Cand. Scient. Thesis
in
Informatics

Efficient Minimal Triangulation of Graphs by Using Tree Decompositions

by

Yngve Villanger

June, 2002

Department of Informatics
University of Bergen
Norway

Contents

1 Introduction **7**
1.1 Time complexity 8
1.2 Graph terminology 9
1.2.1 Tree decompositions 11
1.3 Cholesky factorization 12

2 Chordal Graphs and triangulations **17**
2.1 Characterization of chordal graphs 17
2.2 Clique graphs and clique trees 20
2.3 Elimination game 22
2.4 Maximum Cardinality Search (MCS) 24
2.5 Minimal triangulations 25
2.6 Lex-M 26
2.6.1 An example 27
2.6.2 Limitations of Lex-M 27

3 LB-triangulation **31**
3.1 Basic triangulation 31
3.1.1 The correctness of *LB-triang* 32
3.1.2 An example of *LB-triang* 33
3.1.3 Straight forward implementation of *LB-triang* 34
3.1.4 Time and space analysis 35
3.2 Improving the implementation 35
3.2.1 Decide if the separator is new 36
3.2.2 Using an adjacency matrix 36
3.2.3 Using an adjacency list *(LB-list)* 37
3.2.4 Final remarks 38

4 Algorithm *LB-treedec* **41**
4.1 A tree decomposition approach 41
4.2 A simple description of the algorithm 43
4.3 Finding pairs in G 43
4.4 Decide which pairs to insert into T 44

4.5 Inserting a pair $\mathbf{\Psi_{xi}}$ into T 46
4.6 Data structure 47
4.7 Finding $\mathbf{N_{G_x}(x)}$ 49
4.7.1 Properties of the tree T 50
4.7.2 Grouping subtrees into different types 52
4.7.3 Finding InnerOuter and BorderOuter 52
4.7.4 Separate the InnerOuter from the BorderOuter 52
4.7.5 Finding Inner 53
4.7.6 Summing it up 53

5 Time analysis of *LB-treedec* **55**
5.1 Introduction 55
5.2 Finding pairs in G 56
5.3 Find pairs not inserted into T 57
5.3.1 Traversing the tree 58
5.4 Inserting a pair into T 59
5.4.1 The insertion 59
5.4.2 Reconnecting of edges 59
5.5 Data structure 60
5.5.1 Reading and traversing 60
5.5.2 Insertion into T 60
5.6 Finding $\mathbf{N_{G_x}(x)}$ 61
5.6.1 $\mathbf{Border_x}$ edges 61
5.6.2 Finding *U(x)* 61
5.6.3 Summing it up 62
5.7 Space analysis 62

6 Experimental results **63**
6.1 Analyzing *LB-treedec* 63
6.1.1 Is *LB-treedec* an O(nm) algorithm ? 65
6.1.2 The maximum point in *LB-treedec* time curve 67
6.1.3 The roughness of *LB-treedec* 68
6.2 *LB-treedec* versus *LB-list* 70
6.3 *Lex-M* versus *LB-treedec* 73
6.3.1 Compare computation time 73
6.3.2 Compare fill 75

7 Concluding remarks **79**
7.1 An overview of our results 79
7.2 Open questions 81

Acknowledgments

First and foremost I would like to thank my advisor Associate professor Pinar Heggernes. Her teaching and guidance have been invaluable to me in the work of this thesis, and she shall have a lot of credit for the results in this thesis.

I would also like to thank Tone M. Sæle for always being patient, and for reading the thesis.

Lastly I will thank my friends and the staff at the Algorithms Research Group for good social environment and many interesting discussions.

Chapter 1

Introduction

Many problems that originate from natural sciences, such as meteorology, oil reservoir simulation and structural engineering, can be represented as linear systems of equations. In many of these cases we can get a better model of the problem if we increase the number of variables in the linear system of equations. This gives us the motivation to solve as large systems as possible on a given computer system. Many of these problems have special sparse structures, such that one variable only depends on few others. If we use a matrix A to represent such a problem, then variables i and j that are not related will be represented by a zero entry $A[i, j]$ in the matrix A. We define a matrix as sparse, if only a few of the elements are non-zero. Larger problems can be stored in the memory of a computer if the problem is sparse, and we only store the non-zero elements. In 1961 Parter [16] presented the *Elimination game* which is a graph theoretical approach to simulate sparse matrix factorization. Given a graph G, representing a matrix A, *Elimination game* outputs a graph G^+ representing a factorization of A, where G is a subgraph of G^+. Later in 1965 Fulkerson and Gross [8] showed that the graphs resulting from *Elimination game* are exactly the class of chordal graphs. We define the new edges introduced by *Elimination game* as *fill*. The size of the *fill* depends on the order in which the vertices are eliminated, and can vary from *minimum* to a complete graph. The resulting graph G^+ is called a *triangulation* of G.

The goal is to minimize the size of the fill, since this minimizes both the space used by the result, and the work needed to complete the factorization and solve the system. Rose, Tarjan and Lueker [17] conjectured that the problem of deciding whether a given fill is minimum is NP-complete, and later Yannakakis [20] verified this conjecture. A widely accepted alternative, which has polynomial time solutions, is the problem of finding a minimal triangulation. A triangulation H of G is minimal if no subgraph of H is a triangulation of G. There exist several algorithms that solve this problem, such as [15] and [17] with the algorithm *Lex-M* that solves the problem

in $O(nm)$ time. In this thesis we are focusing on a new algorithm, *LB-triang*, described by Berry [3] in 1999. The time complexity of *LB-triang* was conjectured by Berry to be $O(nm)$. However this remained unproven since it's presentation. The implementation suggested in [3] turned out to require $O(nm')$ time, where m' is the number of edges in the chordal graph.

The original goal of this thesis was to implement and experiment with the *LB-triang* algorithm. But the unsolved question, regarding whether or not an $O(nm)$ solution to *LB-triang* exists, inspired to theoretical work. In this thesis we actually prove the $O(nm)$ time bound by presenting implementation and data structure details to achieve this bound. This result has been accepted at the European Symposium on Algorithms and will be presented in Rome, Italy, September 2002 [10]. The key to this solution is the data-structure, which is based on refining a *tree decomposition* as the minimal separators are discovered. In the end this data-structure contains a *clique tree* of the computed minimal triangulation of G. We call this new $O(nm)$ time and $O(m')$ space algorithm *LB-treedec*. We have also implemented *Lex-M*, *LB-treedec* and Berry's original $O(nm')$ time of *LB-triang*, and used these in different practical experiments. The results from these experiments show that *LB-treedec* behaves in a $\Theta(n^2)$ fashion on randomly triangulated graphs, and requires $\Theta(nm)$ for some special cases.

This thesis is organized as follows. This chapter describes the basic graph terminology, and other basic theory used in this text. Chapter 2 gives the necessary background on chordal graphs, and Chapter 3 describes *LB-triang* and Berry's $O(nm')$ implementation. Description of *LB-treedec*, and the proof that this algorithm keeps the $O(nm)$ time bound are explained in Chapter 4 and 5. Chapter 6 contains the experimental results, and their results and analysis. We conclude this thesis in Chapter 7.

1.1 Time complexity

When we talk about mathematically formulated problems it is often interesting to know how much time and space a given algorithm uses, compared to the size of the input. The notation O, Ω and Θ is commonly used to establish an asymptotic limitation.

Definition 1.1. *Let f and g be two functions $f, g : \mathcal{N} \to \mathcal{R}^+$. Say that $f(n) = O(g(n))$ if some positive integers c and n_0 exist so that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.*

Definition 1.2. *Let f and g be two functions $f, g : \mathcal{N} \to \mathcal{R}^+$. Say that $f(n) = \Omega(g(n))$ if some positive integers c and n_0 exist so that for every integer $n \geq n_0$, $cg(n) \leq f(n)$.*

Definition 1.3. *Let f and g be two functions $f, g : \mathcal{N} \rightarrow \mathcal{R}^+$. Say that $f(n) = \Theta(g(n))$ if some positive integers c_1, c_2 and n_0 exist so that for every integer $n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$.*

We can see from the Definitions 1.1, 1.2 and 1.3, that the O and Ω notation defines an upper and lower bound, while the Θ notation quite accurately describes how much time is used compared to the size of the input. The exact same notations are used to describe the space usage of an algorithm.

Some problems are more time consuming in general than others. First note the dramatic difference between the growth rate of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n. For example let n be 1000, the size of a reasonable input to an algorithm. In that case, n^3 is 1 billion, a large, but manageable number, whereas 2^n is a number much larger than the number of atoms in the universe. Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms rarely are useful. We define P as the class of problems that have polynomial time algorithms, and we define problems that have polynomial time verifications of a solution as the class NP. Classes that go beyond this will not be discussed, since they mainly have theoretical interest.

The largest unsolved problem in informatics is whether or not $P = NP$. The reason that this problem is so important is that many interesting problems are in NP, and these will have a polynomial solution if $P = NP$. One result from the work done in this area is the class of *NP-Complete* problems, which are problems in NP that are at least as difficult as all other problems in NP.

Definition 1.4. *A decision problem B is NP-Complete if it satisfies two conditions:*

1. *B is in NP*
2. *Every problem A in NP is polynomial time reducible to B*

It follows from this that $P = NP$ if and only if any *NP-Complete* problem has a polynomial time solution. A relevant example of such a problem is minimum fill, which is *NP-complete* [20].

1.2 Graph terminology

A graph $G = (V(G), E(G))$ is defined by the set of vertices $V(G)$ and the set of edges $E(G)$. We define the size of $V(G)$ as n and $E(G)$ as m, thus $|V(G)| = n$ and $|E(G)| = m$. An edge $e \in E(G)$ is a pair of vertices (u, v) where $u, v \in V(G)$. All graphs in this thesis are undirected and finite. Graphs in this text do not contain self loops, *i.e.* if $(u, v) \in G$ then we demand that

$u \neq v$. Nor do we allow any multiple edges in our graph G, *i.e.* only one instance of one edge (u, v) is allowed in $E(G)$.

A vertex u is described as adjacent or neighbor of another vertex u in a graph G if $(u, v) \in E(G)$. All adjacent vertices of a vertex x in a graph G are defined as $N_G(x)$, and if we include the vertex x in the set we denote it as $N_G[x]$. A clique is defined as a set of vertices $S \in V(G)$, where every pair of vertices $u, v \in S$ has an edge $(u, v) \in E(G)$. If $S = V(G)$ we denote the graph G as a complete, thus in a complete graph of n vertices there are $\frac{n(n-1)}{2}$ edges.

An ordering α of G is a one-to-one mapping of $\{1, 2, ..., n\}$ into $V(G)$, so α can be seen as a numbering of the vertices of G. The ordering can also be seen as a sequence of the vertices from $V(G)$, with the first vertex in the sequence being the vertex numbered 1 and so forth.

A path is a number of vertices that connect two vertices. The base case of a path can be described as an edge between two vertices. Paths can be described in a more general way as a sequence of vertices $v_1, v_2, ..., v_k$ from $V(G)$ such that for $1 \leq i \leq k-1, (v_i, v_{i+1}) \in E(G)$. The length of the path is the number of edges it contains. A path is called simple if all the vertices are different except that we may have $v_1 = v_k$, in which case we call the path a cycle. Graphs can be described as connected if there exists a path from a vertex $x \in V(G)$ to every vertex in $V(G)$. Trees are a class of graphs where the graph is connected but no subset of the vertices forms a cycle.

A subgraph H is a subset of the vertices or edges in a graph; if H is a subgraph of G, then $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. If some of the nodes or edges in G are not present in H, then H is a proper subgraph of G, *i.e.* $H \subset G$. Furthermore, we call H an induced subgraph of G if $V(H) \subseteq V(G)$, and where $(u, v) \in E(H) \Leftrightarrow (u, v) \in E(G)$ and $\{u, v\} \subseteq V(H)$. If S is a subset of $V(G)$, then the notation $G[S]$ means the subgraph of G induced by S, i. e. $G[S] = (S, F)$ where $(u, v) \in F$ iff $(u, v) \in E(G)$ and $u, v \in S$. If H is a subgraph of G, then we denote G as a supergraph of H. Furthermore if $S \subset V(G)$, then the neighbors in G of S are denoted as $N_G(S)$.

Graphs can be represented by two different data structures, as an *adjacency matrix*, or an *adjacency list*. The representations differ in space usage, access time and how to update. Adjacency matrix is the simplest representation, which requires $O(n^2)$ space, but gives us direct access to every edge. List representation is based on keeping a vector for each vertex in the graph, where the elements in the vector are the non-zero elements in the characteristic vector representing this vertex in the adjacency matrix. This only requires $O(m)$ space, and is therefore dynamicly changing space with the number of edges in the graph. Adding an edge can be done in $O(1)$ for both representations, while updating, removing or reading an edge is an $O(1)$ operation in the matrix, and $O(n)$ for the list representation. However, listing the neighbors of a vertex costs $\Theta(n)$ in a matrix, and only requires $\Theta(N(x))$ for the list representation. Since we are working on sparse graphs,

and often require the neighborhood of a vertex, then the obvious choice is list representation to minimize space and time usage.

1.2.1 Tree decompositions

The improvement of the *LB-triang* algorithm present in this thesis relies on two important structures, called *tree decompositions* and *clique trees*. A Tree decomposition is a special kind of data-structure that tries to represent the graph as a tree.

Definition 1.5. *A tree decomposition of a graph* $G = (V, E)$ *is a pair* $(\{X_i \mid i \in I\}, T = (I, M))$ *where* $\{X_i \mid i \in I\}$ *is a collection of subsets of* V*, and* T *is a tree, such that:*

- $\bigcup_{i \in I} X_i = G(V)$
- $(u, v) \in G(E) \Rightarrow \exists\ i \in I$ *such that* $u, v \in X_i$
- *For all vertices* $v \in V, \{i \mid v \in X_i\}$ *induces a connected subtree of* T

The last condition of Definition 1.5 can be replaced by the following equivalent condition:

- $i, j, k \in I$ *and* j *is on the path from* i *to* k *in* $T \Rightarrow X_i \cap X_k \subseteq X_j$.

Thus each tree node corresponds to a vertex subset X_i, also called a bag (in which the graph vertices are placed). The width of a decomposition $(\{X_i \mid i \in I\}, T = (I, M))$ is $max_{i \in I}|X_i| - 1$, and the treewidth of a graph G is the minimum width over all tree decompositions of G.

Theorem 1.1. (Arnborg, Corneil and Proskurowski [1]) *The following problems are NP-complete:*

- *Given a graph* $G = (V, E)$ *and an integer* $c < |V|$*, is the treewidth of* $G \leq c$*?*
- *Given a graph* $G = (V, E)$ *and an integer* $c < |V|$*, is the pathwidth of* $G \leq c$*?*

A path decomposition is a tree decomposition $(\{X_i \mid i \in I\}, T = (I, M))$ such that T is a path. Even though we are not capable of deciding these questions, we have some useful information about the graph G given a tree decomposition of G. The largest clique in G may not be larger than the width + 1 of our tree decomposition, and the intersection between two adjacent bags in the tree T is a separator in the graph G. A separator is a set of vertices in $V(G)$, whose removal from G results in a disconnected graph.

A clique tree is a special kind of tree decomposition, where every bag contains a maximal clique in the graph G. We will discuss this in the next chapter since clique trees are only defined for chordal graphs.

1.3 Cholesky factorization

Since the problem that we are studying is motivated by applications in sparse matrix computations, we find it useful to give some background here on *Cholesky factorization*. We start by defining a SPD matrix, since Cholesky factorization only applies to these kinds of matrices.

Definition 1.6. *A matrix A with m rows and n columns is said to be symmetric positive definite (SPD), if it satisfies the following three properties:*

- *A is square ($m = n$).*
- *A is symmetric.*
- *For any non-zero vector x we have $x^T Ax > 0$.*

We have previously talked about natural problems which are solvable via a linear system of equations. If the matrix of the linear system is SPD we may use Cholesky factorization to solve this kind of problems. The procedure consists of three steps:

1. Decompose A such that $A = LL^T$, with L lower triangular.
2. Solve $Ly = b$ for y using forward substitution.
3. Solve $L^T x = y$ for x using backward substitution.

The matrix L has non-zero elements in the positions of all non-zero elements in the lower triangle of A. In addition, L may have more non-zero elements. In this work, we are only interested in operation 1, where we find the unique Cholesky factor L. The reason for this is the fact that the new non-zero elements in L are equivalent to the fill created by a triangulation of A. A result of this is that a triangulation of A that creates little fill will keep many zero elements in L, and therefore be less work to compute. Let us describe the actual computation of L. We start by expanding $A = LL^T$ into

$$A = \begin{bmatrix} a_{11} & b^T \\ b & \hat{A} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ v & \hat{L} \end{bmatrix} \begin{bmatrix} l_{11} & v^T \\ 0 & \hat{L}^T \end{bmatrix} = LL^T, \tag{1.1}$$

where a_{11} and l_{11} are positive scalars, b,v and 0 are $(n-1)$ dimensional vectors and $\hat{A}$ as well as $\hat{L}$ are $(n-1) \times (n-1)$ matrices. From this we get

$$l_{11} = \sqrt{a_{11}}, v = \frac{1}{l_{11}} b \tag{1.2}$$

and

$$\hat{A} - vv^T = \hat{L}\hat{L}^T. \tag{1.3}$$

It is possible to prove that the new matrix $\hat{L}\hat{L}^T$ is SPD if A is SPD. We can now apply this recursive procedure until L is computed. If we compare L with the lower part of A, then the interesting information is how many non-zero elements are in L compared to A. The new non-zero elements that are added during Cholesky factorization are called *fill*. In this thesis we are studying a method that is trying to reduce fill.

Symbolic Cholesky factorization introduces the same set of non-zero elements as the regular Cholesky factorization, but does not compute the actual value of each element in the matrix. We can obtain the same fill if we use the *Elimination game* [16] on the graph $G = G(A)$. The *Elimination game* algorithm will be discussed in more detail in the next chapter, when we talk about triangulations and chordal graphs.

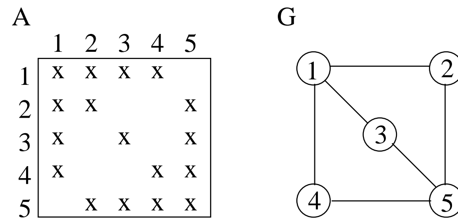

Figure 1.1: The matrix A, and the graph $G = G(A)$.

In Figure 1.1 we have an example of a matrix A, and the graph $G = G(A)$, where each x represents a non-zero element in the matrix. We are now going to compute one iteration of the symbolic Cholesky factorization.

$$a_{11} = x \qquad \hat{A} = \begin{bmatrix} x & & & x \\ & x & & x \\ & & x & x \\ x & x & x & x \end{bmatrix} \qquad b = \begin{bmatrix} x \\ x \\ x \\ \\ \end{bmatrix} \qquad b^T = \begin{bmatrix} x & x & x & \end{bmatrix}$$

We read out the values a_{11}, b, b^T and $\hat{A}$ from the matrix A. Since we are only interested in the non-zero values, then the only interesting operation is the computation of $Nz(\hat{L}\hat{L}^T)$, where Nz an abbreviation of Non-Zero.

$$Nz(\hat{L}\hat{L}^T) = Nz(\hat{A} - vv^T) = Nz(\hat{A} - bb^T) = \begin{bmatrix} x & y & y & x \\ y & x & y & x \\ y & y & x & x \\ x & x & x & x \end{bmatrix}$$

The y elements in the matrix represent the introduced fill. Since the matrix $Nz(\hat{L}\hat{L}^T)$ contains no non-zero values, then no new non-zero values may be changed through the rest of the algorithm.

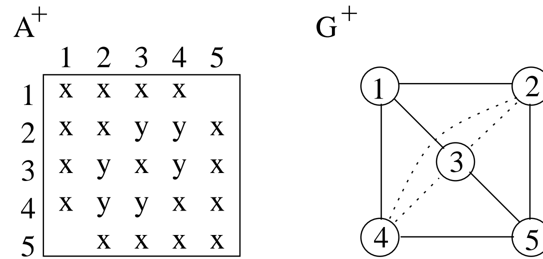

Figure 1.2: The resulting graph G^+.

In Figure 1.2 we have drawn the graph representing the non-zero elements in the new matrix A^+. The edges representing the new non-zero elements in the matrix A are the dotted lines. If we use a *permutation matrix* to perform a symmetric reordering of the rows and columns of A and apply the Symbolic Cholesky method on the equivalent system $(PAP^T)(Px) = Pb$, we may reduce the amount of fill, sometimes substantially.

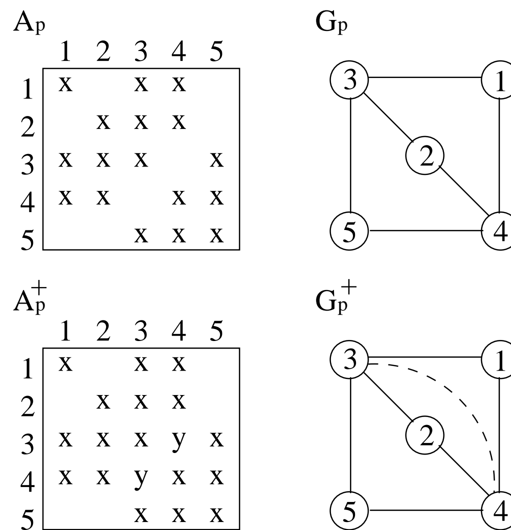

Figure 1.3: Symbolic Cholesky Factorization of a permutation of A.

The matrix A_p and the graph G_p in Figure 1.3 show a permutation of the previously discussed matrix A. This is a concrete example where we reduce the size of the fill or the number of introduced edges by doing a permutation of the matrix. The factorization of A introduced 6 new non-

zero elements, while the factorization of A_p only introduced 2. Observe that the permutation of A is equivalent to a new ordering of the vertices in the graph, and that each new edge corresponds to two new elements in the matrix.

Chapter 2

Chordal Graphs and triangulations

Chordal graphs were originally defined as an extension of trees by Gavril [9]. Gavril defined this big and important class as the intersection graphs of subtrees of a tree. In literature Chordal graphs are often referred to as triangulated graphs.

Definition 2.1. *A graph is chordal if every cycle of length ≥ 4 has a chord.*

A chord is defined as an edge joining two non consecutive vertices of a cycle. Chordal graphs arise from several practical applications, like sparse matrix computations. Another reason for the research done in this area is the restrictions in the definition, which allow us to solve many problems much faster than for general graphs.

2.1 Characterization of chordal graphs

Several different characterizations of chordal graphs exist, and we are going to mention some of them to get a better picture of the class of chordal graphs. The first characterization is by Dirac who uses the definition of *minimal separators* to characterize chordal graphs. Separators and minimal separators are often mentioned when discussing chordal graphs.

Definition 2.2. *Let $G = (V, E)$ be a graph where $u, v \in V(G)$, and let $S \subseteq V(G) - \{u, v\}$. Then S is a u, v-separator if there exists no path from u to v in $G[V - S]$.*

A direct consequence of the separator definition is that at least two different connected subgraphs will remain if a separator is removed from the graph, i.e. those containing u and v.

Definition 2.3. *Let $G = (V, E)$ be a graph where $u, v \in V(G)$, and let S be a u, v-separator. Then S is a minimal u, v-separator if no subset of S is a u, v-separator.*

Observation 2.1. *Let S be a minimal u, v-separator, then every vertex $x \in S$ has a path to u and v in $G[V(G) - S]$.*

The reason that Observation 2.1 is true is quite simple. If a vertex $x \in S$ only has neighbors in one subgraph then this vertex may be removed from S, and $S \cup \{x\}$ will still separate u from v.

Definition 2.4. *A separator S is a minimal separator in G, if S is a minimal u, v-separator for some pair of vertices $u, v \in V(G)$.*

Definition 2.5. *Let S separate the two components C_1 and C_2 in the graph G. S is a minimal C_1, C_2-separator if S is a minimal u, v-separator for every pair of vertices u, v where $u \in C_1$ and $v \in C_2$.*

A minimal separator is not necessarily a minimal u, v-separator for every pair of vertices u, v separated by the separator. In Figure 2.1 we have an example of a graph G, where the separator $S = \{x, y\}$ is a minimal u, v-separator. Observe that S also separates the pair u, w, while the minimal u, w-separator is only $\{y\}$, thus S is not a minimal u, w-separator. But observe that both $\{x, y\}$ and $\{y\}$ are minimal separators of G.

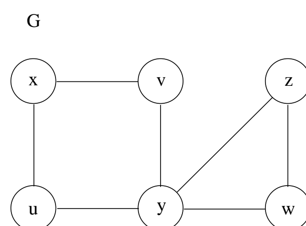

Figure 2.1: Example of separators in a graph

Characterization 2.1. (Dirac[7]) *A graph is chordal iff every minimal separator is a clique.*

Definition 2.6. *A vertex x is called simplicial if $N(x)$ induces a clique.*

Dirac defined simplicial vertices, and proved that every chordal graph that is not complete has at least two non adjacent simplicial vertices. In the special case where the graph is complete, then every vertex is simplicial. Fulkerson and Gross [8] used this and the fact that chordality is a hereditary property when they proved that every simplicial vertex may be removed from

a chordal graph without destroying chordality. Since the remaining graph is still chordal this may be done recursively. They used this to define a *perfect elimination order*(PEO), which is an order in which simplicial vertices can be removed from a chordal graph. From Dirac we know that every chordal graph has at least two simplicial vertices. This implies that there exist at least 2^n different PEOs of a chordal graph.

Definition 2.7. *The following process defines a perfect elimination order (PEO). We find the PEO of $G = (V, E)$ by removing simplicial vertices. This continues until no vertices remain.*

Many graphs do not have a PEO, a simple example is the 4-cycle. Because non of the vertices in the cycle is simplicial. Let us now describe the class of graphs that has a PEO.

Characterization 2.2. (Fulkerson and Gross [8]) *A graph is chordal iff it has a PEO.*

Lekkerkerker and Boland [14] also characterized the chordal graphs, but they used a different approach. By defining the notion of substars of a vertex x, which are sets of vertices included in $N(x)$, they characterized chordal graphs as graphs for which each substar is a clique. It has been proven that these substars are minimal separators contained in the neighborhood of the vertex x. Since every minimal separator in a chordal graph is in a vertex neighborhood, this result is in fact closely related to Dirac's characterization. We use the following definition to reformulate a result of Lekkerkerker and Boland [14]. The abbreviation LB stands for Lekkerkerker and Boland.

Definition 2.8. (Berry and Bordat and Heggernes [4])*A vertex x is LB-simplicial iff every minimal separator contained in $N(x)$ is a clique.*

Using this definition Lekkerkerker's and Boland's characterization of a chordal graph can be rewritten in the following way [4]:

Characterization 2.3. (Lekkerkerker and Boland [14]) *A graph is chordal iff every vertex is LB-simplicial.*

These characterizations give us different perspectives on chordal graphs, and are useful when working on these graphs. Both Characterizations 2.2 and 2.3 describe a chordal graph from the properties of a vertex neighborhood. The difference is that Lekkerkerker's and Boland's characterization does not require any specific order or relation between the selected vertices. The *LB-triang* algorithm described in the next chapter is based on this last characterization.

2.2 Clique graphs and clique trees

Definition 2.9. *Given a graph $G = (V, E)$, the clique graph $\mathcal{G} = (\mathcal{K}, \mathcal{E})$ of G is a graph where $\mathcal{K}$ is the set of maximal cliques in G and $(U, W) \in \mathcal{E} \iff U \cap W \neq \emptyset$, for two maximal cliques U and W in G. The weight of edge $(U, W) = w(U, W) = |U \cap W|$.*

Clique graphs are not very useful on general graphs, since these can contain $n!/(k!(n-k)!)$ different cliques of size k. It follows from this that clique graphs of general graphs can be exponentially large, and it is no surprise that finding the maximal clique for general graphs is hard. Chordal graphs on the other hand have limitations that make clique graphs useful.

Lemma 2.1. (Dirac [7]) *A chordal graph G contains at most n maximal cliques.*

We will now give an alternative proof to this lemma.

Proof. We know from Characterization [8] that every chordal graph has a PEO α. Every vertex in $u \in \alpha$ is simplicial if every vertex previous to u in α is removed. The removal of a simplicial vertex will either remove a maximal clique from G, or reduce a maximal clique with one vertex. The fact that G is connected implies that the two last vertices in α is connected, and therefore are contained in the same clique. This limits the number of maximal cliques in a chordal graph. □

This limitation alone is not enough to get a more practical representation of a chordal graph, since the clique graph may contain $O(n^2)$ edges. Gavril [9] proved that every chordal graph can be represented by a clique tree limiting the number of edges to $O(n)$.

Theorem 2.1. (Gavril [9]) *Let $G = (V, E)$ be an undirected graph, and let $\mathcal{K}$ be the set of maximal cliques of G, with $\mathcal{K}_v$ the set of all maximal cliques of G that contain vertex v. The following statements are equivalent:*

1. *G is chordal.*
2. *G is the intersection graph of a family of subtrees of a tree.*
3. *There exists a tree $T = (\mathcal{K}, \mathcal{E})$ whose vertex set is the set of maximal cliques of G such that $T[\mathcal{K}_v]$ induces a connected subtree for each $v \in V$.*

Definition 2.10. *A tree as described in Theorem 2.1 (3) is a clique tree.*

The following results give us some useful properties of clique trees.

Observation 2.2. *A clique tree has at most n nodes and $n - 1$ edges.*

This follows directly from Lemma 2.1 and the fact that T is a tree.

Theorem 2.2. (Bernstein and Goodman [2]) *Any maximum weight spanning tree of the clique graph of a chordal graph G is a clique tree of G.*

Theorem 2.3. (Ho and Lee [11]) *Given a chordal graph G and a clique tree T of G, a set of vertices S is a minimal separator of G iff $S = C_i \cap C_j$ for an edge (C_i, C_j) in T.*

From this we can conclude that for any given clique tree T of G, every edge of the clique graph not in T is a subset of or equal to an edge in T.

Corollary 2.1. (Ho and Lee [11]) *A chordal graph G has at most $n-1$ minimal separators.*

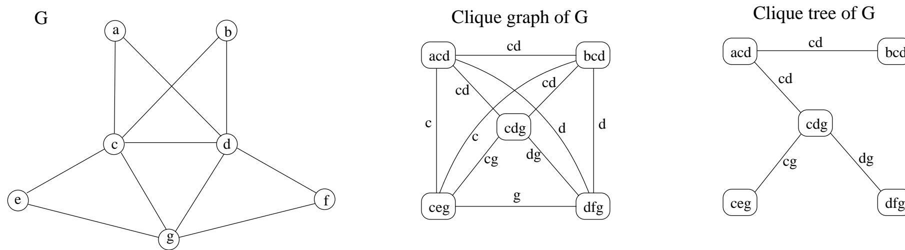

Figure 2.2: The clique graph and a clique tree of the chordal graph G.

In Figure 2.2 we have an example of a chordal graph, with the clique graph and a clique tree representation. The clique graph is unique for a given chordal graph, while the number of clique trees can be as large as 2^n. If a separator is a minimal separator for more than two components we get several different clique trees. If we remove the minimal separator $\{c, d\}$ from the graph G in Figure 2.2, then three subgraphs will remain. Since these subgraphs are connected with the same minimal separator, we get a clique of size three in the clique graph where all edges are labeled with the minimal separator $\{c, d\}$. Since all these edges have the same weight and are minimal separators of G, then any subset connecting all cliques can be used in the clique tree. This gives us an upper limit of 2^n different clique trees.

If we choose to represent a chordal graph as a clique tree, then this will require $O(n + m)$ space. We know that every chordal graph G has a PEO α, and that removing a simplicial vertex only affects one maximal clique in G. Then there exists a vertex $u \in \alpha$, for each maximal clique U in G, such that u is the first vertex in α which is also contained in U. Let us now assume that we remove the vertices in the order described by α. Then we will remove $|U|$ edges when the vertex u is removed from the graph, because

u is simplicial and has only edges to vertices in U. This ensures that the sum of all bags in the clique tree is less than or equal to m. If we add the edges needed to connect the tree nodes, then we get $m + n$ as a total.

Notice that clique trees meet all the three requirements for tree decompositions. This implies that clique trees are proper subsets of tree decompositions, since tree decompositions not necessarily are clique trees.

2.3 Elimination game

We have studied properties of chordal graphs. We are now going to study the problem of making any given non chordal graph chordal by adding edges. The chordal supergraph we obtain from this process is called a *triangulation* of the original graph. Since chordal graphs have many desirable properties, finding triangulations with special requirements is an important problem of graph theory. First we will look at Algorithm 2.1 *The Elimination game.* This algorithm was first proposed by Parter [16] as a graph theoretical approach to simulate sparse matrix factorization. Later Fulkerson and Gross [8] showed that the graphs resulting from *Elimination game* are exactly the class of chordal graphs.

Algorithm 2.1 The elimination game

Input: $G = (V, E)$ and a ordering α of the vertices in $V(G)$
Output: G^+_α
$G_{\alpha 0} = G$
for $u = \alpha(1)$ to $\alpha(n)$ **do**
  $G_{\alpha i} = G_{\alpha i-1}$
  Make $N_{G_{\alpha i}}(u)$ into a clique by adding the necessary edges
  $N_{G^+_\alpha}(u) = N_{G_{\alpha i}}(u)$
  Remove u from $G_{\alpha i}$
end for

Lemma 2.2. (Fulkerson and Gross [8]) *The class of graphs produced by Elimination game $\equiv$ the class chordal graphs.*

Elimination game produce the triangulated graph G^+_α which is also called the filled graph of G with respect to α. The advantage of Elimination game is that it has very fast implementations. In 1984 Tarjan and Yannakakis [19] introduced an $O(m')$ implementation of Elimination game. This algorithm does not make $N(x)$ into a clique when the vertex x is processed, since this alone is an $O(m')$ operation. The solution is to add only the edges from $N(x)$ to u where u is the first vertex in α that is also in $N(x)$. Meaning that u is the first vertex in $N(x)$ to be processed after x. Using this algorithm disables us from choosing the vertices in an on-line fashion. We will now present a

new version that we call *ElGame-online* and show how to implement it to run in $O(m')$ time on-line.

Lemma 2.3. *ElGame-online can be executed in an on-line fashion in $O(m')$ time.*

Proof. We prove this by constructing an algorithm that solves the problem.

Algorithm 2.2 *ElGame-online*

Input: $G = (V, E)$
Output: α,G^+_α
$H = G$
$\forall u \in V(H)$ set u as unmarked
for $i = 1$ to n **do**
  Choose an arbitrary vertex u for elimination
  $\alpha(i) = u$
  for all v such that $v \in N_H(u)$ and v is marked **do**
    for all $w \in N_H(v)$ **do**
      Insert the edge (w, u) into H
    end for
    Remove v and every edge connected to v
  end for
  $N_{G^+_\alpha}(u) = N_H(u)$
  mark u
end for

Algorithm 2.2 will clearly produce the same graphs as elimination game since the neighborhood of a vertex u in the filled graph is exactly the vertices that has a path to u in G there every vertex in this path is already selected. The proof that *ElGame-online* runs in $O(m')$ time is actually quite simple since every edge is only read twice and deleted. We do have to include the inserted edges, but since every edge is only inserted once and every inserted edge is in G^+ then the total time becomes $O(m')$. There is no numbering or labeling of the vertices which enables us to select the vertices in an on-line fashion. □

Producing a chordal graph G^+ with elimination game ensures that we get a chordal supergraph of G, and that the order α we select the vertices is a PEO of G^+. But we know very little about the quality of the triangulation with respect to the size of the fill.

Definition 2.11. *Let $G = (V, E)$ be a graph and let α be the an ordering of G. Then fill $F(G_\alpha)$ is defined as the set of edges added by Elimination game to compute the chordal supergraph G^+_α. The size of the fill F is the number of added edges, $|F|$.*

For many practical problems, finding triangulations with small fill is important to reduce work load in computations.

Lemma 2.4. (Rose,Tarjan,Lueker[17]) *Let* $G_\alpha = (V, E, \alpha)$ *be an ordered graph. Then* (v, w) *is an edge of* $G^+_\alpha = (V, E \cup F(G_\alpha))$ *iff there exists a path* $\mu = [v = v_1, v_2, \cdots, v_{k+1} = w]$ *in* G_α *such that* $\alpha^{-1}(v_i) < min\{(\alpha^{-1}(v), \alpha^{-1}(w)\}$, $2 \leq i \leq k$.

This lemma provides a characterization of the fill produced by any elimination ordering. We can use this to demonstrate how much fill *Elimination game* produces on a badly chosen input ordering. If the first selected vertex x has a neighborhood such that $|N(x)| = |V(G)|-1$, then the chordal graph will be complete. The worst example is a tree with only one non leaf tree node x, and $n-1$ leafs. Let x be the first selected vertex, then $(n^2-3n+2)/2$ edges will be added while the minimum fill is empty. From this we can conclude that no guarantees can be given about the fill created by *Elimination game*.

2.4 Maximum Cardinality Search (MCS)

Tarjan and Yannakakis [19] introduced the algorithm Maximum Cardinality Search(MCS) in 1984. This was an improvement of the Lex-P algorithm introduced by Rose, Tarjan and Lueker [17] in 1976. Both these algorithms produce an ordering α of the vertices in $V(G)$, which is a PEO iff the graph is chordal. If we are using these algorithms to decide if a graph is chordal, we have to use elimination game and the ordering α to test if any fill is added. Both the MCS and Lex-P has a time complexity of $O(n + m)$, but MCS is preferable since it is simpler than Lex-P. In Algorithm 2.3 we give a description of MCS.

Algorithm 2.3 MCS

Input: A graph $G(V, E)$
Output: An ordering α of $V(G)$
$\forall\ v \in V(G)$ set the label $L_v = 0$
Pick any vertex $v \in V(G)$ and assign v the number n
$\forall\ u \in N(v)$: $L_u = L_u + 1$
for $i = n - 1$ to 1 **do**
  Let v be an unnumbered vertex with the highest L_v
  Assign v the number i
  $\forall u \in N(v)$: $L_u = L_u + 1$
end for
{The numbering of the vertices gives the ordering α}

Inspired by the algorithm MCS, Blair and Peyton [6] modified the MCS algorithm to also be able to find the maximal cliques of the chordal graph.

The difference is that after numbering the vertex v we set the variable $k = 0$. The second change is that we set k' = the number of L_v. If $k' > k$ we are in the same clique, and if $k' \leq k$, when we start a new clique. After each incrementation of the loop we set $k = k'$. Only minor changes are required to produce the clique tree directly from this algorithm.

This is a concrete example of an NP-hard problem for general graphs, which is polynomially solvable for chordal graphs. The book Introduction to the Theory of Computation [18] by Sipser contains an NP-completeness proof for deciding whether the maximum clique is of size $\geq k$ in a general graph.

2.5 Minimal triangulations

It is often desirable to find a triangulation that creates little fill, since this keeps the problem small, and reduce work in remaining computations. An example is the Cholesky factorization problem.

In Section 2.3 we computed a chordal graph G^+ from G by adding edges or fill, and we claimed that *elimination game* not always produces little fill. Yannakakis [20] showed that computing minimum fill is NP-hard. A closely related problem, which is the main theme of this thesis, is finding a minimal triangulation.

Definition 2.12. *A triangulation H is minimal if no subgraph of H is a triangulation of G.*

This is the simple, and straight forward definition of minimal triangulation. In the paper [17] Rose, Tarjan and Lueker prove several different theorems and lemmas regarding minimal triangulation.

Theorem 2.4. (Rose, Tarjan and Lueker [17]) *Let $G = (V, E)$ be a graph, and let $H = (V, E \cup F)$ be a triangulation with $E \cap F = \emptyset$. F is a minimal triangulation iff for each $f \in F$, $H - f = (V, E \cup F - \{f\})$ is not triangulated.*

The remarkable thing about Theorem 2.4 is that we can remove one edge at the time, until minimal triangulation is achieved. This makes it easy to verify that a triangulation is minimal, and to make any triangulation minimal, by removing edges.

Lemma 2.5. (Rose, Tarjan and Lueker [17]) *Let $G = (V, E)$ be a triangulated and $f \in E$. Then either $G - f$ is triangulated or $G - f$ has a chordless cycle of length 4.*

Lemma 2.5 gives us a tool to check that a graph remains chordal when an edge is removed from a triangulated graph. In this way we do not have to recompute a PEO, to check if the graph remains chordal.

Theorem 2.5. (Rose, Tarjan and Lueker [17]) *Let $G = (V, E)$ be a graph, and let $H = (V, E \cup F)$ be triangulated. Then $f \in F$ is a minimal triangulation iff each $f \in F$ is a unique chord of a 4-cycle in H.*

A unique chord is defined as the last chord in a 4-cycle i.e if we have the 4-cycle u, v, w, z then u, w is a unique chord iff the chord v, z is not represented. Theorem 2.5 is used by Rose, Tarjan and Lueker[17] when they prove the correctness of *Lex-M*.

2.6 Lex-M

Lex-M was introduced by Rose, Tarjan and Lueker[17] in 1976. This was one of the first known algorithms that produced a minimal triangulation G^+_α of a graph G in $O(nm)$ time. A nice property of *Lex-M* is that it returns an ordering α of the vertices such that G^+_α is the same G^+_α as produced by Elimination game with G and α. When G^+_α is a minimal triangulation of G, α is called a minimal elimination ordering (MEO).

Algorithm 2.4 Lex-M

> **Input:** A graph $G(V, E)$
> **Output:** A minimal triangulation G^+_α of G
> $G^+_\alpha = G$
> $\forall\ u \in V(G)$ set $L_u = \emptyset$
> **for** $i = n$ to 1 **do**
>   Pick the unnumbered vertex v with the largest label
> $\alpha(i) = v$
>   **for** each unnumbered vertex w such that there is a path $[v = v_1, v_2, ... v_{k+1} = w]$ with v_j unnumbered and $L_{v_j} <_{lex} L_w$, $j = 2, 3, ..., k$ **do**
>   add i to L_w
>   add fill edge (v, w) to G^+_α
>   **end for**
> **end for**

In Algorithm 2.4 the labels are compared lexicographically. Thus $L_u <_{lex} L_v \Leftrightarrow$ The list L_u is lexicographically smaller than the list L_v.

Theorem 2.6. (Rose,Tarjan,Lueker[17]) *Lex-M produces a minimal triangulation G^+_α of a graph G.*

We will now discuss some of the properties of the *Lex-M* algorithm. The first observation is that the numbered vertices always form a connected subgraph of G. To explain this we take a closer look at the circumstances when the first number is added to the label L_u of a vertex u. Let us assume that the vertex v is given the first number added to L_u, when there exists

a lower labeled path from v to u in the graph. But since $L_u = \emptyset$ before the number of v is added, then it follows that a direct edge from v to u is the only possible path where every vertex in the path has a lower label than both u and v. A result of this is that G_α^+ is a supergraph of G since every edge in $E(G)$ is considered as a lower numbered path. It follows from this that the set of unnumbered vertices with a non-empty label is the neighborhood of the numbered vertices. If we on the contrary assume that the numbered vertices are not connected, and let u be the first numbered vertex that is not connected to the previous numbered vertices, then $L_u = \emptyset$, which is a contradiction since the graph is connected.

We can now conclude that there exists a path through numbered vertices between every pair of vertices which has a non-empty label, and we can also conclude that *Lex-M* only introduces a new edge (u, v) if u is the vertex with the highest label and the label $L_v \neq \emptyset$. An edge (u, v) is represented as the number of u in the label of v if $L_u \geq L_v$, and we know that the edge (u, v) is represented in the input graph if $L_v = \emptyset$.

A consequence of this is that every edge (u, v) introduced by *Lex-M* is a chord of some cycle in G. We have already argued that there exists a numbered path from u to v, and that both $L_u \neq \emptyset$ and $L_v \neq \emptyset$ since $(u, v) \notin E(G)$. We know that *Lex-M* only introduces an edge or adds the number of u to the label L_v if there exists an unnumbered path from u to v where the label of every vertex in the path has a lower label than L_u and L_v. The result is that *Lex-M* only introduce an edge (u, v) if there exist two disjoint paths from u to v.

2.6.1 An example

We show an example of how *Lex-M* works in Figure 2.3. In this case vertex c is chosen as the last vertex in α, and apart from this we let *Lex-M* choose the remaining order of the vertices. The bold face numbers indicate the numbering of the vertices i.e. α. But *Lex-M* also produces the triangulated graph, this information is stored in the labeling of the vertices when the algorithm halts. If the edge $(u, v) \in G^+$ and $\alpha^{-1}(v) > \alpha^{-1}(u)$ then $\alpha^{-1}(v) \in L_u$, i.e. if v is numbered higher than u, then the number of v is in L_u. With this information we are able to compute the triangulated graph directly from the labeling.

2.6.2 Limitations of Lex-M

Lex-M uses a lexicographic ordering scheme which is a special type of breadth-first search. A result of this is that all vertices at level i from the last vertex in α are chosen before the first vertex at level $i + 1$. This limits the number of different possible triangulations created by *Lex-M*. We are allowed to choose the last vertex in the ordering α, and apart from this we

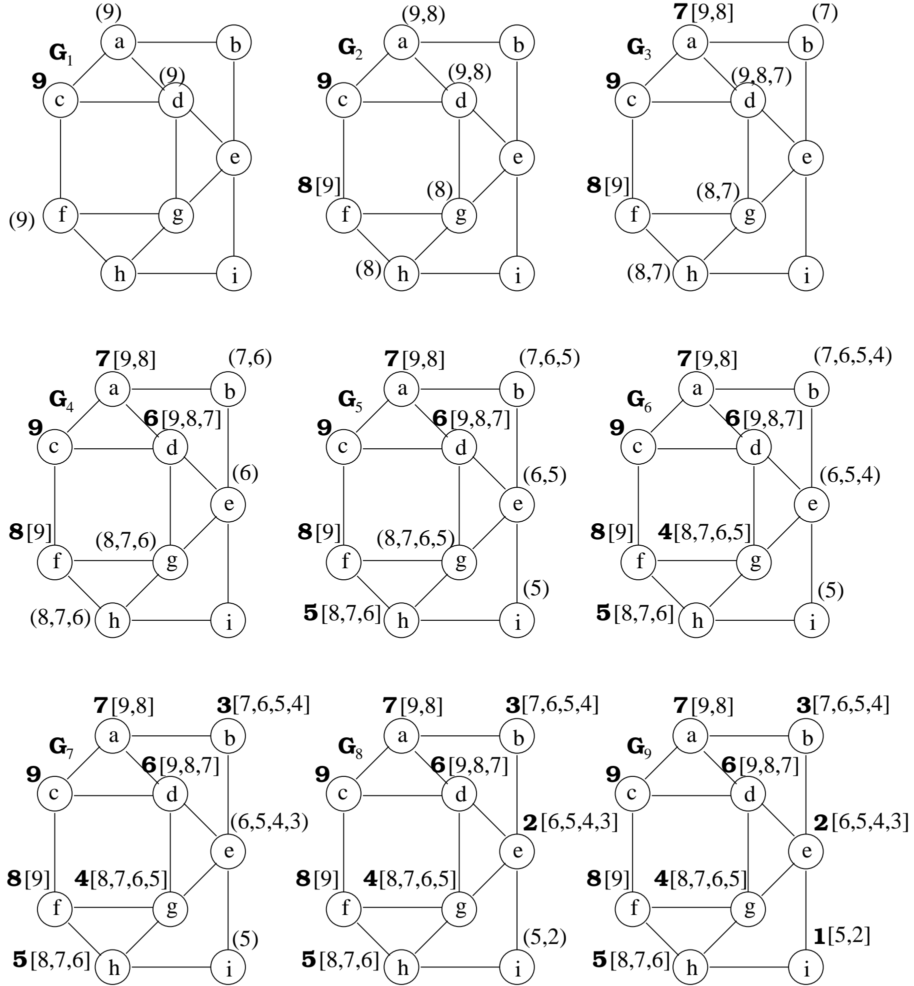

Figure 2.3: An execution of Lex-M on the graph G.

only have an option when several vertices have the largest labeling. The result is that *Lex-M* is not capable of producing some triangulations, including the minimum for some graphs.

We illustrate this by an example. In Figure 2.4 we have an example of such a graph. The original graph is indicated by the solid lines in the figure. The minimum triangulation of this graph is achieved by adding the edges $(c, d), (c, g), (d, g)$ in the center. Since the graph is symmetric we only have to consider two cases, that is when e and c is the last in the ordering α. The numbering defines the level from the last vertex in α. For graph A, where e is selected to be last in α we end up with d as the first vertex in

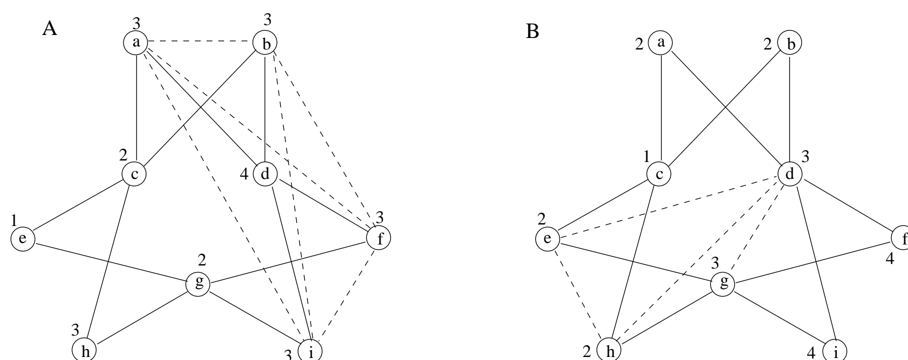

Figure 2.4: The number of the vertices indicates the level of each vertex related to the vertex e in graph A and c in graph B. The solid lines are the edges in the original graph G.

α, since d is the only vertex at level 4. *Lex-M* makes every vertex at one level simplicial before any vertex at the next level. This implies that *Lex-M* makes the neighborhood of d into a clique, as the dashed lines described at graph A.

The second case is drawn in graph B. We use c as the last vertex in α. This leveling forces us to fill in the edge $\{g, d\}$ which is one of the edges in the minimum fill-in. Next we either have to pick the vertex d or g since these are the only ones at level three. Since the graph is symmetric we get the same number of fill edges, undependently of how we choose the first vertex. In this case we choose g, and the created fill is the dashed lines in graph B. We can observe from the dashed lines that this is not a minimum triangulation. This proves that *Lex-M* is unable of finding the minimum triangulations for all graphs with a star structure.

Chapter 3

LB-triangulation

In this chapter, we will study a newer algorithm for minimal triangulations, *LB-triang*. The algorithm *LB-triang* was originally described by Berry [3] in 1999. This is an algorithm that produces a minimal triangulation G_α^{LB} of any given graph $G = (V, E)$. G_α^{LB} is defined as the chordal graph obtained by making the vertices of G *LB-simplicial* in the order described by α. The algorithm is based on Lekkerkerker's and Boland's Characterization 2.3 of chordal graphs, and use the definition of LB-simplicial vertices, to ensure the chordality of the resulting graph. We define a set of vertices as saturated if the set induces a clique in the graph after adding the necessary fill edges. As a reminder, a vertex x is *LB-simplicial* if every minimal separator in $N(x)$ is saturated. Note that this algorithm can process the vertices in an arbitrary order. Thus any order can be chosen by the user, and this order can even be supplied in an on-line fashion, if desired. At the end of an execution, $\alpha = x_1, x_2, ..., x_n$ is the order in which the vertices have been processed.

Definition 3.1. *Given a graph $G = (V, E)$ and an ordering α of $V(G)$, then G_u is the graph obtained after every vertex previous to $u \in \alpha$ is made LB-simplicial.*

3.1 Basic triangulation

The basic steps of the algorithm are quite simple. Pick the next vertex u from α, and make u *LB-simplicial.* This operation will introduce new edges between vertices in $N(x)$. This set of edges is called LB-deficiency.

Definition 3.2. (Berry et. al. [5]) *The deficiency of a vertex x in a graph G, $Def_G(x)$, is the set of edges that has to be added to G to make x simplicial. The LB-deficiency of a vertex x in G, $LBDef_G(x)$, is the set of edges that has to be added to G to make x LB-simplicial.*

The new edges will give some of the remaining vertices a larger neighborhood, which forces us to consider these when we find the neighborhood

Algorithm 3.1 Basic LB-triangulating of a graph G

Input: A graph $G = (V, E)$
Output: A minimal chordal graph G_α^{LB}
$G_0 = G$
for $i = 1$ to n **do**
  Choose an arbitrary vertex u that is not processed
  $G_{i+1} = G_i \cup LBDef_{G_i}(u)$
end for$G_\alpha^{LB} = G_n$

of the remaining vertices.

3.1.1 The correctness of *LB-triang*

We are not going to repeat the entire proof from [3] and [5], but we will talk about the idea behind the proof, and mention some of the results. The correctness of *LB-triang* is based on two steps. The first is to show that every vertex is *LB-simplicial* when the algorithm stops, which leads to chordality, according to Lekkerkerker and Boland Characterization 2.3.

Invariant 3.1. (Berry et. al. [5]) *If a vertex u is LB-simplicial in G_u, then u is LB-simplicial for all graphs G_v, where $v > u$.*

Lemma 3.1 follows directly from Invariant 3.1, since $G_n = G_\alpha^{LB}$.

Lemma 3.1. (Berry et. al. [5])*The graph G_α^{LB} resulting from Algorithm LB-triang is a triangulation of G.*

An interesting intermediate result from the proof of Lemma 3.1 is that no edge is added incident to a vertex after it has been made *LB-simplicial*.

Lemma 3.2. (Berry et. al. [5]) *During Algorithm LB-triang, if a vertex u is LB-simplicial in G_u, then no edge is added incident to u at step $v \geq u$.*

Definition 3.3. (Kloks, Kratsch, and Spinrad [12]) *A pair of separators S and S' are defined as crossing if a pair of vertices $u, v \in S$ are separated by S'.*

Secondly we have to ensure that the triangulation actually becomes minimal. This is done by proving that only non-crossing separators are saturated, which corresponds to Kloks's, Kratsch's and Spinrad's [13] description of minimal triangulation. Two lemmas are used to ensure this. The first states that the separators in the neighborhood of a vertex do not cross, and the second states that a new minimal separator does not cross any previous saturated separator.

Lemma 3.3. (Berry et. al. [5]) *When vertex u is processed, the set of minimal separators included in $N_{G_u}(u)$ are pairwise non-crossing.*

Lemma 3.4. (Berry et. al. [5]) *When a minimal separator S is chosen at step i, S is non-crossing with all the minimal separators chosen at some previous step.*

Theorem 3.1 follows directly from these lemmas and the invariant.

Theorem 3.1. (Berry et. al. [5]) *Algorithm LB-triang computes a minimal triangulation of the input graph.*

Property 3.1. (Berry, Bordat and Heggernes [4]) *For a vertex x in a graph G, the set of minimal separators of G included in $N(x)$ is exactly $\{N(C) \mid C \in C(N[x])\}$.*

Definition 3.4. (Ohtsuki, Cheung and Fujisawa [15]) *An ordering α is defined as minimal elimination ordering* (MEO) *if G^+_α is a minimal triangulation of G.*

There exist several interesting properties regarding *LB-triang*, but one theorem will be mentioned especially since we will use it later in this thesis.

Theorem 3.2. (Berry et. al. [5]) *Given a graph $G = (V, E)$, and an ordering α of the vertices in $V(G)$, then $G^{LB}_\alpha = G^+_\alpha$ iff α is a MEO.*

In other words *Elimination game* and *LB-triang* will create the same triangulation if they where given a graph G and a MEO of the vertices in the graph.

3.1.2 An example of *LB-triang*

In Figure 3.1 we have an example of a graph G_A which is triangulated using *LB-triang*. Let the numbering of the vertices describe the ordering α. The graphs G_B, G_C and G_D correspond to G after the vertices $1, 2, 3$ have been processed. Since no edge is introduced after adding the $LBDef_G(3)$, then this makes $G^{LB}_\alpha = G_D$.

Table 3.1 describes each stage of the *LB-triang* algorithm. Each row in the table has four values, the number of the processed vertex **Vnr**, the neighborhood of the vertex **N(Vnr)**, minimal separators in the neighborhood **Min sep** and finally the new edges **new edges** to saturate the separators.

Observe that a single minimal separator can be found several times during the execution of *LB-triang*. We can also find separators that are subset of, superset of or intersects with previously found separators. The separators $\{3, 5\}, \{4, 5, 9\}, \{5, 7\}$ and $\{5, 9\}$ in Figure 3.1 are examples of this. Only crossing is not allowed since this breaks the minimal triangulation. Vertex 5 is a special case since $N[5] = V(G)$. A direct consequence of this is that $N(5)$ contains no minimal separators, because there are no vertices left for the components.

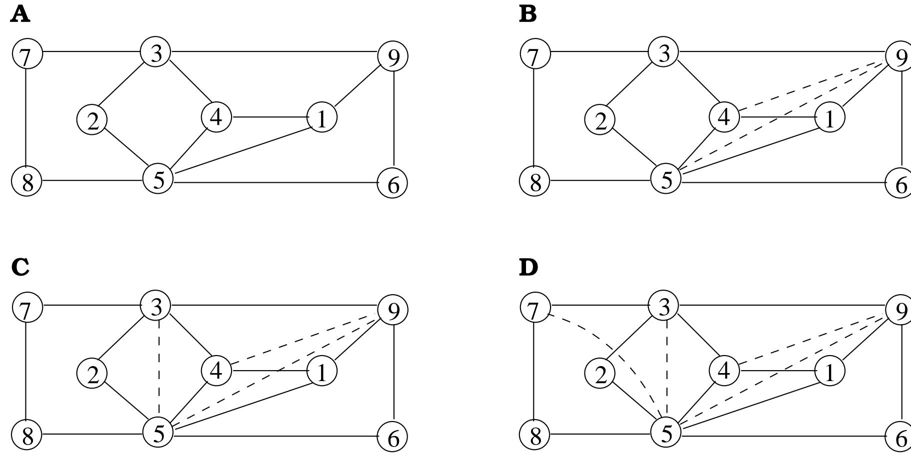

Figure 3.1: Example of triangulating a graph using LB-triang.

Vnr	N(Vnr)	Min sep	New edges
1	{4,5,9}	{4,9},{4,5,9}	(4,9),(5,9)
2	{3,5}	{3,5}	(3,5)
3	{2,4,5,7,9}	{4,9},{5,7},{5,9}	(5,9),(5,7)
4	{1,3,5,9}	{3,5},{5,9}	
5	{1,2,3,4,6,7,8,9}		
6	{5,9}	{5,9}	
7	{3,5,8}	{3,5}	
8	{5,7}	{5,7}	
9	{1,3,4,5,6}	{3,5}	

Table 3.1: Specify the triangulation of the graph in Figure 3.1.

3.1.3 Straight forward implementation of *LB-triang*

By updating the graph with adding the new edges, we can do a straight forward implementation of *LB-triang*. Since there exists no simple way to compute overlap between different separators, we end up with saturating all discovered separators even if they have been saturated previously. Checking whether an edge exists before writing it will not improve the time complexity since both operations are $O(1)$.

We can now assume that every previously found separator is saturated, and therefore finding $N_{G_\alpha^{LB}}(u)$ is easy since we have G_u, and $N_{G_u}(u) = N_{G_\alpha^{LB}}(u)$.

The final operation is to find the minimal separators in $N(x)$. Let x be

the next vertex to be made *LB-simplicial*. The simplest way to visualize how we find the minimal separators in $N(x)$, is to assume that $N[x]$ is removed from the graph. This will leave one or more subgraphs as long as $|N[x]| < |V(G)|$.

Algorithm 3.2 Straight forward *LB-triang*

Input: A graph $G = (V, E)$
Output: A minimal chordal graph G_α^{LB}
$G_0 = G$
for all $u \in V(G)$ **do**
  $G_u = G_{u-1}$
  for all Components $C \in G_u - N_{G_u}[u]$ **do**
    Update G_u with the edges needed to saturate $N_{G_u}(C)$
  end for
end for
$G_\alpha^{LB} = G_n$

3.1.4 Time and space analysis

Let us first do the time analysis of this straight forward version of *LB-triang*. The first observation is that we run through all the vertices, and find minimal separators in their neighborhoods and saturate them. Finding them is an $O(m')$ operation since we risk to read the entire graph, where m' is the number of edges in the triangulated graph G_α^{LB}. We have to use m' because we update the graph with new edges during the algorithm. Since the number of minimal separators in the neighborhood of one vertex is limited to n and each of them can introduce m' edges, then the time used to saturate is $O(nm')$ for each vertex. This gives us a total of $n(m' + nm') = O(n^2m')$ for the straight forward implementation of *LB-triang*.

The algorithm is based on saturating the separators, which is the most time consuming part. If we are going to keep this time complexity, then updating an edge has to be an $O(1)$ operation. This forces us to use an adjacency matrix to store the graph, since an adjacency list representation of the graph does not support constant update of an edge. An adjacency matrix requires $O(n^2)$ space, which gives us the space complexity of the straight forward *LB-triang*.

3.2 Improving the implementation

LB-list is an improvement of *LB-triang* which is described by Berry et. al. [5]. The idea is to reduce the number of saturations. This can clearly be done since a single separator is often saturated several times. We can see that this actually happens several times in the example in Figure 3.1 and Table 3.1.

The fact that every chordal graph contains less than n minimal separators, and that saturating a maximal set of non crossing minimal separators leaves a triangulated graph, ensures us that the number of different minimal separators are less than n. If we now only saturate each separator once, then this will only require $O(nm')$ time. This leaves us with the challenge of finding an efficient way to decide if the separator is already saturated.

3.2.1 Decide if the separator is new

It is clearly preferable to accomplish this in $O(nm')$ time since this is the time used by the saturation. In order to decide if a separator has been saturated before, we store the separators in a main separator list M_S. Further let the separators in M_S be stored in a lexicographic order. We now want to limit the size of M_S to m'.

Lemma 3.5. *The total sum of the sizes of all minimal separators in a chordal graph with m edges is less than or equal to m.*

Proof. Every chordal graph $G = (V, E)$ has a PEO α. Let u be the first vertex in α and therefore simplicial. It follows that $N_G(u)$ contains a minimal u, v-separator for any vertex $v \notin N_G[u]$. Let S be the u, v-separator in $N_G(u)$, where $S \subset N_G(u)$ and $|S| \leq |N_G(u)|$. If we now remove u from G, then the number of removed edges is greater or equal to the size of the removed minimal separator. We can now do this for all vertices in $V(G)$ in the order described by α, and the number of removed edges will always be at least the sum of the sizes of the removed minimal separators. □

Since the number of minimal separators in the neighborhood of a vertex is limited by n, then this limits the number of discovered minimal separators to n^2. The sum of the sizes of all minimal separators in the neighborhood of a vertex is clearly limited to m', since each vertex in a minimal separator has an edge to a vertex in the component it is separating. Let x be the next vertex to be made *LB-simplicial*, and let the minimal separators in $N(x)$ be inserted into the separator list SL_x. We can now sort the separators, and the vertices in each separator in SL_x by using *counting-sort* in $O(m')$ time. Finally we traverse the lists M_S and SL_x, and update M_S with the new separators in SL_x.

3.2.2 Using an adjacency matrix

In the matrix solution we saturate a new separator when we insert it into the M_S list. This makes the operation of finding $N_{G_x}(x)$ easy since we read it directly from the graph G_x. The catch is that we store the minimal separators both in the list M_S and the graph.

Time and space analysis for adjacency matrix implementation

Finding the components and the minimal separators is an $O(m')$ operation since we update the graph with new edges. Next we sort the vertices in the separators which is an $O(m')$ operation. Saturating a separator is an $O(m')$ operation since the clique may contain more edges than m. Sorting the vertices and the separators in each SL_x list is an $O(m')$ operation size we use *counting-sort*, and that the list may contain m' vertices. All of these operations are executed n times, and we will therefore get a total time complexity of $O(nm')$.

The matrix and main separator list are the only space users in this algorithm. From Lemma 3.5 we know that the size of the main list is limited to m'. When we add the $O(n^2)$ space of the matrix representation of the graph, we get a total size of $O(n^2 + m') = O(n^2)$.

3.2.3 Using an adjacency list *(LB-list)*

The list solution does not saturate the separators when they are inserted into the list M_S. This results in faster update after new separators have been found. But we have to pay it back when we are going to find the neighborhood of the next vertex x. We compute this by finding $N_{G_x}(x+1) = N_G(x) \cup U(x)$.

Definition 3.5. *Let us assume that we are at stage x and we want to compute G_{x+1}. Then $U(x)$ is defined as the union of all minimal separators containing x that are found in any previous stage.*

We have to read the separator S one time for each of the $|S|$ vertices in the separator in order to find the neighborhood of each of the vertices. The work of reading a separator $|S|$ times is exactly the same work as updating the $|S|(|S| - 1)/2$ edges. We can conclude that the work of the list, and matrix solutions are equivalent.

Time and space analysis of adjacency list implementation

The only difference in the time analysis from the matrix solution, is that we skip the saturation of separators and do this work later, when we compute $U(x)$. We can now conclude that the list solution has the same time complexity as the matrix solution i.e. $O(nm')$.

When we compare the space usage between the list and matrix solution, then the only significant difference is that list solution does not update the graph. Because of this we can store the graph as an adjacency list. This results in a total space usage of $O(m')$, since the graph uses $O(m)$ space, and the main list M_S uses $O(m')$.

As a remark we can observe that list uses less space than matrix, while matrix has a lower constant since saturation is a simpler operation than finding $U(x)$.

3.2.4 Final remarks

We can do several improvements to the algorithm *LB-list*, but non of them are able to improve the saturation of separators, or the computing of $U(x)$. The result of this is that the best known implementation of *LB-list* is $O(nm')$.

We are now going to prove that the component search, sorting of the separators and updating of M_S can be done in $O(m)$ time. Berry claims in [3] that the component search can be done in the original graph, but no proof is given. We will now present a proof that states that this actually is the case.

Observation 3.1. *Let H be a supergraph of G. Then every separator S in H is also a separator in G.*

Proof. This follows directly from the fact that $G[V(G) - S] \subseteq H[V(H) - S]$. □

Lemma 3.6. *Let H be a supergraph of G, where a set of minimal separators in G are saturated in H. Then every edge in $E(H) - E(G)$ is a chord of some cycle in G.*

Proof. We know that a minimal separator separates two components, where every vertex in the separator has a neighbor in both of the components. It follows directly from this that for every pair of vertices u, v in the separator there exists a path from u to v in both components. Because there is a path in both components for every pair of vertices in the separator, then clearly every added edge is a chord of some cycle. Since the order of saturation does not affect the set of added edges, then the same argument holds for a set of minimal separators. □

Lemma 3.7. *Let H be a graph with a minimal separator S' that is a clique. Then no minimal separator S in H crosses S'.*

Proof. We prove this by contradiction. Let us assume that S is a minimal separator for the components C_1 and C_2, and that S' is a minimal separator for the components C'_1 and C'_2. Let us further assume that S and S' are crossing, i.e. there exists a pair of vertices $u, v \in S'$ such that u, v are separated by S. But since S' is a clique, then the edge $(u, v) \in E(H)$. This is a contradiction since neighbors can not be separated. □

Lemma 3.8. *Let G be a non-chordal graph, and let a minimal separator S' of G be saturated to result in a new graph H. Then all minimal separators of H are minimal separators of G.*

Proof. We prove this by contradiction. Let us assume that $S, S' \in V(G)$ and that S is not a minimal separator of G. Let us further assume that $S \neq S'$, and that S becomes a minimal separator of H, after the saturation of S'. We know from Lemma 3.7 that S does not cross S', since S' is saturated and S is a minimal separator of H. Let us assume that S is a minimal separator for the components C_1 and C_2, and that S' is a minimal separator for the components C'_1 and C'_2. We can further assume that S' and C'_1 are contained in $C_1 \cup S$, since S and S' do not cross. Every vertex $x \in S$ has an edge to a vertex in C_1 and C_2 after the saturation of S' since S is minimal separator in H. Since S is not minimal before the saturation of S', then there exists a vertex $u \in S$ that does not have an edge to any vertex in C_1 before the saturation. Further let the edge (u, v) be introduced during the saturation, where $v \in C_1$. It follows from this that $u, v \in S'$. Then there exist two disjoint paths from u to v in C'_1 and C'_2, since S' is a minimal C'_1, C'_2-separator. Since C'_1 contains a path from u to v, and C'_1 is totally contained within C_1, we now have a contradiction to the fact that u does not have any neighbors in C_1. □

Corollary 3.1. *Let H be a supergraph of G, where a set of non-crossing minimal separators in G are saturated in H. Then every minimal separator in H is also a minimal separator in G.*

Proof. This follows directly from Lemma 3.8. □

Lemma 3.9. *Let H be a supergraph of G, where a set of non-crossing minimal separators in G are saturated in H. Then $N_H(C) = N_G(C) = S$ for every minimal separator S in H.*

Proof. Let us assume on the contrary that S does not create the same components in G and H. There must exist two components C_1 and C_2 such that S separates C_1 and C_2 in G, and not in H. It follows from this that there exists a saturated separator S' in H, that contains vertices from both C_1 and C_2. This is a contradiction since S and S' will be crossing. □

Lemma 3.10. *Let H be a supergraph of G, where a set of non-crossing minimal separators in G is saturated in H, and let S be a minimal C_1, C_2-separator in H. Then $N_G(C_1) = N_H(C_1)$ and $N_G(C_2) = N_H(C_2)$.*

Proof. This follows directly from the fact that S is a minimal C_1, C_2-separator in both G and H, and that every vertex in a minimal separator has edges to both the components separated by the separator. □

It follows directly from Lemma 3.1 that we get the same result if we do the component search in G or H, and that the size of LS_x for any vertex $x \in V(G)$ is limited to m. If each separator in the main list has a pointer to the first different vertex in the next separator, then we do not have to read the end of the current separator, and the beginning of the next. This enables us to update M_S with LS_x in $O(m)$ time.

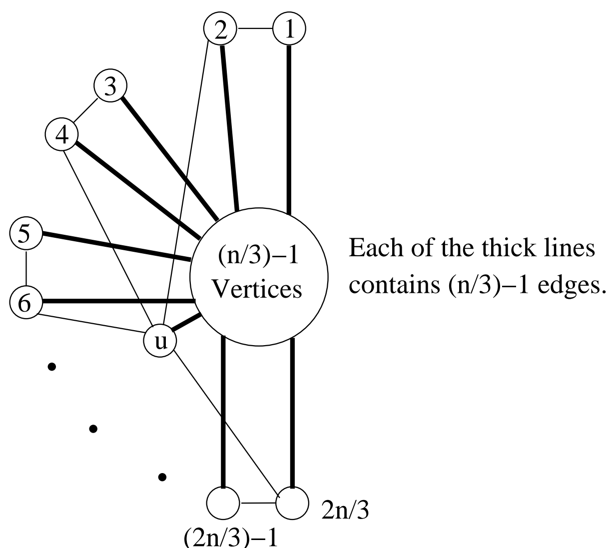

Figure 3.2: A graph that computes $\frac{n}{3}$ separators of size $\frac{n}{3}$.

We will now give an example of how difficult it is to create a system of the overlap between the separators. In Figure 3.2 we have a graph that will create $\frac{n}{3}$ different separators of size $\frac{n}{3}$. While the number of edges is $\frac{2n}{3} + (\frac{2n}{3} + 1) * (\frac{n}{3} - 1) = \frac{n}{9}(2n + 3) - 1$. If we now randomly remove edges from the thick lines which originally contain $(\frac{n}{3} - 1)$ edges. It gets quite complex to find an efficient algorithm to compute $U(x)$ in $O(m)$ time, since the size of m may have dropped below $O(n^2)$. We are doing this example to illustrate how bad the input graph can make the algorithm behave. However, it should be noted that this is a special graph, and that on most graphs the overlap will not be this heavy.

Chapter 4

Algorithm *LB-treedec*

In this chapter we present Algorithm *LB-treedec* which is the main contribution of this thesis, and prove that *LB-treedec* is an $O(nm)$ time implementation of the algorithm *LB-triang*.

The algorithm *LB-treedec* requires a graph $G = (V, E)$ and an ordering α of the vertices in $G(V)$ as input parameters. We denote the resulting graph as G_α^{LB}. Before we start the explanation we give the following definition to help in the coming discussions.

Definition 4.1. *We define stage x, as the point where every vertex previous to x in α is made LB-simplicial, and the current stage is to make x LB-simplicial.*

Since we want to achieve $O(nm)$ time complexity, we can only afford to do $O(m)$ work at each stage. In the previous chapter we described the algorithm *LB-list* which achieved a time complexity of $O(nm')$. The operation that causes *LB-list* to use $O(nm')$ is the computation of $N_{G_x}(x) = N_G(x) \cup U(x)$ where $U(x)$ is defined in Definition 3.5. In order to achieve our requested time limit we have to avoid reading every separator that contains a vertex x in order to compute $U(x)$.

4.1 A tree decomposition approach

Our goal in general *LB-triang* is to identify the minimal separators of the resulting minimal triangulation. Before we have any information of the separators, we may treat the triangulated graph G_α^{LB} as complete. As we encounter the minimal separators, we get more and more information on the non-existing edges in G_α^{LB}. This way of viewing the problem enables us to attack the problem with a different approach. But we have to find an appropriate way to store which vertices that do not have edges between them. Clearly we can use a compliment graph to store these edges, but this will bring us too close to the previous approach. Instead, we group the vertices in

bags such that graph edges only exist between vertices that appear together in the same bag.

This is actually quite close to the definition of *tree decompositions*, and we can introduce the remaining restrictions of *tree decompositions* without any conflict with our previous requirements. An intuitive explanation of how this is done in *LB-treedec* is as follows: In the beginning of the algorithm, all vertices of G are placed in one bag, and the tree T that represents the triangulated graph consists of one tree node which is equivalent to this bag. As we encounter new minimal separators at each step of the algorithm, we insert these as edges in T, splitting the appropriate tree-node into two new tree-nodes.

At each step of the algorithm the tree T is actually also a tree decomposition of G, which will be proved later in this chapter. After every minimal separator found in G is inserted into T, then T is a clique tree representing the minimal triangulated graph G_α^{LB}.

The purpose of the tree T is to allow us to compute $U(x)$ efficiently. But since the operation of computing $U(x)$ from T is quite complex, we will first explain how to update the tree every time we find new minimal separators, and then how to find $U(x)$.

When processing vertex x every minimal separator $S_{xi} \subseteq N_{G_x}(x)$ separates a component C_{xi} from x and every other component found at this stage. We use the separator S_{xi} and the component C_{xi} to form a pair $\Psi_{xi} = (S_{xi}, C_{xi})$, and by using the pair Ψ_{xi} we are able to define the inverse component $C_{xi}^{-1} = G - (S_{xi} \cup C_{xi})$. S_{xi} thus separates C_{xi}^{-1} and C_{xi}, where C_{xi}^{-1} may be disconnected.

To be able to use the tree T we need to update T every time a new pair Ψ_{xi} is found. From the definition of the separator and the component, we have some invariants regarding a pair Ψ_{xi} found at stage x.

Invariant 4.1. *The following is always true.*

In a pair $\Psi_{xi} = (S_{xi}, C_{xi})$, S_{xi} is a minimal x,u-separator where u is any vertex in C_{xi}.

- *S_{xi} separates every vertex in C_{xi} from every vertex in C_{xi}^{-1}.*

First some general information about how the tree is updated. Every step will be carefully explained later. In the base case there is only one tree-node X containing every vertex in $V(G)$. Let C_{xi} be a component found when making vertex x LB-simplicial. We use C_{xi} to form the pair $\Psi_{xi} = (S_{xi}, C_{xi})$. If this pair is not already inserted into T there exists a tree-node $X \in T$ that contains at least one vertex from C_{xi} and one vertex from C_{xi}^{-1}. To update T we split the tree-node X into X_1 and X_2 in such a way that the vertices in C_{xi} and C_{xi}^{-1} are separated. We connect X_1 and X_2 with a tree-edge containing the separator S_{xi}. Further, we replace X with X_1 and X_2 and reconnect every incident tree-edge to X to either X_1 or X_2.

In Figure 4.1 we have an example of a graph G where the pair Ψ_{xi} is inserted into T. At *stage* 1 the tree is in the base case where every vertex in G is in the same tree-node. In *stage* 2 vertex 1 is made LB-simplicial and the component $\{4\}$ is discovered. This component creates the pair $\Psi_{11} = (\{2,3\},\{4\})$ where $\{2,3\}$ is the minimal separator. If there exists a tree-node $X \in T$ containing vertex 1 and at least one vertex from the component, then X has to be split. In this example there exists a tree-node containing 1 and 4. In *stage* 2 we can see the tree T after the insertion of the pair Ψ_{11}.

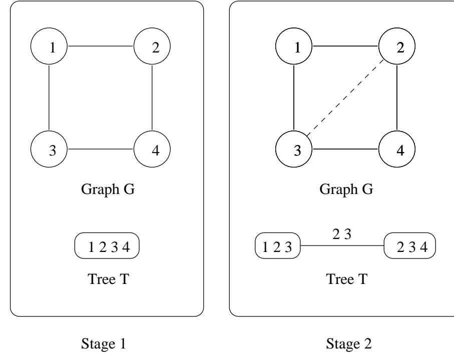

Figure 4.1: Example of inserting a pair Ψ_{11} into T

4.2 A simple description of the algorithm

In Algorithm 3.1 the basic steps of *LB-treedec* is presented. Observe that each of the steps in the outer for loop has to execute in $O(m)$ time to achieve a total time complexity of $O(nm)$. Some of these steps need an amortized time analysis to prove the $O(m)$ time complexity.

Because finding $U(x)$ is the most complex part and that it is depending on all the other steps, this will be explained last. For now let us assume that we are able to find $U(x)$ from the tree T in $O(m)$ time.

4.3 Finding pairs in G

At this point we are at stage x, and have already found the new neighborhood $N_{G_x}(x)$ of vertex x. Our goal is to return a pair Ψ_{xi} for each connected subgraph left after removing $N_{G_x}[x]$ from G. Each component C_{xi} is used to form a pair $\Psi_{xi} = (N(C_{xi}), C_{xi}) = (S_{xi}, C_{xi})$.

Algorithm 4.1 *LB-treedec*

Input: $G = (V, E)$ and an ordering α of $V(G)$.
Output: G_α^{LB}
for $x = \alpha(1)$ to $\alpha(n)$ **do**
  Compute $U(x)$ from the tree T
  $N_{G_\alpha^{LB}}(x) = N_G(x) \cup U(x)$
  for each connected component C_{xi} of $G - N_{G_\alpha^{LB}}(x)$ **do**
    $S_{xi} = N_G(C_{xi})$
    $C_{xi}^{-1} = V(G) - (C_{xi} \cup S_{xi})$
    $\Psi_{xi} = (S_{xi}, C_{xi})$
    $\Psi_{xi}^{-1} = (S_{xi}, C_{xi}^{-1})$
    if (Ψ_{xi} or Ψ_{xi}^{-1}) is not already inserted into T **then**
      Insert Ψ_{xi} into T
    end if
  end for
end for
Return G_α^{LB}

4.4 Decide which pairs to insert into T

We have now found the pairs $\Psi_{x1}..\Psi_{xk}$, and want to update the tree T with these new pairs. Some of these pairs may have been found at some stage previous to x, which forces us to detect whether or not each of the k different pairs have been inserted before. First we will look at which pairs that need to be inserted.

Lemma 4.1. *Let the pair $\Psi_{xi} = (S_{xi}, C_{xi})$ be found at stage x, and let no tree-node X in T contain both x and a vertex $u \in C_{xi}$. Let $S_{T_{xi}}, 1 \leq i \leq k$ be the separators representing the tree-edges between a tree-node that contains x and a tree-node that does not contain x. Then there do not exist a pair of separators $S_{T_{xi}}, S_{T_{xj}}, 1 \leq i, j \leq k$ and $i \neq j$, where $S_{T_{xi}}$ separates v from x and $S_{T_{xj}}$ separates w from x and $v, w \in C_{xi}$.*

Proof. The contradiction follows directly since there exists a path from v to w in C_{xi} and every path from v to w has to pass through a tree-node that does not contain any vertex from C_{xi}. □

Lemma 4.2. *Let the pair $\Psi_{xi} = (S_{xi}, C_{xi})$ be found at stage x. Then the pair Ψ_{xi} or (S_{xi}, C_{xi}^{-1}) is not already inserted into T iff there exists a tree-node $X \in T$ that contains both the vertices u and x, where u is any vertex in C_{xi}.*

Proof. $\Rightarrow$ Let $\Psi_{xi} = (S_{xi}, C_{xi}) \in T$ and let there exists a tree-node $X \in T$ as described. Since S_{xi} is separating x from every vertex in C_{xi}, then this is a contradiction.

$\Leftarrow$ Let $\Psi_{xi} = (S_{xi}, C_{xi}) \notin T$ and let there not exist a tree-node $X \in T$ as described. Let Y be a tree-node in T that contains x, and let the tree-node U contain the vertex $u \in C_{xi}$. There exists a path from Y to U in T. Let the separator S represent the first tree-edge in this path that does not contain x. It follows from Lemma 4.1 that there exists a component C such that $N(C) = S$ and $C_{xi} \subset C$. Let us now assume that a vertex $y \notin C_{xi}$ is in C. This is a contradiction since $y \in S_{xi} \cup C_{xi}^{-1}$ and clearly $y \notin S_{xi}$ since the edge (x, y) will cross S, and $y \notin C_{xi}^{-1}$ since every path from y to u passes through S_{xi} and there exists a path from y to u in C, and therefore an edge in S. It follows from this that $C = C_{xi}$ and clearly $S = S_{xi}$ since $N(C_{xi}) = S_{xi}$. Now we have a contradiction since $\Psi_{xi} = (S_{xi}, C_{xi}) \notin T$. □

Lemma 4.3. *If a pair $\Psi_{xi} = (S_{xi}, C_{xi}) \notin T$ then there exists exactly one tree-node $X \in T$ that contains x and u, where u is any vertex in C_{xi}.*

Proof. Let us on the contrary assume that $u, v \in C_{xi}$, and that there exist two tree-nodes $U, W \in T$ such that $u, x \in U$ and $v, x \in W$. We know that no pair inserted into T contains a separator S such that $x, u \in S$ or $x, v \in S$, since x is separated from u and v. There exists a path from U to W in T since T is connected, and let S' be a separator representing an edge in this path. It follows that S' contains x since both U and W contain x. Observe that S' separates u and v since S' is in the path from U to V and S' can not contain u or v. We have now a contradiction since there exists a path from u to v in C_{xi} that does not contain any vertex from $N_{G_x}(x)$ and S' is a minimal u, v-separator where $S' \in N_{G_x}(x)$ since S' is saturated in G_x and $x \in S'$. □

Now we know that there is at most one tree-node in T that contains both x and a vertex u of C_{xi}. Thus the search to find this tree-node can be explained as follows. We start with any vertex u that belongs to C_{xi}. We find a tree-node U in T that contains u. From U we do a depth first search until we find a tree-node Y that contains x. Now the search can stop. If Y contains a vertex of C_{xi} then we have found our desired tree-node to split. If Y does not contain a vertex y of C_{xi} then no other tree-node in T contains both x and a vertex y of C_{xi}, because T is a tree decomposition (this will be proved later), and the search can stop. The following lemma shows how we can efficiently decide whether or not Y contains a vertex y of C_{xi}.

Lemma 4.4. *Let S_j be the separator representing the edge used to reach Y, and let S_{xi} be the separator in the pair $\Psi_{xi} = (S_{xi}, C_{xi})$, which we want to insert into T. $S_j \neq S_{xi}$ iff Y contains any vertex u belonging to C_{xi}.*

Proof. $\Rightarrow$ Let $S_j \neq S_{xi}$ and let Y contain no vertex from the component C_{xi}. This is a contradiction since $S_j = S_{xi}$ (Lemma 4.2).

$\Leftarrow$ Let $S_j = S_{xi}$ and let the tree-node Y contain at least one vertex u from the component C_{xi}. This is a contradiction since S_{xi} is separating x and u, and $S_j = S_{xi}$. □

From Lemma 4.4 it is easy to see that we have to split Y in order to insert Ψ_{xi} into T. The efficiency of these operations will be discussed in the following chapter.

4.5 Inserting a pair Ψ_{xi} into T

We are still at stage x, and have found the pairs $\Psi_{x1}..\Psi_{xl}$, which are not inserted, and the corresponding tree-nodes $Y_1..Y_l$ that contain vertices which are separated by the separator in the pair. Let us focus on how to update T with a single pair $\Psi = (S, C)$, and let Y be the found tree-node containing vertices separated by S. In order to update T we have to split the tree-node Y. This split is done by replacing Y with X_1 and X_2. The vertices in $X_1 = S \cup (Y \cap C^{-1})$ and $X_2 = S \cup (Y \cap C)$. The separator S is representing the tree-edge between X_1 and X_2, and is used when we create the tree-edge (X_1, X_2). In order to complete the update we have to reconnect every tree-edge connected to Y to either X_1 or X_2.

Invariant 4.2. *At all stages of the algorithm, for any pair of adjacent tree-nodes X and Y, the separator S_{XY} representing the tree-edge (X, Y) is a subset of both tree-nodes X and Y, and $S = X \cap Y$.*

The invariant is definitely true when there are no edges in the tree and the tree consists only of one tree-node. When the very first tree-node is split the invariant is true by the definition of X_1 and X_2. The following discussion explains that the invariant will remain true after each split operation.

In Lemma 4.5 we prove that the vertices in S_{XY} representing the edge (X, Y) are not partitioned when the tree-node Y is split. So it is always possible to reconnect (X, Y) to either X_1 or X_2.

Lemma 4.5. *Let S_{XY} be the separator in the tree-edge $(X, Y) \in E(T)$, and let the separator $S \in \Psi$ split Y into X_1 and X_2. Then one of these three situations will occur:*

1. $S_{XY} \subseteq X_1$, $S_{XY} \nsubseteq X_2$
2. $S_{XY} \subseteq X_2$, $S_{XY} \nsubseteq X_1$
3. $S_{XY} \subseteq X_1$, $S_{XY} \subseteq X_2$

Proof. Assume that none of the three cases occurs, so that $S_{XY} \nsubseteq X_1$ and $S_{XY} \nsubseteq X_2$. We know that $S_{XY} \subset Y = X_1 \cup X_2$ and that $S = X_1 \cap X_2$. From this we may conclude that $(X_1 - S) \cap S_{XY} \neq \emptyset$ and that $(X_2 - S) \cap S_{XY} \neq \emptyset$. Then S_{XY} and S will be crossing, because S is separating vertices in S_{XY}, which gives a contradiction. □

Because of Lemma 4.5 we are able to do a straight forward reconnection of the tree-edges connected to Y. If the separators in the tree-edge is a subset of X_1 we reconnect to X_1, and if not we reconnect to X_2. In the third case where $S_{XY} \subseteq X_1$ and $S_{XY} \subseteq X_2$, then $S_{XY} \subseteq S$, since $S = X_1 \cap X_2$. When this situation occurs the tree-edge may be reconnected to either X_1 or X_2. Since we are building a tree, it does not matter which tree-node we connect the tree-edge to. In this way we maintain the Invariant 4.2 that the separator in the tree-edge is a subset of both the tree-nodes that the tree-edge is connected to.

Definition 4.2. *Let us define U_A as the union of all separators from the pairs inserted into T.*

In section 4.7 of this chapter we want to find $U(x)$, and to be able to do this we need the union of the separators from all the pairs inserted into T. This union is updated every time a new pair is inserted into T. We will implement U_A as a characteristic vector of length n where $U_A[v] = 1$ if v is in this union and 0 otherwise.

4.6 Data structure

The only remaining operation is to compute $U(x)$, which is the first operation in each stage. Since this operation is closely connected to the underlying data structure, we will do a more accurate description of the tree T. The complete tree data structure is built out of four different objects: *tree*, *tree-node*, *tree-edge* and *tree-node element*. It is important to observe that every vector not specified to the size of n, is dynamic and is only storing necessary data.

Tree object

The first object is the *tree* T which is displayed in Figure 4.2. As we can see from Figure 4.2 the tree T contains two vectors of size n. The tree-node pointer vector is a vector of size n, where element u contains a pointer to a tree-node $U \in T$ containing the vertex u. With this vector we get direct access to a tree-node $U \in T_u$ for all $u \in V(G)$. This will be useful to decide a starting point for depth first search from a tree-node containing a vertex u.

We defined U_A as the union of all separators in the pairs Ψ inserted into T. U_A is a characteristic vector, where every element in U_A is initially set to zero. Element u of U_A is set to one if $u \in S$, and some pair $\Psi = (S, C)$ is inserted into T.

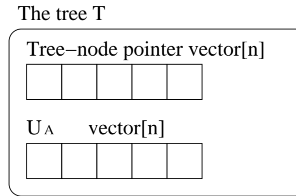

Figure 4.2: Tree data object.

The tree-node and tree-node elements

Every tree-node object consists of an element vector and a tree-edge pointer vector. Every vertex stored in the tree-node is represented with an element object, in the element vector. Figure 4.3 shows the relationship between a tree-node and its elements. Each tree-node element consists of four values. Table 4.1 explains the different elements in the tree-node element object.

Variable name	Description
V_n	The vertex number which the element represents.
N_n	The number of neighboring tree-nodes containing the vertex stored in V_n.
S_c	The last stage this element was requested.
C_o	Counter variable.

Table 4.1: Description of element values.

The tree-node also contains a vector of pointers to tree-edges, where every tree-edge points to a neighbor of the tree-node. In Figure 4.5 we see an example of a tree with two tree-nodes and a connecting edge between them.

Tree-edge object

The only object left to explain is the tree-edge object. Tree-edges are always created in pairs. We think of tree-edges as bidirectional and we represent both directions. Every tree-edge has a pointer to the edge in the opposite direction. The tree-edge object also stores a pointer to the tree-node it is directed to, and a vector of pointers to elements in this tree-node. The pointers in the vector are pointing to elements that represent the separator in this edge. Figure 4.4 shows the relationship between a tree-edge and a tree-node.

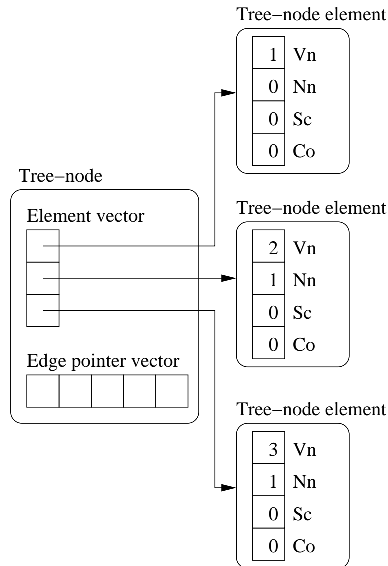

Figure 4.3: Diagram of a tree-node with the relation to three stored elements.

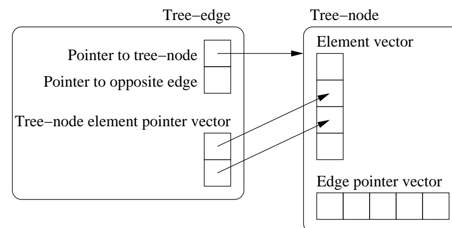

Figure 4.4: The interaction between a tree-edge and the neighbor tree-node.

4.7 Finding $\mathbf{N_{G_x}(x)}$

Finding $U(x)$ is actually the only reason for building the tree T. We use $U(x)$ to find $N_{G_x}(x)$ which is the union of $N_G(x)$ and $U(x)$. A straight forward way of computing $U(x)$ is to scan all the minimal separators found so far that contain x. In the beginning of this chapter we promised to find $U(x)$ without reading every separator containing the vertex x. Actually we are going to find $U(x)$ without reading any separator containing the vertex x.

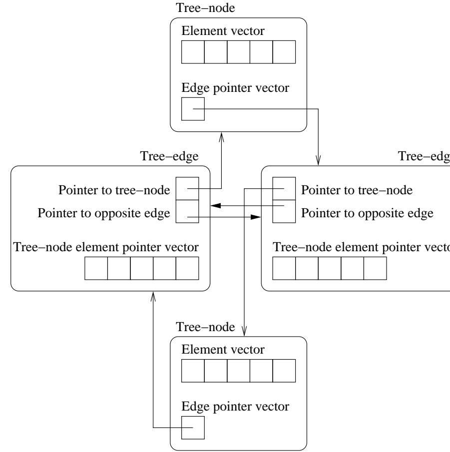

Figure 4.5: An illustration of the two bidirectional tree-edges representing an edge in T.

4.7.1 Properties of the tree T

Invariant 4.3. *At each stage x, the tree T is a tree decomposition of G_x.*

Proof. For the tree T to be a legal tree decomposition of G_x, it has to fulfill three properties at all times.

(i) $\bigcup_{X \in T} = V(G)$.

(ii) For every $(u, v) \in E(G_x)$ there exists a tree-node $Y \in T$, such that $u, v \in Y$.

(iii) For all vertices $v \in V(G)$, the set of tree-nodes containing the vertex v induces a connected subtree T_v of T.

We are going to prove each property by induction. Clearly all three properties hold for the base case where T is only one bag containing all vertices of G. For each of the properties, assume now that the property holds for T, and we will show that it holds also after a tree-node is split into two tree-nodes.

(i) If a tree-node X is splited into X_1 and X_2, then $X_1 \cup X_2 = X$. This guarantees that the set of vertices in the tree T is unaffected by splitting the tree-nodes.

(ii) By the way we described the split operation, two vertices x, y that are neighbors are never splited into two different tree-nodes and the result follows.

(iii) Let us on the contrary assume that the subtree T_v is connected before the split of X and not connected after the operation, and violating the last requirement of a tree decomposition. Vertex v must belong to the tree-edges (X, Z) and (X, Y) of T for two neighbours Y and Z of X. Each of these tree-edges is incident to either X_1 or X_2 after the split of X. We have now a contradiction since (X, Z) and (X, Y) can not be incident to the same tree-node since this makes T_v connected, and the edge (X_1, X_2) also makes T_v connected.

We have now proved that T is a tree decomposition of G. Since every minimal separator in G_α^{LB} is also a minimal separator in G, and the fact that $V(G) = V(G_\alpha^{LB})$, then T is also a tree decomposition of G_α^{LB}. □

Corollary 4.1. *At the end of the algorithm the tree T is a clique tree, and is representing a chordal graph.*

Proof. The tree T is a clique tree iff the vertices in a tree-node is representing a clique and every subtree T_v where $v \in V(G)$ is connected (F. Gavril [9]). We have already proved that every subtree is connected in Invariant 4.3, and each tree-node is already representing a clique. Clearly the tree T is always representing a chordal graph, but T is only representing a minimal triangulation of G when every pair in G is inserted into T. □

Since T is a tree decomposition of G, then for every vertex $u \in V(G)$ there exists a connected subtree $T_u \in T$. T_u is the set of tree-nodes and tree-edges containing the vertex u. We know from Definition 1.5 of tree decompositions that every subtree $T_u \in T$ is connected. From Lemma 4.6 it is easy to see that the set of subtrees in T that have an edge in common with T_x is the same as the set of vertices in $U(x)$.

Lemma 4.6. *Let T_u and T_v be two connected subtrees of T. There exists a separator S_{XY} where the vertices $u, v \in S_{XY}$ iff T_u and T_v have a common edge in T.*

Proof. $\Rightarrow$ Let there exist a separator S_{XY} so that the vertices $u, v \in S_{XY}$, and let there not exist a common edge between T_u and T_v. Let the separator S_{XY} represent the tree-edge $(X, Y) \in T$, where $u, v \in X$ and $u, v \in Y$. The edge $(X, Y) \in T_u, T_v$ since T_u, T_v are connected and $X, Y \in T_u, X, Y \in T_v$. This is a contradiction since there does not exist a common edge between T_u and T_v.

$\Leftarrow$ Let there not exist a separator S_{XY} so that the vertices $u, v \in S_{XY}$, and let there exist a common edge between T_u and T_v. This is a contradiction since there exists a common edge between T_u and T_v, and therefore there exists a separator S_{XY} where $u, v \in S_{XY}$. □

Subtree Type	Description
NoEdge	The tree T_u is only one tree-node.
Inner	Every tree-edge in T_u is in T_x.
InnerOuter	At least one tree-edge in T_u is in T_x, and at least one tree-edge in T_u is not in T_x.
BorderOuter	No tree-edge in T_u is in T_x, and there exists a tree-node $X \in T_u$ where $x \in X$.
Outer	No tree-edge in T_u is in T_x, and there does not exist a tree-node $X \in T_u$ where $u \in X$.

Table 4.2: Different subtree types

4.7.2 Grouping subtrees into different types

To be able to find the set of subtrees from T that have a common edge with T_x, we classify the subtrees into five different types. Table 4.2 contains the definition of these subtrees.

If we are able to find the union of Inner and InnerOuter subtrees then the resulting set will be $U(x)$, because Inner and InnerOuter are the only subtrees that have a common tree-edge with T_x. To be able to do this we first find the union between the InnerOuter and the BorderOuter. Then we separate the InnerOuter from the BorderOuter. Next we remove the NoEdge, InnerOuter and BorderOuter from the complete set. This leaves us with the union of Inner and Outer. Finally we separate Inner and Outer and we are able to find the union between Inner and InnerOuter.

4.7.3 Finding InnerOuter and BorderOuter

Definition 4.3. *A tree-edge $(U, X) \in T$ is a $Border_x$ edge if X contains the vertices x and u, where $u \in U$ and $x \notin U$.*

Since InnerOuter and BorderOuter always contain a $Border_x$ edge (4.3), and are the only subtrees that contain $Border_x$ edges we are using this property to find the union of InnerOuter and BorderOuter.

In fact the $Border_x$ edges are the tree-edges in T with exactly one end-point in T_x. By doing a depth first search from a tree-node X where the vertex $x \in X$ we are able to find T_x, and every $Border_x$ edge. The $Border_x$ edges are the set of tree-edges not containing the vertex x, and are connected to T_x. If we find the union of all the separators stored in the $Border_x$ edges we will get the union of InnerOuter and the BorderOuter subtrees.

4.7.4 Separate the InnerOuter from the BorderOuter

The next step is to separate InnerOuter and BorderOuter. This operation is by far the most complex part of the algorithm. The difference between

InnerOuter and BorderOuter is that InnerOuter has at least one edge in common with T_x, and BorderOuter has none.

Let X be the tree-node in the $Border_x$ edge containing the vertex x. When separating InnerOuter and the BorderOuter it is tempting to check every edge in T_x connected to X, to decide what kind of subtree it is. But by doing this we risk to read all separators containing the vertex x.

To avoid this we use the properties of the tree data structure. The separator in the edge is a vector of pointers to tree-node elements in the tree-node. Initially the S_c value from the tree-node element is set to -1. We use this value to decide if this is the first time this element is visited at this stage. If it is so, we update the S_c value to this stage and copy the N_n value to C_o. Every time an element is visited we decrement the C_o value by one. If this results in that the C_o value is zero, then the subtree representing this vertex is a BorderOuter, and if not it is an InnerOuter subtree.

Simply by reading every separator in the $Border_x$ edges we are able to separate the InnerOuter from the BorderOuter subtree.

4.7.5 Finding Inner

The complement of U_A gives us the NoEdge subtrees. These subtrees have no edges, and therefore they are not in the set of subtrees which have a common edge with T_x. If we also remove InnerOuter and BorderOuter, the remaining subtrees are either Inner or Outer. Since every tree-node in an Inner subtree contains the vertex x, and there does not exist a tree-node in an Outer subtree that contains the vertex x, we just pick a tree-node and check if it contains x.

4.7.6 Summing it up

We have now described an algorithm and a data structure that are capable of computing LB-triang of a given graph G. It is important to observe that we did manage to find $U(x)$ only by reading the separators in the $Border_x$ edges.

Comparing this algorithm to the previous described LB-triang algorithm, there are several important differences. Maybe the most important is the fact that the tree T stores the relationship between the minimal separators. This is the main reason that we are able to find the subtrees of T.

Chapter 5

Time analysis of *LB-treedec*

5.1 Introduction

In the previous chapter we presented the algorithm *LB-treedec* that implements *LB-triang*. This chapter is explaining and proving that *LB-treedec* runs in $O(nm)$ time. Some of the operations in *LB-treedec* can use more than $O(m)$ time in a specific stage. So in order to achieve the total time complexity of $O(nm)$ we have to use amortized time analysis for these operations.

Since we also want to limit space requirements we use a list representation of the vertices in the tree-node. This represents a problem since the algorithm is heavily based on deciding whether or not a vertex x is represented in the vertex list of a given tree-node.

We will now do an amortized time analysis to prove that these questions can be answered in $O(n^2 + k)$ time, where k is the number of requests. Before continuing we need to introduce some limitations. First we limit the number of created tree-nodes in the tree T. In Section 4.5 we described how we increment the number of tree-nodes, i.e. deleting one tree-node and replacing it with two new ones.

Lemma 5.1. *The total number of tree-nodes created during algorithm LB-treedec is less than* $2n$*.*

Proof. Since the tree T is a clique tree, then we know that the final number of tree-nodes in T is less than or equal to n. Increasing the number of tree-nodes in T is done by removing one tree-node and replacing it with two new ones. This operation may only be repeated $n-1$ times since $V(T) \leq n$. If we add the tree-node created in the base case, then the total number of created tree-nodes is less than or equal to $1 + 2(n-1) = 2n - 1$. □

Lemma 5.2. *Let k be the number of requests to decide if the vertex x belongs to X, where X is a tree-node in T, and let the requested vertices be sorted in a nondecreasing order. Then the total time for all requests is $O(n^2 + k)$.*

Proof. To be able to do this we demand several properties from the tree-nodes, and these properties have to be maintained through the whole algorithm.

- The Vector V_x containing vertices in a tree-node X is always sorted in a non decreasing order.
- Every tree-node X has a pointer p_x pointing to the last read vertex in the vector V_x.
- A vertex x is never listed twice in the same tree-node vector.
- The number of vertices in a tree-node is less than or equal to n.
- The number of tree-nodes to be processed is less than $2n$.

Since the vector V_x containing the vertices in X is sorted, and the requests come in a non decreasing order, then the pointer p_x stays in the same place or is moved to a greater index in the vector V_x. As a result of this every Vector V_x is traversed at most once. We have to consider all tree-nodes that have been created at some point, and not only the ones currently present in T. By considering the maximal length of the Vector V_x, and the maximal number of created tree-nodes, then we get a total time of $O(n(2n) + k) = O(n^2 + k)$. It is necessary to add the k since the number of requests may be larger than n^2. □

Clearly the number of requests k may not exceed $O(nm)$, since we want to achieve a total time complexity of $O(nm)$. In Section 5.3 we will explain that the total number of requests will never exceed $O(nm)$. When we are doing the time analyzing of LB-triang, we do not have to consider this operation since the total time is limit to $O(nm)$ for $O(nm)$ requests.

5.2 Finding pairs in G

When we start this operation we are at stage x, and have already found $N_{G_x}(x)$. If we remove $N_{G_x}[x]$ from G_x, then each connected remaining subgraph is defined as a component C_{xi}. Each C_{xi} has a minimal separator S_{xi} separating the component from the rest of the graph G_x. We use the separator S_{xi} and the component C_{xi} to define the pair $\Psi_{xi} = (S_{xi}, C_{xi})$. But since $E(G_\alpha^{LB}) = m'$, and we want to achieve $O(m)$, we have to do this search in the original graph G. The difference between G and G_α^{LB} is that some minimal separators in G are saturated in G_α^{LB}, but we know from Lemma 3.1 that every minimal separator in G_α^{LB} is also a minimal separator in G. Lemma 3.9 states that the exact same components will remain if we remove a minimal separator S from both G and G_α^{LB}. We can conclude from this that

component search can be done in $O(m)$ time since we get the exact same result if we do the search in G or G_{α}^{LB}. Before continuing we need to define some properties regarding the number of pairs and the size of components and separators.

Definition 5.1. *Let $C_{x1}, C_{x2}...C_{xk}$ be the connected subgraphs, after removing $N_{G_x}[x]$ from the graph G. Further let $S_{x1}, S_{x2}...S_{xk}$ be the minimal separators defined by the neighborhoods of these components in G.*

Property 5.1. *The sum $\sum_{i=1}^{k} |C_{xi}| < n$.*

Proof. Clearly $C_{xi} \cap C_{xj} = \emptyset$ for $1 \leq i, j \leq k, i \neq j$, since the subgraphs are not overlapping. Each component contains at least one vertex, and therefore $k < n$. □

Property 5.2. *The sum $\sum_{i=1}^{k} |S_{xi}| < m$.*

Proof. Let S_{xi} be the minimal separator of a component C_{xi}. It follows directly that every vertex $u \in S_{xi}$ has an edge to at least one vertex in C_{xi}. Clearly this limits the size of $\sum_{i=1}^{k} |S_{xi}| < m$. □

Achieving the time complexity

The separators and components found at each stage must have their vertex lists sorted, this is done once for each stage. To reduce the work we sort all the vertices from separators and components in one big sorting. For every vertex u in a component C_{xi} or separator S_{xi} we define a pair (u, S_{xi}) or (u, C_{xi}), where i is the number of the current separator or component. Next we use *Counting sort* to sort all these pairs according to the vertex number. Since the vertex numbers are in the interval $[1, n]$, and the the maximal number of elements is $n + m$, then counting sort runs in $O(n + m)$ time. By parsing through the sorted list and reading the second value in the pair, we are able to decide which separator or component the vertex number belongs to and reinsert the vertex into its sorted place in the separator or component. In this way we are able to reconstruct every component and separator in a sorted order.

5.3 Find pairs not inserted into T

When deciding if a pair $\Psi_{xi} = (S_{xi}, C_{xi})$ is already inserted into T we find a tree-node $U \in T$ that contains any vertex u, where $u \in C_{xi}$. This is a straight forward $O(1)$ operation since we use the tree-node pointer vector (Figure 4.2).

Next we do a depth first search in T from U until we hit a tree-node Y, where the vertex $x \in Y$. Let S_j be the separator representing the tree-edge we use to reach the tree-node Y. In Lemma 4.4 we proved that $S_j \neq S_{xi}$ iff

the pair Ψ_{xi} is not already inserted into T. To decide if $S_j \neq S_{xi}$ is a $|S_{xi}|$ operation since the separators may differ by only the last vertex.

Property 5.2 states that the sum of all separators found at stage x is $< m$. By using this we are able to compare every separator found at stage x with it's matching S_j in $O(m)$ time.

5.3.1 Traversing the tree

We need to describe how much time is used to traverse the tree T. The number of components found at stage x is $< n$. If we combine this with the size of T, it becomes $O(n^2)$ which is more time than we can afford. We solve this by proving that the searches do not overlap, with the only exception of the tree-node Y, which is containing the vertex x. Thus the search concernens the separators found at stage x are done in disjoint subtrees for each separator.

The tree search at stage x starts in a tree-node U containing any vertex u from the component C_{xi} of the pair $\Psi_{xi} = (S_{xi}, C_{xi})$ found at stage x. We search from U until we find Y. Since we are dealing with a tree, there is only one edge connecting the subtree containing U to Y.

Lemma 5.3. *Let the pair $\Psi_Y = (S_Y, C_Y)$ define the edge S_Y used to reach Y, and let C_Y^{-1} contain the vertex x. Further let Ψ_{xi} and Ψ_{xj} be two other pairs found at stage x, where $u \in C_{xi}$ and $v \in C_{xj}$. Then the component C_Y may not contain both the vertices u and v.*

Proof. Let assume that C_Y contains both u and v. Since Ψ_{xi} and Ψ_{xj} are found at stage x, then both S_{xi} and S_{xj} are separating u from v. This is a contradiction since S_Y is separating x from u and v, and S_u is separating U from x and v. The result is crossing separators. □

Corollary 5.1. *Each tree-edge (X, Y) used to reach Y is only traversed once at stage x.*

Proof. Let T_{XU} be the subtree left containing U after removing (X, Y) from T. Lemma 5.3 proves that this subtree only contains vertices from one component. Therefore only one component is capable of starting the search in T_{XU}, and traversing the tree-edge (X, Y). □

If we remove every tree-node Y containing the vertex x from T then a forest F will be left. From Corollary 5.1 it is easy to see that each of the subtrees of F only contains vertices from one component in a pair Ψ found at stage x. Every edge in the tree T is only traversed once, and therefore the total time for all tree searches in stage x is $O(n)$.

The traversal of each edge involves a request of type "does x belong to Y". Thus the total number of such requests during the whole algorithm is $O(n^2)$.

5.4 Inserting a pair into T

Since the maximal number of tree-nodes in T is limited by n, this limits the number of inserted pairs Ψ_{xi} into T to n. With only n insertions, then each insertion is allowed to use $O(m)$ time if we want to achieve a total time of $O(nm)$.

5.4.1 The insertion

When deciding if a pair $\Psi_{xi} = (S_{xi}, C_{xi})$ is already inserted into T, we end the search in the tree T with a tree-node Y, which contains the vertex x. If Y also contains any vertex u where $u \in C_{xi}$, then the pair Ψ_{xi} is not inserted. Let Y contain the vertices u and x, and let S_{xi} separate these vertices. To remove the edges between vertices separated by S_{xi}, from the chordal graph represented by the tree T, we have to split Y.

The vertices u and x are separated by S_{xi}, and therefore we have to split the tree-node Y into X_1 and X_2. The split of Y, and the insertion of the tree-edge between X_1 and X_2 is a straight forward $O(n)$ operation. The reason for this is the simple way to find the set of vertices in X_1 and X_2, and the fact that the separator S_{xi} is representing the tree-edge between X_1 and X_2.

5.4.2 Reconnecting of edges

Since the tree-node Y is deleted, we have to reconnect every tree-edge (X, Y) in T to either X_1 or X_2. This is done to keep the tree connected, and to maintain the invariant that T is a tree decomposition of G. Because of Lemma 4.5 we know that it is always possible to reconnect the tree-edge to either X_1 or X_2.

To decide if (X, Y) should be reconnected to X_1 or X_2, we have to read the separator S_{XY} representing the tree-edge. If $S_{XY} \subset X_1$ we reconnect the edge to X_1, and if not we reconnect it to X_2.

With a straight forward analysis this is an $O(n^2)$ operation, since the number of edges connected to Y is $\leq n$ and the size of each separator is $\leq n$. But $O(n^2)$ is to expensive so we have to improve the analysis.

Lemma 5.4. *If we remove every vertex $u \in Y$ from G, then one subgraph will remain for each tree-edge connected to Y.*

Proof. Every tree-edge (X, Y) connected to Y stores a separator S_{XY}. This separator is a subset of both the tree-nodes the tree-edge is connected to. Therefore every separator is represented by a tree-edge connected to Y is removed from G. Each of the removed separators will leave a connected subgraph, i.e. the component separated by the separator. □

Lemma 5.5. *The sum of the sizes of the separators representing the tree-edges incident to Y is less than m.*

Proof. The separator $S_{xi} = N(C_{xi})$, where C_{xi} is a connected subgraph of G. A result of this is that for every vertex $v \in S_{xi}$, there exists a vertex $u \in C_{xi}$ such that $(u, v) \in E(G)$. If we include Lemma 5.4, then one subgraph will remain for each tree-edge connected to Y, and for each vertex in the separators separating the subgraphs there exists an edge in $E(G)$ connecting it to the component. This limits the sum of the sizes of all separators in the tree-edges incident to Y to $\leq m$. □

In the view of this analysis we are able to split the tree-node, create the two new, connect them and reconnect every tree-edge connected to Y in $O(m)$ time. By combining this with the maximum number of insertions, we are able to do all insertions in $O(nm)$ time.

5.5 Data structure

The level of details in the tree data structure has increased, and we have to prove that this does not affect the time complexity of the algorithm. The difference are mainly that we use pointers instead of direct access, and that the tree-edges are duplicated to make the effect of bidirectional edges. Both these difference only increase time complexity with a constant.

5.5.1 Reading and traversing

Information needed when reading or traversing the tree is done by reading vertices in a tree-node, reading separators and finding neighbors of a tree-node. The only difference regarding the tree-nodes is that the vector containing the vertex number now contains a set of variables, and among them the vertex number. Storing of the separators is actually the biggest changes. As previously the separator is stored in the tree-edge, but the vector only contains pointers to tree-node elements, which are storing the vertex number. The neighbor vector in a tree-node consists of pointers to tree-edges, which points to the neighbor. These three operations have one thing in common, we have indirect access instead of direct.

5.5.2 Insertion into T

Insertion of pairs is the second operation that we need to explain since it is actually creating the tree. The basic operations of the insertion are splitting, connecting the new tree-nodes and reconnecting the edges.

When inserting a pair into the new data structure, we do the split as usual. But to be able to reconnect the tree-edges and connect the new tree-nodes in $O(m)$ time, we create one vector of size n for each of the new

tree-nodes. This new vector is created in such a way that element i contains a pointer to the tree-node element containing vertex i. If the vertex does not exist in the tree-node then the value is set to zero. By using this vector we are able to connect the tree-nodes and reconnect the tree-edges almost as usual. Since we create these vectors in $O(n)$ time, and the extra pointers only add a constant factor, we still manage to execute the insertion in $O(m)$ time.

5.6 Finding $\mathbf{N_{G_x}(x)}$

$N_{G_x}(x)$ is defined as the union of $N_G(x)$ and $U(x)$. In the previous chapter we proved that $U(x)$ is equal to the union of the Inner and InnerOuter subtrees of T. We find InnerOuter by separating out BorderOuter and InnerOuter, and then separate these two by deciding if they have a common tree-edge with T_x. Inner is found by removing NoEdge, InnerOuter and BorderOuter, and then separating Inner from Outer.

5.6.1 $\mathbf{Border_x}$ edges

BorderOuter and InnerOuter are the only subtrees containing a $Border_x$ edge, and we use this property to find them. $Border_x$ edges are the edges having one endpoint in T_x. So finding the $Border_x$ edges is done by traversing the T_x subtree. From the tree object we get a pointer to a tree-node X containing the vertex x. We start the traversing of T in X, and every time we find a tree-node not containing x, then the last traversed tree-edge is a $Border_x$ edge. Since deciding if a tree-node contains x is an amortized $O(1)$ operation at stage x, this is an $O(n)$ operation.

Next we read the separators in these $Border_x$ edges. To decide the sum of these separators we remove every vertex in T_x from the graph G and one subgraph will remain for each $Border_x$ edge. Lemma 5.5 proves that each of these subgraphs have an edge to every vertex in the separator separating it. This guarantees that the sum of the separators in $Border_x$ edges $\leq m$. With this guarantee we separate BorderOuter and InnerOuter as described in the previous chapter, with a time complexity of $O(m)$.

5.6.2 Finding *U(x)*

The remaining job is quite simple, we remove BorderOuter, InnerOuter and NoEdge from the set of all subtrees. NoEdge is the set of non-zero elements in the U_A vector, which is clearly an $O(n)$ operation.

From the definition we know that every tree-node in an Inner subtree contains the vertex x, and no tree-node in an Outer subtree contains this vertex. By using this we are able to decide in amortized $O(1)$ time if a

remaining subtree is an Inner or an Outer. Since the total number of subtrees is n, then this is clearly an $O(n)$ operation.

5.6.3 Summing it up

Theorem 5.1. *The time complexity of LB-treedec is $O(nm)$.*

Proof. As shown in this chapter every step of *LB-treedec* is an $O(m)$ operation, either with direct or amortized time analysis. Therefore the total time for this algorithm is $O(nm)$. □

The original algorithm *LB-triang* is on-line, meaning the ordering of the vertices can be decided while the algorithm is running. *LB-treedec* does not implement *LB-triang* completely since an ordering of the vertices is required as an input parameter. The reason for this is to avoid each tree-node to use $O(n)$ space which will require a total of $O(n^2)$ space. It is quite clear that adding a characteristic vector of size n in each tree-node enables us to decide if a vertex is contained in the tree-node in $O(1)$ time, independently of the order of the requests. It follows that *LB-treedec* can be implemented to run on-line in $O(nm)$ time if we increase the space requirement to $O(n^2)$.

5.7 Space analysis

The $O(nm)$ implementation of *LB-treedec* uses a tree data structure to solve the problem of storing separators and finding the new neighborhood of vertices. An interesting aspect about *LB-treedec* is the space requirement.

Obviously we need to store the graph G, since we do the component searches in the original graph. But the clique tree will also require some space. All vertices are stored in one tree-node in the base case, which requires $O(n)$ space. The only operation in the actual algorithm that requires more space is the operation that insert a new pair Ψ_{xi} into T, which split a tree-node and inserts a tree-edge.

Lemma 5.6. *Insertion of a pair $\Psi_{xi} = (S_{xi}, C_{xi})$ into T, requires $O(|S_{xi}|)$ extra space.*

Proof. Let the pair Ψ_{xi} split the tree-node Y into X_1 and X_2, and insert the tree-edge (X_1, X_2). The split operation requires $|S_{xi}|$ space, since $S_{xi} = X_1 \cap X_2$. Insertion of the new tree-edge also requires $|S_{xi}|$ space, since the tree-edge only stores a pointer to the opposite tree-edge, the neighbor tree-node and $|S_{xi}|$ pointers to vertices in the neighbor tree-node. □

From Lemma 5.6 it is easy to conclude that the total space requirement for *LB-treedec* is $O(m+m') = O(m')$. The reason for this is that Lemma 3.5 proves that the sum of all separators in the inserted pairs is less than m'.

Chapter 6

Experimental results

In this chapter we present experimental runtime results from our implementations of *LB-treedec*, *LB-list* and *Lex-M*. We are analyzing the practical behavior of *LB-treedec* in three different ways. First we report from experiments on simply the run time of *LB-treedec*, and show that it exhibits very good runtime in practice. Although the practical runtime is very good, we also show an example that meets the $O(nm)$ time bound, showing that *LB-treedec* is $\Theta(nm)$.

We then compare this algorithm to the algorithms *LB-list* and *Lex-M*. It is interesting to compare the two different implementations of *LB-triang*, since this gives us exact information about how the difference in data structure affects the time used by the application. Comparing *LB-treedec* against *Lex-M* is interesting since both algorithms create a minimal triangulation in $O(nm)$ time, by using different approaches to find the minimal triangulation. This gives us the opportunity to compare both the size of the created fill, and the time used by the different implementations.

The implementations of the algorithms are done in **C++**, and we have used the *Standard Template Library* SDL, to create the data structures. **C++** was selected because we want a fast implementation that easily can be moved between different platforms. The tests where executed on a machine with an Intel Pentium III 1GHz processor and 512 MB RAM. Our code can be obtained via anonymous FTP from ftp.ii.uib.no at directory pub/pinar/LB-treedec/.

6.1 Analyzing *LB-treedec*

It is especially interesting to do some practical analysis on *LB-treedec* because it uses a rather big data structure, and the fact that it is a new $O(nm)$ time algorithm which solves minimal triangulation. Let us first have a look at how the run time changes as the size of the graph changes. In this first experiment we use randomly generated graphs, where the number of ver-

tices is fixed at 1000, and the number of edges ranges from 1000 to 499500 or $n(n-1)/2$. We make sure that all these graphs are connected, thus only connected random graphs are tested and presented. By keeping the number of vertices constant, and changing the number of edges, we get the nm function as a straight line in the figure. This is an advantage when we are analyzing the result of the experiment.

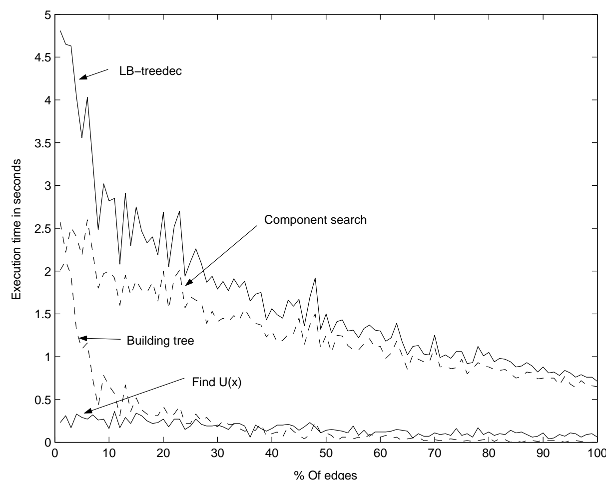

Figure 6.1: Run time of *LB-treedec*, and three different time components.

We have four curves in Figure 6.1, which display the time usage of three main operations in *LB-treedec* and the fourth is the total time. The first and quite surprising observation is that the run time actually decreases as the number of edges increases and that the decrease is quite significant. We can also observe the roughness of the curve indicating that the actual structure of the graph affects the run time. The structure of the graphs will change form step to step since we add the edges randomly.

The operation **Find U(x)** in Figure 6.1 is the operation that finds the new neighborhood of a vertex from the clique tree. The time of this operation varies some with the construction of the triangulated graph, but apart from this it seems to be quite unaffected by the increase in the number of edges. The second operation is **Building tree**, which represents the time used to decide if a pair Ψ or its complement pair is already inserted, and the time used to build and maintain the tree. This operation has an interesting development since it starts out with using approximately half of the run time, and then rapidly drops to a quarter of the time. It is likely that this operation is closely connected to the number of maximum cliques in the resulting triangulated graph. The third operation is **Component search** which is the time used to find the minimal separators in the neighborhood

of a vertex. As we can see from Figure 6.1 this is the dominating time component. If we take a closer look at Figure 6.1, we can actually see that each of the three components decreases as the number of edges increases. It is quite surprising that *LB-treedec* behaves like an $O(n^2)$ algorithm for randomly generated graphs. A possible explanation for this is that the edges in $G[N[x]]$ do not have to be read during the component search. This will clearly become significant when the size of $N[x]$ gets large, and the average neighborhood will clearly increase when the number of edges increases.

6.1.1 Is *LB-treedec* an O(nm) algorithm ?

Is it possible that *LB-treedec* actually is an $O(n^2)$ algorithm that we only have been able to prove an $O(nm)$ time limit for? Probably unlikely since we do a search in the original graph each time we find the connected components and theirs separators. Now we will actually show an example of a graph that causes *LB-treedec* to use $\Theta(nm)$ time. The fact that the number of edges incident to a set of vertices is limited to m, causes the operations of deciding what pair to insert, updating the tree and finding $U(x)$ to have a time complexity of $O(nm)$. So in order to create a worst possible graph we have to use these properties. We solve this by creating a graph class β.

Definition 6.1. *The graph $G = (V, E) \in \beta$ if the following properties are for filled.*

- $|V(G)| = 3k$ *for some* $k \geq 1$
- $A \cup B \cup C = V(G)$
- $|A| = |B| = |C| = k$
- $\forall\ v \in A : N_G(v) = V(G) - v$
- $\forall\ v \in B : N_G(v) = A \cup (B - v)$
- $\forall\ v \in C : N_G(v) = A$

Let us first find the size of m. The number of neighbors for each of the sets are: $v \in A \Rightarrow |N(v)| = n - 1$, $v \in B \Rightarrow |N(v)| = (2n/3) - 1$ and $v \in C \Rightarrow |N(v)| = n/3$. It follows that $m = n/3((n-1) + ((2n/3) - 1) + n/3)/2 = 4n^2/9 - 2n/3 = O(n^2)$. The sets A and B induce a clique of size $2n/3$, and clearly A and B are cliques of size $n/3$. The first selected vertex not present in A will find the separator A, and create the complete clique tree, which contains $(n/3) + 1$ tree-nodes and $(n/3)$ tree-edges. Every vertex $v \in C$ is only contained in one tree-node that contains the vertices $A \cup \{v\}$, and the extra tree-node contains the vertices $A \cup B$. Every tree-edge contains the same separator A.

Let us first take a closer look at the component search done in *LB-treedec*. For every vertex $v \in C$ we have to do a component search that covers the vertices $B \cup (C - v)$. Since every vertex in B has $(2n/3) - 1$ neighbors and every vertex in C has $n/3$ neighbors, then the number of edge read for the vertex v component search is $n/3(2n/3 - 1) + (n/3 - 1)n/3 = n/3(n - 2)$ which is clearly $\Omega(m)$. For every vertex $v \in B$ we have to do a component search from every vertex in C. This results in reading $n^2/9$ edges, which is of size $\Omega(m)$. It follows from this that the component searches for vertices in B and C uses $\Omega(m)$ time.

Since all vertices in B and C find the separator A and therefore find $n/3$ different components with a minimal separator of size $n/3$, then the size of the found pairs will be $\Omega(m)$. This causes sorting of the separators and detecting if the pair is inserted to cost $\Omega(m)$.

For all the vertices $v \in B$, the computation of $U(v)$ will force us to read all the $n/3$ edges in the clique tree. Since each of these edges contains a separator of size $n/3$ then this becomes an $\Omega(m)$ operation. It is quite clear that *LB-treedec* will use $\Theta(nm)$ time on graphs from the class β.

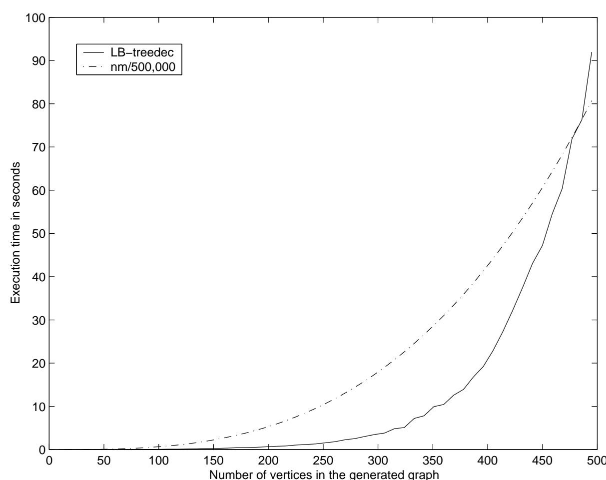

Figure 6.2: Run time of *LB-treedec*, on graphs from the class β.

Figure 6.2 is the result from an experiment, where we use this graph class as input to *LB-treedec*. We can clearly see that this behavior is quite different to the result in Figure 6.1, where we used 1000 vertices. The result in Figure 6.2 is expected to be worse than the previous experiments since we force *LB-treedec* to use $\Theta(nm)$ time. We have also introduced the $nm/500,000$ curve in Figure 6.2. The even worse than expected behavior of *LB-treedec* compared to the nm curve probably arises from the memory architecture. The speed of memory is very closely connected to the price, so in order to

get good value for the funding computers are built with a large slow main memory. In order to speed up the computer, several layers of smaller and faster memory are placed between the CPU and the main memory, such that the size decreases and the speed increases when we are getting closer to the CPU. It follows from this that increasing a given problem forces some of the operations to use a new level of memory. This can have major influences on operations that are executed often in an algorithm. We can conclude from this that the time used by a computer to solve a problem also depends on the architecture of the computer, and not only the algorithm and the given input.

6.1.2 The maximum point in ***LB-treedec*** time curve

An interesting question from Figure 6.1 is how the start of the time curve looks like. This is especially interesting since the triangulation of sparse graphs is in this area. To get a picture of this we do another experiment, where we only focus on graphs that has 0.25% to 10% of the maximum number of edges present.

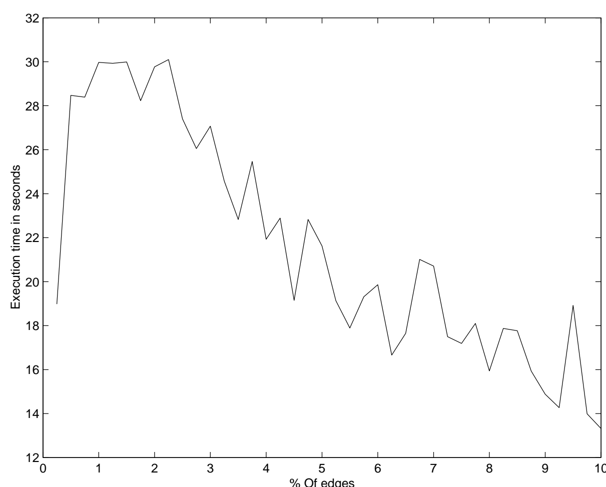

Figure 6.3: Run time of *LB-treedec*, on sparse graphs.

In Figure 6.3 we use randomly generated graphs with 2000 vertices. We can clearly see that *LB-treedec* has a maximum point at approximately $1,5\%$ of the maximum number of edges. A very interesting question now is whether or not the position of this top point depends on the number of vertices. In order to get an answer to this question we did an experiment. For a given number n of vertices, our idea was to generate graphs with varying number of edges and identify the one that resulted in the highest runtime. Thus for

each n we want to find the m that give the worst runtime. We let the number of vertices range from 1000 to 2000 in steps of 10, and then compute the maximum point for each of these steps. Thus $n = 1000 + 10k$, $1 \leq k \leq 100$, and for each k we create 100 random graphs where $m = n(n-1)/2 * l/2000$, $1 \leq l \leq 100$, i.e 0.05% of the maximum number of edges to 5% in steps of 0.05%. In order to decide how the maximal point develop as the number of edges change we register the size of m that causes *LB-treedec* to use most time for a given n. In Figure 6.4 we plot these maximum edge values against the number of vertices.

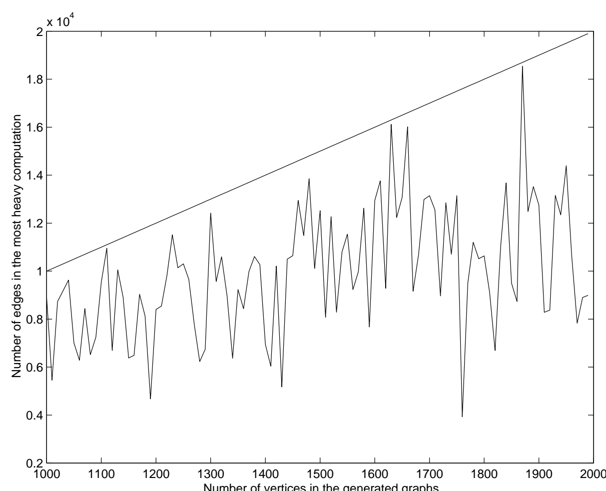

Figure 6.4: Displays the number of edges that creates most work for *LB-treedec*.

We insert the line $10n$ in Figure 6.4, to get a picture of how this maximum value developed compared to the number of vertices in the input graph. The structural changes in the input graph cause the variations in the position of this maximum point. But the indications are quite clear, the maximum point for m keeps below $10n$. This shows that every one of these triangulations was done in an $O(n^2)$ time complexity. We may add that every other random experiment (not especially designed to prove $\Theta(nm)$) has shown the same behavior during our tests.

6.1.3 The roughness of *LB-treedec*

We have observed that every figure containing the running time of *LB-treedec* has an uneven curve. Figure 6.1 shows that this is the case for all the main operations. This is clearly connected to the structure of the graph since the difference between two almost equal graphs in size can vary this much.

The time used by *LB-treedec* is divided into three components. Observe that the roughness descends from all of the time components, even though the graph search is dominating. Another interesting aspect is that there seems to be some kind of connection between the roughness of the graph search and the tree-node separation. A possible theory is that low fill generates a bigger tree data structure, and smaller separators force the component search to read more edges.

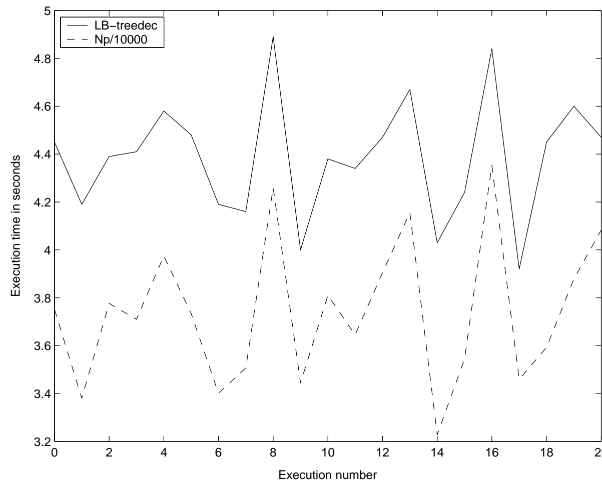

Figure 6.5: Displays execution time of *LB-treedec*, and number N_p of found separator component pairs.

Another possible explanation is the number of found separator and component pairs is large. We will clearly use more time to find the new pairs, which causes the tree operations to use more time. Since we have found many pairs, then it is likely that we get a clique tree with many tree-nodes. We will at least get as many tree-nodes as the largest number of separators in the neighborhood of one vertex. Every leaf tree-node L contains a vertex l that only exists in L. The component search from such a vertex, will only find one component that contains every vertex not in L.

Let us define N_p as the total number of pairs found during the execution of the algorithm. Figure 6.5 displays the run time of *LB-treedec*, and $N_p/10000$. We have divided N_p by 10000 to make it fit into the figure. In this experiment we keep the number of vertices and edges constant i.e. $n = 1000$ and $m = 10000$. The graphs are randomly generated, and therefore the execution time, and N_p will alter around a middle value. The interesting observation is that every major change in N_p, makes an almost equivalent reaction in the execution time. This indicates that there might be a connec-

tion between the roughness of *LB-treedec* and N_p.

In this experiment N_p has an average value of 37256, and an interesting question is: what are we able to tell about a graph with 1000 vertices and a N_p of 37256? We can conclude from this that the average subtree T_x has 37.3 $Border_x$ edges, and it follows from this that the clique tree consists of at least 38 tree-nodes and at least 37 leafs.

6.2 *LB-treedec* versus *LB-list*

We are now going to show how *LB-treedec* compares to *LB-list* in practice. Comparing these two different implementations of *LB-triang* is interesting since they only differ in how to store the minimal separators, and how to find $U(x)$. The *LB-list* algorithm stores the minimal separators in a list, and reads every separator containing the vertex x, in order to compute the new neighborhood of x. Finding $U(x)$ is the operation that makes this into an $O(nm')$ algorithm. In practice, this is a quite simple operation that adds a relatively low constant to the runtime of the algorithm. The fact that many triangulations have little overlap between separators enables this algorithm to triangulate in $O(nm)$ time in many cases.

The *LB-treedec* algorithm on the other hand dynamically builds a clique tree as the separators are discovered, and uses this to compute $U(x)$ in $O(m)$ time. The extra work of dynamically maintain and build this clique tree gives this algorithm a larger constant than *LB-list*.

It is actually very simple to create graphs, and graphs classes that force *LB-list* to use $\Theta(nm')$ time. The overlap between separators decides the size of the separator list, and the size of the separator list decides the run time of *LB-list*.

We are now going to define a graph class that ensures big overlap between the minimal separators, and thereby forcing *LB-list* to actually use $\Theta(nm')$ time. In Figure 6.6 we describe the graph class γ for this property.

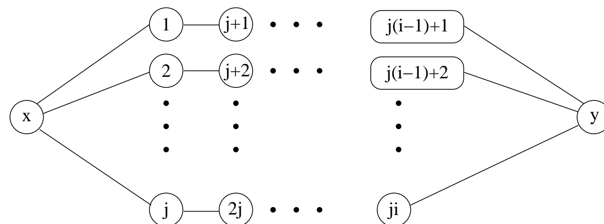

Figure 6.6: A class of graphs that will create big separator overlap.

Any graph from this class will have $ji+2$ vertices and $(i+1)j$ edges, for

given integers i and j. We use the vertex number as the ordering α of which the vertices are made LB-simplicial. This will guarantee $j(i-1)+1$ minimal separators of size j, which gives us a total size of all minimal separators of $j^2(i-1)+j$. The fact that we number the vertices in an order such that every neighbor with a large number is a minimal separator or the rest of the graph, ensures that the vertices are numbered in a MEO order. If we now count every edge to a vertex with a higher number, then it is easy to compute m'. The first $j(i-1)+2$ vertices has j neighbors with a higher number, and the remaining j vertices have $j-1$ to 0 larger neighbors. This sums up to $m' = j^2 i - j^2/2 + 3j/2$. For all graphs in this class where $j \geq 3$ and $i \geq 2$, the sum of the sizes of the minimal separators is larger than m. We can now use the class γ to give a practical verification that the separator list version uses more time. The fact that γ forces *LB-treedec* to update the tree at each operation makes the experiment more interesting, since non of the implementations has any specific advantage, apart from the proven time bound.

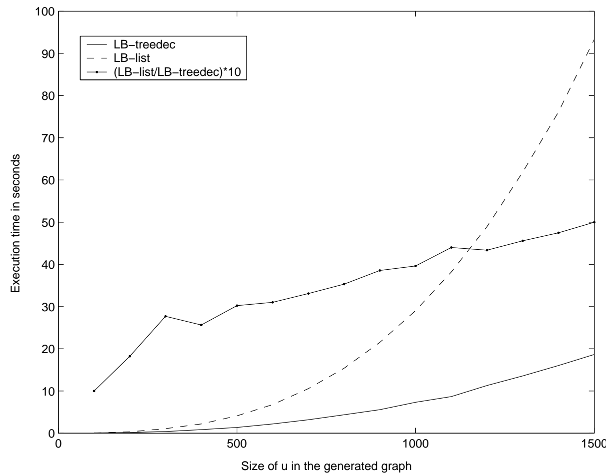

Figure 6.7: Display run time of *LB-treedec* and *LB-list*, when we use graphs from the class γ as input.

In Figure 6.7 we have used graphs from our predefined class and measured the time used to triangulate, where j ranges from 100 to 1500 and i is set to 2. We can clearly see that the difference between the implementations rapidly increases as the size of the graph increases. The curve with the dots gives us the time used by (*LB-list*/*LB-treedec*)*10. We can see that the algorithms start with a one to one relation, and end up with a 5 to 1. The interesting part here is the fact that the difference increases with the size of the graph. This indicates that these two algorithms do not have the same

time complexity, because if *LB-treedec* has a constant k then there exists an n such that *LB-list* uses more time then *LB-treedec*. The reason for this comes from the fact that the sum of the sizes of the separators is $O(m')$.

Because of these conditions we want to run an experiment using randomly generated graphs. By using randomly generated graphs we will not have the problem of the previous experiment where we selected the vertices in a MEO order such that the algorithms only find one minimal separator in the higher numbered vertices. If a separator is a minimal separator for more than two components, then the number of minimal separators are reduced, but this separator will be discovered more than one time. In this experiment we will keep the number of vertices fixed at 1000, and vary the number of edges from $1000 \rightarrow 100000$. This will give us a picture of how the implementations deal with different fill in the input graph. We measure the number of edges in a graph by comparing the number of edges to the maximum possible number of edges, i.e. $n(n-1)/2$. By doing it this way we manage to range the number of edges from $1\% \rightarrow 20\%$. This is the most interesting interval, because after we pass 20% fill we will mainly end up with a few large cliques. The results are presented in Figure 6.8.

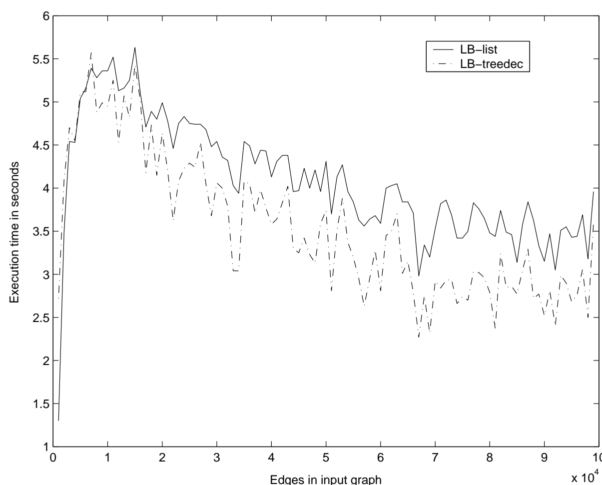

Figure 6.8: Display running time of *LB-treedec* and *LB-list*, with randomly generated graphs i.e. $n = 1000$ and $1000 \leq m \leq 100000$.

There are several things to notice about the results in Figure 6.8. Let us first focus on the fact that these algorithms seem to use less time when the number of edges increases. We have already discussed this for *LB-treedec*, but the fact that *LB-list* has the same behavior indicates that this does not depend on the data structure storing the separator, or the way we compute

the neighborhood of a vertex. This leaves us with the general properties of algorithm *LB-triang* which both *LB-treedec* and *LB-list* implement. We may also observe that *LB-treedec* generally uses less time than the *LB-list* even though it has a larger constant. A final observation may be that a big change in one of the algorithms often results in a change in both curves. This indicates that big variation in run time is connected to the component searches since this is exactly the same for both implementations.

6.3 *Lex-M* versus *LB-treedec*

Both algorithms *Lex-M* and *LB-treedec* have a provable $O(nm)$ runtime, and this makes it interesting to compare them. An interesting question will be whether or not the previously discussed limitations in *Lex-M* will be reflected in the results. We will do two types of experiments, first to compare run time, and second to compare the size of the fill.

6.3.1 Compare computation time

We are now going to compare run time between *Lex-M* and *LB-treedec*. The optimal configuration for such an experiment is when both algorithms create the same result on the same input data, without any restrictions. In this case we will have a problem when it comes to making both algorithms produce the same output, since *Lex-M* is not able to compute some triangulations, while *LB-treedec* is capable of creating every minimal triangulation. This fact forces us to use *LB-treedec* to create the same graph as *Lex-M*. We know that *Lex-M* creates a MEO of the input graph. If we now force *LB-treedec* to make the vertices *LB-simplicial* in the MEO order described by *Lex-M*, then *LB-treedec* will create the exact same graph as *Lex-M*. This follows directly from Lemma 3.2. This is not an optimal solution since we have to set restrictions on the order given to *LB-treedec*, but it is probably as close we can get.

In the next experiment we use randomly generated graphs, with 1000 vertices. We define the maximal number of edges as $n(n-1)/2$. In this experiment we use 100 graphs that contain from 1% to 100% of the maximum possible number of edges.

Let us first look at *Lex-M* in Figure 6.9. Two interesting observations can be done. The first observation is that the curve of *Lex-M* is very smooth. We have some minor changes around 45% fill, but apart from this the curve seems to be almost perfectly smooth. This indicates that *Lex-M* is practically unaffected by the structure of the given graph. The second observation is that the time used by *Lex-M* is very close to a straight line. If we now use the correct constants in $(c1 + c2 * nm) = O(nm)$ then we can get a perfect match between the line of nm and *Lex-M* . This gives us the advantage of being able to compute the running time of *Lex-M* quite accurate.

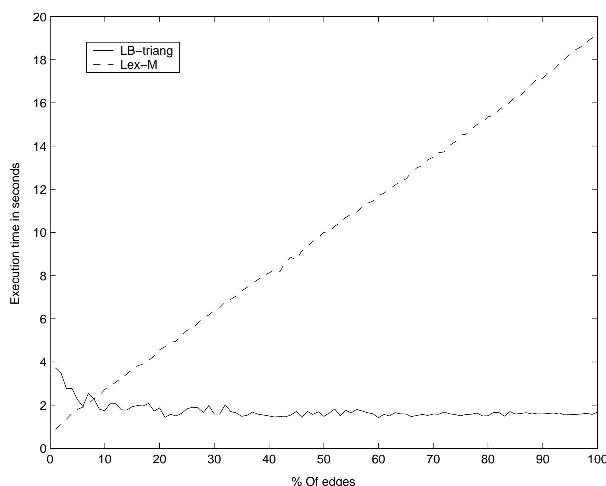

Figure 6.9: Displaying running time of *Lex-M* and *LB-treedec* when the number of edges range from $n(n-1)/200$ to $n(n-1)/2$.

Let us now compare *Lex-M* and *LB-treedec* . As we can see from Figure 6.9 it is preferable to use *Lex-M* as long as less than 7% of the edges are represented in a graph with 1000 vertices. Since the maximal computation in *LB-treedec* seems to be depending on n, then it is likely that the intersection between *Lex-M* and *LB-treedec* also depends on the number of vertices. In the interval from 7% to 100% it is clearly preferable to use *LB-treedec* . The area below each of the curves will give us an indication of which algorithm that is preferable in the general case. We estimate the area below *LB-treedec* to $2 * 100 = 200$, and the area below *Lex-M* to $10 * 100 = 1000$. If we are now given a connected graph with 1000 vertices and a random number of edges between $n(n-1)/200$ and $n(n-1)/2$, then *Lex-M* will approximately use five times as long time as *LB-treedec* in average.

An interesting point for further analysis is the intersection between *Lex-M* and *LB-treedec*. Let us now keep the relationship between m and n constant, and let the number of edges range from 100 to 3000 in steps of 100. Let $f = (n(n-1)/2)/m$, and let f be set to 0.07 in this experiment meaning that we use graphs with 7% of the edges represented.

The result in Figure 6.10 is actually quite close to what we should expect from previous experiments. In Figure 6.4 we followed the maximal value of *LB-treedec* as the number of vertices grows, and observed that it was kept below the $10n$ line. A natural consequence of this is that *LB-treedec* improves compared to *Lex-M* when we use a fixed value for the percentage of edges in the graph, and increase the number of vertices. This is the exact behavior

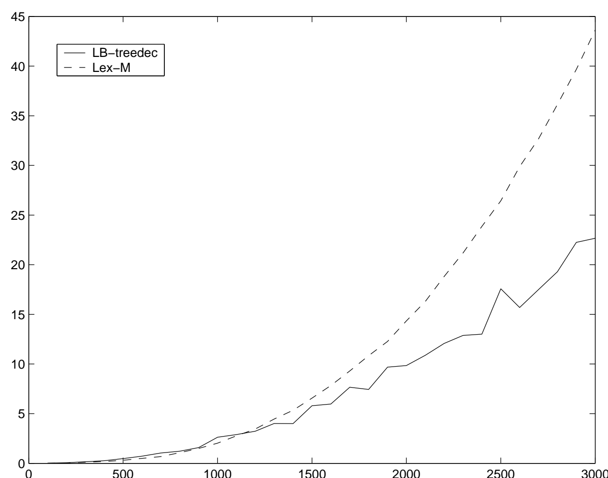

Figure 6.10: Displays time usage at 7% of maximum possible edges.

we see in Figure 6.10.

6.3.2 Compare fill

In the final experiment we are comparing how much fill the two algorithms create in general. The fill is defined to be the edges added to achieve the resulting minimal triangulations. Minimal fill can be far from minimum, and thus good minimal triangulations, with respect to fill, are desirable. The computation of minimum fill is NP-hard, and therefore we do not have any reference value for the minimum triangulation of the different graphs. But we have two different algorithms, and can compare them against each other. We have optimal conditions to do this test since we can use the same input data for both algorithms, and the interesting result is the difference in fill. But since the size of the fill depends on the ordering of the vertices then we give *LB-treedec* a random ordering α of the vertices and we pick a random vertex from *Lex-M* to start in. The input graphs are randomly generated, with the number of vertices fixed at 1000, and the number of edges range from 1000 to 100,000 in steps of 1000.

In Figure 6.11 we have subtracted the fill created by *LB-treedec* from the fill created by *Lex-M*. Observe that *Lex-M* only produces less fill in two cases. Since we do not know the minimum fill we are not able tell whether or not the algorithms produce good triangulations, but we can clearly see that *Lex-M* produces much more fill in general. This indicates that the strategy of *LB-treedec* is better than *Lex-M* in general. It is quite natural that the difference in fill decreases as the number of edges in the input graph

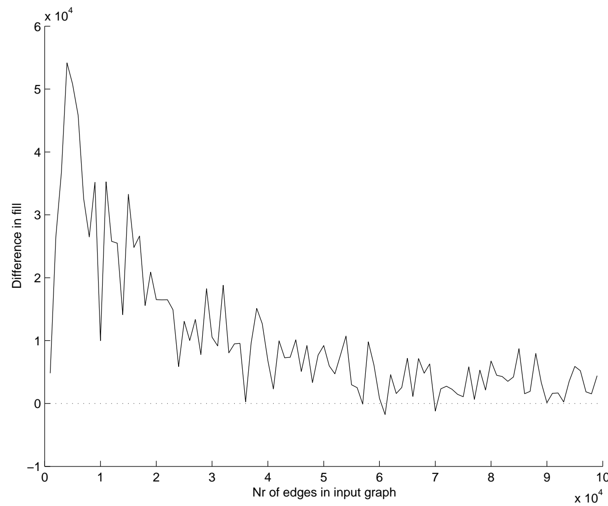

Figure 6.11: Difference in created fill between *Lex-M* and *LB-treedec* .

increases. The reason for this is that the number of edges in the minimum triangulation is getting closer to the maximum possible number of edges, as the number of edges in the input graph increases. This makes the difference between the maximum number of edges and minimum triangulation smaller, and since every triangulation is contained in this gap then it follows that the difference between triangulations decreases.

Table 6.1 displays the resulting number of edges where we have randomly created 20 different graphs. The number of edges in the input graph ranges from 1 to 20 percent of the maximum number of edges, and the number of vertices are set to 1000. The five first columns give us the number of edges in the resulting graph using *Lex-M* where we give *Lex-M* different starting points. *LB-treedec* is used to triangulate the same graph as *Lex-M*, where we randomly created five different orderings in which the vertices was made LB-simplicial. The number of edges in the resulting graph is presented in the five last columns. We can observe that *LB-treedec* produces less fill than *Lex-M* in general eventhough *Lex-M* produces less fill in some of the cases. The difference between *LB-treedec* and *Lex-M* is especially large on those graphs with few edges. This indicates that *LB-treedec* is preferable on sparse graphs since the difference in created fill is significant.

	Lex-M					**LB-treedec**				
1 %	274397	273917	278616	281858	282521	241843	237137	245288	242770	243225
2 %	371722	374749	383642	377670	372814	349392	335589	340122	334617	338274
3 %	409777	407773	418479	421156	418136	383997	395973	402198	385406	385904
4 %	432518	434996	435605	433822	438232	414362	418422	413788	420357	419351
5 %	448021	446361	447598	449648	448147	438790	440700	437585	433299	428113
6 %	458513	461172	457904	460316	453492	443779	440557	446028	447645	445575
7 %	456806	461098	465087	461322	463296	456741	450553	461591	455298	450754
8 %	469811	465522	467547	469611	468191	465296	457334	463035	459295	458801
9 %	473206	469407	472269	470549	469337	461957	465112	463173	465288	464324
10 %	473922	476090	477207	475082	473721	462518	468470	470772	470490	473115
11 %	473836	475241	478226	478119	477205	473781	475679	466891	471231	471278
12 %	476795	478323	478943	478162	480894	474309	472237	473944	473485	473385
13 %	482298	479746	479021	483700	480089	478232	476422	471327	475969	472272
14 %	483828	484341	479245	484519	484025	474393	476095	477679	478020	478714
15 %	483590	484876	480854	484817	484031	475437	479936	480047	482473	478012
16 %	486520	483936	484679	485010	486185	477238	484187	481861	482272	480359
17 %	485645	486807	486495	483263	484501	481405	481487	482086	484678	483233
18 %	484465	487625	486876	486711	487478	485095	483306	486973	481656	482968
19 %	488344	487392	488363	489226	487354	484982	481984	485349	485715	485811
20 %	487233	488283	487747	489364	488951	488060	487041	485991	485243	485064

Table 6.1: Number of edges in the resulting minimal triangulations by *Lex-M* and *LB-treedec* on five permutations of 20 graphs.

Chapter 7

Concluding remarks

In this chapter we are summing up the work done in this thesis. Chapters 1 to 3 contain mainly results established prior to our work, to give a background for the results of this thesis. Chapters 4 to 6 present new theoretical and practical results. We will summarize this work in the next section. The open questions section discusses different unanswered questions, and possible improvements on the *LB-treedec* algorithm.

7.1 An overview of our results

In Chapter 2 we defined chordal graphs, and described three different characterizations. These characterizations are the main ideas behind most of the presented algorithms regarding chordal graphs. We did also mention the clique tree, which is an alternative way to represent a chordal graph. *Elimination game* is the first presented algorithm that produces a chordal supergraph from any given graph. The great advantage of this algorithm is that it can be implemented to run in $O(m')$ time, which is optimal. The disadvantage is that it can produce arbitrarily much fill, even when the given graph is chordal. Next we described the Maximum Cardinality Search(MCS) [19] which is an efficient algorithm for recognizing chordal graphs, and can also be used to compute a clique tree from a chordal graph.

Rest of the thesis is mainly discussing different aspects of minimal triangulation. First we described the algorithm *Lex-M* which was introduced by Rose, Tarjan and Lueker [17] in 1976. This was one of the first presented algorithms that computed a minimal triangulation in $O(nm)$ time. An interesting contribution of this thesis regarding *Lex-M* is a class of graphs for which *Lex-M* is unable to compute a minimum triangulation regardless of starting point and local selections, presented in chapter 2.

Chapter 3 contains description of the basic *LB-triang* algorithm, and several different algorithms that are based on *LB-triang*. The basic *LB-triang* algorithm does not describe how to store the minimal separators from

previous calculations. There does not seem to be an obvious solution to this problem, which results in the discussion of these different implementations of *LB-triang*. We mentioned the *LB-list* algorithm in special, since this was the best *LB-triang* implementation before the work of this theses started.

Chapter 4 presented the *LB-treedec* algorithm which is the first algorithm to implement *LB-triang* in $O(nm)$ time, and the main result of this thesis. The time complexity of *LB-triang* was conjectured by Berry to be $O(nm)$, however this remained unproven until our *LB-treedec* result. Algorithm *LB-treedec* differs from previous discussed *LB-triang* implementations in the way the minimal separators are stored. A tree decomposition representation is used, and updated each time a new separator with components is discovered. The great advantage of this data structure is that it also stores the relationship between the separators.

The operation of finding the neighborhood of the next vertex causes algorithms like *LB-list* to use more than $O(nm)$ time. In order make *LB-treedec* keep the time limit of $O(nm)$, we had to prove that the total time of maintaining the clique tree data-structure is $O(nm)$, and that finding the neighborhood of the next vertex can be done in $O(m)$ time. The interesting part is how we computed the neighborhood, since this is the problem in previously discussed algorithms.

Let x be the next vertex, which forces us to compute $N_{G_x}(x) = N_G(x) \cup U(x)$. Since the sum of all separators containing x is of size m', then we had to compute $U(x)$ in a different way. We used the fact that our tree data-structure is a tree decomposition, and that every tree-node containing a vertex v forms a connected subtree T_v. We defined five different classes of subtrees, depending on their relations to T_x. If T_x and T_v have a common edge in T, then there exists a separator that contains both x and v. We now computed $U(x)$ by finding the union between the classes of subtree that has an edge in common with T_x.

In Chapter 6 we did some practical experiments on *LB-treedec*, and between *LB-treedec*, *LB-list* and *Lex-M*. The first experiments were on *LB-treedec*, where we have examples of classes of graphs that forces *LB-treedec* to use $\Theta(nm)$ time, while the algorithm behaved like $\Theta(n^2)$ on randomly generated graphs. Secondly we compared *LB-list* with *LB-treedec*. They seem to behave similarly on randomly generated graphs, even though *LB-list* used a little more time in general. We were also able to find a graph class which forces *LB-list* to use $\Theta(nm')$ time. Finally we compared *Lex-M* and *LB-treedec*. Since these algorithms use different basic approaches to triangulate the graph, they do not necessarily produce the same result. This enables us to compare both the size of the fill, and the time used if they compute the same graph. It turned out that *Lex-M* behaves in a $\Theta(nm)$ fashion also in practice, and in general creates more fill than *LB-treedec*.

7.2 Open questions

The *LB-treedec* algorithm presented in this thesis requires the ordering of the vertices as an input parameter, while the basic LB-triang algorithm is on-line. This is the only limitation in *LB-treedec* compared to the basic *LB-triang*. It is an interesting question whether or not the ordering of the vertices can be given in a on-line fashion without breaking the time and space limits. This can probably be done if we only focus on either time or space. But it seems to be a little more difficult to achieve this without breaking either the space or the time limit.

Regarding how to optimize *LB-treedec*, it is an interesting question whether or not we can take advantage of the incomplete clique tree when we do the component searches in the graph. The reason for this is that if we cross a separator in the component search, then the rest of the information is available in the tree T. The really interesting question is whether or not this gives us a better upper limit of the searches done in the graph. The second interesting question regarding *LB-treedec* is how much more space efficient it is possible to get the algorithm. It is possible to do some improvements if we process all border edges connected to one tree-node at once. This will enable us to remove the counter variable and the variable containing the stage number. Even after including this improvement, it will be interesting to know whether or not the space constant can be reduced even more.

There exist several algorithms that produce a minimal triangulation in $O(nm)$ time, but no known algorithm has a better time complexity than this. A natural question for further work will be whether or not this is the lower limit for minimal triangulation. The fact that there exist at least three different $O(nm)$ algorithms, imply that this might be a limit, if not it seems at least to be much harder to find such an algorithm. The lower limit is clearly $O(m')$ since this is the size of the result. Algorithm *Elimination game* can be implemented to triangulate a graph in $O(m')$ time given the ordering of the vertices. From this we can conclude that finding an ordering that gives us a minimal triangulation is the task that causes us most problems, and not the part of adding the edges and constructing the graph.

Bibliography

[1] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[2] Philip A. Bernstein and Nathan Goodman. Power of natural semijoins. *SIAM J. Comput.*, 10(4):751–771, 1981.

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

[4] Anne Berry, Jean-Paul Bordat, and Pinar Heggernes. Recognizing weakly triangulated graphs by edge separability. *Nordic J. Comput.*, 7(3):164–177, 2000.

[5] Anne Berry, Jean-Paul Bordat, Pinar Heggernes, Simonet Genevieve, and Yngve Villanger. A wide-range algorithm for minimal triangulation from an arbitray ordening. In *Preparation for journal submition.*

[6] Jean R. S. Blair and Barry Peyton. An introduction to chordal graphs and clique trees. In *Graph theory and sparse matrix computation*, pages 1–29. Springer, New York, 1993.

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

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

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

[10] Pinar Heggernes and Yngve Villanger. Efficient implementation of a minimal triangulation algorithm. In *To be presented at European Symposium on Algorithms, Rome, Italy*, 2002.

[11] Chin Wen Ho and R. C. T. Lee. Counting clique trees and computing perfect elimination schemes in parallel. *Inform. Process. Lett.*, 31(2):61–68, 1989.

[12] Ton Kloks, Dieter Kratsch, and Jeremy Spinrad. Treewidth and pathwidth of cocomparability graphs of bounded dimension, 1993.

[13] Ton Kloks, Dieter Kratsch, and Jeremy Spinrad. On treewidth and minimum fill-in of asteroidal triple-free graphs. *Theoret. Comput. Sci.*, 175(2):309–335, 1997. Orders, algorithms and applications (Lyon, 1994).

[14] 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, 1962/1963.

[15] Tatsuo Ohtsuki, Lap Kit Cheung, and Toshio Fujisawa. Minimal triangulation of a graph and optimal pivoting order in a sparse matrix. *J. Math. Anal. Appl.*, 54(3):622–633, 1976.

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

[17] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5(2):266–283, 1976.

[18] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[19] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

[20] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Algebraic Discrete Methods*, 2(1):77–79, 1981.