Thinning a Triangulation of a Bayesian Network or Undirected Graph to Create a Minimal Triangulation

Edmund Jones*

*School of Mathematics, University of Bristol,
University Walk, Bristol, BS8 1TW, UK
Department of Public Health & Primary Care,
University of Cambridge, Worts' Causeway, Cambridge, CB1 8RN, UK
edmundjones79@gmail.com*

Vanessa Didelez

*Leibniz Institute for Prevention Research and Epidemiology — BIPS,
Achterstr. 30, D-28359 Bremen, Germany
didelez@leibniz-bips.de*

Received 7 July 2015
Revised 4 August 2016

In one procedure for finding the maximal prime decomposition of a Bayesian network or undirected graphical model, the first step is to create a minimal triangulation of the network, and a common and straightforward way to do this is to create a triangulation that is not necessarily minimal and then thin this triangulation by removing excess edges. We show that the algorithm for thinning proposed in several previous publications is incorrect. A different version of this algorithm is available in the R package gRbase, but its correctness has not previously been proved. We prove that this version is correct and provide a simpler version, also with a proof. We compare the speed of the two corrected algorithms in three ways and find that asymptotically their speeds are the same, neither algorithm is consistently faster than the other, and in a computer experiment the algorithm used by gRbase is faster when the original graph is large, dense, and undirected, but usually slightly slower when it is directed.

Keywords: Bayesian network; graphical model; minimal triangulation; maximal prime decomposition.

1. Introduction

“Thinning” is one step in a widely used procedure to create maximal prime decompositions of Bayesian networks and undirected graphical models. Formal definitions are given in the subsequent sections, but in summary, for a graph and a set of

*Corresponding author.

This is an Open Access article published by World Scientific Publishing Company. It is distributed under the terms of the Creative Commons Attribution 4.0 (CC-BY) License. Further distribution of this work is permitted, provided the original work is properly cited.

extra edges that have been added to it to create a triangulation, thinning consists of removing excess edges from the set of extra edges to create a minimal triangulation.

This paper presents three algorithms for thinning. Algorithm I has appeared in previous publications^{1–3} but is incorrect, and we give examples where it fails. Algorithms II and III are correct and we give proofs of their correctness. Algorithm III is a corrected version of Algorithm I and Algorithm II is a simpler version of Algorithm III. Algorithm III is performed by a function in the R package gRbase,⁴ which is used by several other packages in the field of graphical models, but there does not seem to have been any published proof that it is correct. Alternative algorithms for minimal triangulation exist that are asymptotically faster or do not involve thinning,⁵ but Algorithms II and III are simpler and easier to implement, which is presumably why gRbase uses Algorithm III. Alternative algorithms are also available for the whole procedure of creating a maximal prime decomposition.^{6,7}

Minimal triangulations and maximal prime decompositions are used in many ways and in many fields. Maximal prime decompositions of Bayesian networks, also known as directed acyclic graphical models, can be used for probabilistic inference on large networks, hybrid propagation algorithms for very large networks that combine exact and approximate methods, incremental model-building, and the development of algorithms for parallel processors.³ They are also used in Bayesian methods for learning the structure of Gaussian graphical models^{8–10} with the generalized hyper inverse Wishart prior distribution on the covariance matrix^{11–14} — in these methods, calculating the posterior probability of a graph requires finding its maximal prime decomposition. Minimal triangulations on their own have numerous applications in other fields, for example sparse matrix computation, expert systems, database management, and computer vision.^{5,15–18}

Section 2 provides basic definitions. Section 3 defines minimal triangulation, presents the elimination algorithm for creating triangulations, and briefly discusses alternative ways to create triangulations that have desirable properties for probabilistic inference. Section 4 defines thinning. Section 5 presents the incorrect Algorithm I and several graphs for which it goes wrong. Sections 6 and 7 present Algorithms II and III respectively, and the proofs of correctness. Section 8 compares the speed of the two correct algorithms in terms of whether one might be consistently faster and in terms of asymptotic complexity, Section 9 compares them by means of a computer experiment, and Section 10 is a discussion.

2. Basic Definitions and Notation

Let (V, E) be a finite graph, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. If the graph is directed then each edge is an ordered pair, usually drawn as an arrow from the first vertex to the second, and if the graph is undirected then each edge is an unordered pair, drawn as a line between the two vertices. Bayesian networks use directed acyclic graphs (DAGs), where acyclic means there is no set of vertices $x_1, \dots, x_k$ such that $(x_1, x_2), \dots, (x_{k-1}, x_k), (x_k, x_1) \in E$. In a directed graph, the parents of a vertex x are $pa(x) = \{y : (y, x) \in E\}$.

In an undirected graph, the neighbours of a vertex x are $ne(x) = \{y \in V : \{x, y\} \in E\}$ and the degree of x is $deg(x) = |ne(x)|$. An induced subgraph is a graph (V', E') where $V' \subseteq V$ and $E' = \{\{x, y\} \in E : x, y \in V'\}$; if $G = (V, E)$ then (V', E') is also written as $G[V']$. A clique is a complete subgraph, which means a graph (V', E') such that $V' \subseteq V$, $E' \subseteq E$, and $\{x, y\} \in E'$ for all $x, y \in V'$.

An undirected graph (V, E) is said to be triangulated if every cycle of length four or more contains a chord. More formally, for (V, E) to be triangulated means that if $\{x_1, x_2\}, \{x_2, x_3\}, \dots, \{x_{k-1}, x_k\}, \{x_k, x_1\} \in E$, all the x_i 's are distinct, and $k \geq 4$, then there must be an edge $\{x_i, x_j\}$ where $i, j \in \{1, \dots, k\}$ and x_i and x_j are not adjacent in the cycle. The property of being triangulated is equivalent to the property of being decomposable,¹⁹ and this is the term normally used in statistics. In mathematical graph theory these graphs are called chordal.^{20,21} They have also been called rigid circuit graphs, monotone transitive graphs, and perfect elimination graphs.^{15,22,23}

3. Minimal Triangulation

Olesen & Madsen³ give a sequence of steps to find the maximal prime decomposition of a DAG and represent it as a junction tree, also known as a join tree. (Maximal prime decomposition itself is not the topic of this paper but definitions can be found in numerous publications.^{3,6,11,24,25}) The first step is moralization: for each vertex x , add an edge between any pair of parents of x that do not already have an edge between them; and then replace all directed edges by undirected edges. Maximal prime decomposition for undirected graphs is the same as for directed graphs, except that the moralization step is omitted.

The next two steps are to find a triangulation and then a minimal triangulation. These are defined as follows. For an undirected graph (V, E) , a triangulation is a set of extra edges $T \subseteq V \times V$ (often called fill edges, fill-in edges, or triangulation edges) such that $(V, E \cup T)$ is triangulated and $E \cap T = \emptyset$. A minimal triangulation is one such that $(V, E \cup T \setminus \{t\})$ is not triangulated for any $t \in T$. Minimum triangulations, on the other hand, are triangulations for which there are no triangulations with fewer edges; these are also useful⁵ but finding them is NP-hard.²⁶

The triangulation is found by an algorithm called elimination.^{5,27} This works as follows. Put the vertices in some order, called the elimination ordering; for each vertex in turn, add edges as necessary to make all the neighbours of the vertex be connected to each other, and then remove the vertex and all its incident edges. The triangulation consists of all the edges that are added during this loop.

There are many ways to choose the elimination ordering. One popular choice is minimum-degree ordering, where at each step a vertex is chosen from among the vertices that have the smallest degree. (This ordering is sometimes called minimum size, because it minimizes the size of the clique that results when the vertex is eliminated, or minimum width.^{1,28}) Elimination with this choice of ordering is known

as minimum-degree elimination and often produces minimal triangulations straight away, but not always.²⁹

Another choice is minimum-fill ordering, where at each step a vertex is chosen from among the vertices whose elimination would produce the smallest number of extra edges.^{28,30} A third choice is minimum-weight elimination, where the vertex is chosen so as to minimize the weight of the resulting clique; the weight of a clique is the product of the numbers of states of its vertices in the Bayesian network.^{1,28} This ordering is often used when the goal is to create a junction tree for probabilistic inference, because the computational complexity of the inference is proportional to the sum of the clique weights. This sum is known as the total table size, and several algorithms have recently been published whose purpose is to find a triangulation that minimizes it.^{28,31–34}

4. Thinning

The second step in finding a minimal triangulation is thinning, which means removing excess edges from the triangulation until no more of them can be removed. Olesen & Madsen's algorithm for thinning originates in Kjaerulff (1993),² but the relevant material also appears in Kjaerulff (1990),¹ so we refer to the earlier publication. We present this algorithm as Algorithm I in Section 5, together with examples to show that it is incorrect.

The R package gRbase contains a function called `minimalTriang` that creates a minimal triangulation by getting a triangulation, either from the user or by calling another function, and then performing thinning. The only source cited in the documentation is Olesen & Madsen,³ but `minimalTriang` actually performs Algorithm III, not Algorithm I. (This information is correct as of August 2016 and gRbase version 1.7-5. The actual code for Algorithm III is in a function called `.minimalTriang` and can be seen by loading the package and typing `gRbase:::.minimalTriang` or by downloading the source code.)

5. Algorithm I

In all three algorithms, (V, E) is the given graph, T is the triangulation, and $G = (V, E \cup T)$ is the triangulated version of the graph. The input to the algorithms is the initial values of (V, E) and T . Both T and G always change at the same time, so it is always true that $G = (V, E \cup T)$.

Kjaerulff and Olesen & Madsen both present essentially the same algorithm for thinning, and they both call it “recursive thinning”, though only Kjaerulff's version is actually recursive, meaning that it calls itself. Algorithm I as shown here is the version from Olesen & Madsen, since this is easier to understand. (Kjaerulff's algorithm starts with $G = (V, E \cup T)$, whereas Olesen & Madsen's starts with $G = (V, E)$, but this seems to be just a minor error in the latter.)

Fig. 1. A graph, shown with solid lines, and a triangulation of it, shown with dashed lines. Algorithm I does not work on this triangulation.

Algorithm I (incorrect)

- (1) Set $R = T$
- (2) Repeat:
 - (3) Set $U = \{\{x, y\} \in R : G[ne(x) \cap ne(y)] \text{ is complete}\}$
 - (4) Set $T = T \setminus U$ (and update G , which is $(V, E \cup T)$)
 - (5) Set $R = \{e_1 \in T : \exists e_2 \in U \text{ such that } e_1 \cap e_2 \neq \emptyset\}$
- (6) Until $U = \emptyset$
- (7) Return T

In Algorithm I, U is the set of edges that get removed on the current iteration of the Repeat loop.

The problem with Algorithm I is that it removes more than one edge at a time, instead of updating the graph after each individual edge-removal and checking whether the other edges can still be removed. Algorithms II and III are similar to Algorithm I but work correctly because they update T and G after each individual edge-removal.

Figure 1 shows a graph and a triangulation for which Algorithm I does not work. In the first iteration of the loop, U consists of the two extra edges (the dashed lines in the figure), and so the algorithm removes both extra edges. This is obviously wrong, because the resulting graph is not triangulated. The graph in Fig. 1 cannot result from moralizing a DAG (proof omitted), but it can arise as an undirected graphical model. The triangulation in Fig. 1 arises if the elimination ordering starts with the vertex at the top. (Kjaerulff assumed that triangulations are always produced by elimination but did not make any restrictions on the elimination ordering; Olesen & Madsen did not even explicitly require that the triangulation must have been produced by elimination.)

Algorithm I can also fail when the original graph is a DAG. An example is shown in Fig. 2. In this example the triangulation is made by minimum-degree elimination. Firstly the moralization is done, in the usual way, as shown in the figure. The elimination ordering then starts with the bottom and top vertices in the central hexagon, in that order. When the second vertex is eliminated, the two extra edges are added; for all the remaining vertices, no extra edges are added when they are eliminated (so the third and subsequent eliminations are not shown in the figure). Algorithm I is then applied to the triangulated graph (that is, the second

Fig. 2. A DAG that shows how Algorithm I goes wrong when minimum-degree elimination is used. If moralization is performed on the DAG, followed by minimum-degree elimination, followed by Algorithm I, then Algorithm I fails, because it removes the two extra edges (shown by dashed lines) at the same time. Only the first two vertex-eliminations are shown; the others do not produce any extra edges.

graph in the figure plus the extra edges shown in the fourth graph), and it wrongly removes both extra edges in the first iteration of the loop — in a very similar way to how it wrongly removes the two extra edges in Fig. 1.

Figure 2 also serves as an example of how Algorithm I can fail when the triangulation is produced by minimum-weight elimination, because if all the vertices have the same state-spaces, then minimum-weight elimination is the same as minimum-degree elimination. The second graph in Fig. 2 serves as an example of how Algorithm I can fail when the original graph is undirected and minimum-degree elimination is used.

Figure 3 shows how Algorithm I can go wrong when the triangulation is made by minimum-fill elimination. Again only the essential steps are shown. The elimination ordering starts with eleven vertices whose eliminations produce no extra edges, and then the top vertex in the central hexagon, whose elimination produces two extra edges. When the remaining vertices are eliminated, extra edges are added within the octagon at the left and within the octagon at the right (these extra edges have multiple possible positions and are not shown), but not between the two octagons. When Algorithm I is applied to the triangulated graph it wrongly removes the two extra edges that are shown in the figure.

6. Algorithm II

We present Algorithm II next since the proof of correctness for Algorithm III uses the proof for Algorithm II. Algorithm II consists of a For loop nested within a Repeat loop. The For loop checks each extra edge in turn and removes it if possible, and the Repeat loop does the For loop over and over until no extra edges are removed. The Repeat loop is necessary because, depending on the order in which the edges are checked, the For loop might not remove all the edges that can be removed (this is illustrated in the left part of Fig. 4).

Algorithm II

- (1) Put the edges in T in some arbitrary order
- (2) Repeat:
 - (3) For each edge $\{x, y\} \in T$ in turn, in order:
 - (4) If $G[ne(x) \cap ne(y)]$ is complete
 - (5) Remove $\{x, y\}$ (from T and G)
 - (6) Until “no edges were removed this time”
- (7) Return T

We present a preliminary proposition (which is similar to Kjaerulff’s Corollary 4¹) and then a proof that Algorithm II is correct. Assuming that $G = (V, E \cup T)$ is triangulated, we say that the edge $\{x, y\} \in T$ is “removable” from the current T if removing it preserves the property that G is triangulated — in other words, $\{x, y\} \in T$ is removable if $(V, E \cup T \setminus \{x, y\})$ is triangulated. The proposition

Fig. 3. A DAG that shows how Algorithm I goes wrong when minimum-fill elimination is used. When moralization is performed, followed by minimum-fill elimination, followed by Algorithm I, then Algorithm I wrongly removes the two extra edges in the figure (shown by dashed lines). Only the eliminations of the first twelve vertices are shown; the other eliminations produce extra edges within each of the two octagons, but not from one octagon to the other (one reason for not showing these is that the extra edges have multiple possible positions).

refers to the condition in line 4 of Algorithm II and shows that the For loop simply checks each edge in T in turn and removes the edge if and only if it is removable (updating the graph after each removal).

Proposition. $\{x, y\} \in T$ is removable if and only if $G[ne(x) \cap ne(y)]$ is complete.

Proof. Suppose $G[ne(x) \cap ne(y)]$ is not complete. There must be vertices a and b such that $\{\{x, a\}, \{y, a\}, \{x, b\}, \{y, b\}\} \subseteq E \cup T$ and $\{a, b\} \notin E \cup T$. Removing $\{x, y\}$ would make there be a chordless cycle of length four, $x-a-y-b-x$, which would mean that G would no longer be triangulated. So the condition that $G[ne(x) \cap ne(y)]$ is complete is necessary for $\{x, y\}$ to be removable.

Now suppose that $G[ne(x) \cap ne(y)]$ is complete. Firstly, suppose that removing $\{x, y\}$ makes there be a chordless cycle of length 5 or more. Then there must have been a chordless cycle of length 4 or more before $\{x, y\}$ was removed. But G was triangulated, so this is impossible. Secondly, suppose that removing $\{x, y\}$ causes the appearance of a chordless cycle of length 4. This cycle would have to be of the form $x-a-y-b-x$, for some $\{a, b\} \notin E \cup T$. But this is impossible, because it contradicts the condition that $G[ne(x) \cap ne(y)]$ is complete. So removing $\{x, y\}$ does not lead to the appearance of any chordless cycle of length 4 or more. This shows that $G[ne(x) \cap ne(y)]$ being complete is sufficient for $\{x, y\}$ to be removable. $\square$

Proof of correctness for Algorithm II. It suffices to prove that (a) the final T is a triangulation, (b) this triangulation is minimal, and (c) the algorithm finishes in finite time. Here and in the next section, we use the word “run” to mean a single iteration of the Repeat loop.

(a) G is triangulated to start with, by definition. It is only ever modified by the removal of an edge, and this only ever happens when the condition in line 4 is fulfilled. The proposition therefore shows that G is always triangulated and T is always a triangulation.

(b) On the final run, the algorithm checks the condition in line 4 for every edge in T , and finds that it is not fulfilled for any of them. By the proposition, this means that removing any of the edges in T would make G become untriangulated, which means that the triangulation is minimal.

(c) Let t be the initial number of edges in T . On each run except the last, the Repeat loop removes at least one edge, so the number of runs is at most $t + 1$. On the i th run, the For loop checks all the remaining edges in T , which means at most $t - i + 1$ edges. These are both finite, so the algorithm is certain to finish in finite time. $\square$

7. Algorithm III

Algorithm III is a corrected version of Algorithm I, and like Algorithm II it basically consists of a For loop within a Repeat loop. The relations between the three algorithms are described in more detail after Algorithm III.

Algorithm III

- (1) Put the edges in T in some arbitrary order
- (2) Set $R = T$
- (3) Repeat:
 - (4) Set $B = \emptyset$
 - (5) For each edge $\{x, y\} \in R$ in turn, in order:
 - (6) If $G[ne(x) \cap ne(y)]$ is complete
 - (7) Remove $\{x, y\}$ (from T and G)
 - (8) Add x and y to B
 - (9) Set $R = \{\{x, y\} \in T : x \in B \text{ or } y \in B\}$
- (10) Until $B = \emptyset$
- (11) Return T

In Algorithm III, $R \subseteq T$ is the set of edges that are candidates for removal, and B is the set of vertices at the end of edges that have been removed on this run. R and B both change during the algorithm. Line 9 sets R to be the set of remaining extra edges that share a vertex with one or more of the edges that was removed on this run. Line 5 in Algorithm I does the same thing. There is no analogous line in Algorithm II.

If Algorithms II and III are run in parallel on the same graph and triangulation, they might remove the same edges as each other on every run, or they might not. For an example where they do not, see Fig. 4. On the first run, Algorithm III does not include 1–3 in R , so on the second run it misses the chance to remove this edge.

In Algorithm I, the idea of R is that time can be saved by not checking edges for which the sets of neighbours of the two vertices did not change on the previous run — since even if those edges are checked, they will not be removed.¹ The same idea is used in Algorithm III. But this idea is mistaken, because the sets of neighbours sometimes change from one iteration of the For loop to the next, and if that happens then it may become possible to remove an edge part-way through a run. So Algorithm III works correctly but its attempt to save time does not always work as intended.

This is illustrated in Fig. 4. For both algorithms, when the second run starts, the sets of neighbours for the edge 1–3 are the same as in the first run. But in Algorithm II when the For loop checks 1–3, the neighbours have changed and the edge can be removed. (Figure 4 also illustrates that in Algorithm III it is possible for an edge to be excluded from R but later reappear in it and be removed from T .)

Proof of correctness for Algorithm III. Algorithm III is the same as Algorithm II, except that on each run Algorithm II checks all the edges in T , whereas Algorithm III only checks the edges in R , which is either the same as T or a proper subset of it. When the For loop finishes, B is the set of vertices such that edges incident to them have been removed during the current run. So line 9 has the effect of ensuring that if $\{x, y\} \in T \setminus R$ then neither $ne(x)$ nor $ne(y)$ changed during the

current run, which means that $ne(x) \cap ne(y)$ did not change during the current run. The contrapositive of this is that if $ne(x) \cap ne(y)$ changed during the current run, then $\{x, y\} \in R$. (The converse is not true — sometimes line 9 puts $\{x, y\}$ in R even though $ne(x) \cap ne(y)$ did not change during the current run.)

Line 10 in Algorithm III is the same as line 6 in Algorithm II, and the equivalence of the other parts of the two algorithms is obvious. So it suffices to prove that on the final run of Algorithm III, the edges that are not checked are not removable. The proof will work by considering an edge $\{x, y\}$ that does not get checked on the final run, and looking back through the runs to find the last run where it was checked (and, obviously, found to be not removable). It will be shown that after that $ne(x) \cap ne(y)$ never changed, from which it follows that $\{x, y\}$ is not removable during the final run.

Suppose that $\{x, y\}$ does not get checked on the final run, and say this run was the i th. Just after line 9 on the $(i - 1)$ th run, $\{x, y\}$ must have been in $T \setminus R$. This means that $ne(x) \cap ne(y)$ did not change during the $(i - 1)$ th run.

Either (a) $\{x, y\}$ was checked during the $(i - 1)$ th run, or (b) it was not. If (a), then $G[ne(x) \cap ne(y)]$ must have been found to be incomplete, otherwise $\{x, y\}$ would have been removed from T . Since $ne(x) \cap ne(y)$ did not change during this run, even if $\{x, y\}$ was checked on the final run then $G[ne(x) \cap ne(y)]$ would still be found to be incomplete (since edges are never added to G). In other words, during the final run, $\{x, y\}$ is not removable.

If (b), then $\{x, y\}$ must have been in $T \setminus R$ just after line 9 in the $(i - 2)$ th run. This means that $ne(x) \cap ne(y)$ did not change during the $(i - 2)$ th run. Either (a2) $\{x, y\}$ was checked during the $(i - 2)$ th run, or (b2) it was not. If (a2), then $G[ne(x) \cap ne(y)]$ must have been found to be incomplete during the $(i - 2)$ th run. It is now known that $ne(x) \cap ne(y)$ did not change during the $(i - 2)$ th or $(i - 1)$ th runs. It follows that $G[ne(x) \cap ne(y)]$ is still incomplete after line 9 in the $(i - 1)$ th run, so on the final run $\{x, y\}$ is not removable.

If (b2), then $\{x, y\}$ must have been in $T \setminus R$ just after line 9 in the $(i - 3)$ th run. This reasoning can be continued backwards through the runs. All the edges were checked on the first run, so eventually this search backwards through the runs is certain to find a run where $\{x, y\}$ was checked and $G[ne(x) \cap ne(y)]$ was found to be incomplete; and $ne(x) \cap ne(y)$ has not changed since this run — if it had changed, $\{x, y\}$ would have been put in R and checked on the next run. It follows that $G[ne(x) \cap ne(y)]$ is still not complete after line 9 in the $(i - 1)$ th run, so even if $\{x, y\}$ was checked on the i th run it would not be removed. $\square$

Underlying the proofs of correctness for Algorithms II and III is the result that “if $G \subset H$ for two chordal graphs G and H on the same vertex set, then there is a sequence of edges that can be removed from H one by one, such that the resulting graph after each removal is chordal, until we reach G ”⁵ (see also the original result²³). This explains why it is sensible to remove one edge at a time. Algorithms II and III say nothing about the order in which the edges are checked,

Fig. 4. An undirected graph and a triangulation for which Algorithms II and III do not remove the same extra edges on each run. The solid edges are the graph and the dashed edges are the extra edges. On each run the extra edges are checked in the order 1-2, 1-3, 2-5. Each algorithm also does one final run, which is not shown, where no edges are removed. (The original graph is already triangulated, but this is just an example for illustration.) The figure also illustrates why the Repeat loops are necessary in the two algorithms: not all the removable edges are removed on the first run.

and this order can even be different on different runs. But when writing a computer program it is natural to make it use the same order on every run.

8. Speed and Comparison with Other Algorithms

Algorithms II and III check the extra edges over and over and are clearly not intended to be fast. Their advantage over other algorithms is that they are short and easy to implement; and Algorithm III has been available in gRbase since 2009, which suggests that it is sufficiently fast for many applications. If these two algorithms are not fast enough then it will be necessary to use more sophisticated algorithms such as the one by Yang *et al.*,³⁵ and if a triangulation with minimal total table

size is required then it will be necessary to use an algorithm that achieves that purpose.^{28,31–34} There are other algorithms for thinning and algorithms that find a minimal triangulation in a single step, without thinning.⁵ (Incidentally, Algorithms II and III are more general than most algorithms for thinning, because they do not require the triangulation to have been done by elimination. But triangulation is invariably done by elimination, so this is not a useful advantage.)

Even though Algorithms II and III are not fast, it is still interesting to see whether one of them is faster than the other. Therefore we compare the speeds of Algorithms II and III in three ways: in terms of whether one algorithm might be consistently faster than the other, asymptotically, and (in the next section) by means of a computer experiment.

For the first of these, recall that Algorithm III can be regarded as a refinement of Algorithm II, and the extra lines with R and B are intended to make it faster. However, in the example in Fig. 4, Algorithm II is faster than Algorithm III. Algorithm II can also be faster for a triangulation produced by minimum-degree elimination. An example is shown in Fig. 5 (which is based on the original graph in Fig. 4). One possible minimum-degree elimination begins by eliminating vertices 4, 3, and 1 and creates the same three extra edges as in Fig. 4, so that Algorithms II and III proceed in the same way as in Fig. 4.

Fig. 5. An undirected graph, and a triangulation that can be produced by minimum-degree elimination, for which Algorithm II is faster than Algorithm III. The two algorithms proceed in the same way as in Fig. 4.

On the other hand, Algorithm III is sometimes faster than Algorithm II, even with triangulations produced by minimum-degree elimination. Again consider the graph and the triangulation shown at the top of Fig. 4 or the ones in Fig. 5. But this time suppose the edges are checked in the order 1–3, 1–2, 2–5. On the first run both algorithms remove 2–5, on the second run they both remove 1–2, and on the third run they both remove 1–3. But on the second run, Algorithm III saves time by not checking 1–3. This makes it faster overall, assuming that creating R and B does not take any time. (Kjaerulff recommends checking the edges in the reverse of the order in which they were added during the elimination;¹ it is not clear whether one of the algorithms is consistently faster than the other if this advice is followed.)

Secondly we consider asymptotic speed. Algorithms II and III are intended to be used following the creation of a triangulation by elimination. Most implementations of minimum-degree elimination have an upper bound on their time of $O(|V|^3)$,

though faster ones exist.⁵ Algorithms II and III both run in time $O(|V|^2|T|^2)$: testing whether a subgraph of G is complete takes time $O(|V|^2)$, and the For loop does that test $|T|$ times in the first run of the Repeat loop, at most $|T| - 1$ times in the second, and so on (and the Repeat loop runs at most $|T| + 1$ times). Algorithm III's extra steps with R and B add or save only linear amounts of time. So asymptotically there is no difference in speed between the two algorithms. Before doing elimination and thinning it might be worthwhile to test whether the graph is already triangulated, and this can be done in time $O(|V| + |E|)$.¹⁷

9. Computer Experiment to Compare the Algorithms

To compare the speed of Algorithms II and III empirically we created functions to perform them in R. These were simplified versions of the `minimalTriang` function. We ran these two functions on undirected graphs and DAGs (which were moralized first) with $|V| = 10, 20, 50, 100, 200$, and 500 , and three different densities. The details are as follows.

We created the undirected graphs by generating from the Erdős–Rényi random graph model $G(|V|, p)$,^{36,37} in which there are $|V|$ vertices and each edge appears independently with probability p , and rejecting graphs that were already triangulated. To achieve three different densities, for each value of $|V|$ we set p to $3/(|V|-1)$, $5/(|V|-1)$, and $7/(|V|-1)$. These values were chosen so that the expected degree of a vertex was 3, 5, and 7 respectively before the rejection of the triangulated graphs (to see this, note that in $G(|V|, p)$ the expected degree is $(|V|-1)p$). For each value of $|V|$ and each value of p we created 30 graphs.

We then created triangulations of these graphs. This was done by using elimination with the vertices in their natural orders (that is, $x_1, x_2, \dots, x_{|V|}$), because minimum-degree elimination often creates triangulations that are already minimal. We ran Algorithms II and III on all these triangulated graphs and recorded the time taken. To see whether the differences in speed were due to the algorithms themselves rather than implementation overhead, we also recorded the number of edges that were checked by the Repeat loops.

We created the DAGs by using the `random_dag` function in `gRbase`. One of the arguments to this function is the maximum number of parents each vertex can have, which we call m , and another is a weight, which we call w . The function generates graphs in which the probability of a vertex having i parents is proportional to w^i , for $i = 0, \dots, m$. We set m to 10, and to make three different densities we set w to 0.3, 0.4, and 0.5. For each value of $|V|$ and each value of w we created 30 DAGs. We moralized the DAGs, triangulated them in the same way as for the undirected graphs, and ran the two algorithms.

The experiments were performed in R, version 3.2.1, on a Linux machine with 2.4MHz Xeon E5-2695 processors. The results are shown in Table 1. The $|E|$ columns contain the mean numbers of edges in the original graphs that we

generated, before elimination or moralization; these show that the densities were roughly similar for the undirected graphs and the DAGs — for example, with $|V| = 100$ and the highest of the three densities, the mean number of edges was 348 in the undirected graphs and 353 in the DAGs.

For the undirected graphs, Algorithm III was faster for the large and dense graphs, which means that the extra steps performed by Algorithm III had the intended effect. But for the other graphs the speeds were similar. For all sizes and densities Algorithm III checked fewer edges, as expected, but for the smaller and less dense graphs that did not translate to faster speeds. The time spent dealing with the set R must have cancelled out the gains from checking fewer edges.

For the DAGs, Algorithm III was slower for almost all sizes and densities, and especially so for the lower densities — as with the undirected graphs, it did better in this comparison with Algorithm II when the density was higher. But the differences between the two algorithms were small. Algorithm III checked slightly fewer edges than Algorithm II but not to a sufficient extent to make it faster.

10. Discussion

Our main contributions are to prove that Algorithm I is incorrect and Algorithms II and III are correct. The statement of Algorithm II is also new. For Algorithm I we have provided graphs (Figs. 2 and 3) to show that the algorithm can go wrong when any of the three most popular elimination orderings are used, starting with DAGs or undirected graphs.

As for the relative speeds of Algorithms II and III, neither of them is always faster than the other, and they have the same asymptotic speed. The computer experiment suggested that Algorithm III is somewhat faster for dense undirected graphs, especially large ones, because it economizes on the number of edges that are checked. So the extra steps in Algorithm III are probably worthwhile, but Algorithm II would be adequate for many purposes.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (Grant number EP/P50483X/1). Financial support from the Leverhulme Trust (Research Fellowship RF-2011-320) is gratefully acknowledged.

Data Access Statement

The programs used for the computer experiment are available at <https://doi.org/10.5281/zenodo.161528>.

Table 1. Results of the experiment to compare the speed of Algorithms II and III on undirected graphs and DAGs. For full details see Section 9. In the Alg. II and III columns, the first quantity in each cell is the mean time per graph taken by the algorithm, in seconds, and the second number (in parentheses) is the mean number of edges that were checked in its Repeat loop. The p and w columns contain the parameters that determined the densities of the graphs and the $|E|$ columns show the sample mean numbers of edges in the graphs that were used, before elimination or moralization.

$ V $	Undirected graphs				DAGs (Bayesian networks)			
	p	$ E $	Alg. II	Alg. III	w	$ E $	Alg. II	Alg. III
10	3/9	15.9	0.00190 (39.3)	0.00200 (37.4)	0.3	8.1	0.00020 (1.9)	0.00027 (1.9)
	5/9	25.2	0.00157 (30.6)	0.00153 (28.5)	0.4	14.3	0.00047 (6.8)	0.00050 (6.8)
	7/9	35.1	0.00083 (16.6)	0.00083 (13.4)	0.5	18.2	0.00043 (5.9)	0.00053 (5.9)
20	3/19	29.8	0.0145 (247)	0.0135 (227)	0.3	19.1	0.00090 (10)	0.00103 (10)
	5/19	49.3	0.0138 (305)	0.0141 (265)	0.4	34.7	0.00210 (31)	0.00230 (31)
	7/19	71.1	0.0114 (279)	0.0112 (235)	0.5	56.3	0.00440 (65)	0.00413 (64)
50	3/49	76.1	0.113 (2157)	0.122 (1926)	0.3	53.3	0.0110 (76)	0.0114 (76)
	5/49	126.7	0.157 (3393)	0.162 (2827)	0.4	96.1	0.0246 (297)	0.0254 (293)
	7/49	176.7	0.200 (4048)	0.193 (3195)	0.5	162.4	0.0490 (801)	0.0545 (781)
100	3/99	153	0.825 (12322)	0.846 (10321)	0.3	110	0.045 (332)	0.058 (329)
	5/99	254	1.752 (24255)	1.618 (18727)	0.4	221	0.168 (1966)	0.190 (1923)
	7/99	348	2.445 (28602)	2.097 (21950)	0.5	353	0.347 (4722)	0.381 (4552)
200	3/199	299	6.86 (65755)	6.65 (54514)	0.3	234	0.243 (1105)	0.322 (1100)
	5/199	496	22.95 (150357)	20.24 (117424)	0.4	443	1.020 (8495)	1.174 (8269)
	7/199	693	46.96 (189426)	43.01 (141942)	0.5	746	3.519 (29484)	3.701 (27885)
500	3/499	748	285 (761656)	248 (637234)	0.3	620	2.7 (8226)	3.5 (8156)
	5/499	1242	1285 (1968550)	1216 (1499796)	0.4	1095	16.4 (65237)	18.1 (62800)
	7/499	1740	2714 (2925613)	2063 (2028666)	0.5	1866	102.8 (292812)	106.8 (272508)

References

1. U. Kjaerulff, Triangulation of graphs: algorithms giving small total state space, Technical report R90-09 (Aalborg University, 1990).
2. U. Kjaerulff, Aspects of efficiency improvement in Bayesian networks, PhD thesis (Aalborg University, 1993).
3. K. G. Olesen and A. L. Madsen, Maximal prime subgraph decomposition of Bayesian networks, *IEEE T. Syst. Man Cybern.* **32** (2002) 21–31.
4. C. Dethlefsen and S. Højsgaard, A common platform for graphical models in R: the gRbase package, *J. Stat. Softw.* **14** (2005) 1–12.
5. P. Heggernes, Minimal triangulations of graphs: a survey, *Discrete Math.* **306** (2006) 297–317.
6. H.-G. Leimer, Optimal decomposition by clique separators, *Discrete Math.* **113** (1993) 99–123.
7. D. Wu, Maximal prime subgraph decomposition of Bayesian networks: a relational database perspective, *Int. J. Approx. Reason.* **46** (2007) 334–345.
8. A. P. Dawid and S. L. Lauritzen, Hyper Markov laws in the statistical analysis of decomposable graphical models, *Ann. Stat.* **21** (1993) 1272–1317.
9. P. Giudici and P. J. Green, Decomposable graphical Gaussian model determination, *Biometrika* **86** (1999) 785–801.
10. P. J. Green and A. Thomas, Sampling decomposable graphs using a Markov chain on junction trees, *Biometrika* **100** (2013) 1–20.
11. A. Roverato, Hyper inverse Wishart distribution for non-decomposable graphs and its application to Bayesian inference for Gaussian graphical models, *Scand. J. Stat.* **29** (2002) 391–411.
12. A. Atay-Kayis and H. Massam, A Monte Carlo method for computing the marginal likelihood in nondecomposable Gaussian graphical models, *Biometrika* **92** (2005) 317–335.
13. A. Lenkoski and A. Dobra, Computational aspects related to inference in Gaussian graphical models with the G-Wishart prior, *J. Comput. Graph. Stat.* **20** (2011) 140–157.
14. H. Wang and S. Z. Li, Efficient Gaussian graphical model determination under G-Wishart prior distributions, *Electron. J. Stat.* **6** (2012) 168–198.
15. D. J. Rose, A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations, in *Graph Theory and Computing*, ed. R. C. Read (New York, Academic Press, 1972).
16. S. L. Lauritzen and D. J. Spiegelhalter, Local computations with probabilities on graphical structures and their application to expert systems, *J. Roy. Stat. Soc. B Met.* **50** (1988) 157–224.
17. R. E. Tarjan and M. Yannakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, *SIAM J. Comput.* **13** (1984) 566–579.
18. F. R. K. Chung and D. Mumford, Chordal completions of planar graphs, *J. Comb. Theory B* **62** (1994) 96–106.
19. S. L. Lauritzen, *Graphical Models* (Oxford University Press, Oxford, 1996).
20. R. Diestel, *Graph Theory*, 3rd edn. (Springer, Berlin, 2005).
21. J. A. Bondy and U. S. R. Murty, *Graph Theory* (Springer, 2008).
22. G. A. Dirac, On rigid circuit graphs, *Abh. Math. Semin. Univ. Hambg.* **25** (1961) 71–76.
23. D. J. Rose, R. E. Tarjan and G. S. Lueker, Algorithmic aspects of vertex elimination on graphs, *SIAM J. Comput.* **5** (1976) 266–283.

24. M. J. Flores, J. A. Gamez and K. G. Olesen, Incremental compilation of Bayesian networks, *UAI-03* (2003) 233–240.
25. M. J. Flores, J. A. Gamez and K. G. Olesen, Incremental compilation of Bayesian networks based on maximal prime subgraphs, *Int J. Uncertain. Fuzz.* **19** (2011) 155–191.
26. M. Yannakakis, Computing the minimum fill-in is NP-complete, *SIAM J. Algebra Discr.* **2** (1981) 77–79.
27. S. Parter, The use of linear graphs in Gauss elimination, *SIAM Rev.* **3** (1961) 119–130.
28. T. J. Ottosen and J. Vomlel, All roads lead to Rome — new search methods for the optimal triangulation problem, *Int. J. Approx. Reason.* **53** (2012), 1350–1366.
29. J. R. S. Blair, P. Heggernes and J. A. Telle, A practical algorithm for making filled graphs minimal, *Theor. Comput. Sci.* **250** (2001) 125–141.
30. A. Darwiche, *Modeling and Reasoning with Bayesian Networks* (Cambridge University Press, Cambridge, 2009).
31. T. J. Ottosen and J. Vomlel, Honour thy neighbour — clique maintenance in dynamic graphs, in *Proc. 5th European Workshop on Probabilistic Graphical Models* (Helsinki, 2010), pp. 201–209.
32. T. J. Ottosen and J. Vomlel, All roads lead to Rome — new search methods for optimal triangulation, in *Proc. 5th European Workshop on Probabilistic Graphical Models* (Helsinki, 2010), pp. 209–217.
33. C. Li and M. Ueno, A depth-first search algorithm for optimal triangulation of Bayesian network, in *Proc. 6th European Workshop on Probabilistic Graphical Models* (Granada, 2012), pp. 187–194.
34. C. Li and M. Ueno, A fast clique maintenance algorithm for optimal triangulation of Bayesian Networks, in *Advanced Methodologies for Bayesian Networks: Second International Workshop, AMBN 2015* (New York, 2015), pp. 152–167.
35. Z. Yang, H. Huo and T. Fang, Ordering strategies for removing redundant fill edges, *IEEE Adv. Comput. Conf.* (2014) 744–753.
36. E. N. Gilbert, Random graphs, *Ann. Math. Stat.* **30** (1959) 1141–1144.
37. B. Bollobás, *Random Graphs*, 2nd edn. (Cambridge University Press, Cambridge, 2001).