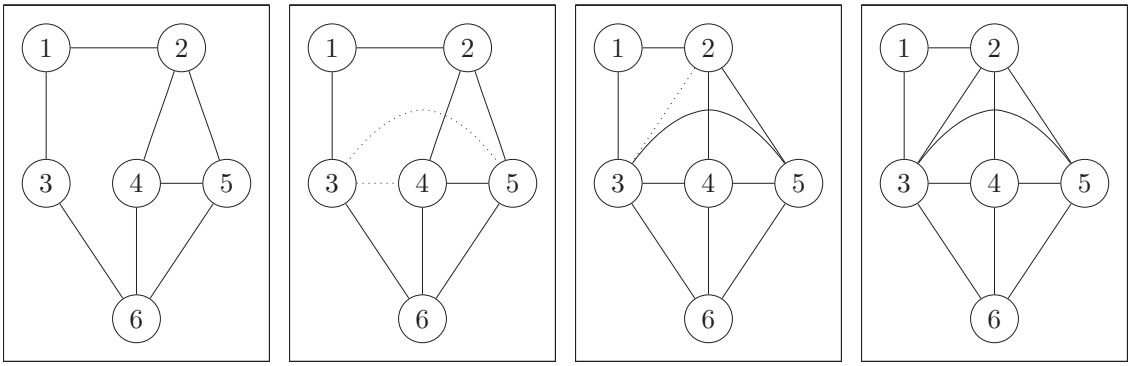

Figure 1: Example of the fill-in edges and partially triangulated graphs induced by an elimination order that starts with the sequence $\langle 6, 4 \rangle$: the dotted edges are fill-in edges. Left: the initial graph. Middle left: the fill-in edges induced by eliminating vertex 6. Middle right: the fill-in edges induced by eliminating vertex 4. Right: the final triangulated graph.

3 The Search Space for Triangulation Algorithms

Our goal is to explore the space $\mathcal{T}(\mathrm{G})$ encoding all possible 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 labelled 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 5 we shall explain the algorithms in detail.

Example 1. Consider the graphs in Figure 1 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 has 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.

Observe that 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 5 are admissible.

4 Dynamic Clique Maintenance

To compute the cliques of a graph associated with a node in the search graph, we may use a standard algorithm for this task, for example, the well-known Bron-Kerbosch algorithm (Cazals and Karande, 2008). However, as we

can see from the above example, each path in the search graph corresponds to a sequence of graphs where the difference between adjacent graphs is quite small. Therefore we may exploit this similarity among adjacent graphs to avoid the quite expensive recomputation of the cliques and total table size of the graphs.

(Stix, 2004) investigated this problem, however, his method leads to many redundant computations when the added edges appear close together (as is the case for fill-in edges). Therefore we give a new algorithm that performs significantly faster for this type of update.

The general idea behind this method is simple: instead of searching for cliques in the whole graph, simply run a clique enumeration algorithm on a smaller subgraph where all the new cliques appear and existing cliques disappear. This algorithm for dynamic clique maintenance is presented as Algorithm 1, and as a side-effect it also updates the total table size and the current graph. This implies that the total table size does not need to be computed from scratch either. In our case we employ Bron-Kerbosch for the local search in `FindCliques(·)`. Its correctness follows from the following result.

Theorem 1. (Ottosen and Vomlel, 2010). *Let* $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *be an undirected graph, and let* $\mathrm{G}' = (\mathrm{V}, \mathrm{E} \cup \mathrm{F})$ *be the graph resulting from adding a set of new edges* F *to* G. *Let* $\mathrm{U} = \mathcal{V}(\mathrm{F})$. *The cliques of* $\mathcal{C}(\mathrm{G}')$ *can be found by removing the cliques from* $\mathcal{C}(\mathrm{G})$ *that intersect with* U *and adding cliques of* $\mathrm{G}'[\mathrm{fa}(\mathrm{U}, \mathrm{G}')]$ *that intersect with* U.

(Xiang and Lee, 2006) describes a set of vertices called a *cruz* which is central to their method for learning. The method described above may also be used to efficiently determine the cruz.

Algorithm 1 Incremental maintenance of cliques and total table size by local search

```
1:  procedure IncrUpdate(&G, &C, &tts, F)
2:      Input: A graph G = (V, E),
3:             the set of cliques C of G,
4:             the total table size tts of G, and
5:             the set of new edges F.
6:      Set G = (V, E ∪ F)
7:      Let U = V(F)
8:      Let C^new = FindCliques(G, fa(U, G))
9:      for all C ∈ C do  ▷ Remove old cliques
10:         if C ∩ U ≠ ∅ then
11:             Set tts = tts - ts(C)
12:             Set C = C \ {C}
13:         end if
14:     end for
15:     for all C ∈ C^new do ▷ Add new cliques
16:         if C ∩ U ≠ ∅ then
17:             Set tts = tts + ts(C)
18:             Set C = C ∪ {C}
19:         end if
20:     end for
21: end procedure
```

5 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 $\mathcal{T}(\mathrm{G})$, 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 node 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 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 $\Theta(|\mathrm{V}|)$ memory for a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$. 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 ∪ 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 \ 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 4 . 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 $O(\beta(|\mathrm{V}|)\cdot|\mathrm{V}|!)$ (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 \ W] is the same no matter what order the vertices in W have been eliminated in, so we can use *coalescing* of nodes and thus reduce the worst case complexity to $O(\beta(|\mathrm{V}|)\cdot 2^{|\mathrm{V}|})$ (Darwiche, 2009).

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

Both $\gamma(|\mathrm{V}|)$ and $\beta(|\mathrm{V}|)$ take at least $O(|\mathrm{V}|^3)$ time as they are dominated by the removal of simplicial vertices (the lookup into the coalescing map takes $O(|\mathrm{V}|)$ time due to the computation of the hash-key, and the priority queue look-up for best-first search may take $O(|\mathrm{V}|)$ 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.

In Algorithm 2 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 3. 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 4). The procedure `EliminateSimplicial(·)` removes all simplicial vertices from the remaining graph.

Algorithm 2 Best-first search with coalescing

```
function TriangulationByBFS(G)
    Let s = (G, G, C(G), tts(G), ⟨⟩)
    EliminateSimplicial(s)
    if |V(s.R)| = 0 then
        return s
    end if
    Let map = ∅                          ▷ Coalescing map
    Let O = {s}                          ▷ The open set
    while O ≠ ∅ do
        Let n = arg min_{x∈O} x.tts
        if |V(n.R)| = 0 then
            return n
        end if
        Set O = O \ {n}
        for all v ∈ V(n.R) do
            Let m = Copy(n)
            EliminateVertex(m, v)
            EliminateSimplicial(m)
            if map[m.R].tts ≤ m.tts then
                continue
            end if
            Set O = O \ {map[m.R]}
            Set map[m.R] = m
            Set O = O ∪ {m}
        end for
    end while
end function
```

6 Results

In this section we describe experiments with the optimal methods as well as several heuristics derived from these. For that purpose we have generated 50 random graphs with varying size and density. In this paper we have only performed experiments on bipartite graphs—these graphs

Algorithm 3 Depth-first search with coalescing and upper-bound pruning

```
function TriangulationByDFS(G)
    Let s = (G, G, C(G), tts(G), ⟨⟩)
    EliminateSimplicial(s)
    if |V(s.R)| = 0 then
        return s
    end if
    Let best = MinFill(s)          ▷ Best path
    Let map = ∅                    ▷ Coalescing map
    ExpandNode(s, best, map)
    return best
end function
procedure ExpandNode(n, &best,&map)
    for all v ∈ V(n.R) do
        Let m = Copy(n)
        EliminateVertex(m, v)
        EliminateSimplicial(m)
        if |V(m.R)| = 0 then
            if m.tts < best.tts then
                Set best = m
            end if
        else
            if m.tts ≥ best.tts then
                continue
            end if
            if map[m.R].tts ≤ m.tts) then
                continue
            end if
            Set map[m.R] = m
            ExpandNode(m, best, map)
        end if
    end for
end procedure
```

result from the application of rank-one decomposition to BN2O networks—see (Savicky and Vomlel, 2009) for details. The main reason for using these graphs is that they are among the most difficult to triangulate. This is because (1) moralization should not be applied after using rank-one decomposition, and (2) 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.

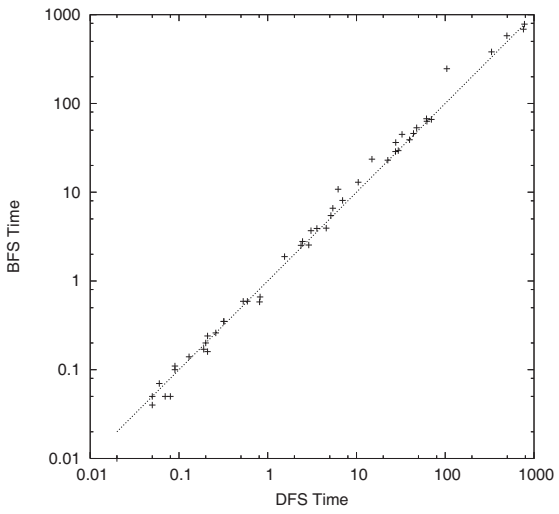

Figure 2: Comparison of the running time of best-first search and depth-first search. Both algorithms terminate with an optimal triangulation (values above the line indicate depth-first search was faster).

We have performed two different tests on this dataset: (1) a comparison of depth-first search and best-first search, and (2) a comparison between heuristic methods. For Test 2 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 and min-fill heuristics implemented so that a successor node is generated for all ties instead of breaking ties randomly. We refer to these algorithms as *min-width** and *min-fill**, respectively.

The results from Test 1 are given in Figure 2. It appears that best-first search performs better when the computational time is large.

Table 1: 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	DFS	BFS
20 (10)	0.3s	0.3s
30 (20)	9.1s	17.5s
40 (10)	30.6s	31.8s
50 (5)	30.4s	33.0s
60 (5)	591.7s	607.0s
% Fastest	71	37

Table 2: 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 50 graphs.

	% op.	% dev.	time
min-width*	82	23,916	1s
min-fill*	84	1,322	1s
lim.-br.-2 DFS	94	53	83s
lim.-br.-3 DFS	98	27	270s
lim.-br.-4 DFS	100	0	512s
lim.-mem-100 BFS	96	2	2234s
lim.-mem-1k BFS	100	0	6447s

In Table 1 we give summary statistics for this test. We have computed the p-value of the two-sided Wilcoxon two-sample test of the null hypothesis that the distribution of depth-first time minus best-first time is symmetric about 0. The p-value is 0.001069, which means that the null hypothesis is rejected, that is, differences are significant (in favor of depth-first search).

The results from Test 2 are given in the Table 2. Here we have run the heuristics on the 50 graphs from Test 1. From this 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.

Notice that the BN2O networks 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.

7 Discussion

The fact that depth-first search came out as the fastest algorithm 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 $\Theta(\gamma(|\mathrm{V}|)\cdot|\mathrm{V}|!)$ to $O(\gamma(|\mathrm{V}|)\cdot|\mathrm{V}|!)$. The experiments indicate that best-first search actually runs in $O(\gamma(|\mathrm{V}|)\cdot 2^{|\mathrm{V}|})$ time. Secondly, it is worth mentioning that depth-first search only needs very few (otherwise expensive) free-store allocations. 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. In light of this discussion we believe depth-first search should be reconsidered also for the minimum treewidth criterion.

8 Conclusion

The contributions of this paper are three-fold. First, we have described new methods for finding optimal total table size triangulations of undirected graphs. The methods rely heavily on efficient dynamic maintenance of the cliques and total table size of a graph. These methods are mainly supposed to be used off-line, but they may also be transformed into any-time heuristics.

Secondly, experiments show that depth-first search is faster than best-first search—this 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 it will be beneficial to reconsider depth-first search for triangulation with the minimum treewidth criterion.

Third, we have examined the quality of common heuristic algorithms on a set of graphs that are quite difficult to triangulate. 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 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 could be required in some cases.

Acknowledgement

The authors would like to thank the three anonymous reviewers for their helpful comments.

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.

References

Hans L. Bodlaender, Arie M.C.A. Koster, and Frank van den Eijkhof. 2005. Preprocessing rules for triangulation of probabilistic networks. *Computational Intelligence*, 21:286–305.

F. Cazals and C. Karande. 2008. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1-3):564–568, November.

Adnan Darwiche. 2009. *Modelling and Reasoning with Bayesian Networks*. Cambridge University Press.

P. Alex Dow and Richard E. Korf. 2007. Best-first search for treewidth. In *AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence*, pages 1146–1151. AAAI Press.

M. Julia Flores and José A. Gámez. 2003. Triangulation of Bayesian networks by retriangulation. *International Journal of Intelligent Systems*, 18:153–164.

M. Julia Flores and José A. Gámez. 2007. A review on distinct methods and approaches to perform triangulation for Bayesian networks. *Advances in Probabilistic Graphical Models*, pages 127–152.

Vibhav Gogate and Rina Dechter. 2004. A complete anytime algorithm for treewidth. In *Proceedings of the Proceedings of the Twentieth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-04)*, pages 201–208, Arlington, Virginia. AUAI Press.

Hanns-Georg Leimer. 1993. Optimal decomposition by clique separators. *Discrete Math.*, 113(1-3):99–123.

Thorsten J. Ottosen and Jiří Vomlel. 2010. Honour thy neighbour—clique maintenance in dynamic graphs. In *Proceedings of the Fifth European Workshop on Probabilistic Graphical Models*.

Petr Savicky and Jiří Vomlel. 2009. Triangulation heuristics for BN2O networks. In C. Sossai and G. Chemello, editors, *Proceedings of the 10th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 566–577. Springer.

Kirill Shoikhet and Dan Geiger. 1997. A practical algorithm for finding optimal triangulations. In *AAAI'97: Proceedings of the 14th national conference on Artificial intelligence*, pages 185–190. AAAI Press.

Volker Stix. 2004. Finding all maximal cliques in dynamic graphs. *Comput. Optim. Appl.*, 27(2):173–186.

Robert E. Tarjan. 1985. Decomposition by clique separators. *Discrete Mathematics*, 55(2):221 – 232.

Wilson Wen. 1990. Optimal decomposition of belief networks. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence (UAI-90)*, pages 209–224, New York, NY. Elsevier Science.

Y. Xiang and J. Lee. 2006. Learning decomposable markov networks in pseudo-independent domains with local evaluation. *Mach. Learn.*, 65(1):199–227.

Mihalis Yannakakis. 1981. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79.