New results on minimal triangulations

Yngve Villanger

Department of Informatics
University of Bergen
Norway

January 2006

Acknowledgements

First and foremost I would like to thank my supervisor Professor Pinar Heggernes, for excellent guidance and motivation. After I graduated from the master program as a Cand. Scient she encouraged me to apply for a PhD position, something that I am very grateful for today. Since then Pinar has been generous with her time, both when discussing my more or less vague ideas and when reading my drafts. Thanks Pinar, you have taught me a lot about research and life in general. :-)

I would also like to thank my co-authors Anne Berry, Jean-Paul Bordat, Pinar Heggernes, Dieter Kratsch, Genevieve Simonet, Karol Suchan, Jan Arne Telle, and Ioan Todinca for interesting discussions and productive collaboration.

In the spring semester of 2005 I spent two periods of almost three months at the LIFO (Laboratoire d'Informatique Fondamentale d'Orléans) of the University of Orléans, France. Many thanks goes to the people at LIFO for making this time interesting and productive. In particular I would like to thank Ioan and Alice Todinca, and Karol Suchan for making these two periods rewarding and memorable, both academically and socially. Especially when I, my wife and our little daughter where living in Orléans as a family.

The people in the Algorithm group at the Department of Informatics in Bergen deserves thanks for an open, friendly, and inspiring work environment. Special thanks goes to Christian Sloper as a colleague and a friend.

This work has been financially supported by a research fellow position from the University of Bergen. In addition to this L. Meltzers Høyskolefond and the AURORA mobility programme for research collaboration between France and Norway have contributed with travel founding.

I would also like to thank my parents Per Magne and Gerd who have always showed interest in my work and been supportive.

Finally I wish to thank my marvellous wife Tone for proof reading the introduction to this thesis. She also deserves thanks for being so understanding and patient especially during my work on completing this thesis.

This thesis is dedicated to our wonderful daughter Kristin.

Bergen, January 2006
Yngve Villanger

Introduction to the Thesis

Yngve Villanger

1 Background and motivation

Computing the *treewidth* and the *minimum fill-in* of a graph are two of the most well studied problems in the field of graph algorithms. Treewidth can be seen as a parameter that describes how close a graph is to a tree, while the minimum fill-in can be seen to describe how close a graph is to a *chordal* graph. The solution of each of these problems is equivalent to embedding a given graph into a chordal supergraph with some special properties. Unfortunately both these problems are *NP-hard* on general graphs [1, 47], thus no efficient polynomial time algorithms are known.

During the work on their graph minor project, Robertson and Seymour [40] introduced several new graph parameters as tools to prove their results, and treewidth was one of them. Later, treewidth has proved useful in many areas, where VLSI layout and evolution theory [11] are some examples. Even more important, treewidth is now widely accepted as one of the most important graph parameters, because a wide range of NP-hard problems on general graphs can be solved in polynomial time when the treewidth is bounded by some constant. (These algorithms are polynomial in the size of the input graph, but they are exponential in the treewidth of the graph.)

The minimum fill-in problem is also known as the *minimum triangulation* problem. An embedding of an input graph into a chordal graph can be obtained by adding edges until the graph becomes chordal. The edges added to the graph are called *fill* edges, and the resulting chordal graph is called a *triangulation*. Minimum triangulation is the problem of obtaining a chordal graph by adding the fewest possible number of fill edges, and the minimum fill-in is the number of such edges. The problem of finding the minimum fill-in was first studied in sparse matrix computations [42], but it has also applications in other areas, like database management [2, 43] and computer vision [15].

A *tree decomposition* is a way of decomposing the input graph into a tree, where each node of the tree corresponds to a vertex subset of the input graph. The definition of treewidth is based on tree decompositions. However it is interesting to notice that a tree decomposition describes a triangulation and every triangulation describes a tree decomposition, thus these two structures are equivalent. The problem of computing the treewidth can be restated as the problem of computing a triangulation where the size of the largest clique is minimized. A consequence of this is that triangulations and tree decompositions can be used interchangeably when working on one of the two NP-hard problems mentioned above.

A *minimal triangulation* is a triangulation of the input graph such that no subset of the added edges results in a triangulation of the graph. Unlike treewidth and minimum triangulation, a minimal triangulation can be computed in polynomial time, where the best known time bounds are $O(nm)$ [6, 8, 36, 41] for sparse graphs and $O(n^{2.376})$ [26] for dense graphs, for an input graph with n vertices and m edges. For a survey about chordal graphs and minimal triangulation see [24].

Minimum fill-in and treewidth require searching for triangulations with different properties, but the optimal solution for both problems can be found among the minimal triangulations. Thus, minimum fill-in and treewidth problems can be solved by searching through the set of minimal triangulations of the input graph, which might be exponentially large. Minimal triangulations can be characterized in several different ways. Examples are characterizations through a tree decomposition, an *elimination ordering* of the vertices [37], and a set of *minimal separators* [38]. These and other characterizations will be explained in subsection 2.4.

Any ordering of the vertices in a graph defines a triangulation, which can be obtained by an algorithm called *the elimination game* [39]. If no fill edges are added by this algorithm, then the ordering is called a *perfect elimination ordering*. Fulkerson and Gross [21] showed that a graph is chordal if and only if it has a perfect elimination ordering. This was later used by Ohtsuki, Cheung, and Fujisawa [37] to define a minimal triangulation through an elimination ordering. Such an elimination ordering is called a *minimal elimination ordering*.

Vertex separators are structures that are central both in tree decompositions and in triangulations, and these separators can easily be reduced to *minimal separators* without increasing the treewidth or fill-in of a triangulation. The minimal separators of a graph are powerful enough to describe all interesting tree decompositions and triangulations [38], and thus the solution of treewidth, minimum triangulation, and a minimal triangulation can be defined by a set of minimal separators. It follows that each of these three problems can be reformulated as a problem of finding a set of minimal separators with some given property.

In this thesis we study properties of tree decompositions, minimal separators, and elimination orderings, and how these can be used as tools when constructing new algorithms for problems like minimal triangulation. We will now give some examples, where these structures are used. Besides from being a decomposition of a graph, a tree decomposition can also be used as a data structure that stores vertex separators. In contrast to a simple list structure, this tree structure contains information about the relation between the separators. By using this information, it is possible to compute the union of a set of minimal separators in a more efficient way, as we showed in [6]. In special cases the union of a set of minimal separators can be found even faster by using an alternative representation of the tree decomposition, and an example of this data structure is presented in [8].

Minimal separators do not only separate the graph into at least two connected components, but they also separate the minimal triangulation problem into independent subproblems [31]. Even though this characterization of minimal triangulations has been implicit for some years, almost no algorithm took advantage of this property until 2004. In [26] we use this in combination with other techniques, to improve the running time for minimal triangulation of dense graphs.

Not all minimal triangulation algorithms are able to produce every minimal triangulation of an input graph. Even though two algorithms produce different sets of minimal elimination orderings, it is possible that they produce the same set of minimal triangulations, since many different elimination orderings can define the same triangulated graph. In [29] we define a set of modifications that can be done to an elimination ordering without changing the resulting triangulation. Based on these observations it is shown in [45] that two algorithms, Lex M [41] and MCS-M [4], that produce different sets of elimination orderings, actually produce the same set of triangulations.

As mentioned above, the treewidth and minimum fill-in can be found by searching through the set of minimal triangulations of the input graph. For each minimal triangulation, there exists a tree decomposition of the input graph defining the same set of fill edges. Each of the tree nodes in this tree decomposition is defined by and contains the information of a set of minimal separators in the minimal triangulation. These tree nodes are called *potential maximal cliques* [12] of the input graph, and can be used to define minimal triangulations [12], or to improve the time bound of exponential time algorithms [20].

The purpose of this introduction is to emphasize a set of relations and links between the papers that present the technical results of this thesis. The thesis consists of this introduction and five attached research papers following it, where the introduction is organized as follows. Section 2 provides definitions, and shows how several different structures and definitions that are commonly used in triangulation algorithms can be considered as separators in a graph. Section 3 presents the history behind some of the attached papers, and a summary of the technical results in each paper. The five papers that define the main body of this thesis are listed below. The list of papers is sorted chronologically according to the date each paper was submitted to a journal.

I. Anne Berry, Jean-Paul Bordat, Pinar Heggernes, Genevieve Simonet, and Yngve Villanger. *A wide-range algorithm for minimal triangulation from* *an arbitrary ordering.* Journal of Algorithms. Volume 58, Issue 1, Pages 33-66, Year 2006. [6]

II. Anne Berry, Pinar Heggernes, and Yngve Villanger. *A Vertex Incremental Approach for Maintaining Chordality.* Discrete Mathematics. To appear. [8]

III. Yngve Villanger. *Lex M versus MCS-M.* Discrete Mathematics. To appear. [45]

IV. Pinar Heggernes, Jan Arne Telle, and Yngve Villanger. *Computing Minimal Triangulations in Time $O(n^{\alpha} \log n) = o(n^{2.376})$.* SIAM Journal on Discrete Mathematics. Volume 19, Number 4, Pages 900-913, Year 2005. [26]

V. Fedor V. Fomin, Ioan Todinca, Dieter Kratsch, and Yngve Villanger. *Exact algorithms for treewidth and minimum fill-in.* Submitted to SIAM Journal on Computing. [20]

2 Viewing everything as separators

Chordal graphs and the process of creating chordal graphs by adding edges to arbitrary graphs are two subjects that are common to all results presented in this thesis. If a chordal graph is obtained by adding edges to a non chordal graph, then this resulting graph is called a triangulation of the non chordal input graph. Both chordal graphs and triangulations can be characterized in several different ways, which will be discussed further, later in this introduction. These characterizations are useful tools when designing new triangulation algorithms, since each characterization defines a way to recognize or create a chordal graph. New triangulation algorithms are usually obtained by finding a new characterization and then combining this with already known characterizations, or by combining several known characterizations in a new way. Thus, knowing and understanding these characterizations are important when designing such algorithms. But before we can define and discuss these characterizations, some definitions are required. In order to give an alternative view of these problems and definitions, we will redefine several known structures used in triangulation algorithms as different types of separators.

2.1 Basic definitions

Graphs considered in this thesis are simple and undirected. A graph $G = (V, E)$ is a pair consisting of a set of vertices V and a set of edges E. The number of vertices is denoted by n, and the number of edges is denoted by m. Two vertices u, v are considered as *neighbors* if uv is an edge in E. The *neighborhood* of a vertex u is denoted by the vertex set $N(u)$, where $v \in N(u)$ if $uv \in E$, and the *closed neighborhood* $N(u) \cup \{u\}$ of u is denoted by $N[u]$. For a vertex set $A \subseteq V$ the edge set $E(A)$ is given by $\{uv \in E \mid u, v \in A\}$. Let $G[A]$ denote the subgraph $(A, E(A))$ of G. We call $G[A]$ the subgraph of G *induced* by A. For simplicity we will write $G \setminus A$ for the induced subgraph $G[V \setminus A]$ of G. A vertex set $A \subseteq V$ is a *clique* if $uv \in E$ for every pair $u, v \in A$, and A is a *maximal clique* if there exists no clique A' such that $A \subset A'$. The opposite of a clique is an independent set, and the vertex set $I \subseteq V$ is an independent set if $uv \notin E$ for every pair $u, v \in I$.

An *ordering* of the vertices in a graph G is a function $\alpha : V \leftrightarrow \{1, 2, ..., n\}$. Let $v_0, v_1, ..., v_k$ denote a path from v_0 to v_k in G of length k, i.e., $v_i \neq v_j$ for $i \neq j$ and $v_i v_{i+1} \in E$ for $0 \leq i < k$. In the same way $v_0, v_1, ..., v_k, v_0$ denotes a cycle of length $k + 1$. A vertex set C containing u induces a *connected component* of G, if $v \in C$ for every pair u, v, such that there exists a path from u to v in G. A connected graph G is a *tree*, if G contains no cycles, and for every pair u, v of vertices in V, there exists a path between u and v in G.

2.2 Separators

A *vertex separator* is a vertex set such that a connected component of a graph becomes disconnected by removing this set. A vertex set $S \subset V$ is a *u, v-separator* in a connected graph $G = (V, E)$ with $u, v \in V$ if u and v are contained in different connected components of $G \setminus S$. Given a graph $G = (V, E)$, let $S \subset V$ be a u, v-separator of G, then S is a *minimal u, v-separator* of G if no proper subset of S separates u and v. If S is a minimal u, v-separator of $G = (V, E)$ for some pair $u, v \in V$, then S is a *minimal separator* of G.

Lemma 2.1 (Folklore) *Given a graph $G = (V, E)$, let $S \subset V$, and let $C_1, C_2, ..., C_k$ be the connected components of $G \setminus S$. Then S is a minimal separator if and only if there exists a pair i, j, with $1 \leq i < j \leq k$, such that $S = N(C_i) = N(C_j)$.*

Proof. Let i and j be integers such that $S = N(C_i) = N(C_j)$, where $1 \leq i < j \leq k$, and let u and v be vertices such that $u \in C_i$ and $v \in C_j$. Since $S = N(C_i) = N(C_j)$ then it follows that u and v are contained in the same component of $G \setminus (S \setminus \{x\})$ for every vertex $x \in S$. Thus, S is a minimal u, v-separator since no subset of S separates u and v.

If S is a minimal separator, then there exists a pair u, v such that no subset of S separates u and v. Then u and v are contained in the same component C of $G \setminus (S \setminus \{x\})$ for every vertex $x \in S$, and every path from u to v in C contains the vertex x. Let C_u and C_v be the connected components of $G \setminus S$ containing u and v. Since both u and v have a path through vertices in $V \setminus S$ to every vertex $x \in S$, then it follows that $S = N(C_u) = N(C_v)$. ■

A connected component C of $G \setminus S$ is called a *full component* of S if $S = N(C)$. Lemma 2.1 can now be restated as follows: A separator is minimal if and only if it has at least two full components. Actually the neighborhood of any component of $G \setminus S$ is a minimal separator if S is a minimal separator of G, see Lemma 2.2. Another property of minimal separators is that no pair u, v of non adjacent vertices contained in a minimal separator S can be separated by a subset of the vertices in S. Notice that this is only true if $G \setminus S$ contains a connected component C, such that $u, v \in N(C)$. We can now use this component to restate the property: For any non adjacent pair u, v contained in a minimal separator S, there exists a component C of $G \setminus S$ such that $u, v \in N(C)$. Even though this property is trivial for minimal separators since they have at least two full components, it will be useful when we generalize the definition of separators further later in this text.

Lemma 2.2 *Given a graph $G = (V, E)$, let $S \subset V$, and let $C_1, C_2, ..., C_k$ be the connected components of $G \setminus S$. Then S is a minimal separator if and only if*

1. *C_j is a full component of S, for some j satisfying $1 \leq j \leq k$, and*
2. *$N(C_i)$ is a minimal separator of G, for every i satisfying $1 \leq i \leq k$, and*
3. *for any non adjacent pair $u, v \in S$, there exists an i such that $u, v \in N(C_i)$, where $1 \leq i \leq k$. (This requirement follows from the first, since $S = N(C_j)$ and thus $u, v \in N(C_j)$.)*

Proof. Let C_j be a full component of G, where $1 \leq j \leq k$, and let $N(C_1)$, $N(C_2), ..., N(C_k)$ be minimal separators of G. Then S is a minimal separator since $S = N(C_j)$ and $N(C_j)$ is a minimal separator of G.

Let S be a minimal separator of G. Then it follows from Lemma 2.1 that there exist at least two full components C_p and C_q of S, where $1 \leq p, q \leq k$ and $p \neq q$. The set $N(C_i)$ is a minimal separator for $1 \leq i \leq k$, since $G \setminus (C_i \cup S)$ contains one of the two full components of S, and thus there exists a full component of $N(C_i)$ in $G \setminus N[C_i]$. Finally for every non adjacent pair $u, v \in S$, the vertices u, v are contained in $N(C_p)$, since C_p is a full component, and thus $u, v \in S = N(C_p)$. ∎

A vertex separator is defined as a vertex set separating at least two vertices. This can be restated as separating every pair of vertices in a vertex set I, where $|I| = 2$. We can now generalize the definition of separators to separate every pair of vertices in a vertex set I, where $|I| \geq 0$[1]. Obviously none of the vertices contained in I can be adjacent.

It is tempting to define such a separator for a set in the following way. Let $I \subset V$ be an independent set in $G = (V, E)$, let $K \subseteq (V \setminus I)$, and let $C_1, C_2, ..., C_k$

[1]For the purpose of generalization we allow that $|I| < 2$, even though this is a bit against the intuition of separation since, no vertices are separated in this case.

be the connected components of $G \setminus K$. Then K is a *set separator* separating I if $|I \cap C_i| \leq 1$ for $1 \leq i \leq k$.

Since a set separator is a generalization of a vertex separator, we want minimal set separators to have similar properties as minimal separators. Unfortunately, inclusion minimal set separators do not behave like minimal separators. For instance, we want to maintain the property that no subset of a minimal set separator separates vertices contained in that minimal set separator. The following example shows that this does not hold for the above definition of set separators. Consider a simple cycle with eight vertices. Start from any vertex and number the vertices 1 to 8 in a clockwise order. Let I be vertices with odd number, and let K be vertices with even number. The set K is clearly a minimal set separator, since every vertex in K has two vertices from I in its neighborhood. Notice that no component of the graph where K is removed contains both the vertices numbered 2 and 6 or 4 and 8 in its neighborhood. Now we have a counterexample to the property since set $\{4, 8\}$ separates 2 and 6, and the set $\{2, 6\}$ separates 4 and 8.

Another property that we would like to preserve is that the neighborhood of the resulting connected components are minimal separators. The following example shows that this is not the case for the above definition of set separators. Consider a simple path with five vertices. Start in one end of the path and number the vertices successively towards the second end, with the numbers 1 to 5. Let I be the set of odd numbered vertices, and let K be the set of even numbered vertices. The set K is clearly a minimal set separator, since every vertex in K contains two vertices from I in its neighborhood. The vertex with number 3 is one of the resulting connected components when K is removed from the graph. Let us call this component C. This component does not satisfy the desired property, since $K = N(C)$, but K is not a minimal separator in the graph.

We will now define a separator for sets such that the following two properties are preserved in the inclusion minimal version of the separator: No subset of a minimal separator for a set separates vertices contained in the separator, and the neighborhood of any remaining connected component when the separator is removed is a minimal separator. Notice the similarities between Lemma 2.2 and Definition 2.3.

Definition 2.3 *Given a graph $G = (V, E)$, let $I \subset V$ be an independent set, let $K \subseteq (V \setminus I)$, and let $C_1, C_2, ..., C_k$ be the connected components of $G \setminus K$. Then K is a* BT-separator *separating I if*

1. *$|C_i \cap I| \leq 1$ for every i satisfying $1 \leq i \leq k$, and*
2. *$N(C_i)$ is a minimal separator of G, for every i satisfying $1 \leq i \leq k$, and*
3. *for every non adjacent pair $u, v \in K$ there exists an i, such that $u, v \in N(C_i)$, where $1 \leq i \leq k$.*

Definition 2.4 *Given a graph $G = (V, E)$, let $I \subset V$ be an independent set, and let $K \subseteq (V \setminus I)$ be a BT-separator separating I. Then K is a* minimal BT-separator *separating I if no subset of K is a BT-separator separating I.*

If K is a BT-separator separating some independent set I in a graph G, then we say that K is a BT-separator of G. We can now obtain the following result.

Lemma 2.5 *Given a graph $G = (V, E)$, let $K \subseteq V$, and let $C_1, C_2, ..., C_k$ be the connected components of $G \setminus K$. Then K is a* BT-separator *of G if and only if*

1. *$N(C_i)$ is a minimal separator of G, for $1 \leq i \leq k$, and*
2. *for every non adjacent pair $u, v \in K$ there exists an i, such that $u, v \in N(C_i)$, where $1 \leq i \leq k$.*

Proof. Let K be a vertex set such that $N(C_1), N(C_2), ..., N(C_k)$ are minimal separators of G, and such that for every pair $u, v \in K$, there exists an integer i such that $u, v \in N(C_i)$ and $1 \leq i \leq k$. Let I be a vertex set obtained by selecting one vertex from each of the connected components $C_1, C_2, ..., C_k$. The vertex set K separates I and thus it follows from Definition 2.3 that K is a BT-separator of G. The opposite direction of the proof follows directly, since the requirements are a subset of the requirements in Definition 2.3. ■

Notice that if $G \setminus K$ has a full component for a BT-separator K of G, then K is a minimal separator, and thus K has at least two full components. It follows that a BT-separator has either zero or at least two full components. A BT-separator which is not a minimal separator, and thus has no full components, is known as a potential maximal clique [12]. This will be studied in detail in subsection 2.4.

Two vertex separators S and T of a graph G, are said to be *crossing* if S is a u, v-separator for a pair of vertices $u, v \in T$, or if T is an x, y-separator for a pair of vertices $x, y \in S$. This can be stated even stronger for minimal separators. Two *minimal* separators are said to be *crossing* if S is a u, v-separator for a pair of vertices $u, v \in T$, in which case T is an x, y-separator for a pair of vertices $x, y \in S$ [31, 38]. We will say that two BT-separators K and L of a graph G are *crossing* if there exists a component C_K of $G \setminus K$ and a component C_L of $G \setminus L$, such that $N(C_K)$ and $N(C_L)$ are crossing minimal separators.

Actually a BT-separator can be considered as a well chosen set of non-crossing minimal separators [12]. Notice that this set can actually be the empty set, like the BT-separator V in the complete graph $G = (V, E)$. We will now generalize the definition of separators further, such that a larger set of separators can be represented in the same structure. Crossing separators can be considered as rivals of each other, since at least one of them separates vertices in the other. If we limit our selves to only consider sets of non-crossing separators, then this allows us to represent several separators in a single structure.

Definition 2.6 *Given a graph $G = (V, E)$, let $S_1, S_2, ..., S_k$ be a set of non-crossing separators in G. Then $H = (V, F')$ is a* tree separator *representing the separators $S_1, S_2, ..., S_k$ if $uv \in F'$ for every pair of vertices u, v such that u and v are not separated by S_i for $1 \leq i \leq k$.*

Tree separators can be considered as a generalization of separators and thus also BT-separators, since every separator or BT-separator can be represented by a tree separator, while the opposite is not true. Some properties can be noticed about a tree separator H of G. Since no separator separates adjacent vertices in G, then $E \subseteq F'$, and since all the separators are non-crossing, then the vertex sets $S_1, S_2, ..., S_k$ are all cliques in H. Actually any BT-separator in H is a clique.

Lemma 2.7 *Let $H = (V, E \cup F)$ be a tree separator representing a set of non-crossing separators in $G = (V, E)$. Then every BT-separator of H is a clique.*

Proof. Let K be a BT-separator in H. If K is a clique, then there is nothing to prove. If K is not a clique, then there exists a pair u, v of non adjacent vertices in K. From the definition of H, we know that there exists a separator S represented by H separating u and v. Let S' be an inclusion minimal subset of S separating u and v. Since $uv \notin E \cup F$ and $u, v \in K$, then there exists a component C of $H \setminus K$ such that $u, v \in N(C)$. Let T be the minimal separator $N(C)$ in H. The minimal separator S' is now separating $u, v \in T$. Thus, it follows from [31, 38] that T separates two vertices $x, y \in S'$. This is a contradiction since $S' \subseteq S$, where S is a clique in H. We can now conclude that K is a clique in H, since there exists an edge in H between every pair of vertices in K. ■

If a tree separator can be defined by a set of BT-separators, then we call this tree separator a *BT-tree separator*. We have now defined several new types of separators, and some of these contain others as special cases. By using these inclusion relations we can create a hierarchy between the definitions. The different inclusion relations are displayed on the left side of Figure 1. Separator definitions can also be partitioned into two groups, depending on whether they are defined through a vertex set or an edge set. The partition is as follows: minimal separators, minimal BT-separators, BT-separators, and vertex separators are defined through a vertex set, while BT-tree separators and tree separators are defined through an edge set. When first looking at Figure 1 it might seam like the tree separator is the optimal structure, since it contains all the other definitions as special cases. But notice that vertex set representation only requires $O(n)$ space, while an edge representation might require as much as $O(n^2)$ space. But a tree separator can also be represented with a set of vertex separators, and thus it can be represented with $O(n)$ times the number of vertex separators of space. This does not make a list of vertex separators an equal structure to a tree separator, since tree separators only store sets of non-crossing separators, while a list can store any set of separators. In the next section we will see that finding sets of non-crossing separators are of interest to us, so the fact that tree separators never represent crossing separators will be a useful property.

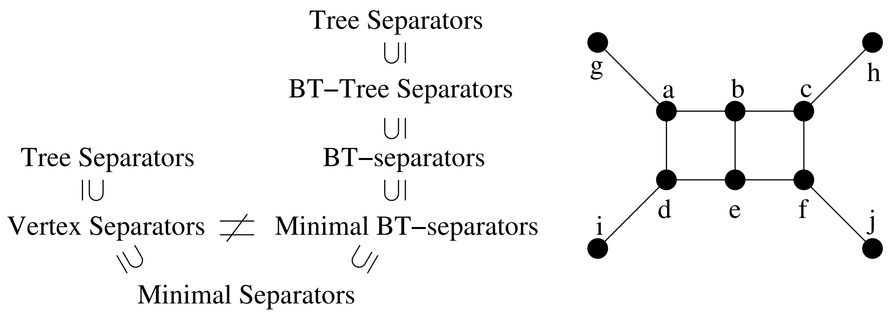

Figure 1: The left part of the figure shows the inclusion relation between the different types of separators defined in this section. We will use the graph on the right part of the figure to prove that the equality relation only holds in one direction. Let us start with the left branch of the inclusion relations to the left. The vertex pair $\{b, d\}$ in the graph to the right is a minimal separator, since it has two full components, and $\{a, b, d, e\}$ is a vertex separator since it separates g and i, and the vertex separators $\{a, b, d, e\}$ and $\{b, c, e, f\}$ define a tree separator, since they are non-crossing. The vertex separator $\{a, b, d, e\}$ is not a minimal separator, since it does not have a full component, and no vertex separator can represent the tree separator defined by the two vertex separators $\{a, b, d, e\}$ and $\{b, c, e, f\}$, since they separate vertices contained in one of the remaining connected components when the other vertex separator is removed. Let us now consider the right branch of the inclusion relations. The vertex pair $\{b, d\}$ is a minimal separator, and the vertex pairs $\{a, g\}$ and $\{b, d\}$ are minimal BT-separators, and $\{a, b, d\}$ is a BT-separator. The BT-separators $\{a, b, d\}$ and $\{c, e, f\}$ define a BT-tree separator, and finally the separators $\{a, b, c, d, e\}$ and $\{a, b, c, e, f\}$ define a tree separator. The minimal BT-separator $\{a, g\}$ is not a minimal separator since there is no full component, and the BT-separator $\{a, b, d\}$ is not a minimal BT-separator, since $\{b, d\}$ is a BT-separator separating the same set of vertices. Using the same arguments as for the left branch, we can argue that no single BT-separator can represent BT-tree separator defined by BT-separators $\{a, b, d\}$ and $\{c, e, f\}$, and the tree separator $\{a, b, c, d, e\}$ and $\{a, b, c, e, f\}$ can not be defined by a set of BT-separators, since no single BT-separator contains both a and c. It might also be noticed that the sets vertex separators $\neq$ minimal BT-separators, since $\{a, b, d, e\}$ is a separator and not a minimal BT-separator, and $\{a, g\}$ is a minimal BT-separator and not a vertex separator.

2.3 Chordal graphs

A graph is *chordal* if every cycle of length more than three has a chord. A chord is an edge between two non consecutive vertices in a cycle. The class of chordal graphs has been thoroughly studied since the early sixties, and has several interesting properties that will be useful to us. One of the first published results was that a chordal graph is either complete, or has two non adjacent simplicial vertices [18]. A vertex u is *simplicial* if $N[u]$ induces a clique in the graph. If the empty graph can be obtained from a graph by repeatedly removing simplicial vertices, then the order in which we remove the vertices is called a *perfect elimination ordering* (PEO) [21].

The definitions of *intersection graphs* and *tree decompositions* are useful when we talk about chordal graphs. A graph $G = (V, E)$ is the *intersection graph* of subtrees of a tree if there is a tree $T = (I, F)$, and for each vertex $u \in V$ a subtree T_u of T, such that for every pair $u, v \in V$, $uv \in E$ if and only if the trees T_u and T_v have at least one vertex in common.

Definition 2.8 *A* tree decomposition *of a graph $G = (V, E)$, is a pair (X, T) in which $T = (V_T, E_T)$ is a tree and $X = \{X_i \mid i \in V_T\}$ is a family of subsets of V such that:*

1. $\bigcup_{i \in V_T} X_i = V$*, and*

2. *for each edge $uv \in E$ there exists an $i \in V_T$ such that both u and v belong to X_i, and*

3. *for all $u \in V$, the set of tree nodes $\{i \in V_T \mid u \in X_i\}$ induces a connected subtree of T.*

Tree decompositions were defined and used by Robertson and Seymour [40] to define the *treewidth* of a graph. The width of a tree decomposition $(X, T = (V_T, E_T))$ of a graph G is the maximum of $|X_i| - 1$ for every $i \in V_T$, and the treewidth of the graph G is the minimum width over all tree decompositions of G. From now on we will simply refer to T when we mention a tree decomposition $(X, T = (V_T, E_T))$. The vertex subsets contained in X will be referred to as *tree nodes* of T, and the vertex set $X_i \cap X_j$ for an edge $ij \in E_T$ will be referred to as *tree edges* of T.

From the definition we know that chordal graphs do not contain chordless cycles. One consequence of this that can be deduced from [18] is that a chordal graph contains at most n maximal cliques. In [10] this property is used in a linear algorithm that lists all the maximal cliques and creates a tree decomposition of the chordal graph, where every tree node is a maximal clique. A result of this is that treewidth and the *clique number*(the size of the largest clique) of a chordal graph can be computed in linear time. But not all problems have efficient polynomial time algorithms for chordal graphs. Examples of the opposite are *graph isomorphism* [35] and the problem of computing the *pathwidth* [23] for chordal graphs. This property that some problems that are NP-hard for general graphs have efficient algorithms for chordal graphs, while other problems remains NP-hard is one of several reasons that makes chordal graphs interesting. The number of different characterizations of chordal graphs gives an intuition of how well studied this class is. Some characterizations, directly or indirectly related to algorithms or proofs presented in this thesis, are listed below.

1. A graph is chordal if and only if every minimal separator is a clique. (1961 [18])
2. A graph is chordal if and only if every minimal separator contained in the neighborhood of a vertex is a clique. (1962 [34])
3. A graph is chordal if and only if it has a *perfect elimination ordering* (PEO). (1965 [21])
4. A graph is chordal if and only if it is the intersection graph of subtrees of a tree. (1974 [22])
5. A graph G is chordal if and only if there exists a tree decomposition of G such that every tree node is a maximal clique in G, and every tree edge is a minimal separator of G. (1972-74 [14, 22, 46])
6. A graph G is chordal if and only if all minimal separators in G are pairwise non-crossing. (1997 [38])
7. A graph $G = (V, E)$ is chordal if and only if $N(u) \cap N(v)$ is a minimal separator in $(V, E \setminus \{uv\})$, for every edge $uv \in E$. (2004 [8])
8. A graph $G = (V, E)$ is chordal if and only if G is a tree separator representing every BT-separator of G. (2006 here)

The following corollary can now be obtained from Characterization 6.

Corollary 2.9 *A graph G is chordal if and only if all BT-separators in G are pairwise non-crossing.*

Proof. Every minimal separator is also a BT-separator, so we know from Characterization 6 that the graph is chordal if all BT-separators in G are pairwise non-crossing. Now for the opposite direction of the proof. If two BT-separators are crossing, then by the definition of crossing BT-separators, the graph contains two crossing minimal separators. We can now conclude, also by Characterization 6 that the graph is not chordal. ■

We will now use this new corollary to prove Characterization 8 in the list of characterizations.

Characterization 2.10 *A graph $G = (V, E)$ is chordal if and only if G is a tree separator representing every BT-separator of G.*

Proof. Let $H = (V, F')$ be a tree separator representing every BT-separator of G. Since no BT-separator of G separates consecutive vertices in G, and since all pairwise non consecutive vertices in G are separated by some BT-separator of G, then $F' = E$ and $H = G$. By Lemma 2.7 all BT-separators of H and G are cliques, and as a result all pairs of BT-separators of H and G are non-crossing. Thus, by Corollary 2.9 we can conclude that G is chordal.

Let G be chordal. Then we know from Corollary 2.9 that all BT-separators in G are pairwise non-crossing. From the definition of tree separators it follows that a tree separator $H = (V, F')$ representing every BT-separator of G can be constructed. For every pair of non adjacent vertices in G, there exists a BT-separator in G separating these vertices, and no BT-separator of G separates consecutive vertices in G. Thus, $G = H$, and the proof is complete. ∎

Actually the technique used in the proof of Characterization 2.10 is not the only way to relate tree separators to chordal graphs. The intersection graph of subtrees of a tree can also be used to create such a relation. Let T be a tree decomposition of a graph $G = (V, E)$. For every $u \in V$, let T_u be the tree nodes of T containing the vertex u, and by Definition 2.8 we know that T_u induces a connected subtree of T. The tree T and the subtrees T_u for every $u \in V$ define an intersection graph $H = (V, F')$ of subtrees of a tree. We will say that the graph H is *defined* by the tree decomposition T. From Characterizations 4 and 8 we know that H is chordal and that H is a tree separator representing some of the separators in G, and thus $E \subseteq F'$. A final notice might be that G is not necessarily a chordal graph, and in this case H is a chordal supergraph of G that is obtained by adding edges to G.

2.4 Triangulation and minimal triangulation

As we have seen in the previous subsection, chordal graphs can be obtained from non chordal graphs by adding edges. This is always possible, since a single tree node containing every vertex of the input graph is a tree decomposition of the input graph, and thus also defines a complete chordal supergraph of the input graph. If $H = (V, E \cup F)$ is a chordal supergraph of $G = (V, E)$, where $E \cap F = \emptyset$ then H is called a *triangulation* of G, and edges in F are called *fill* edges. We will say that the edge set F *defines* the triangulation H.

Some natural questions arise. What is the minimum number of fill edges that defines some triangulation of a given graph? Finding such a set of edges is known as the minimum fill-in or the minimum triangulation problem. This problem was conjectured to be NP-hard [41] in 1976, a conjecture that was confirmed by Yannakakis [47] five years later.

A simplification and a polynomial time version of the problem is to find an inclusion minimal set of fill edges, called a *minimal triangulation*. If $H = (V, E \cup F)$ is a triangulation of $G = (V, E)$, then H is a *minimal triangulation* of G if $H' = (V, E \cup F')$ is not a triangulation of G, for any edge set $F' \subset F$. Such triangulations can be obtained by a wide range of algorithms, where [9],[26], and [41] are some examples. Notice that a minimum triangulation is also a minimal triangulation, so the problem of finding a minimum triangulation can be considered as a problem of finding the *right* minimal triangulation.

Trivially a triangulation can be obtained by adding fill edges one by one until the graph becomes chordal. Even though checking if the obtained graph is chordal can be done in linear time [41, 43], it can be time consuming if we do this check at each step. In order to avoid this, triangulation algorithms introduce fill edges in a way that ensures that the resulting graph is chordal. This certificate is usually obtained by producing a perfect elimination ordering, or a tree decomposition which defines the resulting graph.

Without being aware of it, Parter[2] [39] presented the first triangulation algorithm, known as the *elimination game* in 1961. A graph $G = (V, E)$ and a vertex ordering α of G define the input to the algorithm. The elimination game adds edges to the input graph, such that the provided vertex ordering becomes a perfect elimination ordering of the resulting graph [21]. As a result, it follows that any graph produced by the elimination game is chordal. Another nice property of the algorithm is that it can be implemented to run in $O(n + m')$ time [43], where m' is the number of edges in the produced triangulation.

Using a tree decomposition is another way of defining a triangulation of the input graph. A triangulation defined by a tree decomposition can simply be created by completing every tree node in the tree decomposition into a clique. Notice that any triangulation algorithm that is based on finding and completing some separating vertex set into a clique, actually defines a tree separator or equivalently a tree decomposition. These triangulations can also be obtained in $O(n + m')$ time, by using a similar approach as the one used for the elimination game. But there is one difference: it can be easily verified that all triangulations can be defined by a tree decomposition, while this is not always possible by using an elimination ordering. For example, a complete graph can not be generated unless the input graph has a vertex adjacent to all other vertices. Despite this limitation, any minimal triangulation can be defined by a minimal elimination ordering [37]. This was proved by defining a *minimal elimination ordering*, which is an elimination ordering, such that no other ordering defines a triangulation using a strict subset of the fill edges.

Since a triangulation can be obtained in linear time and is defined by the elimination ordering or the tree decomposition, then the problem of finding trian-

[2]His aim was to give an algorithm that simulates Gaussian elimination on sparse matrices.

gulations with interesting properties is reduced to finding interesting elimination orderings or tree decompositions. We will now discuss different ways minimal triangulations can be defined.

In 2001 Bouchitté and Todinca [12] defined a *potential maximal clique* of a graph G to be a maximal clique in some minimal triangulation of G. These potential maximal cliques were key structures when Bouchitté and Todinca [13] showed that treewidth is polynomially tractable for all classes of graphs with a polynomial number of minimal separators. At first glance a potential maximal clique does not seem to be a very interesting structure when we are searching for minimal triangulations. Since a potential maximal clique is defined through a minimal triangulation of the input graph, it may seem that a triangulation should be computed before potential maximal cliques can be found. But this is not the case, since a potential maximal clique can be recognized directly in the input graph, and thus can be used to create the minimal triangulation it was defined from.

Theorem 2.11 ([12]) *Given a graph $G = (V, E)$, let $K \subseteq V$, and let C_1, $C_2, ..., C_k$ be the connected components of $G \setminus K$. Then K is a potential maximal clique if and only if*

1. *there exists no i such that C_i is a full component of K, where $1 \leq i \leq k$, and*

2. *$N(C_i)$ is a minimal separator of G, for $1 \leq i \leq k$, and*

3. *for any non adjacent pair $u, v \in K$ there exists an i such that $u, v \in N(C_i)$, where $1 \leq i \leq k$.*

The second requirement of Theorem 2.11 follows from the first and the third [12], but we add it to make it more similar to a previously defined structure: a BT-separator. It is interesting to notice that the only difference between the definition of a potential maximal clique in Theorem 2.11, and the definition of a minimal separator in Lemma 2.2, is the change in the first requirement from *none* to *some*. When we discussed BT-separators, we pointed out that BT-separators are either minimal separators, or have no full component. If a BT-separator has no full component, then it follows from Theorem 2.11 that this BT-separator is a potential maximal clique. Because of this strong similarity we have called these separators Bouchitté,Todinca-separators, or BT-separators for short. Thus, the set of BT-separators is the union of minimal separators and potential maximal cliques. Minimal BT-separators contain the set of minimal separators, but not the complete set of potential maximal cliques. An example of this is provided by a chordless four cycle. Every triple of vertices is a potential maximal clique, and only the two pairs of non adjacent vertices are minimal BT-separators.

Some of the motivation for finding characterizations of chordal graphs originates from the desire for finding new algorithms for minimal triangulations. Partly by the same motivation, the problem of finding minimal triangulations has been intensively studied. As a result, several characterizations for minimal triangulations have been published. We now give a list of characterizations of minimal triangulations.

1. A triangulation is minimal if and only if every fill edge is the unique chord of a 4-cycle in the triangulation. (1976 [41]) (Alternative formulation: A triangulation is minimal if and only if the removal of any single fill edge results in a non chordal graph.)

2. A triangulation is minimal if and only if it is defined by a minimal elimination ordering. (1976 [37])

3. Let S be a minimal separator of $G = (V, E)$, and let $G' = (V, E')$ be the graph obtained from G by completing S into a clique. Let further $C_1, C_2, ..., C_k$ be the connected components of $G \setminus S$. The graph $H = (V, E' \cup F)$ is a minimal triangulation of G if and only if $F = \bigcup_{i=1}^{k} F_i$, where F_i is the set of fill edges of a minimal triangulation of $G'[S \cup C_i]$. (1997 [31])

4. A triangulation $H = (V, E \cup F)$ is a minimal triangulation of $G = (V, E)$ if and only if it can be obtained by completing a maximal set of pairwise non-crossing minimal separators of G into cliques. (1997 [38])

5. Let K be a potential maximal clique of $G = (V, E)$, and let $G' = (V, E')$ be the graph obtained from G by completing K into a clique. Let further $C_1, C_2, ..., C_k$ be the connected components of $G \setminus K$, and $S_i = N(C_i)$ for $1 \leq i \leq k$. The graph $H = (V, E' \cup F)$ is a minimal triangulation of G if and only if $F = \bigcup_{i=1}^{k} F_i$, where F_i is the set of fill edges of a minimal triangulation of $G'[S \cup C_i]$. (2001 [12])

6. A tree separator $H = (V, E \cup F)$ defines a minimal triangulation of $G = (V, E)$ if and only if u and v are contained in a BT-separator of G represented by H, for every edge $uv \in F \setminus E$. (2006 here)

As mentioned earlier, the set of BT-separators is the union of minimal separators and potential maximal cliques. We can now merge Characterizations 3 and 5 into a single characterization in the following corollary.

Corollary 2.12 ([12, 31]) *Let K be a BT-separator of $G = (V, E)$, let $G' = (V, E')$ be the graph obtained from G by completing K into a clique. Let further $C_1, C_2, ..., C_k$ be the connected components of $G \setminus K$, and $S_i = N(C_i)$ for $1 \leq$*

$i \leq k$. The graph $H = (V, E' \cup F)$ is a minimal triangulation of G if and only if $F = \bigcup_{i=1}^{k} F_i$, where F_i is the set of fill edges of a minimal triangulation of $G'[S \cup C_i]$.

Characterization 4 can also be generalized to include BT-separators.

Characterization 2.13 *A triangulation $H = (V, E \cup F)$ is a minimal triangulation of $G = (V, E)$ if and only if it can be obtained by completing a maximal set of pairwise non-crossing BT-separators of G into cliques.*

Proof. Let $\mathcal{K}$ be a maximal set of pairwise non-crossing BT-separators of G. We will now use the set $\mathcal{K}$ to find a maximal set $\mathcal{S}$ of non-crossing minimal separators. For every BT-separator $K \in \mathcal{K}$, and for every connected component C of $G \setminus K$ add the minimal separator $N(C)$ to $\mathcal{S}$. Then all minimal separators in $\mathcal{S}$ are pairwise non-crossing. We will prove this by contradiction, where we assume that the minimal separators S and T in $\mathcal{S}$ are crossing. By [12] all minimal separators in a BT-separator are pairwise non-crossing, so S and T are added to $\mathcal{S}$ using two different BT-separators K_S and K_T. The BT-separators K_S and K_T are now crossing, since $S = N(C_S)$ and $T = N(C_T)$ for a component C_S of $G \setminus K_S$ and a component C_T of $G \setminus K_T$. This is a contradiction since all BT-separators in $\mathcal{K}$ are pairwise non-crossing, and K_S and K_T are contained in $\mathcal{K}$. Now back to the main proof. The set $\mathcal{S}$ is also a maximal set of minimal separators, since $\mathcal{K}$ is a maximal set of BT-separators, and minimal separators are BT-separators. Thus, any minimal separator that could be added to the set $\mathcal{S}$ could also be added to $\mathcal{K}$, which contradicts that $\mathcal{K}$ is a maximal set. By Characterization 4 it follows that a minimal triangulation of G is obtained by completing every BT-separator in $\mathcal{K}$ into a clique.

Let H be a minimal triangulation of G. By Characterization 4 we know that H can be obtained by completing a maximal set $\mathcal{S}$ of pairwise non-crossing minimal separators into cliques. Since minimal separators are also BT-separators, add every separator of $\mathcal{S}$ to the set $\mathcal{K}$, which contains BT-separators. Let us further add BT-separators to $\mathcal{K}$, such that $\mathcal{K}$ becomes a maximal set of pairwise non-crossing BT-separators. Notice that for every BT-separator $K \in \mathcal{K}$, and for every connected component C of $G \setminus K$ the minimal separator $N(C)$ is contained in $\mathcal{S}$, since K is not crossing any BT-separator in $\mathcal{S}$, and $\mathcal{S}$ is a maximal set of non-crossing minimal separators. Thus, there exists a maximal set $\mathcal{K}$ of non-crossing BT-separators, such that H is obtained by completing every BT-separator in $\mathcal{K}$ into a clique. ■

Let us finally prove Characterization 6 which is new here.

Characterization 2.14 *A tree separator $H = (V, E \cup F)$ defines a minimal triangulation of $G = (V, E)$ if and only if u and v are contained in a BT-separator of G represented by H, for every edge $uv \in F$ where $E \cap F = \emptyset$.*

Proof. Let H be a tree separator of G, such that for every edge $uv \in F$, the vertices u and v are contained in a BT-separator of G represented by H. Let $\mathcal{K}$ be the set of BT-separators of G, which is represented by H. Only non-crossing BT-separators can be represented by a tree separator, so every pair of BT-separators in $\mathcal{K}$ are non-crossing. Notice that every BT-separator K of G represented by H is also a BT-separator of H, since every edge of G is contained in H, and every pair of vertices separated by K in G, is non adjacent in H. By Lemma 2.7 and Corollary 2.9 every BT-separator of H is a clique, and H is chordal. Thus, a triangulation of G can be obtained by completing the BT-separators in $\mathcal{K}$ into cliques, and by Characterization 2.13 H is a minimal triangulation of G.

Let H be a minimal triangulation of G. Then there exists a set $\mathcal{K}$ of pairwise non-crossing BT-separators of G, such that H is obtained by completing these BT-separators into cliques (Characterization 2.13). Thus, for every $uv \in F$ there exits a BT-separator in $\mathcal{K}$, which contains both u and v. ■

As we have seen in this section, tree decompositions and potential maximal cliques can be considered as, or defined by, a set of pairs of vertices which are not adjacent. Most papers that discuss some kind of triangulation focus on finding fill edges and not pairs of vertices to separate in the final triangulation. There are at least two reasons for this. The first is that the triangulation problem is defined through a set of fill edges and not pairs of vertices to separate, and secondly it is harder to draw examples when the decision is to separate pairs of vertices and not add fill edges. But these two approaches are not necessarily equal. If some subset of the set of pairs of vertices we have decided to separate defines a BT-separator, then we might as well complete the BT-separator into a clique. But if this set of pairs of vertices only defines a separator and not a BT-separator, then this can be considered as a subproblem of the general triangulation problem. The reason why we can consider this as a subproblem is that some choices are made, and thus there are fewer final triangulations to choose among. Some examples of algorithms using this separating approach can be found in [6],[9], and [26].

3 History and relation to introduction

Apart from this introduction, this thesis consists of five papers, of which all are submitted to journals, and four are accepted for publication[3]. With the exception of Paper III, the main results of all of these papers are presented at some conference. Each of papers II and IV is a final full version of a single paper first presented at a conference. Each of papers I and V is a journal paper based on two separate conference papers. Each of the journal papers I and V was the result of merging a conference paper containing our results with another

[3]Some of the papers attached to this thesis have minor editing changes compared to the submitted version.

conference paper by different authors. Since a thesis should contain the most recent, well written, and full version of each result, we attach the merged journal papers in this thesis, and not the preliminarily conference versions.

This section contains five subsections, one for each paper. Each subsection has two purposes. The first is to tell a bit of the history behind the result. By history we mean to identify the conference papers (if any) the paper is based on, and in some cases also how the project started. The second purpose is to emphasize the relation to the introduction, by explaining how tree decompositions, minimal separators, and elimination orderings are used to obtain the results.

3.1 (Paper I [6]) Tree decomposition as a tool to compute minimal triangulations efficiently

Several different minimal triangulation algorithms have been presented since the mid seventies, where the LB-Triang algorithm by Berry [3] is one of the more recent. LB-Triang was first presented at ACM-SIAM Symposium on Discrete Algorithms (SODA) in 1999 [3], and it is based on Lekkerkerker and Boland's [34] characterization of chordal graphs. This characterization states that a graph is chordal if and only if every minimal separator contained in the neighborhood of a vertex is a clique. The LB-Triang algorithm describes a procedure for adding fill edges, such that this property is established. The time bound for LB-Triang was claimed in [3] to be $O(nm)$ without a proof, and the existence of such an implementation remained unproved until our result in 2002. A straight forward implementation of LB-Triang gives $O(nm')$ time complexity, where m' is the number of edges in the resulting minimal triangulation. Since $O(nm)$ time algorithms already existed [36, 41], the $O(nm')$ time complexity was not satisfactory, although the LB-Triang algorithm has many other nice properties.

An $O(nm)$ time implementation of LB-Triang was presented by Heggernes and Villanger at European Symposium on Algorithms (ESA) [28] in 2002. The implementation is based on a tree decomposition which preserves the information about the set of minimal separators found so far by the algorithm. This tree decomposition structure allows us to compute the information we need from the new fill edges in $O(m)$ time, and thus we obtain an $O(nm)$ time algorithm.

Paper I is based on the two conference papers [3] and [28]. The use of minimal separators in the algorithm, and the use of tree decompositions in the $O(nm)$ implementation, relate this paper to the structures mentioned in the introduction. We will now discuss how, and a bit why, tree decompositions make the difference in the time complexity. The LB-Triang algorithm processes each vertex of the graph in some order. This order is part of the input in the $O(nm)$ implementation, but can be provided in an on-line fashion for the $O(nm')$ implementation. For each vertex, LB-Triang completes the minimal separators contained in the neighborhood of the current vertex into a clique, and then this procedure is repeated with the next vertex in the ordering. The final result is that every minimal separator contained in the neighborhood of some vertex is a clique. Thus, it follows from [34] that the resulting graph is a triangulation.

Completing minimal separators into cliques is the operation that requires $O(m')$ time for each separator in a naive implementation, and gives an $O(n^2m')$ algorithm. This can easily be improved to $O(nm')$ by recognizing duplicate separators, and complete each minimal separator into a clique only once. To improve the time bound further we need the observation that the added fill edges are only required to compute $N(u)$ if u is the next vertex to be processed. Thus, an $O(nm')$ time algorithm can also be obtained by scanning every minimal separator containing u once from a list containing only unique minimal separators, and in this way compute $N(u)$. Another important observation is that each minimal separator separates the remaining set of separators into subsets. Unlike a list structure, a tree decomposition can be used to store the minimal separators, while preserving this information. This extra information enables us to compute $N(u)$ from a subset of the minimal separators containing u, and to bound the sum of these to m. As a result we obtain an $O(nm)$ time implementation.

3.2 (Paper II [8]) A vertex incremental approach to compute minimal triangulations

An early version of the results of Paper II was presented at the International Symposium on Algorithms and Computation (ISAAC) [7] in 2003. The project resulting in Paper II started with the following question: Given a graph $G = (V, E)$, and a partition V_1, V_2 of V, such that $G[V_1]$ and $G[V_2]$ are chordal graphs, does there always exist a minimal triangulation H of G, such that $u \in V_1$ and $v \in V_2$, for every fill edge uv in H? As we can see in the example of Figure 2, there might not even be such a triangulation. This negative result removes the possibility of finding a minimal triangulation of a graph G, by triangulating $G[V_1]$ and $G[V_2]$ and then adding fill edges between V_1 and V_2, for any partition V_1, V_2 of V.

Let us now try to reformulate the question, such that we can obtain a positive result. By studying the graphs in Figure 2, we can observe that such a triangulation can be obtained if either V_1 or V_2 induces a clique. Let V_1 be the vertex set that induces a clique, and let us fix the size of V_1 to one. Fixing the size of V_1 to one has two advantages, the first is that all new fill edges are incident to the single vertex in V_1. The second is a bit more complicated. Let V_1 and V_2 be a partition of V for a graph $G = (V, E)$, and let H_2 be a minimal triangulation of $G[V_2]$. For a vertex $u \in V_1$ let $\{u\}, V_2$ be a partition of the vertices in $G[V_2 \cup \{u\}]$, notice that H_2 is still a minimal triangulation of $G[V_2]$. A minimal triangulation H_u of $G[V_2 \cup \{u\}]$ can now be obtained by supplying the fill edges of H_2 by some fill edges incident to u. In this way we obtain a new partition $V_1 \setminus \{u\}, V_2 \cup \{u\}$ of

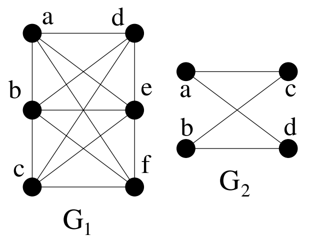

Figure 2: Let the partition of V in G_1 be as follows $V_1 = \{a, b, c\}$ and $V_2 = \{d, e, f\}$. No triangulation of G_1 can be obtained by adding edges between V_1 and V_2, since all these edges are already presented, and a, d, c, f, a is a chordless cycle of length four. If we allow $G[V_1]$ and $G[V_2]$ to be disconnected, then a smaller example can be found. By using the partition $V_1 = \{a, b\}$ and $V_2 = \{c, d\}$ of V in G_2 we obtain a similar example as obtained by G_1. Every edge between vertices in V_1 and V_2 is already present and a, c, b, d, a is a chordless cycle of length four. Unlike the example using G_1, the vertex sets V_1 and V_2 are disconnected, and of size two.

V, where H_u is a minimal triangulation of $G[V_2 \cup \{u\}]$. A minimal triangulation of G can now be obtained by repeating this until $V_1 = \emptyset$. The topic discussed in Paper II, corresponds exactly to this procedure, where $|V_2| \leq 1$ before the first iteration.

Paper II presents a single algorithm that can be used to compute both minimal triangulations and maximal subtriangulations of an input graph. A graph $H = (V, D)$, is a *maximal subtriangulation* of $G = (V, E)$ if D is an inclusion maximal subset of E, such that H is a chordal graph. This algorithm is based on a vertex incremental approach, a new technique in triangulation algorithms, but an already used technique on other problems like different types of graph recognition [17] and [32]. Since then the incremental approach has proved useful on other minimal completion problems [25].

The algorithm in Paper II uses the minimal separators in the chordal subgraph to find the new fill edges every time a new vertex u is introduced to the chordal subgraph. The fill edges incident to u are obtained by adding to $N(u)$ the union of a set of minimal separators in the chordal subgraph. In order to compute this set efficiently we need data structure that allows us to find and compute the union of these minimal separators efficiently. This is obtained by representing a tree decomposition in a new way.

The paper presents also a new way to implement tree decompositions. Unlike a regular representation of a tree decomposition, which represents each tree node as a vertex set, we only store the difference between tree nodes. This enables us to manipulate the tree decomposition such that every tree edge between two tree nodes are minimal u, v-separators for some vertex v, and to compute the union of these minimal separators in $O(n)$ time, which is not possible in a standard vertex set representation. When this is combined with an amortized time analysis an $O(nm)$ time algorithm for minimal triangulation and maximal subtriangulation is obtained.

3.3 (Paper III [45]) Comparing minimal triangulation algorithms

Lex M [41] is one of the first two minimal triangulation algorithms published in 1976, and is based on a lexicographic breadth-first search. Almost 30 years later Berry, Blair, Heggernes, and Peyton presented a similar minimal triangulation algorithm called MCS-M [4]. MCS-M combines the cardinality labeling of neighbors used in MCS [43] (a linear time recognition algorithms for chordal graphs), with the labeling along paths used in Lex M. Both algorithms find minimal triangulations by producing a minimal elimination ordering. Some of the similarities between these algorithms can be exemplified with a simple cycle of length six. Both algorithms are capable of producing the same set of minimal triangulations, and none of them are able to produce minimal triangulations that are the result of only adding fill edges incident to a single vertex, or to create a minimal triangulation that is the result of adding three fill edges that define a cycle of length three. The difference between Lex M and MCS-M can be exemplified by two simple cycles of length five sharing a single vertex. If the vertex contained in both cycles are chosen to be the last in the elimination ordering, then the set of elimination orderings produced by Lex M and MCS-M are disjoint sets. Despite this, the same set of minimal triangulations is obtained by these sets of elimination orderings. A natural question emerges, which is also left as an open question in [4]: Does Lex M and MCS-M produce the same set of triangulations?

Paper III attends this problem, and shows that the two algorithms actually produce the same set of triangulations. The idea behind the proof is as follows. Each minimal triangulation can be obtained by a set of minimal elimination orderings, and any pair of orderings from this set are said to be equivalent. Paper III proves that an ordering can be obtained by MCS-M if and only if Lex M can produce an equivalent ordering from the same input graph.

3.4 (Paper IV [26]) Combining several new techniques into a faster minimal triangulation algorithm

Until the spring of 2004 it was not known whether or not minimal triangulations could be computed with better time bound than $O(nm)$, which is $O(n^3)$ for dense graphs. Attempts to improve the time complexity of known minimal triangulation algorithms below $O(n^3)$ mainly meet two obstacles. The first is to avoid searching the input graph $O(n)$ times, and the second is to compute the set of new fill edges after each iteration. In most cases these fill edges can be obtained by completing a set of vertex sets into cliques. This problem can be restated as a binary multiplication problem, and thus solved in time $O(n^{2.376})$ [16]. Independently of each other, two research groups used this technique to find new and faster minimal triangulation algorithms for dense graphs.

Kratsch and Spinrad [33] designed an $O(n^{2.69})$ time algorithm, which is a new implementation of Lex M. It executes several steps of Lex M at once, and then uses binary matrix multiplication to update all the labels in one operation. Heggernes, Telle, and Villanger [26] designed an $o(n^{2.376})$ time algorithm, by using minimal separators and potential maximal cliques to partition the triangulation problem into subproblems, and used binary matrix multiplication to find the new subproblems. The second result is presented as Paper IV. A preliminary version of Paper IV appeared at ACM-SIAM Symposium on Discrete Algorithms (SODA) [27] in 2005.

From [31] and [12] it is known that minimal separators and potential maximal cliques can be used to separate the minimal triangulation problem into independent subproblems. The problem with this approach is to bound the depth of the recursion tree, since it might be $O(n)$ in the worst case. In the algorithm presented in Paper IV, the height of this recursion tree is $O(\log n)$. This is obtained by a partitioning algorithm that finds a set of non-crossing minimal separators, such that no subproblem contains more than some fraction of the non edges in the input graph. Let $O(n^\alpha)$ be the time bound of multiplying two $n \times n$ matrices. Currently the lowest value of α is $2.375 < \alpha < 2.376$ by the algorithm of Coppersmith and Winograd [16]. An $O(n^\alpha \log n)$ time algorithm for minimal triangulation can now be obtained by implementing the balanced partition algorithm to run in $O(n^2 - |E|)$ time for an input graph $G = (V, E)$, and then use matrix multiplication to complete the minimal separators into cliques.

Even though tree decompositions are not used directly in this algorithm, the algorithm can be considered as an algorithm that refines a tree decomposition, until it defines a minimal triangulation. Unlike the algorithm, one of the proof used to claim the time complexity heavily depends on tree decompositions. This makes tree decompositions to one of the basic structures used to obtain this result.

3.5 (Paper V [20]) Computing treewidth in exponential time using potential maximal cliques

Computing the treewidth of a graph is a problem that has received a lot of attention, but it is quite recent that exact exponential time algorithms for treewidth have been published. The first was an exact algorithm for treewidth and minimum fill-in by Fomin, Kratsch, and Todinca [19], presented at International Colloquium on Automata, Languages and Programming (ICALP) 2004. This algorithm requires every potential maximal clique of G as part of the input, and computes the treewidth of G in time $O(n^3\Pi_G)$, where Π_G is the number of potential maximal cliques in G. Listing the potential maximal cliques of G is the most time consuming operation in [19], and thus the time complexity of the algorithm becomes $O^*(1.9601^n)$, which is the time required to list all potential maximal cliques of the input graph. Improving the time required for listing the potential maximal cliques would thus improve the time complexity of the algorithm, and this was also left as an open question in [19].

The problem of finding a better upper bound for the number of potential maximal cliques in a graph, and to list these more efficiently, is addressed in [44], which will be presented at Latin American Theoretical Informatics Symposium (LATIN) 2006. The $O^*(1.9601^n)$ time algorithm for listing all potential maximal cliques presented in [19], is improved to $O^*(1.8899^n)$, and we show that the number of potential maximal cliques contained in a graph is $O^*(1.8135^n)$. The second result will be the new running time if somebody manages to list all potential maximal cliques of a graph, with a polynomial delay for each potential maximal clique, i.e. listing all the potential maximal cliques in $O^*(\Pi_G)$ time. Such algorithms exist for listing minimal separators [5, 30], so it is an interesting open question if this type of algorithms exists for potential maximal cliques.

Paper V is obtained by combining the results of [19] and [44]. The time bound for algorithm in [19] is obtained by proving that every potential maximal clique can be defined by a set of $2n/5$ vertices. In [20] this is improved, such that any potential maximal clique can be defined by at most $n/3$ vertices, which improves the time bound for listing potential maximal cliques, and thus also for finding the treewidth, to $O^*(1.8899^n)$. One of the results of [13] is that all potential maximal cliques of a graph can be found by finding the *nice* potential maximal cliques, and then use these to generate the potential maximal cliques that are not nice. Another result in [13] is that any nice potential maximal clique can be defined by two crossing minimal separators. These separators and other observations are used in Paper V to partition the vertex set of the graph into three independent sets, such that any of these can be used to define the potential maximal clique.

References

[1] S. Arnborg, D.G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8:277–284, 1987.

[2] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database systems. *Journal of the Association for Computing Machinery*, 30:479–513, 1983.

[3] A. Berry. A wide-range efficient algorithm for minimal triangulation. In *SODA: Proceedings of the annual ACM-SIAM symposium on Discrete algorithms*, pages 860–861. Society for Industrial and Applied Mathematics, 1999.

[4] A. Berry, J. Blair, P. Heggernes, and B. Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287–298, 2004.

[5] A. Berry, J.P. Bordat, and O. Cogis. Generating all the minimal separators of a graph. *International Journal of Foundations of Computer Science*, 11(3):397–403, 2000.

[6] A. Berry, J.P. Bordat, P. Heggernes, G. Simonet, and Y. Villanger. A wide-range algorithm for minimal triangulation from an arbitrary ordering. *Journal of Algorithms*, 58(1):33–66, 2006.

[7] A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for dynamically maintaining chordal graphs. In *ISAAC*, volume 2906 of *Lecture Notes in Computer Science*, pages 47 – 57. Springer Verlag, 2003.

[8] A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for maintaining chordality. *Discrete Mathematics*, 2006. To appear.

[9] J.R.S. Blair, P. Heggernes, and J.A. Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 250:125–141, 2001.

[10] J.R.S. Blair and B.W. Peyton. An introduction to chordal graphs and clique trees. In J.A. George, J.R. Gilbert, and J.W.H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, pages 1–30. Springer Verlag, 1993. IMA Volumes in Mathematics and its Applications, Vol. 56.

[11] H.L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[12] V. Bouchitté and I. Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM Journal on Computing*, 31:212–232, 2001.

[13] V. Bouchitté and I. Todinca. Listing all potential maximal cliques of a graph. *Theoretical Computer Science*, 276(1-2):17–32, 2002.

[14] P. Buneman. A characterization of rigid circuit graphs. *Discrete Mathematics*, 9:205–212, 1974.

[15] F.R.K. Chung and D. Mumford. Chordal completions of planar graphs. *Journal of Combinatorial Theory B*, 62(1):96–106, 1994.

[16] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[17] D.G. Corneil, Y. Perl, and L.K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal on Computing*, 14(4):926–934, 1985.

[18] G.A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg*, 25:71–76, 1961.

[19] F.V. Fomin, D. Kratsch, and I. Todinca. Exact (exponential) algorithms for treewidth and minimum fill-in. In *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 568–580. Springer Verlag, 2004.

[20] F.V. Fomin, D. Kratsch, I. Todinca, and Y. Villanger. Exact (exponential) algorithms for treewidth and minimum fill-in. *Submitted to SIAM Journal on Computing*, 2005.

[21] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15:835–855, 1965.

[22] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory B*, 16:47–56, 1974.

[23] J. Gustedt. On the pathwidth of chordal graphs. *Discrete Applied Mathematics*, 45(3):233–248, 1993.

[24] P. Heggernes. Minimal triangulations of graphs: A survey. *Discrete Mathematics*, 2006. To appear.

[25] P. Heggernes, K. Suchan, I. Todinca, and Y. Villanger. Minimal interval completions. In *ESA*, volume 3669 of *Lecture Notes in Computer Science*, pages 403–414. Springer Verlag, 2005.

[26] P. Heggernes, J.A. Telle, and Y. Villanger. Computing minimal triangulations in time $O(n^{\alpha} logn) = o(n^{2.376})$. *SIAM Journal on Discrete Mathematics*, 19(4):900–913, 2005.

[27] P. Heggernes, J.A. Telle, and Y. Villanger. Computing minimal triangulations in time $O(n^{\alpha} logn) = o(n^{2.376})$. In *SODA: Proceedings of the annual ACM-SIAM symposium on Discrete algorithms*, pages 907–916. Society for Industrial and Applied Mathematics, 2005.

[28] P. Heggernes and Y. Villanger. Efficient implementation of a minimal triangulation algorithm. In *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 550–561. Springer Verlag, 2002.

[29] P. Heggernes and Y. Villanger. Simple and efficient modifications of elimination orderings. In *PARA*, volume 3732 of *Lecture Notes in Computer Science*, pages 788–797. Springer Verlag, 2004.

[30] T. Kloks and D. Kratsch. Listing all minimal separators of a graph. *SIAM Journal on Computing*, 27(3):605–613, 1998.

[31] T. Kloks, D. Kratsch, and J. Spinrad. On treewidth and minimum fill-in of asteroidal triple-free graphs. *Theoretical Computer Science*, 175:309–335, 1997.

[32] N. Korte and R.H. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM Journal on Computing*, 18(1):68–81, 1989.

[33] D. Kratsch and J. Spinrad. Minimal fill in $o(n^3)$ time. *Discrete Mathematics*, 2006. To appear.

[34] C.G. Lekkerkerker and J.C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamentals of Mathematics*, 51:45–64, 1962.

[35] G.S. Lueker and K.S. Booth. A linear time algorithm for deciding interval graph isomorphism. *Journal of the Association for Computing Machinery*, 26(2):183–195, 1979.

[36] T. Ohtsuki. A fast algorithm for finding an optimal ordering in the vertex elimination on a graph. *SIAM Journal on Computing*, 5:133–145, 1976.

[37] T. Ohtsuki, L.K. Cheung, and T. Fujisawa. Minimal triangulation of a graph and optimal pivoting ordering in a sparse matrix. *Journal of Mathematical Analysis and Applications*, 54:622–633, 1976.

[38] A. Parra and P. Scheffler. Characterizations and algorithmic applications of chordal graph embeddings. *Discrete Applied Mathematics*, 79:171–188, 1997.

[39] S. Parter. The use of linear graphs in Gauss elimination. *SIAM Review*, 3:119–130, 1961.

[40] N. Robertson and P. Seymour. Graph minors II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[41] D. Rose, R.E. Tarjan, and G. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5:146–160, 1976.

[42] D.J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R.C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, New York, 1972.

[43] 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 Journal on Computing*, 13:566–579, 1984.

[44] Y. Villanger. Improved exponential-time algorithms for treewidth and minimum fill-in. In *LATIN*, Lecture Notes in Computer Science. Springer Verlag, 2006. To appear.

[45] Y. Villanger. Lex M versus MCS-M. *Discrete Mathematics*, 2006. To appear.

[46] J. Walter. *Representations of rigid cycle graphs.* PhD thesis, Wayne State University, USA, 1972.

[47] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, 1981.

Paper I

A wide-range algorithm for minimal triangulation from an arbitrary ordering

Anne Berry[*] Jean-Paul Bordat[†] Pinar Heggernes[‡]
Geneviève Simonet[†] Yngve Villanger[‡]

Abstract

We present a new algorithm, called LB-Triang, which computes minimal triangulations. We give both a straightforward $O(nm')$ time implementation and a more involved $O(nm)$ time implementation, thus matching the best known algorithms for this problem.

Our algorithm is based on a process by Lekkerkerker and Boland for recognizing chordal graphs which checks in an arbitrary order whether the minimal separators contained in each vertex neighborhood are cliques. LB-Triang checks each vertex for this property and adds edges whenever necessary to make each vertex obey this property. As the vertices can be processed in any order, LB-Triang is able to compute any minimal triangulation of a given graph, which makes it significantly different from other existing triangulation techniques.

We examine several interesting and useful properties of this algorithm, and give some experimental results.

1 Background and motivation

Computing a *triangulation* consists in embedding a given graph into a *triangulated*, or *chordal*, graph by adding a set of edges called a *fill*. If no proper subset of the fill can generate a chordal graph when added to the given graph, then this fill is said to be *minimal*, and the resulting chordal graph is called a *minimal triangulation*. The fill is said to be *minimum* if its cardinality is the smallest over all possible minimal fills, and the corresponding triangulation is called a *minimum*

[*]LIMOS UMR CNRS 6158, Ensemble Scientifique des Cézeaux, Université Blaise Pascal, F-63 170 Aubière, France. berry@isima.fr

[†]LIRMM, 161 Rue Ada, F-34392 Montpellier, France. bordat@lirmm.fr simonet@lirmm.fr

[‡]Department of Informatics, University of Bergen, N-5020 Bergen, Norway. pinar@ii.uib.no yngvev@ii.uib.no

triangulation. The motivation for finding a fill of small cardinality originates from the solution of sparse symmetric systems [14, 27, 28], but the problem has applications in other areas of computer science [2, 9, 16, 20], and has been studied by many researchers during the last decades.

Given a graph G and an ordering α on its vertices, hereafter denoted by (G, α), one way of computing a triangulation [13] is the following *Elimination Game* by Parter [24]: Repeatedly choose the next vertex x in order α, and add the edges that are necessary to make the neighborhood of x into a clique in the remaining graph (thus making vertex x *simplicial* in the resulting graph), before deleting x. The edges that are added at each step altogether define the fill edges of this process. The triangulated graph obtained by adding this fill to the original graph G is denoted by G^+_α. In this paper, we will refer to such graphs as *simplicial filled graphs.* Different orderings of the input graph result in different simplicial filled graphs. An ordering α on G is called a *perfect elimination ordering* (PEO) if $G^+_\alpha = G$. Consequently, α is a PEO of G^+_α. If G^+_α is a minimal triangulation of G, then α is called a *minimal elimination ordering (*MEO*)* of G [22].

The elimination game was originally introduced [24] in order to describe the fill added during symmetric factorization of the associated matrix M of G (i.e., the non-zero pattern of M is the adjacency matrix of G). Fulkerson and Gross [13] showed later that triangulated graphs are exactly the class of graphs that have perfect elimination orderings; hence all simplicial filled graphs are triangulated. Simplicial filled graphs are in general neither minimal nor minimum triangulations of the original graph, and the size of the introduced fill depends on the order in which the vertices are processed by the elimination game. Computing an order that will result in a minimum fill is NP-hard on general graphs [31]. Several heuristics have been proposed for finding elimination orderings that produce small fill, such as Minimum Degree [27] and Nested Dissection [14]. Although these are widely used and produce good orderings in practice, they do not guarantee minimum or minimal fill.

In 1976 Ohtsuki, Cheung, and Fujisawa [22], and Rose, Tarjan, and Lueker [28] simultaneously and independently showed that a minimal triangulation can be found in polynomial time, presenting two different algorithms of $O(nm)$ time for this purpose, where n is the number of vertices and m is the number of edges of the input graph G. No minimal triangulation algorithm has achieved a better time bound since these results. One of these algorithms, LEX M [28], has become one of the classical algorithms for minimal triangulation. Despite its complexity merits, LEX M yields only a restricted family of minimal triangulations, and the size of the resulting fill is not small in general. Recently a new algorithm for computing minimal triangulations, which can be regarded as a simplification of LEX M, has been introduced [5]. This algorithm, called MCS-M, has the same asymptotic time complexity and the same kind of properties regarding fill as LEX M.

In order to combine the idea of small fill with minimal triangulations, Minimal Triangulation Sandwich Problem was introduced by Blair, Heggernes, and Telle [7]: Given (G, α), find a minimal triangulation H of G such that $G \subseteq H \subseteq G^+_\alpha$. This approach enables the user to affect the produced fill by supplying a desired elimination ordering to the algorithm, while computing a triangulation which is minimal. In [7] the authors present an algorithm that removes fill edges from G^+_α in order to solve this problem. The complexity of their algorithm is $O(f(m+f))$, where f is the number of filled edges in the initial simplicial filled graph G^+_α, thus the algorithm works fast for elimination orderings resulting in low fill. Dahlhaus [11] later presented an algorithm for solving the same problem with a time complexity evaluated as $O(nm)$, which uses a clique tree representation of the graph as an intermediate structure. The most recent among algorithms solving the Minimal Triangulation Sandwich Problem is presented by Peyton [25]. This algorithm also removes unnecessary fill from a given triangulation, and although it appears fast in practice, no theoretical bound for its runtime is proven.

Using a totally different approach, Berry [4] introduced Algorithm LB-Triang, which, given (G, α), produces a minimal triangulation directly, and also solves the Minimal Triangulation Sandwich Problem. In fact, the ordering need not be chosen beforehand, but can be generated dynamically, allowing an on-line approach and a wide variety of strategies for finding special kinds of fills. LB-Triang gives new insight about minimal triangulations as it is a characterizing algorithm; any minimal triangulation of an input graph can be produced by LB-Triang through some ordering of the vertices. It is the only minimal triangulation algorithm so far that solves the Minimal Triangulation Sandwich Problem directly from the input graph, without removing fill from a given triangulation.

In this paper, we study Algorithm LB-Triang extensively, prove its correctness, and show several of its interesting properties. We prove that any minimal triangulation can be obtained by LB-Triang, and that LB-Triang also directly solves the sandwich problem mentioned above without computing G^+_α. We discuss several variants and implementations of the algorithm, and compare it to other algorithms, both in a theoretical fashion and by performance analysis.

This paper is organized as follows: In Section 2, we give the necessary graph theoretical background and introduce the notations used throughout the paper. Section 3 presents some recent research results on minimal triangulation that will be the basis for our proofs. Section 4 introduces LB-Triang and proves its correctness. In Section 5, we examine various properties of this minimal triangulation process. Section 6 gives a complexity analysis of a straightforward implementation, and in Section 7 we describe an implementation which improves the complexity to $O(nm)$. We give some experimental results in Section 8, and conclude in Section 9.

2 Preliminaries

All graphs in this work are undirected and finite. A graph is denoted $G = (V, E)$, with $n = |V|$, and $m = |E|$. $G(A)$ is the subgraph induced by a vertex set $A \subseteq V$, but we often denote it simply by A when there is no ambiguity. A *clique* is a set of vertices that are all pairwise adjacent. An *independent* set of vertices is a set of vertices that are pairwise non-adjacent.

For all the following definitions, we will omit subscript G when it is clear from the context which graph we work on. The *neighborhood* of a vertex x in G is $N_G(x) = \{y \neq x \mid xy \in E\}$; $N_G[x] = N_G(x) \cup \{x\}$. The neighborhood of a set of vertices A is $N_G(A) = \cup_{x\in A} N_G(x) \setminus A$. A vertex is *simplicial* if its neighborhood is a clique. We say that we *saturate* a set of vertices X in graph G if we add the edges necessary to make $G(X)$ into a clique.

For a connected graph $G = (V, E)$ with $X \subseteq V$, $\mathscr{C}_G(X)$ denotes the set of connected components of $G(V \setminus X)$. $S \subset V$ is called a *separator* if $|\mathscr{C}(S)| \geq 2$, an *ab-separator* if a and b are in different connected components of $\mathscr{C}(S)$, a *minimal ab-separator* if S is an ab-separator and no proper subset of S is an ab-separator, and a *minimal separator* if there is some pair $\{a, b\}$ such that S is a minimal ab-separator. Equivalently, S is a minimal separator if there exist two distinct components C_1 and C_2 in $\mathscr{C}(S)$ such that $N(C_1) = N(C_2) = S$ (such components are called *full* component). $\mathscr{S}(G)$ denotes the set of minimal separators of G. If G is not connected, we call S a minimal separator iff it is a minimal separator of a connected component of G. A minimal separator S of G is called a *clique minimal separator* if $G(S)$ is a clique.

A *chord* of a cycle is an edge connecting two non-consecutive vertices of the cycle. A graph is *triangulated*, or *chordal*, if it contains no chordless cycle of length ≥ 4.

3 Triangulated Graphs and Triangulations

3.1 Triangulated Graphs

Triangulated graphs were defined as extensions of a tree. The first significant results on this class were obtained by two contemporary and independent works, due to Dirac [12], and Lekkerkerker and Boland [21], which present similar results, but with a different approach. Dirac defined the concept of minimal separator, which extends the notion of articulation node in a tree, and used this to characterize triangulated graphs:

Characterization 3.1 (Dirac [12]) *A graph G is triangulated iff every minimal separator in G is a clique.*

Dirac also proved that every triangulated graph which is not a clique has at least two nonadjacent simplicial vertices. Using this, Fulkerson and Gross [13] observed that any simplicial vertex can be removed from a graph without destroying chordality, yielding the following characterization for triangulated graphs:

Characterization 3.2 (Fulkerson and Gross [13]) *A graph is triangulated iff it has a* PEO.

Using this characterization for the recognition of triangulated graphs requires computing a PEO. This can be done in linear time [28, 29].

Lekkerkerker and Boland [21] used a quite different approach to characterize triangulated graphs. They introduced the notion of substars of a vertex x, and they characterized triangulated graphs as graphs for which each substar is a clique. A *substar* S of x is a subset of $N(x)$ such that $S = N(C)$ for a connected component C of $G(V \setminus N[x])$. We now know that these substars are precisely the minimal separators contained in the vertex neighborhoods. Since in a triangulated graph, every minimal separator belongs to a vertex neighborhood, this result is in fact closely related to Dirac's characterization. We will restate the characterization of Lekkerkerker and Boland using the following definition. (The abbreviation LB stands for Lekkerkerker-Boland.)

Definition 3.3 *A vertex x is* LB-simplicial *iff every minimal separator contained in the neighborhood of x is a clique.*

Characterization 3.4 (Lekkerkerker and Boland [21]) *A graph is triangulated iff every vertex is LB-simplicial.*

It is interesting to note that Lekkerkerker and Boland used this characterization both in a static and in a dynamic way, as they also proved that a triangulated graph can be recognized by repeatedly choosing any vertex, checking it for LB-simpliciality, and removing it, until no vertex is left. Thus they had established, several years before Fulkerson and Gross, a characterizing elimination scheme for triangulated graphs. They estimated the complexity as $O(n^4)$, but as it will be obvious from the discussion in Section 6, this algorithm can be implemented in $O(nm)$, which would have solved their problem of recognizing interval graphs in $O(n^3)$.

Although triangulated graphs can now be recognized in linear time using MCS, Lekkerkerker and Boland's algorithm has interesting aspects, one of which is that it can process the vertices in an *arbitrary* order, meaning in particular that this check can be done in parallel for all vertices simultaneously. All the vertices in a triangulated graph are LB-simplicial, but not necessarily simplicial, and therefore finding a PEO cannot be parallelized in the same way as the independent check for LB-simpliciality of all vertices simultaneously. Recently, the algorithm of Lekkerkerker and Boland has been extended to the characterization and recognition of weakly triangulated graphs by Berry, Bordat and Heggernes [6]. In this paper, we will use it to compute a minimal triangulation of an arbitrary graph.

3.2 Minimal Triangulation

Computing a minimal triangulation requires computing a fill F such that no proper subset of F will give a triangulation. The classical triangulation techniques force the graph into respecting Fulkerson and Gross' characterization, but recent approaches have been made in the direction of forcing the graph into respecting Dirac's characterization.

Recent research has shown that minimal triangulation is closely related to minimal separation [3, 19, 23, 30]: the process of repeatedly choosing a minimal separator and adding edges to make it into a clique until all the minimal separators of the resulting graph are cliques, will compute a minimal triangulation. Conversely, any minimal triangulation can be obtained by some instance of this process. A graph has, in general, an exponential number of minimal separators, and a triangulated graph has less than n [26]. The process described above chooses at most $n-1$ minimal separators of the input graph and saturates them. Whenever a saturation step is executed, this causes a number of initial minimal separators to disappear from the graph. Thus, during the process, the set of minimal separators shrinks until it reaches its terminal size of at most $n-1$. The minimal separators that disappear are well defined. Kloks, Kratsch and Spinrad [18] introduced the notion of *crossing separators*, and they showed that a minimal triangulation corresponds to the saturation of a set of non-crossing minimal separators. Parra and Scheffler [23] extended this result to characterize minimal triangulations as graphs obtained by saturating a maximal set of pairwise non-crossing minimal separators.

Definition 3.5 (Kloks, Kratsch, and Spinrad [19]) *Let S and T be two minimal separators of G. Then S* crosses *T if there exist two components $C_1, C_2 \in \mathscr{C}(T)$, $C_1 \neq C_2$, such that $S \cap C_1 \neq \emptyset$ and $S \cap C_2 \neq \emptyset$.*

In [23] it is shown that the crossing relation is symmetric. This follows also from Lemma 3.10 below. We compress the results obtained in [3], [19], and [23] into the following:

Property 3.6 *Let G be a graph and let G' be the graph obtained from G by saturating a set $\mathscr{S}$ of pairwise non-crossing minimal separators of G.*

a) A clique minimal separator of G does not cross any minimal separator of G.

b) $\mathscr{S}$ is a set of clique minimal separators of G'.

c) Any clique minimal separator of G is a minimal separator of G'.

d) Any minimal separator of G' is a minimal separator of G.

e) Any set of pairwise non-crossing minimal separators of G' is a set of pairwise non-crossing minimal separators of G.

f) If $\mathscr{S}$ is a maximal set of pairwise non-crossing minimal separators of G then G' is a minimal triangulation of G.

For our proofs, we will need the following extra results concerning the preservation of the minimal separators and of the components of $\mathscr{C}(S)$ and of their neighborhoods.

Observation 3.7 *Let $G = (V, E)$ be a graph and $C, S \subseteq V$. If $C \neq \emptyset$, $C \subseteq V \backslash S$, $G(C)$ is connected and $N(C) \subseteq S$ then $C \in \mathscr{C}(S)$.*

Lemma 3.8 *Let $G = (V, E)$ and $G' = (V, E')$ be graphs such that $E \subseteq E'$, and let $S \subseteq V$. If $\forall C \in \mathscr{C}_G(S)$, $N_G(C) = N_{G'}(C)$ then $\mathscr{C}_G(S) = \mathscr{C}_{G'}(S)$.*

Proof. It is sufficient to show that $\mathscr{C}_G(S) \subseteq \mathscr{C}_{G'}(S)$. Let $C \in \mathscr{C}_G(S)$. $C \neq \emptyset$, $C \subseteq V \setminus S$, $G'(C)$ is connected (because $G(C)$ is connected and $E \subseteq E'$) and $N_{G'}(C) = N_G(C) \subseteq S$ then by Observation 3.7 $C \in \mathscr{C}_{G'}(S)$. ∎

Lemma 3.9 *Let $G = (V, E)$ and $G' = (V, E')$ be graphs such that $E \subseteq E'$, and $x \in V$. If $\forall C \in \mathscr{C}_G(N_G[x])$, $N_G(C) = N_{G'}(C)$ then $N_G(x) = N_{G'}(x)$ and $\mathscr{C}_G(N_G[x]) = \mathscr{C}_{G'}(N_G[x])$.*

Proof. Let us assume that $\forall C \in \mathscr{C}_G(N_G[x])$, $N_G(C) = N_{G'}(C)$. By Lemma 3.8, $\mathscr{C}_G(N_G[x]) = \mathscr{C}_{G'}(N_G[x])$. Suppose that $N_G(x) \neq N_{G'}(x)$. Let $y \in N_{G'}(x) \backslash N_G(x)$ and let C be the component of $\mathscr{C}_G(N_G[x])$ containing y. Then $x \in N_{G'}(C) \setminus N_G(C)$, then $N_G(C) \neq N_{G'}(C)$, which contradicts the initial assumption. ∎

Lemma 3.10 *Let $G = (V, E)$ be a graph, and let S and T be two minimal separators of G. If T does not cross S in G, then there is a component C of $\mathscr{C}(T)$ such that $S \subseteq C \cup N(C)$.*

Proof. T does not cross S in G and there are at least two full components in $\mathscr{C}(S)$ then there is a full component C_1 of $\mathscr{C}(S)$ that does not intersect T. Let C be the component of $\mathscr{C}(T)$ containing C_1. $S = N(C_1)$, so $S \setminus T \subseteq C$ and $S \cap T \subseteq N(C)$, thus $S \subseteq C \cup N(C)$. ∎

Lemma 3.11 *Let G be a graph, let G' be the graph obtained from G by saturating a set $\mathscr{S}$ of minimal separators of G, and let T be a minimal separator of G. If T does not cross any separator of $\mathscr{S}$ in G then $\mathscr{C}_G(T) = \mathscr{C}_{G'}(T)$ and $\forall C \in \mathscr{C}_G(T)$, $N_G(C) = N_{G'}(C)$ (thus T is also a minimal separator of G').*

Proof. Since T does not cross any separator of $\mathscr{S}$ in G then by Lemma 3.10, for any separator S of $\mathscr{S}$ there is a component C of $\mathscr{C}_G(T)$ such that $S \subseteq C \cup N_G(C)$. Then $\forall C \in \mathscr{C}_G(T)$, $N_G(C) = N_{G'}(C)$ and then by Lemma 3.8, $\mathscr{C}_G(T) = \mathscr{C}_{G'}(T)$. This implies that there are also at least two full components in $\mathscr{C}_{G'}(T)$, so T is also a minimal separator of G'. ∎

Lemma 3.12 *Let G be a graph, and let G' be the graph obtained from G by saturating a set $\mathscr{S}$ of pairwise non-crossing minimal separators of G. Then $\forall S \in \mathscr{S}$, $\mathscr{C}_G(S) = \mathscr{C}_{G'}(S)$ and $\forall C \in \mathscr{C}_G(S)$, $N_G(C) = N_{G'}(C)$ (thus S is also a minimal separator of G').*

Proof. Lemma 3.12 immediately follows from Lemma 3.11. ∎

Lemma 3.13 *Let G be a graph, let G' be the graph obtained from G by saturating a set $\mathscr{S}$ of pairwise non-crossing minimal separators of G, and let T be a minimal separator of G'. Then $\mathscr{C}_G(T) = \mathscr{C}_{G'}(T)$ and $\forall C \in \mathscr{C}_G(T)$, $N_G(C) = N_{G'}(C)$ (thus T is also a minimal separator of G).*

Proof. By Property 3.6 b), for any S in $\mathscr{S}$, S is a clique minimal separator of G', then by Property 3.6 a), S does not cross T in G'. Then T does not cross S in G', and since $\mathscr{C}_G(S) = \mathscr{C}_{G'}(S)$ by Lemma 3.12, T does not cross S in G. We conclude with Lemma 3.11. ∎

Lemma 3.14 *Let G be a graph and let G' be the graph obtained from G by saturating a set $\mathscr{S}$ of pairwise non-crossing minimal separators of G. If G' is triangulated then G' is a minimal triangulation of G.*

Proof. Let $\mathscr{S}'$ be a maximal set of pairwise non-crossing minimal separators of G containing $\mathscr{S}$ and let H be the graph obtained from G by saturating the separators of $\mathscr{S}'$. By Property 3.6 f), H if a minimal triangulation of G then, as $G \subseteq G' \subseteq H$ and G' is triangulated, $G' = H$. Therefore G' is a minimal triangulation of G. ∎

4 LB-Triangulation: Basic algorithmic process

We now use Characterization 3.4 to compute a minimal triangulation by forcing each vertex into being LB-simplicial by a local addition of edges. We will prove that the triangulation obtained is minimal by showing that the process chooses and saturates a set of pairwise non-crossing minimal separators of the input graph.

4.1 The algorithm

```
Algorithm LB-Triang
input   : A graph $G = (V, E)$.
output  : A minimal triangulation of $G$.

begin
    foreach $x \in V$ do
        Make $x$ LB-simplicial;
end
```

At the end of an execution, $\alpha = (x_1, x_2, ..., x_n)$ is the order in which the vertices have been processed, and G^{LB}_{α} will denote the triangulated graph obtained. Note that the algorithm processes the vertices in an arbitrary order. Thus any ordering can be chosen by the user, and this ordering can be supplied in an on-line fashion if desired.

Definition 4.1 *The* deficiency *of a vertex x in a graph G, denoted $Def_G(x)$, is the set of edges that has to be added to G to make x simplicial. We define* LB-deficiency *of a vertex x in G, denoted $LBDef_G(x)$, to be the set of edges that has to be added to G to make x LB-simplicial.*

Clearly, for any graph G, $LBDef_G(x) \subseteq Def_G(x)$ for every vertex x in G. For the remaining discussion on Algorithm LB-Triang, we will use the following notations. G_i denotes the graph at the beginning of step i, x_i is the vertex processed during step i, F_i denotes the set of fill edges added at step i to make x_i LB-simplicial in G_i, and finally, $\mathscr{S}_i$ denotes the set of minimal separators included in $N_{G_i}(x_i)$. Thus $F_i = LBDef_{G_i}(x_i)$ and G_{i+1} is the graph obtained from G_i by adding the set of edges F_i, or equivalently, by saturating the separators of $\mathscr{S}_i$. Making a vertex x_i LB-simplicial by Definition 3.3 requires computing the set $\mathscr{S}_i$ of minimal separators included in $N_{G_i}(x_i)$. For this, we use the following from [6].

Property 4.2 (Berry, Bordat, and Heggernes [6]) *For a vertex x in a graph G, the set of minimal separators of G included in $N(x)$ is exactly $\{N(C) \mid C \in \mathscr{C}(N[x])\}$.*

Consequently, computing the edge set F_i whose addition to G_i will make x_i LB-simplicial in the resulting G_{i+1} requires the following three steps:
- Computing $N_{G_i}[x_i]$
- Computing each connected component C in $\mathscr{C}_{G_i}(N_{G_i}[x_i])$
- Computing the neighborhood $N_{G_i}(C)$ for each C.

One of the interesting properties of Algorithm LB-Triang is that when x_i is LB-simplicial in G_{i+1}, it will remain LB-simplicial throughout the rest of the process, and thus be LB-simplicial in G_{α}^{LB}. This will become clear when we prove Invariant 4.7.

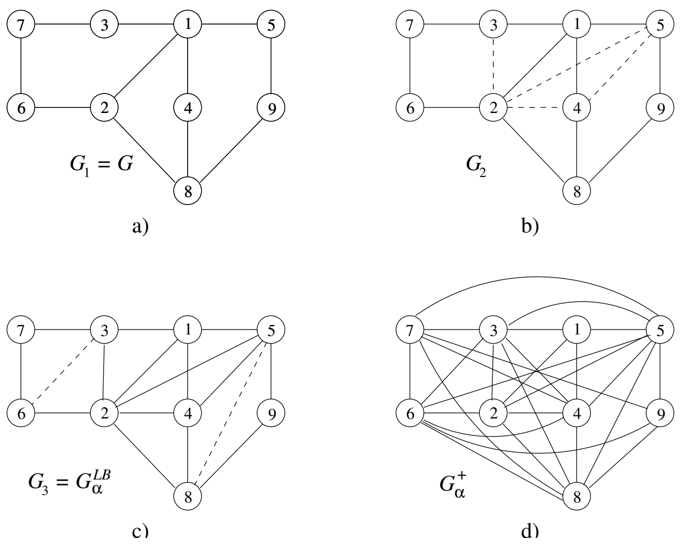

Figure 1: An example of how Algorithm LB-Triang proceeds.

Example 4.3 In Figure 1 a), a graph G is given with an ordering α on its vertices. Let us simulate how LB-Triang proceeds in an execution which processes the vertices in the given order.
Step 1: $N_{G_1}[1] = \{1,2,3,4,5\}$, and $\mathscr{C}_{G_1}(N_{G_1}[1]) = \{\{6,7\},\{8,9\}\}$. $N_{G_1}(\{6,7\}) = \{2,3\}$, and $N_{G_1}(\{8,9\}) = \{2,4,5\}$. Thus $F_1 = \{(2,3),(2,4),(2,5),(4,5)\}$. The resulting G_2 is given in Figure 1 b).
Step 2: $N_{G_2}[2] = \{1,2,3,4,5,6,8\}$, and $\mathscr{C}_{G_2}(N_{G_2}[2]) = \{\{7\},\{9\}\}$. $N_{G_2}(\{7\}) = \{3,6\}$, and $N_{G_2}(\{9\}) = \{5,8\}$. Thus $F_2 = \{(3,6),(5,8)\}$, and G_3 is shown in Figure 1 c).

No more fill edges are added at later steps since $G_3 = G_{\alpha}^{LB}$ is chordal. Figure 1 d) gives G_{α}^{+}.

4.2 Proof of correctness

We will first show that we indeed obtain a triangulation. The following lemmas are necessary in order to state and prove an invariant for the algorithm.

Lemma 4.4 *Let G be a graph, and let x be a vertex of G. The minimal separators included in $N(x)$ are pairwise non-crossing in G.*

Proof. Let S and S' be two minimal separators included in the neighborhood of x in G. Let C be the component of $\mathscr{C}(S)$ containing x. Since $S' \subseteq N(x) \subseteq C \cup N(C) \subseteq C \cup S$, S' does not cross S in G. ▪

Lemma 4.5 *Let G be a graph, let G' be the graph obtained from G by saturating a set of pairwise non-crossing minimal separators of G, and let x be an LB-simplicial vertex of G. Then $N_G(x) = N_{G'}(x)$*

Proof. By Lemma 3.9, it is sufficient to show that $\forall C \in \mathscr{C}_G(N_G[x])$, $N_G(C) = N_{G'}(C)$. Let C be a connected component of $\mathscr{C}_G(N_G[x])$. Let us show that $N_G(C) = N_{G'}(C)$. Vertex x is LB-simplicial in G, so by Property 4.2, $N_G(C)$ is a clique minimal separator of G, and then by Property 3.6 c), $N_G(C)$ is a minimal separator of G'. By Lemma 3.13 and the fact that C is a connected component of $\mathscr{C}_G(N_G(C))$, $N_G(C) = N_{G'}(C)$. ▪

Lemma 4.6 *Let G be a graph, let G' be the graph obtained from G by saturating a set of pairwise non-crossing minimal separators of G, and let x be an LB-simplicial vertex of G. Then x is LB-simplicial in G'.*

Proof. Let us show that x is LB-simplicial in G', i.e. that any minimal separator of G' included in $N_{G'}(x)$ is a clique in G'. Let S be a minimal separator of G' included in $N_{G'}(x)$. By Property 3.6 d), S is a minimal separator of G and by Lemma 4.5, S is included in $N_G(x)$. As x is LB-simplicial in G, S is a clique in G, but also in G', as $G \subseteq G'$. ▪

We are now able to prove the following invariant, which is the basis for the proof of correctness of the algorithm.

Invariant 4.7 *During an execution of Algorithm LB-Triang, any vertex that is LB-simplicial at a particular step remains LB-simplicial at all later steps.*

Proof. For any i from 1 to n, by Lemma 4.4 G_{i+1} is obtained from G_i by saturating a set of pairwise non-crossing minimal separators of G_i; by Lemma 4.6, any LB-simplicial vertex of G_i remains LB-simplicial in G_{i+1}. ▪

Lemma 4.8 *The graph G_α^{LB} resulting from Algorithm LB-Triang is a triangulation of G.*

Proof. By Invariant 4.7, at the end of an execution, every vertex of G_α^{LB} is LB-simplicial. By Characterization 3.4, G_α^{LB} is triangulated. ▪

We will now prove that the triangulation obtained is minimal.

Invariant 4.9 *For any i from 1 to $n+1$, the set $\cup_{1 \leq j < i} \mathscr{S}_j$ of minimal separators already saturated at the beginning of step i is a set of pairwise non-crossing minimal separators of G.*

Proof. By induction on i. The property is trivially true at the beginning of step 1. Assume that it is true at the beginning of step i, and let us show that it is then true at the beginning of step $i+1$. $\cup_{1 \leq j < i} \mathscr{S}_j$ is a set of pairwise non-crossing minimal separators of G, so by Property 3.6 b), it is a set of clique minimal separators of G_i. By Property 3.6 a), no separator of $\cup_{1 \leq j < i} \mathscr{S}_j$ crosses in G_i any minimal separator of G_i. Moreover, by Lemma 4.4, $\mathscr{S}_i$ is a set of pairwise non-crossing minimal separators of G_i, so $\cup_{1 \leq j < i+1} \mathscr{S}_j$ is a set of pairwise non-crossing minimal separators of G_i, and therefore a set of pairwise non-crossing minimal separators of G by Property 3.6 e). ∎

With these results, we are ready to state and prove the correctness of Algorithm LB-Triang:

Theorem 4.10 *Algorithm LB-Triang computes a minimal triangulation of the input graph.*

Proof. By Lemma 4.8, the obtained graph is triangulated, and by Invariant 4.9, G_α^{LB} is obtained from G by saturating a set of pairwise non-crossing minimal separators of G. By Lemma 3.14, G_α^{LB} is a minimal triangulation of G. ∎

5 Some important properties of LB-Triang

In this section, we examine some central properties of G_α^{LB}. First we show that LB-Triang can be implemented as an elimination scheme. Then we give some important connections between G_α^{LB} and G_α^+, showing in particular the relation between the transitory graphs at each step in the constructions of G_α^{LB} and G_α^+. We prove that LB-Triang solves the Minimal Triangulation Sandwich Problem automatically, and we examine the case when α is a MEO. Finally, we also show that LB-Triang is a process that characterizes minimal triangulation.

5.1 LB-Triang as an elimination scheme

Lekkerkerker and Boland [21] used Characterization 3.4 as an elimination scheme, meaning that each vertex was removed from the graph as its LB-simpliciality was established. We show in this section that Algorithm LB-Triang can likewise be implemented as an elimination scheme, removing each vertex after processing. The following lemmas will lead us to the desired result which is stated in Theorem 5.3.

Lemma 5.1 *Let $G = (V, E)$ be a graph and $a, b, y \in V$. Edge ab belongs to $LBDef(y)$ iff there is a chordless cycle $a, y, b, x_1, ..., x_k, a$ with $k \geq 1$ in G.*

Proof. We know that $ab \in LBDef(y)$ iff $ab \in N(y)$, $a \neq b$, $ab \notin E$ and there is a path in G from a to b, the intermediate vertices of which belong to $V \setminus N[y]$. Let $a, x_1, ..., x_k, b$, with $k \geq 1$, be a shortest possible such path. Then $a, y, b, x_1, ..., x_k, a$ is the desired chordless cycle of length ≥ 4. ∎

Lemma 5.2 *Let $G = (V, E)$ be a graph, X a set of LB-simplicial vertices of G, and y an vertex belonging to $V \setminus X$. Then $LBDef_G(y) = LBDef_{G(V \setminus X)}(y)$.*

Proof. The inclusion $LBDef_{G(V \setminus X)}(y) \subseteq LBDef_G(y)$ follows immediately from Lemma 5.1. Let us show that $LBDef_G(y) \subseteq LBDef_{G(V \setminus X)}(y)$. Let $ab \in LBDef_G(y)$. We will show that $ab \in LBDef_{G(V \setminus X)}(y)$. By Lemma 5.1, there is in G a chordless cycle $\mu = a, y, b, x_1, ..., x_k, a$ of length ≥ 4. Let us first show that no vertex of μ is LB-simplicial in G. Let x be a vertex of μ and a', b' be its neighbors in μ. By Lemma 5.1, $a'b' \in LBDef_G(x)$, so x is not LB-simplicial in G. Therefore μ is in $G(V \setminus X)$, and by Lemma 5.1, $ab \in LBDef_{G(V \setminus X)}(y)$. ∎

Theorem 5.3 *LB-Triang computes the same fill regardless of whether or not each LB-simplicial vertex is deleted at the end of each step of the algorithm.*

Proof. We show by induction on the number of already processed vertices that eliminating every vertex after processing it, does not affect the computed fill. Remember that G_i is the graph at the beginning of step i and F_i the fill computed at step i in the version of the algorithm without elimination. Let G'_i be the graph at the beginning of step i and F'_i the fill computed at step i in the version of the algorithm with elimination. In particular, $G_1 = G'_1 = G$. Let us show by induction on i ($1 \leq i \leq n$) that $F_i = F'_i$.

Induction hypothesis: $F_k = F'_k$, for $1 \leq k \leq i - 1$.

Clearly, $F_1 = F'_1$, since no vertices are removed before the end of the first step. We now assume that the induction hypothesis is true, and we will show that this implies that $F_i = F'_i$ for step i. Let us compare graphs G_i and G'_i at the beginning of step i before we process vertex x_i. Since $F_k = F'_k$, for $1 \leq k \leq i-1$, $G'_i = G_i(V \setminus \{x_1, x_2, ..., x_{i-1}\})$. By Invariant 4.7, vertices $x_1, x_2, ..., x_{i-1}$ are LB-simplicial in G_i. By Lemma 5.2, $LBDef_{G_i}(x_i) = LBDef_{G'_i}(x_i)$. We can thus conclude that $F_i = LBDef_{G_i}(x_i) = LBDef_{G'_i}(x_i) = F'_i$. ∎

We have in fact proved a stronger statement, namely that any LB-simplicial vertex can be eliminated in a preprocessing step without affecting the resulting fill generated by the restriction of the ordering on the remaining graph; such a preprocessing step would cost $O(nm)$.

LB-Triang may thus be run as an elimination process. Chances are that the removal of the LB-simplicial vertices during the course of the algorithm will rapidly disconnect the graph, thus allowing the process to run on small subgraphs. The fact that the graph searches must be run on the transitory graph instead of the input graph as we will see in Section 6 is not necessarily a drawback, as the transitory graph, although it grows by edges, shrinks by vertices because of the removal of the LB-simplicial vertices.

Corollary 5.4 (of Theorem 5.3) *LB-Triang elimination scheme computes a minimal triangulation of the input graph.*

We will finish this subsection by remarking that instead of making the vertices LB-simplicial one by one, it is possible to process and eliminate an independent set of vertices at each step. We use the following Lemma, which is a stronger version of Lemma 4.4:

Lemma 5.5 *Let G be a graph, let X be an independent set of vertices of V. The minimal separators included in the sets $N(x)$, for $x \in X$ are pairwise non-crossing in G.*

Proof. Let $x, x' \in X$ and S, S' be two minimal separators included in the neighborhood of x and x' respectively in G. Let C be the component of $\mathscr{C}(S)$ containing x' ($x' \notin S$ because $S \subseteq N(x)$ and $x' \notin N(x)$). $S' \subseteq N(x') \subseteq C \cup N(C) \subseteq C \cup S$. Then S' does not cross S in G. ■

It is easy to prove (using Lemmas 3.11 and 3.9) that making the vertices of an independent set X LB-simplicial in a graph G yields the same result whether the corresponding connected components are computed globally in G or by processing the vertices of X one by one.

Note that a recent result of Kratsch and Spinrad (see [17]) shows that it is possible to compute the connected components defined by all the vertex neighborhoods of a graph in a global $O(n^{2.83})$ time. A parallel implementation which repeatedly processes an independent set of vertices might prove interesting.

5.2 LB-Triang solves the Minimal Triangulation Sandwich Problem

As mentioned in the introduction, it is of interest for some applications when an ordering α is given as input, to find a minimal triangulation which is a subgraph of G^+_α. We now show that Algorithm LB-Triang computes such a triangulation.

Theorem 5.6 *Given a graph G and any ordering α on the vertices of G, G^{LB}_α solves the Minimal Triangulation Sandwich Problem with $G \subseteq G^{LB}_\alpha \subseteq G^+_\alpha$.*

Proof. The inclusion $G \subseteq G_\alpha^{LB}$ is evident. Let us show that $G_\alpha^{LB} \subseteq G_\alpha^+$. Let $G'_i = (V_i, E'_i)$, where $V_i = V \setminus \{x_1, x_2, ...x_{i-1}\}$, be the graph at the beginning of step i and F'_i the fill computed at step i of the LB-Triang elimination scheme and let $G^i = (V_i, E^i)$ be the graph at the beginning of step i of the elimination game. In particular, $G'_1 = G^1 = G$ and $G'_{n+1} = G^{n+1} =$ the empty graph. Let us show by induction on i $(1 \leq i \leq n)$ that $E'_i \subseteq E^i$ and $F'_i \subseteq E^{i+1}$.

As $G'_1 = G^1$, we have $E'_1 \subseteq E^1$ and $F'_1 = LBDef_{G'_1}(x_1) \subseteq Def_{G^1}(x_1) \subseteq E^2$. We now assume that $E'_{i-1} \subseteq E^{i-1}$ and $F'_{i-1} \subseteq E^i$. For any set X, $Pairs(X)$ denotes the set of all pairs of elements of X. Let us show that $E'_i \subseteq E^i$. $E'_i = (E'_{i-1} \cup F'_{i-1}) \cap Pairs(V_i) \subseteq (E^{i-1} \cup E^i) \cap Pairs(V_i) = E^i$. Let us show that $F'_i \subseteq E^{i+1}$. $F'_i = LBDef_{G'_i}(x_i) \subseteq Pairs(N_{G'_i}(x_i)) \subseteq Pairs(N_{G^i}(x_i)) \subseteq E^{i+1}$. For any i from 1 to n, any edge of F'_i is an edge of G^{i+1} and therefore an edge of G_α^+. We can conclude that $G_\alpha^{LB} \subseteq G_\alpha^+$. ∎

Corollary 5.7 *Given (G, α), α is a* MEO *of G iff $G_\alpha^{LB} = G_\alpha^+$.*

We will now give a connection to the elimination game. Ohtsuki, Cheung, and Fujisawa [22] give the following characterization of a MEO of a graph G:

Characterization 5.8 (Ohtsuki, Cheung, and Fujisawa [22]) *An ordering α of the vertices of a graph G is a* MEO *of G iff at each step i of the elimination game, for each pair $\{a, b\}$ of non-adjacent vertices of $N_{G^i}(x_i)$, there is a path in G^i from a to b with all intermediate vertices in $V \setminus N_{G^i}[x_i]$, where x_i and G^i denote the processed vertex and the transitory graph at step i.*

We denote this property of vertex x_i in G^i as follows:

Definition 5.9 *We will call a vertex x of G an* OCF-vertex *if for each pair $\{a, b\}$ of non-adjacent vertices of $N(x)$, there is a path in G from a to b with all intermediate vertices in $V \setminus N[x]$.*

The abbreviation OCF stands for Ohtsuki, Cheung, and Fujisawa. We connect Characterization 5.8 to Algorithm LB-Triang in the following fashion:

Lemma 5.10 *A vertex x in G is an OCF-vertex iff $LBDef(x) = Def(x)$.*

Proof. For any pair $\{a, b\}$ of non-adjacent vertices of $N(x)$, there is a path in G from a to b with all intermediate vertices in $V \setminus N[x]$ iff there is a component C of $\mathscr{C}(N[x])$ such that $N(C)$ contains a and b. Then a vertex x in G is an OCF-vertex iff $Def(x) \subseteq LBDef(x)$, i.e. iff $LBDef(x) = Def(x)$, as the inclusion of $LBDef(x)$ in $Def(x)$ is always true. ∎

Thus the implication from right to left of Characterization 5.8 follows from Corollary 5.7: if an OCF-vertex is chosen at each step, then by Lemma 5.10, the fill added at each step of the elimination game is identical to the fill added at each step of the LB-Triang elimination scheme. Hence, $G_\alpha^+ = G_\alpha^{LB}$, and by Corollary 5.7, α is a MEO of G.

5.3 LB-Triang characterizes minimal triangulation

We now end this section by showing that LB-Triang characterizes minimal triangulation, which is to say that not only does the algorithm compute a minimal triangulation, but conversely any minimal triangulation of the input graph can be obtained by some execution of LB-Triang. This is not the case with other classical minimal triangulation algorithms such as LEX M.

Property 5.11 (Ohtsuki, Cheung, and Fujisawa [22]) *H is a minimal triangulation of G iff $H = G^+_\alpha$ where α is a* MEO *of G.*

Theorem 5.12 *Given a graph G and any minimal triangulation H of G, there exists an ordering α of the vertices of G, such that $G^{LB}_\alpha = H$.*

Proof. By Property 5.11, there exists a MEO α of G such that $G^+_\alpha = H$. By Corollary 5.7, $G^{LB}_\alpha = G^+_\alpha = H$. ■

The set of orderings of the vertices of an arbitrary graph G can thus be partitioned into equivalence classes, each class defining the same minimal triangulation of G by LB-Triang. The set of equivalence classes represents the set of minimal triangulations of G.

We will now characterize the orderings for which LB-Triang will yield a *given* minimal triangulation H of G.

Characterization 5.13 *Let $H = (V, E + F)$ be a minimal triangulation of $G = (V, E)$, and let α be an ordering of the vertices of G. The following are equivalent:*

(a) $H = G^{LB}_\alpha$

(b) *At each step i of LB-Triang, $LBDef_{G_i}(x_i) \subseteq F$.*

(c) *At each step i of LB-Triang, any minimal separator of G_i included in $N_{G_i}(x_i)$ is a minimal separator of H.*

Proof. (a) $\Leftrightarrow$ (b) : If $H = G^{LB}_\alpha$, then at each step i of the LB-Triang process, $LBDef_{G_i}(x_i) \subseteq F$, as $LBDef_{G_i}(x_i)$ is the set F_i of fill edges added at step i. Conversely, if at each step i of the LB-Triang process, $LBDef_{G_i}(x_i) \subseteq F$ then $G^{LB}_\alpha \subseteq H$. As G^{LB}_α is a triangulation of G by Lemma 4.8 and H is a minimal triangulation of G, $H = G^{LB}_\alpha$. (a) $\Leftrightarrow$ (c) : If $H = G^{LB}_\alpha$ then at each step i of the LB-Triang process, any minimal separator of G_i included in $N_{G_i}(x_i)$ is an element of the set $\mathscr{S}_i$ of separators saturated at step i, and therefore is a minimal separator of H by Invariant 4.9 and Property 3.6 b). Conversely, we suppose that at each step i of the LB-Triang process, any minimal separator of G_i included in $N_{G_i}(x_i)$ is a minimal separator of H. Thus any fill edge has both endpoints in some minimal separator of H. As H is triangulated, any minimal separator of H is a clique by Characterization 3.1, so at each step i, $LBDef_{G_i}(x_i) \subseteq F$, and by the previous equivalence, $H = G^{LB}_\alpha$. ■

6 Complexity of a straightforward implementation

In this section, we propose an implementation with an $O(nm')$ time bound, where m' is the number of edges of G_α^{LB}.

Algorithm LB-TRIANG

```
input  : A graph $G = (V, E)$, with $|V| = n$ and $|E| = m$.
output : A minimal fill $F$ of $G$, with $|E + F| = m'$
         the order $\alpha$ in which the vertices are processed,
         a minimal triangulation $G_\alpha^{LB}$ of $G$, $G_\alpha^{LB} = (V, E + F)$.

begin
    $F \leftarrow \emptyset$;
    $G_1 \leftarrow G$;
    for $i = 1 \ldots n$ do
        Pick any unprocessed vertex $x$, and number it as $x_i$;
        Compute edges $F_i$ whose addition makes $x_i$ LB-simplicial in $G_i$;
        $F \leftarrow F + F_i$;
        $G_{i+1} \leftarrow (V, E + F)$;
    $\alpha \leftarrow [x_1, x_2, ..., x_n]$;
    $G_\alpha^{LB} \leftarrow G_{n+1}$;
    return $(F, \alpha, G_\alpha^{LB})$;
end
```

With this implementation, the only difficulty consists in computing the set of edges F_i. As the same component may be encountered many times, thus defining the same minimal separator many times, we aim to saturate each minimal separator of the minimal triangulation under construction exactly once. We claim that this will cost $O(nm')$.

Lemma 6.1 *Let $G = (V, E)$ be a graph, and let $S \subseteq V$. Then $\Sigma_{C \in \mathscr{C}(S)} |N(C)| \leq m$.*

Proof. For each C in $\mathscr{C}(S)$, let $InOut(C)$ denote the set of edges xy of G such that $x \in C$ and $y \in N(C)$. For each C in $\mathscr{C}(S)$, $|InOut(C)| \geq |N(C)|$, and for any distinct C and C' in $\mathscr{C}(S)$, $InOut(C) \cap InOut(C') = \emptyset$. Then $\Sigma_{C \in \mathscr{C}(S)} |N(C)| \leq \Sigma_{C \in \mathscr{C}(S)} |InOut(C)| = | \cup_{C \in \mathscr{C}(S)} InOut(C)| \leq |E| = m$. ∎

Lemma 6.2 *Let G be a graph, let x be a vertex of G and let G' be the graph obtained from G by saturating a set of pairwise non-crossing minimal separators of G. Then $\mathscr{C}_{G'}(N_{G'}[x]) = \mathscr{C}_G(N_{G'}[x])$ and for each C in $\mathscr{C}_{G'}(N_{G'}[x])$, $N_{G'}(C) = N_G(C)$.*

Proof. It is sufficient to show that for each C in $\mathscr{C}_{G'}(N_{G'}[x])$, C is in $\mathscr{C}_G(N_{G'}[x])$ and $N_{G'}(C) = N_G(C)$. Let C be a connected component of $\mathscr{C}_{G'}(N_{G'}[x])$. We first show that $N_{G'}(C) = N_G(C)$. By Property 4.2, $N_{G'}(C)$ is a minimal separator of G', then by Lemma 3.13 and the fact that C is a connected component of $\mathscr{C}_{G'}(N_{G'}(C))$, C is in $\mathscr{C}_G(N_{G'}(C))$ and $N_{G'}(C) = N_G(C)$. We will now show that C is in $\mathscr{C}_G(N_{G'}[x])$. $C \neq \emptyset$ and $C \subseteq V \setminus N_{G'}[x]$ (because $C \in \mathscr{C}_{G'}(N_{G'}[x])$), $G(C)$ is connected (because $C \in \mathscr{C}_G(N_{G'}(C))$) and $N_G(C) \subseteq N_{G'}[x]$ (because $N_G(C) = N_{G'}(C)$ and $N_{G'}(C) \subseteq N_{G'}[x]$ as C is a component of $\mathscr{C}_{G'}(N_{G'}[x])$). By Observation 3.7, C is in $\mathscr{C}_G(N_{G'}[x])$. ∎

Lemma 6.3 *At each step i of the LB-Triang process, the neighborhoods of the connected components of $\mathscr{C}(N_{G_i}[x_i])$ may be computed in G instead of G_i.*

Proof. This follows immediately from Invariant 4.9 and Lemma 6.2. ∎

Lemma 6.4 *The number of minimal separators of a triangulated graph is smaller than n.*

Proof. This is a direct consequence of Theorem 3 from [26]. ∎

Theorem 6.5 *The time complexity of LB-Triang is in $O(nm')$.*

Proof. At each step i of Algorithm LB-Triang, the elements of the set $\mathscr{S}_i$ (i.e. the minimal separators included in $N_{G_i}(x_i)$) have to be saturated. In order to avoid saturating the same separator several times, we store the separators in a data structure as we saturate them. Thus after a minimal separator is computed, it is searched for in the data structure and if it is not found, it is inserted and saturated. Consequently, we have to evaluate the complexity of the following three actions at each step i: 1) computing $\mathscr{S}_i$, 2) searching/inserting the minimal separators of $\mathscr{S}_i$ in the data structure, 3) saturating the new minimal separators.

1) By Property 4.2, $\mathscr{S}_i = \{N_{G_i}(C) \mid C \in \mathscr{C}_{G_i}(N_{G_i}[x_i])\}$, and by Lemma 6.3, $\mathscr{S}_i = \{N_G(C) \mid C \in \mathscr{C}_G(N_{G_i}[x_i])\}$. $N_{G_i}[x_i]$ may be computed in $O(n)$ and the sets $N_G(C)$, $C \in \mathscr{C}_G(N_{G_i}[x_i])$ in $O(m)$. Thus computing all the sets $\mathscr{S}_i$ requires $O(nm)$.

2) We choose a data structure allowing to search/insert a separator S in $O(|S|)$ time. We represent the set of already inserted minimal separators by an n-ary rooted tree, each successor of a node being numbered from 1 to n. Initially, the tree is reduced to its root. We suppose that $V = \{1, 2, ..., n\}$. If for instance we want to insert the separator $\{2, 3, 7\}$ into the initial tree, we create the successor number 2 of the root (representing the set $\{2\}$), then the successor number 3 of this node (representing the set $\{2, 3\}$) and then the successor number 7 of this node (representing the set $\{2, 3, 7\}$). Thus, if the separator $\{2\}$, $\{2, 3\}$ or $\{2, 3, 7\}$

is computed afterwards, it will be found in the tree and will not be saturated again. To avoid initializing the vector of pointers to the successors in each node of the tree, we use the technique of back pointers suggested by A. V. Aho et al. [1] and explained in more detail by A. Cournier [10]. Searching/inserting a separator S requires $O(|S|)$ time, so by Lemma 6.1 we obtain a complexity of $O(m)$ at each step. Note that the elements of each minimal separator have to be inserted in increasing order. The following algorithm puts the elements of $N_G(C)$ in increasing order into the variable $Neighbor(C)$ for each C in $\mathscr{C}_G(N_{G_i}[x_i])$ in $O(m)$ time.

```
begin
    foreach $C$ in $\mathscr{C}_G(N_{G_i}[x_i])$ do
        $Neighbor(C) \leftarrow \emptyset$;
    foreach $y$ in $N_{G_i}(x_i)$ in increasing order do
        foreach $z$ in $N_G(y) \setminus N_{G_i}[x_i]$ do
            let $C \in \mathscr{C}_G(N_{G_i}[x_i])$ containing $z$;
            if $y \neq last(Neighbor(C))$ do
                add $y$ to $Neighbor(C)$;
end
```

The search/insert operation thus globally requires $O(nm)$ time.

3) By Lemma 4.8, G_α^{LB} is triangulated and by Invariant 4.9 and Property 3.6 b), $\mathscr{S}_i$ is a set of minimal separators of G_α^{LB} then by Lemma 6.4, the total number of new minimal separators saturated at all steps is smaller than n. Saturating a separator S requires *O(number of edges of* G_α^{LB}*(S))*, which is $O(m')$, so saturating all the minimal separators requires $O(nm')$.
We obtain a global time complexity of $O(nm')$ for this straightforward implementation of Algorithm LB-Triang. ■

Note that the implementation presented in this section is extremely simple. The only operation among those described above which requires more than $O(nm)$ time is the actual saturation of the minimal separators. In the next section, we will describe an implementation that uses a new data structure based on a tree decomposition, which enables representing the minimal triangulation obtained without actually adding the saturating edges, and thus ensuring an $O(nm)$-time complexity. However, numerical tests reported in Section 8 show that, even with the already presented straightforward implementation, LB-Triang tends to run faster than LEX M.

7 Improving the complexity to $O(nm)$

The purpose of this section is to provide an implementation of LB-Triang which improves the complexity from $O(nm')$ to $O(nm)$. A first version of this implementation was presented by Heggernes and Villanger in [15].

As mentioned before, the only operation in the straightforward implementation of LB-Triang which requires more than $O(nm)$ time is the actual saturation of the minimal separators. To achieve an $O(nm)$ time implementation, we do not actually add the edges necessary to saturate the minimal separators, but store each minimal separator as a vertex list, with the understanding that it is a clique. In this fashion, we save time in computing the cliques; however it becomes more costly to compute the neighborhood of x_i in the transitory graph G_i at each step i. Recall that fill edges of G_i appear only within already computed minimal separators, thus in order to compute $N_{G_i}[x_i]$, we have to search for the already computed minimal separators which include x_i. The union of such minimal separators, together with the original neighborhood of x_i in G, gives us $N_{G_i}[x_i]$. We will explain and prove how this can be done within the time limit of $O(nm)$.

In this implementation, we maintain a tree structure TS which we will prove to be a *tree decomposition* of G. In the beginning, all vertices of G belong to the same node of the tree TS. This corresponds to the situation where we do not know anything about the minimal separators of G, so that parts of the graph are not separated from each other. At each step of the algorithm, when new minimal separators in the neighborhood of x_i are computed, they are inserted as edges of TS. Whenever a minimal separator S separating x_i from a component $C \in \mathscr{C}(N_{G_i}[x_i])$ is computed, the node X of TS which contains S, x_i, and at least one vertex of C is split into two nodes X_1 and X_2. The vertices of S are inserted as an edge X_1X_2 in TS, and X_1 and X_2 contain the parts of X that are subsets of $C \cup S$ and $V \setminus C$ respectively. This way, nodes of TS are split, and edges added, whenever we compute new minimal separators.

Due to the properties of tree decompositions, and using subtrees and edges of TS, we are able to compute the union of the minimal separators containing x_i at step i in $O(m)$ time, giving a total time of $O(nm)$ for the whole algorithm. In the rest of this section, we give the details and formal proofs of this approach.

7.1 Tree decomposition

Definition 7.1 *Let $G = (V, E)$ be a graph. A tree structure on G is a structure $TS(T, (X_u)_{u \in U_T}, (S_{uv})_{uv \in E_T})$, where $T = (U_T, E_T)$ is a tree, X_u is a subset of V for each u in U_T and S_{uv} is a subset of V for each uv in E_T.*

The vertices of G will be noted x, y, z, etc. and the nodes of T will be noted u, v, w, etc. In this section, TS will implicitly denote a tree structure ($T = (U_T, E_T)$,

$(X_u)_{u\in U_T}$, $(S_{uv})_{uv\in E_T})$ on a graph $G = (V, E)$. Given a tree structure TS on G, we define the sets U_x, U_C and the graphs T_x, T_C and T_{uv} as follows.

- $\forall x \in V$, $U_x = \{u \in U_T \mid x \in X_u\}$ and $T_x = T(U_x) = (U_x, E_x)$,
- $\forall C \subseteq V$, $T_C = (\cup_{x\in C} U_x, \cup_{x\in C} E_x) = (U_C, E_C)$,
- $\forall uv \in E_T$, T_{uv} and T_{vu} are the two connected components of $T' = (U_T, E_T \setminus \{uv\})$ respectively containing u and v.

Definition 7.2 *A tree decomposition of G is a tree structure TS on G such that:*

a) $\cup_{u\in U_T} X_u = V$*,*

b) $\forall xy \in E,\ \exists u \in U_T \mid x, y \in X_u$ *(i.e.* $U_x \cap U_y \neq \emptyset$*),*

c) $\forall x \in V,\ T_x$ *is a subtree of* T*,*

d) $\forall uv \in E_T,\ S_{uv} = X_u \cap X_v$*.*

Tree decomposition is used to define the treewidth of a graph. For more information on tree decompositions and their importance, the reader is referred to [8]. We give some basic properties of a tree decomposition which will be used in this section.

Property 7.3 *Let TS be a tree decomposition of G. Then $\forall x \in V$, $\forall uv \in E_T$, $x \in S_{uv}$ iff uv is an edge of T_x.*

Proof. Vertex $x \in S_{uv}$ iff $x \in X_u \cap X_v$, i.e. $u, v \in U_x$ or uv is an edge of T_x (because uv is an edge of T). ∎

Property 7.4 *Let TS be a tree decomposition of G, and C be a subset of V. If $G(C)$ is connected then T_C is a subtree of T.*

Proof. Let $u, v \in U_C$. Let us show that there is a path in T_C from u to v. Let $x, y \in C$ such that $u \in U_x$ and $v \in U_y$, and let $\lambda = (x = x_0, x_1, ..., x_k = y)$ be a path in $G(C)$ from x to y. For i from 0 to k, T_{x_i} is a subtree of T and if $i < k$ then $x_i x_{i+1} \in E$, which implies that $U_{x_i} \cap U_{x_{i+1}} \neq \emptyset$. Then there is a path in T_C from u to v. ∎

Property 7.5 *Let TS be a tree decomposition of G. Then $\forall uv \in E_T$, $\forall C \in \mathscr{C}_G(S_{uv})$, $T_C \subseteq T_{uv}$ or $T_C \subseteq T_{vu}$.*

Proof. By Property 7.4, T_C is a subtree of T and by Property 7.3 and the fact that $C \cap S_{uv} = \emptyset$, uv is not an edge of T_C, so $T_C \subseteq T_{uv}$ or $T_C \subseteq T_{vu}$. ■

Thus, if in G S_{uv} separates two components C_1 and C_2 of $\mathscr{C}_G(S_{uv})$, then S_{uv} may separate C_1 and C_2 also in T, in the sense that one of the subtrees T_{C_1} and T_{C_2} is included in T_{uv} and the other is included in T_{vu}. We will call *tree decomposition of G by minimal separators* any tree decomposition of G such that for any edge uv of T, S_{uv} is a minimal separator of G separating in T two full components of $\mathscr{C}_G(S_{uv})$.

Definition 7.6 *A tree decomposition of G by minimal separators is a tree decomposition TS of G satisfying the extra property:*

e) $\forall uv \in E_T,\ \exists C_1, C_2$ *full components of* $\mathscr{C}_G(S_{uv}) \mid T_{C_1} \subseteq T_{uv}$ *and* $T_{C_2} \subseteq T_{vu}$.

Our $O(nm)$ time complexity follows from the fact that the tree structure constructed in LB-Treedecomp process is a tree decomposition of G by minimal separators at every step of this process.

We will denote by *search in T* any graph search in the tree T (for instance breadth-first or depth-first search).

7.2 An $O(nm)$ time implementation

Algorithm LB-Treedecomp
input : A graph $G = (V, E)$, with $|V| = n$ and $|E| = m$.
output : The order α in which the vertices are processed, and the graph G_α^{LB}.

```
begin
    $H \leftarrow (V, \emptyset)$;
    $T \leftarrow (\{u_0\}, \emptyset)$;
    $X_{u_0} \leftarrow V$;
    InitVariables();
    for $i = 1 \ldots n$ do
        Pick any unprocessed vertex $x$, and number it as $x_i$;
        $N_H[x_i] \leftarrow$ Neighbors($G, x_i, TS$);
        foreach $C \in \mathscr{C}_G(N_H[x_i])$ do
            $S \leftarrow N_G(C)$;
            Search/Insert $S$ in the S/I data structure;
            if $S$ has not been found in the S/I data structure do
                Let $c$ be a vertex of $C$;
                Search in $T$ from $u(c)$ until a node $w$ such that $x_i \in X_w$ is reached;
                Split $w$ into $w_1$ and $w_2$;
                $X_{w_1} \leftarrow X_w \cap (C \cup S)$;
                $X_{w_2} \leftarrow X_w \setminus C$;
                Replace each edge $wv$ by $w_1v$ with $S_{w_1v} = S_{wv}$ if $S_{wv}$
                $\subseteq C \cup S$ and by $w_2v$ with $S_{w_2v} = S_{wv}$ otherwise;
                Add edge $w_1w_2$;
                $S_{w_1w_2} \leftarrow S$;
                **UpdateVariables**();
    $\alpha \leftarrow [x_1, x_2, ..., x_n]$;
    **return**$(\alpha, H)$;
**end**
```

As in the straightforward implementation of LB-Triang, we use a Search/-Insert data structure to avoid processing already saturated minimal separators (see the proof of Theorem 6.5) that we denote by S/I data structure. In order to compute at each step i the neighborhood of x_i in the transitory graph G_i, we use a tree structure TS on the input graph G (which we will prove to be a tree decomposition of G by minimal separators). This computation is performed by function **Neighbors** whose specifications are the following (the implementation of this function will be given later).

Function Neighbors (G, x, TS)
input : A graph $G = (V, E)$, a vertex x of G, a tree structure $TS = (T = (U_T, E_T), (X_u)_{u \in U_T}, (S_{uv})_{uv \in E_T})$ on G.
precondition: TS is a tree decomposition of G.
output : the set $N_{G'}[x]$, where G' is the graph obtained from G by saturating the elements of the sets S_{uv} for each uv in E_T, i.e. the set $N_G[x] \cup (\cup_{uv \in E_T | x \in S_{uv}} S_{uv})$.

Procedures **InitVariables** and **UpdateVariables** respectively initialize and update some variables which are only used in function **Neighbors**, except for the variables $u(x)$ which are also used in the following algorithm: for any vertex x of G, $u(x)$ contains an arbitrary node of U_x. The implementation of these procedures will be given later.

Example 7.7 In Figure 1 a), a graph G is given with an ordering α on its vertices. Let us simulate how LB-Treedecomp proceeds in an execution which processes the vertices in the given order. The successive states of tree T are shown in Figure 2. Figure 2 a) shows the initial state of T.

Step 1: **Neighbors**$(G, 1, TS) = N_G[1] = \{1, 2, 3, 4, 5\}$, and $\mathscr{C}_G(\{1, 2, 3, 4, 5\}) = \{\{6, 7\}, \{8, 9\}\}$. $N_G(\{6, 7\}) = \{2, 3\}$, and $N_G(\{8, 9\}) = \{2, 4, 5\}$. In the process

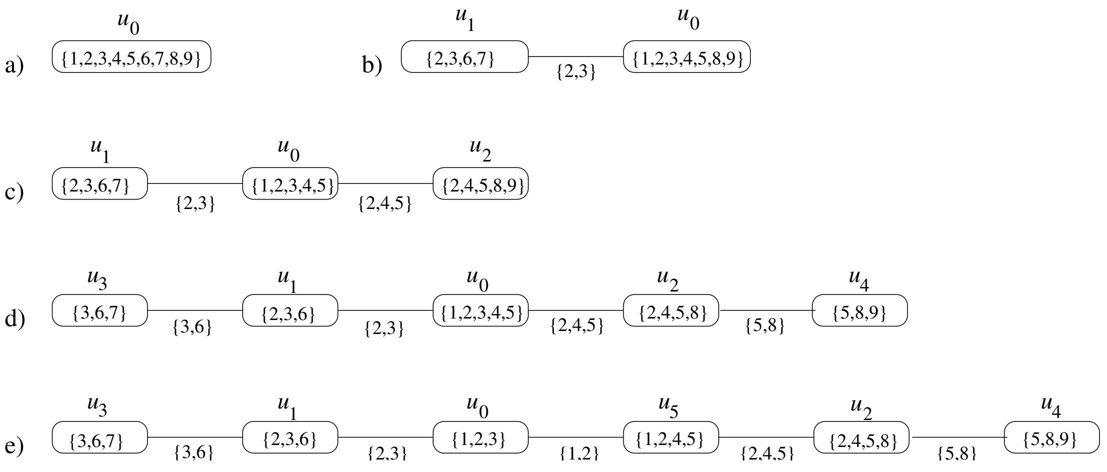

Figure 2: The successive states of tree T in the execution of Algorithm LB-Treedecomp on graph G of Figure 1 a)

of $\{6,7\}$, u_0 is split into u_1 and u_0 (Figure 2 b), and in the process of $\{8,9\}$, u_0 is split into u_2 and u_0 (Figure 2 c).

Step 2: $\textbf{Neighbors}(G,2,TS) = N_G[2] \cup \{2,3\} \cup \{2,4,5\} = \{1,2,3,4,5,6,8\}$, and
$\mathscr{C}_G(\{1,2,3,4,5,6,8\}) = \{\{7\},\{9\}\}$. $N_G(\{7\}) = \{3,6\}$, and $N_G(\{9\}) = \{5,8\}$. In the process of $\{7\}$, u_1 is split into u_3 and u_1, and in the process of $\{9\}$, u_2 is split into u_4 and u_2 (Figure 2 d).

Step 3: $\textbf{Neighbors}(G,3,TS) = N_G[3] \cup \{2,3\} \cup \{3,6\} = \{1,2,3,6,7\}$, and $\mathscr{C}_G(\{1,2,3,6,7\}) = \{\{4,5,8,9\}\}$. $N_G(\{4,5,8,9\}) = \{1,2\}$. In the process of $\{4,5,8,9\}$, u_0 is split into u_5 and u_0 (Figure 2 e).

No further split operation is performed in the tree T at later steps. We obtain the graph G_α^{LB} shown in Figure 1 c). Note that the sets X_u for node u of the final tree T (Figure 2 e) are the maximal cliques of G_α^{LB}, and T a clique tree of the chordal graph G_α^{LB}. This is not always the case because, according to Algorithm LB-Treedecomp, a given minimal separator may only appear in one edge of T, whereas it may appear in several edges of a clique tree of a chordal graph.

7.3 Proof of correctness and complexity

7.3.1 Algorithm LB-Treedecomp

The implementation of LB-Treedecomp we present here is similar to the straightforward one presented in Section 6. Instead of being saturated, the minimal separators that have not been found in the S/I data structure are inserted as edges into the tree T of the tree structure TS and their saturation is simulated in function **Neighbors**. Thus the correctness of LB-Treedecomp depends on that of function **Neighbors**.

Let us recall that for each i from 1 to $n+1$, G_i denotes the transitory graph at the beginning of step i of the LB-Triang process, and $\mathscr{S}_i$ denotes the set of minimal separators saturated at step i, so that $G_1 = G$ and G_{i+1} is obtained from G_i by saturating the elements of $\mathscr{S}_i$. In the same way, let G'_i denote the graph obtained from G by saturating the sets S processed so far at the beginning of step i of the LB-Treedecomp process, and let $\mathscr{S}'_i$ denote the set of the sets S processed at step i, so that $G'_1 = G$ and G'_{i+1} is obtained from G'_i by saturating the elements of $\mathscr{S}'_i$. Note that G'_i is also the graph obtained from G by saturating the sets S_{uv} for each uv in E_T at the beginning of step i, as the only sets S that are processed but not inserted as edges into T have been found in the S/I data structure and therefore are included in already processed sets.

Invariant 7.8 *For any i from 0 to n, if function **Neighbors** is correct and if TS is a tree decomposition of G at the beginning of each step $\leq i$ of LB-Treedecomp process, then the following property P_j holds for any j between 0 and i.*

P_j: (if $j > 0$ then $N_H[x_j] = N_{G_j}[x_j]$ and $\mathscr{S}'_j = \mathscr{S}_j$) and $G_{j+1} = G'_{j+1}$.

Proof. By induction on j. P_0 holds, as $G_1 = G'_1 = G$. Assume that P_{j-1} holds for some j, $1 \leq j \leq i$. Let us show that P_j holds. TS is a tree decomposition of G at the beginning of step j, so the precondition of function **Neighbors** is satisfied so that, with the assumption that this function is correct, it will return the set $N_{G'_j}[x_j]$ at step j. Therefore $N_H[x_j] = N_{G'_j}[x_j]$ and, by induction hypothesis, $G_j = G'_j$,so $N_H[x_j] = N_{G_j}[x_j]$. $\mathscr{S}'_j = \{N_G(C) \mid C \in \mathscr{C}_G(N_H[x_j])\} = \{N_G(C) \mid C \in \mathscr{C}_G(N_{G_j}[x_j])\}$,so by Lemma 6.3, $\mathscr{S}'_j = \mathscr{S}_j$. Hence the graph obtained from G_j by saturating the elements of $\mathscr{S}_j$ is exactly the graph obtained from G'_j by saturating the elements of $\mathscr{S}'_j$, i.e. $G_{j+1} = G'_{j+1}$ ■

The correctness of Algorithm LB-Treedecomp follows from the the fact that Property P_i holds for any i from 1 to n (see Theorem 7.22 below). However, it remains to give the implementation of function **Neighbors** and prove its correctness and the satisfaction of its precondition at each step of the LB-Treedecomp process.

7.3.2 Function Neighbors

Remember that, given a graph G, a vertex x of G and a tree decomposition TS of G, function **Neighbors** returns the set $N_G[x] \cup (\cup_{uv \in E_T | x \in S_{uv}} S_{uv})$, i.e. by Property 7.3 the set $N_G[x] \cup \{y \in V \mid T_x \textit{ and } T_y \textit{ have at least one common edge}\}$. Let us give the following definitions:

Definition 7.9 *Let TS be a tree decomposition of G and x be a vertex of G. We define the following sets:*

• *OneEdge* $= \{y \in V \mid T_y$ *has at least one edge*$\}$
• *Inner*$(x) = \{y \in$ *OneEdge* $\mid T_y$ *is included in* $T_x\}$
• *InnerOuter*$(x) = \{y \in$ *OneEdge* $\mid T_y$ *has at least one edge in* T_x *and at least one edge out of* $T_x\}$
• *BorderOuter*$(x) = \{y \in$ *OneEdge* $\mid T_x$ *and* T_y *have exactly one node in common* $\}$
• *Outer*$(x) = \{y \in$ *OneEdge* $\mid T_y$ *is disjoint from* $T_x\}$
• *CommonEdge*$(x) = \{y \in$ *OneEdge* $\mid T_x$ *and* T_y *have at least one edge in common* $\}$
• *ThroughBorder*$(x) = \{y \in$ *OneEdge* $\mid$ *some edge of* T_y *has exactly one of its extremities in* $T_x\}$

Definition 7.10 *Let* $T' = (U_{T'}, E_{T'})$ *be a subtree of a tree* $T = (U_T, E_T)$.
$Border_T(T') = \{(u,v) \in U_{T'} \times (U_T \setminus U_{T'}) \mid uv \in E_T\}$.

Lemma 7.11 *Let* TS *be a tree decomposition of* G *and* x *be a vertex of* G.
a) $OneEdge = Inner(x) + InnerOuter(x) + BorderOuter(x) + Outer(x)$,
b) $CommonEdge(x) = Inner(x) + InnerOuter(x)$,
c) $ThroughBorder(x) = InnerOuter(x) + BorderOuter(x)$,
d) $OneEdge = \cup_{uv \in E_T} S_{uv}$,
e) $CommonEdge(x) = \cup_{uv \in E_T | x \in S_{uv}} S_{uv}$,
f) $ThroughBorder(x) = \cup_{(u,v) \in Border_T(T_x)} S_{uv}$.

Proof.
a), b) and c) are evident properties on the relative position of a subtree T_y having at least one edge with respect to a subtree T_x in any tree T.
c), d) and e) follow from Property 7.3. ■

Our goal is to compute the set $\cup_{uv \in E_T | x \in S_{uv}} S_{uv}$, i.e. by Lemma 7.11 b) and e), the union of the sets $Inner(x)$ and $InnerOuter(x)$. Set *OneEdge* will be computed in a global variable of LB-Treedecomp. $Border_T(T_x)$ can be computed by a search in T from an arbitrary node of T_x, which allows us to compute set $ThroughBorder(x)$. It remains to distinguish the vertices of $InnerOuter(x)$ from those of $BorderOuter(x)$ in set $ThroughBorder(x)$ and to distinguish the vertices of $Inner(x)$ from those of $Outer(x)$ in set $OneEdge \backslash ThroughBorder(x)$. For the first point, we introduce the notion of *degree* in T of a node u of T with respect to a vertex y of X_u.

Definition 7.12 *Let* TS *be a tree decomposition of* G.
$\forall u \in U_T,\ \forall y \in X_u,\ Degree_T(u,y) = |\{v \in N_T(u) \mid y \in S_{uv}\}|$.

Lemma 7.13 *Let* TS *be a tree decomposition of* G *and* x *be a vertex of* G.
a) $\forall y \in ThroughBorder(x),\ \forall (u,v) \in Border_T(T_x) \mid y \in S_{uv}$,

$y \in InnerOuter(x)$ iff $|\{v' \in N_T(u) \mid y \in S_{uv'}$ and $(u,v') \in Border_T(T_x)\}| < Degree_T(u,y)$
b) $\forall y \in OneEdge \setminus ThroughBorder(x)$, if $u(y) \in U_y$ then
$y \in Inner(x)$ iff $x \in X_{u(y)}$

Proof.
a) Let us assume that $y \in InnerOuter(x)$. u is a node both of T_x and of T_y and $y \notin BorderOuter(x)$ then there is another common node, say u', of T_x and T_y. Let $(u, v', ..., u')$ be the unique path in T from u to u'. The edge uv' is an edge of T_x and T_y. Then $y \in S_{uv'}$ (by Property 7.3) and $(u,v') \notin Border_T(T_x)$, therefore $|\{v' \in N_T(u) \mid y \in S_{uv'}$ and $(u,v') \in Border_T(T_x)\}| < Degree_T(u,y)$. Conversely, assume on the contrary that $y \notin InnerOuter(x)$. Then by Lemma 7.11 $y \in BorderOuter(x)$, so it is clear that $|\{v' \in N_T(u) \mid y \in S_{uv'}$ and $(u,v') \in Border_T(T_x)\}| = Degree_T(u,y)$.
c) By Lemma 7.11 $OneEdge \setminus ThroughBorder(x) = Inner(x) \oplus Outer(x)$. If $y \in Inner(x)$ then $x \in X_u$ for any node u of U_y and if $y \in Outer(x)$ then $x \notin X_u$ for any node u of U_y. Therefore it is sufficient to test whether belonging x belongs to X_u for an arbitrary node u of U_y to decide whether y belongs to $Inner(x)$ or not. ∎

We will now implement function **Neighbors**. For this purpose, we maintain in Algorithm LB-Treedecomp variables $OneEd$, $u(y)$ and $Deg(u,y)$ which respectively contain the current values of $OneEdge$, an arbitrary node of U_y and $Degree_T(u,y)$, with the following initializations and updates.

Procedure InitVariables()

```
begin
    $OneEd \leftarrow \emptyset$;
    foreach $y \in V$ do
        $u(y) \leftarrow u_0$;
        $Deg(u_0, y) \leftarrow 0$;
end
```

Procedure UpdateVariables()

```
begin
    $OneEd \leftarrow OneEd \cup S$;
    for $j = 1 \ldots 2$ do
        foreach $y \in X_{w_j}$ do
            $u(y) \leftarrow w_j$;
            $Deg(w_j, y) \leftarrow 0$;
        foreach $v \in N_T(w_j)$ do
            foreach $y \in S_{w_j v}$ do
                Increment $Deg(w_j, y)$;
end
```

In function **Neighbors**, we use the local variables $InnerOuter$, $Inner$ and $Count(u, y)$ which respectively contain the current values of $InnerOuter(x)$, $Inner(x)$ and $Degree_T(u, y) - |\{v \in N_T(u) \mid (y \in S_{uv} \text{ and } (u, v) \in Border_T(T_x))\}|$.

Function Neighbors (G, x, TS)
input : A graph $G = (V, E)$, a vertex x of G, a tree structure $TS = (T = (U_T, E_T), (X_u)_{u \in U_T}, (S_{uv})_{uv \in E_T})$ on G.
precondition: TS is a tree decomposition of G.
output : the set $N_{G'}[x]$, where G' is the graph obtained from G by saturating the elements of the sets S_{uv} for each uv in E_T, i.e. the set $N_G[x] \cup (\cup_{uv \in E_T | x \in S_{uv}} S_{uv})$.

```
begin
    Compute $Border_T(T_x)$ by search in $T$ from $u(x)$;
    $InnerOuter \leftarrow \emptyset$;
    $Inner \leftarrow OneEd$;
    foreach $(u, v) \in Border_T(T_x)$ do
        foreach $y \in S_{uv}$ do
            Add $y$ to $InnerOuter$;
            Remove $y$ from $Inner$;
            $Count(u, y) \leftarrow Deg(u, y)$;
    foreach $(u, v) \in Border_T(T_x)$ do
        foreach $y \in S_{uv}$ do
            Decrement $Count(u, y)$;
            if $Count(u, y) = 0$ do
                Remove $y$ from $InnerOuter$;
    foreach $y \in Inner$ do
        if $x \notin X_{u(y)}$ do
            Remove $y$ from $Inner$;
    return $(N_G[x] \cup Inner \cup InnerOuter)$;
end
```

Theorem 7.14 *Function **Neighbors** is correct (provided that TS is a tree decomposition of G).*

Proof. Let us assume that TS is a tree decomposition of G. It is clear from procedures **InitVariables** and **UpdateVariables** that variables $OneEd$, $u(y)$ and

$Deg(u,y)$ respectively contain the current values of $\cup_{uv \in E_T} S_{uv}$ (and therefore of $OneEdge$ by Lemma 7.11 d)), an arbitrary node of U_y and $Degree_T(u,y)$. By Lemmas 7.11 and 7.13, the local variables $InnerOuter$, $Inner$ and $Count(u,y)$ respectively contain the current values of $InnerOuter(x)$, $Inner(x)$ and $Degree_T(u,y) - |\{v \in N_T(u) \mid y \in S_{uv}$ and $(u,v) \in Border_T(T_x)\}|$. By Lemma 7.11 b) and e), the function returns $N_G[x] \cup (\cup_{uv \in E_T | x \in S_{uv}} S_{uv})$. ■

7.3.3 Complexity

The following lemma is the key of $O(nm)$ time complexity of LB-Treedecomp.

Lemma 7.15 *Let TS be a tree decomposition of G by minimal separators and T' be a subtree of T. Then $\Sigma_{(u,v) \in Border_T(T')} |S_{uv}| \leq m$.*

Proof. For each $(u,v) \in Border_T(T')$, let $C_{(u,v)}$ be a full component of $\mathscr{C}_G(S_{uv})$ such that $T_{C_{(u,v)}} \subseteq T_{vu}$, and let $InOut(C_{(u,v)})$ denote the set of edges xy of G such that $x \in C_{(u,v)}$ and $y \in N_G(C_{(u,v)}) = S_{uv}$. For each $(u,v) \in Border_T(T')$, $|InOut(C_{(u,v)})| \geq |N_G(C_{(u,v)})| = |S_{uv}|$. Let $(u,v), (u',v')$ be distinct elements of $Border_T(T')$. Let us show that $InOut(C_{(u,v)}) \cap InOut(C_{(u',v')}) = \emptyset$. It is sufficient to show that no vertex of $C_{(u,v)}$ nor of S_{uv} can be in $C_{(u',v')}$. If $x \in C_{(u,v)}$, then $T_x \subseteq T_{vu}$, and if $x \in S_{uv}$, then by Property 7.3 uv is an edge of T_x. In neither case is T_x included in $T_{v'u'}$, then x is not in $C_{(u',v')}$. Therefore, $InOut(C_{(u,v)}) \cap InOut(C_{(u',v')}) = \emptyset$. Hence $\Sigma_{(u,v) \in Border_T(T')} |S_{uv}| \leq \Sigma_{(u,v) \in Border_T(T')} |InOut(C_{(u,v)})| = | \cup_{(u,v) \in Border_T(T')} InOut(C_{(u,v)})| \leq |E| = m$. ■

Theorem 7.16 *If TS is a tree decomposition of G by minimal separators at the beginning of each process of a set S, then the time complexity of LB-Treedecomp is $O(nm)$.*

Proof. All sets (in particular sets X_u and S_{uv}) are implemented with the data structure mentioned in the proof of Theorem 6.5, which was suggested by A. V. Aho et al. [1] and explained in more detail by A. Cournier [10]. This data structure allows us to initialize a set, add or remove an element, test for the presence of an element, etc. in in $O(1)$ time and to read the elements of a set S in $O(|S|)$. By the hypothesis on TS, Theorem 7.14 and Invariant 7.8, the sets S processed at each step are the same as in Algorithm LB-Triang. Therefore, as in the proof of the complexity of LB-Triang (Theorem 6.5), computing the components of $\mathscr{C}_G(N_H[x_i])$ and their neighborhoods and searching/inserting the minimal separators into the S/I data structure require $O(nm)$, and the number of new (i.e. not found in the S/I data structure) separators to be processed is smaller than n, which implies that the tree T has at most n nodes. Initializations only require $O(n)$. It remains to show that computing $N_H[x_i]$ and processing a new separator S may be done in $O(m)$.

Computing $N_H[x_i]$: T has at most n nodes, so computing $Border_T(T_x)$ by search in T costs $O(n)$. Processing the elements of $Border_T(T_x)$ requires $O(\Sigma_{(u,v)\in Border_T(T_x)}|S_{uv}|)$, which by Lemma 7.15 is in $O(m)$. Computing $N_H[x_i]$ therefore requires $O(m)$ time.

Processing a new separator S: Since T has at most n nodes, searching T to reach w costs $O(n)$. Splitting w into w_1 and w_2 costs $O(n)$. Replacing edges wv with w_1v or w_2v and updating $Deg(u,y)$ require $O(\Sigma_{(w,v)\in Border_T(T')}|S_{wv}|)$, where T' is the subtree of T reduced to node w, w_1 or w_2, and therefore cost $O(m)$ by Lemmas 7.15. Adding edge w_1w_2, updating $OneEd$ and $u(y)$ cost $O(n)$. Processing a new separator S thus requires $O(m)$. ∎

7.3.4 Proof of the Invariant on TS

To complete the proof of correctness and complexity of Algorithm LB-Treedecomp, it remains to show that TS is a tree decomposition of G by minimal separators at the beginning of each processing step of a set S. We first prove two lemmas about tree decompositions (Lemmas 7.17 and 7.18) which we apply to Algorithm LB-Treedecomp (Lemmas 7.19 and 7.20). These lemmas aim at proving Lemma 7.20 which will be used in the proof of Invariant 7.21.

Lemma 7.17 *Let TS be a tree decomposition of G, let G' be the graph obtained from G by saturating the elements of the sets S_{uv} for each uv in E_T, let $x \in V$ and $C \in \mathscr{C}_G(N_{G'}[x])$. Then $|U_C \cap U_x| \leq 1$.*

Proof. Assume by contradiction that $|U_C \cap U_x| > 1$. By Property 7.4, T_C and T_x are subtrees of T, so the unique path in T connecting two given different nodes of $U_C \cap U_x$ is also a path in T_C and T_x. T_C and T_x have at least one edge in common. Let uv be a common edge of T_C and T_x and let y be a vertex of C such that uv is an edge of T_y. By Property 7.3, $x, y \in S_{uv}$, so $y \in N_{G'}[x]$, whereas $y \in C$ and $C \in \mathscr{C}_G(N_{G'}[x])$, a contradiction. ∎

Lemma 7.18 *Under the hypothesis of Lemma 7.17, let $S = N_G(C)$ and λ be a path in T of minimal length from a node of T_C to a node of T_x. Then for any node u of λ, $S \subseteq X_u$.*

Proof. We have to show that for any vertex s of S, λ is a path in T_s. By Lemma 7.17, $|U_C \cap U_x| \leq 1$, so it is sufficient to show that for any vertex s of S, $U_C \cap U_s \neq \emptyset$ and $U_x \cap U_s \neq \emptyset$ (because in that case λ is a subpath of the unique path in T from some node of $U_C \cap U_s$ to some node of $U_x \cap U_s$, which is also a path in T_s). Let $y \in C \mid ys \in E$. $U_y \cap U_s \neq \emptyset$, so $U_C \cap U_s \neq \emptyset$. $xs \in E'$, so $xs \in E$ or $\exists uv \in E_T \mid x, s \in S_{uv}$. If $xs \in E$ then $U_x \cap U_s \neq \emptyset$ else, by Property 7.3, uv is a common edge of T_x and T_s, which implies that $U_x \cap U_s \neq \emptyset$. ∎

Lemma 7.19 *Let S be a set processed at some step i of Algorithm LB-Tree-decomp, with $S = N_G(C)$, $C \in \mathscr{C}_G(N_H[x_i])$. Let us assume that TS is a tree decomposition of G at the beginning of the process of S' for each set S' processed before S or equal to S. At the beginning of the processing of S, if S is not found in the S/I data structure then there is a node u of T such that $U_C \cap U_{x_i} = \{u\}$ and $S \subseteq X_u$.*

Proof. We will show that this property is true at the beginning of step i and is preserved until the beginning of the processing of S. At the beginning of step i, let λ be a path in T of minimal length from a node of T_C to a node of T_{x_i}. $C \in \mathscr{C}_G(N_H[x_i]) = \mathscr{C}_G(N_{G'_i}[x_i])$ then by Lemmas 7.17 and 7.18, $|U_C \cap U_{x_i}| \leq 1$ and for any node u of λ, $S \subseteq X_u$. To prove that the property is true at the beginning of step i, it remains to show that $U_C \cap U_{x_i} \neq \emptyset$. Let us assume by contradiction that $U_C \cap U_{x_i} = \emptyset$. In this case, λ has at least one edge uv, with $S \subseteq X_u \cap X_v = S_{uv}$, so some set S_{uv} containing S has been processed at some previous step j. Because of the hypothesis on TS, Theorem 7.14 and Invariant 7.8, $\mathscr{S}'_j = \mathscr{S}_j$ for any $j \leq i$. Therefore $S \in \mathscr{S}_i$, so by Invariant 4.9 and Lemma 3.12, S is a minimal separator of G_j. Hence, as $S \subseteq S_{uv} \subseteq N_{G_j}(x_j)$, S is a minimal separator of G_j included in $N_{G_j}(x_j)$, i.e. $S \in \mathscr{S}_j$, so $S \in \mathscr{S}'_j$. As S is processed at step j, it will be found in the S/I data structure at step i, a contradiction. Therefore, at the beginning of step i, there is a node u of T such that $U_C \cap U_{x_i} = \{u\}$ and $S \subseteq X_u$. Let us show that this property is preserved when processing a set S' at step i before processing S, with $S' = N_G(C')$, $C' \in \mathscr{C}_G(N_H[x_i])$. If S' is found in the S/I data structure then TS is unchanged and the property is preserved. Otherwise, let w' be the node of T which is split when S' is processed. If $w' \notin U_C$ then T_C is unchanged and the property is preserved. Otherwise $w' \in U_C \cap U_{x_i} = \{u\}$, so u is split into nodes u_1 and u_2. As neither x_i nor any vertex of C belongs to $C' \cup S'$, the new trees T_C and T_{x_i} are obtained from the previous ones by replacing node u by u_2 with the same neighbors. Furthermore, no vertex of S belongs to C', so that $S \subseteq X_{u_2}$. Hence $U_C \cap U_{x_i} = \{u_2\}$ and $S \subseteq X_{u_2}$. Therefore, the property is preserved until the beginning of the processing of S. ▪

Lemma 7.20 *Under the hypothesis of Lemma 7.19, let w be the node of T which is split when processing S. At the beginning of the processing of S, $S \subseteq X_w$ and $X_w \cap C \neq \emptyset$.*

Proof. By Lemma 7.19, at the beginning of the processing of S, there is a node u of T such that $U_C \cap U_{x_i} = \{u\}$ and $S \subseteq X_u$. w is the first node of U_{x_i} reached during a search in T from node $u(c)$ of U_C, so $w = u$. Hence $S \subseteq X_w$ and as $w \in U_C$, $X_w \cap C \neq \emptyset$. ▪

Invariant 7.21 *TS is a tree decomposition of G by minimal separators at the beginning of the processing of each set S in any execution of Algorithm LB-Treedecomp.*

Proof. This property is trivially true at the initialization. Let us show that it is preserved during the processing of each set S. Let S be a set processed at some step i of the execution of LB-Treedecomp, $TS = (T = (U_T, E_T)$, $(X_u)_{u \in U_T}$, $(S_{uv})_{uv \in E_T})$ before processing S and $TS' = (T' = (U_{T'}, E_{T'})$, $(X'_u)_{u \in U_{T'}}$, $(S'_{uv})_{uv \in E_{T'}})$ after processing S. We suppose that the property holds until the beginning of the processing of S (and so by Theorem 7.14 and Invariant 7.8, $\mathscr{S}'_j = \mathscr{S}_j$ for any $j \leq i$). Let us show that it still holds after processing S. If S has been found in the S/I data structure then the property is trivially preserved. Otherwise, w is split in T into the nodes w_1 and w_2.

a) $X_w = X'_{w_1} \cup X'_{w_2}$, so a) is preserved.

b) Let $xy \in E$. Let us show that $\exists u \in U_{T'} \mid x, y \in X'_u$. By b) on TS, $\exists u \in U_T \mid x, y \in X_u$. If $u \neq w$, then $u \in U_{T'}$ and $x, y \in X'_u$. Otherwise, if at least one of x and y belongs to C, then $x, y \in C \cup N_G(C)$ (because $xy \in E$) and then $x, y \in X'_{w_1}$, else $x, y \in X'_{w_2}$, with $w_1, w_2 \in U_{T'}$.

c) Let $x \in V$. Let us show that T'_x is a subtree of T'. If $x \notin X_w$ then $T'_x = T_x$. If $x \in S$ then T'_x is obtained from T_x by splitting w into w_1 and w_2 and reconnecting the neighbors of w in T_x either to w_1 or to w_2 in T'_x. For $j = 1, 2$, if $x \in X'_{w_j} \setminus S$ then T'_x is obtained from T_x by replacing w by w_j with the same neighbors of w in T_x as of w_j in T'_x. In every case T'_x is still a subtree of T'.

d) Let $uv \in E_{T'}$. Let us show that $S'_{uv} = X'_u \cap X'_v$. If $uv = w_1 w_2$ then $S'_{uv} = S$ and $X'_u \cap X'_v = X_w \cap (C \cup S) \cap (X_w \setminus C) = X_w \cap S = S$ (because $S \subseteq X_w$ by Lemma 7.20). In this case, $S'_{uv} = X'_u \cap X'_v$. Otherwise, we may assume that $v \notin \{w_1, w_2\}$. If $u \notin \{w_1, w_2\}$ then $uv \in E_T$ and $S'_{uv} = S_{uv} = X_u \cap X_v = X'_u \cap X'_v$. If $u = w_1$ then $S'_{uv} = S_{wv} \subseteq C \cup S$ and $S'_{uv} = S_{wv} = X_w \cap X_v = X_w \cap X'_v$, therefore $S'_{uv} = X_w \cap (C \cup S) \cap X'_v = X'_{w_1} \cap X'_v = X'_u \cap X'_v$. If $u = w_2$, then $S'_{uv} = S_{wv} \not\subseteq C \cup S$ and $S'_{uv} = S_{wv} = X_w \cap X_v = X_w \cap X'_v$. Let us show that $S'_{uv} \cap C = \emptyset$. $S, S'_{uv} \in \cup_{1 \leq j \leq i} \mathscr{S}'_j = \cup_{1 \leq j \leq i} \mathscr{S}_j$, so by Invariant 4.9 S'_{uv} does not cross S in G and, as $S'_{uv} \not\subseteq C \cup S$, $S'_{uv} \cap C = \emptyset$; therefore, $S'_{uv} = (X_w \setminus C) \cap X'_v = X'_{w_2} \cap X'_v = X'_u \cap X'_v$.

e) Let $uv \in E_{T'}$. Let us show that $\exists C_1, C_2$ full components of $\mathscr{C}_G(S'_{uv}) \mid T'_{C_1} \subseteq T'_{uv}$ and $T'_{C_2} \subseteq T'_{vu}$. If $uv \neq w_1 w_2$ then S'_{uv} has not changed and one of the subtrees T'_{uv} and T'_{vu} of T' has not changed, and therefore it still contains exactly one of T'_{C_1} and T'_{C_2}. By Property 7.5, the other of T'_{uv} and T'_{vu} contains the other of T'_{C_1} and T'_{C_2}. If $uv = w_1 w_2$, so $S'_{uv} = S$. $S \in \mathscr{S}'_i = \mathscr{S}_i$, then $x_i \notin S$. Let $C_1 = C$ and let C_2 be the component of $\mathscr{C}_G(S)$ containing x_i. C_1 and C_2 are full components of $\mathscr{C}_{G_i}(S)$, and hence also of $\mathscr{C}_G(S)$ by Invariant 4.9 and Lemma 3.12. By Lemma 7.20, $X_w \cap C \neq \emptyset$, so $X'_{w_1} \cap C \neq \emptyset$, i.e. w_1 is a node of T'_{C_1}. $x_i \in X_w \setminus C$, so $x_i \in X'_{w_2}$, so w_2 is a node of T'_{C_2}. By Property 7.5, $T'_{C_1} \subseteq T'_{w_1 w_2} = T'_{uv}$ and $T'_{C_2} \subseteq T'_{w_2 w_1} = T'_{vu}$. ■

7.3.5 Correctness and $O(nm)$ time complexity

Theorem 7.22 *Given a graph G, Algorithm LB-Treedecomp computes an ordering α on the vertices of G and the graph G_α^{LB} with a time complexity of $O(nm)$.*

Proof. Let H be the graph computed by the algorithm. For every i from 1 to n, by Invariant 7.21, Theorem 7.14 and Invariant 7.8, $N_H[x_i] = N_{G_i}[x_i]$ and by Theorem 5.3, $N_{G_i}[x_i] = N_{G_\alpha^{LB}}[x_i]$. Therefore $N_H(x) = N_{G_\alpha^{LB}}(x)$ for every vertex x of G, which means that $H = G_\alpha^{LB}$. The $O(nm)$ time complexity follows from Invariant 7.21 and Theorem 7.16. ■

8 Experimental results

In this section we report results from practical implementations of LB-Triang, and compare it to other minimal triangulation algorithms.

8.1 Comparing the run time of minimal triangulation algorithms

In the first test, we compare an $O(nm')$ time implementation of LB-Triang to LEX M from [28]. In this test we also include an $O(nm)$ time implementation of LB-Triang called LB-Treedec [15], a slightly different version of LB-Treedecomp explained in Section 7. For this test, we randomly generated 100 connected input graphs, all on 2000 vertices, and with increasing number of edges. LB-Triang and LB-Treedec processed the vertices of each graph in the same random order, and the last vertex in this order was the starting vertex of LEX M. The practical implementation of all three algorithms is done in C++, and run on an Intel Pentium 4 2.2GHz processor with 512MB RAM and 512MB level-2 cache. The results from this test is shown in Figure 3.

From this we can see that LB-Triang, even with the $O(nm')$ time implementation, exhibits a run time pattern that is significantly superior to LEX M. We would like to emphasize that the behavior that can be observed from the figure is typical for all the tests that we have run, thus the tests indicate that the practical run time of LB-Triang is mostly dependent on n. As can be seen from the figure, we have run the test on also very dense graphs. For practical applications, it is definitely most interesting to study the first half of this chart, with input graphs containing up to 50 percent of the maximum number of potential edges. Only on very sparse graphs is LEX M superior to LB-Triang, and it is never superior to LB-Treedec. As expected, the run times of the $O(nm)$ and $O(nm')$ time implementations meet for very dense graphs, since $m' = O(m)$ in these cases. We can thus conclude that Algorithm LB-Triang is inherently fast regardless of implementation.

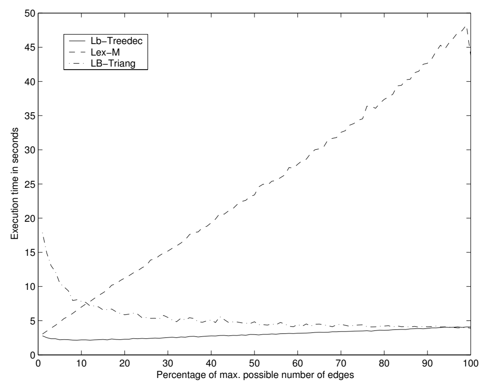

Figure 3: Comparing the running times of LB-Triang, LB-Treedec, and LEX M.

In the second test, we tested the $O(nm')$ time implementation of LB-Triang also against the previously mentioned Algorithm MinimalChordal (MC) from [7]. Since we did not have a C++ implementation of MC, we did a naive and straightforward implementation of MC, LB-Triang, and LEX M in Matlab. Since Matlab is slower, we generated smaller input graphs for this test. The 12 randomly generated graphs have 200 vertices and an increasing number of edges up to 50 percent of maximum potential number of edges. Since MC is practical only with orderings that generate small fill, we computed a minimum degree (MD) ordering of each graph first, and each graph was processed by MC and LB-Triang in this ordering. This second test was done on an UltraSPARC-IIi 300MHz processor, and the run time is measured in seconds. The results are shown in Figure 4.

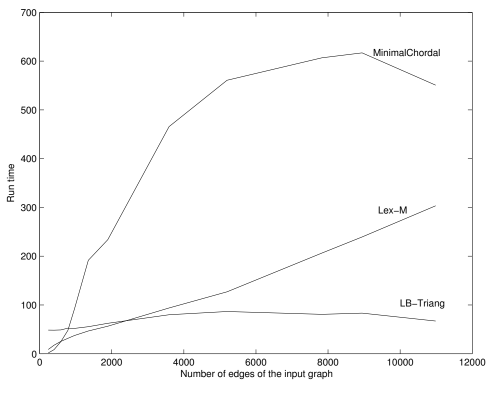

Figure 4: Comparing the running times of MinimalChordal, LB-Triang and LEX M.

Again, we observe the same kind of relationship between the runtimes of LEX M and LB-Triang, even though the Matlab codes are simple and quite different from the C++ codes of these algorithms. From this test, as expected in view of the worst case time analysis, we can see that Algorithm Minimal Chordal is practical only for very sparse input graphs. We should mention that we also tested these three algorithms on graphs originating from real problems. However, all such graphs that we have at hand are very sparse, and they demonstrate the same behavior as can be observed from the already presented charts.

One might also be interested in knowing the fill generated by each of the three algorithms. We can report that MC and LB-Triang have produced the same fill on all of the tested graphs. This fill was only slightly less than the fill produced by the MD algorithm. LEX M produced fills that were excessive, and was significantly inferior to the other algorithms for this purpose. Note that the given ordering has little effect on the fill that LEX M produces, whereas both MC and LB-Triang produce minimal small fills given a good ordering.

8.2 Dynamically computing an ordering that results in small fill

The third test that we present shows results from an implementation of LB-Triang that attempts to compute a minimal triangulation with small fill by dynamically choosing an appropriate vertex at each step, without having been given a particular ordering of the vertices initially. The MD algorithm chooses, at each step i of the elimination game, a vertex of smallest degree in G^i. Using the same approach, we have implemented a dynamic version of LB-Triang that chooses, at each step i, an unprocessed vertex x with smallest $|N_{G_i}(x) \setminus \{x_1, ..., x_{i-1}\}|$. In this test, we compare the quality of the produced triangulation with respect to the size of fill, to the triangulation produced by the MD algorithm, and also to the regular LB-Triang processing the vertices in a given MD ordering. The test results are shown in Table 1. We have again generated random graphs of various density. The first two columns show the number of vertices and edges for each graph G. In column 3, the fill generated by an MD ordering α is shown. The standard LB-Triang algorithm is then run on (G, α), and the size of fill in G_α^{LB} is given in column 4. Finally in column 5, the fill generated by Dynamic LB-Triang choosing a vertex of minimum transitory degree at each step as described above is shown.

We see that Dynamic LB-Triang produces less fill than standard LB-Triang processing the vertices in a given MD ordering on all of these examples. We have actually not been able to create an example where Dynamic LB-Triang computes a larger fill than standard LB-Triang or MD.

This test indicates that Dynamic LB-Triang produces slightly better triangulations than MD. It should be noted that MD is an $O(nm')$ time algorithm, whereas Dynamic LB-Triang can be implemented in $O(nm)$ time using the same approach as described in Section 7. We have not tested the practical run time of

n	m	MD	Standard	Dynamic
100	245	622	617	617
100	474	1460	1449	1449
100	1297	2404	2398	2391
200	587	3191	3182	3177
200	971	5695	5683	5681
200	1358	7436	7422	7422
300	452	1367	1358	1355
300	1325	11158	11147	11140
300	3863	24356	24351	24324

Table 1: Comparing the size of the fill generated by Minimum Degree, Standard LB-Triang and Dynamic LB-Triang.

Dynamic LB-Triang against MD, since MD has been subject to extensive code optimization through the last two decades, whereas we have merely a straight forward implementation of Dynamic LB-Triang.

9 Conclusion

We would like to conclude this paper by summarizing the properties of LB-Triang that were proven in the previous sections.

LB-Triang is a practical minimal triangulation algorithm which has the following properties: It can create any minimal triangulation of a given graph, thus it is a characterizing process. It is in fact the fist $O(nm)$ time process that can yield any triangulation of a given graph. The vertices can be processed in any order or in an on-line fashion. LB-Triang can be implemented as an elimination scheme; in particular, all LB-simplicial vertices can be eliminated simultaneously at the same step. LB-Triang solves the Minimal Triangulation Sandwich Problem directly from the input graph, without having to remove fill from the given triangulation. In addition, several heuristics, like Minimum Degree, can be integrated into LB-Triang in order to make it produce a minimal triangulation with low fill or with other desired properties with promising experimental results. LB-Triang has a very simple $O(nm')$ time implementation, and a more complicated $O(nm)$ time implementation, involving data structures which might prove useful for solving other problems as well. LB-Triang is fast in practice even with a straightforward $O(nm')$ time implementation.

References

[1] A. V. Aho, I. E. Hopcroft and J. D. Ullman. The design and analysis of computer algorithms. *Addison-Wesley,* p. 71, ex. 2.12,1974.

[2] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database systems. *J. Assoc. Comput. Mach.*, 30:479–513, 1983.

[3] A. Berry. Désarticulation d'un graphe. *PhD Dissertation, LIRMM, Montpellier, December 1998.*

[4] A. Berry. A wide-range efficient algorithm for minimal triangulation. In *Proceedings of SODA'99*, pages S860–S861, 1999.

[5] A. Berry, J. R. S. Blair, P. Heggernes, and B. Peyton. Maximum Cardinality Search for Computing Minimal Triangulations of Graphs. *Algorithmica*, 39-4:287 – 298, 2004.

[6] A. Berry, J.-P. Bordat, and P. Heggernes. Recognizing weakly triangulated graphs by edge separability. *Nordic Journal of Computing*, 7:164–177,2000.

[7] J. R. S. Blair, P. Heggernes, and J. A. Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 250:124–141,2001.

[8] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21,1993.

[9] F. R. K. Chung and D. Mumford. Chordal completions of planar graphs. *J. Comb. Theory,* 31:96–106,1994.

[10] A. Cournier. Quelques Algorithmes de Décomposition de Graphes. *PhD Dissertation, LIRMM, Montpellier, France, February 1993.*

[11] E. Dahlhaus. Minimal elimination ordering inside a given chordal graph. In *Graph Theoretical Concepts in Computer Science - Proceedings of WG '97*, pages 132–143. Springer Verlag, 1997. Lecture Notes in Computer Science 1335.

[12] G.A. Dirac. On rigid circuit graphs. *Anh. Math. Sem. Univ. Hamburg,* 25:71–76, 1961.

[13] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Math.*, 15:835–855, 1965.

[14] J. A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[15] P. Heggernes and Y. Villanger. Efficient Implementation of a Minimal Triangulation Algorithm. In *Algorithms - Proceedings of ESA 2002*, pages 550–561. Springer Verlag, 2002. Lecture Notes in Computer Science 2461.

[16] D. Hudson, S. Nettles, and T. Warnow. Obtaining highly accurate topology estimates of evolutionary trees from very short sequences. In *Proceedings of RECOMB'99*, pages 198–207. 1999.

[17] D. Kratsch and J. Spinrad. Between $O(nm)$ and $O(n^{\alpha})$. In *Proceedings of SODA 2003*, pages 709–716, 2003.

[18] T. Kloks, D. Kratsch, and J. Spinrad. Treewidth and pathwidth of cocomparability graphs of bounded dimension. *Res. Rep. 93-46, Eindhoven University of Technology*, 1993.

[19] T. Kloks, D. Kratsch, and J. Spinrad. On treewidth and minimum fill-in of asteroidal triple-free graphs. *Theoretical Computer Science*, 175:309–335, 1997.

[20] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their applications to expert systems. *J. Royal Statist. Soc., ser B*, 50:157–224, 1988.

[21] C. G. Lekkerkerker and J. Ch. Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1962.

[22] T. Ohtsuki, L. K. Cheung, and T. Fujisawa. Minimal triangulation of a graph and optimal pivoting order in a sparse matrix. *Journal of Math. Analysis and Applications*, 54:622–633, 1976.

[23] A. Parra and P. Scheffler. How to use the minimal separators of a graph for its chordal triangulation. *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming (ICALP '95), Lecture Notes in Computer Science*, 944:123–134, 1995.

[24] S. Parter. The use of linear graphs in Gauss elimination. *SIAM Review,* 3:119–130, 1961.

[25] B. Peyton. Minimal orderings revisited. *SIAM J. Matrix Anal. Appl.*, 23:271–294, 2001.

[26] D. J. Rose. Triangulated graphs and the elimination process. *J. Math. Anal. Appl.*, 32:597–609, 1970.

[27] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

[28] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.

[29] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, est acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13:566–579, 1984.

[30] I. Todinca. *Aspects algorithmiques des triangulations minimales des graphes.* PhD thesis, LIP, ENS Lyon, 1999.

[31] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.

Paper II

A Vertex Incremental Approach for Maintaining Chordality

Anne Berry[*] Pinar Heggernes[†] Yngve Villanger[†]

Abstract

For a chordal graph $G = (V, E)$, we study the problem of whether a new vertex $u \notin V$ and a given set of edges between u and vertices in V can be added to G so that the resulting graph remains chordal. We show how to resolve this efficiently, and at the same time, if the answer is no, specify a maximal subset of the proposed edges that can be added along with u, or conversely, a minimal set of extra edges that can be added in addition to the given set, so that the resulting graph is chordal. In order to do this, we give a new characterization of chordal graphs and, for each potential new edge uv, a characterization of the set of edges incident to u that also must be added to G along with uv. We propose a data structure that can compute and add each such set in $O(n)$ time. Based on these results, we present an algorithm that computes both a minimal triangulation and a maximal chordal subgraph of an arbitrary input graph in $O(nm)$ time, using a totally new vertex incremental approach. In contrast to previous algorithms, our process is on-line in that each new vertex is added without reconsidering any choice made at previous steps, and without requiring any knowledge of the vertices that might be added subsequently.

1 Introduction

Chordal graphs (also called triangulated graphs) are a well-studied class of graphs, with applications in many fields. Some applications require that chordality be maintained incrementally, that is, as edges and/or vertices are added or deleted from the graph, they desire to maintain chordality. Ibarra [28] gives a dynamic algorithm for adding or removing a given edge in $O(n)$ time in a chordal graph if this does not destroy chordality, where n is the number of vertices of the input

[*]LIMOS, UMR CNRS 6158, Universite Clermont-Ferrand II, F-63177 Aubiere, France. Email: `berry@isima.fr`

[†]Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Emails: `Pinar.Heggernes@ii.uib.no` and `yngvev@ii.uib.no`

graph. More recently, a *2-pair* [10] has been defined as a pair of non-adjacent vertices in a chordal graph, such that the graph remains chordal when the edge between these vertices is added to the graph.

A chordal graph can be obtained from any non chordal graph by: adding edges until the graph becomes chordal, a process called *triangulation*, or by removing edges until the graph becomes chordal, thus computing a *chordal subgraph*. Adding or removing a minimum number of edges has been shown to be NP-hard [30, 36]. However, adding or removing an inclusion minimal set of edges can be accomplished in polynomial time. Given an arbitrary chordal subgraph, e.g., an independent set on the vertices of the graph (resp. supergraph, e.g., a complete graph on the same vertex set) of the input graph, edges can be added (resp. removed) one by one after testing that the resulting graph remains chordal, until no further candidate edge can be found. This ensures that maximality (resp. minimality) is achieved, by the results of [32]. The problem of maintaining a chordal graph by edge addition or deletion and the problem of computing a maximal chordal subgraph or a minimal chordal supergraph are thus strongly related.

The problem of adding an inclusion minimal set of *fill* edges, called *minimal triangulation*, has many applications in various fields such as sparse matrix computation [31] and database management [3]. The problem has been well studied since 1976, and several $O(nm)$ time algorithms exist for solving it [4, 5, 6, 16, 32], where m is the number of edges in the input graph. None of these algorithms use an edge incremental approach as described above. However, the algorithm proposed by Blair, Heggernes, and Telle [11], which requires even less time when the fill is small, does use an edge deletion approach.

The reverse problem of computing a maximal chordal subgraph has also been studied, with applications to sparse matrix computation, computing a large clique or a large independent set, and improving phylogenetic data [2, 10, 15, 17, 20, 35]. There exist several algorithms that compute a maximal chordal subgraph in $O(\Delta m)$ time, where Δ is the maximum degree in the graph [2, 17, 35].

In this paper, we present a new process for adding a *vertex* with a given set of incident edges to a chordal graph while maintaining chordality, which we are able to implement more efficiently than if we were to add the corresponding edges one by one. Our process is based on two new characterizations. The first is a characterization of a chordal graph by its edges, which can be regarded as a specialization of the edge characterization for weakly chordal graphs introduced by Berry, Bordat, and Heggernes [8]. The second is a characterization of a unique set of edges $R(G, u, v)$ incident to a vertex u that must be added to a chordal graph G along with edge uv to ensure that chordality is preserved, given that we are only allowed to add edges incident to u. We show that we can compute this set $R(G, u, v)$ of edges in $O(n)$ time, by proposing a data structure that corresponds to a clique tree of the current chordal subgraph. A similar data structure was used by the authors to prove an $O(nm)$ time bound for one of their minimal triangulation algorithms [4, 6, 26]; however, here we present a new implementation of clique trees that allows a more efficient data structure for our purposes.

We use our results to compute both a minimal triangulation and a maximal chordal subgraph of a given arbitrary graph in $O(nm)$ time. This is done by an incremental process that repeatedly adds a new vertex u to the already constructed chordal graph H along with a maximal set of edges between u and H, or a minimal set of extra edges between u and H in addition to the originally specified edges.

Some of the existing algorithms that compute a maximal chordal subgraph or a minimal triangulation also use a vertex incremental process [2, 5, 6, 17, 32, 35], though none of them compute both chordal graphs at the same time. In addition, all these previous algorithms require knowing the whole graph in advance, as either vertices that are not yet processed are marked in some way to define the next vertex in the process, or edges are added between pairs of vertices that are not yet processed. Furthermore, these algorithms require the added vertex to be a simplicial vertex of the transitory chordal graph. One exception from this requirement is the algorithm of [6], but it does add edges between pairs of neighbors of the added vertex that are not yet processed.

Our approach here is completely different from the previous ones, as it is more general: at each vertex addition step, we do not require the added vertex to be or to become simplicial, thereby enabling processing of vertices in any order. Moreover, we add only edges incident to the new vertex, so that we never need to reconsider or change the chordal graph that has been computed thus far.

As a result, our process can add any vertex with any proposed neighborhood, and efficiently give a correction if the resulting graph fails to be chordal, either by computing a maximal subset of the edges to be added, or a minimal set of extra edges along with the proposed ones. In addition, the transitory chordal graph is maintained in a dynamic fashion, as making the desired or necessary additions to the graph does not require a recomputation.

This paper is organized as follows: in the next section we give the necessary graph theoretic background and terminology. Section 3 contains our new characterizations. The algorithms are presented and proved correct in Section 4, whereas the data structure details and time complexity analysis are given in Section 5. We conclude in Section 6.

2 Graph theoretic background and notation

A graph is denoted $G = (V, E)$, with $n = |V|$, and $m = |E|$. A vertex sequence $v_1 - v_2 - ... - v_k$ describes a *path* if $v_i v_{i+1}$ is an edge for $1 \leq i < k$. The *length* of a path is the number of edges in it. A *cycle* is a path that starts and ends with the same vertex, and the length of the cycle is the number of vertices or edges it contains. A *chord* of a cycle (path) is an edge connecting two non-consecutive vertices of the cycle (path). A *clique* is a set of vertices that are pairwise adjacent.

For the following definitions, we will omit subscript G when the graph is clear from the context. The *neighborhood* of a vertex v in G is $N_G(v) = \{u \neq v \mid uv \in E\}$, and for a set of vertices A, $N_G(A) = \cup_{x \in A} N_G(x) \setminus A$. A *simplicial* vertex is one whose neighborhood induces a clique. $G(A)$ is the subgraph induced by a vertex set $A \subseteq V$, but we often denote it simply by A when there is no ambiguity. We would like to stress that we distinguish between subgraphs and induced subgraphs.

For any vertex set $S \subseteq V$ and any vertex $x \in V \setminus S$, C_S^x denotes the connected component of $G(V \setminus S)$ containing x. A subset S of V is called a *separator* if $G(V \setminus S)$ is disconnected. S is a *u, v-separator* if vertices u and v are in different connected components of $G(V \setminus S)$, and a *minimal u, v-separator* if no subset of S is a u, v-separator. S is a *minimal separator* of G if there is some pair $\{u, v\}$ of vertices in G such that S is a minimal u, v-separator. Equivalently, S is a minimal separator if there exist two connected components C_1 and C_2 of $G(V \setminus S)$ such that $N_G(C_1) = N_G(C_2) = S$.

A pair of non-adjacent vertices $\{u, v\}$ is a *2-pair* in G if there is no chordless path of length 3 or more between u and v [24]. If G is not connected, then two vertices that belong to different connected components constitute a 2-pair by definition. If G is connected, it has been shown that $\{u, v\}$ is a 2-pair if and only if $N(u) \cap N(v)$ is a minimal u, v-separator of G [1, 33].

A graph is *chordal* if it contains no chordless cycle of length ≥ 4. Consequently, all induced subgraphs of a chordal graph are also chordal. G is chordal if and only if every minimal separator of G is a clique [19]. Chordal graphs are the intersection graphs of subtrees of a tree [14, 22, 34], and the following result gives a very useful tool which we will use as a data structure in our algorithm.

Theorem 2.1 (Buneman [14], Gavril [22], Walter[34]) *A graph G is chordal if and only if there exists a tree T, whose vertex set is the set of maximal cliques of G, that satisfies the following property: for every vertex v in G, the set of maximal cliques containing v induces a connected subtree of T.*

Such a tree is called a *clique tree* [12], and we will refer to the vertices of T as *tree nodes* to distinguish them from the vertices of G, and sometimes also as *bags* since these contain several graph vertices. Each tree node of T is thus a vertex set of G corresponding to a maximal clique of G. We will not distinguish between maximal cliques of G and their corresponding tree nodes. In addition, it is customary to let each edge K_iK_j of T hold the vertices of $K_i \cap K_j$, where K_i and K_j are maximal cliques of G. Thus, edges of T are also vertex sets. Although a chordal graph can have many different clique trees, all chordal graphs share the following important properties that are related to an efficient implementation of our algorithm.

Theorem 2.2 (Buneman [14], Ho and Lee[27], Lundquist [29]) *Let T be a clique tree of a chordal graph G. A set S is a minimal separator of G if and only if $S = K_i \cap K_j$ for an edge K_iK_j in T, and if $S = K_i \cap K_j$ for an edge K_iK_j in T, then S is a minimal u, v-separator for any $u \in K_i \setminus S$ and $v \in K_j \setminus S$.*

Theorem 2.3 (Blair and Peyton [12]) *T is a clique tree of G if and only if T is a tree whose nodes are the maximal cliques of G, and for every pair of distinct maximal cliques K_i and K_j in G the intersection $K_i \cap K_j$ is contained in every node of T (maximal clique of G) appearing on the path between K_i and K_j in T.*

Note that as a consequence, the intersection $K_i \cap K_j$ is also contained in every edge of T (i.e., in every minimal separator of G) appearing on the path between K_i and K_j in T. A chordal graph has at most n maximal cliques [19] and hence the number of nodes and edges in a clique tree is $O(n)$ [21].

From any given non-chordal graph, one can obtain a chordal graph on the same vertex set by either adding (the added edges are called *fill* edges) or removing edges. $M = (V, F)$ is called a *triangulation* of an arbitrary graph $G = (V, E)$ if $E \subseteq F$ and M is chordal. M is a *minimal triangulation* of G if no proper subgraph of M is a triangulation of G. Similarly, $H = (V, D)$ is called a *chordal subgraph*, or equivalently a *subtriangulation*, of G if $D \subseteq E$ and H is chordal. H is a *maximal chordal subgraph*, or a *maximal subtriangulation*, if (V, D') is non-chordal for every set D' that satisfies $D \subset D' \subseteq E$. By the results of [32], a given triangulation (subtriangulation) is minimal (maximal) if and only if no single fill edge can be removed (no single removed edge can be added back) without destroying chordality.

3 A new characterization of chordal graphs

In this section we present a new characterization of chordal graphs that will be the basis of our algorithm.

Definition 3.1 *An edge uv is* mono-saturating *in $G = (V, E)$ if $\{u, v\}$ is a 2-pair in $G' = (V,\ E \setminus \{uv\})$.*

Theorem 3.2 *A graph is chordal if and only if every edge is mono-saturating.*

Proof. Let $G = (V, E)$ be chordal, and assume on the contrary that there is an edge $uv \in E$ that is not mono-saturating. Then there is a chordless path P of length more than 2 between u and v in $G' = (V,\ E \setminus \{uv\})$, and thus the following is a chordless cycle of length at least 4 in G: $u - P - v - u$, which contradicts our assumption that G is chordal.

For the other direction, let every edge in G be mono-saturating, and assume on the contrary that G is not chordal. Thus, there exists a chordless cycle C of length at least 4 in G. No edge of C is mono-saturating, a contradiction. ∎

As a corollary of Theorem 3.2, we can deduce the following characterization by 2-pairs by [10], a result that was also observed with a different formulation in [18].

Corollary 3.3 (Berry et al. [10]) *Given a chordal graph $G = (V, E)$, where $uv \notin E$, the graph $H = (V,\ E \cup \{uv\})$ is chordal if and only if $\{u, v\}$ is a 2-pair in G.*

Proof. Let us on the contrary assume that H is not chordal, and that $\{u, v\}$ is a 2-pair in G. The edge uv is mono-saturating in H since $\{u, v\}$ is a 2-pair of G. By Theorem 3.2 there exists an edge xy in G that is not mono-saturating in H, and by Definition 3.1 there exists a chordless path P in $H' = (V,\ (E \setminus \{xy\}) \cup \{uv\})$ preventing xy from being mono-saturated in H. One of the edges in P is uv, since G is chordal and by Theorem 3.2 xy is mono-saturating in G, and by Definition 3.1 P do not exists in $G' = (V,\ E \setminus \{xy\})$. By removing the edge uv from P and inserting xy we obtain a chordless path P' in G, which prevents $\{u, v\}$ from being a 2-pair in G, and thus we have a contradiction.

For the other direction, we know that $\{u, v\}$ is not a 2-pair in G, and thus the edge uv in H is not mono-saturating, and by Theorem 3.2 H is not chordal. ∎

As a consequence, while maintaining a chordal graph by adding edges, we could check every edge of the input graph to see if the endpoints constitute a 2-pair in the transitory chordal subgraph. However, this approach requires that we check every edge several times, as pairs of vertices can become 2-pairs only after the addition of some other edges. Our main result, to be presented as Theorem 3.8, gives a more powerful tool that allows examining each edge of the input graph only *once* during such a process.

Assume the following scenario: given a chordal graph G, we want to add an edge uv to G. Since we want the resulting graph to remain chordal, it may be necessary to add other edges to achieve this. However, we allow addition of edges only incident to u.[1] Naturally, if we add every edge between u and the other vertices of G, the resulting graph is chordal. Our main goal is to add as few edges as possible.

Definition 3.4 *Given a chordal graph $G = (V, E)$ and any pair of non-adjacent vertices u and v in G, $R(G, u, v) = \{ux \mid x$ belongs to a minimal u, v-separator of $G\}$. We will call $R(G, u, v)$* the incident-to-u set of required edges for uv.

[1]Note that in the incremental approach described in the next section, vertex u is the most recently added vertex.

Lemma 3.5 *Let $G = (V, E)$ be a chordal graph and let u and v be non-adjacent vertices of G. Then $R(G, u, v) = \{ux \mid x$ is an intermediate vertex of a chordless path in G between u and $v\}$.*

Proof. Let $ux \in R(G, u, v)$, S be a minimal u, v-separator of G containing x, P_1 be a chordless path in G between u and x with all intermediate vertices belonging to C^u_S, and P_2 be a chordless path in G between x and v with all intermediate vertices belonging to C^v_S. The path obtained by concatenating P_1 and P_2 is a chordless path in G between u and v having x as an intermediate vertex.

Conversely, let x be an intermediate vertex of a chordless path P in G between u and v. Vertex set S' obtained from V by removing all vertices of P except x is a u, v-separator of G. Let S be a minimal u, v-separator of G included in S'. Vertex x belongs to S because otherwise P would be a path in $G(V \setminus S)$ between u and v. Therefore $ux \in R(G, u, v)$. ∎

Lemma 3.6 *Let $G = (V, E)$ be a chordal graph, let u and v be non-adjacent vertices of G, let S be a minimal u, v-separator of G, and let graph $M = (V, E \cup \{uv\} \cup R(G, u, v))$. Then any chordless cycle in M of length at least 4 containing u contains at most one vertex of S.*

Proof. Suppose on the contrary that some chordless cycle C in M of length at least 4 contains u and distinct vertices x and x' of S. As a minimal separator of a chordal graph, S is a clique in G, so xx' is an edge of M, and by definition of $R(G, u, v)$, ux and ux' are also edges of M. So C has a chord in M, a contradiction. ∎

Lemma 3.7 *Let $G = (V, E)$ be a chordal graph, let u and v be non-adjacent vertices of G, and let S and S' be minimal u, v-separators of G. Then ($S' \subseteq S \cup C^u_S$ and $S \subseteq S' \cup C^v_{S'}$) or ($S' \subseteq S \cup C^v_S$ and $S \subseteq S' \cup C^u_{S'}$).*

Proof. We may assume without loss of generality that S and S' are distinct. As minimal separators of a chordal graph, S and S' are cliques, and since S and S' are distinct minimal u, v-separators of G, $S \setminus S' \neq \emptyset$. Let $x \in S \setminus S'$. Observe that $S' \cap (C^u_S \cup C^v_S) \neq \emptyset$, because otherwise $C^u_S \cup \{x\} \cup C^v_S$ would be a connected subset of $V \setminus S'$, which would contradict S' being a u, v-separator of G. Let $x' \in S' \cap (C^u_S \cup C^v_S)$. We first study the case when $x' \in C^u_S$: Since S' is a clique containing x', $S' \subseteq \{x'\} \cup N(x') \subseteq S \cup C^u_S$. It follows that $\{x\} \cup C^v_S$ is a connected subset of $V \setminus S'$, and therefore $x \in C^v_{S'}$. Since S is a clique containing x, $S \subseteq \{x\} \cup N(x) \subseteq S' \cup C^v_{S'}$. For the case when $x' \in C^v_S$, we prove in a similar way that $S' \subseteq S \cup C^v_S$ and $S \subseteq S' \cup C^u_{S'}$. ∎

Theorem 3.8 *Let $G = (V, E)$ be a chordal graph, let u and v be non-adjacent vertices of G, and let graph $M = (V,\ E \cup \{uv\} \cup R(G, u, v))$. Then M is chordal and M is a subgraph of any triangulation of $G' = (V,\ E \cup \{uv\})$ obtained from G' by adding edges incident to u only.*

Proof. Let us first show that M is chordal. Assume on the contrary that M is not chordal, and let C be a chordless cycle of length at least 4 in M. Since G is chordal, C contains an edge $ux \in \{uv\} \cup R(G, u, v)$. Let $C = u - x' - y_1 - y_2 - ... - y_k - x - u$ with $k \geq 1$, and $P_1 = x' - y_1 - y_2 - ... - y_k - x$, which is a chordless path in G. It is sufficient to show that P_1 is a subpath of a chordless path P in G between u and v, since then by Lemma 3.5 uy_1 would belong to $R(G, u, v)$ and therefore would be a chord of C in M, giving a contradiction. In the following $Q_1 \cdot Q_2$ denotes the path obtained by concatenating paths Q_1 and Q_2.

First case : $x = v$ or $x' = v$. Say, $x = v$. If $ux' \in E$ then we are done with $P = u - x' \cdot P_1$. Otherwise $ux' \in R(G, u, v)$, let S be a minimal u, v-separator of G containing x', and let P_0 be a chordless path in G between u and x' with all intermediate vertices belonging to C^u_S. By Lemma 3.6, x' is the only vertex of S in P_1, so all intermediate vertices of P_1 belong to C^v_S. It follows that path $P = P_0 \cdot P_1$ is a chordless path in G between u and v.

Second case : $x \neq v$ and $x' \neq v$. In this case, $ux \in R(G, u, v)$. Let S be a minimal u, v-separator of G containing x and let P_2 be a chordless path in G between x and v with all intermediate vertices belonging to C^v_S. If $ux' \in E$ then by Lemma 3.6, all intermediate vertices of $u - x' \cdot P_1$ belong to C^u_S, so path $P = u - x' \cdot P_1 \cdot P_2$ is a chordless path in G between u and v. Otherwise $ux' \in R(G, u, v)$. Let S' be a minimal u, v-separator of G containing x' and let P_0 be a chordless path in G between u and x' with all intermediate vertices belonging to $C^u_{S'}$. By Lemma 3.7, ($S' \subseteq S \cup C^u_S$ and $S \subseteq S' \cup C^v_{S'}$) or ($S' \subseteq S \cup C^v_S$ and $S \subseteq S' \cup C^u_{S'}$). We may assume without loss of generality that $S' \subseteq S \cup C^u_S$ and $S \subseteq S' \cup C^v_{S'}$. By Lemma 3.6, $x' \notin S$ and $x \notin S'$, so $x' \in C^u_S$ and $x \in C^v_{S'}$. Hence by Lemma 3.6, all intermediate vertices of P_1 belong to $C^v_{S'}$ and C^u_S. Since P_0 and P_1 are chordless and all vertices of P_0 other than x' belong to $C^u_{S'}$, and those of P_1 other than x' belong to $C^v_{S'}$, path $Q = P_0 \cdot P_1$ is chordless. Since $S \subseteq S' \cup C^v_{S'}$, $C^u_{S'} \subseteq C^u_S$, so all vertices of P_0 belong to C^u_S. Since Q and P_2 are chordless and all vertices of Q other than x belong to C^u_S, and those of P_2 other than x belong to C^v_S, path $P = Q \cdot P_2$ is a chordless path in G between u and v, which completes the proof of chordality of M.

Let M' be a triangulation of $G' = (V,\ E \cup \{uv\})$ obtained from G' by adding edges incident to u only. Let us show that M is a subgraph of M', i.e., that every edge of $R(G, u, v)$ is an edge of M'. Suppose on the contrary that there is some edge $ux \in R(G, u, v)$ which is not an edge of M'. By Lemma 3.5, x is an intermediate vertex of a chordless path $u - y_1 - y_2 - ... - y_i = x - ... - y_k = v$ in G. Thus $C = u - y_1 - y_2 - ... - y_k = v - u$ is a cycle in M' such that path

$y_1 - y_2 - ... - y_k = v$ is chordless in M'. Let r be the largest integer smaller than i such that y_r is adjacent to u in M' and s be the smallest integer larger than i such that y_r is adjacent to u in M'. Then $u - y_r - y_{r+1} - ... - y_i = x - ... - y_{s-1} - y_s - u$ is a chordless cycle in M' of length at least 4, which contradicts the chordality of M'. ∎

The following corollary follows directly from Theorem 3.8.

Corollary 3.9 *Let $G = (V, E)$ be a chordal graph and let u and v be any pair of non-adjacent vertices in G. Then $M = (V,\ E \cup \{uv\} \cup R(G, u, v))$ is the unique minimal triangulation of $G' = (V,\ E \cup \{uv\})$ obtained from G' by adding edges incident to u only.*

A significant consequence of our main theorem is that it is sufficient to determine $R(G, u, v)$ and add it to the current graph G to obtain a new chordal graph. This can be done efficiently, as will be explained in Section 5. This involves maintaining the minimal separator structure of a chordal graph, a problem for which we have a new and efficient data structure associated with a clique tree, which we will describe in Section 5.

4 A vertex incremental algorithm for simultaneous maximal subtriangulation and minimal triangulation

In this section we apply our results of Section 3 to the problem of computing a maximal chordal subgraph $H = (V, D)$ and a minimal triangulation $M = (V, F)$ of an arbitrary graph $G = (V, E)$, where $D \subseteq E \subseteq F$.

Our algorithm is based on the following vertex incremental principle. Start with an empty subset U of V, and a maximal chordal subgraph H of $G(U)$ (respectively a minimal triangulation M of $G(U)$ if we want a minimal triangulation algorithm). The incremental approach is to increase U with a vertex u from $V \setminus U$ at each step. Observe that H (resp. M) is chordal and disconnected after the introduction of u as long as $|U| \geq 1$, since no edges are introduced along with this vertex, and H (resp. M) was chordal before this step. Then for each edge of G incident to u and some vertex v in $U \setminus \{u\}$, we do computations according to Theorem 3.8 and obtain the set $R(H, u, v)$ (resp. $R(M, u, v)$) of edges incident to u that must be added along with uv in order to obtain a chordal supergraph of H (resp. M).

In the case of the maximal subtriangulation algorithm, we will only add uv and $R(H, u, v)$ to $E(H)$ if $R(H, u, v) \subset E(G)$. In the case of the minimal triangulation algorithm, the required edges $R(M, u, v)$ and the edge uv are added to $E(M)$. To prove that this approach actually produces a maximal chordal subgraph (resp. minimal triangulation) we rely on the results in the following two lemmas.

Lemma 4.1 *Given $G = (V, E)$, let $U \subseteq V$, and let H and H' be graphs such that H is a subgraph of H', H' is a chordal subgraph of G, and $H(U)$ is a maximal chordal subgraph of $G(U)$. Then $H(U) = H'(U)$.*

Proof. As an induced subgraph of a chordal graph $H'(U)$ is chordal, and therefore is a chordal subgraph of $G(U)$ having $H(U)$ as a subgraph. So, since $H(U)$ is a maximal chordal subgraph of $G(U)$, $H(U) = H'(U)$. ▪

For the computation of H, assume that H is a maximal chordal subgraph of $G(U)$ on vertex set U. At the first step, U contains a single vertex of G and $H = G(U)$. At each step, a vertex $u \in V \setminus U$ is chosen and added to U and thus to H. Now for each edge uv of G with $v \in U$, we add edge uv to H if and only if every edge of $R(H, u, v)$ is present in G. If uv is added to H, we also add every edge of $R(H, u, v)$ at the same time. After this, none of the edges that are added need to be examined again for possible addition, since they already appear in the transitory chordal subgraph. If some edge of $R(H, u, v)$ is not an edge of G, then we cannot add uv at this step by Theorem 3.8, since we only allow addition of edges incident to u. When we prove the correctness of our algorithm, it will be clear that uv never needs to be examined again for addition. [2] Thus, each edge is examined for addition at most once, and in many cases several edges are added at the same time and disappear from the list of edges that still need to be examined, which is the strength of our algorithm with respect to time complexity. In addition, our algorithm does not touch the unprocessed vertices. Thus, these vertices need not be known in advance, and we can actually take a new vertex u as input in an on-line fashion at each step.

Lemma 4.2 *Given $G = (V, E)$, let $U \subseteq V$, and let M and M' be graphs such that M' is a subgraph of M, M' is a triangulation of G, and $M(U)$ is a minimal triangulation of $G(U)$. Then $M(U) = M'(U)$.*

Proof. As an induced subgraph of a chordal graph, $M'(U)$ is chordal, and therefore is a triangulation of $G(U)$ and a subgraph of $M(U)$. So, since $M(U)$ is a minimal triangulation of $G(U)$, $M(U) = M'(U)$. ▪

For the computation of M, assume that M is a minimal triangulation of $G(U)$ on the vertex set U. The only difference from the discussion above in this case is that, for each edge uv of G with $v \in U$, we add to M edge uv as well as *every edge belonging to* $R(M, u, v)$ regardless of whether or not these edges belong to G. Thus, the difference between the two processes is merely a single **if**

[2]Note that considering $R(H, v, u)$ does not help, since we have already concluded that adding any edge between v and vertices in $U \setminus \{u\}$ will create a chordless cycle.

statement. Our algorithm can be changed by inserting or deleting this **if** line in order to change between the processes of computing a minimal triangulation and a maximal chordal subgraph, though of course both graphs can be computed by a single algorithm within the same time bound.

With the data structure details given in the next section, we will show that computing and adding the set $R(H, u, v)$ can be done in $O(n)$ time for each examined edge uv. From our algorithm and its proof of correctness, it will be clear that every edge needs to be examined at most once. We are now ready to present our algorithm. We begin with the maximal chordal subgraph version.

Algorithm Incremental Maximal Subtriangulation **(IMS)**
Input: $G = (V, E)$.
Output: A maximal chordal subgraph $H = (V, D)$ of G.

```
01. Pick a vertex $s$ of $G$;
02. $U = \{s\}$;
03. $D = \emptyset$;
04. for $i = 2$ to $n$ do
05.     Pick a vertex $u \in V \setminus U$;
06.     $U = U \cup \{u\}$;
07.     $N = \emptyset$;
08.     for each vertex $w \in N_G(u)$
09.         if $w \in U$ then
10.             $N = N \cup \{w\}$;
11.         end-if
12.     end-for
13.     while $N$ is not empty do
14.         Pick a vertex $v \in N$;
15.         $N = N \setminus \{v\}$;
16.         $X = \{x \mid x$ belongs to a minimal $u, v$-separator of $H = (U, D)\}$;
17.         $R = \{ux \mid x \in X\}$;
18.         if $R \subseteq E$ then
19.             $D = D \cup \{uv\} \cup R$;
20.             $N = N \setminus X$;
21.         end-if
22.     end-while
23.     $H = (U, D)$;
24. end-for
```

Let us call **IMT** (Incremental Minimal Triangulation) the algorithm that results from removing lines 18 and 21 of Algorithm **IMS**. Thus, in **IMT**, edge set $\{uv\} \cup R$ is always added to the transitory graph for every examined edge uv.

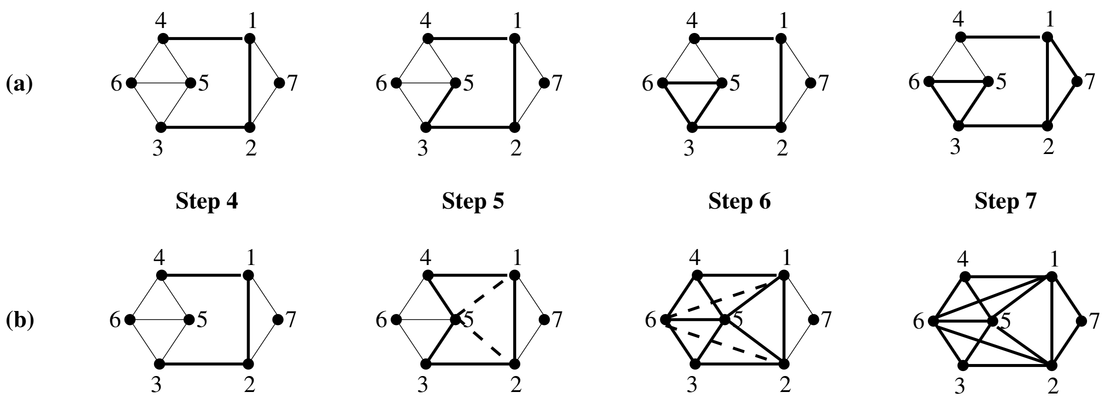

Figure 1: The figure shows graph H in thick lines after steps 4, 5, 6, and 7 of (a) Algorithm **IMS** when computing a maximal chordal subgraph and (b) Algorithm **IMT** when computing a minimal triangulation.

In Example 4.3, executions of both of these algorithms are shown on the same input graph. Figure 1 (a) shows **IMS** and (b) shows **IMT**.

Example 4.3 *Consider Figure 1. The vertices of the input graph are processed in the order shown by the numbers on the vertices. At step 1, only vertex* 1 *is added to H. At step 2, vertex* 2 *and edge* 21 *are added, and similarly at steps 3 and 4, vertex* 3 *and edge* 32*, and vertex* 4 *and edge* 41 *are added, respectively. The first column of the figure shows graph H with thick lines on the input graph after these 4 steps. The chordal graph so far is the same for both the maximal chordal subgraph (a), and the minimal triangulation (b). We will explain the rest of the executions in more detail.*

(a) *At step 5, $N = \{3, 4\}$, and edge* 53 *is examined first. In this case, set X is empty, and edge* 53 *is thus added. For the addition of edge* 54*, $X = \{1, 2, 3\}$, and since required edges* 51 *and* 52 *are not present in G, edge* 54 *is not added. At step 6, $N = \{3, 4, 5\}$, and edge* 63 *is examined first and added since X is empty. For the addition of edge* 64*, $X = \{1, 2, 3\}$, and since required edges* 61 *and* 62 *are not present in G, edge* 64 *is not added. For the addition of edge* 65*, $X = \{3\}$, and* 65 *is added since edge* 63 *is present in G and in H.*

(b) *At step 5, edge* 53 *is added as in (a), and in addition, edge* 54 *is added along with the required edges* 51 *and* 52*. At step 6, edge* 63 *is added as in (a). For the addition of edge* 64*, $X = \{1, 2, 3, 5\}$ since the minimal* 6, 4*-separators are* $\{1, 5\}, \{2, 5\}$*, and* $\{3\}$*. Thus, edge* 64 *and required edges* 61*,* 62*, and* 65 *are added to M.*

Step 7 adds edges 71 *and* 72 *in both (a) and (b) without requiring any additional edges in either case.*

Theorem 4.4 *Algorithm* **IMT** *computes a minimal triangulation, and Algorithm* **IMS** *computes a maximal chordal subgraph, of the input graph.*

Proof. Let $(u_1, u_2, ..., u_n)$ be the sequence of vertices of G successively added to U in an execution of Algorithm **IMT** (resp. **IMS**), and let $U_i = \{u_1, u_2, ..., u_i\}$ for any i from 1 to n.

Algorithm **IMT** : Let $M = (V, F)$ be the output graph. We show by induction that M is a minimal triangulation of G.

Induction hypothesis: $M(U_i)$ is a minimal triangulation of $G(U_i)$, for $1 \leq i \leq n$.

The base case $i = 1$ trivially holds. Assume that $M(U_{i-1})$ is a minimal triangulation of $G(U_{i-1})$ for some i between 2 and n, and we will show that this implies that $M(U_i)$ is chordal and is equal to any triangulation M' of $G(U_i)$ that is a subgraph of $M(U_i)$.

Let M' be a triangulation of $G(U_i)$ that is a subgraph of $M(U_i)$. By Lemma 4.2, $M(U_{i-1})$ is chordal and $M(U_{i-1}) = M'(U_{i-1})$. Let $u = u_i$, $U = U_i$, and $(v_1, v_2, ..., v_k)$ be the sequence of neighbors of u in G successively picked out of N after adding u to U. Let $M_j = (U, F_j)$ be the transitory graph after processing edge uv_j, for $0 \leq j \leq k$ (M_0 is the transitory graph after adding u to U). Let us prove by induction that M_j is chordal and is a subgraph of M' for $0 \leq j \leq k$. The base case $j = 0$ holds since M_0 is obtained from $M(U_{i-1})$ by adding vertex u, $M(U_{i-1})$ is chordal and $M(U_{i-1}) = M'(U_{i-1})$. Assume that M_{j-1} is chordal and is a subgraph of M', for some j between 1 and k. Let us show that this implies that the same is true for M_j, too. $M_j = (U,\ F_{j-1} \cup \{uv_j\} \cup R(M_{j-1}, u, v_j))$. Thus M' is a triangulation of $M'' = (U,\ F_{j-1} \cup \{uv_j\})$ obtained from M'' by adding edges incident to u only, since $M''(U_{i-1}) = M(U_{i-1}) = M'(U_{i-1})$. By Theorem 3.8, M_j is chordal and is a subgraph of M', which completes this part of the proof by induction on j.

As a consequence, $M(U_i) = M_k$ is chordal and is a subgraph of M' and since M' is a subgraph of $M(U_i)$, $M' = M(U_i)$. This completes the proof by induction on i, and thus M is a minimal triangulation of G.

Algorithm **IMS** : Let $H = (V, D)$ be the output graph. We show again by induction that H is a maximal chordal subgraph of G.

Induction hypothesis: $H(U_i)$ is a maximal chordal subgraph of $G(U_i)$, for $1 \leq i \leq n$.

The base case $i = 1$ trivially holds. Assume that $H(U_{i-1})$ is a maximal chordal subgraph of $G(U_{i-1})$, for some i between 2 and n. Let us show that $H(U_i)$ is chordal and is equal to any chordal subgraph H' of $G(U_i)$ having $H(U_i)$ as a subgraph.

Let H' be a chordal subgraph of $G(U_i)$ having $H(U_i)$ as a subgraph. By Lemma 4.1, $H(U_{i-1})$ is chordal and $H(U_{i-1}) = H'(U_{i-1})$. We define u, U, $(v_1, v_2, ..., v_k)$, and $H_j = (U, D_j)$ for $0 \leq j \leq k$ as above. Let us prove by induction on j that H_j is chordal, and if $j \geq 1$ and uv_j is an edge of H', then $uv_j \in D_j$. The base case $j = 0$ holds since H_0 is obtained from chordal graph $H(U_{i-1})$ by adding vertex u. Suppose that H_{j-1} is chordal, and if $j-1 \geq 1$ and uv_{j-1} is an edge of H', then $uv_{j-1} \in D_{j-1}$, for some j between 1 and k. We show that this implies that the same is true for H_j, v_j and D_j. Let $K = (U,\ D_{j-1} \cup \{uv_j\} \cup R(H_{j-1}, u, v_j))$. H_j is either equal to H_{j-1} or to K, and since H_{j-1} is chordal, by Theorem 3.8, H_j is chordal. Now we assume that uv_j is an edge of H'. Let us show that $uv_j \in D_j$. H' is a triangulation of $H'' = (U,\ F_{j-1} \cup \{uv_j\})$ obtained from H'' by adding edges incident to u only (since $H''(U_{i-1}) = H(U_{i-1}) = H'(U_{i-1})$). By Theorem 3.8, K is a subgraph of H', and therefore of $G(U)$. It follows that $R(H_{j-1}, u, v_j) \subseteq E$, which is the condition for adding edge uv_j, so $uv_j \in D_j$. Thus we have completed the part of the proof by induction on j.

As a consequence, $H(U_i) = H_k$ is chordal and every edge of H' incident to u which has been processed is an edge of $H(U_i)$. Since moreover $H(U_{i-1}) = H'(U_{i-1})$, unprocessed edges of G are edges of H and $H(U_i)$ is a subgraph of H', $H' = H(U_i)$. This completes the proof by induction on i, and thus H is a maximal chordal subgraph of G. ■

5 Data structure details and time complexity

The input graph G is represented by adjacency list data structure, and we use a clique tree T of H as an additional data structure to store and work on the transitory graph H. Thus, after the first step, T has only one tree node, which contains start vertex s. As H grows, T will grow maintaining a correct clique tree of H at all steps. Note that T will not always be connected at intermediate steps, as H is not necessarily connected. In this case, each connected component of T will be a correct clique tree of the corresponding connected component of H.[3]

In what follows we describe an implementation of each of the following operations.

1. Compute the union X of all minimal u,v-separators in H, which gives the required edge set $R(H, u, v)$.

2. If $R(H, u, v) \cup \{uv\}$ is to be added to H, update T to reflect this modification of H.

Each of these operations will be shown to require only $O(n)$ time for each examined edge uv of G. We will devote a subsection to each of the above mentioned operations. Subsection 5.1 describes how T is modified to obtain a path $P_{u,v}$

[3]We could have picked the new vertex u such that $u \in N_G(U)$, but this would result in less general algorithms unnecessarily, and we want our algorithms to have on-line implementations.

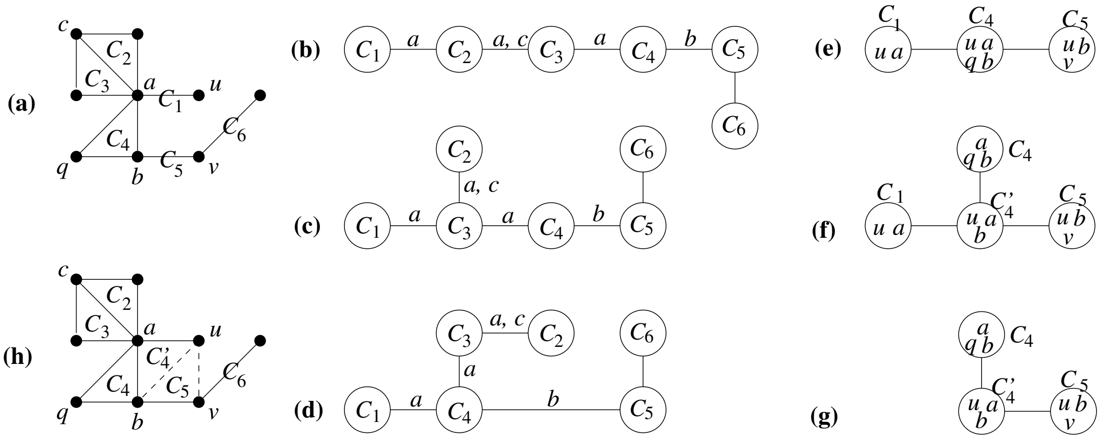

Figure 2: A chordal graph H is given in (a), and (b),(c), and (d) shows a clique tree of H, where $C_u = C_1$, $C_v = C_5$, and $P_{u,v}$ is the path between C_1 and C_5. After steps (c) and (d), path $P_{u,v}$ between C_1 and C_5 is in the desired form, and only this portion of the tree is shown after step (d). In step (e), u is placed in every tree node on $P_{u,v}$, and in step (f) C_4 is separated from the path since edge uq is not intended. C_1 is removed in (g) since it becomes non-maximal. The new corresponding graph H of which the modified tree is a clique tree is shown in (h).

such that every tree edge on $P_{u,v}$ is a distinct minimal u, v-separator (Figure 2(b)-(d)), and how the union X of all these minimal separators is computed from $P_{u,v}$. Subsection 5.2 describes how T is further modified to reflect the addition of new edges to H (Figure 2(e)-(g)), and how to ensure that every tree node in T is a unique maximal clique of H after the modifications.

Since we examine each edge at most once, and there are m edges, the desired time bound will then follow. An illustration of what happens for each examined edge uv is summarized in Figure 2; we will refer to parts of this figure as we explain the details in the coming subsections. The main idea is to use a path $P_{u,v}$ of the current clique tree T between a tree node C_u that contains u and a tree node C_v that contains v, and compute the union of tree edges on this path that correspond to minimal u, v-separators. Unfortunately, the sum of the sizes of these edges can be larger than $O(n)$; in fact each edge can be of size $O(n)$. Thus, if the tree nodes and tree edges of T are implemented simply as vertex lists containing vertices of each tree node and edge, then Operation 1 described above cannot be accomplished in $O(n)$ time. For this reason, we present a special kind of implementation of the clique tree, as described below.

Every edge CC' of T is implemented as two lists that we will call *difference lists* (*difflists* for short). One list contains vertices belonging to $C \setminus C'$. This list has two names; it is called both *add*(C', C) and *remove*(C, C'). The other list contains vertices belonging to $C' \setminus C$. This list is called *add*(C, C') and also

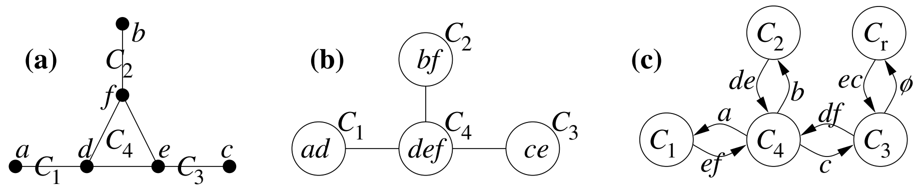

Figure 3: A chordal graph H and its maximal cliques are given in (a). A clique tree of H is given in (b), where each tree node contains the vertices of a unique maximal clique in H. The difference list representation of the same clique tree as in (b) is presented in (c). Tree node C_r is an empty tree node, that is used when we want to compute the set of vertices contained in a tree node. The arrows represent the add and remove lists, and the vertices contained in each list are given by the label attached to the arrow.

$remove(C', C)$. Now, if every tree node C of T contains pointers to its *add* and *remove* lists, *add* and *remove* lists are in edges of T and not in every clique C of T then C actually does not need to store a list of vertices that it contains. Vertices belonging to C can be computed by using the edges incident to C as follows. For every edge CC', if we know the set of vertices belonging to C', then we add $add(C', C)$ to this set and remove $remove(C', C)$ from this set to get the set of vertices belonging to C. In order to have a starting tree node we need to know the content of one tree node, this might be an empty tree node. Note that the add and remove lists and the structure of the tree are the only stored information in this data structure. In Figure 3 a chordal graph is given in (a), while the regular clique tree of this graph is given in (b), and the clique tree represented by difference lists is given in (c).

5.1 Finding the minimal u, v-separators and computing X

Let K_u be any maximal clique of H that contains u, and let K_v be any maximal clique that contains v. If K_u and K_v are contained in different connected components of T, then $X = \emptyset$ and there is nothing to compute. Let us for the rest of this subsection assume that K_u and K_v are contained in the same connected component of T.

On the path from K_u to K_v in T, do a search from K_u to K_v and let C_u be the tree node closest to K_v containing u. Do a similar search from K_v to K_u and let C_v be the tree node closest to K_u containing v. Let $P_{u,v}$ denote the path of T between C_u and C_v. An example graph H and $P_{u,v}$ are given in Figure 2 (a) and (b), respectively.

Claim 5.1 *Every minimal u, v-separator appears as an edge of $P_{u,v}$, and every edge on $P_{u,v}$ separates u and v.*

Proof. By Theorem 2.1 we know that every internal node of $P_{u,v}$ contains neither u nor v. Theorem 2.3 ensures that every minimal separator S appearing as an edge of $P_{u,v}$ separates u and v, since removing S will separate C_u from C_v in the clique tree, thus also separate u and v in the graph. Conversely, for any minimal u, v-separator S of the graph, S contains as a subset some edge S' of $P_{u,v}$, since removing S will separate u and v in the graph, thus also separate C_u from C_v in the clique tree. Now, since S is a minimal u, v-separator containing S' and by the first part of this proof, the edge S' of $P_{u,v}$ separates u and v, S is equal to S'. Thus S appears as an edge of $P_{u,v}$. ■

However, some of the tree edges on $P_{u,v}$ might be non-minimal u, v-separators, and some minimal u, v-separators might appear several times as edges of $P_{u,v}$. We will first modify T in $O(n)$ time so that path $P_{u,v}$ between C_u and C_v contains only distinct minimal u, v-separators as its edges.

Observe first that, since every vertex can appear only once in an *add* list and once in a *remove* list on the path $P_{u,v}$, the sum of the lengths of the *add* and *remove* lists on the path $P_{u,v}$ is at most $2n$. We obtain our time bound by reading every *add* and *remove* list in $P_{u,v}$ at most a constant number of times. The maximal cliques (tree nodes) on $P_{u,v}$ are named $C_1, C_2, ..., C_k$, where $C_u = C_1$ is the tree node containing vertex u, and $C_v = C_k$ is the tree node containing v, and edge $S_i = S_{i,i+1} = C_i C_{i+1}$ is an edge of $P_{u,v}$, for $1 \leq i \leq k-1$. We will describe how unnecessary maximal cliques can be removed from $P_{u,v}$, and after each removal, we will assume that the remaining maximal cliques and minimal separators are resorted as $C_1, C_2, ...C_k$, so that before we explain each new modification, we have a consecutive numbering of the maximal cliques on $P_{u,v}$. When we must remove edges of $P_{u,v}$ and insert edges between two non-consecutive tree nodes of $P_{u,v}$, we will need a more general way of naming the intersections between two maximal cliques that are not necessarily adjacent in T. We let $S_{i,j} = C_i \cap C_j$ denote the intersection between C_i and C_j.

Claim 5.2 *Assume that there is a tree edge S_j on $P_{u,v}$, such that either S_j is not a minimal u, v-separator in H or S_j is equal to another minimal separator appearing on $P_{u,v}$. Then there exists a tree edge S_i such that $S_i \subseteq S_{i+1}$ or $S_i \subseteq S_{i-1}$.*

Proof. Observe first that, by Claim 5.1, every minimal u, v-separator appears as an edge of $P_{u,v}$, and every edge on this path separates u from v. Thus, S_j is a (not necessarily minimal) u, v-separator. Consequently, there exists a minimal u, v-separator S_i such that $S_i \subseteq S_j$ and $i \neq j$. Let S_{i+1} or S_{i-1} be the edge adjacent to S_i in $P_{u,v}$ in the direction of S_j. Note that $i+1$ or $i-1$ and j might be equal. It follows from Theorem 2.3 that $S_i \subseteq S_{i+1}$ or $S_i \subseteq S_{i-1}$ since $S_i \subseteq S_j$. ■

From Claim 5.2, we can conclude that, if no two adjacent tree edges are comparable by subset relation, then $P_{u,v}$ contains only distinct minimal u, v-separators. We will test adjacent tree edges on $P_{u,v}$, and remove the ones that include their neighbors as a subset. In order to obtain our time bound we have to do this test in the difflist data structure.

Claim 5.3 *Let the following path C_i, S_i, C_j, S_j, C_l be a subpath of $P_{u,v}$. Then $S_i \subseteq S_j$ if and only if $remove(C_j, C_l) \subseteq add(C_i, C_j)$.*

Proof. Assume that $remove(C_j, C_l) \subseteq add(C_i, C_j)$. Observe that $S_i \cup add(C_i, C_j) = S_j \cup remove(C_j, C_l) = C_j$, since $S_i = C_i \cap C_j$ and $S_j = C_j \cap C_l$. Remember that *add* and *remove* lists only contain the new vertices, and thus $S_i \cap add(C_i, C_j) = S_j \cap remove(C_j, C_l) = \emptyset$. We can now conclude that $S_i \subseteq S_j$, since $remove(C_j, C_l) \subseteq add(C_i, C_j)$ and $S_i \cup add(C_i, C_j) = S_j \cup remove(C_j, C_l)$ and $S_i \cap add(C_i, C_j) = S_j \cap remove(C_j, C_l) = \emptyset$. For the other direction, assume that $S_i \subseteq S_j$. By the same arguments as in the opposite direction it follows that $remove(C_j, C_l) \subseteq add(C_i, C_j)$, since $S_i \cup add(C_i, C_j) = S_j \cup remove(C_j, C_l)$ and $S_i \cap add(C_i, C_j) = S_j \cap remove(C_j, C_l) = \emptyset$. ■

Claim 5.4 *Let $S_{i,i+1}$ and $S_{j,j+1}$ be tree edges on the path $P_{u,v}$ in the clique tree T, such that $S_{i,i+1} \subseteq S_{j,j+1}$ and $i < j$. Let T' be a clique tree obtained from T, by deleting the tree edge $S_{i,i+1}$ and inserting the tree edge $S_{i,j+1}$. Then T' is a clique tree of the chordal graph H represented by T.*

Proof. The maximal cliques in the clique tree T, are untouched by this operation, so there exists a maximal clique in T' containing the vertex pair u, v if and only if $uv \in E(H)$, and every vertex of H is contained in some maximal clique of T'. Since $S_{i,i+1} \subset C_i$ and $S_{i,i+1} \subseteq S_{j,j+1} \subset C_{j+1}$ then $S_{i,j+1} = C_i \cap C_{j+1} = S_{i,i+1}$, thus it follows that the set of maximal cliques containing a vertex of H induces a connected tree in T' since this is true for T. ■

Given two tree edges $S_{i,i+1}$ and $S_{j,j+1}$ of $P_{u,v}$ then Claim 5.4 can be used to reduce the length of the path $P_{u,v}$ Thus, after this modification, the subpath of $P_{u,v}$ between C_i and C_{j+1} is reduced to $C_i, S_{i,j+1}, C_{j+1}$. This situation corresponds to the change from (b) to (c) in Figure 2. The new *add* and *remove* lists for the tree edge $S_{i,j+1}$ can be computed in the following way:

$$add(C_i, C_{j+1}) = \bigcup_{i \leq q < j+1} add(C_q, C_{q+1}) \setminus \bigcup_{i < q < j+1} remove(C_q, C_{q+1}) \qquad (1)$$

$$remove(C_i, C_{j+1}) = \bigcup_{i \leq q < j+1} remove(C_q, C_{q+1}) \setminus \bigcup_{i < q < j+1} add(C_{q-1}, C_q). \qquad (2)$$

The list $add(C_i, C_{j+1})$ can be computed in time $O(|\bigcup_{i\leq q<j+1} add(C_q, C_{q+1})| + |\bigcup_{i<q<j+1} remove(C_q, C_{q+1})|)$ in the following way. Let A be a characteristic vector of size n, where every element is 0. For every vertex $u \in \bigcup_{i\leq q<j+1} add(C_q, C_{q+1})$, set $A[u] = 1$, and then for every vertex $u \in \bigcup_{i<q<j+1} remove(C_q, C_{q+1})$, set $A[u] = 0$. Now $add(C_i, C_{j+1})$ can be computed as follows. For every vertex $u \in \bigcup_{i\leq q<j+1} add(C_q, C_{q+1})$ where $A[u] = 1$, add u to $add(C_i, C_{j+1})$. In order to reuse the vector, we clean up: for every vertex $u \in \bigcup_{i\leq q<j+1} add(C_q, C_{q+1})$, set $A[u] = 0$. The list $remove(C_i, C_{j+1})$ is computed in the same way.

Claim 5.5 *Let $S_{i,i+1}$ and $S_{j,j+1}$ be tree edges on the path $P_{u,v}$ in the clique tree T, such that $i < j$. Then $S_{i,i+1} \subseteq S_{j,j+1}$ if and only if $remove(C_q, C_{q+1}) \subseteq add(C_i, C_q)$ for $i < q \leq j$.*

Proof. Let us first show that $S_{i,i+1} \subseteq S_{j,j+1}$ if $remove(C_q, C_{q+1}) \subseteq add(C_i, C_q)$ for $i < q \leq j$. This is proved by induction on j, $i < j$.for any given i, where Claim 5.3 corresponds to $j = i + 1$, which we will use as the base case. Now for the induction hypothesis, let us assume that $S_{i,i+1} \subseteq S_{j,j+1}$ if $remove(C_q, C_{q+1}) \subseteq add(C_i, C_q)$ for $i < q \leq j$, and let us prove that $S_{i,i+1} \subseteq S_{j+1,j+2}$ if moreover $remove(C_{j+1}, C_{j+2}) \subseteq add(C_i, C_{j+1})$. Since $S_{i,i+1} \subseteq S_{j,j+1}$ then by Claim 5.4, the path from C_i to C_{j+1} can be reduced to $C_i, S_{i,j+1}, C_{j+1}$, and from the proof of Claim 5.4 we know that $S_{i,j+1} = S_{i,i+1}$. Finally it follows by Claim 5.3 that $S_{i,i+1} = S_{i,j+1} \subseteq S_{j+1,j+2}$ since $remove(C_{j+1}, C_{j+2}) \subseteq add(C_i, C_{j+1})$ in the new path from C_i to C_{j+2}.

For the other direction, we want to show that $remove(C_q, C_{q+1}) \subseteq add(C_i, C_q)$ for $i < q \leq j$ if $S_{i,i+1} \subseteq S_{j,j+1}$. Let q such that $i < q \leq j$. From Theorem 2.3 it follows that $S_{i,i+1} \subseteq S_{q-1,q}$ and $S_{i,i+1} \subseteq S_{q,q+1}$. Since $S_{i,i+1} \subseteq S_{q-1,q}$, from Claim 5.4 the tree edge $S_{i,i+1}$ can be replaced with $S_{i,q}$, where $S_{i,i+1} = S_{i,q}$. Since $S_{i,q} = S_{i,i+1} \subseteq S_{q,q+1}$, it follows by Claim 5.3 that $remove(C_q, C_{q+1}) \subseteq add(C_i, C_q)$. ∎

If we do this reduction for every pair S_i, S_{i+1} and S_{i+1}, S_i of edges on the path $P_{u,v}$, then it follows from Claim 5.2 that every tree edge on $P_{u,v}$ is a distinct minimal u, v-separator. Thus, we are done with the part that is illustrated in Figure 2(b)-(d). However, it remains to explain how to examine adjacent tree edges in such a way that the total time bound $O(n)$ is maintained.

The idea is to do this in two scans. One from C_u to C_v, and one from C_v to C_u. The same operation is done for both directions, so we will only explain the scan from C_u to C_v. Consider the tree edges in the order given by $P_{u,v}$. For a given tree edge S_i (starting from S_1) we will try to find the largest number t such that the intersection $S_{i,i+t} = C_i \cap C_{i+t}$ is equal to S_i. Replace S_i by $S_{i,i+t}$ and continue by finding the next t using S_{i+t} as the new S_i, and repeat this until $C_{i+t} = C_v$.

A consequence of Theorem 2.3 is that there exists no $q > 0$ such that $S_i = S_{i,i+t} \subseteq S_{i+t+q}$, since $S_{i,i+t} \not\subseteq S_{i+t+1}$, which is the property we want for all tree edges on $P_{u,v}$ in both directions.

Computing the $\mathit{add}(C_i, C_{i+t})$ and $\mathit{remove}(C_i, C_{i+t})$ lists can be done as previously described by only reading the *add* and *remove* lists on the path $P_{u,v}$ between C_i and C_{i+t}. The next search starts from C_{i+t}, and thus the total time used to compute all such lists are $O(n)$, since every *add* and *remove* list on the path $P_{u,v}$ is only used to compute the difflists for one new tree edge, and the new tree edges are never used to create other tree edges.

It remains to efficiently compute the value t, given a path $P_{u,v}$ and a tree edge $S_i = S_{i,i+1}$. The basic idea is as follows. Start with $t = 1$. While $S_{i,i+1} \subseteq S_{i,i+t+1}$, increment t and repeat the test until $S_{i+1} \not\subseteq S_{i,i+t+1}$ or $i + t = k$. From Claim 5.5 we know that this is equivalent to testing whether $\mathit{remove}(C_{i+q}, C_{i+q+1}) \subseteq \mathit{add}(C_i, C_{i+q})$, for $1 \leq q \leq t$. It is important to notice that if $S_{i+1} \subseteq S_{i,i+t}$, then we can verify if $S_{i+1} \subseteq S_{i,i+t+1}$ by testing if $\mathit{remove}(C_{i+t}, C_{i+t+1}) \subseteq \mathit{add}(C_i, C_{i+t})$, since we already know that $\mathit{remove}(C_{i+q}, C_{i+q+1}) \subseteq \mathit{add}(C_i, C_{i+q})$, for $1 \leq q < t$. The $\mathit{remove}(C_{i+t}, C_{i+t+1})$ is only read once for each time we increment t, and notice that $\mathit{remove}(C_i, C_{i+1})$ is not used in this test. Let us now argue that we do not read any of the *remove* lists over again when we continue to find the next t. Every remove list we read is removed from the path $P_{u,v}$, except $\mathit{remove}(C_{i+t}, C_{i+t+1})$ which we used to decide that $S_{i,i+1} \not\subseteq S_{i,i+t+1}$. The $\mathit{remove}(C_{i+t}, C_{i+t+1})$ list becomes the new $\mathit{remove}(C_i, C_{i+1})$ list since we use C_{i+t} as the new C_i when we search for the next t. The newly created $\mathit{remove}(C_i, C_{i+t})$ will not be used to find the next t since we use C_{i+t} as C_i. Thus it follows that every remove list on the path $P_{u,v}$ is only used once for testing.

So it remains to explain how the list $\mathit{add}(C_i, C_{i+t})$ is computed and checked against $\mathit{remove}(C_{i+t}, C_{i+t+1})$ list within the time bound. This is done by using a characteristic vector A of size n as an additional data structure. We will manipulate the vector A, such that $A[u] = 1$ if and only if $u \in \mathit{add}(C_i, C_{i+t})$, and then we check in $O(|\mathit{remove}(C_{i+t}, C_{i+t+1})|)$ time if $\mathit{remove}(C_{i+t}, C_{i+t+1}) \subseteq \mathit{add}(C_i, C_{i+t})$. These checks can be done within the time bound, given that the vector A contains the $\mathit{add}(C_i, C_{i+t-1})$ list and that these add lists are provided in an increasing order for the parameter t.

Equation 1 can be rewritten in the following way: $\mathit{add}(C_i, C_{i+t}) = \mathit{add}(C_i, C_{i+t-1}) \cup \mathit{add}(C_{i+t-1}, C_{i+t}) \setminus \mathit{remove}(C_{i+t-1}, C_{i+t})$. This enables us to obtain $\mathit{add}(C_i, C_{i+t})$ by setting $A[u] = 1$ for every vertex $u \in \mathit{add}(C_{i+t-1}, C_{i+t})$, and setting $A[u] = 0$ for every vertex $u \in \mathit{remove}(C_{i+t-1}, C_{i+t})$, given that A contains $\mathit{add}(C_i, C_{i+t-1})$. Notice that $\mathit{add}(C_{i+t-1}, C_{i+t})$ and $\mathit{remove}(C_{i+t-1}, C_{i+t})$ are not used to compute $\mathit{add}(C_i, C_{i+t-1})$. Therefore, by setting $A[u] = 1$ if $u \in \mathit{add}(C_i, C_{i+1})$ when $t = 1$, then the sequence of lists $\mathit{add}(C_i, C_{i+q})$, for $q \leq k + 1$, can be created in A in increasing order by only reading the *add* and *remove* lists on the path between C_i and C_{i+t+1} once. We have to ensure that every element of A is 0 before we start to compute the next t. This is done within the time bound by reading the *add* lists between C_i and C_{i+t+1} once more, and setting $A[u] = 0$ for every vertex u contained in one of these *add* lists.

We have now argued that every operation required to reduce the path $P_{u,v}$ so that every tree edge in $P_{u,v}$ is a distinct minimal u, v-separator, can be executed by only reading each *add* and *remove* list of $P_{u,v}$ a constant number of times. Since every vertex only can appear once in a *add* list of $P_{u,v}$ and once in a *remove* list, it follows that the reduction of $P_{u,v}$ is an $O(n)$ operation.

Now we will see how to compute X. A pair ux belongs to $R(H, u, v)$ if there exists a minimal u, v-separator containing x. Our goal is to compute the set X of vertices, where $x \in X$ if $ux \in R(H, u, v)$. Observe that $C_1 \subseteq N(u)$, and thus, only the vertices not in S_1 are of interest. The path $P_{u,v}$ is already modified such that every tree edge is a minimal u, v-separator, and every minimal u, v-separator is a tree edge in this path. We can compute X in the following way: start in C_u with an empty vertex set X. Then for $1 \leq i < k-1$ add the vertices contained in $\mathit{add}(C_i, C_{i+1}) \setminus \mathit{remove}(C_{i+1}, C_{i+2})$ to X. There might be vertices that are only contained in a single maximal clique C_i, and thus not contained in any tree edge. These vertices will be contained in both $\mathit{add}(C_i, C_{i+1})$ and $\mathit{remove}(C_{i+1}, C_{i+2})$. We obtain the desired set X using a characteristic vector A. For each vertex $u \in \mathit{remove}(C_{i+1}, C_{i+2})$ set $A[u] = 1$, then for each vertex $u \in \mathit{add}(C_i, C_{i+1})$ where $A[u] = 0$ add u to X. Finally for the clean up we set $A[u] = 0$ for each vertex $u \in \mathit{remove}(C_{i+1}, C_{i+2})$. Thus, we obtain the desired set X in $O(n)$ time since the *add* and *remove* lists are read a constant number of times and the total sum of these lists on the path $P_{u,v}$ is $O(n)$.

5.2 Modifying T to reflect the addition of uv and $R(H, u, v)$ to H

Let us now discuss how to the clique tree T of H is built and updated as we decide to add edges and vertices to H. When a new vertex u is added to the set U, then H gets a new vertex, and we update T by adding a new maximal clique containing u.

Let H' denote the graph that results from adding uv and $R(H, u, v)$ to H. We will modify T to obtain a clique tree T' of H'. If u and v are not contained in the same connected component of H and T, then we update T in the following way. Find a tree node K_v of T containing v, and a tree node K_u of T containing u. If $|K_v| > 1$ and $|K_u| > 1$ then we create a new tree node K_{uv} containing the vertices $\{u, v\}$, and insert the tree edges $K_v K_{uv}$ and $K_u K_{uv}$. The add and remove lists for $K_v K_{uv}$ and $K_{uv} K_u$ can be computed straightforwardly in $O(n)$ time. If $|K_v| = 1$ or $|K_u| = 1$, let us say $|K_u| = 1$, then K_u has no neighbor in T. The new tree T' is created by adding vertex v to K_u to obtain tree node K_{uv} and either deleting K_v (if $|K_v| = 1$) or inserting the tree edge $K_v K_{uv}$ (otherwise). Adding vertex v to K_u (resp. deleting K_v) is an $O(n)$ operation since K_u (resp. K_v) has no neighbors, and inserting the tree edge $K_v K_{uv}$ takes $O(n)$ time.

Let us assume that u and v are contained in the same connected component of T and H for the rest of this subsection. In order to update T to reflect that u has now become a neighbor of v and of every vertex in X, we simply place u in every tree edge and every tree node appearing on $P_{u,v}$ in T. This is illustrated in Figure 2(e). However, we must check the resulting tree T' after doing so, because there might be a tree node C on this path containing a vertex q not appearing in any minimal u,v-separator, and in this case u was not supposed to be a neighbor of q. Detecting such a tree node C is easy because then q cannot appear in any other tree node of the path, since X is already computed and $q \notin X$. For any such C, we remove u from C, and we introduce a new tree node C' that contains u and every vertex of C except the vertices that do not appear in any other tree node of $P_{u,v}$. Tree edges incident to C on $P_{u,v}$ are redirected to be incident to C' instead, and tree edge $C'C$ is added to give a clique tree T' that reflects the neighborhood relations of H' correctly. This is illustrated in Figure 2(f), where C_4 corresponds to the mentioned C. If C_u has become a subset of another maximal clique because of this operation, then we must correct T' accordingly. This is shown in Figure 2(g).

Let us now discuss the practical implementation of this $O(n)$ time. First remove the vertex u from the $remove(C_1, C_2)$ list. This ensures that u belongs to every maximal clique (tree node) on $P_{u,v}$. Let us now consider each maximal clique C_i, $2 \leq i \leq k$, in the order given by $P_{u,v}$. The first step is to decide if C_i contains any vertex q as described above. We know that no such vertex q appears in a tree edge of $P_{u,v}$, and that X is the union of the tree edges in $P_{u,v}$. Thus, $Q = add(C_{i-1}, C_i) \setminus (X \cup \{v\})$ is exactly the set of such vertices q that are only contained in C_i. If $Q = \emptyset$, then we add u to C_i. This is done by adding u to every $add(C_l, C_i)$ list, where $l \notin \{i-1, i+1\}$ and C_l is a neighbor of C_i outside of $P_{u,v}$. The value of i can now be incremented, such that the process can continue from the next tree node. In the case where $Q \neq \emptyset$, we have to create a new tree node $C_{i'} = C_i \cap (X \cup \{v\}) \cup \{u\}$, and a new tree edge $S_{i',i}$ between $C_{i'}$ and C_i. This is done by simply creating the new lists $add(C_i, C_{i'})$ and $remove(C_i, C_{i'})$ as follows: $add(C_i, C_{i'}) = \{u\}$, since $C_{i'} \setminus C_i = \{u\}$, and $remove(C_i, C_{i'}) = Q$. The lists $add(C_{i-1}, C_{i'})$, $remove(C_{i-1}, C_{i'})$, $add(C_{i'}, C_{i+1})$ and $remove(C_{i'}, C_{i+1})$ are not created, but obtained by altering $add(C_{i-1}, C_i)$, $remove(C_{i-1}, C_i)$, $add(C_i, C_{i+1})$ and $remove(C_i, C_{i+1})$. This is done by moving the pointers from C_i to $C_{i'}$, and removing all the vertices in Q from these lists.

Let us show that the $O(n)$ time bound is kept during the modifications explained above. The vertex set Q is computed by storing X in a characteristic vector A of size n, such that $A[u] = 1$ if and only if $u \in X$. A vertex u is contained in Q if $u \in add(C_{i-1}, C_i)$ and $A[u] = 0$, thus it follows that Q can be computed in $O(|add(C_{i-1}, C_i)|)$ time.

Creating each new tree node $C_{i'}$ is a constant time operation. Every time a new $C_{i'}$ is created, we also create a new tree edge $S_{i,i'}$. We first argue that the sum of the sizes of all the $add(C_i, C_{i'})$ and $remove(C_i, C_{i'})$ lists for all such new tree edges is $O(n)$. This immediately follows from the fact that $Q \subseteq add(C_{i-1}, C_i)$, $add(C_{i-1}, C_i) \cap add(C_{j-1}, C_j) = \emptyset$ for $1 \leq i, j < k$ $i \neq j$, and $\sum_{1 \leq i < k} |add(C_{i-1}, C_i)| \leq n$. Thus, the total cost of creating all such new tree edges $S_{i,i'}$ is $O(n)$. In order to move the tree edges $S_{i-1,i}$ and $S_{i,i+1}$ to $S_{i-1,i'}$ and $S_{i',i+1}$ we must change some pointers, and read through the lists to remove vertices in $C_i \setminus C_{i'} = Q$. The total cost of all such operations is less or equal to the sum of all *add* and *remove* lists in $P_{u,v}$, given that Q also is stored in a characteristic vector. It follows that this altogether is an $O(n)$ time operation.

We will now, through the next three claims, prove that tree T' that results from the modifications explained above is a clique tree of $H' = (U, D \cup \{uv\} \cup R(H, u, v))$.

Claim 5.6 *Given a chordal graph $H = (V, D)$, a clique tree T of H, an edge uv, and the required set of edges $R(H, u, v)$, let H' be the graph $(V, D \cup \{uv\} \cup R(H, u, v))$ and let T' be the resulting clique tree after updating T as explained above. Then for each pair of vertices x and y, there is a tree node in T' that contains both x and y if and only if $xy \in D \cup \{uv\} \cup R(H, u, v)$.*

Proof. Before any modifications to T at this step, there is a tree node $C \in T$, that contains both the vertices x and y if and only if $xy \in D$. A tree node C_d is only deleted during the modification process if there exists a remaining tree node C'_d, such that $C_d \subseteq C'_d$. Thus, for every edge $xy \in D$ there exists a tree node in T' that contains both x and y.

Before appropriate tree nodes of T are expanded to contain u, every newly created tree node $C_{i'}$ is a subset of some other tree node C_i. At this point we have the property that the vertex set of every tree node of T is either a maximal clique in H or a subset of a maximal clique in H. Thus T has still the property that there exists a tree node containing x and y if and only if $xy \in D$.

Then u is added to every tree node C of T on the modified path $P_{u,v}$, where $C \subseteq X \cup \{v\}$, and we obtain T'. It follows that for every edge $xy \notin E(H')$, there is no tree node C of T' that contains both x and y, since u is only added to a tree node C if $C \subseteq X \cup \{v\}$.

For the other direction we have to show that for every edge $ux \in R(H, u, v) \cup \{uv\}$ there exists a tree node C in T' containing u and x. By Claim 5.1 every minimal u, v-separator is an edge of $P_{u,v}$, thus there exists a tree node C in T on $P_{u,v}$ containing x for every $ux \in R(H, u, v)$. The tree node C_v in the end of $P_{u,v}$ contains v. If $C \not\subseteq X \cup \{v\}$ for a tree node C in T on $P_{u,v}$, then a new tree node $C' = C \cap (X \cup \{v\})$ is created and used in the path $P_{u,v}$. We can now conclude that for every edge $ux \in R(H, u, v) \cup \{uv\}$ there exists a tree node of T' which contains both u and x. ∎

Claim 5.7 *Subtree T'_x induced by the tree nodes in T' that contain vertex x is connected, for every vertex $x \in U$.*

Proof. We assume that all subtrees are connected in T before the last modification. Let us now consider the operations one by one. First operation is when $C_i \notin (X \cup \{v\})$. A new tree node $C_{i'}$ is created, where $C_{i'} \subset C_i$. A tree edge is inserted between C_i and $C_{i'}$, but all subtrees are connected since $C_{i'} \subset C_i$. Next step is to move the edges $S_{i-1,i}$ and $S_{i,i+1}$ to become $S_{i-1,i'}$ and $S_{i',i+1}$. This will not create separated subtrees since $S_{i-1,i} \cup S_{i,i+1} \subseteq C_{i'}$, thus $S_{i-1,i'} = S_{i-1,i}$ and $S_{i',i+1} = S_{i,i+1}$. The second operation is adding the vertex u to $C_{i'}$ in the case where a new tree node $C_{i'}$ is created, and to C_i if no new tree node is created. This changes only the tree induced by the tree nodes containing the vertex u. Since we consider the tree nodes in the order C_2 to C_v, then it follows that the tree T_u is always connected. ■

Claim 5.8 *The tree nodes of T' are exactly the distinct maximal cliques of H', except for C_u in case $C_u \subseteq C_2$.*

Proof. Let us first show that every maximal clique of H' is a tree node of T'. By Claims 5.6 and 5.7, T' defines what is called a tree decomposition of H'. So, by [13] every clique in H' is contained in some node of T'. Since by Claim 5.6 every tree node of T' is a clique in H', a maximal clique in H' cannot be strictly contained in some tree node of T' and therefore is equal to one of them.
Conversely, let us show that tree nodes of T' are distinct maximal cliques of H', except for C_u in case $C_u \subseteq C_2$. Suppose on the contrary that some tree node C of T' is not a maximal clique of H' or is a maximal clique of H' equal to another node of T'. Since every maximal clique of H' is a tree node of T' and since by Claim 5.6, C is a clique of H', there is a tree node C' different from C containing C. Let C'' be the neighbor of C on the path in T' between C and C'. By Claim 5.7, $C = C \cap C' \subseteq C''$. It follows that it is sufficient to show that no tree node of T' is a subset of one of its neighbors, except for C_u in case $C_u \subseteq C_2$.

We will now prove by induction that $C \nsubseteq C''$ unless $C = C_u$ and $C'' = C_2$, where C and C'' are tree nodes of T'. This is clearly true in the base case, where H' consist of only one vertex and T' consist of a single tree node. Let $H = (V, D)$ be the chordal graph such that $H' = (V, D \cup \{uv\} \cup R(H, u, v))$ and let T be the given maximal clique tree of H. There are two cases. The first is when the vertex u is added to a tree node C. The expanded C cannot become a subset of another tree node, but it can become a superset of a tree node C_j, if $C_j \setminus C = \{u\}$. Since C_u is the only tree node in the neighborhood of any tree node different from C_u in $P_{u,v}$ that contains u, then this can only happen to C_u. The second case is when a new tree node $C_{i'}$ is created as a subset of C_i, where $u \notin C_i$ and $C_i \setminus C_{i'} \neq \emptyset$. In this situation $C_{i'}$ is a tree node on the path $P_{u,v}$ and C_i is not, u is added to $C_{i'}$ and not to C_i, thus C_i and $C_{i'}$ are not subsets of each other. From the construction of $C_{i'}$ we know that every neighbor of $C_{i'}$ different from C_i is on the path $P_{u,v}$. Let us now on the contrary assume that $C_{i'} \subseteq C_{i-1}$ or $C_{i'} \subseteq C_{i+1}$. If

$C_{i'} \subseteq C_{i+1}$ then $S_{i-1,i'} \subseteq C_{i'} = S_{i',i+1}$, which is a contradiction to the fact that every edge of $P_{u,v}$ is a unique minimal u,v-separator. If $C_{i'} \subseteq C_{i-1}$ then $i = k$ since otherwise $S_{i',i+1} \subseteq C_{i'} = S_{i-1,i'}$ which is a contradiction to the fact that every edge of $P_{u,v}$ is a unique minimal u,v-separator. The only remaining case is that $C_{i'} = C_v$ on the path $P_{u,v}$ with $C_{i'} \subseteq C_{i-1}$. Then $v \in C_{i'}$, so $v \in C_{i-1} \neq C_v$ which is a contradiction to the fact that only C_v in $P_{u,v}$ contains v. ∎

Let us re-sort the tree nodes of the modified path $P_{u,v}$ of T' from $C_u = C_1$ to $C_k = C_v$. With the above three claims, if $C_u \not\subseteq C_2$, then we have proved that T' is a legal clique tree of H'. If $C_u \subseteq C_2$, then we will simply remove C_u, and again we can conclude that T' with this final modification is a legal clique tree of H'.

However, it remains to explain how this final update of removing $C_u = C_1$ can be done in $O(n)$ time, which is challenging. It is easy to check if $C_1 \subseteq C_2$, since $remove(C_1, C_2) = \emptyset$ in this case. Tree node C_1 is deleted in the following way: For every tree edge $S_{1,j}$ where $C_j \neq C_2$, we delete the tree edge $S_{1,j}$ and insert $S_{2,j}$. Afterwards we delete tree edge $S_{1,2}$ and tree node C_1. In order to do this efficiently we actually alter the *add* and *remove* lists and move the tree edges from C_1 to C_2. Let us consider the new tree node C_j, and how to create the *add* and *remove* lists from C_j to C_2. From the previous described technique they can be computed as follows: $add(C_j, C_2) = add(C_j, C_1) \cup add(C_1, C_2) \setminus remove(C_1, C_2)$ and $remove(C_j, C_2) = remove(C_1, C_2) \cup remove(C_j, C_1) \setminus add(C_j, C_1)$. Remember that $remove(C_1, C_2) = \emptyset$, since $C_1 \subseteq C_2$, and that $remove(C_j, C_1) \cap add(C_j, C_1) = \emptyset$. Computing the lists can then be reduced to: $add(C_j, C_2) = add(C_j, C_1) \cup add(C_1, C_2)$ and $remove(C_j, C_2) = remove(C_j, C_1)$. The obstacle regarding the time complexity is that $add(C_1, C_2)$ will be read once for each neighbor of C_1. Thus, we have to ensure that this work does not sum up to more than $O(n)$. Let us count the number of times this can happen.

Claim 5.9 *Let $C_u \subseteq C_2$ in T', and let H'' be the chordal graph right before the first edge uv' incident to u and the set $R(H'', u, v')$ was added to H'', and let T'' be the clique tree of H''. Then $C_u \setminus \{u\}$ is not a tree node of T'' or in other words $C_u \setminus \{u\}$ is not a maximal clique of H''.*

Proof. The tree T' is obtained from T'' by adding new tree nodes which are subsets of tree nodes in T'' and by adding u to tree nodes of this new tree, since only edges incident to u are processed between T'' and T'. Clearly $C_u \setminus \{u\}$ is not a tree node in T'', since $C_u \setminus \{u\} \subseteq C_2 \setminus \{u\}$ and $C_2 \setminus \{u\}$ is contained in some tree node of T'', and every tree node of T'' is a unique maximal clique in H''. ∎

Claim 5.10 *Reducing path $P_{u,v}$ such that it contains only distinct minimal u,v-separators, can increase the degree of C_u by at most 1.*

Proof. Two different scans are done on $P_{u,v}$ to reduce the number of tree nodes. The first starts in C_u, and finds the maximal clique furthest from C_u that is a superset of $S_{u,2}$. In this case one tree edge incident to C_u is deleted, and one is created, and the degree of C_u remains the same. In the direction from C_v to C_u, we may find a tree edge that is a subset of $S_{u,2}$. In this case C_u gets a new neighbor and the degree of C_u increases by 1. ∎

Observe that the process of reducing the path $P_{u,v}$ can increase the degree of at most one tree node containing the vertex u. This follows from the fact that C_u is the only tree node in $P_{u,v}$ containing the vertex u.

Claim 5.11 *Adding u to every tree node in $P_{u,v}$ does not increase the degree of C_u.*

Proof. One of two things will happen. In one case vertex u is added to C_2, which is the neighbor of C_u in the path $P_{u,v}$. This will not change the degree of C_u in the clique tree T. The second case is if C_2 is not a subset of $X \cup \{v\}$. Then a new tree node C_2' is created, and the tree edge between C_u and C_2 is removed, and inserted between C_u and C_2'. It follows that the degree of C_u is unchanged. ∎

Claim 5.12 *The degree of each newly created tree node $C_{i'}$ in T' is at most* 3.

Proof. When a new tree node $C_{i'}$ is created, it is a subset of an existing tree node C_i. Let d be the number of neighbors C_i has in $P_{u,v}$. Thus, d is either 1 or 2. A tree edge is introduced between $C_{i'}$ and C_i, and $C_{i'}$ replaces C_i in the path $P_{u,v}$. The degree of $C_{i'}$ becomes $d+1$, and thus the degree is at most 3. ∎

Remember that the obstacle in obtaining the $O(n)$ time bound was that $add(C_u, C_2)$ is read once for each neighbor of C_u, when $add(C_j, C_2) = add(C_j, C_u) \cup add(C_u, C_2)$ is computed. It remains to show that if $C_u \subseteq C_2$ in T' then $\sum_{C_jC_u \textit{ edge of } T',\, j\neq 2} |add(C_u, C_2)|$ is $O(n)$. We will use an amortized time analysis. Let vertex u be given, and let $d(u)$ denote the degree of u in G. For any neighbor v of u in G such that edge uv is processed in the execution of the algorithm, let $T'(v)$ denote the tree T' when processing edge uv, let $C_u(v)$ denote tree node C_u of this tree, and let $C_j(v)$ denote tree node C_j of $T'(v)$. Let V_1 be the set of neighbors v of u in G such that edge uv is processed and $C_u(v) \subseteq C_2(v)$. Let $S = \sum_{v\in V_1} \sum_{C_j(v)C_u(v) \textit{ edge of } T'(v)} |add(C_u(v), C_2(v))|$. It is sufficient to show that S is $O(n \cdot d(u))$. For any $v \in V_1$, let $E(v)$ be the set of edges incident to $C_u(v)$ in $T'(v)$, and let $E_1(v)$, $E_2(v)$, $E_3(v)$, and $E_4(v)$ be the following subsets of $E(v)$:

- $E_1(v)$ is the set of edges of $E(v)$ created at the same time as $C_u(v)$;
- $E_2(v)$ is the set of edges of $E(v)$ inserted when reducing path $P_{u,v'}$, with $C_u(v') = C_u(v)$, for some edge uv' processed before uv;

- $E_3(v)$ is the set of edges of $E(v)$ inserted when suppressing $C_u(v')$ because $C_u(v') \subseteq C_2(v')$, with $C_2(v') = C_u(v)$, for some edge uv' processed before uv;

- $E_4(v)$ is the set of edges of $E(v)$ inserted as an edge $K_uK_{uv'}$ or $K_uK_{v'}$ with $K_u = C_u(v)$, when previously processing an edge uv' such that u and v' are in different connected components of the current graph.

It follows from Claim 5.9 that $C_u(v)$ is created when inserting some edge uv' previous to uv. The set $E_2(v)$, contains the edge incident to $C_u(v')$ that may be inserted when reducing path $P_{u,v'}$ during the scan from $C_{v'}(v')$ to $C_u(v')$, but not the edge that may be inserted during the scan from $C_u(v')$ to $C_{v'}(v')$ in replacement of edge $C_u(v')C_2(v')$: these two edges (the replaced and replacing ones) are identified in our counting process. In the same way, the edge of $E(v)$ that may be inserted in replacement of $C_u(v')C_2(v')$ when adding u to $C_2(v')$ is identified with the replaced edge. So by Claim 5.11, no edge of $E(v)$ has been inserted when adding u to $C_2(v')$ for any edge uv' processed before uv, and every edge of $E(v)$ belongs to one of the sets $E_1(v)$, $E_2(v)$, $E_3(v)$, and $E_4(v)$.

Let $e = C_j(v)C_u(v)$ be an edge of $E_3(v)$, and let v' be the previously processed vertex such that e was inserted when suppressing $C_u(v')$ because $C_u(v') \subseteq C_2(v')$, with $C_2(v') = C_u(v)$. Then $v' \in V_1$ and e is obtained from some edge $e' = C_{j'}(v')C_u(v')$ belonging to $E_1(v') \cup E_2(v') \cup E_3(v') \cup E_4(v')$. We say that e *derives from* e'. If $e' \in E_3(v')$ then e' derives from some edge e''. It follows that there are sequences $(v_0, v_1, ..., v_p = v)$ of vertices of V_1 and $(e_0, e_1, ..., e_p = e)$ of edges such that $e_0 \in E_1(v_0) \cup E_2(v_0) \cup E_4(v_0)$ and for any i from 1 to p, $e_i \in E_3(v_i)$ and e_i derives from e_{i-1}. Conversely, for any $v \in V_1$ and $e \in E_1(v) \cup E_2(v) \cup E_4(v)$, there are unique such sequences $seq(e) = (v = v_0, v_1, ..., v_p)$ and $(e = e_0, e_1, ..., e_p)$ such that no edge derives from e_p. So S can be rewritten as follows: $S = \sum_{v \in V_1} \sum_{e \in E_1(v) \cup E_2(v) \cup E_4(v)} \sum_{v' \in seq(e)} |add(C_u(v'), C_2(v'))|$.

If $seq(e) = (v = v_0, v_1, ..., v_p)$, then since $C_u(v_i) \subseteq C_2(v_i)$ for any i from 0 to p and $C_2(v_i) = C_u(v_{i+1})$ for any i from 0 to $p-1$, we have that $\sum_{v' \in seq(e)} |add(C_u(v'), C_2(v'))| = \sum_{0 \leq i < p}(|C_u(v_{i+1})| - |C_u(v_i)|) + (|C_2(v_p)| - |C_u(v_p)|) = |C_2(v_p)| - |C_u(v_0)| \leq n$.

Moreover, for any $v \in V_1$, if $C_u(v)$ was created as a node $\{u, v'\}$ (when processing an edge uv' such that u and v' are in different connected components of the current graph) then $|E_1(v)| \leq 2$, and otherwise, by Claims 5.9 and 5.12, $|E_1(v)| \leq 3$, and by Claim 5.10, $\sum_{v \in V_1} |E_2(v)| \leq d(u)$, and finally $\sum_{v \in V_1} |E_4(v)| \leq d(u)$ since only one tree edge is added for each edge uv' such that u and v' are contained in different connected components of T'. Hence $S \leq n(\sum_{v \in V_1}(|E_1(v)| + |E_2(v)| + |E_4(v)|) \leq n(3|V_1| + 2d(u)) \leq n \cdot 5d(u) = O(n \cdot d(u))$.

6 Concluding remarks

In this paper, we contribute new theoretical results on chordality as well as an efficient handling of the corresponding data structures. Not only do we have a new $O(nm)$ time on-line algorithm for minimal triangulation of a graph G, but we are able to compute at the same time a maximal chordal subgraph, thus "minimally sandwiching" the graph between two chordal graphs: $H_1 \subseteq G \subseteq H_2$.

This special feature of our algorithm enables the user, at no extra cost, to choose at each vertex addition step whether he wants to *add* or *delete* edges, or even to do so at each edge addition step. This may be interesting for applications such as updating databases or for sampling techniques in the context of artificial intelligence when maintaining a chordal graph is required or desirable.

Recent work has shown that minimal separation plays an important role in the process of minimal triangulation. Our new characterization of chordal graphs, which uses minimal separation, leads us to believe that there is a corresponding relationship when computing a maximal chordal subgraph.

A continuation of this work would be to compare the running time of our algorithm to other minimal triangulation algorithms with experimental tests. Since often several edges are found and inserted at the time cost of one edge, we conjecture that our algorithm may be very fast in practice. Another important issue to inquire about would be how well our algorithm performs when used as a heuristic for hard problems, such as computing a minimum triangulation or a maximum subtriangulation. Standard ideas from existing heuristics, like picking a vertex of minimum degree at each step, could be integrated into our algorithm and possibly result in higher probability of less fill in minimal triangulations and more edges in maximal subtriangulations.

It appears that chordal graphs are in many ways similar to weakly chordal graphs [23, 8, 7]. It would be interesting to extend our results to define a process which maintains a weakly chordal graph, thus enabling efficient computation of a weak minimal super or maximal sub triangulation, which is an important issue for recent applications to formal concept analysis and data mining [9]. As we pointed out in Section 3, the required set of edges can be seen as a succession of 2-pairs which is computed efficiently. In view of the important role that 2-pairs play in weakly chordal graph recognition [24, 33, 25], our results could possibly be extended to efficiently handle such a succession of 2-pairs in a weakly chordal graph, with the hope of improving the current $O(m^2)$ [25] time complexity for this problem.

Acknowledgments

The authors would like to thank the referees for useful comments, which have improved the readability of the text in general, and of some of the proofs in

particular.

References

[1] S. Arikati and P. Rangan. An efficient algorithm for finding a two-pair, and its applications. *Disc. Appl. Math.*, 31:71–74, 1991.

[2] E. Balas. A fast algorithm for finding an edge-maximal subgraph with a TR-formative coloring. *Disc. Appl. Math.*, 15:123–134, 1986.

[3] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database systems. *J. Assoc. Comput. Mach.*, 30:479–513, 1983.

[4] A. Berry. A wide-range efficient algorithm for minimal triangulation. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[5] A. Berry, J. Blair, P. Heggernes, and B Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287–298, 2004.

[6] A. Berry, J. Bordat, P. Heggernes, G. Simonet, and Y. Villanger. A wide-range algorithm for minimal triangulation from an arbitrary ordering. *Journal of Algorithms*, 58(1):33–66, 2006.

[7] A. Berry and J-P. Bordat. Triangulated and weakly triangulated graphs: Simpliciality in vertices and edges. *6th International Conference on Graph Theory (ICGT 2000)*, 2000. Communication.

[8] A. Berry, J-P. Bordat, and P. Heggernes. Recognizing weakly triangulated graphs by edge separability. *Nordic Journal of Computing*, 7:164–177, 2000.

[9] A. Berry and A. Sigayret. Obtaining and maintaining polynomial-size concept lattices. In *Proceedings of FCAKDD, (ECAI 2002)*, pages 3–6, 2002.

[10] A. Berry, A. Sigayret, and C. Sinoquet. Maximal sub-triangulation as improving phylogenetic data. Technical Report RR-02-02, LIMOS, Clermont-Ferrand, France, 2002.

[11] J. R. S. Blair, P. Heggernes, and J. A. Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 250:125–141, 2001.

[12] J. R. S. Blair and B. W. Peyton. An introduction to chordal graphs and clique trees. In J. A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 1–30. Springer Verlag, 1993. IMA Volumes in Mathematics and its Applications, Vol. 56.

[13] Hans L. Bodlaender and Rolf H. Möhring. The pathwidth and treewidth of cographs. *SIAM J. Discrete Math.*, 6(2):181–188, 1993.

[14] P. Buneman. A characterization of rigid circuit graphs. *Discrete Math.*, 9:205–212, 1974.

[15] T. F. Coleman. A chordal preconditioner for large-scale optimization. *Applied Math.*, 40:265–287, 1988.

[16] E. Dahlhaus. Minimal elimination ordering inside a given chordal graph. In R. H. Möhring, editor, *Graph Theoretical Concepts in Computer Science - WG '97, LNCS 1335*, pages 132–143. Springer Verlag, 1997.

[17] P. M. Dearing, D. R. Shier, and D. D. Warner. Maximal chordal subgraphs. *Disc. Appl. Math.*, 20:181–190, 1988.

[18] A. Deshpande, M. Garofalakis, and M. I. Jordan. Efficient stepwise selection in decomposable models. In *Proceedings of UAI*, pages 128–135, 2001.

[19] G. A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg*, 25:71–76, 1961.

[20] P. Erdös and R. Laskar. On maximum chordal subgraph. *Cong. Numerantium*, 39:367–373, 1983.

[21] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15:835–855, 1965.

[22] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Combin. Theory Ser. B*, 16:47–56, 1974.

[23] R. Hayward. Generating weakly triangulated graphs. *J. Graph Theory*, 21:67–70, 1996.

[24] R. Hayward, C. Hoàng, and F. Maffray. Optimizing weakly triangulated graphs. *Graphs and Combinatorics*, 5:339–349, 1989.

[25] R. Hayward, J. Spinrad, and R. Sritharan. Weakly chordal graph algorithms via handles. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[26] P. Heggernes and Y. Villanger. Efficient implementation of a minimal triangulation algorithm. In R. H. Möhring, editor, *Algorithms - ESA 2002, LNCS 2461*, pages 550–561. Springer Verlag, 2002.

[27] C-W. Ho and R. C. T. Lee. Counting clique trees and computing perfect elimination schemes in parallel. *Inform. Process. Lett.*, 31:61–68, 1989.

[28] L. Ibarra. Fully dynamic algorithms for chordal graphs. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[29] M. Lundquist. *Zero patterns chordal graphs and matrix completions.* PhD thesis, Clemson University, USA, 1990.

[30] A. Natanzon, R. Shamir, and R. Sharan. Complexity classification of some edge modification problems. *Disc. Appl. Math.*, 113:109–128, 2001.

[31] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

[32] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.

[33] J. Spinrad and R. Sritharan. Algorithms for weakly triangulated graphs. *Disc. Appl. Math.*, 59:181–191, 1995.

[34] J. Walter. *Representations of rigid cycle graphs.* PhD thesis, Wayne State University, USA, 1972.

[35] J. Xue. Edge-maximal triangulated subgraphs and heuristics for the maximum clique problem. *Networks*, 24:109–120, 1994.

[36] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.

Paper III

Lex M versus MCS-M

Yngve Villanger*

Abstract

We study the problem of minimal triangulation of graphs. One of the first algorithms to solve this problem was Lex M, which was presented in 1976. A new algorithm, and a simplification of Lex M called MCS-M, was presented in 2002. In this paper we compare these two algorithms and show that they produce the same set of triangulations, answering an open question mentioned by the authors of MCS-M.

1 Introduction

Graph theory has several important problems that involve creating a chordal supergraph from a given graph by adding a set of edges. The set of added edges is called *fill*, and the chordal supergraph is called a *triangulation* of the given graph. Different goals may be desired; one is to introduce as few new edges as possible (called *minimum fill*), and another is to create a triangulation such that the largest clique is as small as possible, which corresponds to the *treewidth* of the graph. Both of these problems are NP-hard [1, 13].

Minimal fill, also called *minimal triangulation*, is the problem of adding an inclusion minimal set of fill edges. There exist several practical algorithms that solve this problem [2, 4, 5, 6, 7, 9, 11, 12]. Since minimum fill is hard to compute, minimal fill may be used as an alternative, even though the difference in the number of fill edges may be quite large. One of the algorithms that solve the minimal triangulation problem is Lex M (Rose, Tarjan, and Lueker [12]), which is a classical algorithm based on a special breadth first search and lexicographic labeling of the vertices. Recently (Berry, Blair, Heggernes, and Peyton [3]) introduced a new algorithm called MCS-M. This is a simplification of Lex M so that cardinality weights are used instead of lexicographic labels.

A triangulation of a graph can also be obtained by using the *elimination game* [10] algorithm. This algorithm takes a graph and an ordering of the vertices as input. The ordering of the vertices given to the elimination game is also called

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway. yngvev@ii.uib.no

an *elimination ordering.* This ordering uniquely defines the set of fill edges for a given graph, but there may be many different elimination orderings that introduce the same set of fill edges. If an ordering produces a minimal triangulation, then the ordering is called a *minimal elimination ordering* (meo).

Both Lex M and MCS-M produce an meo. The user of Lex M and MCS-M can select the last vertex in the ordering, and may have some choices during the execution of the algorithm. Because of these choices, both algorithms produce a set of minimal orderings for a given graph. Some of the orderings may only occur in one of the sets, and it follows that the number of orderings in each of these sets can be quite different. However, in this paper we show that for every Lex M ordering, there exists an MCS-M ordering that creates exactly the same fill edges, and for every MCS-M ordering there exists a Lex M ordering that creates exactly the same fill edges. It follows that Lex M and MCS-M create exactly the same set of triangulations.

2 Elimination orderings, Lex M, and MCS-M

We consider finite, simple, undirected and connected graphs. Given a graph $G = (V, E)$, we denote the number of vertices as $n = |V|$ and the number of edges as $m = |E|$. The *neighborhood* of a vertex $u \in V$ is denoted by $N_G(u) = \{v$ for $(u, v) \in E\}$, and $N_G[u] = N_G(u) \cup \{u\}$. In the same way we define the neighborhood of a set $A \subseteq V$ of vertices by $N_G(A) = \cup_{u \in A} N_G(u) \backslash A$. A sequence $v_1 - v_2 - ... - v_k$ of distinct vertices describes a *path* if (v_i, v_{i+1}) is an edge for $1 \leq i < k$. The *length* of a path is the number of edges in the path. A *cycle* is defined as a path except that it starts and ends with the same vertex. If there is an edge between every pair of vertices in a set $A \subseteq V$, then the set A is called a *clique.*

Chordal graphs are the family of graphs where every cycle of length greater than three has a chord. A *chord* is an edge between two non-consecutive vertices of a cycle. Chordal graphs can be computed from non-chordal graphs by introducing new edges, called *fill* edges. This process is called *triangulation* of a graph. An *ordering* of V is a function $\alpha : \{1, 2, ..., n\} \leftrightarrow V$, and we use $\alpha = [v_1, v_2, ..., v_n]$ to denote that $\alpha(i) = v_i$ for $1 \leq i \leq n$. Given a graph G and an ordering α of the vertices in G, the *elimination game* [10] can be used to obtain a triangulation G^+_α of the given graph G. The triangulation is obtained by picking the first vertex from the ordering, making its neighborhood into a clique, and then removing the vertex from the graph. This is repeated until no vertex remains. The ordering α is called an *elimination ordering.* The vertex at position i is given by $\alpha(i)$, and $\alpha^{-1}(u)$ gives us the position of the vertex u in the ordering. Theorem 2.1 gives a precise description of what edges exist in the resulting graph.

Theorem 2.1 *(Rose, Tarjan, and Lueker [12]) Given a graph $G = (V, E)$ and an elimination ordering α of G, (y, z) is an edge in G^+_α if and only if $(y, z) \in E$ or there exists a path $y, x_1, x_2, ..., x_k, z$ in G where $\alpha^{-1}(x_i) < min\{\alpha^{-1}(y), \alpha^{-1}(z)\}$, for $1 \leq i \leq k$.*

The set of vertices *monotonely adjacent* to a vertex is the set of higher numbered neighbors, and is defined as follows. Given a graph $G = (V, E)$ and an ordering α of the vertices, then $madj_{G^+_\alpha}(z) = \{w \text{ for which } (z, w) \in E(G^+_\alpha), \alpha^{-1}(z) < \alpha^{-1}(w)\}$. Our first result, before we continue with minimal triangulations, concerns changes that can be done to an elimination ordering without altering the resulting triangulation. Our approach is to consider two consecutive vertices in the ordering, and decide if they can switch places in the ordering without altering the triangulation.

Lemma 2.2 *Given a graph $G = (V, E)$, and an ordering $\alpha = [x_1, x_2, ..., x_k, u, v, x_{k+3}, ..., x_n]$ of $V(G)$, let $\beta = [x_1, x_2, ..., x_k, v, u, x_{k+3}, ..., x_n]$ (u and v are swapped). If $(u, v) \notin E(G^+_\alpha)$ then $G^+_\alpha = G^+_\beta$.*

Proof. We want to show that $madj_{G^+_\alpha}(z) = madj_{G^+_\beta}(z)$ for each $z \in V$, since it then follows that $G^+_\alpha = G^+_\beta$. Let z be any vertex in $V \setminus \{u, v\}$. The set of vertices appearing prior to z in α and in β is exactly the same. It follows from Theorem 2.1 that $madj_{G^+_\alpha}(z) = madj_{G^+_\beta}(z)$ for any $z \in V \setminus \{u, v\}$. Let us now consider the vertices u and v. The edge $(u, v) \notin E(G^+_\alpha)$, and due to Theorem 2.1, there exists no path from u to v in G that passes through only vertices from among $x_1, x_2, ..., x_k$. We show that $madj_{G^+_\alpha}(u) = madj_{G^+_\beta}(u)$. In order to do this we will show that both $madj_{G^+_\alpha}(u) \setminus madj_{G^+_\beta}(u)$ and $madj_{G^+_\beta}(u) \setminus madj_{G^+_\alpha}(u)$ are empty sets. Let us first on the contrary assume that there exists a vertex $z \in madj_{G^+_\beta}(u) \setminus madj_{G^+_\alpha}(u)$. Then there must exist a path from u to z in G that passes through only vertices from $x_1, x_2, ..., x_k, v$, and this path must contain v, since there does not exist any path in G between u and z that uses only vertices from $x_1, x_2, ..., x_k$, because $z \notin madj_{G^+_\alpha}(u)$. This gives a contradiction since there exists no path in G from u to v that only uses vertices from among $x_1, x_2, ..., x_k$, and thus no such path between u and z through v can exist. Now let us on the contrary assume that there exists a vertex $z \in madj_{G^+_\alpha}(u) \setminus madj_{G^+_\beta}(u)$. This is a contradiction since there must exist a path from u to z in G that passes through only vertices from $x_1, x_2, ..., x_k$, but no such path that passes through only vertices from $x_1, x_2, ..., x_k, v$. It follows that $madj_{G^+_\alpha}(u) = madj_{G^+_\beta}(u)$. It remains to show that $madj_{G^+_\alpha}(v) = madj_{G^+_\beta}(v)$. The proof is the same as the one for u. Let us first on the contrary assume that there exists a vertex $z \in madj_{G^+_\alpha}(v) \setminus madj_{G^+_\beta}(v)$. Then there must exist a path from v to z in G that passes through only vertices from $x_1, x_2, ..., x_k, u$. This path must contain u because there does not exist any path in G that passes through only vertices from $x_1, x_2, ..., x_k$, since $z \in madj_{G^+_\alpha}(v) \setminus madj_{G^+_\beta}(v)$. This is a contradiction since there does not exist a path from u to v in G, that passes through only vertices from $x_1, x_2, ..., x_k$. Let us on the contrary assume that there exists a vertex $z \in madj_{G^+_\beta}(v) \setminus madj_{G^+_\alpha}(v)$. This is a contradiction because there must exist a path from v to z in G that passes through only vertices from $x_1, x_2, ..., x_k$, but there must not exist any path in G passes through only vertices from $x_1, x_2, ..., x_k, u$, since $z \in madj_{G^+_\beta}(v) \setminus madj_{G^+_\alpha}(v)$. It follows that $madj_{G^+_\alpha}(v) = madj_{G^+_\beta}(v)$. ∎

Lex M computes a minimal elimination ordering given a graph. The elimination order is produced in reverse order, and in some implementations of Lex M, the highest-numbered vertex in the ordering can be selected arbitrarily by the user. Each vertex in Lex M is assigned a label. This label is a sequence of numbers ordered in decreasing order. Let $L(u)$ be the label of vertex u, and let $L_k(u)$ be the number at position k in the sequence $L(u)$. The labels can be compared in the following way: $L(u) = L(v)$ if $|L(u)| = |L(v)|$ and $L_i(u) = L_i(v)$ for $1 \leq i \leq |L(u)|$. Furthermore $L(u) < L(v)$ if $L_k(u) < L_k(v)$, where k is the smallest number such that $L_k(u) \neq L_k(v)$, or $L_i(u) = L_i(v)$ for $1 \leq i \leq |L(u)|$ and $|L(u)| < |L(v)|$.

Algorithm Lex M (Rose, Tarjan, and Lueker [12])
Input: $G = (V, E)$.
Output: A minimal elimination ordering α and G^+_α.

```
$G^+_\alpha = G$;
for all vertices $u$ in $G$
    $L(u) = \emptyset$;
for $i = n$ to 1
    let $v$ be one of the unnumbered vertices with largest label;
    $\alpha^{-1}(v) = i$;
    for each unnumbered vertex $u$ such that there exists a path
    $u = x_0, x_1, ..., x_k = v$ in $G$, where $x_j$ is unnumbered and
    $L(x_j) < L(u)$ for $0 < j < k$
        add $i$ to $L(u)$;
        add fill edge $(v, u)$ to $G^+_\alpha$;
```

Just as Lex M does, MCS-M produces an elimination ordering in reverse order, and like Lex M the highest-numbered vertex in the ordering can be selected arbitrarily by the user in some implementations of MCS-M. MCS-M differs from Lex M by using cardinality weights instead of lexicographic labels. MCS-M basically uses the same approach as Lex M to search the graph.

Algorithm MCS-M (Berry, Blair, Heggernes, and Peyton [3])
Input: $G = (V, E)$.
Output: A minimal elimination ordering α and G^+_α.

```
$G^+_\alpha = G$;
for all vertices $u$ in $G$
    $w(u) = 0$;
for $i = n$ to 1
    let $v$ be one of the unnumbered vertices with largest weight;
    $\alpha^{-1}(v) = i$;
    for each unnumbered vertex $u$ such that there exists a path
    $u = x_0, x_1, ..., x_k = v$ in $G$, where $x_j$ is unnumbered and
    $w(x_j) < w(u)$ for $0 < j < k$
        $w(u) = w(u) + 1$;
        add fill edge $(v, u)$ to $G^+_\alpha$;
```

Both Lex M and MCS-M may provide the user with choices from the set of unnumbered vertices with largest label or weight, respectively. These choices are not necessarily the same for the two algorithms. In Figure 1, there is an example where Lex M and MCS-M do not have the same choices.

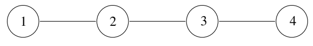

Figure 1: Let 2 be the starting vertex in the given graph. In this situation Lex M is capable of creating the following set of elimination orderings $[\{4, 3, 1, 2\}, \{4, 1, 3, 2\}]$, while MCS-M is capable of creating the following set of orderings $[\{4, 3, 1, 2\}, \{4, 1, 3, 2\}, \{1, 4, 3, 2\}]$. Observe that every one of these orderings is a perfect elimination ordering (peo) [8] for the given graph.

To make it easier to discuss Lex M and MCS-M we give an exact description of the label and weight for each vertex at each step of the algorithm. Let $L_{z-}(x)$ be the label of vertex x in Lex M right before z has been assigned the number $\alpha^{-1}(z)$, and let $L_{z+}(x)$ be the label of x right after z has been assigned the number $\alpha^{-1}(z)$ and Lex M has added this number to the labels described by Lex M. Lemma 2.3 describes how the relationship between labels changes as the algorithm proceeds.

Lemma 2.3 *(Rose, Tarjan, and Lueker [12]) Let $G = (V, E)$ be a graph, and let u, v be vertices of G. If $L_{\alpha(i)-}(v) < L_{\alpha(i)-}(u)$, then $L_{\alpha(j)-}(v) < L_{\alpha(j)-}(u)$ for all $1 \leq j \leq i$.*

For MCS-M we do the same, let $w_{z-}(x)$ be the weight of vertex x in MCS-M right before z has been assigned the number $\alpha^{-1}(z)$, and let $w_{z+}(x)$ be the weight of x right after z has been assigned the number $\alpha^{-1}(z)$ and MCS-M has used z to increase the weight of other vertices as described by MCS-M. Given a set $A \subseteq V$ of vertices, then $hW_{z-}(A)$ is the set of vertices in A with the highest weight assigned by MCS-M right before z has been assigned a number, and $hL_{z-}(A)$ is the set of vertices in A with the largest labels assigned by Lex M right before z has been assigned a number.

3 Labeling in Lex M

Lex M and MCS-M do a quite similar search along paths of unnumbered vertices, and use this to find the set of vertices of which they change the labels (resp. weight). An easy observation is that the length of a label in Lex M increases by exactly one every time Lex M changes it. We will now study the relation between the length and value of a pair of labels in Lex M, when there is an unnumbered path between the vertices containing the labels.

Lemma 3.1 *Assume that there is an unnumbered path $x_0, x_1, ..., x_k$ in G right before step $\alpha^{-1}(z)$ of Lex M, where $k \geq 1$, $u = x_0$, and $v = x_k$, and let $L_{z-}(x_i) \leq L_{z-}(u)$ where $0 < i < k$. Then $|L_{z-}(u)| > |L_{z-}(v)|$ if and only if $L_{z-}(u) > L_{z-}(v)$.*

Proof. ($\Rightarrow$) Let us first on the contrary assume that $|L_{z-}(u)| > |L_{z-}(v)|$ and $L_{z-}(u) \leq L_{z-}(v)$. Let u' be a vertex such that $\alpha^{-1}(u')$ is a number in $L_{z-}(u) \setminus L_{z-}(v)$, which does exist since $|L_{z-}(u)| > |L_{z-}(v)|$. It follows that $\alpha^{-1}(u') > \alpha^{-1}(z)$ since $\alpha^{-1}(u') \in L_{z-}(u)$. Let p be the largest number such that $0 \leq p < k$ and $\alpha^{-1}(u') \in L_{u'+}(x_p)$. We will show by contradiction that $L_{u'-}(x_p) \geq L_{u'-}(x_i)$ for $p < i \leq k$. Let q be the smallest number such that $p < q \leq k$ and $L_{u'-}(x_q) > L_{u'-}(x_p)$. Now we have a path $x_p, x_{p+1}, ..., x_{p+l} = x_q$, where $L_{u'-}(x_q) > L_{u'-}(x_j)$ for $p \leq j < p + l$, and since $\alpha^{-1}(u') \in L_{u'+}(x_p)$ there exists a path from x_p to u' where the labels of all intermediate vertices in the path are smaller than both the labels of x_p and u'. Thus we have a path from x_q to u', where every intermediate vertex has a smaller label than x_q and u'. This is a contradiction since $\alpha^{-1}(u') \notin L_{u'+}(x_q)$. Now we return to our main proof. Since $L_{u'-}(x_p) \geq L_{u'-}(x_i)$ and $\alpha^{-1}(u') \notin L_{u'+}(x_i)$ for $p < i \leq k$, while $\alpha^{-1}(u') \in L_{u'+}(x_p)$, we have $L_{u'+}(x_p) > L_{u'+}(x_i)$ for $p < i \leq k$. It follows from Lemma 2.3 that $L_{z-}(x_p) > L_{z-}(v)$, where $0 \leq p < k$, since $L_{u'+}(x_p) > L_{u'+}(v = x_k)$. Now we have a contradiction since we assumed that $L_{z-}(x_i) \leq L_{z-}(u)$ where $0 < i < k$ and that $L_{z-}(v) \geq L_{z-}(u)$.
($\Leftarrow$) Let us next on the contrary assume that $|L_{z-}(u)| \leq |L_{z-}(v)|$ and $L_{z-}(u) > L_{z-}(v)$. Let v' be a vertex such that $\alpha^{-1}(v')$ is a number in $L_{z-}(v) \setminus L_{z-}(u)$; such a vertex does exist since $|L_{z-}(v)| \geq |L_{z-}(u)|$ and $L_{z-}(v) < L_{z-}(u)$. It follows that $\alpha^{-1}(v') > \alpha^{-1}(z)$ since $\alpha^{-1}(v') \in L_{z-}(v)$. Let q be the smallest number such that $0 < q \leq k$ and $\alpha^{-1}(v') \in L_{v'+}(x_q)$. We will show by contradiction

that $L_{v'-}(x_q) \geq L_{v'-}(x_i)$ for $0 \leq i < q$. Let p be the largest number such that $0 \leq p < q$ and $L_{v'-}(x_p) > L_{v'-}(x_q)$. Now we have a path $x_p, x_{p+1}, ..., x_{p+l} = x_q$, where $L_{v'-}(x_p) > L_{v'-}(x_j)$ for $p < j \leq p + l$, and since $\alpha^{-1}(v') \in L_{v'+}(x_q)$ there exists a path from x_q to v' where the labels of all intermediate vertices in the path are smaller than both the labels of x_q and v'. Thus we have a path from x_p to v', where every intermediate vertex has a smaller label than x_p and v'. This is a contradiction since $\alpha^{-1}(v') \notin L_{v'+}(x_p)$. Now we return to our main proof. Since $L_{v'-}(x_q) \geq L_{v'-}(x_i)$ and $\alpha^{-1}(v') \notin L_{v'+}(x_i)$ for $0 \leq i < q$, while $\alpha^{-1}(v') \in L_{v'+}(x_q)$, we have $L_{v'+}(x_q) > L_{v'+}(x_i)$ for $0 \leq i < q$. It follows from Lemma 2.3 that $L_{z-}(x_q) > L_{z-}(u)$, where $0 < q \leq k$, since $L_{v'+}(x_q) > L_{v'+}(u = x_0)$. Now we have a contradiction since we assumed that $L_{z-}(x_i) \leq L_{z-}(u)$ where $0 < i < k$ and that $L_{z-}(u) > L_{z-}(v)$. ∎

Lemma 3.2 *Assume that there is an unnumbered path $x_0, x_1, ..., x_k$ in G right before step $\alpha^{-1}(z)$ of Lex M, where $k \geq 1$, $u = x_0$, and $v = x_k$, and let $L_{z-}(x_i) \leq L_{z-}(u)$ where $0 < i < k$. Then $|L_{z-}(u)| < |L_{z-}(v)|$ if and only if $L_{z-}(u) < L_{z-}(v)$.*

Proof. ($\Leftarrow$) Let us assume that $L_{z-}(u) < L_{z-}(v)$ and then prove that $|L_{z-}(u)| < |L_{z-}(v)|$. It follows that $L_{z-}(x_i) < L_{z-}(v)$ for $0 < i < k$ since $L_{z-}(u) < L_{z-}(v)$. Lemma 3.1 can now be used on the path $x_k, x_{k-1}, ..., x_0$ for $k \geq 1$ where $u = x_k, v = x_0$; thus $|L_{z-}(u)| < |L_{z-}(v)|$.
($\Rightarrow$) Let us on the contrary assume that $|L_{z-}(u)| < |L_{z-}(v)|$ and $L_{z-}(u) \geq L_{z-}(v)$. Let v' be a vertex such that $\alpha^{-1}(v')$ is a number in $L_{z-}(v) \setminus L_{z-}(u)$; such a vertex does exist since $|L_{z-}(v)| > |L_{z-}(u)|$. It follows that $\alpha^{-1}(v') > \alpha^{-1}(z)$ since $\alpha^{-1}(v') \in L_{z-}(v)$. Let q be the smallest number such that $0 < q \leq k$ and $\alpha^{-1}(v') \in L_{v'+}(x_q)$. We will show by contradiction that $L_{v'-}(x_q) \geq L_{v'-}(x_i)$ for $0 \leq i < q$. Let p be the largest number such that $0 \leq p < q$ and $L_{v'-}(x_p) > L_{v'-}(x_q)$. Now we have a path $x_p, x_{p+1}, ..., x_{p+l} = x_q$, where $L_{v'-}(x_p) > L_{v'-}(x_j)$ for $p < j \leq p + l$, and since $\alpha^{-1}(v') \in L_{v'+}(x_q)$ there exists a path from x_q to v' where the labels of all intermediate vertices in the path are smaller than both the labels of x_q and v'. Thus we have a path from x_p to v', where every intermediate vertex has a smaller label than x_p and v'. This is a contradiction since $\alpha^{-1}(v') \notin L_{v'+}(x_p)$. Now we return to our main proof. Since $L_{v'-}(x_q) \geq L_{v'-}(x_i)$ and $\alpha^{-1}(v') \notin L_{v'+}(x_i)$ for $0 \leq i < q$, while $\alpha^{-1}(v') \in L_{v'+}(x_q)$, we have $L_{v'+}(x_q) > L_{v'+}(x_i)$ for $0 \leq i < q$. It follows from Lemma 2.3 that $L_{z-}(x_q) > L_{z-}(u)$, where $0 < q \leq k$, since $L_{v'+}(x_q) > L_{v'+}(u = x_0)$. Now we have a contradiction since we assumed that $L_{z-}(x_i) \leq L_{z-}(u)$ where $0 < i < k$ and that $L_{z-}(u) \geq L_{z-}(v)$. ∎

The last case, where $|L_{z-}(u)| = |L_{z-}(v)|$ if and only if $L_{z-}(u) = L_{z-}(v)$ is now easy to prove. We can sum up the two previous lemmas as follows.

Lemma 3.3 *Assume that there is an unnumbered path $x_0, x_1, ..., x_k$ in G right before step $\alpha^{-1}(z)$ of Lex M, where $k \geq 1$, $u = x_0$, and $v = x_k$, and let $L_{z-}(x_i) \leq L_{z-}(u)$ where $0 < i < k$. Then we have*
1. $|L_{z-}(u)| > |L_{z-}(v)|$ if and only if $L_{z-}(u) > L_{z-}(v)$,
2. $|L_{z-}(u)| < |L_{z-}(v)|$ if and only if $L_{z-}(u) < L_{z-}(v)$,
3. $|L_{z-}(u)| = |L_{z-}(v)|$ if and only if $L_{z-}(u) = L_{z-}(v)$.

Proof. The first case is Lemma 3.1, while the second case is Lemma 3.2. The third case follows, since no alternatives are left. ■

4 Lex M versus MCS-M

Lex M and MCS-M are not that different when it comes to altering labels and weights. If a vertex z is selected as the next vertex to be numbered for both algorithms, both Lex M and MCS-M do a search among unnumbered vertices that can be reached from z. In order to better compare the algorithms, these unnumbered vertices are partitioned into components.

Definition 4.1 *Let S be the set of numbered vertices, at some step of Lex M or MCS-M on $G = (V, E)$. Then an* unum component *is a connected component of $G(V \setminus S)$.*

Definition 4.2 *For any vertex u of G, CC_{u-} (resp. CC_{u+}) denotes the set of unum components of G right before (resp. after) numbering vertex u.*

In the proof that Lex M and MCS-M create exactly the same set of triangulations, we need some basic results regarding Lex M, MCS-M, and unum components. First we show that when Lex M or MCS-M processes a vertex in an unum component C they will only change the labels or weights of vertices contained in C. We then prove that if the length of the label in Lex M and the weight in MCS-M are the same for every vertex in an unum component C, then Lex M can choose a vertex z in C as the first vertex to be numbered in C if and only if MCS-M can choose z as the first vertex to be numbered in C. Then we prove that, under the same conditions, the length and the weight are still equal when a vertex in C is processed and the weight for MCS-M and labels for Lex M are updated.

Lemma 4.3 *In any execution of Lex M or MCS-M on a graph G, processing a vertex z of G only affects the unum component of CC_{z-} containing z (i.e. any other unum component of CC_{z-} is still an unum component of CC_{z+} with the same labels or weights).*

Proof. Let C be an unum component of CC_{z-} not containing z. It is evident that after removal of z, C is still an unum component of CC_{z+}. No labels or weights are changed in C, since for any vertex v whose label or weight is modified when processing z, there is a path of unnumbered vertices between z and v, so that v is in the same unum component of CC_{z-} as z. ▪

Lemma 4.4 *We consider two executions of Lex M and MCS-M respectively on a graph G. Let u and u' be vertices of G, and let C be a set of vertices of G such that C is an unum component of G right before processing u (resp. u') in the execution of Lex M (resp. MCS-M) and for every vertex v of C, $|L_{u-}(v)| = w_{u'-}(v)$. Then $hL_{u-}(C) = hW_{u'-}(C)$.*

Proof. We want to show that $hW_{u'-}(C) \subseteq hL_{u-}(C)$ and $hL_{u-}(C) \subseteq hW_{u'-}(C)$ and thus $hW_{u'-}(C) = hL_{u-}(C)$. The first step is to prove that $hW_{u'-}(C) \subseteq hL_{u-}(C)$. Let us on the contrary assume that there exists a vertex $m \in hW_{u'-}(C) \setminus hL_{u-}(C)$, and let l be any vertex in $hL_{u-}(C)$. The unum component C is connected, and every vertex in C is unnumbered. Thus there exists an unnumbered path $x_0, x_1, ..., x_k$ for $0 < k \leq |C| - 1$, where $l = x_0, m = x_k$, and $x_i \in C$ for $0 \leq i \leq k$. Then $L_{u-}(x_i) \leq L_{u-}(l)$ for $0 < i \leq k$ since $l \in hL_{u-}(C)$. We have $L_{u-}(l) > L_{u-}(m)$ since $l \in hL_{u-}(C)$ and $m \notin hL_{u-}(C)$. We have $w_{u'-}(l) \leq w_{u'-}(m)$ since $m \in hW_{u'-}(C)$. From the premises of the lemma, we then have that $|L_{u-}(l)| = w_{u'-}(l) \leq w_{u'-}(m) = |L_{u-}(m)|$. It follows that the path $x_0, x_1, ..., x_k$ is a contradiction to Lemma 3.3.

Next we want to prove that $hL_{u-}(C) \subseteq hW_{u'-}(C)$, and thus $hL_{u-}(C) = hW_{u'-}(C)$. Let us on the contrary assume that there exists a vertex $l \in hL_{u-}(C) \setminus hW_{u'-}(C)$, and let m be any vertex in $hW_{u'-}(C)$. Then there exists a path $x_0, x_1, ..., x_k$ for $0 < k \leq |C| - 1$, where $l = x_0, m = x_k$, and $x_i \in C$ for $0 \leq i \leq k$. We have $L_{u-}(x_i) \leq L_{u-}(l)$ for $0 < i \leq k$ since $l \in hL_{u-}(C)$. We have $w_{u'-}(l) < w_{u'-}(m)$ since $l \notin hW_{u'-}(C)$ and $m \in hW_{u'-}(C)$. Therefore the path $x_0, x_1, ..., x_k$ is a contradiction to Lemma 3.3. ▪

Lemma 4.5 *We consider two executions of Lex M and MCS-M respectively on a graph G. Let z be a vertex of G, and let C be a set of vertices of G such that C is an unum component of G right before processing z in both executions and for every vertex u of C, $|L_{z-}(u)| = w_{z-}(u)$. Then $|L_{z+}(u)| = w_{z+}(u)$ for every vertex u of $C \setminus \{z\}$.*

Proof. Let us on the contrary assume that $|L_{z+}(u)| \neq w_{z+}(u)$ for some $u \in C \setminus \{z\}$. From Lemma 4.3 we know that $z \in C$ if $|L_{z+}(u)| \neq w_{z+}(u)$. Two cases are possible. The first case is $|L_{z+}(u)| = w_{z+}(u) + 1$. There exists at least one path $x_0, x_1, ..., x_k$ for $k \geq 1$, where $u = x_0$, $z = x_k$, $x_i \in C$, and $L_{z-}(x_i) < L_{z-}(u)$ for $0 < i < k$, since $|L_{z+}(u)| = |L_{z-}(u)| + 1$ and C is an unum component of G containing u right before processing z. Then for every such path there exists a vertex x_j where $0 < j < k$ such that $w_{z-}(x_j) \geq w_{z-}(u)$, since $w_{z+}(u) = w_{z-}(u)$. The path $x_0, x_1, ..., x_j$ is a contradiction to Lemma 3.3 because (1) $L_{z-}(x_i) < L_{z-}(u = x_0)$ for $0 < i \leq j$, and specifically $L_{z-}(u) > L_{z-}(x_j)$, and (2) $w_{z-}(u) \leq w_{z-}(x_j)$ and hence, due to our assumption, $|L_{z-}(u)| \leq |L_{z-}(x_j)|$. The second case is when $|L_{z+}(u)|+1 = w_{z+}(u)$ for some vertex $u \in C \setminus \{z\}$. Then there has to exist at least one path $x_0, x_1, ..., x_k$ for some $k \geq 1$, where $u = x_0$, $z = x_k$, $x_i \in C$ for $0 \leq i \leq k$, and $w_{z-}(x_i) < w_{z-}(u) \leq w_{z-}(z)$ for $0 < i < k$, since $w_{z+}(u) = w_{z-}(u) + 1$. Then for every such path there exists a vertex x_j for $0 < j < k$ such that $L_{z-}(x_j) \geq L_{z-}(u)$, since $|L_{z+}(u)| = |L_{z-}(u)|$. Let j be the smallest number such that $L_{z-}(x_j) \geq L_{z-}(u)$. The path $x_0, x_1, ..., x_j$ is a contradiction to Lemma 3.3 because (1) $L_{z-}(x_i) < L_{z-}(u = x_0)$ for $0 < i < j$ and moreover $L_{z-}(u) \leq L_{z-}(x_j)$, and (2) $w_{z-}(u) > w_{z-}(x_j)$ and hence, due to our assumption, $|L_{z-}(u)| > |L_{z-}(x_j)|$. ∎

The three previous lemmas are local observations, and require that Lex-M and MCS-M have an unum component consisting of the same vertices, where the weight in MCS-M is equal to the length of the label in Lex M for every vertex in the unum component. The following definition will be useful to formalize the fact that both algorithms break ties in the same way in unum components.

Definition 4.6 *Let $G = (V, E)$ and ϕ be a mapping from the set of all subsets of V to V, such that if $\phi(S) = u$ then $u \in S$, for each $S \subseteq V$. An execution of Lex M (resp. MCS-M) on G is said to be* compatible with ϕ *if for any vertex u of G, $u = \phi(hL_{u-}(C))$ (resp. $\phi(hW_{u-}(C))$), where C is the unum component of CC_{u-} containing u.*

The idea behind ϕ is the following. If S is a set of vertices in Lex M with the highest label belonging to an unum component, or a set of vertices in MCS-M with the highest weight belonging to an unum component, then $\phi(S)$ is the vertex that is chosen next among vertices of this unum component.

Note that two different executions of Lex M (resp. MCS-M) on G can be compatible with the same mapping ϕ, since ϕ tells which vertex to choose next to be numbered in a given unum component, but does not tell in which unum component to choose the next vertex to be numbered in case some vertices with largest label or weight lie in different unum components.

Lemma 4.7 *We consider two executions of Lex M and MCS-M respectively on a graph $G = (V, E)$. If these executions are compatible with the same mapping ϕ from the set of all subsets of V to V, then they produce the same minimal triangulation of G.*

Proof. We define the following property $P(k)$.
$P(k)$: for any vertices u and u' of G and any set C of k vertices of G, if C is

an unum component of G right before processing u (resp. u') in the execution of Lex M (resp. MCS-M) and for every vertex v of C, $|L_{u-}(v)| = w_{u'-}(v)$ then, the fill edges produced when processing the vertices of C are the same in both executions.
It is sufficient to prove that $P(k)$ holds for $k = n$, since in that case $C = V$, which is an unum component at the beginning of both executions with empty labels and null weights, hence the sets of fill edges produced are the same in both executions.
Let us prove that $P(k)$ holds for k from 1 to n by induction on k.
$P(1)$ is true since the unique vertex of C can produce no fill edge by Lemma 4.3.
We assume that $P(k)$ holds. Let us show that $P(k+1)$ holds. Let u and u' be vertices of G, and let C be a set of $k+1$ vertices of G such that C is an unum component of G right before processing u (resp. u') in the execution of Lex M (resp. MCS-M) and for every vertex v of C, $|L_{u-}(v)| = w_{u'-}(v)$. By Lemma 4.3, these conditions are maintained until a vertex z (resp. z') of C is numbered for the first time, from the moment when u (resp. u') is about to be numbered in the execution of Lex M (resp. MCS-M) (possibly $z = u$ or $z' = u'$, if u or u' belongs to C). By Lemma 4.4, $hL_{z-}(C) = hW_{z'-}(C)$, and as both executions are compatible with ϕ, $z = \phi(hL_{z-}(C)) = \phi(hW_{z'-}(C)) = z'$. By Lemma 4.5, $|L_{z+}(v)| = w_{z+}(v)$ for every vertex v of $C \setminus \{z\}$. So the processing of z modifies the labels or weights of the same vertices of C in both executions, and since by Lemma 4.3 the labels or weights of the vertices of $G \setminus C$ are unchanged, the processing of z produces the same fill edges in both executions. Moreover, the new unum components obtained from C by removing z are the same in both executions. So, by the induction hypothesis on these new unum components which contain at most k vertices and for which the condition on labels and weights holds after processing z, we have the fill edges produced when processing the vertices of $C \setminus \{z\}$ are the same in both executions, which completes the proof. ∎

In order to complete the proof that Lex M and MCS-M produce the same set of chordal graphs, two more arguments are required. The first is to show that for any execution of Lex M (resp. MCS-M) there exists a mapping ϕ compatible with this execution. The second is to show that for any mapping ϕ from the set of all subsets of V to V such that for any subset S of V, $\phi(S)$ belongs to S, there is an execution of MCS-M (resp. Lex M) compatible with ϕ. Then the rest will follow from Lemma 4.7.

Theorem 4.8 *Lex M and MCS-M produce the same minimal triangulations of a given graph $G = (V, E)$.*

Proof. Observe that for any execution of Lex M on G producing the triangulated graph H there exists a compatible mapping ϕ. This mapping ϕ can simply be constructed as follows: For every vertex $z \in V$ set $\phi(hL_{z-}(C))$ to z, where C is the unum component in CC_{z-} containing z. Any mapping ϕ which fulfills this requirement will be compatible with the execution of Lex M producing H. Note that during Lex M, $hL_{z-}(C) \neq hL_{z'-}(C')$ for all vertices $z \neq z'$ with $z \in C$ and $z' \in C'$ where $C \in CC_{z-}$ and $C' \in CC_{z'-}$, since the highest numbered of z and z' does not belong to both sets. Thus we never consult $\phi(S)$ for the same set S of vertices more than once.

We now consider an execution of MCS-M on G compatible with ϕ. Such an execution exists. At each step it is sufficient to choose an unum component C containing a vertex with largest weight and to choose $\phi(hW(C))$ as next vertex to be numbered. By Lemma 4.7, this execution of MCS-M produces the graph H.

The proof in the other direction is completely symmetric. ■

5 Conclusion

Even though MCS-M and Lex M can create different orderings, we prove that they create the same set of triangulations, and thereby answer an open question given in [3]. We show this by defining unum components, which are the connected subgraphs when the numbered vertices are removed from the graph. Then we show that two executions of Lex M and MCS-M breaking ties in the same way in unum components compute the same minimal triangulation of the input graph, so that Lex M and MCS-M compute the same set of minimal triangulations of any graph.

We also observe that each of the unum components can be computed individually since they do not affect each other. This property could possibly be used to improve the practical running time for both algorithms.

Acknowledgment:

The author wishes to thank Pinar Heggernes for her useful comments and suggestions, and the two anonymous referees who assisted in the presentation of this paper.

References

[1] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.

[2] A. Berry. A wide-range efficient algorithm for minimal triangulation. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, pages S860–S861, 1999.

[3] A. Berry, J. R. S. Blair, P. Heggernes, and B. W. Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39:287–298, 2004.

[4] A. Berry, J.P. Bordat, P. Heggernes, G. Simonet, and Y. Villanger. A wide-range algorithm for minimal triangulation from an arbitrary ordering. *Journal of Algorithms*, 58(1):33–66, 2006.

[5] A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for dynamically maintaining chordal graphs. In *Proceedings 14th International Symposium on Algorithms and Computation - ISAAC 2003*, pages 47 – 57. Springer Verlag, 2003. LNCS 2906.

[6] J. R. S. Blair, P. Heggernes, and J. A. Telle. A practical algorithm for making filled graphs minimal. *Theor. Comput. Sci.*, 250:125–141, 2001.

[7] E. Dahlhaus. Minimal elimination ordering inside a given chordal graph. In R. H. Möhring, editor, *Graph Theoretical Concepts in Computer Science - WG '97*, pages 132–143. Springer Verlag, 1997. Lecture Notes in Computer Science 1335.

[8] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Math.*, 15:835–855, 1965.

[9] T. Ohtsuki. A fast algorithm for finding an optimal ordering in the vertex elimination on a graph. *SICOMP*, 5:133–145, 1976.

[10] S. Parter. The use of linear graphs in Gauss elimination. *SIAM Review*, 3:119–130, 1961.

[11] B. W. Peyton. Minimal orderings revisited. *SIAM J. Matrix Anal. Appl.*, 23(1):271–294, 2001.

[12] D. Rose, R.E. Tarjan, and G. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:146–160, 1976.

[13] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.

Paper IV

Computing Minimal Triangulations in Time $O(n^\alpha \log n) = o(n^{2.376})$

Pinar Heggernes Jan Arne Telle Yngve Villanger

pinar@ii.uib.no telle@ii.uib.no yngvev@ii.uib.no

Department of Informatics, University of Bergen, N-5020 Bergen, Norway

Abstract

The problem of computing minimal triangulations of graphs, also called minimal fill, was introduced and solved in 1976 by Rose, Tarjan, and Lueker [17] in time $O(nm)$, thus $O(n^3)$ for dense graphs. Although the topic has received increasing attention since then, and several new results on characterizing and computing minimal triangulations have been presented, this first time bound has remained the best. In this paper we introduce an $O(n^\alpha \log n)$ time algorithm for computing minimal triangulations, where $O(n^\alpha)$ is the time required to multiply two $n \times n$ matrices. The current best known α is less than 2.376, and thus our result breaks the long standing asymptotic time complexity bound for this problem. To achieve this result, we introduce and combine several techniques that are new to minimal triangulation algorithms, like working on the complement of the input graph, graph search for a vertex set A that bounds the size of the connected components when A is removed, and matrix multiplication.

1 Introduction and motivation

Any graph can be embedded in a chordal graph by adding a set of edges called fill, and the resulting graph is called a triangulation of the input graph. When the added set of fill edges is inclusion minimal, the resulting triangulation is called a minimal triangulation. The first algorithms for computing minimal triangulations were given in independent works of Rose, Tarjan, and Lueker [17], and Ohtsuki, Cheung, and Fujisawa [13, 14] already in 1976. Among these, the algorithms of [13] and [17] have a time bound of $O(nm)$, where n is the number of vertices and m is the number of edges of the input graph. These first algorithms were motivated by the need to find good pivotal orderings for Gaussian elimination, and the mentioned papers gave characterizations of minimal triangulations through minimal elimination orderings. Since then, the problem has received increasing attention, and several new characterizations of minimal triangulations connected to minimal separators of the input graph have been given [5, 10, 15], totally independent of the connection to Gaussian elimination. The connection to minimal separators has increased the importance of minimal triangulations from a graph theoretical point of view, and minimal triangulations have proved useful in reconstructing evolutionary history through phylogenetic trees [9]. As a result, algorithms based on the new characterizations have been given [3, 8], while at the same time new algorithms based on elimination orderings also appeared [4, 7, 16]. However, the best time bound remained unchanged, and trying to break the asymptotic $O(n^3)$ bound of computing minimal triangulations, in particular for dense graphs, became a major theoretical challenge concerning this topic.

In this paper, we introduce an $O(n^\alpha \log n)$ time algorithm to compute minimal triangulations of arbitrary graphs, where $O(n^\alpha)$ is the time bound of multiplying two $n \times n$ matrices. Currently the lowest value of α is $2.375 < \alpha < 2.376$ by the algorithm of Coppersmith and Winograd [6]. Hence the current time bound for our algorithm is $o(n^{2.376})$, since $\log n = o(n^\epsilon)$ for all $\epsilon > 0$. In order to achieve this time bound, we use several different techniques, one of which is matrix multiplication to make parts of the input graph into cliques. Our algorithm runs for $O(\log n)$ iterations, and at each iteration the total work is bounded by the time needed for matrix multiplication. In order to achieve $O(\log n)$ iterations, we show how to recursively divide the problem into independent subproblems of a constant factor smaller size using a specialized search technique. In order to bound the amount of work at each iteration by $O(n^\alpha)$, we store and work on the complement graphs for each subproblem, in which case the subproblems do not overlap in any (non)edges. In addition, we use both the minimal separators and the potential maximal cliques of the input graph, combining the results of [5], [10], and [15].

Independent of our work, a very recent and thus yet unpublished result of Kratsch and Spinrad [12] uses matrix multiplication to give a new implementation of the minimal triangulation algorithm Lex M from 1976 [17]. Based on the matrix multiplication algorithm of [6] their presented time complexity is $O(n^{2.688})$. Other than the use of matrix multiplication, their approach is totally different from ours. Kratsch and Spinrad used matrix multiplication for similar problems in their SODA 2003 paper [11].

After the next section which contains some basic definitions, we give the main structure of our algorithm in Section 3, followed by the important subroutine for partitioning into balanced subproblems in Section 4, before tying these parts together in the last section.

2 Background and notation

We consider simple undirected and connected graphs $G = (V, E)$ with $n = |V|$ and $m = |E|$. When G is given, denote the vertex and edge set of G by $V(G)$ and $E(G)$, respectively. For a set $A \subseteq V$, $G(A)$ denotes the subgraph of G induced by the vertices in A. A is called a *clique* if $G(A)$ is complete. The process of adding edges to G between the vertices of $A \subseteq V$ so that A becomes a clique in the resulting graph is called *saturating* A. The *neighborhood* of a vertex v in G is $N_G(v) = \{u \mid uv \in E\}$, and the *closed neighborhood* of v is $N_G[v] = N_G(v) \cup \{v\}$. Similarly, for a set $A \subseteq V$, $N_G(A) = \cup_{v \in A} N_G(v) \setminus A$, and $N_G[A] = N_G(A) \cup A$. $|N_G(v)|$ is the *degree* of v. When graph G is clear from the context, we will omit subscript G.

A vertex set $S \subset V$ is a *separator* if $G(V \setminus S)$ is disconnected. Given two vertices u and v, S is a *u, v-separator* if u and v belong to different connected components of $G(V \setminus S)$, and S is then said to *separate* u and v. Two separators S and T are said to be *crossing* if S is a u, v-separator for a pair of vertices $u, v \in T$, in which case T is an x, y-separator for a pair of vertices $x, y \in S$ [10, 15]. A u, v-separator S is *minimal* if no proper subset of S separates u and v. In general, S is a *minimal separator* of G if there exist two vertices u and v in G such that S is a minimal u, v-separator. It can be easily verified that S is a minimal separator if and only if $G(V \setminus S)$ has two distinct connected components C_1 and C_2 such that $N_G(C_1) = N_G(C_2) = S$. In this case, C_1 and C_2 are called *full components*, and S is a minimal u, v-separator for *every* pair of vertices $u \in C_1$ and $v \in C_2$.

A *chord* of a cycle is an edge connecting two non-consecutive vertices of the cycle. A graph is *chordal*, or equivalently *triangulated*, if it contains no chordless cycle of length ≥ 4. A graph $G' = (V, E \cup F)$ is called a *triangulation* of $G = (V, E)$ if G' is chordal. The edges in F are called *fill edges*. G' is a *minimal triangulation* if $(V, E \cup F')$ is non-chordal for every proper subset F' of F. It was shown in [17] that a triangulation G' is minimal if and only if every fill edge is the unique chord of a 4-cycle in G'. Another characterization of minimal triangulations which is central to our results is that G' is a minimal triangulation of G if and only if G' is the result of saturating a maximal set of pairwise non-crossing minimal separators of G [15].

By the results of Kloks, Kratsch, and Spinrad [10], and Parra and Scheffler [15], it can be shown that the following recursive procedure creates a minimal triangulation of G: Take any connected vertex subset K and let $A = N[K]$, compute the connected components $C_1, ..., C_k$ of $G(V \setminus A)$, saturate each set $N(C_i)$ for $1 \leq i \leq k$ and call the resulting graph G', then compute a minimal triangulation of each subgraph $G'(N[C_i]), 1 \leq i \leq k$, and of $G'(A)$ independently. The key to understand this is to note that the saturated sets $N(C_i)$ are non-crossing minimal separators of G and G'. Thus the problem decomposes into independent subproblems overlapping only at the saturated minimal separators, and we can continue recursively on each subproblem that is not complete. This procedure is basic to the main structure of our algorithm.

An extension of the above mentioned results, which we also use in our algorithm, was presented by Bouchitté and Todinca in [5]. There, a *potential maximal clique (pmc)* of G is defined to be a maximal clique in some minimal triangulation of G. If A is a pmc, then it is shown in [5] that whole A will automatically be saturated in the above recursive procedure instead of appearing as a subproblem, and that this modified procedure indeed characterizes minimal triangulations. In this case A is not necessarily $N[K]$ for a connected set K. The following theorem from [5] characterizes a pmc, and it will be used to prove the correctness of our balanced partition algorithm in Section 4.

Theorem 2.1 (Bouchitté and Todinca [5]) *Given a graph $G = (V, E)$, let $P \subseteq V$ be any set of vertices, and let $C_1, C_2, ..., C_k$ be the connected components of $G(V \setminus P)$. P is a pmc of G if and only if*

1. *$G(V \setminus P)$ has no full component, and*
2. *P is a clique when every $N(C_i)$ is saturated for $1 \leq i \leq k$.*

3 The new algorithm and the data structures

Observe that the total work for saturating all sets $N(C_i), 1 \leq i \leq k$, in the recursive procedure described in the previous section requires $O(n^3)$ time if it is done straightforwardly, as these sets might overlap heavily and contain $O(n)$ vertices each. With help of matrix multiplication, this total time can be reduced to $O(n^\alpha)$. We construct the following matrix $M = M_{G,A}$: for each vertex $v \in V(G)$ there is a row in M, for each connected component C of $G(V \setminus A)$ there is a column in M, and entry $M(v, C) = 1$ if $v \in N(C)$. All other entries are zero. Now we perform the multiplication MM^T, and in the resulting symmetric matrix, entry $(u, v) = (v, u)$ is nonzero if and only if u and v both belong to a common set $N(C)$ for some C. Thus MM^T is the adjacency matrix of a graph in which each $N(C)$ is a clique. The use of matrix multiplication for this purpose was first mentioned in [11].

Once MM^T is computed, the edges indicated by its nonzero entries can be added to G, resulting in the partially filled graph G', and the subproblems $G'(N[C_i]), 1 \leq i \leq k$, and $G'(A)$ can be extracted. Now for each subproblem this process can be repeated recursively. However, it is important that we do not perform a matrix multiplication for each subproblem in the further process, but create only *one* matrix and perform a single matrix multiplication for all subproblems of each level in the recursion tree. Thus in the resulting matrix MM^T, entry (u, v) is nonzero if and only if there is a connected component C of one of the subproblems of this level such that $u, v \in N_{G'}(C)$. For this reason, we cannot actually use recursion, and we have to keep track of all subproblems belonging to the same level. We do this by using two queues Q_1 and Q_2 which will memorize all subproblems for the current and next level respectively. Only those new subproblems that are not cliques in the partially filled graph should survive to the next iteration. For a new subproblem on vertex set $N[C_i]$ appearing from a connected component C_i after removing A we check this before the saturation, as we already know that the saturation will make $N(C_i)$ into a clique and not add any other edges to the graph induced by $N[C_i]$. However, for the subproblem on vertex set A itself we must wait until after the saturation before checking whether A now induces a clique, and for that reason we store the vertex sets A temporarily in a third queue Q_3.

Our algorithm, which we call FMT - Fast Minimal Triangulation - is given in Figure 1. The process of computing a good vertex set A is the most complicated part of this algorithm, and this part will be explained in the next section when we give the details of Algorithm Partition that returns such a set A. For the time being, and for the correctness of Algorithm FMT it is important and sufficient to note that Algorithm Partition returns a set A, where either $A = N[K]$ for some connected vertex set K, or A is a pmc.

The following lemma proves the correctness of our algorithm, as well as the correctness of the recursive procedure described in the previous section.

Lemma 3.1 *Algorithm FMT computes a minimal triangulation of the input graph, as long as the Partition(H) subroutine returns a set $A \subset V(H)$ where either $A = N[K]$ for some connected vertex set K or A is a pmc.*

Proof. Let $G = (V, E)$ be the input graph and let K be a set of vertices such that $G(K)$ is connected. It is shown in [1] that the set of minimal separators of G that are subsets of $N(K)$ is exactly the set $\{N(C) \mid C$ is a connected component of $G(V \setminus N[K])\}$. In [5] it is shown that if P is a pmc then the set of minimal separators that are contained in P is exactly the set $\{N(C) \mid C$ is a connected component of $G(V \setminus P)\}$.

Since A is always chosen so that either $A = N[K]$ for a connected set K, or A is a pmc (this will be proved in Section 4), then it follows that all sets that are saturated at the first iteration of Algorithm FMT are minimal separators of G. We will now argue that these minimal separators are non-crossing. Assume on the contrary that two crossing separators $S = N(C_1)$ and $T = N(C_2)$ are saturated at the first iteration, where C_1 and C_2 are two distinct connected components of $G(V \setminus A)$. Thus there are two vertices $u, v \in T$ with $u, v \notin S$ such that S is a minimal u, v-separator in G. Since $u, v \in T = N(C_2)$, and S does not contain any vertex of C_2, the removal of S cannot separate u and v as there is a path

[1]What we want to do here is to take $H(N_H[C])$, make $N_H(C)$ into a clique, and then insert the resulting graph into Q_2. However, we do not have time to even compute $H(N_H[C])$. Thus we start with a complete graph on vertex set $N_H[C]$ and remove only edges uv with an endpoint in u that do not appear in D.

```
Algorithm FMT - Fast Minimal Triangulation
Input: An arbitrary non-complete graph $G = (V, E)$.
Output: A minimal triangulation $G'$ of $G$.

Let $Q_1, Q_2$ and $Q_3$ be empty queues;  Insert $G$ into $Q_1$;  $G' = G$;
repeat
    Construct a zero matrix $M$ with a row for each vertex in $V$;
    (columns are added later);
    while $Q_1$ is nonempty do
        Pop a graph $H = (U, D)$ from $Q_1$;
        Call Algorithm Partition($H$) which returns a vertex subset $A \subset U$;
        Push vertex set $A$ onto $Q_3$;
        for each connected component $C$ of $H(U \setminus A)$ do
            Add a column in $M$ such that $M(v, C) = 1$ for all vertices $v \in N_H(C)$;
            if $\exists$ non-edge $uv$ in $H(N_H[C])$ with $u \in C$ then
                Push $H_C = (N_H[C], D_C)$ onto $Q_2$, where $uv \notin D_C$ only if
                $u \in C$ and $uv \notin D$; 1
        end-for
    end-while
    Compute $MM^T$;
    Add to $G'$ the edges indicated by the nonzero elements of $MM^T$;
    while $Q_3$ is nonempty do
        Pop a vertex set $A$ from $Q_3$;
        if $G'(A)$ is not complete then Push $G'(A)$ onto $Q_2$;
    end-while
    Swap names of $Q_1$ and $Q_2$;
until $Q_1$ is empty
```

Figure 1: Algorithm FMT : Fast Minimal Triangulation.

between u and v through vertices of C_2. This contradicts the assumption that S is a u, v-separator, and thus we can conclude that the minimal separators saturated at the first step are all pairwise non-crossing. It is important to observe that once these separators are saturated, all minimal separators of G that cross any of these will disappear, as the saturated sets do not contain pairs of vertices that are separable. At each iteration, any minimal separator of G' is a minimal separator of G [15]. Thus the minimal separators that we discover at each iteration will not cross the minimal separators discovered and saturated at previous iterations.

At each new iteration, the above argument can be applied to each subgraph H, and thus we compute a set of non-crossing minimal separators of each subgraph H at each iteration. We have already argued that these cannot cross any of the saturated minimal separators of previous iterations. We must also argue that no minimal separator of a subgraph of an iteration crosses a minimal separator of another subgraph of the same iteration. But this is straightforward as these subgraphs only intersect at cliques, and thus their sets of minimal separators are disjoint.

So, our algorithm computes and saturates a set of non-crossing minimal separators at each iteration. Since we continue this process until all minimal separators of G' are saturated, by the results of [10] and [15], we create a minimal triangulation. ■

If we consider merely correctness, any set A that fulfills the requirements can be chosen arbitrarily; for example $A = N[u]$ for a single vertex u, as in [2]. In order to achieve the desired time complexity, we will devote the next section to describing how to carefully choose a vertex subset A in each subproblem so that the number of iterations of the **repeat**-loop becomes $O(\log n)$.

In this section, we will argue that each iteration of the algorithm can be carried out in $O(n^\alpha)$ time. We start with the following lemma, which will give us the desired bound for the matrix multiplication step.

Lemma 3.2 *At each iteration of Algorithm FMT, the number of columns in matrix M is less than n.*

Proof. The sequence of iterations of the algorithm gives rise to an iterative refinement of a tree-decomposition of the graph G', a property first shown for the LB-treedec algorithm discussed in [8]. Simplifying the standard notation, we say that a *tree-decomposition* T_i of a graph G is a collection of *bags*, subsets of the vertex set of G, arranged as nodes of a tree such that the bags containing any given vertex induce a connected subtree, and such that every pair of adjacent vertices of G is contained in some bag (see *e.g.* page 549 of [19] for the standard definition.) At the first iteration we have the trivial tree-decomposition T_1 with all vertices of G' in a single bag, until the last iteration p where the tree-decomposition T_p is in fact a clique tree of the now chordal graph G', with each bag inducing a unique maximal clique. We prove this by showing the following:

Loop invariant: At the start of iteration s we have a tree-decomposition T_s of the current partially filled graph G' whose bags consist of some vertex subsets inducing cliques, which are the vertices of subproblems inducing cliques as discovered so far by our algorithm, and where remaining bags are the vertex sets of subproblems in Q_1. The intersection of two neighboring bags in T_s is a saturated minimal separator of G' and thus induces a clique. T_s is non-redundant, meaning that if A, B are bags of T_s then we do not have $A \subseteq B$.

The invariant is clearly true for the trivial tree-decomposition T_1 with a single bag. Let vertex set U be a bag of T_s appearing as subproblem $H = (U, D)$ in

Q_1. The algorithm proceeds to find $A \subset U$ and produces new vertex subsets $A, N[C_1], N[C_2], ..., N[C_k]$ where each C_i is a component of $G'(U \setminus A)$. The node of bag U in T_s is in T_{s+1} split into a k-star with center-bag A and leaf-bags $N[C_1], N[C_2], ..., N[C_k]$. Since A is a pmc or $A = N[K]$, it follows that this star is a tree-decomposition of $G'(U)$ which is non-redundant. The node of a neighboring bag X of U in the tree of T_s will also be split into a star, unless X induces a clique in which case it remains a single node, i.e. a trivial star. These two stars appearing from adjacent nodes in T_s will be joined in T_{s+1} by an edge between two bags U' and X' that each contain $U \cap X$. Such a bag must exist in each star since $U \cap X$ already induced a clique.

The tree-decomposition T_{s+1} is constructed by applying the construction above to each bag, and to adjacent pairs of bags, of T_s. After newly found minimal separators in G' have been saturated then T_{s+1} will be a tree-decomposition of G', as is easily checked. It remains to show that T_{s+1} is non-redundant. We do this by showing that none of the new vertex subsets $A, N[C_1], N[C_2], ..., N[C_k]$ are contained in $U \cap X$. The crucial fact is that each vertex in $U \cap X$ has a neighbor in $U \setminus X$, since $U \setminus X$ was a component of the minimal separator $U \cap X$. If A was chosen as $A = N[K]$ then even if $K \subseteq U \cap X$ we would therefore not have $A \subseteq U \cap X$. Likewise, we could have some component C_i of $G'(U) \setminus A$ with $C_i \subseteq U \cap X$, but we would never have $N[C_i] \subseteq U \cap X$. If A instead was chosen as a pmc then we cannot have $A \subseteq U \cap X$, as $U \cap X$ was a minimal separator and a maximal clique cannot be part of a minimal separator. Thus, T_{s+1} is non-redundant. Since any bag of T_{s+1} that does not induce a clique is put back onto Q_1 before the next iteration we have established the loop invariant.

Note that each column added to matrix M in the algorithm gives rise to a unique bag of T_{s+1}. Since the number of bags in the final tree-decomposition T_p is at most n, one for each maximal clique in a chordal graph, and the number of bags in trees $T_1, ..., T_p$ is strictly increasing, we have proved the lemma. ∎

Consequently, the matrix multiplication step requires $O(n^\alpha)$. In order to be able to bound the time for the rest of the operations of each iteration by $O(n^\alpha)$, we will store and work on the *non-edges*, i.e., the edges of the complement graph for each subproblem. Note that subproblems can overlap both in vertices and in edges, which makes it difficult to bound the sum of their sizes for the desired time analysis. A non-edge uv is discarded when it becomes an edge (that is, when it is added to the graph) or when vertices u and v are separated into different subproblems, and if it is not discarded it only appears in a single subproblem in the next iteration. Hence subproblems overlap only in cliques, so if we work on the complement of these subgraphs, then they actually do not overlap in any edges at all!

For each subgraph $H = (U, D)$ in Q_1, let $\bar{E}(H) = \binom{U}{2} \setminus D$ be the set of non-edges of H. Our data structure for each subproblem H is the adjacency list of $\bar{H} = (U, \bar{E}(H))$, where we also store the degree of each vertex in $\bar{H}$. It is an easy exercise to show that all linear time operations that we need to do for H, like computing the connected components and neighborhoods, can be done using only $\bar{H}$ in time $O(|\bar{E}(H)| + |V(H)|)$.

An interesting point is also that, when complement graphs are used, matrix multiplication is not necessary to saturate $N_H(C)$ of each subproblem $N_H[C]$, however it is still necessary in order to saturate the subsets of A that become cliques. In the implementation of our algorithm, for each subproblem $H(N_H[C])$, we push the complement graph consisting of all non-edges of $H(N_H[C])$ with at least one endpoint in C onto Q_2. (This corresponds to Line 12 of Algorithm FMT.) We do this only if such a non-edge of H exists. Since these complement graphs consist of non-edges of $H(C)$ and non-edges of $H(N_H[C])$ between C and $N_H(C)$, all such subproblems can be computed in a total time of $O(|\bar{E}(H)| + |V(H)|)$ for H. Since we omit all non-edges between vertices belonging to $N_H(C)$, this actually corresponds to saturating $N_H(C)$ automatically.

After the matrix multiplication step, we look up in MM^T every edge of the complement of G' to check whether or not this non-edge should survive or should be deleted because it has now become a fill edge of G'. Since subproblems do not overlap in any non-edges checking whether or not $G'(A)$ is now complete can be done in a total of $O(n^2)$ time for all vertex subsets A in Q_3.

Thus, for the implementation of our algorithm, we compute $\bar{G}$ at the beginning, and use the complement graphs throughout the algorithm. This way, all operations described within an iteration can be completed within $O(n^\alpha)$ time. For clearness, we will give the algorithms on the actual graphs and not on complement graphs. We denote the set of non-edges of graph H by $\bar{E}(H)$.

With the given data structures and explanations, it should be clear that all operations during one iteration, outside of Algorithm Partition, can be performed in $O(n^\alpha)$ time.

4 Efficient Partition into balanced subproblems

In this section we will show how to compute vertex subsets A for each subproblem H in order to achieve an even partitioning into subproblems. Since each subproblem that results from H will not contain more than $\frac{4}{5}|\bar{E}(H)|$ non-edges, this will guarantee $O(\log n)$ iterations of the while loop of Algorithm FMT. [2] The algorithm that we present for doing this will have running time $O(|\bar{E}(H)| + |V(H)|)$ on each input subgraph H.

The computation of vertex subset A for each subgraph $H = (U, D)$ is done by Algorithm Partition which is given in Figure 2. This algorithm examines every

[2]The constant $\frac{4}{5}$ can in fact be replaced by $\frac{q-1}{q}$ for any $q \geq 5$. An implementation could make use of this fact to experimentally find the best value of q.

Algorithm Partition
Input: A graph $H = (U, D)$ (a subproblem popped from Q_1).
Output: A subset $A \subset U$ such that either $A = N[K]$ for some connected $H(K)$ or A is a pmc of H (and G').

```
Part I: defining P
Unmark all vertices of $H$;
$k = 1$;
while $\exists$ unmarked vertex $u$ do
    if $\mathcal{E}_{\bar{H}}(U \setminus N_H[u]) < \frac{2}{5}|\bar{E}(H)|$ then
        Mark $u$ as an s-vertex (stop vertex);
    else
        $C_k = \{u\}$;
        Mark $u$ as a c-vertex (component vertex);
        while $\exists\, v \in N_H(C_k)$ which is unmarked or marked as an s-vertex do
            if $\mathcal{E}_{\bar{H}}(U \setminus N_H[C_k \cup \{v\}]) \geq \frac{2}{5}|\bar{E}(H)|$ then
                $C_k = C_k \cup \{v\}$;
                Mark $v$ as a c-vertex (component vertex);
            else
                Mark $v$ as a p-vertex (pmc vertex);
                Associate $v$ with $C_k$;
            end-if
        end-while
        $k = k + 1$;
    end-if
end-while
$P$ = the set of all p-vertices and s-vertices;

Part II: defining A
if $H(U \setminus P)$ has a full component $C$ then
    $A = N_H[C]$;
else if there exist two non-adjacent vertices $u, v$ such that $u$ is an s-vertex
and $v$ is an s-vertex or a p-vertex then
    $A = N_H[u]$;
else if there exist two non-adjacent p-vertices $u$ and $v$, where $u$ is associated
with $C_i$ and $v$ is associated with $C_j$ and $u \notin N_H(C_j)$ and $v \notin N_H(C_i)$ then
    $A = N_H[C_i \cup \{u\}]$;
else
    $A = P$;
end-if
```

Figure 2: Algorithm Partition.

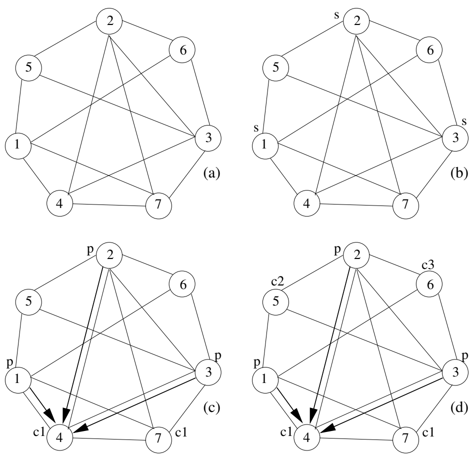

Figure 3: We give an example of how set P is found from graph H. In (a), the number of non-edges $|\bar{E}(H)| = 7$, and the important bound for finding P is therefore $\frac{2}{5}|\bar{E}(H)| = 2.8$. First the algorithm decides if vertex 1 can be contained in component C_1 by performing the test $\mathcal{E}_{\bar{H}}(U \setminus N_H[1]) < \frac{2}{5}|\bar{E}(H)|$. Vertex set $U \setminus N_H[1] = \{2, 3\}$, and $\mathcal{E}_{\bar{H}}(\{2, 3\}) = |N_{\bar{H}}(2)| + |N_{\bar{H}}(3)| = 2$, thus 1 cannot be contained in a component and it is marked as an **s**-vertex. The same result is obtained when testing 2 and 3, as shown in (b). Vertex set $U \setminus N_H[4] = \{5, 6\}$, and $\mathcal{E}_{\bar{H}}(\{5, 6\}) = |N_{\bar{H}}(5)| + |N_{\bar{H}}(6)| = 6 > 2.8$, thus vertex 4 becomes the first vertex in component C_1. The algorithm will now try to extend C_1 by including vertices from $N(C_1)$ in C_1. Observe that $1 \in N(C_1)$, and that including it in C_1 will make the value of the test $\mathcal{E}_{\bar{H}}(U \setminus N_H[C_1 \cup \{1\}]) \geq \frac{2}{5}|\bar{E}(H)|$ false, and thus 1 becomes a **p**-vertex and it is associated to C_1 as shown in (c). The same argument is used to change the marks of 2 and 3 as **p**-vertices, and to associate these with C_1. For vertex 7 we get the opposite result from the test, and therefore this vertex is placed in C_1. Figure (c) shows that 4 and 7 are marked as **c**-vertices, and the index after the **c** indicates that they belong to C_1. Finally, in (d) we create the components C_2 and C_3 containing vertices 5 and 6, respectively. All vertices in the neighborhood of these components are already marked as **p**-vertices and thus there is nothing more to do. As a result, the computed set $P = \{1, 2, 3\}$. For the rest of Algorithm Partition, since each connected component of $H(U \setminus P)$ is a full component, case 1 will apply, and the resulting returned set A is simply the union of P with one of these components, for example $A = \{1, 2, 3, 6\}$. Note that there exist extreme cases where every vertex is marked as an **s**-vertex. An example of this is a cycle of length 16 with added chords so that every vertex is adjacent to all vertices except the one on the opposite side of the cycle. The number of non-edges in this graph is 8 and the graph induced by any vertex and its neighborhood contains 7 non-edges. Such an extreme case causes no problem for our algorithm, as case 2 will apply and an appropriate $A \subset U$ will still be found.

vertex of H, and tries to place it into a connected component C that results from removing some set P of vertices from H, as long as $H(N_H[C])$ does not become too large with respect to the number of non-edges. The vertices that cannot be placed into any C with a small enough $H(N_H[C])$ in this way, constitute exactly the set P whose removal from H results in these balanced connected components. This way, we compute a vertex set P such that all connected components C of $H(U \setminus P)$ have the nice property that $H(N_H[C])$ contains less than a constant factor of the non-edges of H. The computation of P is illustrated by an example given in Figure 3.

However, after P is computed, we cannot bound the number of non-edges that will belong to $G'(P)$ after the saturation. Furthermore it might be the case that neither $P = N_H[K]$ for a connected vertex set K as required, nor P is a potential maximal clique, which implies that $N(C)$ is not necessarily a minimal separator for every connected component C of $H(U \setminus P)$. Thus we cannot simply use P as our desired set A. The set A is instead obtained using information gained through the computation of P, and we prove in Theorem 4.3 that it fulfills the requirements that were used to prove the correctness of Algorithm FMT, and that the resulting subproblems all have at most $\frac{4}{5}|\bar{E}(H)|$ non-edges.

During Algorithm Partition, the vertices that we are able to place into small enough connected components are marked as **c**-vertices. The remaining vertices (which constitute P) are of two types: **p**-vertices have neighbors in a connected component of $H(U \setminus P)$, whereas **s**-vertices do not. For each connected component C of $H(U \setminus P)$ we want to ensure that the number $|\bar{E}(H(N_H[C]))|$, i.e. the number of non-edges with both endpoints in $N_H[C]$, is less than some fraction of $|\bar{E}(H)|$. The obstacle is that we cannot compute this number straightforwardly for all connected components of $H(U \setminus P)$ in the given time, since the non-edges between vertices in $P \cap N_H[C]$ could be contained in too many such computations. However, we are able to give upper and lower bounds on $|\bar{E}(H(N_H[C]))|$ by summing the degrees in $\bar{H}$ of vertices in each $N_H[C]$, which we compute in the following roundabout manner in order to stay within the time limits. Define $\mathcal{E}_{\bar{H}}(S)$ to be the sum of degrees in $\bar{H}$ of vertices in $S \subseteq U = V(H)$. Since sum of degrees is equal to twice the number of edges, we have $\mathcal{E}_{\bar{H}}(S) = 2|\bar{E}(H)| - \mathcal{E}_{\bar{H}}(U \setminus S)$. The quantity $\mathcal{E}_{\bar{H}}(U \setminus N_H[C])$ we indeed do have the time to compute, as we will explain in the proof of Lemma 4.1.

$$\mathcal{E}_{\bar{H}}(U \setminus N_H[C]) = \sum_{v \in U \setminus N[C]} |N_{\bar{H}}(v)|$$

When checking whether $\mathcal{E}_{\bar{H}}(U \setminus N_H[C_k \cup \{v\}]) \geq \frac{2}{5}|\bar{E}(H)|$ in Algorithm Partition we are indirectly checking whether $|\bar{E}(N_H[C_k \cup \{v\}])| \leq \frac{4}{5}|\bar{E}(H)|$, which is what we indeed want to know. The discussion in the proof of Lemma 4.2 explains this connection. The value $\mathcal{E}_{\bar{H}}(U \setminus N_H[C_k \cup \{v\}])$ can be computed in $O(|N_{\bar{H}}(v)|)$

time for each vertex v in U, as we show in the proof of the following lemma.

Lemma 4.1 *Running Algorithm Partition on all subgraphs H of a single iteration of Algorithm FMT requires a total of $O(n^2)$ time.*

Proof. First we prove that the running time of Algorithm Partition on input subgraph H is $O(|\bar{E}(H)| + |V(H)|)$, and then we will argue for the overall time bound at the end. Note that, as explained in the previous section, also for Algorithm Partition we will work on the complement graph $\bar{H}$ for an efficient implementation. Observe that between a connected component C and $U \setminus N_H[C]$, we have a complete bipartite graph in $\bar{H}$, meaning that no vertex of C is adjacent to any vertex of $U \setminus N_H[C]$ in H. These non-edges will be used as an argument to obtain the desired time bound.

The pseudocode of Algorithm Partition is presented in two bulks. Let us call the first bulk "defining P", and the second bulk "defining A".

The first operation in the "defining P" part is to unmark every vertex in H. The value $\mathcal{E}_{\bar{H}}(U \setminus N_H[u])$ for a single vertex u is computed straightforwardly by summing the degrees in the complement graph of all vertices in $U \setminus N_H[u] = N_{\bar{H}}(u)$, which is an $O(|N_{\bar{H}}(u)|)$ operation.

When a component C_k is created from a first vertex u we label every vertex $w \in N_{\bar{H}}(u)$ with the value $nk + |C_k| = nk + 1$. By labeling the vertices in this way, we assign a unique value to every vertex set that constitutes a component during the algorithm and ensure that only vertices in $U \setminus N_H[C_k]$ can have the label $nk + |C_k|$. The value $\mathcal{E}_{\bar{H}}(U \setminus N_H[C_k \cup \{v\}])$ can now be computed in $O(|N_{\bar{H}}(v)|)$ time, since the set of vertices in $N_{\bar{H}}(v)$ which are labeled $nk + |C_k|$ is exactly the set $U \setminus N[C_k \cup \{v\}]$. If v is going to be added to C_k then this increases the size of C_k by one and may affect the set $N[C_k]$. We update the labels of the vertices in $U \setminus N[C_k \cup \{v\}]$, by adding 1 to the label of every vertex in $N_{\bar{H}}(v)$ labeled with $nk + |C_k|$, and then add v to C_k. This requires $O(|N_{\bar{H}}(v)|)$ time for each vertex and $O(|\bar{E}(H)| + |V(H)|)$ in total for the "defining P" part, since every vertex is considered once and marked as a **p**, **c** or **s**-vertex. The **s**-vertices may be reconsidered once, and changed to **p**-vertices, but this does not affect the time complexity.

The "defining A" part consists of an if-else statement with 4 cases. In the first case we can do the required test by simply finding the largest neighborhood of a component and checking if its size is $|P|$. Without increasing the time complexity of the "defining P" part we can store the values $|C|$ and $|U \setminus N_H[C]|$ for each component C of $H(U \setminus P)$. Thus $|N_H(C)| = |U| - (|C| + |U \setminus N_H[C]|)$.

In the second case, we check every non-edge in $H(P)$, which is also an $O(|\bar{E}(H)| + |V(H)|)$ operation.

In the third case we will mark non-edges and components, as follows. For each **p**-vertex u, and then for each component C of $H(U \setminus P)$ where $C \subseteq N_{\bar{H}}(u)$, we mark C with the label u. We go through vertices in $N_{\bar{H}}(u)$, check which components they belong to, add up these numbers for each component, and check if it matches the total size of the component. Then for every **p**-vertex $v \in N_{\bar{H}}(u)$, where v is associated to a component labeled u, we add u to the label of non-edge uv. This takes $O(|N_{\bar{H}}(u)|)$ time for each **p**-vertex. The third case will now exist if and only if there is a non-edge uv marked by both u and v. Thus the total time for this case is $O(|\bar{E}(H)| + |V(H)|)$ for each subgraph H.

The fourth case requires constant time, and thus the total running time of Algorithm Partition on input subgraph H is $O(|\bar{E}(H)| + |V(H)|)$.

The operations that require $O(|\bar{E}(H)| + |V(H)|)$ on each subgraph H add up to $O(n^2)$ for all subgraphs of the same iteration of FMT, since they do not overlap in non-edges, and there are at most $O(n)$ such graphs by Lemma 3.2. Thus the total time complexity for all subgraphs H at the same iteration is $O(n^2)$. ■

We now give upper and lower bounds on the number of non-edges in various subgraphs of H related to vertex set P.

Lemma 4.2 *Let P be as computed by Algorithm Partition(H). Then each of the following is true:*

(i) $|\bar{E}(H(N_H[C]))| \leq \frac{4}{5}|\bar{E}(H)|$ *for each connected component C of $H(U \setminus P)$.*

(ii) $|\bar{E}(H(N_H[v]))| > \frac{3}{5}|\bar{E}(H)|$ *for each* **s**-*vertex v.*

(iii) $|\bar{E}(H(N_H[C \cup \{v\}]))| > \frac{3}{5}|\bar{E}(H)|$ *for each* **p**-*vertex v associated to C, where C is a connected component of $H(U \setminus P)$.*

Proof. *(i)* From Algorithm Partition we know that $\mathcal{E}_{\bar{H}}(U \setminus N_H[C]) \geq \frac{2}{5}|\bar{E}(H)|$ for each connected component C of $H(U \setminus P)$. Each non-edge uv outside of $H(N_H[C])$ contributes to the degree-sum $\mathcal{E}_{\bar{H}}(U \setminus N_H[C])$ by 1 if one of u or v is outside $N_H[C]$, and by 2 if both are outside. Thus there are at least $\frac{1}{5}|\bar{E}(H)|$ non-edges outside $H(N_H[C])$ and consequently at most $\frac{4}{5}|\bar{E}(H)|$ non-edges inside $H(N_H[C])$. Hence, $|\bar{E}(H(N_H[C]))| \leq \frac{4}{5}|\bar{E}(H)|$ for each connected component C of $H(U \setminus P)$, which completes the proof of *(i)*.

(ii) - (iii) From Algorithm Partition we know that $\mathcal{E}_{\bar{H}}(U \setminus N_H[v]) < \frac{2}{5}|\bar{E}(H)|$ for each **s**-vertex v, and $\mathcal{E}_{\bar{H}}(U \setminus N_H[C \cup \{u\}]) < \frac{2}{5}|\bar{E}(H)|$ for each **p**-vertex u associated to C. It follows by the same argument as case *(i)* that $|\bar{E}(H(N_H[v]))| > \frac{3}{5}|\bar{E}(H)|$ and $|\bar{E}(H(N_H[C \cup \{u\}]))| > \frac{3}{5}|\bar{E}(H)|$. This completes the proof of *(ii)* and *(iii)*. ■

We are now ready to prove the main result of this section, namely that the vertex set A returned by Partition results in subproblems of size bounded by a constant factor of the number of non-edges, given in Theorem 4.3.

Theorem 4.3 *Let A be the vertex set returned by Algorithm Partition on input $H = (U, D)$. Then both of the following are true, where G' is as defined in Algorithm FMT:*

(i) A is a proper subset of U such that either $A = N_H[K]$ where $K \subset U$ and $H(K)$ is connected, or A is a pmc of H.

(ii) Both the number of non-edges in $G'(A)$ and the number of non-edges in $G'(N_H[C])$ for each connected component C of $H(U \setminus A)$ are at most $\frac{4}{5}|\bar{E}(H)|$.

Proof. We will examine each of the 4 cases of the if-else statement in the "defining A" part of Algorithm Partition. We omit the subscript H in $N_H(C)$ and $N_H[C]$ to increase readability. The reader should keep in mind that throughout this proof we regard neighborhoods in H (and not in $\bar{H}$).

Case 1. $H(U \setminus P)$ has a full component C, i.e., $P = N(C)$.

This implies in particular that no vertices could have been marked as **s**-vertices. By Lemma 4.2 we know that the number of non-edges in $H(N[C_i])$ is less than $\frac{4}{5}|\bar{E}(H)|$ for each connected component C_i of $H(U \setminus P)$, in particular for C. In this case, Algorithm Partition gives $A = N[C]$, and thus $P \subset A$. C is a connected set since it was computed by adding new members from its neighborhood, and so *(i)* is satisfied. Observe that the connected components C_i of $H(U \setminus A)$ are exactly the connected components C_i of $H(U \setminus P)$, except C. It follows that the number of non-edges in $H(A) = H(N[C_i])$ and in $H(N[C_j])$ for each connected component C_j of $H(U \setminus A)$ is less than $\frac{4}{5}|\bar{E}(H)|$, already before the minimal separators are saturated. After the saturation, this number cannot increase but only decrease.

Case 2. There exist two vertices u, v, such that $uv \notin E(H)$, u is marked as an **s**-vertex, and v is marked as an **s**-vertex or a **p**-vertex.

We give the proof in two parts: the subcase where both u, v are **s**-vertices, and the subcase where u is an **s**-vertex and v a **p**-vertex. The arguments for the two subcases are very similar, and note that they are also very similar to the next *Case 3* where both u, v are **p**-vertices.

Assume both u and v are marked as **s**-vertices. By Lemma 4.2, $|\bar{E}(H(N[u]))| > \frac{3}{5}|\bar{E}(H)|$ and $|\bar{E}(H(N[v]))| > \frac{3}{5}|\bar{E}(H)|$, and thus for their common part we have $|\bar{E}(H(N(u) \cap N(v)))| = |\bar{E}(H(N[u] \cap N[v]))| > \frac{1}{5}|\bar{E}(H)|$, where the first equality holds since $u \notin N[v]$. The algorithm gives $A = N[u]$ in this case, satisfying *(i)*, which means that v will belong to a component C of $H(U \setminus A)$ with $N(C) \subseteq A$ thus being a u, v-separator. Since any u, v-separator must contain $N(u) \cap N(v)$, it follows that $N(C) \subseteq A$ induces at least $\frac{1}{5}|\bar{E}(H)|$ non-edges. All these non-edges will become edges and disappear from G'. Thus, there are at most $\frac{4}{5}|\bar{E}(H)|$ non-edges left that can appear in subproblems $G'(A)$ or $H(N[C_i])$ for a component C_i of $H(U \setminus A)$, thereby satisfying also *(ii)*.

Assume u is marked as an **s**-vertex, v is marked as a **p**-vertex and let j be the index such that v is associated to C_j. We know that such a C_j exists since v is marked as a **p**-vertex. An important observation now is that $u \notin N[C_j \cup \{v\}]$.

Otherwise u would have been marked as a **p**-vertex or **c**-vertex during execution of the inner while-loop in Algorithm Partition during computation of C_j. By Lemma 4.2, $|\bar{E}(H(N[u]))| > \frac{3}{5}|\bar{E}(H)|$ and $|\bar{E}(H(N[C_j \cup \{v\}]))| > \frac{3}{5}|\bar{E}(H)|$, and thus for their common part we have $|\bar{E}(H(N(u) \cap N(C_j \cup \{v\})))| = |\bar{E}(H(N[u] \cap N[C_j \cup \{v\}]))| > \frac{1}{5}|\bar{E}(H)|$, where the first equality holds since $u \notin N[C_j \cup \{v\}]$, as we established above. The algorithm gives $A = N[u]$ in this case, satisfying *(i)*, which means that $C_j \cup \{v\}$ will be contained in a component C of $H(U \setminus A)$ with $N(C) \subseteq A$ thus separating u from $C_j \cup \{v\}$. Since any such separator must contain $N(u) \cap N(C_j \cup \{v\})$, it follows that $N(C) \subseteq A$ induces at least $\frac{1}{5}|\bar{E}(H)|$ non-edges. All these non-edges will become edges and disappear from G'. Thus, there are at most $\frac{4}{5}|\bar{E}(H)|$ non-edges left that can appear in subproblems $G'(A)$ or $H(N[C_i])$ for a component C_i of $H(U \setminus A)$, thereby satisfying also *(ii)*.

Case 3. There exist two vertices u, v marked as **p**-vertices, such that $uv \notin E(H)$, u is associated to C_i, v is associated to C_j, $u \notin N(C_j)$ and $v \notin N(C_i)$.

The important observation now is that there are no edges between $C_i \cup \{u\}$ and $C_j \cup \{v\}$. By Lemma 4.2, $|\bar{E}(H(N[C_i \cup \{u\}]))| > \frac{3}{5}|\bar{E}(H)|$ and $|\bar{E}(H(N[C_j \cup \{v\}]))| > \frac{3}{5}|\bar{E}(H)|$, and thus for their common part we have $|\bar{E}(H(N(C_i \cup \{u\}) \cap N(C_j \cup \{v\})))| = |\bar{E}(H(N[C_i \cup \{u\}] \cap N[C_j \cup \{v\}]))| > \frac{1}{5}|\bar{E}(H)|$, where the first equality holds since there are no edges between $C_i \cup \{u\}$ and $C_j \cup \{v\}$. The algorithm gives $A = N[C_i \cup \{u\}]$ in this case, satisfying *(i)*, which means that $C_j \cup \{v\}$ will be contained in a component C of $H(U \setminus A)$ with $N(C) \subseteq A$ thus separating $C_i \cup \{u\}$ from $C_j \cup \{v\}$. Since any such separator must contain $N(C_i \cup \{u\}) \cap N(C_j \cup \{v\})$, it follows that $N(C) \subseteq A$ induces at least $\frac{1}{5}|\bar{E}(H)|$ non-edges. All these non-edges will become edges and disappear from G'. Thus, there are at most $\frac{4}{5}|\bar{E}(H)|$ non-edges left that can appear in subproblems $G'(A)$ or $H(N[C_i])$ for a component C_i of $H(U \setminus A)$, thereby satisfying also *(ii)*.

Case 4. None of the above cases apply.

First we show that P is a pmc of H in this case. Due to Theorem 2.1, all we have to show is that if none of the Cases 1, 2, and 3 applies, then $H(U \setminus P)$ has no full component associated to P, and for every pair of non-adjacent vertices $u, v \in P$ there is a connected component C of $H(U \setminus P)$ such that $u, v \in N(C)$. Since Case 1 does not apply, we know that $H(U \setminus P)$ has no full components. Since Case 2 does not apply either, then the **s**-vertices altogether induce a clique and they all have edges to all **p**-vertices. So, since P consists only of **p** and **s** vertices, the only non-edges that are possible within P are those non-edges uv where both u and v are **p**-vertices. Since Case 3 does not apply either, then for any non-adjacent $u, v \in P$, if they are not associated to the same component then one of them must be in the neighborhood of the component that the other one is associated to. Thus P is a pmc of H, and *(i)* is satisfied since Algorithm Partition gives $A = P$ in this case. In this case, whole A is saturated in G', and thus $G'(A)$ has no non-edges. The remaining subproblems will each have at most $\frac{4}{5}|\bar{E}(H)|$ non-edges by Lemma 4.2, since the connected components of $H(U \setminus A)$ are the same as the connected components of $H(U \setminus P)$. ∎

5 The total $O(n^\alpha \log n)$ time complexity

Theorem 5.1 *Algorithm FMT described in Section 3, using Algorithm Partition described in Section 4, computes a minimal triangulation of the input graph in $O(n^\alpha \log n)$ time.*

Proof. By Lemma 3.1 and Theorem 4.3*(i)*, Algorithm FMT computes a minimal triangulation. By Lemma 3.2, the matrix multiplication at each iteration of FMT requires $O(n^\alpha)$ time. By the discussion that follows Lemma 3.2 in Section 3, all other operations outside of Algorithm Partition can be performed in $O(n^2)$ time at each iteration of FMT. Using Lemma 4.1, we conclude that the total time required at each iteration of FMT is $O(n^\alpha)$, since $\alpha \geq 2$ for any matrix multiplication algorithm. By Theorem 4.3*(ii)*, the number of non-edges in each subproblem decreases by a constant factor for each iteration, and since subproblems in one iteration do not overlap in non-edges we can at most have $\log n^2 = O(\log n)$ iterations of FMT. ∎

We have thus given the details of a new algorithm to compute minimal triangulations of arbitrary graphs in $O(n^\alpha \log n)$ time. It is important to use a matrix multiplication algorithm with running time $o(n^3)$ to achieve an improvement compared to existing minimal triangulation algorithms, thus standard matrix multiplication is not interesting. If we use the matrix multiplication algorithm of Coppersmith and Winograd [6], then α is strictly less than 2.376, and thus the total running time of our algorithm becomes $o(n^{2.376})$. If we instead use the matrix multiplication algorithm of Strassen [18] which has a worse asymptotic time bound of $\Theta(n^{\log_2 7}) = o(n^{2.81})$ but is considered more practical due to large constants in [6], then our time bound becomes $O(n^{\log_2 7} \log n) = o(n^{2.81})$. Using Strassen's algorithm, the time bound claimed by Kratsch and Spinrad mentioned previously becomes $O(n^{2.91})$ [12]. In fact, our algorithm is asymptotically faster than theirs regardless of the matrix multiplication algorithm used.

References

[1] A. Berry, J-P. Bordat, and P. Heggernes. Recognizing weakly triangulated graphs by edge separability. *Nordic Journal of Computing*, 7:164–177, 2000.

[2] A. Berry, J.P. Bordat, P. Heggernes, G. Simonet, and Y. Villanger. A wide-range algorithm for minimal triangulation from an arbitrary ordering. *Journal of Algorithms*, 58(1):33–66, 2006.

A. Berry, J-P. Bordat, P. Heggernes, G. Simonet, and Y. Villanger. A wide-range algorithm for minimal triangulation from an arbitrary ordering. *J. Algorithms.* To appear.

[3] A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for dynamically maintaining chordal graphs. In *Algorithms and Computation - ISAAC 2003*, pages 47 – 57. Springer Verlag, 2003. LNCS 2906.

[4] J. R. S. Blair, P. Heggernes, and J. A. Telle. A practical algorithm for making filled graphs minimal. *Theor. Comput. Sci.*, 250:125–141, 2001.

[5] V. Bouchitté and I. Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM J. Comput.*, 31:212–232, 2001.

[6] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comp.*, 9:1–6, 1990.

[7] E. Dahlhaus. Minimal elimination ordering inside a given chordal graph. In *Graph Theoretical Concepts in Computer Science - WG '97*, pages 132–143. Springer Verlag, 1997. LNCS 1335.

[8] P. Heggernes and Y. Villanger. Efficient implementation of a minimal triangulation algorithm. In *Algorithms - ESA 2002*, pages 550–561. Springer Verlag, 2002. LNCS 2461.

[9] D. Hudson, S. Nettles, and T. Warnow. Obtaining highly accurate topology estimates of evolutionary trees from very short sequences. In *Proceedings of RECOMB'99*, pages 198–207. 1999.

[10] T. Kloks, D. Kratsch, and J. Spinrad. On treewidth and minimum fill-in of asteroidal triple-free graphs. *Theor. Comput. Sci.*, 175:309–335, 1997.

[11] D. Kratsch and J. Spinrad. Between $O(nm)$ and $O(n^\alpha)$. In *Proceedings of the* 14*th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 709–716, 2003.

[12] D. Kratsch and J. Spinrad. Minimal fill in $o(n^3)$ time. 2004. Submitted.

[13] T. Ohtsuki. A fast algorithm for finding an optimal ordering in the vertex elimination on a graph. *SIAM J. Comput.*, 5:133–145, 1976.

[14] T. Ohtsuki, L. K. Cheung, and T. Fujisawa. Minimal triangulation of a graph and optimal pivoting ordering in a sparse matrix. *J. Math. Anal. Appl.*, 54:622–633, 1976.

[15] A. Parra and P. Scheffler. Characterizations and algorithmic applications of chordal graph embeddings. *Disc. Appl. Math.*, 79:171–188, 1997.

[16] B. W. Peyton. Minimal orderings revisited. *SIAM J. Matrix Anal. Appl.*, 23(1):271–294, 2001.

[17] D. Rose, R.E. Tarjan, and G. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:146–160, 1976.

[18] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14:354–356, 1969.

[19] J. van Leeuwen. Graph algorithms. In *Handbook of Theoretical Computer Science, A: Algorithms and Complexity Theory*. North Holland, 1990.

Paper V

Exact algorithms for treewidth and minimum fill-in[*][†]

Fedor V. Fomin[‡]
Department of Informatics
University of Bergen
5020 Bergen, Norway
fomin@ii.uib.no

Dieter Kratsch
LITA, Université de Metz
57045 Metz Cedex 01, France
kratsch@univ-metz.fr

Ioan Todinca
LIFO, Université d'Orléans
45067 Orléans Cedex 2, France
Ioan.Todinca@lifo.univ-orleans.fr

Yngve Villanger
Department of Informatics
University of Bergen
5020 Bergen, Norway
yngvev@ii.uib.no

Abstract

We show that there are $\mathcal{O}(1.8899^n)$ time algorithms to compute the treewidth and the minimum fill-in of each graph G on n vertices. Our result is based on a combinatorial proof that each graph on n vertices has at most $n \cdot 1.7087^n$ minimal separators and that all potential maximal cliques can be listed in $\mathcal{O}(1.8899^n)$ time. For the class of AT-free graphs we obtain $\mathcal{O}(1.4142^n)$ time algorithms to compute treewidth and minimum fill-in.

Keywords: Exact exponential algorithm, treewidth, fill-in, minimal separators, potential maximal clique, minimal triangulation

[*]A preliminary version [26] of this paper has been presented at the 31st International Colloquium on Automata, Languages and Programming, Turku, Finland, July 2004.

[†]Supported by The Aurora Programme Collaboration Research Project between Norway and France.

[‡]Fedor Fomin acknowledges support of Norges forskningsråd, project 160778/V30.

1 Introduction

Exact exponential algorithms. The interest in exact (fast) exponential algorithms dates back to Held and Karp's paper [29] on the travelling salesman problem in the early sixties. Mention just a few examples: time $\mathcal{O}^*(1.4422^n)$ algorithm for Knapsack (Horowitz and Sahni [30]); time $\mathcal{O}^*(1.2600^n)$ and $\mathcal{O}^*(1.2109^n)$ algorithms for Independent Set (Tarjan and Trojanowski [44], Robson [40]); 3-Coloring in time $\mathcal{O}^*(1.4422^n)$ (Lawler [34]); 3-SAT in time $\mathcal{O}^*(1.6181^n)$ (Monien and Speckenmeyer [35]).

In this paper we use a modified big-Oh notation that suppresses all other (polynomially bounded) terms. For functions f and g we write $f(n) = \mathcal{O}^*(g(n))$ if $f(n) = \mathcal{O}(g(n)poly(n))$, where $poly(n)$ is a polynomial. This modification may be justified by the exponential growth of $f(n)$.

Nowadays, it is common believe that NP-hard problems can not be solved in polynomial time. For a number of NP-hard problems, we even have strong evidence that they cannot be solved in sub-exponential time. In order to obtain exact solutions to these problems, the only hope is to design exact algorithms with good exponential running times. How good can these exponential running times be? Can we reach 2^{n^2} for instances of size n? Can we reach 10^n? Or even 2^n? Or can we reach c^n for some constant c that is very close to 1? The last years have seen an emerging interest in attacking these questions for concrete combinatorial problems: There is an $\mathcal{O}^*(2.4150^n)$ time algorithm for Coloring (Byskov [15]); an $\mathcal{O}^*(1.3289^n)$ time algorithm for 3-Coloring (Beigel and Eppstein [3]); an $\mathcal{O}^*(1.7325^n)$ time algorithm for Max-Cut (Williams [47]); an algorithms for 3-SAT in time $\mathcal{O}^*(1.4726^n)$ (Brueggemann and Kern [14]); an $\mathcal{O}^*(1.5129^n)$ time algorithm for Dominating Set (Fomin et al. [25]).

There can be several explanations why now the algorithmic community witnesses the revival of the interest in fast exponential algorithms:

- The design and analysis of exact algorithms leads to a better understanding of NP-hard problems and initiates interesting new combinatorial and algorithmic challenges.
- For certain applications it is important to find exact solutions. With the increased speed of modern computers, fast algorithms, even though they have exponential running times in the worst case, may actually lead to practical algorithms for certain NP-hard problems, at least for moderate instance sizes.
- Approximation, randomized algorithms and different heuristics are not always satisfactory. Each of these approaches has weak points like necessity of exact solutions, difficulty of approximation, limited power of the method itself and many others.

- A reduction of the base of the exponential running time, say from $\mathcal{O}(2^n)$ to $\mathcal{O}(1.8^n)$, increases the size of the instances solvable within a given amount of time by a constant *multiplicative* factor. However running a given exponential algorithm on a faster computer can enlarge the mentioned size only by a constant *additive* factor.

For overviews and introductions to the field see the recent surveys by Iwama [31], Schöning [42], and Woeginger [48, 49].

Treewidth and minimum fill-in. Treewidth is one of the most basic parameters in Graph Algorithms [6] and it plays an important role in structural Graph Theory. It serves as one of the main tools in Robertson and Seymour's Graph Minors project [39]. Treewidth also plays a crucial role in parameterized complexity theory [21]. The minimum fill-in problem (also known as minimum chordal graph completion) has important applications in sparse matrix computations and computational biology.

The problems to compute the treewidth and minimum fill-in of a graph are known to be NP-hard even when the input is restricted to complements of bipartite graphs (so called cobipartite graphs) [2, 50]. Despite of the importance of treewidth almost nothing is known on how to cope with its intractability. For a long time the best known approximation algorithm for treewidth had a factor $\log OPT$ [1, 11] (see also [7]). Recently, Feige et al. [23] obtained factor $\sqrt{\log OPT}$ approximation algorithm for treewidth. Furthermore it is an old open question whether the treewidth can be approximated within a constant factor.

Treewidth is known to be fixed parameter tractable. Moreover, for any fixed k, there is a linear time algorithm to compute the treewidth of graphs of treewidth at most k (unfortunately there is a huge hidden constant in the running time) [5]. There is a number of algorithms that for a given graph G and integer k, either report that the treewidth of G is at least k, or produce a tree decomposition of width at most $c_1 \cdot k$ in time ${c_2}^k \cdot n^{O(1)}$, where c_1, c_2 are some constants (see e.g. [1]). Fixed parameter algorithms are known for the minimum fill-in problem as well [17, 32].

We are not aware about any previous work on exact algorithms for the treewidth or minimum fill-in problem. There are three relatively simple approaches resulting in time $\mathcal{O}^*(2^n)$ algorithms:

- One can reformulate the problems as problems of finding special vertex elimination orderings and then find an optimal permutation by using the dynamic programming based technique like in the article of Held & Karp [29] for the travelling salesman problem;

- With some modifications, the algorithm of Arnborg et al. [2] for a given k deciding in time $O(n^{k+1})$ if the treewidth of a graph is at most k, can be used to compute the treewidth (and similarly fill-in) in time $\mathcal{O}^*(2^n)$;

- Both problems can be solved by making use of game theoretic approach, by finding a specific path in the graph of possible states of Cop and Robber game [24].

However it is not clear if any of the mentioned approaches can bring us to time $\mathcal{O}^*(c^n)$ algorithm for some $c < 2$.

Our results. In this paper we obtain the first exact algorithm computing the treewidth in time $\mathcal{O}^*(c^n)$ for $c < 2$. Additionally it can be adapted to solve a number of other minimal triangulation problems like minimum fill-in.

Our main result is an $\mathcal{O}^*(1.8899^n)$ algorithm computing the treewidth and minimum fill-in of a graph on n vertices. The algorithm can be regarded as dynamic programming across partial solutions and is based on results of Bouchitté & Todinca [9, 10]. The analysis of the running time is difficult and is based on combinatorial properties of special structures in a graph, namely, potential maximal clique, i.e. vertex subsets in a graph that can be maximal cliques in some minimal triangulation of this graph. (See the next section for the definition.)

More precisely, first we modify the algorithm of Bouchitté & Todinca [9] which computes the treewidth and minimum fill-in of a graph G with the given set Π_G of all potential maximal cliques of G and then improve the analysis of its running time to obtain an $\mathcal{O}^*(|\Pi_G|)$ time complexity. The most technical and difficult part of the paper is the proof that all potential maximal cliques can be listed in time $\mathcal{O}^*(1.8899^n)$. Very roughly, our listing algorithms is based on the following combinatorial result: every "large" potential maximal clique is either "almost" a minimal separator, or can be represented by a "small" vertex subset. The technique developed to prove this combinatorial result can be interesting on its own.

For several special graph classes, for which both problems remain NP-complete, we are able to prove that our approach guarantees significantly better bounds. To exemplify this we show that for the class of AT-free graphs the number of minimal separators and the number of potential maximal cliques, and thus the running time of our algorithm, is $\mathcal{O}^*(2^{n/2})$.

This paper is organized as follows. In Section 2 we give basic definitions. In Section 3 we show how Bouchitté & Todinca's approach can be used to compute the treewidth and fill-in in time linear in the number of potential maximal cliques. In Section 4 we prove that every graph on n vertices has $\mathcal{O}(n \cdot 1.7087^n)$ minimal separators. The results of Section 4 are used in Section 5, where we derive the most difficult and important combinatorial result of this paper, namely, that all potential maximal cliques of a graph can be listed in time $\mathcal{O}^*(1.8899^n)$. Combining with the results from Section 3, this yields the main result of the paper, that the treewidth and minimum fill-in can be computed in time $\mathcal{O}^*(1.8899^n)$. In

Section 6 we design faster $\mathcal{O}^*(2^{n/2})$ algorithm on AT-free graphs. We conclude with open problems and final remarks in Section 7.

2 Basic definitions

We denote by $G = (V, E)$ a finite, undirected and simple graph with $|V| = n$ vertices and $|E| = m$ edges. For any non-empty subset $W \subseteq V$, the subgraph of G induced by W is denoted by $G[W]$. For $S \subseteq V$ we often use $G \setminus S$ to denote $G[V \setminus S]$. The *neighborhood* of a vertex v is $N(v) = \{u \in V : \{u, v\} \in E\}$ and for a vertex set $S \subseteq V$ we set $N(S) = \bigcup_{v \in S} N(v) \setminus S$. A *clique* C of a graph G is a subset of V such that all the vertices of C are pairwise adjacent. By $\omega(G)$ we denote the maximum clique-size of a graph G.

Treewidth and minimum fill-in of graphs. The notion of treewidth is due to Robertson & Seymour [38]. A *tree decomposition* of a graph $G = (V, E)$, denoted by $TD(G)$, is a pair (X, T) in which $T = (V_T, E_T)$ is a tree and $X = \{X_i | i \in V_T\}$ is a family of subsets of V such that:

(i) $\bigcup_{i \in V_T} X_i = V$;

(ii) for each edge $e = \{u, v\} \in E$ there exists an $i \in V_T$ such that both u and v belong to X_i; and

(iii) for all $v \in V$, the set of nodes $\{i \in V_T | v \in X_i\}$ induces a connected subtree of T.

The maximum of $|X_i| - 1$, $i \in V_T$, is called the *width* of the tree decomposition. The *treewidth* of a graph G, denoted by $\mathrm{tw}(G)$, is the minimum width taken over all tree decompositions of G.

A graph H is *chordal* (or *triangulated*) if every cycle of length at least four has a chord, i.e. an edge between two non-consecutive vertices of the cycle. A *triangulation* of a graph $G = (V, E)$ is a chordal graph $H = (V, E')$ such that $E \subseteq E'$. H is a *minimal triangulation* if for any intermediate set E'' with $E \subseteq E'' \subset E'$, the graph $F = (V, E'')$ is not chordal.

The following result is very useful for our algorithms.

Theorem 1 (Folklore). *For any graph G, $\mathrm{tw}(G) \leq k$ if and only if there is a triangulation H of G such that $\omega(H) \leq k + 1$.*

Thus the treewidth of a graph G can be defined as the minimum of $\omega(H) - 1$ taken over all triangulations H of G, of $\omega(H) - 1$.

The *minimum fill-in* of a graph $G = (V, E)$, denoted by $\mathrm{mfi}(G)$, is the smallest value of $|E_H - E|$, where the minimum is taken over all triangulations $H = (V, E_H)$ of G.

In other words, computing the treewidth of G means finding a (minimal) triangulation with the smallest maximum clique-size, while computing the minimum fill-in means finding a (minimal) triangulation with the smallest number of edges. Clearly, in both cases it is sufficient to consider only minimal triangulations of G, which makes minimal separators and potential maximal cliques important tools of our algorithmic approach.

Minimal separators. Minimal separators and potential maximal cliques are the most important tools used in our proofs. Let a and b be two non adjacent vertices of a graph $G = (V, E)$. A set of vertices $S \subseteq V$ is an *a, b-separator* if a and b are in different connected components of the graph $G \setminus S$. A connected component C of $G \setminus S$ is a *full* component (associated to S) if $N(C) = S$. S is a *minimal a, b-separator* of G if no proper subset of S is an a, b-separator. We say that S is a *minimal separator* of G if there are two vertices a and b such that S is a minimal a, b-separator. Notice that a minimal separator can be strictly included in another one. We denote by Δ_G the set of all minimal separators of G. A set of vertices $\Omega \subseteq V$ of a graph G is called a *potential maximal clique* if there is a minimal triangulation H of G such that Ω is a maximal clique of H. We denote by Π_G the set of all potential maximal cliques of G. Clearly, $|\Delta_G| \leq 2^n$ and $|\Pi_G| \leq 2^n$ for every graph G on n vertices, and no better upper bounds had been known prior to our work.

The following result will be used to list all minimal separators of a graph.

Theorem 2 ([4]). *There is an algorithm listing all minimal separators of an input graph G in $\mathcal{O}(n^3|\Delta_G|)$ time.*

There is a very useful relationship between the minimal separators of a graph and its minimal triangulations. Two minimal separators S and T of a graph G are said to be *crossing* if S is a minimal u, v-separator for a pair of vertices $u, v \in T$, in which case T is a minimal x, y-separator for a pair $x, y \in S$.

Theorem 3 ([37]). *The graph H is a minimal triangulation of the graph G if and only if there is a maximal set of pairwise non-crossing minimal separators $\{S_1, S_2, \ldots, S_p\}$ of G such that H can be obtained from G by completing each S_i, $i \in \{1, 2, \ldots, p\}$, into a clique.*

Although we do not use this characterization explicitly it is fundamental for our paper.

Potential maximal cliques. The following structural characterization of potential maximal cliques is extremely useful for our purposes.

For a set $K \subseteq V$, a connected component C of $G \setminus K$ is a *full component associated to K* if $N(C) = K$.

Theorem 4 ([9]). *Let $K \subseteq V$ be a set of vertices of the graph $G = (V, E)$. Let $\mathcal{C}(K) = \{C_1(K), \dots, C_p(K)\}$ be the set of the connected components of $G \setminus K$ and let $\mathcal{S}(K) = \{S_1(K), S_2(K), \dots, S_p(K)\}$ where $S_i(K)$, $i \in \{1, 2, \dots, p\}$, is the set of those vertices of K which are adjacent to at least one vertex of the component $C_i(K)$. Then K is a potential maximal clique of G if and only if :*

1. *$G \setminus K$ has no full component associated to K, and*
2. *the graph on the vertex set K obtained from $G[K]$ by completing each $S_i \in \mathcal{S}(K)$ into a clique, is a complete graph.*

Moreover, if K is a potential maximal clique, then $\mathcal{S}(K)$ is the set of the minimal separators of G contained in K.

Remark. By Theorem 4, for every potential maximal clique Ω of G, the sets $S_i(\Omega)$ are exactly the minimal separators of G contained in Ω. Let us point out that for each minimal separator $S_i = S_i(\Omega)$, all vertices of $\Omega \setminus S_i$ are contained in the same component of $G \setminus S_i$.

The following result is an easy consequence of Theorem 4.

Theorem 5 ([9]). *There is an algorithm that, given a graph $G = (V, E)$ and a set of vertices $K \subseteq V$, verifies if K is a potential maximal clique of G. The time complexity of the algorithm is $\mathcal{O}(nm)$.*

According to [10], the number of potential maximal cliques of a graph G is at least $|\Delta_G|/n$ and at most $n|\Delta_G|^2 + n|\Delta_G| + 1$. We will show later that a graph on n vertices has $\mathcal{O}^*(1.7087^n)$ minimal separators and $\mathcal{O}^*(1.8899^n)$ potential maximal cliques.

Let us emphasize that it is an open question whether there is an algorithm listing all potential maximal cliques of any graph with a running time $\mathcal{O}(\text{poly}(n) \cdot |\Pi_G|)$ for some polynomial poly(n).

3 Computing treewidth and minimum fill-in

We describe a modification of the algorithm of [9] that, given a graph, all its minimal separators and all its potential maximal cliques, computes the treewidth and the minimum fill-in of the graph. The running time stated in [9] could be reformulated as $\mathcal{O}(n^2\, |\Delta_G| \cdot |\Pi_G|)$. We show how the algorithm can be implemented to run in time $\mathcal{O}(n^3 \cdot |\Pi_G|)$.

For a minimal separator S and a component $C \in \mathcal{C}(S)$ of $G \setminus S$, we say that (S, C) is a *block* associated to S. We sometimes use the notation (S, C) to denote the set of vertices $S \cup C$ of the block. It is easy to notice that if $X \subseteq V$

corresponds the set of vertices of a block, then this block (S, C) is unique: indeed $S = N(V \setminus X)$ and $C = X \setminus S$.

A block (S, C) is called *full* if C is a full component associated to S. The graph $R(S, C) = G_S[S \cup C]$ obtained from $G[S \cup C]$ by completing S into a clique is called the *realization* of the block B.

Theorem 6 ([33]). *Let G be a non-complete graph. Then*

$$\operatorname{tw}(G) = \min_{S \in \Delta_G} \max_{C \in \mathcal{C}(S)} \operatorname{tw}(R(S, C))$$

$$\operatorname{mfi}(G) = \min_{S \in \Delta_G} (\operatorname{fill}(S) + \sum_{C \in \mathcal{C}(S)} \operatorname{mfi}(R(S, C)))$$

where $\operatorname{fill}(S)$ *is the number of non-edges of* $G[S]$.

Remark. In the equations of Theorem 6 we may take the minimum only over the inclusion-minimal separators of G. Then all the blocks in the equations are full.

Unfortunately, Theorem 6 is not sufficient for computing the treewidth and the minimum fill-in. Therefore we now express the treewidth and the minimum fill-in of realizations of full blocks from realizations of smaller full blocks. Let Ω be a potential maximal clique of G. We say that a block (S', C') is *associated to* Ω if C' is a component of $G \setminus \Omega$ and $S' = N(C')$.

Theorem 7 ([9]). *Let (S, C) be a full block of G. Then*

$$\operatorname{tw}(R(S, C)) = \min_{S \subset \Omega \subseteq (S,C)} \max(|\Omega| - 1, \operatorname{tw}(R(S_i, C_i)))$$

$$\operatorname{mfi}(R(S, C)) = \min_{S \subset \Omega \subseteq (S,C)} \Big(\operatorname{fill}(\Omega) - \operatorname{fill}(S) + \sum \operatorname{mfi}(R(S_i, C_i))\Big)$$

where the minimum is taken over all potential maximal cliques Ω such that $S \subset \Omega \subseteq (S, C)$ and (S_i, C_i) are the blocks associated to Ω in G such that $S_i \cup C_i \subset S \cup C$.

Theorem 8. *There is an algorithm that, given a graph G together with the list of its minimal separators Δ_G and the list of its potential maximal cliques Π_G, computes the treewidth and the minimum fill-in of G in $\mathcal{O}(n^3 |\Pi_G|)$ time. Moreover, the algorithm constructs optimal triangulations for the treewidth and the minimum fill-in.*

Proof. W.l.o.g. we may assume that the input graph G is connected (otherwise we can run the algorithm for each connected component of G).

The algorithm for computing the treewidth and the minimum fill-in of a graph, using its minimal separators and its potential maximal cliques is given below. It is a slightly different version of the algorithm given in [9].

```
Input: $G$, all its potential maximal cliques and all its minimal separators
Output: $\text{tw}(G)$ and $\text{mfi}(G)$
begin
    compute all the full blocks $(S, C)$ and sort them by the number of vertices
    for each full block $(S, C)$ taken in increasing order
        $\text{tw}(R(S, C)) := |S \cup C| - 1$ if $(S, C)$ is inclusion-minimal
            and $\text{tw}(R(S, C)) := \infty$ otherwise
        $\text{mfi}(R(S, C)) := \text{fill}(S \cup C)$ if $(S, C)$ is inclusion-minimal
            and $\text{mfi}(R(S, C)) := \infty$ otherwise
        for each p.m.c. $\Omega$ with $S \subset \Omega \subseteq (S, C)$
            compute the blocks $(S_i, C_i)$ associated to $\Omega$ s.t. $S_i \cup C_i \subset S \cup C$
            $\text{tw}(R(S, C)) := \min(\text{tw}(R(S, C)),$
                        $\max_i(|\Omega| - 1, \text{tw}(R(S_i, C_i))))$
            $\text{mfi}(R(S, C)) := \min(\text{mfi}(R(S, C)),$
                        $\text{fill}(\Omega) - \text{fill}(S) + \sum_i(\text{mfi}(R(S_i, C_i))))$
        end_for
    end_for
    let $\Delta_G^*$ be the set of inclusion-minimal separators of $G$
    $\text{tw}(G) := \min_{S \in \Delta_G^*} \max_{C \in \mathcal{C}(S)} \text{tw}(R(S, C))$
    $\text{mfi}(G) := \min_{S \in \Delta_G^*} (\text{fill}(S) + \sum_{C \in \mathcal{C}(S)} \text{mfi}(R(S, C)))$
end
```

For the sake of completeness we shortly discuss the correctness proof. According to Theorem 7, at the end of the outer `for` loop the values of $\text{tw}(R(S, C))$ and $\text{mfi}(R(S, C))$ are correctly computed, for each full block (S, C) of G. Then the treewidth and the minimum fill-in of the graph are computed using Theorem 6 and the fact that in Theorem 6 one can work only with inclusion-minimal separators.

Let us show how the algorithm can be implemented in time $\mathcal{O}(n^3 \cdot |\Pi_G|)$.

To store and manipulate the minimal separators, potential maximal cliques and blocks we use data structures that allow to search, to insert or to check whether an element is inclusion-minimal in $\mathcal{O}(n)$ time.

During a preprocessing step, we realize the following operations.

- Compute the list of all full blocks and, for each minimal separator S, store a pointer towards each full block of type (S, C). These operations are performed as follows. For each minimal separator S, we compute the connected components of $G \setminus S$. For each such component C, if $N(C) = S$ then the block (S, C) is full, so we add it to the list of full blocks and store the pointer from S to (S, C). Note that this procedure will generate all the full blocks, and each of them is encountered exactly once. For a given minimal separator S, there are at most n full blocks associated to it, so at most n pointers to be stored. The insertion of these blocks into the list of full blocks requires $\mathcal{O}(n)$ time for each block. Hence the whole step costs $\mathcal{O}(n^2|\Delta_G|)$ time.

- For each potential maximal clique Ω, store a pointer to each full block associated to it as follows: compute the components C_i of $G \setminus \Omega$ and then $(N(C_i), C_i)$ are precisely the blocks associated to Ω. In particular there are at most n such blocks. This computation can be done in $\mathcal{O}(n^2)$ time for each potential maximal clique, so globally in $\mathcal{O}(n^2|\Pi_G|)$ time.

- Compute all the *good triples* (S, C, Ω), where (S, C) is a full block and Ω is a potential maximal clique such that $S \subset \Omega \subseteq S \cup C$. Moreover, for each good triple we store a pointer from (S, C) to Ω. By Theorem 4, there are at most n minimal separators $S \subset \Omega$ each of them being the neighborhood of a component of $G \setminus \Omega$ and for each such S there is exactly one component $G \setminus S$ intersecting Ω (in particular there are at most $n|\Pi_G|$ good triples). For each component C' of $G \setminus \Omega$ we take $S = N(C')$, find the component C of $G \setminus S$ intersecting Ω and store the pointer from (S, C) to Ω. Thus this computation takes $\mathcal{O}(nm)$ time for each potential maximal clique, so $\mathcal{O}(nm|\Pi_G|)$ globally.

Hence this preprocessing step costs $\mathcal{O}(n^2|\Delta_G| + nm|\Pi_G|)$. Sorting the blocks by their size can be done in $\mathcal{O}(n|\Delta_G|)$ time using a bucket sort.

Observe that the cost of one iteration of the inner `for` loop is $\mathcal{O}(n^2)$, by the fact that there are at most n blocks associated to a potential maximal clique. With the data structures obtained during the preprocessing step, each full block (S, C) keeps a pointer towards each potential maximal clique Ω such that (S, C, Ω) form a good triple. Thus the number of iterations of the two nested loops is exactly the number of good triples, that is at most $n|\Pi_G|$. It follows that the two loops cost $\mathcal{O}(n^3|\Pi_G|)$ time.

After the execution of the loops, computing the set Δ_G^* of inclusion-minimal separators costs $\mathcal{O}(n|\Delta_G|)$ time. Each inclusion-minimal separator S keeps the list of its associated blocks, obtained during the preprocessing step. Computing the maximum required by the two last instructions costs $\mathcal{O}(n)$ time for a given S. This last step costs $\mathcal{O}(n|\Delta_G|)$ time.

Altogether, the algorithm runs in time $\mathcal{O}(n^2 \cdot |\Delta_G| + n^3 \cdot |\Pi_G|)$. It is known [10] that each minimal separator is contained in at least one potential maximal clique. According to Theorem 4, each potential maximal clique contains at most n minimal separators. Therefore $|\Pi_G| \geq |\Delta_G|/n$. We conclude that the algorithm runs in $\mathcal{O}(n^3 \cdot |\Pi_G|)$ time.

The algorithm can be easily transformed in order to output not only the treewidth and the minimum fill-in of the graph, but also optimal triangulations with respect to these parameters. It is sufficient to keep, for each full block, the set of potential maximal cliques realizing the minimum treewidth and fill-in of its realization. At the end of the algorithm, the potential maximal cliques of the chosen blocks will be the maximal cliques of the computed optimal triangulation: optimal tree decomposition or minimum triangulation. □

Using Theorem 8, the only missing ingredient of our treewidth and minimum fill-in algorithms is an algorithm listing all (minimal separators and) potential maximal cliques of a graph in time $\mathcal{O}^*(c^n)$ for some $c < 2$. This requires exponential upper bounds of the type $\mathcal{O}^*(c^n)$ for some $c < 2$ for the number of minimal separators and for the number of potential maximal cliques in a graph on n vertices. In the next two sections we discuss this issue.

4 Upper bounding the number of minimal separators

In this section we show that any graph with n vertices has $\mathcal{O}(n \cdot 1.7087^n)$ minimal separators. For the main algorithm of this paper the upper bound $\mathcal{O}^*(1.8899^n)$ would be sufficient. However, bounding the number of minimal separators in a graph is a nice combinatorial problem and we prefer to give here the best upper bound we were able to find.

Let S be a separator in a graph $G = (V, E)$. For $x \in V \setminus S$, we denote by $C_x(S)$ the component of $G \setminus S$ containing x. The following lemma is an exercise in [28].

Lemma 9 (Folklore). *A set S of vertices of G is a minimal a, b-separator if and only if a and b are in different full components associated to S. In particular, S is a minimal separator if and only if there are at least two distinct full components associated to S.*

Here is the main combinatorial result.

Theorem 10. *For any graph G, $|\Delta_G| = \mathcal{O}(n \cdot 1.7087^n)$.*

Let us note, that by Theorem 2, Theorem 10 yields that all minimal separators of a graph can be listed in time $\mathcal{O}(n^4 \cdot 1.7087^n)$.

Proof. For a constant α, $0 < \alpha < 1$, we distinguish two types of minimal separators: *small* separators, of size at most αn, and *big* separators, of size more than αn. We denote the cardinalities of these sets by $\#\texttt{small sep}$ and $\#\texttt{big sep}$. Notice that $|\Delta_G| = \#\texttt{small sep} + \#\texttt{big sep}$.

4.1 Upper bounding the number of big separators

Let S be a minimal separator. By Lemma 9, there are at least two full components associated to S. Hence at least one of these full components has at most $n(1-\alpha)/2$ vertices. For every $S \in \Delta_G$ we choose one of these full components, and call it the *small* component of S, denoted by $\mathrm{s}(S)$.

By the definition of a full component, $S = N(\mathrm{s}(S))$. In particular, for distinct minimal separators S and T, we have that $\mathrm{s}(S) \neq \mathrm{s}(T)$. Therefore the number $\#\texttt{big sep}$ of big minimal separators is at most the number of small components and we conclude that $\#\texttt{big sep}$ does not exceed the number of subsets of V of cardinality at most $n(1-\alpha)/2$, i.e.

$$\#\texttt{big sep} \leq \sum_{i=1}^{\lceil n(1-\alpha)/2 \rceil} \binom{n}{i}$$

By making use of Stirling's formula we deduce that:

$$\#\texttt{big sep} \leq \frac{n(1-\alpha)}{2} \left(\pi n(1-\alpha)(1+\alpha)/2\right)^{-\frac{1}{2}} \left[\left(\frac{1-\alpha}{2}\right)^{-\frac{1-\alpha}{2}} \left(\frac{1+\alpha}{2}\right)^{-\frac{1+\alpha}{2}} \right]^{n}$$

4.2 Upper bounding the number of small separators

To count small separators we use a different technique. Let S be a minimal separator, let x be a vertex of a full component $C_x(S)$ associated to S with minimum number of vertices and let $X \subset V$ be a vertex subset. We say that (x, X) is a *bad pair* associated to S if $C_x(S) \subseteq X \subseteq V \setminus S$.

Claim 1. *Let $S \neq T$ be two minimal separators and let (x, X) and (y, Y) be two bad pairs associated to S and T respectively. Then $(x, X) \neq (y, Y)$.*

Proof. Since $C_x(S) \subseteq X$ and $X \cap S = \emptyset$, we have that the connected component of $G[X]$ containing x is $C_x(S)$. Similar, the connected component of $G[Y]$ containing y is $C_y(T)$.

Thus if $x = y$ and $X = Y$, then $C_x(S) = C_y(T)$. Since $C_x(S)$ is a full component associated to S in G, we have that $S = N(C_x(S))$ and $T = N(C_y(T))$. Therefore $S = T$, which is a contradiction. □

By Lemma 9, there are at least two full components associated to every small separator S. For a full component $C_x(S)$ associated to S with the minimum number of vertices, $|V \setminus (S \cup C_x(S))| \geq n \cdot (1-\alpha)/2$. For any $Z \subseteq V \setminus (S \cup C_x(S))$, the pair $(x, Z \cup C_x(S))$ is a bad pair associated to S. Therefore there are at least $2^{n \cdot (1-\alpha)/2}$ distinct bad pairs associated to S. Hence by Claim 1, the total number of bad pairs is at least $\#\texttt{small sep} \cdot 2^{n\cdot(1-\alpha)/2}$. On the other hand, the number of bad pairs is at most $n \cdot 2^n$. We conclude that

$$\#\texttt{small sep} \leq n2^{n\cdot(1+\alpha)/2}$$

Finally, choosing $\alpha = 0.5456$, we obtain

$$|\Delta_G| = \#\texttt{small sep} + \#\texttt{big sep} = \mathcal{O}(n \cdot 1.7087^n).$$

□

5 Upper bounding the number of potential maximal cliques

In this section the we prove the main technical result of this paper, namely that there exists an algorithm to list all potential maximal cliques of any graph in time $\mathcal{O}^*(1.8899^n)$.

Roughly speaking, the idea is to show that each potential maximal clique of a graph can be identified by a set of vertices of size at most $n/3$. The algorithm for generating all the potential maximal cliques of a graph, lists all the sets of vertices of size at most $n/3$ and then, by applying a polynomial time procedure for each set, generates all the potential maximal cliques of the input graph.

Lemma 11. *Let Ω be a potential maximal clique of G, S be a minimal separator contained in Ω and C be the component of $G \setminus S$ intersecting Ω. Then one of the following holds:*

1. *$\Omega = N(C \setminus \Omega)$;*
2. *there is $a \in \Omega \setminus S$ such that $\Omega = N[a]$;*
3. *there is $a \in S$ such that $\Omega = S \cup (N(a) \cap C)$.*

Proof. Since C is a component of $G \setminus S$ and S is contained in Ω, we have that $N(C \setminus \Omega) \subseteq \Omega$. If every vertex of Ω is adjacent to a vertex of $C \setminus \Omega$, then $\Omega = N(C \setminus \Omega)$.

Suppose that there is a vertex $a \in \Omega$ having no neighbor in $C \setminus \Omega$. We consider first the case $a \in \Omega \setminus S$. We claim that in this case $\Omega = N[a]$. Because $a \in \Omega \setminus S \subseteq C$ we conclude that $N[a] \subseteq \Omega$. Thus to prove the claim we need to show that $\Omega \subseteq N[a]$. For sake of contradiction, suppose that there is $b \in \Omega$ which is not adjacent to a. By Theorem 4, every two non adjacent vertices of a potential maximal clique are contained in some minimal separator $S_i(\Omega)$. Thus both a and b should have neighbors in a component $C_i(\Omega)$ of $G \setminus \Omega$. Since $a \in \Omega \setminus S \subseteq C$, we have that $C_i(\Omega) \subseteq C \setminus \Omega$. But this contradicts the assumption that a has no neighbors in $C \setminus \Omega$.

The case $a \in S$ is similar. Suppose that $\Omega \setminus S \neq N(a) \cap C$, i.e. there is a vertex $b \in \Omega \setminus S$ non adjacent to a. Then again, a and b are contained in some minimal separator and thus should have neighbors in a component $C_i(\Omega) \subseteq C$ of $G \setminus \Omega$ which is a contradiction. □

Definition 12. Let Ω be a potential maximal clique of G. The triple (S, a, b) is called a *separator representation* of Ω if S is a minimal separator of G, $a \in S$, $b \in V \setminus S$ and $\Omega = S \cup (N(a) \cap C_b(S))$, where $C_b(S)$ is the component of $G \setminus S$ containing b.

The number of all possible separator representations of a graph is at most $n^2|\Delta_G|$. Unfortunately, not every potential maximal clique has a separator representation. In the next subsection we introduce two different types of representations, the *partial representation* and the *indirect representation*, that allows us to show that all the potential maximal cliques can be represented by small sets of vertices.

5.1 Upper bounding the number of nice potential maximal cliques

Definition 13. Let Ω be a potential maximal clique of a graph G and let $S \subset \Omega$ be a minimal separator of G. We say that S is an *active separator for* Ω if Ω is not a clique in the graph $G_{\mathcal{S}(\Omega)\setminus\{S\}}$, obtained from G by completing all the minimal separators contained in Ω, except S. If S is active, a pair of vertices $x, y \in S$ non adjacent in $G_{\mathcal{S}(\Omega)\setminus\{S\}}$ is called an *active pair*. Otherwise, S is called *inactive for* Ω.

Theorem 14 ([10]). *Let Ω be a potential maximal clique of G and $S \subset \Omega$ a minimal separator, active for Ω. Let (S, C) be the block associated to S containing Ω and let $x, y \in \Omega$ be an active pair. Then $\Omega \setminus S$ is a minimal x, y-separator in $G[C \cup \{x, y\}]$.*

Definition 15. We say that a potential maximal clique Ω is *nice* if at least one of the minimal separators contained in Ω is active for Ω.

We shall prove first that a graph with n vertices has $\mathcal{O}^*(\binom{n}{n/3})$ nice potential maximal cliques.

Lemma 16. *Let Ω be a nice potential maximal clique, S be a minimal separator active for Ω, $x, y \in S$ be an active pair, and C be the component of $G \setminus S$ containing $\Omega \setminus S$. There is a partition (D_x, D_y, D_r) of $C \setminus \Omega$ such that $N(D_x \cup \{x\}) \cap C = N(D_y \cup \{y\}) \cap C = \Omega \setminus S$.*

Proof. By Theorem 14, $\Omega \setminus S$ is a minimal x, y-separator in $G[C \cup \{x, y\}]$. Let C_x be the full component associated to $\Omega \setminus S$ in $G[C \cup \{x, y\}]$ containing x, $D_x = C_x \setminus \{x\}$, and let C_y be the full component associated to $\Omega \setminus S$ in $G[C \cup \{x, y\}]$ containing y, $D_y = C_y \setminus \{y\}$, and $D_r = C \setminus (\Omega \cup D_x \cup D_y)$. Since $D_x \cup \{x\}$ and $D_y \cup \{y\}$ are full components of $\Omega \setminus S$, we have that $N(D_x \cup \{x\}) \cap C = N(D_y \cup \{y\}) \cap C = \Omega \setminus S$. □

Definition 17. For a potential maximal clique Ω of G, we say that a pair (X, c), where $X \subset V$ and $c \in X$ is a *partial representation* of Ω if $\Omega = N(C_c) \cup (X \setminus C_c)$, where C_c is the connected component of $G[X]$ containing c.

Definition 18. For a potential maximal clique Ω of G, we say that a triple (X, x, c), where $X \subset V$ and $x, c \notin X$ is an *indirect representation* of Ω if $\Omega = N(C_c \cup D_x \cup \{x\}) \cup \{x\}$, where

- C_c is the connected component of $G \setminus N[X]$ containing c;
- D_x is the vertex set of the union of all connected components C' of $G[X]$ such that $x \in N(C')$.

Let us note that for a given vertex set X and two vertices x, c one can check in polynomial time whether the pair (X, c) is a partial representation or if the triple (X, x, c) is a separator representation or indirect representation of a (unique) potential maximal clique Ω.

We state now the main tool for upper bounding the number of nice potential maximal cliques.

Lemma 19. *Let Ω be a nice potential maximal clique of G. Then one of the following holds:*

1. *There is a vertex a such that $\Omega = N[a]$;*
2. *Ω has a separator representation;*
3. *Ω has a partial representation (X, c) such that $|X| \leq n/3$;*
4. *Ω has a indirect representation (X, x, c) such that $|X| \leq n/3$.*

Proof. Let S be a minimal separator active for Ω, $x, y \in S$ be an active pair, and C be the component of $G \setminus S$ containing $\Omega \setminus S$. By Lemma 16, there is a partition (D_x, D_y, D_r) of $C \setminus \Omega$ such that $N(D_x \cup \{x\}) \cap C = N(D_y \cup \{y\}) \cap C = \Omega \setminus S$. If one of the sets D_x, D_y, say D_x, is equal the emptyset, then $N(D_x \cup \{x\}) \cap C = N(x) \cap C = \Omega \setminus S$, and thus the triple (S, x, z) is a separator representation of Ω.

Suppose that none of the first two conditions of the lemma holds. Then D_x and D_y are nonempty. In order to argue that Ω has a partial representation (X, c) or a indirect representation (X, x, c) such that $|X| \leq n/3$, we partition the graph further. Let $R = \Omega \setminus S$ and let D_S be the union of all full components associated to S in $G \setminus \Omega$. The vertex set D_x is the union of vertex sets of all connected components C' of $G \setminus (\Omega \cup D_S)$ such that x is contained in the neighborhood of C'. Thus a connected component C' of $G \setminus (\Omega \cup D_S)$ is contained in D_x if and only if $x \in N(C')$. Similarly, a connected component C' of $G \setminus (\Omega \cup D_S)$ is contained in D_y if and only if $y \in N(C')$. We also define $D_r = V \setminus (\Omega \cup D_S \cup D_x \cup D_y)$, which is the set of vertices of the components of $G \setminus (\Omega \cup D_S)$ which are not in D_x and D_y.

We partition S in the following sets

- $S_{\overline{x}} = (S \setminus N(D_x)) \cap N(D_y)$;
- $S_{\overline{y}} = (S \setminus N(D_y)) \cap N(D_x)$;
- $S_{\overline{xy}} = S \setminus (N(D_y) \cup N(D_x))$;
- $S_{xy} = S \cap N(D_y) \cap N(D_x)$.

Thus $S_{\overline{x}}$ is the set of vertices in S with no neighbor in D_x and with at least one neighbor in D_y, $S_{\overline{y}}$ is the set of vertices in S with no neighbor in D_y and with at least one neighbor in D_x, $S_{\overline{xy}}$ is the set of vertices in S with neighbors neither in D_x or D_y, and finally S_{xy} is the set of vertices in S with neighbors both in D_x and D_y. Notice that the vertex sets D_S, D_x, D_y, D_r, R,$S_{\overline{x}}$, $S_{\overline{y}}$, $S_{\overline{xy}}$, and S_{xy} are pairwise disjoint. The set S_{xy} is only mentioned to complete the partition of S, and will not be used in the rest of the proof.

Both for size requirements and because of the definition of indirect representation we can not use the sets $S_{\overline{x}}, S_{\overline{y}}$, and $S_{\overline{xy}}$ directly, they have to be represented by the sets $Z_{\overline{x}}, Z_{\overline{y}}$, and $Z_{\overline{r}}$, which are subsets of the vertex sets D_y, D_x, and D_r. By the definition of $S_{\overline{x}}$ and $S_{\overline{y}}$ it follows that there exists two vertex sets $Z_{\overline{x}} \subseteq D_y$ and $Z_{\overline{y}} \subseteq D_x$ such that $S_{\overline{x}} \subseteq N(Z_{\overline{x}})$ and $S_{\overline{y}} \subseteq N(Z_{\overline{y}})$, let $Z_{\overline{x}}$ and $Z_{\overline{y}}$ be the smallest such sets. By Lemma 11, $\Omega = N(D_x \cup D_y \cup D_r)$, thus it follows that there exists a vertex set $Z_{\overline{r}} \subseteq D_r$ such that $S_{\overline{xy}} \subseteq N(Z_{\overline{r}})$, let $Z_{\overline{r}}$ be the smallest such set. A sketch of how these vertex sets relates to each other is given in Figure 1.

Let C^* be a connected component of $G[D_S]$, remember that $N(C^*) = S$. We define the following sets

- $X_1 = C^* \cup R$;
- $X_2 = D_x \cup Z_{\overline{x}} \cup Z_{\overline{r}}$;
- $X_3 = D_y \cup Z_{\overline{y}} \cup Z_{\overline{r}}$.

First we claim that

- the pair (X_1, c), where $c \in C^*$, is a partial representation of Ω;

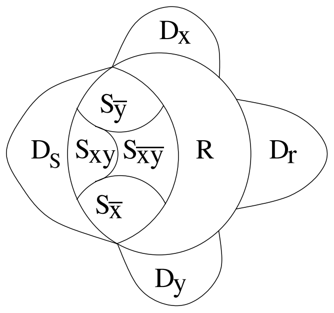

Figure 1: The figure shows a sketch of how the vertex sets $D_S, D_x, D_y, D_r, R, S_{\overline{x}}, S_{\overline{y}}, S_{\overline{xy}}$, and S_{xy} partition the graph G, and how the sets $Z_{\overline{x}}$, $Z_{\overline{y}}$, and $Z_{\overline{r}}$ relates to this partition.

- the triple (X_2, x, c), where $c \in C^*$ is an indirect representation of Ω;
- the triple (X_3, x, c), where $c \in C^*$ is an indirect representation of Ω.

In fact, the pair $(X_1, c) = (C^* \cup R, c)$ is a partial representation of Ω because $N(C^*) \cap R = \emptyset$, C^* induces a connected graph, and $\Omega = N(C^*) \cup R$. Thus (X_1, c) is a partial representation of Ω.

To prove that $(X_2, x, c) = (D_x \cup Z_{\overline{x}} \cup Z_{\overline{r}}, x, c)$ is an indirect representation of Ω, we have to show that $\Omega = N(C_c \cup D'_x \cup \{x\}) \cup \{x\}$ where C_c is the connected component of $G \setminus N[X_2]$ containing c, and D'_x is the vertex set of the union of all connected components C' of $G[X_2]$ such that $x \in N(C')$. Notice that $(S \cup C^*) \cap X_2 = \emptyset$ and that $S \subseteq N(X_2)$ since $S \subseteq N(D_x \cup Z_{\overline{x}} \cup Z_{\overline{r}})$ and $X_2 = D_x \cup Z_{\overline{x}} \cup Z_{\overline{r}}$. Hence the connected component C_c of $G \setminus N[X_2]$ containing c is C^*.

Every connected component C' of $G[X_2]$ is contained in $D_x, Z_{\overline{x}}$, or $Z_{\overline{r}}$ since $\Omega \cap (D_x \cup Z_{\overline{x}} \cup Z_{\overline{r}}) = \emptyset$ and Ω separates $D_x, Z_{\overline{x}}$, and $Z_{\overline{r}}$. From the definition of D_x it follows that $x \in N(C')$ for every component C' of $G[D_x]$, and from the definition of D_y and D_r follows that $x \notin N(C')$ for every component C' of $G[Z_{\overline{x}} \cup Z_{\overline{r}}]$. We can now conclude that D_x is the vertex set of the union of all connected components C' of $G[X_2]$ such that $x \in N(C')$. It remains to prove that $\Omega = N(C^* \cup D_x \cup \{x\}) \cup \{x\}$. By Lemma 16, we have that $\Omega \setminus S = R$ is subset of $N(D_x \cup \{x\})$ and $N(D_y \cup \{y\})$, and remember that $N(C^*) = S$. From this observations it follows that $\Omega = N(C^* \cup D_x \cup \{x\}) \cup \{x\}$ since $N(C^* \cup D_x \cup \{x\}) = (S \cup R) \setminus \{x\}$.

By similar arguments, (X_3, x, c) is an indirect representation of Ω.

To conclude the proof of Lemma, we argue that at least one of the vertex sets X_1, X_2, or X_3 used to represent Ω, contains at most $n/3$ vertices.

We partition the graph in the following three sets:

- $V_1 = D_S \cup R$;
- $V_2 = D_x \cup S_{\overline{x}} \cup S_{\overline{xy}}$;
- $V_3 = D_y \cup S_{\overline{y}} \cup D_r$.

These sets are pairwise disjoint and at least one of them is of size at most $n/3$ and to prove the Lemma we show that $|X_1| \le |V_1|$, $|X_2| \le |V_2|$, and $|X_3| \le |V_3|$.

$\underline{|X_1| \le |V_1|}$. Since $C^* \subseteq D_S$, we have that $X_1 = C^* \cup R \subseteq V_1 = D_S \cup R$.

$\underline{|X_2| \le |V_2|}$. To prove the inequality we need an additional result

$$|Z_{\overline{x}}| \le |S_{\overline{x}}|, |Z_{\overline{y}}| \le |S_{\overline{y}}|, \text{ and } |Z_{\overline{r}}| \le |S_{\overline{xy}}|. \tag{1}$$

In fact, since $Z_{\overline{x}}$ is the smallest subset of D_y such that $S_{\overline{x}} \subseteq N(Z_{\overline{x}})$, we have that for any vertex $u \in Z_{\overline{x}}$, $S_{\overline{x}} \not\subseteq N(Z_{\overline{x}} \setminus \{u\})$. Thus u has a private neighbor in $S_{\overline{x}}$, or in other words there exists $v \in S_{\overline{x}}$ such that $\{u\} = N(v) \cap Z_{\overline{x}}$. Therefore $S_{\overline{x}}$ contains at least one vertex for every vertex in $Z_{\overline{x}}$, which yields $|Z_{\overline{x}}| \le |S_{\overline{x}}|$. The proof of inequalities $|Z_{\overline{y}}| \le |S_{\overline{y}}|$, and $|Z_{\overline{r}}| \le |S_{\overline{xy}}|$ is similar.

Now the proof of $|X_2| \le |V_2|$, which is equivalent to $|D_x \cup Z_{\overline{x}} \cup Z_{\overline{r}}| \le |D_x \cup S_{\overline{x}} \cup S_{\overline{xy}}|$, follows from (1) and the fact that all subsets on each side of inequality are pairwise disjoint.

$\underline{|X_3| \le |V_3|}$. This inequality is equivalent to $|D_y \cup Z_{\overline{y}} \cup Z_{\overline{r}}| \le |D_y \cup S_{\overline{y}} \cup D_r|$. Again, the sets on each side of inequality are pairwise disjoint. $|Z_{\overline{r}}| \le |D_r|$ because $Z_{\overline{r}} \subseteq D_r$, and $|Z_{\overline{y}}| \le |S_{\overline{y}}|$ by (1).

Thus $\min\{|X_1|, |X_2|, |X_3|\} \le n/3$ which concludes the proof of the lemma. $\square$

Lemma 20. *Every graph on n vertices has at most $2n^2 \sum_{i=1}^{n/3} \binom{n}{i}$ nice potential maximal cliques which can be listed in $\mathcal{O}^*(\binom{n}{n/3})$ time.*

Proof. By Lemma 19, the number of the number of possible partial representations (X, c) and indirect representations (X, x, c) with $|X| \le n/3$ is at most $2n^2 \sum_{i=1}^{n/3} \binom{n}{i}$. By Theorem 10, the number of all possible separator representations is at most $n^2|\Delta_G| \le n^2\binom{n}{n/3}$ and we deduce that the number of nice potential maximal cliques is at most $2n^2 \sum_{i=1}^{n/3} \binom{n}{i}$. Moreover, these potential maximal cliques can be computed in $\mathcal{O}^*(\binom{n}{n/3})$ time as follows. We enumerate all the triples (S, a, b) where S is a minimal separator and a, b are vertices, and check if the triple is the separator representation of a potential maximal clique Ω; if so, we store this potential maximal clique. We also enumerate all the potential maximal cliques of type $N[a], a \in V(G)$ in polynomial time. Finally, by listing all the sets X of at most $n/3$ vertices and all the couples of vertices (x, c), we compute all the nice potential maximal cliques with a partial representation (X, c) or a indirect representation (X, x, c). $\square$

5.2 Upper bounding the number of potential maximal cliques

Not all potential maximal cliques of a graph are necessarily nice (see [10] for an example). For upper bounding and listing all potential maximal cliques of a graph, we need the following theorem, used in [10] for showing that the number of potential maximal cliques of G is $\mathcal{O}^*(|\Delta_G|^2)$.

Theorem 21 ([10]). *Let Ω be a potential maximal clique of G, let a be a vertex of G and $G' = G \setminus \{a\}$. Then one of the following cases holds:*

1. *either Ω or $\Omega \setminus \{a\}$ is a potential maximal clique of G'.*
2. *$\Omega = S \cup \{a\}$, where S is a minimal separator of G.*
3. *Ω is nice.*

Theorem 22. *A graph G on n vertices has at most $2n^3 \sum_{i=1}^{n/3} \binom{n}{i} = \mathcal{O}(n^4 \cdot 1.8899^n)$ potential maximal cliques. There is an algorithm to list all potential maximal cliques of a graph in time $\mathcal{O}^*(1.8899^n)$.*

Proof. Let $x_1, x_2, \dots, x_n$ be the vertices of G and $G_i = G[\{x_1, \dots, x_i\}]$, for all $i \in \{1, 2, \dots, n\}$. Theorem 21 and Lemma 20 imply that $|\Pi_{G_i}| \leq |\Pi_{G_{i-1}}| + n|\Delta_{G_i}| + 2n^2 \sum_{i=1}^{n/3} \binom{n}{i}$, for all $i \in \{2, 3, \dots, n\}$. By Theorem 10, $|\Pi_G| \leq 2n^3 \sum_{i=1}^{n/3} \binom{n}{i}$.

Clearly, if we have the potential maximal cliques of G_{i-1}, the potential maximal cliques of G_i can be computed in $\mathcal{O}^*(|\Pi_{G_{i-1}}| + \binom{n}{n/3})$ time by making use of Theorems 21, 10, and Lemma 20. The graph G_1 has a unique potential maximal clique, namely $\{x_1\}$. Therefore Π_G can be listed in time $\mathcal{O}^*(\binom{n}{n/3})$ time which is approximately $\mathcal{O}^*(1.8899^n)$. □

Theorems 8 and 22 imply the main result of this paper.

Theorem 23. *For a graph G on n vertices, the treewidth and the minimum fill-in of G can be computed in $\mathcal{O}^*(1.8899^n)$ time.*

6 AT-free graphs

In this section we establish exact algorithms to compute the treewidth and the minimum fill-in of asteroidal-triple free graphs which are faster than the ones obtained for general graphs in the previous section. Both algorithms are based on new upper bounds on the number of minimal separators and the number of potential maximal cliques in AT-free graphs.

Three pairwise non-adjacent vertices of a graph G form an *asteroidal triple* (AT for short) if any two of them are connected by a path avoiding the neighborhood of the third vertex. Graphs without asteroidal triples are called *AT-free.*

Corneil, Olariu & Stewart studied structural properties of AT-free graphs in their fundamental paper [18]. Among others they showed that every connected AT-free graph has a dominating pair, where two vertices x and y of G form a *dominating pair* (DP for short) if the vertex set of each x, y-path is a dominating set of G.

AT-free graphs contain cocomparability graphs, permutation graphs, interval graphs and cobipartite graphs. Thus the treewidth problem and the minimum fill-in problem remain NP-hard when restricted to AT-free graphs [2, 50].

Remark. There is a well-known cobipartite (and thus AT-free) graph consisting of two cliques of size $n/2$ and a perfect matching between them which has precisely $2^{n/2} - 2$ minimal separators. It is not hard to show that this is indeed the largest number of minimal separators of a cobipartite graph on n vertices.

In a first part we show that $|\Pi_G| = \mathcal{O}^*(|\Delta_G|)$ for AT-free graphs, improving a result in [9] (Corollary 5.2). This also establishes an algorithm to list the potential maximal cliques of an AT-free graph in $\mathcal{O}^*(|\Delta_G|)$ time. In a second part we prove that an AT-free graph on n vertices has at most $2^{n/2+3}$ minimal separators.

First let us summarize some structural properties of potential maximal cliques in AT-free graphs.

Lemma 24 (Proposition 5.1 [9]). *Let Ω be a potential maximal clique of an AT-free graph G. Then the set $\mathcal{S}(\Omega)$ of minimal separators contained in Ω has at most two inclusion-maximal elements.*

Lemma 25 (Theorem 3.10 [9]). *Let G be a graph and Ω be a potential maximal clique of G such that $\mathcal{S}(\Omega)$ has a unique inclusion-maximal element S. Then $\Omega \setminus S$ is a connected component of $G \setminus S$.*

Let S and T be two non-crossing minimal separators of G, incomparable with respect to inclusion. Thus S meets a unique component of $G \setminus T$ say $C_S(T)$, and T meets a unique component of $G \setminus S$ say $C_T(S)$. We define the *piece between S and T* as $P(S,T) = S \cup T \cup (C_T(S) \cap C_S(T))$.

Lemma 26 (Theorem 3.11 [9]). *Let G be a graph and Ω be a potential maximal clique of G such that $\mathcal{S}(\Omega)$ has exactly two inclusion-maximal elements S and T. Then $\Omega = P(S,T)$.*

Lemma 27. *Let G be an AT-free graph and Ω be a potential maximal clique of G such that $\mathcal{S}(\Omega)$ has two inclusion-maximal elements S and T. Choose $s \in S \setminus T$. Then $\Omega = S \cup (N(s) \cap C_T(S))$.*

Proof. By Lemma 26, $\Omega = P(S,T)$. Clearly s is in the unique component $C_S(T)$ of $G \setminus T$ meeting S, so $N(s) \cap C_T(S) \subseteq P(S,T)$. Consequently $S \cup (N(s) \cap C_T(S)) \subseteq \Omega$.

Conversely, suppose there is a vertex $t \in \Omega$, not contained in $S \cup (N(s) \cap C_T(S))$. Let $S' = (S \setminus \{s\}) \cup (N(s) \cap C_T(S))$. Clearly S' separates s and any vertex of $C_T(S) \setminus S'$ in G, in particular S' separates s and t. It follows that there is a minimal separator $S'' \subseteq S'$ of G, contained in Ω and separating two vertices of Ω. According to Theorem 4, for each minimal separator U contained in Ω, Ω intersects exactly one component of $G \setminus U$, which is a contradiction. □

Theorem 28. *An AT-free graph G has at most $n^2|\Delta_G| + n|\Delta_G| + 1$ potential maximal cliques. Furthermore, there is an algorithm to list the potential maximal cliques of an AT-free graph in $\mathcal{O}^*(|\Delta_G|)$ time.*

Proof. If G has no minimal separator, then G is a complete graph and its vertex set is the unique potential maximal clique of G.

Suppose now that G is not complete. Fix a minimal separator S of G. By Lemma 25, the number of potential maximal cliques Ω such that S is the unique inclusion-maximal element of $\mathcal{S}(\Omega)$ is bounded by the number of connected components of $G \setminus S$. Hence, there are at most n such potential maximal cliques.

Now let us consider the potential maximal cliques Ω for which S is one of the two inclusion-maximal separators contained in $\mathcal{S}(\Omega)$. For any component C of $G \setminus S$, there are, by Lemma 27, at most $|S|$ such potential maximal cliques contained in $S \cup C$. It follows that there are at most n^2 potential maximal cliques of this type.

Therefore, G contains at most $(n^2 + n)|\Delta_G| + 1$ potential maximal cliques. These combinatorial arguments can easily be transformed into an algorithm listing the potential maximal cliques of an AT-free graph in time $O^*(|\Delta_G|)$. □

Hence Theorem 8 implies that to construct an $O^*(1.4142^n)$ algorithm computing the tree-width and the minimum fill-in of an AT-free graph it is enough to prove that the number of minimal separators in an AT-free graph is $O^*(1.4142^n)$.

Our proof that the number of minimal separators in an AT-free graph is at most $2^{n/2+3}$ relies on properties of 2LexBFS, i.e. a combination of two runs of lexicographic breadth-first-search (also called 2-sweep LexBFS), on AT-free graphs established by Corneil, Olariu & Stewart in [19].

Definition 29. A vertex ordering $x_n, x_{n-1}, \ldots, x_1$ is said to be a *2LexBFS ordering* of G if some 2LexBFS(G) returns the vertices in this order (starting with x_n) during the second sweep of LexBFS on G where x_n is supposed to be the last vertex of the first sweep of LexBFS on G.

We shall write $u \prec v$ if $u = x_i$, $v = x_j$ and $i < j$. A 2LexBFS ordering and the levels $L_0 = \{x_n\}, L_1 = N(x_n), \ldots, L_i = \{x_j : d(x_j, x_n) = i\}, \ldots, L_r$ are called a *2LexBFS scheme* of G. Consider any 2LexBFS scheme. Clearly all neighbors of a vertex $v \in L_i$ are contained in $L_{i-1} \cup L_i \cup L_{i+1}$. For a vertex $v \in L_i$ we denote $N(v) \cap L_{i-1}$ by $N^{\uparrow}(v)$, and we denote $N(v) \cap L_{i+1}$ by $N_{\downarrow}(v)$.

Theorem 30 ([19]). *Every 2LexBFS ordering $x_n, x_{n-1}, \dots, x_1$ of a connected AT-free graph has the* dominating pair-property *(DP-property for short), i.e., for all $i \in \{1, 2, \dots, n\}$, (x_n, x_i) is a dominating pair of the graph $G[\{x_i, x_{i+1}, \dots, x_n\}]$.*

The following easy consequence of Theorem 30 is useful.

Lemma 31. *Let $x_n, x_{n-1}, \dots, x_1$ be a 2LexBFS ordering of an AT-free graph G and let $L_0, L_1, \dots, L_r$ be the corresponding 2LexBFS scheme. Let $s > r$, $x_s, x_r \in L_i$ and $\{x_r, x_s\} \notin E$. Then $N^{\uparrow}(x_r) \subseteq N^{\uparrow}(x_s)$.*

Proof. Let $w \in N^{\uparrow}(x_r) \setminus N^{\uparrow}(x_s)$. Then the path $x_r, w, u_{i-2}, \dots, u_1, x_n$ with $u_j \in L_j$ and $u_{j-1} \in N^{\uparrow}(u_j)$ for all $j = i-2, \dots, 1$ contains no neighbor of x_s contradicting the DP-property of a 2LexBFS scheme of an AT-free graph. □

Theorem 32. *An AT-free graph on n vertices has at most $2^{n/2+3}$ minimal separators.*

Proof. Let G be an AT-free graph. Let $x_n, x_{n-1}, \dots, x_1$ be a 2LexBFS ordering of G and let $L_0, L_1, \dots, L_r$ be the levels of the corresponding 2LexBFS scheme.

Let S be any minimal separator of G. Let C and C' be two (not necessarily full) components of $G \setminus S$. We claim that at most one level of the 2LexBFS scheme may contain vertices of C and C'. Suppose not. Let L_i and L_{i+1} be levels containing vertices of C and C'. Then there are edges $\{u, v\}$ in C and $\{w, x\}$ in C' such that $u, w \in L_i$ and $v, x \in L_{i+1}$. W.l.o.g. assume $u \prec w$. Then Lemma 31 implies that w and v are adjacent, a contradiction.

Let C and C' be two (not necessarily full) components of $G \setminus S$ such that both contain vertices of some level of the 2LexBFS scheme, say L_i. Furthermore assume $C \cap L_{i-1} \neq \emptyset$ and $C' \cap L_{i-1} = \emptyset$. Hence there is an edge $\{u, v\}$ in C such that $u \in L_i$ and $v \in L_{i-1}$. Then for each $w \in C'$ holds $w \prec u$. Otherwise $u \prec w$, $w \in L_i$ and Lemma 31 would imply that w and v are adjacent, a contradiction.

Finally we claim that in this case $c' \prec c$ for each vertex $c \in C$ and each vertex $c' \in C'$. This is obviously true if one of c and c' is not in L_i. It remains to consider the case $c \in L_i$, $c' \in L_i$. To the contrary assume $c \prec c'$. Since C contains vertices of L_i and L_{i-1} there is a path in C starting in c passing through vertices of $C \cap L_i$ only until it passes through an edge $\{u, v\}$ in C with $u \in L_i$ and $v \in L_{i-1}$. This path can be extended to a path from c to x_n that does not contain a neighbor of c' although $c \prec c'$, a contradiction to the DP-property.

Now we are able to upper bound the number of those minimal separators in an AT-free graph in which no full component contains only vertices of one level. Simply divide the vertex set into two halves: $A = \{x_n, x_{n-1}, \dots, x_{\lceil n/2 \rceil + 1}\}$ and $B = \{x_{\lfloor n/2 \rfloor}, \dots, x_1\}$. Now consider two full components C and C' of a minimal separator S of G, i.e. $S = N(C) = N(C')$. Then either C or C' is a subset of either A or B, and surely each of them C and C' uniquely determines S. Hence we simply consider all subsets of A and all subsets of B as possible full components of a minimal separator of G. Consequently there are at most $2^{n/2+1}$ minimal separators of this type.

It remains to upper bound the number of all those minimal separators S of an AT-free graph G for which each full component is neither a subset of A nor a subset of B. Hence at least one full component of S contains only vertices from one level of the 2LexBFS scheme.

Let S be such a minimal separator of G. Let C and C' be two full components of $G \setminus S$. W.l.o.g. assume $C \subseteq L_i$. Hence $x_{\lfloor n/2 \rfloor} \in L_i$, and thus the level L_i is uniquely determined.

$C' \cap \bigcup_{j=0}^{i-1} L_j = \emptyset$ since otherwise $c \prec c'$ for all $c \in C$ and all $c' \in C'$, and either C or C' is a subset of A or B. Similarly C' must contain vertices of L_i. Consequently $C' \subseteq \bigcup_{j=i}^{r} L_j$. It is easy to see that $C \subseteq L_i$ and $S = N(C)$ imply $N(C') = S \subseteq \bigcup_{j=i-1}^{i+1} L_j$. Furthermore $N(C) = N(C') = S$ implies $S \cap L_{i-1} = N^{\uparrow}(C \cap L_i) = N^{\uparrow}(C' \cap L_i)$.

Now let us consider the graph $G' = G \setminus \bigcup_{j=0}^{i-1} L_j$. Then $S' = S \setminus \bigcup_{j=0}^{i-1} L_j$ is a separator of G'; C and C' are components of $G' \setminus S'$. Furthermore, every vertex of $S' \subseteq S$ has a neighbor in C and C', and thus S' is a minimal separator of G'. Consequently every minimal separator S of G for which no full component is a subset of A or B corresponds uniquely to a minimal separator of G'. Notice G' has at most $n-1$ vertices since we remove at least one vertex of L_{i-1} from G to obtain G'.

Let $f(n)$ be a function such that $f(n)$ is an upper bound for the number of minimal separators in an n-vertex AT-free graph. Then we established the recurrence $f(n) \geq 2^{n/2+1} + f(n-1)$ and conclude with $f(n) = 4 \cdot 2^{n/2+1} = 8 \cdot 2^{n/2}$. $\square$

Combining Theorems 8, 28, and 32 we obtain algorithms for AT-free graphs being faster than the corresponding ones for general graphs.

Theorem 33. *There are algorithms to compute the treewidth and the minimum fill-in of an AT-free graph in $O^*(1.4142^n)$ time.*

7 Open problems and final remarks

Planar graphs. The computational complexity of treewidth restricted to planar graphs is a long standing open problem in Graph Algorithms. The treewidth of planar graphs can be approximated within a constant factor of 1.5. More precisely, Seymour and Thomas [43] gave a polynomial algorithm for computing the *branchwidth* of planar graphs, and the latter parameter differs by at most a factor of 1.5 from the treewidth.

In the case of planar graphs with n vertices, the treewidth is at most $\mathcal{O}(\sqrt{n})$.

Theorem 34 ([27]). *For any planar graph G on n vertices,* $\mathrm{tw}(G) \leq 3.182\sqrt{n} + \mathcal{O}(1)$.

Also given a graph G and a number k, one can decide if $\mathrm{tw}(G) \leq k$ in $\mathcal{O}^*(n^k)$ time, either using the algorithm of Arnborg et al. [2] or using our technique, restricted to potential maximal cliques of size at most $k+1$.

Consequently the treewidth of planar graphs can be computed in time $\mathcal{O}^*(n^{3.182\sqrt{n}}) = 2^{\mathcal{O}(\sqrt{n}\log n)}$.

Unfortunately, although the structure of potential maximal cliques in planar graphs is very particular [12], our approach can not be used for obtaining algorithms of running time $2^{O(\sqrt{n})}$ for planar treewidth. This is because the number of 'large' potential maximal cliques in planar graphs can be 'large'.

Claim 2. *For any integer N, there is a planar graph on $n > N$ vertices with at least $2^{0.49\sqrt{n}\log n}$ potential maximal cliques of size at least $2\sqrt{n}+2$.*

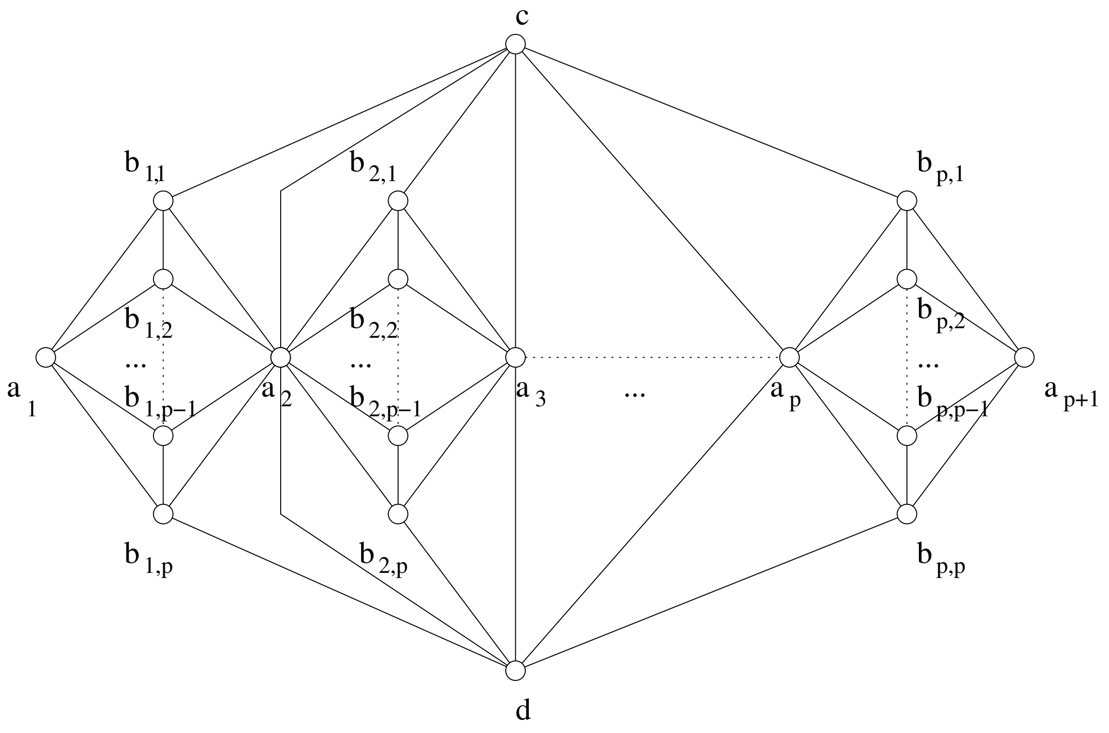

Figure 2: Planar graphs with many small potential maximal cliques

Proof. Consider the planar graph G_p depicted in Figure 2. It has $n = p^2 + p + 3$ vertices. The set of vertices $S = \{a_1, b_{1i_1}, a_2, b_{2i_2}, \ldots, a_p, b_{pi_p}, a_{p+1}\}$ forms a c,d-minimal separator for any values $i_1, i_2, \ldots, i_p$ between 1 and p. By making use of Theorem 4, it is not hard to see that $S \cup \{c\}$ is a potential maximal clique of size $p+1$ in G_p. Consequently, G has at least p^p potential cliques. If $p \geq 2$, we have $p > \sqrt{n}-1$, thus the number of potential maximal cliques is at least $(\sqrt{n}-1)^{\sqrt{n}-1}$. □

Since we do not know if the treewidth of a planar graph can be computed in polynomial time, an interesting task is to design an algorithm of running time $2^{\mathcal{O}(\sqrt{n})}$. As we mentioned, this will need new techniques.

Combinatorial bounds. The running time estimation of our algorithms is based on combinatorial upper bounds on minimal separators and a bound for listing all potential maximal cliques. To find better bounds on the number of minimal separators and potential maximal cliques are interesting combinatorial challenges.

How many potential maximal cliques can be in a graph? Recently Villanger [45] proved that the number of potential maximal cliques in a graph on n vertices is at most $n^3 \cdot 1.8135^n$. Unfortunately, it is not clear if the proof of Villanger can be turned into algorithm listing all potential maximal cliques in time $\mathcal{O}^*(1.8135^n)$. Of course, such an algorithm can be used to speed up our algorithm for treewidth and fill-in. A related interesting question is if it is possible to list potential maximal cliques with polynomial delay.

How many minimal separators can be in a graph? We are aware of the following construction providing the lower bound $3^{n/3} \approx 1.4422^n$ on the number of minimal separators: Let G be a graph on $n = 3k+2$ vertices. G has two vertices a, b that are connected by k vertex disjoint paths of length 4. Every minimal a, b-separator in G contains exactly one inner vertex of each a, b-path. So the number of minimal separators in G is at least $3^{n/3} \approx 1.4422^n$. However the gap between the lower bound and the upper bound $\mathcal{O}(n \cdot 1.7087^n)$ from Theorems 10 is still big.

For some special graph classes, the use of minimal separators can imply faster algorithms for triangulation problems. For example, we have shown that every AT-free graph on n vertices has at most $2^{n/2+3}$ minimal separators and that this upper bound is tight up to a multiplicative constant factor. The interesting question here is whether similar techniques can be used for other graph classes, like bipartite graphs and graphs of small degree.

More generally, it is a natural and challenging question to ask how many subgraphs satisfying a given property can be in a given graph. Surprisingly, despite the question is so natural, there are not so many known results of this type. For example, the number of perfect matchings in a simple k-regular bipartite graph on $2n$ vertices is always between $n!(k/n)^n$ and $(k!)^{n/k}$. (The first inequality was known as van der Waerden Conjecture [46] and was proved in 1980 by Egorychev [20] and the second is due to Bregman [13].) Another example is the famous Moon and Moser [36] theorem stating that every graph on n vertices has at most $3^{n/3}$

maximal cliques (independent sets). Byskov and Eppstein [16] obtain a 1.7724^n upper bound on the number of maximal bipartite subgraphs in a graph. Such combinatorial bounds are of interests not only on their own but also because often they are used for algorithm design as well.

Related problems. Our algorithms for treewidth and minimum fill-in can also be used for solving other problems that can be expressed in terms of minimal triangulations like finding a tree decomposition of minimum cost [8] or computing treewidth of weighted graphs. However, there are two 'width' parameters related to treewidth, namely bandwidth and pathwidth and one parameter called profile, related to minimum fill-in, that do not fit into this framework. Bandwidth can be computed in time $\mathcal{O}^*(10^n)$ [22] and reducing Feige's bounds is a challenging problem. Pathwidth (and profile) can be expressed as vertex ordering problems and thus solved in $\mathcal{O}^*(2^n)$ time by applying a dynamic programming approach similar to Held and Karp's approach [29] for TSP. Let us note that reaching time complexity $\mathcal{O}^*(c^n)$, for any constant $c < 2$ even for the Hamiltonian cycle problem is a long standing problem. So it is unlikely that some modification of Held & Karp's approach provide us with a better exact algorithm for pathwidth or profile. It is tempting to ask if one can reach time complexity $\mathcal{O}^*(c^n)$, for any constant $c < 2$ for these problems.

References

[1] E. Amir, *Efficient approximation for triangulation of minimum treewidth*, in Uncertainty in Artificial Intelligence: Proceedings of the Seventeenth Conference (UAI-2001), San Francisco, CA, 2001, Morgan Kaufmann Publishers, pp. 7–15.

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

[3] R. Beigel and D. Eppstein, *3-coloring in time* $O(1.3289^n)$, Journal of Algorithms, 54 (2005), pp. 168–204.

[4] A. Berry, J.P. Bordat, and O. Cogis, *Generating all the minimal separators of a graph*, Proceedings of the 25th Workshop on Graph-theoretic Concepts in Computer Science (WG'99), LNCS vol. 1665, Springer, Berlin, 1999, pp. 167–172.

[5] H. L. Bodlaender, *A linear-time algorithm for finding tree-decompositions of small treewidth*, SIAM J. Computing, 25 (1996), pp. 1305–1317.

[6] H. L. Bodlaender, *A partial k-arboretum of graphs with bounded treewidth*, Theoret. Comput. Sci., 209 (1998) , pp. 1–45.

[7] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks, *Approximating treewidth, pathwidth, frontsize, and shortest elimination tree*, J. Algorithms, 18 (1995), pp. 238–255.

[8] H. L. Bodlaender and F. V. Fomin, *Tree decompositions with small cost,* Proceedings of the 8th Scandinavian Workshop on Algorithm Theory (SWAT 2002), LNCS vol. 2368, Springer, Berlin, 2002, pp. 378–387.

[9] V. Bouchitté and I. Todinca, *Treewidth and minimum fill-in: grouping the minimal separators,* SIAM J. Computing, 31 (2001), pp. 212–232.

[10] V. Bouchitté and I. Todinca, *Listing all potential maximal cliques of a graph,* Theoret. Comput. Sci., 276 (2002), pp. 17–32.

[11] V. Bouchitté, D. Kratsch, H. Müller, and I. Todinca, *On treewidth approximation,* Discr. Appl. Math., 136 (2004), pp. 183–196.

[12] V. Bouchitté, F. Mazoit, and I. Todinca, *Chordal embeddings of planar graphs*, Discr. Math., 273 (2003), pp. 85–102.

[13] L. M. Brègman, *Certain properties of nonnegative matrices and their permanents*, Doklady Akademii Nauk BSSR, 211 (1973), pp. 27–30.

[14] T. Brueggemann and W. Kern, *An improved determenistic local search algorithm for 3-SAT*, Theoret. Comput. Sci., 329 (2004), pp. 303-313.

[15] J.M. Byskov, *Enumerating maximal independent sets with applications to graph colouring*, Operations Research Letters, 32 (2004),pp. 547–556.

[16] M. Byskov and D. Eppstein, *An algorithm for enumerating maximal bipartite subgraphs*, manuscript, (2004).

[17] L. Cai, *Fixed-parameter tractability of graph modification problems for hereditary properties*, Information Processing Letters, 58 (1996), pp. 171–176.

[18] D.G. Corneil, S. Olariu and L. Stewart, *Asteroidal triple-free graphs*, SIAM J. Discrete Math., 10 (1997), pp. 399–430.

[19] D.G. Corneil, S. Olariu and L. Stewart, *Linear time algorithms for dominating pairs in asteroidal triple-free graphs,* SIAM J. Computing, 28 (2000), pp. 1284–1297.

[20] G. P. Egorychev, *Proof of the van der Waerden conjecture for permanents,* Sibirsk. Mat. Zh., 22 (1981), pp. 65–71, 225.

[21] R. G. Downey and M. R. Fellows, *Parameterized Complexity,* Springer-Verlag, New York, 1999.

[22] U. Feige, *Coping with the NP-hardness of the graph bandwidth problem*, Proceeding of the 7th Scandinavian Workshop on in Algorithm theory (SWAT 2000), LNCS vol. 1851, Springer, Berlin, 2000, pp. 10–19.

[23] U. Feige, M. Hajiaghayi, and J. R. Lee, *Improved approximation algorithms for minimum-weight vertex separators*, in Proceedings of the 37th annual ACM Symposium on Theory of computing (STOC 2005), New York, 2005, ACM Press, pp. 563–572.

[24] F. V. Fomin, P. Fraigniaud, and N. Nisse, *Nondeterministic Graph Searching: From Pathwidth to Treewidth*, Proceedings of the 30th International Symposium on Mathematical Foundations of Computer Science (MFCS 2005), Springer-Verlag Lecture Notes in Computer Science, to appear.

[25] F. V. Fomin, F. Grandoni, and D. Kratsch, *Measure and conquer: Domination – a case study*, in Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP 2005), vol. 3580 of LNCS, Springer, Berlin, 2005, pp. 191–203.

[26] F. V. Fomin, D. Kratsch, and I. Todinca, *Exact (exponential) algorithms for treewidth and minimum fill-in*, Proceedings of the 31st International Colloquium on Automata, Languages and Programming, ICALP 2004, Turku, Finland, July 12-16, 2004, pp. 568–580.

[27] F. Fomin and D. Thilikos, *A simple and fast approach for solving problems on planar graphs*, Proceedings of the 21st Annual Symposium on Theoretical Aspects of Computer Science, LNCS vol. 2996, Springer, Berlin, 2004, pp. 56–67.

[28] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[29] M. Held and R. Karp, *A dynamic programming approach to sequencing problems*, J. Soc. Indust. Appl. Math., 10 (1962), pp. 196–210.

[30] E. Horowitz and S. Sahni, *Computing partitions with applications to the knapsack problem*, Journal of ACM, 21 (1974), pp. 277–292.

[31] K. Iwama, *Worst-case upper bounds for ksat*, Bulletin of the EATCS, 82 (2004), pp. 61–71.

[32] H. Kaplan, R. Shamir, and R. E. Tarjan, *Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs*, SIAM J. Computing, 28 (1999), pp. 1906–1922.

[33] T. Kloks, D. Kratsch, and J. Spinrad, *On treewidth and minimum fill-in of asteroidal triple-free graphs*, Theoret. Comput. Sci., 175 (1997), pp. 309–335.

[34] E. L. Lawler, *A note on the complexity of the chromatic number problem*, Information Processing Letters, 5 (1976), pp. 66–67.

[35] B. Monien and E. Speckenmeyer, *Solving satisfiability in less than 2^n steps* Discr. Appl. Math., 10 (1985), pp. 287–295.

[36] J. W. Moon and L. Moser, *On cliques in graphs*, Israel Journal of Mathematics, 3 (1965), pp. 23–28.

[37] A. Parra and P. Scheffler, *Characterizations and algorithmic applications of chordal graph embeddings,* Discr. Appl. Math., 79 (1997), pp. 171–188.

[38] N. Robertson and P. D. Seymour, *Graph minors. II. Algorithmic aspects of tree-width*, J. Algorithms, 7 (1986), pp. 309–322.

[39] N. Robertson and P. D. Seymour, *Graph minors. X. Obstructions to tree-decomposition*, J. Combin. Theory Ser. B, 52 (1991), pp. 153–190.

[40] J.M. Robson, *Algorithms for maximum independent sets*, J. Algorithms, 7 (1986), pp. 425–440.

[41] R. Schroeppel And A. Shamir, *A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems*, SIAM J. Computing, 10 (1981),pp. 456–464.

[42] Schöning, *Algorithmics in exponential time*, in Proceedings of the 22nd International Symposium on Theoretical Aspects of Computer Science (STACS 2005), vol. 3404 of LNCS, Springer, Berlin, 2005, pp. 36–43.

[43] P. Seymour and R. Thomas, *Call routing and the ratcatcher*, Combinatorica, 14 (1994), pp. 217–241.

[44] R.E. Tarjan and A.E. Trojanowski, *Finding a maximum independent set*, SIAM J. Computing, 6 (1977), pp. 537–546.

[45] Y. Villanger, *Improved exponential-time algorithms for treewidth and minimum fill-in*, In *LATIN*, Lecture Notes in Computer Science. Springer Verlag, 2006. To appear.

[46] B. van der Waerden, *Problem 45*, Jahresber. Deutsch. Math.-Verein., 35 (1926), p. 117.

[47] R. Williams, *A new algorithm for optimal constraint satisfaction and its implications*, Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004), LNCS vol. 3142, Springer, Berlin, 2004, pp. 1227–1237.

[48] G. Woeginger, *Exact algorithms for NP-hard problems: A survey*, in Combinatorial Optimization - Eureka, you shrink!, vol. 2570 of LNCS, Springer-Verlag, Berlin, 2003, pp. 185–207.

[49] ——, *Space and time complexity of exact algorithms: Some open problems*, Proceeding of the 1st International Workshop on Parameterized and Exact Computation (IWPEC 2004), vol. 3162 of LNCS, Springer-Verlag, Berlin, 2004, pp. 281–290.

[50] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Algebraic Discrete Methods, 2 (1981), pp. 77–79.