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 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, 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 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 triangulated graph obtained by adding the fill suggested by this process to the original graph 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].

*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

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 [4]. 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 [6]: 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 [6] 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 [3] 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 [20], 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 [20] 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 [20]) *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 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 [5]. 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 [2, 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 [2], [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 $C \neq \emptyset$, $C \subseteq V \setminus S$, $G(C)$ 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 $C \in \mathscr{C}_{G'}(S)$. $\square$

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 $y \in N_{G'}(x) \setminus N_G(x)$ 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. $\square$

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 $S \subseteq C \cup N(C)$. $\square$

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'. $\square$

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. $\square$

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. $\square$

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. $\square$

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 ∈ 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 [5].

Property 4.2 (Berry, Bordat, and Heggernes [5]) *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^{LB}_α. This will become clear when we prove Invariant 4.7.

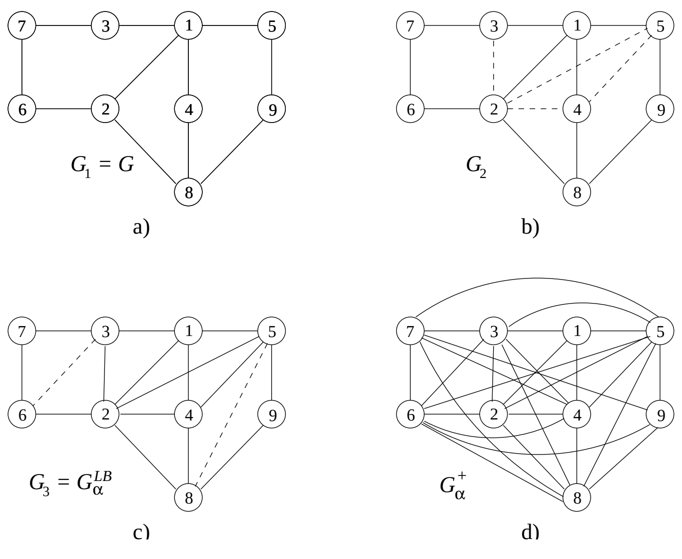

Figure 1: An example of how Algorithm LB-Triang proceeds.

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. $\square$

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). $\square$

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. $\square$

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 [20] 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. $\square$

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, $ab \in LBDef_{G(V\setminus X)}(y)$. $\square$

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 $F_i = LBDef_{G_i}(x_i) = LBDef_{G'_i}(x_i) = F'_i$. $\square$

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. $\square$

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 $G \subseteq G^{LB}_\alpha$ is evident. Let us show that $G^{LB}_\alpha \subseteq G^+_\alpha$. 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 $G^{LB}_\alpha \subseteq G^+_\alpha$. $\Box$

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

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. $\square$

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, $G^+_\alpha = G^{LB}_\alpha$, 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, $G^{LB}_\alpha = G^+_\alpha = H$. $\square$

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, $H = G^{LB}_\alpha$. $\square$

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^{LB}_{α}.

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 α in which the vertices are processed,
a minimal triangulation G^{LB}_{α} of G, $G^{LB}_{\alpha} = (V, E + F)$.

begin
  $F \leftarrow \emptyset$;
  $G_1 \leftarrow G$;
  for $i = 1 \dots n$ **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;
    $F \leftarrow F + F_i$;
    $G_{i+1} \leftarrow (V, E + F)$;
  $\alpha \leftarrow [x_1, x_2, ..., x_n]$;
  $G^{LB}_{\alpha} \leftarrow G_{n+1}$;
  return$(F, \alpha, G^{LB}_{\alpha})$.
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. $\square$

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]. $\square$

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 [9]. 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 𝒞_G(N_{G_i}[x_i]) do
        Neighbor(C) ← ∅ ;
    foreach y in N_{G_i}(x_i) in increasing order do
        foreach z in N_G(y) \ N_{G_i}[x_i] do
            let C ∈ 𝒞_G(N_{G_i}[x_i]) containing z;
            if y ≠ last(Neighbor(C)) then
                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 $G_\alpha^{LB}(S)$)*, 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. $\square$

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 $O(nm)$.

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 [7]. 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). $\square$

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. $\square$

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 $T_C \subseteq T_{vu}$. $\square$

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 ← (V, ∅);
    T ← ({u_0}, ∅);
    X_{u_0} ← V;
    InitVariables() ;
    for i = 1 ... n do
        Pick any unprocessed vertex x, and number it as x_i;
        N_H[x_i] ← Neighbors(G, x_i, TS);
        foreach C ∈ 𝒞_G(N_H[x_i]) do
            S ← N_G(C);
            Search/Insert S in the S/I data structure;
            if S has not been found in the S/I data structure then
                Let c be a vertex of C;
                Search in T from u(c) until a node w such that x_i ∈ X_w is reached;
                Split w into w_1 and w_2;
                X_{w_1} ← X_w ∩ (C ∪ S);
                X_{w_2} ← X_w \ C;
                Replace each edge wv by w_1v with S_{w_1v} = S_{wv} if S_{wv} ⊆ C ∪ S and by w_2v
                with S_{w_2v} = S_{wv} otherwise;
                Add edge w_1w_2;
                S_{w_1w_2} ← S;
                UpdateVariables();
    α ← [x_1, x_2, ..., x_n];
    return(α, 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.

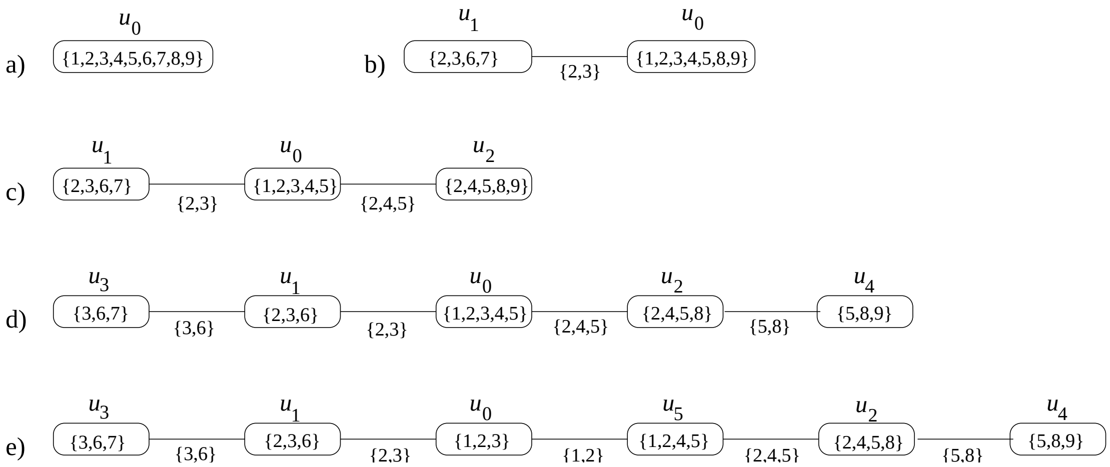

Figure 2: The successive states of tree T in the execution of Algorithm LB-Treedecomp on graph G of Figure 1 a)

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 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: **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: **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^{LB}_α 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^{LB}_α, and T a clique tree of the chordal graph G^{LB}_α. 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. $G_{j+1} = G'_{j+1}$ $\square$

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 $N_G[x] \cup \{y \in V \mid T_x \text{ and } T_y \text{ have at least one common edge}\}$. 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:*

- *$OneEdge = \{y \in V \mid T_y \text{ has at least one edge}\}$*
- *$Inner(x) = \{y \in OneEdge \mid T_y \text{ is included in } T_x\}$*

• $InnerOuter(x) = \{y \in OneEdge \mid T_y$ *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) $CommonEdge(x) = \cup_{uv \in E_T \mid x \in S_{uv}} S_{uv}$,
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 $\cup_{uv \in E_T \mid x \in S_{uv}} S_{uv}$, 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 $OneEdge \setminus ThroughBorder(x)$. 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 $Degree_T(u,y) - |\{v \in N_T(u) \mid (y \in S_{uv}$ 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 ← ∅;
    Inner ← OneEd;
    foreach (u, v) ∈ Border_T(T_x) do
        foreach y ∈ S_uv do
            Add y to InnerOuter;
            Remove y from Inner;
            Count(u, y) ← Deg(u, y);
    foreach (u, v) ∈ Border_T(T_x) do
        foreach y ∈ S_uv do
            Decrement Count(u, y);
            if Count(u, y) = 0 then
                Remove y from InnerOuter;
    foreach y ∈ Inner do
        if x ∉ X_u(y) then
            Remove y from Inner;
    return (N_G[x] ∪ Inner ∪ 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 $N_G[x] \cup (\cup_{uv \in E_T | x \in S_{uv}} S_{uv})$. $\square$

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 [9]. 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-Treedecomp, 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 $uv = w_1w_2$ 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 $uv \neq w_1w_2$ 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 $uv = w_1w_2$, 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, $T'_{C_1} \subseteq T'_{w_1w_2} = T'_{uv}$ and $T'_{C_2} \subseteq T'_{w_2w_1} = T'_{vu}$. $\square$

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^{LB}_α 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, $N_{G_i}[x_i] = N_{G^{LB}_\alpha}[x_i]$. Therefore $N_H(x) = N_{G^{LB}_\alpha}(x)$ for every vertex x of G, which means that $H = G^{LB}_\alpha$. The $O(nm)$ time complexity follows from Invariant 7.21 and Theorem 7.16. $\square$

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 [16], 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.

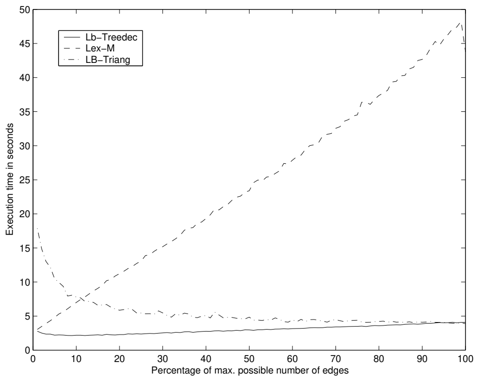

Figure 3: Comparing the running times of LB-Triang, LB-Treedec, and LEX M.

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.

In the second test, we tested the $O(nm')$ time implementation of LB-Triang also against the previously mentioned Algorithm MinimalChordal (MC) from [6]. 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.

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.

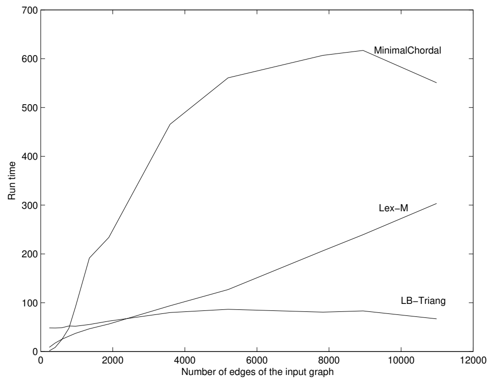

Figure 4: Comparing the running times of MinimalChordal, LB-Triang and LEX M.

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 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.

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.

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

[3] A. Berry. A wide-range efficient algorithm for minimal triangulation. *Proceedings of SODA'99.*

[4] A. Berry, J. R. S. Blair, and P. Heggernes. Maximum Cardinality Search for Computing Minimal Triangulations. In L. Kucera, editor, *Graph Theoretical Concepts in Computer Science - WG 2002*, Springer Verlag, 2002. Lecture Notes in Computer Science.

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

[6] 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.

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

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

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

[10] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. *Proceedings of the* 24^{th} *National Conference of the ACM*, pages 157–172, 1969.

[11] E. Dahlhaus. Minimal elimination ordering inside a given chordal graph. In R. H. Möhring, editor, *Graph Theoretical Concepts in Computer Science*, 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 matrixes 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] J. A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.

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

[17] D. Kratsch and J. Spinrad. Between $O(nm)$ and $O(n^\alpha)$. To appear in Proceedings of SODA 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] 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.

[21] J. W. H. Liu. Equivalent sparse matrix reorderings by elimination tree rotations. *SIAM J. Sci. Stat. Comput.*, 9:424–444, 1988.

[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.