Minimizing fill-in size and elimination tree height in parallel Cholesky factorization

Thesis for the degree of Cand. Scient.
by

Pinar Heggernes

Department of Informatics
University of Bergen, Norway

April, 1992

Acknowledgments

First of all, I thank my supervisor Professor Bengt Aspvall for all guidance and encouragement he has given me throughout my work on this thesis. He always supported my ideas, and had always new ideas when I did not know what to do. His enthusiasm about my work has inspired and encouraged me, and he was always optimistic and comforting whenever I wondered how this work would end.

Special thanks to Fredrik Manne without whom I would never have started working on elimination trees. He has been a second advisor for me as well as a friend, and was helpful in many more ways than I can list here. He listened to my ideas and read my results thoroughly, and his comments were an important contribution to the final product.

I thank Trond-Henning Olesen for introducing me to sparse matrices and elimination trees, and for answering patiently to my countless questions.

The results in Chapter 4 were presented at a workshop at the University of Minnesota in Minneapolis in October, 1991. It was a great experience to meet all the people whose names I had read many times on articles and books. I was happy to be able to present my work, and it was inspiring to hear about recent research on this area. I thank John Gilbert and Robert Schreiber for useful discussions in Minneapolis, and I thank everybody who made this trip possible.

I also thank the graduate students at the Department of Informatics for all the joy and fun we have had together. Special thanks to Linda for being my best friend and for making my graduate student days colorful and beautiful.

Finally, I thank my husband Frank for being there for me when I was worried about my work and needed comforting. He has supported and encouraged me in everything I have done, and without him the work on this thesis would have taken much longer time.

Bergen, April 7, 1992

Pinar Heggernes

Contents

Chapter 1

Introduction

Many problems that frequently arise in science and engineering, require the solution of large, sparse systems of linear equations. These systems are often in the form $A\mathbf{x} = \mathbf{b}$, where A is a symmetric, positive definite matrix. One way to solve this kind of equation systems is *Cholesky factorization.* In this thesis, we look at some approaches towards minimizing the amount of time required in Cholesky factorization.

Since the systems mentioned are often very large, it is quite time consuming to solve them in a straight forward way by simply using Cholesky factorization. In most cases, the matrix A is so large that the available memory may not even be enough to store the whole matrix. Therefore, we have to exploit the sparsity of A by only storing its non-zero elements. We will see later that by changing the order of the non-zero elements in A, it is also possible to reduce the amount of work needed to be done in the Cholesky factorization. In addition, if some of this work can be done in parallel, the time needed can be reduced further.

In the rest of this chapter, we give an introduction to the graph terminology that is used throughout this thesis. We give a description of Cholesky factorization, and we mention a class of difficult problems called the *NP-complete* problems. We end the chapter with a brief section on parallel computing.

Chapter 2 gives further introduction to symbolic Cholesky factorization on graphs. We introduce central terms like *elimination trees* and *fill-in*, and explain how low elimination trees and less fill-in reduce the computation time. The aim of this thesis is to find new algorithms for achieving these goals.

In Chapter 3, we look closer at a method called *Nested Dissection* which is known to usually produce low elimination trees. We show that this method can indeed give good results for some classes of graphs, but we also show that for some other classes of graphs, Nested Dissection may produce very high elimination trees. Some existing results on this method are also mentioned.

Chapters 4 and 5 contain new results on reducing fill-in size and reducing elimination tree height. We introduce *separator trees* and *clique trees* that are used to show these new results, and to develop new algorithms. The approaches in these two chapters are quite different. In Chapter 4, we start with a low elimination tree and try to reduce the amount of fill-in while we preserve the elimination tree structure. In Chapter 5, we find minimum height elimination trees for classes of graphs for which there exist perfect elimination orderings.

Chapter 6 concludes this thesis, and we also mention some interesting open problems.

1.1 Graph terminology

A graph $G = (V, E)$ consists of a set V of *vertices*, and a set E of *edges*. Each edge corresponds to an unordered pair of distinct vertices (v, w). (Sometimes self-loops, which are edges from a vertex to itself, are allowed; they will not be allowed in this text.) Vertices v and w are *neighbors* if $(v, w) \in E$. The *degree* of a vertex v is the number of neighbors of v and is denoted by $d(v)$. The set of all the neighbors of v in G is called $adj_G(v)$. Thus, $d(v) = |adj_G(v)|$. If $S \subset V$, then $adj_G(S) = \{w \mid (v, w) \in E, v \in S, w \notin S\}$.

A *path* from v_1 to v_k is a sequence of vertices $v_1, v_2, ..., v_k$ that are connected by the edges $(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k)$ (these edges are also usually considered to be a part of the path). In this text, we assume that $k \geq 2$, hence a path has at least one edge. A path is *simple* if each vertex appears in it at most once. The *length* of a path is the number of edges in it. An edge can be considered as a path of length 1. A *circuit* is a path whose first and last vertices are the same. A *cycle* is a circuit where, except for the first and last vertices, no vertex appears more than once.

A *subgraph* of a graph $G = (V, E)$ is a graph $H = (U, F)$ such that $U \subseteq V$ and $F \subseteq E$. An *induced subgraph* of a graph $G = (V, E)$ is a graph $H = (U, F)$ such that $U \subseteq V$ and F consists of all the edges in E both of whose vertices belong to U.

A graph is *connected* if there is a path from every vertex to every other vertex. If a graph G is not connected, then it can be partitioned in a unique way into a set of connected subgraphs called the *connected components* (or just *components*) of G. A connected component H of G is a maximal connected subgraph; that is, no other connected subgraph of G contains H.

A *separator* $S \subset V$ is a set of vertices whose removal from G disconnects G into two or more connected components. A separator S is *minimal* if and only if no proper subset of S is a separator in G. A *v-w separator* $S \subset V$ is a set of vertices whose removal from G cuts every path between v and w in G, where $v, w \in V$ and $(v, w) \notin E$. A v-w separator S is *minimal* if and only if no proper subset of S is a v-w separator. Every minimal separator in G is also a minimal v-w separator for some v and $w \in V$.

A graph $G = (V, E)$ is *complete* if there are edges between every pair of vertices in V. A *clique* in G is a complete subgraph of G. A clique C is *maximal* if and only if no other clique in G contains C.

A *bipartite graph* $G = (V_1, V_2, E)$ is a graph whose vertices can be divided into two sets V_1 and V_2 such that all edges in E connect vertices from V_1 to vertices in V_2. Thus, there are no edges between vertices that are both in V_1, or that are both in V_2. A bipartite graph $G = (V_1, V_2, E)$ is *complete* if every vertex in V_1 has edges to every vertex in V_2. A complete bipartite graph G is often denoted by $K_{r,s}$, where $r = |V_1|$ and $s = |V_2|$.

A *directed graph* $G = (V, E)$ is a graph where each edge in E is an ordered pair of distinct vertices $<v, w>$. The order between the two vertices an edge connects is important. The edge $<v, w>$ is *from v to w*. In a directed graph, we also distinguish between the *indegree* and the *outdegree* of a vertex. The indegree of a vertex v is the number of edges from other vertices to v, whereas the outdegree of v is the number of vertices from v to other vertices.

A *forest* is a graph that does not contain a cycle. A *tree* is a connected forest. A *spanning tree* of an undirected graph G is a subgraph of G that is a tree and that contains all the vertices of G. A *rooted tree*

$T = (V, E)$ is a directed tree with one distinguished vertex called the *root*. In a rooted tree, the outdegree of the root is 0, and the outdegree of every other vertex is 1. If the edge $<v, w> \in E$, then w is the *parent* of v, while v is a *child* of w. The vertices in V that do not have any children, are called the *leaves* of T. If there is a directed path in T from a vertex v to a vertex w, then w is an *ancestor* of v, while v is a *descendant* of w.

This brief introduction to graphs is sufficient to get started. More terms will be introduced later on as we proceed with new topics. We will also introduce our own terms when there are no existing standard names for the structures that we wish to express.

1.2 Cholesky factorization

In this section, we give a description of Cholesky factorization. We do not go into details, and this section is meant to provide a brief background for the problems that we will be working on in this thesis. We assume a general knowledge of basic linear algebra, and therefore we do not explain basic algebraic terms. More detailed information can be found in [8].

In this thesis, we consider the solution of equation systems of the form $A\mathbf{x} = \mathbf{b}$, where A is a sparse, symmetric, positive definite matrix. Thus A has many zero elements, but its diagonal consists only of non-zero elements. One way to solve such a system is by using Cholesky factorization. By the way we have defined A, there always exists a lower triangular matrix L such that $A = LL^T$. Matrices L and L^T are called the *Cholesky factors* of A.

Solving $A\mathbf{x} = \mathbf{b}$ by Cholesky factorization can be summerized in the following way: Find L such that $A = LL^T$. Then solve $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$ and $L^T\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$. Matrices L and L^T are found by the following recursive formula:

$$A = \begin{bmatrix} d & \mathbf{v}^T \\ \mathbf{v} & B \end{bmatrix} = \begin{bmatrix} \sqrt{d} & \mathbf{0}^T \\ \mathbf{w} & I \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & \hat{A} \end{bmatrix} \begin{bmatrix} \sqrt{d} & \mathbf{w}^T \\ \mathbf{0} & I \end{bmatrix}$$

where $\mathbf{w} = (1/\sqrt{d})\mathbf{v}$ and $\hat{A} = B - \mathbf{w}\mathbf{w}^T$. (The letters in boldface represent vectors, and the capital letters represent matrices). The Cholesky factors of $\hat{A}$ are found recursively by the same formula, and if $\hat{A} = \hat{L}\hat{L}^T$, then

$$A = \begin{bmatrix} \sqrt{d} & \mathbf{0}^T \\ \mathbf{w} & \hat{L} \end{bmatrix} \begin{bmatrix} \sqrt{d} & \mathbf{w}^T \\ \mathbf{0} & \hat{L}^T \end{bmatrix} = LL^T.$$

We will be concentrating on symbolic Cholesky factorization, therefore the numerical values of the elements in the matrix A are irrelevant for our purposes. What is more important is the number of non-zero elements in A and in its factors L and L^T. We often exploit the sparsity of A by storing only the non-zero elements. For the same reason, we want L and L^T to be as sparse as possible, too. In addition, since most of the algorithms only consider the non-zero elements, computations concerning L take less time if L is sparse.

The element of A in row i and column j is denoted by a_{ij}. We assume that numerical cancellation does not occur, and by the factorization formula, if the element $a_{ij} \neq 0$, then $l_{ij} \neq 0$ in L. Thus $L + L^T$ has all the non-zero elements of A. In addition, the number of non-zero elements in $L + L^T$ is often greater than the number of non-zero elements in A, and in the worst case, $L + L^T$ may be full. If $a_{ij} = 0$ and $l_{ij} \neq 0$, then l_{ij} is called a *fill-in* element. The *fill-in size* is the number of fill-in elements.

Our goal is to get the fill-in size as small as possible, and to be able to perform the factorization in parallel. In order to reduce the fill-in size and achieve parallelity, we perform a symmetric permutation of A before the factorization. This is done by factorizing PAP^T instead of A, where P is a permutation matrix. Dependent of P, PAP^T may have a better structure regarding fill-in size and parallelity. For the numerical solution, we must remember that the equation system is now $(PAP^T)(P\mathbf{x}) = (P\mathbf{b})$, instead of $A\mathbf{x} = \mathbf{b}$.

To summerize, solving $A\mathbf{x} = \mathbf{b}$ by symbolic Cholesky factorization is done in the following two steps.

1. Perform a symmetric permutation of A in order to get better parallelity and low fill-in.

2. Find the numerical solution.

This thesis concentrates on Step 1. Our aim is to find methods for solving the problem described by this step. In Step 1, we work on the non-zero structure of A. We must also be able to find the structure of L in order to see if the permutation has the desired properties. In our approach we use graph theory, and work directly on graphs when trying to solve this problem.

We now connect matrices to graph theory, and show how a symmetric, positive definite matrix A can be expressed as a graph. Since A is symmetric and has no zero elements in its diagonal, the non-zero structure of A can be represented as a graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$, and $(v_i, v_j) \in E$ if and only if $a_{ij} = a_{ji} \neq 0$, and $i \neq j$. Thus the edges in G represent the non-zero elements in A. We write $G(A)$ when we want to emphasize that G is the graph of A. An example is shown in Figure 1.1.

Figure 1.1: A symmetric matrix and its graph representation

If A is reducible, then the graph of A is not connected. In this case, A can be symmetricly permuted so that PAP^T is block diagonal, and each block can be factorized separately. In this thesis, we assume that the matrices we are working with, are irreducible, and thus A always has a connected graph.

In Chapter 2, we will see how we can find the non-zero structure of L by only working on the graph of A. We will therefore only be working on graphs further on. Chapter 2 also contains an explanation of how a symmetric permutation of a matrix affects the corresponding graph. In the rest of this thesis, whenever we mention a matrix A, we mean a sparse, symmetric, positive definite matrix A.

1.3 Parallel computing

As computers have become more and more powerful, the demand for even more powerful computers has increased. Parallelism represents the most feasible way to achieve faster computers, and parallel computers are now commercially available throughout the world.

Parallel computing is a kind of information processing that emphasizes the concurrent manipulation of data elements belonging to one or more processes solving a single problem. A *parallel computer* is a computer for the purpose of parallel computing. Parallel computers have many processors that can work on different parts of a problem simultaneously. *Parallel algorithms* are designed especially for parallel computers, and aim to get as much of the work as possible done in parallel. Most problems have parts that must be done sequentially, and it is important to find out what can be done in parallel.

In *parallel Cholesky factorization*, we can perform the factorization simultaneously on different parts of the matrix that are not dependent of each other. If a matrix does not allow much parallelism in factorization, then it can be symmetricly permuted as we have seen in the previous section, and the resulting matrix may allow more parallelism. In Chapter 2, we will see what this means in graph representation, and we will introduce a structure called an *elimination tree* that expresses the dependencies among the elements.

1.4 NP-Complete problems

Some important problems connected to parallel Cholesky factorization are NP-complete. The class of NP-complete problems contains decision problems for which no polynomial-time algorithm is known. The existence of a polynomial-time algorithm for one NP-complete problem implies the existence of a polynomial-time algorithm for every NP-complete problem.

A *decision problem* with a given input is a question to which the answer is either 'yes', or 'no', dependent of the input. A decision problem Γ is *polynomially reducible* to another decision problem Δ if there exists a polynomial-time algorithm that converts each input x for Γ to another input y for Δ, such that the answer to Δ given y is 'yes' if and only if the answer to Γ given x is 'yes'. The class of decision problems for which there exists a nondeterministic algorithm whose running time is a polynomial in the size of input is called *NP*. A problem Γ is *NP-hard* if and only if every problem in NP is polynomially reducible to Γ. A problem Γ is *NP-complete* if and only if $\Gamma \in$ NP and Γ is NP-hard.

In order to show that a decision problem Γ is NP-complete, we must show that $\Gamma \in$ NP and that another NP-complete problem Δ is polynomially reducible to Γ. To show that a problem Γ is NP-hard, it is enough to show that an NP-complete problem Δ is polynomially reducible to Γ. If a decision problem is NP-complete, then the corresponding search or optimization problem is NP-hard. More detailed information about NP-completeness can be found in [5].

It has been conjectured that no polynomial-time algorithm can solve NP-complete and NP-hard problems. However, this has not yet been shown. In order to approach a solution, we can use heuristics. A *heuristic* is a polynomial-time algorithm that exploits some properties of the specific problem in order to avoid exhaustive search. Some heuristics guarantee to return a solution that is near the optimal, whereas others do not. Although heuristics may return the optimal solution sometimes, we cannot know if the returned answer is optimal.

Chapter 2

Graph elimination

In this chapter, we introduce some more terms concerning symbolic Cholesky factorization of an $n \times n$ matrix $A = (a_{ij})$. We have previously seen how to perform the symbolic factorization on A, and will now see how to do this directly on the graph of A.

Let $G = (V, E)$ be the graph of A, and let $L = (l_{ij})$ be the Cholesky factor of A. The graph of $L + L^T$ is called the *filled graph* $G^* = (V, E^*)$, where $E \subseteq E^*$. It should be clear that G^* has all the edges of G, since $L + L^T$ has all the elements of A. As $L + L^T$ can have more non-zero elements than A, the filled graph G^* can have more edges than G, but it has the same vertex set.

The following algorithm from Parter [17] shows how the filled graph $G^* = (V, E^*)$ of a given graph $G = (V, E)$ is found. We assume that $V = \{v_1, v_2, ..., v_n\}$.

```
Algorithm Eliminate ($G$, $G^*$ : Graph);

begin
    $G^* = G$;
    for i = 1 to $n$ do
        mark $v_i$;
        $F = \{(u, w) \mid u$ and $w$ are unmarked neighbors of $v_i$ in $E^*\}$;
        $E^* = E^* \cup F$;
    end-for;
end;
```

What is done in one step of the loop is called *eliminating* v_i. We see that when v_i is eliminated, the neighbors v_j of v_i, where $j > i$, induce a clique in G^*. The edges in $(E^* - E)$ are called the *fill-in edges* of G^*. The following 'path theorem' from Rose, Tarjan and Lueker [19], shows where the fill-in edges occur. We quote it without giving the proof.

Theorem 2.1 *Let $G = (V, E)$ be a graph, and let $G^* = (V, E^*)$ be the filled graph of G. Then $(v_i, v_j) \in E^*$ if and only if there exists a path $v_i, v_{p_1}, ..., v_{p_t}, v_j$ in G such that all subscripts in $\{p_1, ..., p_t\} < min(i, j)$.*

An *elimination ordering* α on a graph G is an ordering of the vertices in G, where $\alpha : V \rightarrow \{1, 2, ..., n\}$ is a one-to-one function. The new graph which we get by renumbering the vertices in G is called G_α. Running the algorithm *Eliminate* on G_α will return the filled graph G^*_α. Whereas G_α has the same number of edges as G, G^*_α does not necessarily have the same number of edges as G^*. Thus different elimination orderings on a graph can result in different amount of fill-in. If G is the graph of a matrix A, reordering G is equivalent to performing a symmetric permutation on A. Then G_α is the graph of PAP^T, where P is a permutation matrix with each column j having its non-zero element in row $\alpha(v_j)$. An elimination ordering α is called a *perfect elimination ordering* if G^*_α is isomorphic to G_α; i.e., the elimination of G_α results in no fill-in edges. An example showing different elimination orderings on a graph is shown in Figure 2.1

An edge in G is called a *chord* of a cycle if it joins two nonconsecutive vertices on the cycle. A graph is *chordal* if every cycle of length at least four has a chord. Fulkerson and Gross show in [4] that chordal graphs are exactly those graphs with perfect elimination orderings. All filled graphs are chordal. Thus the chordality of a graph G can be established by finding a perfect elimination ordering for G. A *simplicial vertex* in a chordal graph G, is a vertex whose neighbors induce a clique in G. Eliminating a simplicial vertex in G does not cause any fill-in edges. Dirac shows in [3] that a chordal graph is either a complete graph, or has at least two non-adjacent simplicial vertices.

Figure 2.1: a) A graph G and its filled graph G^*. b) G_α and G^*_α. c) G_β and G^*_β where β is a perfect elimination ordering on G.

2.1 Elimination trees

The elimination tree is a structure that plays an important role in sparse, parallel Cholesky factorization. We can view the elimination tree as providing the minimal amount of information on column dependencies in the Cholesky factor. This structure was first introduced by Schreiber [20].

The *elimination tree* of A is a rooted tree T with the vertex set $\{v_1, v_2, ..., v_n\}$. An edge $<v_i, v_j>$ is in T if and only if $j = min\{k \mid l_{ki} \neq 0$ and $k > i\}$. Thus v_j is the parent of v_i in T if the first non-zero element under the diagonal in column i in L is in row j.

The elimination tree T can also be found directly from G, where G is the graph of A. Then T is called the elimination tree of G and is denoted by $T(G)$. The edges in T are determined by the edges in the filled graph G^*. A vertex v_j is the parent of another vertex v_i in T if and only if $j = min\{k \mid (v_i, v_k) \in E^* \text{ and } k > i\}$. An example is shown in Figure 2.2. The dashed line in the graph represents a fill-in edge.

Figure 2.2: A matrix, its filled graph and elimination tree.

Let v be a vertex in T. We let $T(v)$ denote the subtree of T with root at v. Thus $T(v)$ is the subgraph of T induced by v and all of its descendants. If v is the root of T, then $T = T(v)$.

We will now take a closer look at the relationship between elimination trees and Cholesky factorization. In the following results, we let $G = (V, E) = G(A)$, where $A = LL^T$, and $T = T(G)$. Theorems 2.2 and 2.4 - 2.7 are from Liu [13], and Theorem 2.3 is from Schreiber [20].

Theorem 2.2 *For $i < j$, the numerical values of column j depend on column i in L if and only if $(v_i, v_j) \in E^*$.*

Theorem 2.3 *If $(v_i, v_j) \in E^*$, where $i < j$, then v_j is an ancestor of v_i in T.*

We see from these two theorems, that the elimination tree alone, gives enough information about column dependencies in L. If v_j is an ancestor of v_i in T, then column j may depend on column i in L. Therefore, for each descendant v_i of v_j in T, column i must be computed before column j in L. We can thus view the elimination tree as a precedence graph, where each vertex must be eliminated before its parent. The computation begins with the leaves and continues upward to the root.

The following Corollary follows directly from Theorem 2.3.

Corollary 2.1 *If G has a clique of size k, then T cannot be lower than $k-1$.*

We will see now that the height of the elimination tree is important for parallel Cholesky factorization. The following two theorems show which parts of the problem can be solved in parallel.

Theorem 2.4 *Let $T(v_i)$ and $T(v_j)$ be two disjoint subtrees of T. Then for each $v_s \in T(v_i)$ and each $v_t \in T(v_j)$, the edge $(v_s, v_t) \notin E^*$.*

Theorem 2.5 *If $T(v_i)$ and $T(v_j)$ are disjoint subtrees of T, then columns i and j of L can be computed in parallel.*

As we can see from these theorems, low elimination trees are necessary in order to achieve parallelity in Cholesky factorization. Low elimination trees imply branching, and this gives more parallelity since disjoint branches can be computed in parallel.

Theorem 2.6 *Let $k < i < j$. The edge $(v_i, v_j) \in E^*$ if and only if $(v_i, v_j) \in E$, or the vertex v_i is the ancestor of some vertex v_k in T, where $(v_j, v_k) \in E$.*

From Theorem 2.6, it follows that if a vertex v_j is an ancestor of another vertex v_k in T, where $(v_j, v_k) \in E$, then for every vertex v_i on the path from v_k to v_j in T, the edge $(v_i, v_j) \in E^*$. For the next theorem, note that $adj_G(H)$ for a subgraph H of G, is the same as $adj_G(U)$, where U is the vertex set of H.

Theorem 2.7 *The subgraph of G^* induced by $adj_G(T(v_i)) \cup \{v_i\}$ is a clique.*

It should be clear by now that the height of the elimination tree represents a measure for the amount of work in parallel elimination. Therefore, two of our main goals in this thesis, are to find low elimination trees and to show new results on elimination tree height. Liu gives more detailed information and results on elimination trees in [13].

2.2 Elimination orderings

For a sequential factorizing algorithm, only the fill-in size is important for time complexity. Since every element must be accessed sequentially, there is less work to be done if there are few fill-in elements.

Whereas for efficient parallel Cholesky factorization, we have to find orderings that result in both low fill-in size and low elimination trees. For the time complexity of parallel algorithms, elimination tree height is decisive, but the fill-in size still plays an important role. Hafsteinsson shows in [9] that, if we are working with an $n \times n$ matrix, and we have a parallel computer with n^2 processors, then the fill-in size does not affect time complexity, and the computation time depends only on the elimination tree height. But, if n is large, it is not realistic to assume n^2 processors, and with less processors, the fill-in size also affects the computation time.

As we can see, we have two problems: low fill-in and low elimination trees. Unfortunately, both the problem of finding an ordering that results in minimum fill-in size, and the problem of finding an ordering that produces an elimination tree of minimum height, are NP-hard problems. Thus it is highly unlikely that a polynomial time algorithm can solve these problems for general graphs. Therefore we have to use heuristics. We now mention two well known algorithms called *Minimum Degree* and *Nested Dissection*. Minimum Degree was introduced by Rose in [18] in order to find orderings that result in low fill-in size. Nested Dissection aims to produce low elimination trees, and was first used by George in 1973 on grid graphs. An example of these two algorithms applied on the same graph is shown in Figure 2.3.

```
Algorithm Minimum Degree ($G$ : Graph);

begin
    while there are unmarked vertices in $G$ do
        let $v$ be an unmarked vertex with minimum degree;
        eliminate $v$;
        mark $v$;
    end-while;
end;
```

Figure 2.3: a) A chordal graph G. b) G^*_α and $T(G_\alpha)$, where α is a Minimum Degree ordering. c) G^*_β and $T(G_\beta)$, where β is a Nested Dissection ordering.

```
**Algorithm** *Nested Dissection* ($G$ : Graph);

begin
    if $G$ is complete then
        eliminate the vertices in $G$ in an arbitrary order
    else
        let $S$ be a separator in $G$;
        for each component $C$ of $(G - S)$ do
            Nested Dissection ($C$);
        eliminate the vertices in $S$ in an arbitrary order;
    end-if;
end;
```

George and Liu give a detailed study of these algorithms in [6]. In Nested Dissection, the choice of separators is essential for the height of the elimination tree that the algorithm produces. For general graphs, it is NP-hard to find separators for Nested Dissection that give minimum elimination tree height. In the next chapter, we talk more about Nested Dissection, and look at different ways of choosing separators. We also show that the relative numbering of vertices belonging to the same separator can affect the fill-in size.

In Parallel Cholesky factorization, we aim to minimize both the elimination tree height and the fill-in size. Unfortunately, these two goals may be contradictory. There are many examples of graphs for which it is easy to show that one single ordering cannot produce both minimum fill-in size and minimum elimination tree height. Thus, there do not exist orderings that result in both minimum fill-in size and minimum elimination tree height for all graphs. Therefore, the aim has to be to find elimination orderings that produce low elimination trees and that result in low fill-in size. There are two possible methods for achieving this:

1. Find an elimination ordering α that results in low fill-in size. Then find an ordering β that produces a low elimination tree while it preserves the size of the fill-in produced by α.

2. Find an elimination ordering α that produces a low elimination tree. Then find an ordering β that results in low fill-in size while it preserves the elimination tree produced by α.

The usual approach has been to use Method 1. Several algorithms that use this method already exist, like the algorithm that Jess and

Kees present in [11]. Little is done though, regarding algorithms that use Method 2. In this thesis, we present two different approaches for achieving low fill-in and low elimination trees. In Chapter 4, we present a new algorithm that uses method Method 2, and in Chapter 5 we concentrate on a variant of Method 1.

Chapter 3

Nested Dissection

Nested Dissection plays a central role in this thesis, and is used as a background for the algorithms that will be presented in the following chapters. In this chapter, we examine Nested Dissection more closely and show some new results on this method. We concentrate mainly on the height of the elimination trees produced by different Nested Dissection orderings.

In a Nested Dissection ordering, the choice of separators is decisive for the height of the elimination tree produced. If the separators are chosen arbitrarily, the height of the resulting elimination tree can be much higher than minimum. An interesting result is that there exist Nested Dissection orderings for all graphs that result in minimum height elimination trees. The following theorem from Manne [15] shows what kind of separators should be chosen in order to achieve minimum elimination tree height.

Theorem 3.1 *For every graph G, there exists a Nested Dissection ordering with minimal separators that produces an elimination tree of minimum height.*

By a Nested Dissection ordering with minimal separators, we mean that the separators chosen at each step of the algorithm, are minimal. This theorem provides a basis for the algorithms and results that will be presented later in the thesis.

It should be clear that the problem of finding a Nested Dissection ordering with minimal separators that produces a minimum height elimination tree, is also NP-hard. However, it is also shown in [15] that, for any Nested Dissection ordering, we can always make the chosen separators minimal without increasing the elimination tree height. Thus requiring the separators to be minimal can lead to lower elimination trees. In the rest of this thesis, we assume that the separators chosen by Nested Dissection orderings we are working with are minimal.

It has also been suggested to choose separators according to the way they divide the graph into components. George and Liu [6] require the remaining components after removing a separator, to be nearly equal in size. Gilbert [7] suggests choosing separators in such a way that, none of the new components has more than $n/2$ vertices, where n is the number of vertices in the original component. Although this suggestion usually leads to low elimination trees, we show that for some classes of graphs, it results in elimination trees that are exponentially higher than minimum.

In this chapter, we consider both sides of the problem of the height of the elimination trees produced by Nested Dissection. First we show that for some non-trivial classes of graphs it is easy to find low elimination trees with Nested Dissection. Then we show that for some other classes of graphs, Nested Dissection can produce very high elimination trees although the separators are minimal and divide the graph into components of size $\leq n/2$. In the rest of the chapter, when we are working on a graph G, we let n be the number of vertices in G unless specified otherwise.

3.1 Separator theorems

A class S of graphs satisfies an *f(n)-separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that every n-vertex graph $G \in S$ has a set of at most $\beta f(n)$ vertices whose removal leaves no connected component with more than αn vertices.

It is shown in [2] that if a graph G satisfies an $f(n)$-separator theorem, where $f(n) = c$ for a constant c, then there exists a Nested Dissection ordering on G that results in an elimination tree of height $O(c \log n)$.

In this section, we show that if a graph G and all of its subgraphs satisfy an n^ϵ-separator theorem, where $\epsilon > 0$, then an elimination tree of height $O(n^\epsilon)$ can be produced by a Nested Dissection Ordering on G.

Theorem 3.2 *Let G be a graph such that, G and all its subgraphs satisfy an n^ϵ-separator theorem, where $\epsilon > 0$. Then there exists a Nested*

Dissection ordering on G that results in an elimination tree of height $O(n^\epsilon)$.

Proof: By the separator theorem, we know that G has a separator S that contains at most βn^ϵ vertices, and that divides G into components of at most αn vertices. We choose S to be the top separator. For each remaining component, we can repeat this process recursively. Since the components also satisfy the same separator theorem, the height of the elimination tree produced this way, can be expressed by the following recursive function $h(n)$:

$$h(n) = \beta n^\epsilon + h(\alpha n)$$

We let $h(1) = 0$, since this is the height of an elimination tree that consists of a single vertex. The constants $\alpha < 1$ and $\beta > 0$ are given by the separator theorem. We now find an upper bound for $h(n)$.

$$h(n) \leq \beta n^\epsilon + \beta(\alpha n)^\epsilon + \beta(\alpha^2 n)^\epsilon + ... + \beta(\alpha^{\log_{1/\alpha} n} n)^\epsilon$$

$$= \beta \sum_{i=0}^{\log_{1/\alpha} n} (\alpha^i n)^\epsilon \leq \beta \sum_{i=0}^{\infty} (\alpha^i n)^\epsilon = \beta n^\epsilon \sum_{i=0}^{\infty} (\alpha^\epsilon)^i.$$

Since $\alpha^\epsilon < 1$, we have by the 'Geometric Series Theorem', that

$$\sum_{i=o}^{\infty} (\alpha^\epsilon)^i = \frac{1}{1-\alpha^\epsilon}.$$

Thus,

$$h(n) \leq \frac{\beta}{1-\alpha^\epsilon} n^\epsilon = O(n^\epsilon).$$

We have shown that $h(n) = O(n^\epsilon)$ is the height of the elimination tree found by the described Nested Dissection ordering, and this completes the proof. □

3.2 High elimination trees

In this section, we show that Nested Dissection orderings, although they use minimal separators that divide the graph into components of size $\leq n/2$, can result in elimination trees that are exponentially higher than the minimum elimination tree height.

In order to show our result, we define a special type of graph on which we use Nested Dissection. In our definition, we use star graphs. A *star graph* is a complete bipartite graph of the form $K_{1,s}$, and has $s+1$ vertices and s edges. One of the vertices has degree s, while all the others have degree 1.

Definition: For a fixed constant m, a *star path* is a graph $S_m(k) = (V_m(k), E_m(k))$, that is constructed by the following rules:
$S_m(1)$ is a star graph with m vertices, which is the same as $K_{1,m-1}$.
$S_m(k)$ is made by joining $S_m(k-1)$ and a star graph G with $|V_m(k-1)|$ vertices with an edge between a vertex in $S_m(k-1)$ with highest degree and the vertex in G with the highest degree. Ties can be broken arbitrarily. (e.g. in $S_m(2)$, there are two vertices with the highest degree.)

Note that $|V_m(k)| = 2^{k-1}m$. This can be easily shown by induction since $|V_m(1)| = m$ and $|V_m(k)| = 2|V_m(k-1)|$, as the construction of the star path is defined. Any constant $m \geq 3$ may be chosen, since the results in the following text are independent of m.

Lemma 3.1 *There exists a Nested Dissection ordering on $S_m(k)$ that chooses minimal separators such that the remaining components have $\leq n/2$ vertices, and that produces an elimination tree of height k.*

Proof: The proof is by induction, and our induction hypothesis is identical to Lemma 3.1.

Induction hypothesis: There exists a Nested Dissection ordering of $S_m(k)$ that chooses separators as described above, and that produces an elimination tree of height k.

We show this by induction on k. The base case is when $k = 1$. This means that we have $S_m(1)$ which is a star graph with m vertices. We choose the vertex with the highest degree as the separator and thus get an elimination tree of height 1.

We assume now that the induction hypothesis is true for $S_m(k)$, and we will show that the vertices in $S_m(k+1)$ can be eliminated to produce an elimination tree of height $k+1$. We choose the vertex x with the highest degree to be eliminated last. This is shown in Figure 3.1. Thus x is the first separator. We are allowed to do this since the remaining components all have $\leq \lfloor n/2 \rfloor$ vertices. Now we choose to eliminate the vertices in $S_m(k)$ first. We know by the induction hypothesis, that there exists a Nested Dissection ordering of $S_m(k)$ such that the height

Figure 3.1: Choosing the first separator in $S_m(k+1)$.

Figure 3.2: An elimination tree for $S_m(k+1)$.

of $T(S_m(k))$ is k. We then eliminate the leaves that are neighbors of x, and x is eliminated last. This produces one more level on top of $T(S_m(k))$, and $T(S_m(k))$ will hang from x. (See Figure 3.2). Thus this Nested Dissection ordering of $S_m(k+1)$ produces an elimination tree of height $k+1$ as claimed. $\square$

Note that a Nested Dissection ordering as described in Lemma 3.1, could also have chosen the root of the second largest star graph as the separator in each step. This would produce an elimination tree of height $k/2$. But since we are showing that there exists *a* Nested Dissection ordering as required that produces an elimination tree of height k, this causes no problems for the proof. In addition, we will see that an elimination tree of height $k/2$ is still exponentially higher than minimum.

Lemma 3.2 *Minimum elimination tree height for* $S_m(k)$ *is* $\lfloor \log_2 k \rfloor + 1$.

Proof: We use the following elimination ordering on $S_m(k)$: Eliminate all the vertices with degree 1 first, and then eliminate the remaining path by using a Nested Dissection ordering. When all the vertices of degree 1 are eliminated, the remaining graph is a path of k vertices. We

Figure 3.3: Finding the elimination tree of minimum height.

know from Manne [14], that the minimum elimination tree height for such a path is $\lfloor \log_2 k \rfloor$, and that this can be achieved by a Nested Dissection ordering with minimal separators. When we have the elimination tree for the path, it is enough to hang the leaves on the appropriate vertices. This is shown in Figure 3.3. A leaf that hangs from a vertex v in $S_m(k)$, hangs from v in the elimination tree also. This gives an elimination tree of height $\lfloor \log_2 k \rfloor + 1$. $\square$

Note that the minimum elimination tree height for $S_m(k)$ is then $O(\log \log n)$, when $|V_m(k)| = n$. This is because $n = 2^{k-1}m$, and thus $k = O(\log_2 n)$.

Theorem 3.3 *A Nested Dissection ordering with minimal separators and remaining components of size $\leq n/2$ on a star path, produces an elimination tree that is exponentially higher than minimum.*

Proof: Let h_1 be the height of an elimination tree for a star path, produced in the same way as described in the proof of Lemma 3.1. Let h_2 be the height of an elimination tree for the same star path, produced as described in the proof of Lemma 3.2. It is easy to see that $h_1 = \Theta(2^{h_2})$. $\square$

Chapter 4

Reducing the fill-in size

We will in this chapter, look at how we can further reduce the fill-in size when a low elimination tree ordering is already given. We show how to reduce the fill-in size for a given elimination tree without changing its structure. We have seen in Chapter 3 that there always exists a Nested Dissection ordering with minimal separators on a graph G that results in an elimination tree of minimum height. This type of Nested Dissection does not have any requirements on the size of the remaining components. Throughout this chapter, we use Nested Dissection orderings that choose minimal separators in each step, and there are no requirements on the size of the components. We show that for such orderings, the problem of computing minimum fill-in decomposes into subproblems for each separator. These subproblems are shown to be also NP-hard. However, for small separators, they can be solved optimally in reasonable time. We also compare several heuristics for these subproblems.

We now give a brief discussion on why we think it might be better to start with low elimination trees and try to reduce the fill-in size, instead of starting with low fill-in and trying to reduce the elimination tree height. If two vertices v_i and v_j of a graph G are numbered respectively i and j where $i > j$, then v_i might be an ancestor of v_j in the resulting elimination tree although $l_{ij} = 0$. This case is shown in Figure 4.1 a). Thus, low fill-in does not imply a low elimination tree. But if $i > j$ and v_i is not an ancestor of v_j in the elimination tree, as shown in Figure 4.1 b), then l_{ij} must be zero. Reducing the elimination tree height might therefore also result in low fill-in.

Figure 4.1: a) l_{ij} might or might not be 0. b) l_{ij} must be 0.

4.1 A local problem

We first give a formal description of the problem mentioned in the previous section. We will call this problem Γ.

Γ: Given a graph G, a Nested Dissection ordering of G with minimal separators, and the resulting elimination tree T, find an ordering that gives minimum fill-in while preserving the structure of T.

In this section we show that in order to solve Γ, we need to change the relative ordering of the vertices in each separator, and that this local reordering for each separator can be done independently of the other separators in the graph.

Definition: Let G be a graph ordered by a Nested Dissection ordering α with minimal separators, and let T be the resulting elimination tree. We define a *separator tree* corresponding to T, to be a tree whose vertices represent the separators of G chosen by α, and whose leaves represent the components of G that cannot be separated further.

An example of a separator tree is given in Figure 4.2. The vertices of a separator tree are called *s-nodes*. Every s-node consists of one or more vertices of G, and corresponds to a separator. The s-nodes in the separator tree are related to each other in the same way as the separators are related in T; i.e. the root s-node corresponds to the separator in G that is eliminated last. The leaf s-nodes correspond to the components of G that cannot be separated further and that are eliminated first.

Figure 4.2: The elimination tree and the separator tree for a graph.

Lemma 4.1 *Each separator in G chosen by a Nested Dissection ordering with minimal separators is a clique in the filled graph G^*.*

Proof: Lemma 4.1 follows if we show that the s-nodes in a separator tree ST are cliques in G^*. It is easy to see that the leaf s-nodes are cliques since they cannot be separated further by removal of any vertices. We have to show that all the other s-nodes are also cliques. Let S be any s-node in ST that is not a leaf. Let x be the vertex in S which is eliminated first and let y be a child of x in T, where T is an elimination tree that ST corresponds to. (See Figure 4.3). Since S is a minimal separator, there must be an edge in G from each vertex in S to some vertex in *T(y)*. Thus $S \subseteq adj_G(T(y))$. By Theorem 2.1, y has edges in G^* to all the vertices of $adj_G(T(y))$. Therefore when y is eliminated, $adj_G(T(y))$ becomes G^*. Thus S must also be a clique in G^*. $\square$

Figure 4.3: Picturizing the proof of Lemma 4.1.

Lemma 4.2 *Changing the relative ordering of the vertices in an s-node does not result in any changes in the elimination tree structure.*

Proof: Since we choose the separators before we order the vertices in them, a local reordering of vertices in a separator does not affect the choice of separators. It follows from Lemma 4.1 that the vertices in an s-node will induce a path in the elimination tree regardless of their relative ordering, and thus the elimination tree will remain the same. □

Lemma 4.2 shows that we can reorder the vertices in an s-node without changing the elimination tree structure. In order to find a good reordering of the vertices in an s-node, we have to know which portion of the fill-in can be reduced by such a reordering. We divide the fill-in that is caused by the elimination of vertices in S, into four disjoint groups:

1. Fill-in that occurs within S
2. Fill-in that occurs within another s-node
3. Fill-in that occurs between other s-nodes
4. Fill-in that occurs between S and other s-nodes

Some of this fill-in occurs regardless of the local ordering of the vertices in an s-node. The following lemmas show where this kind of fill-in occurs. Lemmas 4.3 and 4.4 are stated without proofs, since they both follow directly from Lemma 4.1.

Lemma 4.3 *Changing the relative ordering of the vertices in an s-node S does not result in any changes in the size of fill-in that occurs within S.*

Lemma 4.4 *Changing the relative ordering of the vertices in an s-node does not result in any changes in the size of fill-in that occurs within another s-node.*

Lemma 4.5 *Changing the relative ordering of the vertices in an s-node S does not result in any changes in the size of fill-in that occurs between other s-nodes.*

Proof: We need only to consider those s-nodes that lie on the path between S and the root s-node. Whatever lies below S is eliminated before S, and the s-nodes that lie elsewhere cannot be affected by the elimination of S. Let S_1, S_2 and S be s-nodes as shown in Figure 4.4. Let x_i be a vertex in S_1 and x_j be a vertex in S_2, where x_i and x_j are numbered respectively i and j, where $(i > j)$. By Theorem 2.1, we know that $l_{ij} \neq 0$ if and only if there exists a path $x_i, x_{p_1}, ..., x_{p_t}, x_j$ in G such that all subscripts in $\{p_1, ..., p_t\}$ are less than j. If no such path exists then a reordering of the vertices in S cannot create such a path, since all the vertices in S are numbered lower than j. By the same argument, if such a path does exist, then it will exist also after a reordering of the vertices in S. □

Figure 4.4: The s-nodes in the proof of Lemma 4.5.

The only vertices whose elimination may result in fill-in edges from S to higher numbered vertices, are the vertices in S and the descendants of S. It should be clear from the previous lemmas that, by reordering S, we can only reduce the number of fill-in edges between S and its higher numbered neighbors, that are caused by the elimination of vertices within S. Thus we need only to consider the fill-in edges that belong to Group 4.

Lemmas 4.1 - 4.5 and the discussion above suggest a bottom-up elimination tree reordering that, beginning with leaf s-nodes, reorders one s-node at a time. However, as we will see in the rest of this section, each s-node can be reordered independently of the other s-nodes. Thus the s-nodes can be reordered in an arbitrary order, or in parallel.

Definition: Let *G=(V,E)* be a graph, S a set of vertices from V, and let α be an ordering of the vertices in V. We define $madj_G(S)$ to be the set $\{x \notin S \mid (x,y) \in E,\ y \in S,\ x \text{ is higher numbered than all the vertices in } S\}$. We will call the set $(madj_{G^*}(S) - madj_G(S))$ the *fill-in neighbors* of S.

We will see in the next lemma that we do not have to consider the fill-in neighbors of S when we reorder S, since these have edges in G^* to all the vertices in S regardless of the relative ordering of the vertices in S.

Lemma 4.6 *Let G be a graph ordered by a Nested Dissection ordering with minimal separators, and let T and ST be the resulting elimination, and separator trees. If there is a fill-in edge in G^* from an s-node S to a higher numbered vertex v in T that is caused by the elimination of a descendant of S, then every vertex in S has an edge in G^* to v.*

Proof: Let S be any s-node in ST that is not a leaf or the root. Since there is a fill-in edge from S to v, by Theorem 2.6, there must be a vertex w as shown in Figure 4.5 that has an edge to v in G. By the same theorem, every vertex in S must also have edges to v. $\square$

Figure 4.5: Picturizing the proof of Lemma 4.6.

We see by Lemma 4.6 that when we reorder S, it is enough to consider the edges in G, rather than G^*. The fill-in neighbors of S caused by the elimination of the descendants of S, have edges to all the vertices in S. Therefore, they cannot cause any new fill-in when we reorder S. Hence we need only to consider edges between S and $madj_G(S)$. But some of the vertices in $madj_G(S)$ might also be fill-in neighbors of S; i.e. they might have fill-in edges to all the vertices in S, caused by the elimination of the descendants of S. Hence we are allowed to exclude such vertices from the neighborhood of S which we will work with, since they cannot introduce any more fill-in to S.

The following algorithm *Reorder* describes how a graph G can be reordered to reduce fill-in, given a separator tree for G. It also gives a description of the local reordering problem for each s-node. We will call this problem Δ.

```
Algorithm Reorder (ST: Separator tree; G: Graph);

begin
    mark the root s-node in ST;
    while there are unmarked s-nodes in ST do
        pick an unmarked s-node S in ST;
        $adj'(S) = madj_G(S) - \cup\{madj_G(C) | C \text{ is a descendant of } S\}$;
        $\Delta$:Find a reordering of S that gives minimum fill-in
            between S and $adj'(S)$;
        mark S;
    end-while;
end;
```

Theorem 4.1 *The algorithm Reorder solves the problem* Γ.

Proof: The proof follows from Lemmas 4.1 - 4.6 and the discussion above. □

We have shown that the problem Γ decomposes into smaller subproblems Δ, which can be solved independently for each s-node. For parallel algorithms, this means that the s-nodes can be reordered in parallel. This will reduce the time required to reorder a graph. In the next section, we show that Δ, though more restricted and structured than the general minimum fill-in problem, is still NP-hard.

Now we want to show that it is worth trying to reduce the fill-in size in an elimination tree, and that in some cases considerable amount of fill-in may be reduced. Let C and S be s-nodes as shown in Figure 4.6

a), where C is the parent of S. Let f be the number of edges in the filled graph between the vertices in $S \cup C$. Then we have

$$f \geq \frac{|C|(|C|-1)}{2} + \frac{|S|(|S|-1)}{2} + |C|,$$

$$f \leq \frac{|C|(|C|-1)}{2} + \frac{|S|(|S|-1)}{2} + |C||S|.$$

Figure 4.6: Illustrating the discussion on the number of fill-in edges.

Thus the maximum number of edges that might be possible to reduce is $|C|(|S|-1)$. Since for most values of C and S we have that

$$|C|(|S|-1) \leq \frac{|C|(|C|-1)}{2} + \frac{|S|(|S|-1)}{2} + |C|$$

there are not as many edges that can be reduced as the ones that must occur since C and S are cliques. But if C has several children as shown in Figure 4.6 b), then if all p children are of equal size, we have

$$f \geq \frac{|C|(|C|-1)}{2} + p\frac{|S|(|S|-1)}{2} + p|C|,$$

$$f \leq \frac{|C|(|C|-1)}{2} + p\frac{|S|(|S|-1)}{2} + p|C||S|.$$

If p is large enough, and $|C|$ is larger enough than $|S|$, then we can have

$$p|C|(|S|-1) \geq \frac{|C|(|C|-1)}{2} + p\frac{|S|(|S|-1)}{2} + p|C|.$$

Thus in some cases we can potentially have at least as many unnecessary fill-in edges as the ones that are bound to occur, and it is possible to reduce the number of such fill-in edges.

4.2 Still NP-hard

In this section we are going to show that the problem Δ, described in the previous section, is NP-hard. Let $adj'(S)$ be as described by the algorithm in the previous section. We can view S and its neighborhood as a separate graph with the vertex set consisting of S and $adj'(S)$. We do not have to consider fill-in between two vertices that are both in S or the fill-in between two vertices that are both in $adj'(S)$. The only edges we are interested in, are the ones between S and $adj'(S)$. Therefore, we can work on a graph that has only the vertices in S and $adj'(S)$, and only the edges that are between S and $adj'(S)$. This observation leads us to the following definition.

Definition: Let S be an s-node. We define G_S to be the bipartite graph $(S, adj'(S), E)$, where E contains only those edges between the vertices in S and their neighbors in $adj'(S)$. (See Figure 4.7 b).

In order to show that Δ is NP-hard, we need some results from Yannakakis who shows in [22] that the general minimum fill-in problem is NP-hard. Lemmas 4.7 and 4.8 are from his article and are quoted without proofs.

Definition: A bipartite graph $G = (P, Q, E)$ is a *chain graph* if the neighborhoods of the vertices in P form a chain; i.e., there is a bijection $\pi : \{1, ..., |P|\} \leftrightarrow P$ such that $adj_G(\pi(1)) \supseteq adj_G(\pi(2)) \supseteq ... \supseteq adj_G(\pi(|P|))$. Then the neighborhoods of the vertices in Q form also a chain.

Definition: Let *G=(P,Q,E)* be a bipartite graph. We define *C(G)* to be the graph (V, E'), where $V = P \cup Q$ and $E' = E \cup \{(u,v)|u,v \in P\} \cup \{(u,v)|u,v \in Q\}$. Thus P and Q are cliques in *C(G)*.

Lemma 4.7 *Let G be a bipartite graph. C(G) is chordal if and only if G is a chain graph.*

Lemma 4.8 *It is NP-hard to find the minimum number of edges whose addition to a bipartite graph G=(P,Q,E) gives a chain graph.*

These two lemmas show that a restriction of the general minimum fill-in problem is NP-hard, implying that the general problem is also NP-hard. We will see in the next lemma, that this restricted problem is indeed equivalent to Δ.

Lemma 4.9 *$C(G_S)$ is chordal if and only if we can eliminate the vertices in S without introducing any fill-in between S and $adj'(S)$.*

Proof: *(if)* This is easy to see. We can first eliminate the vertices of S in $C(G_S)$ without introducing any fill-in between S and $adj'(S)$. This will not introduce any fill-in in $C(G_S)$ since S and $adj'(S)$ are cliques. Then we can eliminate the vertices of $adj'(S)$ and since they induce a clique in $C(G_S)$, this can be done without introducing any fill-in in $C(G_S)$. Thus $C(G_S)$ has a perfect elimination ordering and must be chordal.
(only if) Since $C(G_S)$ is chordal, it has a perfect elimination ordering. We need to show that there exists a perfect elimination ordering of $C(G_S)$ which eliminates the vertices of S before the vertices of $adj'(S)$. We know by Lemma 4.7 that G_S is a chain graph. Let $s = |S|$ and let α be an ordering of S such that $adj_{G_S}(\alpha(1)) \supseteq ... \supseteq adj_{G_S}(\alpha(s))$. The neighborhood of $\alpha(s)$ in $C(G_S)$ is $adj_{C(G_S)}(\alpha(s))$ $= adj_{G_S}(\alpha(s)) \cup (S - \alpha(s))$. This is a clique in $C(G_S)$, since S and $adj'(S)$ are cliques, and each vertex in S has edges to all the vertices of $adj'(S)$ that are neighbors of $\alpha(s)$. Thus $\alpha(s)$ is simplicial in $C(G_S)$ and can be eliminated first without introducing fill-in. When $\alpha(s)$ is eliminated, we can find a new simplicial vertex in the same way, since the chain property is hereditary. Thus we can eliminate the vertices in S first without introducing any fill-in between S and $adj'(S)$. □

Figure 4.7: a) An s-node S and $adj'(S)$. b) G_S. c) $C(G_S)$.

Theorem 4.2 *Δ is NP-hard.*

Proof: The proof follows from Lemmas 4.7 - 4.9. □

4.3 Heuristics

In this section we discuss how we can solve the problem Δ described in the previous section. Since the problem is NP-hard, it is not likely that one will find a polynomial time algorithm that solves it optimally.

Already existing heuristics like Minimum Degree or Nested Dissection may be used in order to solve Δ. However, there are no guarantees for how well these will perform when used for solving Δ. It is easy to find examples which show that these heuristics may result in more fill-in than optimal. On the other hand, since all the s-nodes can be reordered simultaneously, it is possible to find an optimal solution in reasonable time by simply trying all possibilities, if the s-nodes are small enough. Also, both Minimum Degree and Nested Dissection try to reduce the number of fill-in edges globally. Although we can run these heuristics locally on a single s-node, they consider all four groups of fill-in, whereas only fill-in edges that belong to Group 4 need to be considered.

We now present a new heuristic for reordering the vertices in an s-node to reduce fill-in. The algorithm is called *MinimumFillFirst*, and is locally sensitive only to Group 4 of fill-in. The idea behind this heuristic is to find how much fill-in each vertex in S will produce between S and $adj'(S)$ when eliminated first, and to eliminate the one that produces the least number of such fill-in edges first. The following lemma shows the size of the fill-in produced by the vertex that is eliminated first.

Lemma 4.10 *Let $G = (V,E)$ be a graph ordered by a Nested Dissection ordering with minimal separators. Let S be an s-node, v be a vertex in S, and let G_S be as described in the previous section. If v is eliminated first of all vertices in S then the number of fill-in edges between S and $adj'(S)$ in G^* produced by the elimination of v is:*

$$\sum_{w \in adj_{G_S}(v)} |\{x : x \in S \wedge (x,w) \notin E\}|.$$

Proof: Since S is a clique in G^*, v has edges to all the other vertices in S. Therefore, for each vertex $x \in S$, if $(x, w) \notin E$ for a neighbor w of v in G_S ($w \in adj'(S)$ and $w \in adj_G(v)$), then there will be a fill-in edge $(x,w) \in E^*$. This situation is illustrated in Figure 4.8. □

We can now formally describe our heuristic by the following algorithm. First the number of fill-in edges for each vertex in S is found (*FindSum*). Then in each iteration a vertex with the least sum is chosen and numbered. This vertex is taken away and the sums of the other vertices are updated (*Update*).

Figure 4.8: The proof of lemma 4.10.

```
Algorithm MinimumFillFirst (S: S-node; adj'(S): Set of vertices);

begin
    FindSum;
    while there is more than one vertex in $S$ do
        find the vertex $v \in S$ with the least sum;
        number $v$;
        Update(v);
    end-while;
end;
```

Next we give the algorithm for finding the sums. This algorithm uses Lemma 4.10 to compute the number of fill-in edges each vertex produces when eliminated first.

```
Algorithm FindSum;

begin
    for each vertex $v \in S$ do
        $v$.sum = 0;
        for each vertex $w \in adj'(S)$ do
            if $(v, w) \in E$ then
                for each vertex $x \in S$ do
                    if $x \neq v$ and $(w, x) \notin E$ then
                        $v$.sum = $v$.sum + 1;
    end-for;
end;
```

The time complexity of this algorithm is $O(s^2h)$, where $s = |S|$ and h is the height of the elimination tree. This is because $|adj'(S)| \leq h$ since we only look at the neighbors of S on the path from S to the root.

We will now look at the algorithm *Update* which has a vertex v as input parameter. The vertex v is the vertex that is currently being numbered. We look at each neighbor w of v ($w \in adj'(S)$), and decide if the elimination of v results in any fill-in between w and vertices in S.

For each vertex $x \in S$, where $x \neq v$, there is either an edge $(w, x) \in E$ or there will be a fill-in edge $(w, x) \in E^*$ when v is eliminated. If $(w, x) \in E$ then the elimination of x would have produced as many fill-in edges from w to vertices in S as the elimination of v produces. After v is eliminated, the elimination of x will not result in these fill-in edges any more since they already exist. Therefore, the number of such fill-in edges must be subtracted from x.sum. We update the sums for all the vertices x in this way. We must also update x.sum for vertices x that have neighbors which are not neighbors of v. After v is eliminated, there will not be any fill-in edges between v and these neighbors of x.

Note that if a vertex x does not have edges to any of v's neighbors then x will have edges to these when v is eliminated. This does not change x.sum or the sum of any other vertex, since every vertex in S has edges to every neighbor in $adj'(S)$ of v after v's elimination. Thus there will not be any more fill-in between these neighbors and vertices in S.

```
Algorithm Update ($v$: Vertex);

begin
    for each vertex $w \in adj'(S)$ do
        if $(v, w) \in E$ then
            fill = 0;
            unmark all vertices in $S$;
            for each vertex $x \in S$ do
                if $(w, x) \notin E$ then
                    fill = fill + 1
                else
                    mark $x$;
            for each marked vertex $x$ do
                $x$.sum = $x$.sum - fill;
            $adj'(S)$ = $adj'(S)$ - $w$;
        end-if;
    for each vertex $x \in S$, where $(x \neq v)$ do
        for each vertex $w \in adj'(S)$ do
            if $(w, x) \in E$ and $(v, w) \notin E$ then
                $x$.sum = $x$.sum - 1;
    $S = S$ - $v$;
end;
```

It is easy to see that the time complexity of the algorithm *Update* is $O(sh)$. This gives a total time complexity of $O(s^2h)$ for the algorithm *MinimumFillFirst* since the loop is executed s times.

We will now discuss the time complexity of reordering the whole graph $G = (V, E)$. This complexity depends on whether the reordering is done sequentially or in parallel. The time complexity of finding $adj'(S)$ for each s-node must also be considered. We now give a recursive algorithm *FindAdj* that finds $adj'(S)$ for all the s-nodes in the separator tree, when called with the root s-node as parameter.

```
**Algorithm** *FindAdj* (*S:* S-node);

**begin**
    $adj'(S) = \{\}$;
    **if** $S$ is a leaf s-node **then**
        **for** each vertex $v \in S$ **do**
            **for** each vertex $x$ on the path to the root **do**
                **if** $(x, v) \in E$ **then**
                    mark $x$ by $S$;
                    $adj'(S) = adj'(S) \cup \{x\}$;
                **end-if;**
    **else**
        **for** each child $S_1$ of $S$ **do**
            *FindAdj($S_1$);*
        **for** each vertex $v \in S$ **do**
            **for** each vertex $x$ on the path to the root **do**
                **if** the last s-node that marked $x$ is a child of $S$ **then**
                    mark $x$ by $S$
                **else**
                    **if** $(x, v) \in E$ **then**
                        $adj'(S) = adj'(S) \cup \{x\}$;
                        mark $x$ by $S$;
                    **end-if;**
    **end-if;**
**end;**
```

The time complexity of the algorithm *FindAdj* is $\sum_{S_i} |S_i|h = O(nh)$, where $n = |V|$. If the algorithm is implemented in parallel, it is possible to achieve a time complexity of $O(h^2)$ by using one processor per s-node. We now change the algorithm *Reorder* so that it uses all the algorithms that we have given.

```
Algorithm Reorder (ST: Separator tree; G: Graph);

begin
    mark the root s-node in ST;
    FindAdj (the root s-node);
    while there are unmarked s-nodes in ST do
        pick an unmarked s-node S in ST;
        MinimumFillFirst(S,adj'(S));
        mark S;
    end-while;
end;
```

If the algorithm *Reorder* is implemented sequentially, then the time complexity is

$$O(nh) + \sum_{S_i} O(s_i^2 h)$$

where $s_i = |S_i|$ for all s-nodes S_i in ST. Since $s_i \leq h$ and $\sum_{S_i} s_i = n$, we have

$$\sum_{S_i} O(s_i^2 h) = \sum_{S_i} O(s_i h^2) = O(h^2) \sum_{S_i} O(s_i) = O(h^2 n).$$

The time complexity of the algorithm *Reorder* is then $O(hn) + O(h^2 n) = O(h^2 n)$. Note that this is a pessimistic worst case time complexity analysis, and we may expect the algorithm to perform better in practice. It will indeed be difficult to perform this bad, since all the s-nodes cannot be as large as $O(h)$ and simultaneously have $O(h)$ neighbors higher up in the elimination tree.

Note that the time complexity analysis that we have given for the algorithms in this chapter, assumes a space complexity of $O(n^2)$. We must have that much space in order to be able to check if an edge $(v, w) \in E$ in $O(1)$ time. If n is very large, it might be difficult to store $O(n^2)$ entries. It is possible to use other data structures for storing. As an example, each vertex can have a balanced binary tree as its list of neighbors. This way, testing if $(v, w) \in E$ takes $O(\log d)$ time, where d is the largest degree in G. Note also that we have concentrated on making the algorithms as simple and easy to understand as possible. By coding the algorithms in a different way, it might be possible to reduce the time complexity further.

We have shown in Section 2 that the s-nodes can be reordered in parallel. In a parallel algorithm the loop is executed in parallel. Thus the s-nodes are being reordered simultaneously. The complexity of doing this will be bounded by the complexity of reordering the largest s-node in the separator tree. If the s-nodes are small, we can solve the problem of reordering all the s-nodes optimally in reasonable time though this will mean trying every permutation of the vertices in every s-node.

Chapter 5

Low elimination trees

In this chapter, we give an algorithm for finding minimum height elimination trees for chordal graphs. This algorithm is conceptionally simple, and leads to a polynomial-time algorithm for finding minimum height elimination trees for a non-trivial subclass of chordal graphs. A brief discussion on how a variant of this polynomial-time algorithm can be used on general graphs in order to reduce the elimination tree height is also given.

The central terms in this chapter are cliques, separators and chordal graphs. Cliques and separators were introduced in Chapter 1. Since cliques are complete subgraphs, a clique can be identified by the vertices in it. In this chapter, we treat cliques as sets of vertices. Thus a clique C in a graph G is a set of vertices that induce a complete subgraph of G. The difference between a minimal separator and a minimal v-w separator was also mentioned in Chapter 1. Recall that each minimal separator is also a minimal v-w separator for some v and w. In Chapter 2, we gave a brief introduction to chordal graphs. We now give some more results on chordal graphs. Lemma 5.1 is from [3], and Lemma 5.2 is from [21].

Lemma 5.1 *G is a chordal graph if and only if every minimal v-w separator in G is a clique.*

Lemma 5.2 *For every clique C in a chordal graph, there exists a perfect elimination ordering that eliminates the vertices in C last.*

Let $G = (V, E)$ be a chordal graph. The *clique graph* of G is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{C_1, C_2, ..., C_m\}$ is the set of all the maximal cliques in G, and each edge $(C_i, C_j) \in \mathcal{E}$ is a subset of V. If the cliques C_i and C_j have vertices in common in G, then $(C_i, C_j) \in \mathcal{E}$, and $(C_i, C_j) = C_i \cap C_j$. The weight of an edge (C_i, C_j) in $\mathcal{G}$ is equal to $|C_i \cap C_j|$. A *clique tree* $\mathcal{T}$ for G is a maximum weight spanning tree for $\mathcal{G}$. The clique graph of G is unique whereas G may have several clique trees. An example is shown in Figure 5.1. The number of maximal cliques in a chordal graph G is less than or equal to the number of vertices in G ([3]). Thus $|\mathcal{V}| \leq |V|$. More detailed information about clique trees can be found in [1] and [10].

Figure 5.1: a) G. b) The clique graph of G. c) The clique tree of G which, in this case, is unique.

Lemma 5.3 and Corollary 5.1 are from [10], and Lemma 5.4 is from [1]; we quote them without proofs.

Lemma 5.3 *Let G be a chordal graph and let $\mathcal{T}$ be a clique tree of G. For every edge (C_i, C_j) in $\mathcal{T}$, $C_i \cap C_j$ is a minimal v-w separator in G, and for every minimal v-w separator S in G there is at least one edge (C_i, C_j) in $\mathcal{T}$ such that $C_i \cap C_j = S$.*

Corollary 5.1 *If a chordal graph G has m maximal cliques, then the number of minimal v-w separators in G is less than or equal to $m - 1$.*

Let G be a chordal graph with n vertices and m maximal cliques. Since $m \leq n$, by Corollary 5.1, the number of minimal v-w separators in G is less than or equal to $n - 1$.

Lemma 5.4 *Let $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ be a clique tree of a chordal graph G. For every pair of distinct cliques C_i and $C_j \in \mathcal{V}$, the intersection $C_i \cap C_j$ is contained in every maximal clique on the path connecting C_i and C_j in $\mathcal{T}$.*

Corollary 5.2 *Let $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ be a clique tree of a chordal graph G. For every pair of distinct cliques C_i and $C_j \in \mathcal{V}$, the intersection $C_i \cap C_j$ is contained in every edge on the path connecting C_i and C_j in $\mathcal{T}$.*

Proof: Each edge S in $\mathcal{T}$ is the intersection between the two maximal cliques that S is incident to. Thus, if every maximal clique on a path in $\mathcal{T}$ contains $C_i \cap C_j$, then every edge on the same path must also contain $C_i \cap C_j$. $\Box$

In the next section, we will use the results presented here to find minimum height elimination trees for chordal graphs.

5.1 An algorithm for chordal graphs

In this section, we show that we can find a minimum height elimination tree for a chordal graph G by using a clique tree of G. Our approach is as follows: Let G be a chordal graph. Solve the following two problems.

1. Find a clique tree $\mathcal{T}$ of G.
2. Using $\mathcal{T}$, find a minimum height elimination tree for G.

An algorithm for solving Problem 1 can be found in [12]. In this section, we show how Problem 2 can be solved and give an algorithm for this. From Theorem 3.1, we know that there always exists a Nested Dissection ordering with minimal separators, that results in a minimum height elimination tree for G. As we have seen in the previous section, the edges in a clique tree $\mathcal{T}$ of G are the minimal v-w separators in G. Since every minimal separator in G is also a minimal v-w separator for some v and w in G, $\mathcal{T}$ provides all the necessary information about every possible minimal separator in G. The idea behind our algorithm is to exploit this property of $\mathcal{T}$. A minimum height elimination tree for G can then be found by trying out every ordering of the minimal separators in G.

Before we give the algorithm, we show in the next lemma that a clique tree $\mathcal{T}$ of G provides information not only about the minimal separators in G, but also about the minimal separators in every induced subgraph of G.

Lemma 5.5 *Let G be any graph, and let H be an induced subgraph of G. For each minimal v-w separator S_H in H, there exists a minimal v-w separator S_G in G, where $S_H \subseteq S_G$.*

Proof: Let v and w be two vertices in H, and S_H be a minimal v-w separator in H. Since it is possible to separate v and w in H, it is also possible to separate them in G. We now show how to choose S_G such that S_G is a minimal v-w separator in G and $S_H \subseteq S_G$. Let $G' = G - S_H$. If v is connected to w in G', then let S be a minimal v-w separator in G'. Otherwise, let S be empty. We set $S_G = S_H \cup S$. It is easy to see that S_G separates v from w in G. We now show by contradiction that S_G is minimal. Assume that S_G is not minimal and let S_{min} be a subset of S_G that separates v from w in G. Then S_{min} can be partitioned into two disjoint subsets A and B such that at least one of the two following statements is true: 1) $A \subset S$, 2) $B \subset S_H$. But then A and B separate v and w in respectively H and G'. This contradicts the assumption that S and S_H are both minimal, and completes the proof. □

Let G be a chordal graph. By Lemmas 5.3 and 5.5, every minimal separator in an induced subgraph H of G is represented by an edge in $\mathcal{T}$. It is also easy to show that H cannot have several non-identical minimal separators that are represented by one single edge in $\mathcal{T}$. This is because each minimal v-w separator in G is a clique, and it is not possible to disconnect such a separator by removing some of its vertices. Therefore, it can never be the case that H has non-identical minimal separators that are subsets of one unique edge S in the clique tree $\mathcal{T}$. Thus, $\mathcal{T}$ provides all the information needed by each step of a Nested Dissection ordering with minimal separators.

The next theorem shows that we can find minimum height elimination trees for chordal graphs, and an algorithm follows from the proof of the theorem.

Theorem 5.1 *We can find a minimum height elimination tree for a chordal graph G, given a clique tree $\mathcal{T}$ of G.*

Proof: The proof is by induction on the number of maximal cliques. Our induction hypothesis is the following.

Induction hypothesis: We can find a minimum height elimination tree for G if $\mathcal{T}$ contains $\leq m$ maximal cliques.

The base case is when $m = 1$. Then the clique tree contains one maximal clique C which may be empty. Thus we cannot have branching, and the minimum elimination tree height for G is 0 if C is empty, and $|C| - 1$ otherwise.

Assume that the induction hypothesis is true for $m = k$, where $k \geq 1$, and we will now show that it is true for $m = k + 1$. Let $\mathcal{T}$ be a clique tree with $k+1$ maximal cliques for a chordal graph G. Let S be an edge in $\mathcal{T}$ that is a minimal separator in G. Since S is a set of vertices, the removal of S from $\mathcal{T}$ is not only the removal of an edge from $\mathcal{T}$, but also the removal of the vertices in S from every maximal clique and every edge in $\mathcal{T}$. Thus $\mathcal{T} - S$ can have several components. By the induction hypothesis, for each component $\mathcal{T}'$ in $\mathcal{T} - S$, we know how to find a minimum height elimination tree for the subgraph of G corresponding to $\mathcal{T}'$. We find an elimination tree T for G in the following way: S is the top separator in T, and for each $\mathcal{T}'$, S has a minimum height elimination tree of $\mathcal{T}'$ as a child. In order to find the height of T, we add $|S|$ to the height of the highest child of S. However, T is not necessarily a minimum height elimination tree for G. In order to find a minimum height elimination tree, we repeat the process described above for each edge S in $\mathcal{T}$ that is a minimal separator in G. $\Box$

An algorithm can now be designed directly from this proof. For simplicity, we give an algorithm that computes only the minimum elimination tree height, but the actual elimination tree can also be computed in the same way. Algorithm *MinTree* returns the minimum elimination tree height for a chordal graph G, given a clique tree $\mathcal{T}$ of G.

```
Algorithm MinTree ($\mathcal{T}$: Clique tree):integer;

begin
    if $\mathcal{T}$ is empty then
        $MinTree = 0$
    else if $\mathcal{T}$ has only one maximal clique $C$ then
        $MinTree = |C| - 1$
    else
        min = $\infty$;
        for each edge $S$ in $\mathcal{T}$ do
            max = 0;
            for each component $\mathcal{T}'$ in $\mathcal{T} - S$ do
                height = $MinTree(\mathcal{T}')$;
                if max < height then
                    max = height;
            end-for;
            if min > $|S|$+ max then
                min = $|S|$ + max;
        end-for;
        $MinTree$ = min;
    end-if;
end;
```

Note that in the proof of Theorem 5.1, we considered only the minimal separators of G. Since the edges in $\mathcal{T}$ represent all the minimal v-w separators in G, and every minimal separator is a minimal v-w separator for some v and w in G, there is more information in $\mathcal{T}$ than we need. As long as every minimal separator is considered, we see from Theorem 3.1 that a minimum height elimination tree is found. So the correct result is found although we consider the edges in $\mathcal{T}$ that are not minimal separators.

The time complexity of the algorithm *MinTree* is exponential in the number of maximal cliques in G. Thus, this algorithm is not practical for large graphs. However, *MinTree* is useful for theoretical reasons. It is simple and easy to understand and illustrates the idea of exploiting the properties of clique trees to find minimum elimination tree height for chordal graphs. The technique used to develop this algorithm will help us to design a polynomial-time algorithm for a subclass of chordal graphs. This polynomial-time algorithm is presented in the next section.

5.2 The class Ω

Our aim now is to find a class of graphs for which an algorithm similar to *MinTree* finds the minimum elimination tree height in polynomial time. Since *MinTree* is designed for chordal graphs, it is natural to consider a subclass of chordal graphs. We now give a formal definition of a subclass of chordal graphs which we call Ω. We will see from the results in this section that Ω fits our purposes.

Definition: A graph $G \in \Omega$ if and only if G is chordal and has at most two maximal cliques that contain simplicial vertices.

Figure 5.2: Examples of graphs in Ω.

As we can see from this definition, a graph $G \in \Omega$ is either a complete graph, or a chordal graph with exactly two maximal cliques that contain simplicial vertices. We consider the graphs in Ω, and we show that we can find minimum height elimination trees for these graphs by using clique trees. Our approach is the same as in the previous section. As we have seen, by trying out every ordering of the separators in a graph $G \in \Omega$, we can find a minimum height elimination tree for G. However, by exploiting some properties of the graphs in Ω and their clique trees, we can restrict the amount of work, so that we can find a minimum height elimination tree in polynomial time for the graphs in Ω. Examples of graphs in Ω are shown in Figure 5.2.

We now present some results about the graphs in Ω and their clique trees, which we will use to develop an algorithm similar to *MinTree*. First we define a special type of graph which will be used throughout the rest of this chapter.

Definition: A *trail* is a tree $T = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$, and $E = \{(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)\}$.

Lemma 5.6 *A graph $G \in \Omega$ if and only if every perfect elimination ordering on G results in an elimination tree that is either a trail or has exactly two leaves.*

Proof: Figure 5.3 shows examples of elimination trees are meant by Lemma 5.6.

Figure 5.3: Elimination trees for the graphs in Ω.

(if) Assume that every perfect elimination ordering on G results in an elimination tree that has at most two leaves. If G has more than two maximal cliques with simplicial vertices, then there are at least three non-adjacent simplicial vertices in G. Thus these non-adjacent simplicial vertices can be eliminated simultaneously without introducing any fill-in. This however, results in an elimination tree that has at least three leaves, which contradicts our assumption.
(only if) Assume that G has at most two maximal cliques that contain simplicial vertices. If a perfect elimination ordering on G results in an elimination tree with more than two leaves, then there must exist at least three vertices in G that can be eliminated simultaneously. This is only possible when G has at least three non-adjacent simplicial vertices. But then G must have at least three maximal cliques with simplicial vertices. This contradicts the assumption that $G \in \Omega$ and completes the proof. $\Box$

We continue by showing in Theorem 5.2 that a graph $G \in \Omega$ has a unique clique tree that is a trail. The following lemma will help us to prove Theorem 5.2.

Lemma 5.7 *Let G be a chordal graph, and let $\mathcal{T}$ be a clique tree of G. Then every leaf clique in $\mathcal{T}$ contains at least one vertex v, where v is a simplicial vertex in G.*

Proof: Let C be a leaf clique in $\mathcal{T}$, and let C' be the only neighbor of C in $\mathcal{T}$. Thus the edge $C \cap C'$ is the only edge incident to C in $\mathcal{T}$. Let $\mathcal{G}$ be the clique graph of G. For every neighbor C_i of C in $\mathcal{G}$, we have from Corollary 5.2 that $(C_i \cap C) \subseteq (C \cap C')$. Thus, the vertices in $(C - (C \cap C'))$ do not belong to any other maximal cliques than C, and are simplicial vertices in G. Since $C \neq C'$, there must be at least one vertex in $(C - (C \cap C'))$. $\Box$

Theorem 5.2 *Let G be a graph in Ω. Then G has a unique clique tree $\mathcal{T}$ that is a trail.*

Proof: Let $\mathcal{T}$ be a clique tree of G. By the definition of Ω and Lemma 5.7, $\mathcal{T}$ has at most two leaf cliques. If G has only one maximal clique, then this is the only leaf clique in $\mathcal{T}$ and $\mathcal{T}$ is therefore unique. If G has two or more maximal cliques, then any clique tree of G must have at least two leaf cliques. Since $G \in \Omega$, it follows that $\mathcal{T}$ has exactly two leaf cliques, and the leaf cliques are the two maximal cliques in G with simplicial vertices. Therefore $\mathcal{T}$ must be a trail. We now show by induction that $\mathcal{T}$ must be unique.

Induction hypothesis: For every graph $G \in \Omega$ with $\leq m$ maximal cliques, there exists a unique clique tree $\mathcal{T}$.

The base case where $m = 3$ is the first interesting case since all trees with two vertices are isomorphic. Let G have 3 maximal cliques C_1, C_2 and C_3, where C_1 and C_3 are the ones with simplicial cliques. Then in a clique tree $\mathcal{T}$, C_1 and C_3 must be leaf cliques and C_2 must have edges to C_1 and C_3. This is the only possible clique tree of G and thus $\mathcal{T}$ is unique.

Let the induction hypothesis be true for $m = k$, where $k \geq 3$, and we show that it is true for $m = k+1$. Let G be a graph in Ω with $k+1$ maximal cliques, and let C_1 and C_{k+1} be the two maximal cliques in G with simplicial vertices. As we have mentioned before, in every clique tree of G, C_1 and C_{k+1} must be leaf cliques. Let $\mathcal{T}$ be a clique tree of G; we show that $\mathcal{T}$ is unique. Let C_1, C_2 and C_3 be maximal cliques in $\mathcal{T}$ as shown in Figure 5.4 a). Let G' be an induced subgraph of G which we get by taking away the simplicial vertices in C_1, and let $\mathcal{T}'$ be the remaining clique tree when the same vertices are removed from $\mathcal{T}$. Then $\mathcal{T}'$ is a clique tree of G'. By Corollary 5.2, in $\mathcal{T}$, $C_1 \cap C_2$ must contain every vertex that C_1 has in common with other maximal cliques. Thus when the simplicial vertices in C_1 are removed, the remaining vertices of C_1 make a subset of C_2. Therefore, the remaining clique tree $\mathcal{T}'$ has k maximal cliques as shown in Figure 5.4 b). We know that if $G' \in \Omega$, then by the induction hypothesis, $\mathcal{T}'$ is unique. Therefore, we now show that $G' \in \Omega$. In order to show this, we need to show the following two statements:

1. C_2 has simplicial vertices in G'
2. No other maximal clique C_i, where $2 < i < k + 1$ has simplicial vertices in G'

In order to show Statement 1, we note that in $\mathcal{T}$, $C_1 \cap C_2$ must contain at least one vertex v that is contained in no other maximal clique than C_1 and C_2. Otherwise, if every vertex in $C_1 \cap C_2$ were contained in some other maximal clique, then by Corollary 5.2 $C_2 \cap C_3$ would contain $C_1 \cap C_2$. But since C_2 has no simplicial vertices in $\mathcal{T}$, this would mean that $C_2 \subseteq C_3$ and lead to a contradiction. Thus v becomes a simplicial vertex in C_2 in G'. We now show Statement 2. In order for a maximal clique C_i to have simplicial vertices in $\mathcal{T}'$, C_i must have vertices in common with C_1 in $\mathcal{T}$. Otherwise, the removal of the simplicial vertices in C_1 would not affect C_i. But since $C_1 \cap C_2$ contains $C_1 \cap C_i$, every vertex in $C_1 \cap C_i$ is also contained in C_2 and cannot be a simplicial vertex. Thus, $\mathcal{T}'$ is unique, and $\mathcal{T}$ must also be unique since C_1 and C_{k+1} are the only leaf cliques in $\mathcal{T}$. $\square$

Figure 5.4: $\mathcal{T}$ and $\mathcal{T}'$ in the proof of Theorem 5.2.

Note that one of the possible clique trees for a chordal graph G may be a trail although G does not belong to Ω. The example in Figure 5.5 shows this. Also, every clique tree of a chordal graph G may be a trail although G is not in Ω. This is shown in Figure 5.6.

Figure 5.5: a) G. b) An elimination tree produced by a perfect elimination ordering on G. c) One of the possible clique trees of G which is a trail.

Figure 5.6: A chordal graph G which is not in Ω, and all the clique trees of G.

Let G be a graph in Ω. Theorem 5.2 states that G has a unique clique tree $\mathcal{T}$ that is a trail. Any perfect elimination ordering on G must start by eliminating simplicial vertices. Since the leaf cliques in $\mathcal{T}$ are the only maximal cliques in G with simplicial vertices, a perfect elimination ordering can first eliminate the simplicial vertices in the leaf cliques. Let the maximal cliques in $\mathcal{T}$ be numbered from 1 to m consecutively beginning from a leaf clique. A perfect elimination ordering α on G can be found in the following way: Eliminate first the simplicial vertices in C_1. As we have seen in the proof of Theorem 5.2, when the simplicial vertices in C_1 are eliminated, C_2 becomes a leaf clique with simplicial vertices. We continue by eliminating the simplicial vertices in C_2. It is easy to see that whenever a leaf clique C_i is eliminated, C_{i+1} becomes a leaf clique with simplicial vertices in the remaining clique tree. Thus we can continue by eliminating the simplicial vertices in C_{i+1}. We repeat this process until the remaining clique tree is empty. This elimination ordering α results in an elimination tree that is a trail, and implies an ordering of the maximal cliques in G. In the rest of this chapter, we assume that the maximal cliques of G are numbered from 1 to m as described above.

Note that the argument above can be applied to both ends of the clique tree, thus any perfect elimination ordering must proceed from the ends of the clique tree to the interior. We now continue with some more results that will help us to show how $\mathcal{T}$ can be used to find a minimum height elimination tree for G. The number of maximal cliques in G will always be denoted by m, unless specified otherwise.

Lemma 5.8 *Let G be a graph in Ω. Then every minimal v-w separator in G divides G into exactly two components.*

Proof: The proof is by contradiction. Assume that there exists a minimal v-w separator S in G that divides G into at least three components. By Lemma 5.1, S is a clique, and by Lemma 5.2, there exists a perfect elimination ordering α that eliminates S last. Then in each of the three components, there must be simplicial vertices such that each component can be eliminated without introducing fill-in. This implies the existence of at least three maximal cliques in G with simplicial vertices, and thus contradicts the fact that $G \in \Omega$. $\Box$

Lemma 5.9 *Let G be a graph in Ω and let $\mathcal{T}$ be a clique tree of G. Then there do not exist two distinct edges S and S' in $\mathcal{T}$, such that $S' \subseteq S$.*

Proof: Assume that S and S' are two distinct edges in $\mathcal{T}$, such that $S' \subseteq S$. The removal of S implies the removal of S'. By Lemma 5.8, the removal of S divides G and therefore $\mathcal{T}$ into two components. Thus, the removal of S must also imply the removal of every maximal clique between S and S'. This means that, for every maximal clique C_i between S and S', $C_i \subseteq S$. Let C be a maximal clique in $\mathcal{T}$ as shown in Figure 5.7. Since $S \subseteq C$, for every maximal clique C_i between S and S', $C_i \subseteq C$. Then for each C_i, either $C_i = C$ or C_i cannot be a maximal clique. But this leads to a contradiction since $\mathcal{T}$ contains only distinct maximal cliques of G. $\Box$

Figure 5.7: The proof of Lemma 5.9.

The following corollary follows directly from Lemma 5.9.

Corollary 5.3 *Let G be a graph in Ω, and let $\mathcal{T}$ be the clique tree of G. Then every separator represented by the edges in $\mathcal{T}$, is a minimal separator in G.*

To be able to use Nested Dissection with minimal separators to find a minimum height elimination tree, we must find the minimal separators in G. We know from Lemma 5.3 that a clique tree for any chordal graph contains information about every minimal v-w separator in the graph. We have mentioned before that every minimal separator is also a minimal v-w separator for some v and w in G. Thus, clique trees in general, give us more separators than we want, since we are only interested in the minimal separators. For the graphs in Ω however, we see by Corollary 5.3 that the separators provided by the clique trees, are exactly the minimal separators that we need.

Definition: Let G be a graph in Ω, and $\mathcal{T}$ the clique tree of G. For all i, j, where $i \leq j \leq m$, we define H_{ij} to be the subgraph of G induced by the vertices in the maximal cliques $C_i, C_{i+1}, ..., C_j$, excluding the vertices that these cliques have in common with other maximal cliques. In other words, H_{ij} is the graph induced by the vertices in $\{v \mid v \in C_k$ for $i \leq k \leq j$ and $v \notin C_k$ for $k < i$ or $k > j\}$.

Lemma 5.10 *Let G be a graph in Ω, and $\mathcal{T}$ the clique tree of G. For all i, j, where $i \leq j \leq m$, $H_{ij} \in \Omega$.*

Proof: It is easy to see that H_{ij} is chordal, since every induced subgraph of a chordal graph is also chordal. The rest of the proof is by contradiction. Assume that there exist i and j, where $i \leq j \leq m$ such that $H_{ij} \notin \Omega$. Then H_{ij} must have at least three maximal cliques with simplicial vertices. Thus H_{ij} has a perfect elimination ordering α that results in an elimination tree with at least three leaves. Then a perfect elimination ordering for G can be found in the following way. Eliminate the vertices in $C_1, ..., C_{i-1}$ beginning with the simplicial vertices in C_1 as described in the discussion above. Eliminate the vertices in $C_m, C_{m-1}, ..., C_{j+1}$ starting from the simplicial vertices in C_m and continuing backward. Eliminate the vertices in H_{ij} as locally ordered by α. This gives an elimination tree that has at least three leaves, and implies the existence of at least three maximal cliques in G with simplicial vertices. Since G then cannot be in Ω, we get the desired contradiction and the proof is complete. $\Box$

By Lemma 5.10, the results that we have shown on G, where G is a graph in Ω, are also true for H_{ij}. This makes it easier to design a recursive algorithm, since we know that the subgraphs that we work on in each step have the same properties as the original graph G. We have also mentioned in the previous section, that every minimal separator that needs to be considered in each step is represented in the original clique tree $\mathcal{T}$ of G. With this background we are now ready to design an algorithm for finding minimum elimination tree height for graphs in Ω.

As we will see in the proof of the following theorem, an elimination tree of minimum height for a graph $G \in \Omega$ can be found without having to try every permutation of the vertices in G. An algorithm for finding a minimum height elimination tree follows directly from this proof.

Theorem 5.3 *Let G be a graph in Ω, and let $\mathcal{T}$ be the clique tree of G. We can find a minimum height elimination tree for G by using only the information provided by $\mathcal{T}$.*

Proof: The proof is by induction, and our induction hypothesis is as follows.

Induction hypothesis: We know how to find an elimination tree of minimum height for H_{ij}, where $(j - i) \leq k$.

We show this by induction on k. The base case is when $k = 0$. This means that H_{ij} is either empty or a complete graph (H_{11} and H_{mm} are complete graphs, whereas H_{ii} for $1 < i < m$ is empty). If H_{ij} is empty then the elimination tree height is 0. Otherwise, the elimination tree height is one less than the number of the vertices in H_{ij}, and the elimination tree is a trail that consists of the vertices in H_{ij}.

We assume now that the induction hypothesis is true, and show how this enables us to find a minimum height elimination tree for H_{ij}, where $(j-i) = k+1$. Let $\mathcal{T}'$ be the subtree of $\mathcal{T}$ that corresponds to H_{ij}. We choose an edge in H_{ij} to be the top separator in the elimination tree. Let $S = C_p \cap C_q$ be this edge, where C_p and C_q are two consecutive maximal cliques in $\mathcal{T}'$. By Lemma 5.8, the removal of S divides H_{ij} into two components H_{ip} and H_{qj}. By the induction assumption, since $(p-i) \leq k$ and $(j - q) \leq k$, we can find minimum height elimination trees for H_{ip} and H_{qj}. Let $T_{min}(H_{ij})$ denote a minimum height elimination tree for H_{ij}. Further, let $T_S(H_{ij})$ be the elimination tree of H_{ij} with S as the top separator, and $T_{min}(H_{ip})$ and $T_{min}(H_{qj})$ as the two subtrees that hang from S. This is illustrated in Figure 5.8. Then $Height(T_S(H_{ij})) = |S| +$ max $(Height(T_{min}(H_{ip})), Height(T_{min}(H_{qj})))$. In order to find a minimum height elimination tree for H_{ij}, we find $T_S(H_{ij})$ for every edge S in H_{ij}, as described above. Then we choose the elimination tree that has the minimum height. This way, every minimal separator in H_{ij} is tried as the top separator, and the one that gives the minimum height, is chosen. Hence a minimum height elimination tree is found for H_{ij}, since there always exists a Nested Dissection ordering with minimal separators that results in a minimum height elimination tree. □

Figure 5.8: a) $\mathcal{T}'$. b) $T_S(H_{ij})$, where $A = T_{min}(H_{ip})$, and $B = T_{min}(H_{qj})$.

We now give an algorithm that computes the minimum elimination tree height for a graph in Ω as described in the proof of Theorem 5.3. For simplicity, we only compute the height of the elimination tree and not the tree itself. Let $C_1, ..., C_m$ be the maximal cliques in $\mathcal{T}$, and let $S_1, ..., S_{m-1}$ be the edges in $\mathcal{T}$, where $S_i = C_i \cap C_{i+1}$. The algorithm returns the height of a minimum height elimination tree for $H_{fromc,toc}$, where $fromc \leq toc \leq m$.

```
Algorithm Height ($\mathcal{T}$: Clique tree; fromc, toc: integer): integer;

begin
    $A = \{v \in C_j \mid j < fromc$ or $j > toc\}$;
    for i = fromc to toc - 1 do
        $C'_i = C_i - A$;
        $S'_i = S_i - A$;
    end-for;
    $C'_{toc} = C_{toc} - A$;
    if $fromc = toc$ then
        $Height = |C'_{fromc}| - 1$
    else
        opt = $\infty$;
        for i = fromc to toc - 1 do
            min = $|S'_i| + max\ (Height(fromc, i), Height(i+1, toc))$;
            if min < opt then
                opt = min;
        end-for;
    end-if;
    Height = opt;
end;
```

Theorem 5.4 *Let G be a graph in Ω, and let $\mathcal{T}$ be the clique tree of G. Then $Height(\mathcal{T}, 1, m)$ is the minimum elimination tree height for G.*

Proof: Follows directly from the proof of Theorem 5.3 and Algorithm *Height*.

Although the algorithm *Height* produces the desired result, it is not efficient. There are parts that are redundant and many subresults are computed several times. In order to achieve polynomial time complexity, we now present another algorithm that solves the same problem by using dynamic programming. We use a two dimensional array *Opt* [1..m,1..m] to store the optimal elimination tree heights of the subgraphs of G. The number *Opt* $[i, j]$ contains the height of a lowest elimination tree for H_{ij}.

Algorithm *DynHeight* ($\mathcal{T}$: Clique tree; *Opt:* Array of integers);

```
begin
    for i = 1 to $m$ do
        for j = 1 to $m$ do
            Opt [i,j] = $\infty$;
    for i = $m$ downto 1 do
        for j = i to $m$ do
            $A = \{v \in C_k | k < i$ or $k > j\}$;
            for k = i to j - 1 do
                $C'_k = C_k - A$;
                $S'_k = S_k - A$;
            end-for;
            $C'_j = C_j - A$;
            if i = j then
                if $C'_i$ is empty then
                    Opt [i,j] = 0
                else
                    Opt [i,j] = $|C'_i| - 1$;
            else
                for k = i to j - 1 do
                    min = $|S'_k|$ + max (Opt [i,k] , Opt [k+1,j]);
                    if min < Opt [i,j] then
                        Opt [i,j] = min;
                end-for;
        end-for;
    end;
```

As we can see in the algorithm, when we compute *Opt* $[i, j]$ we need to know *Opt* $[i, k]$ and *Opt* $[k{+}1, j]$ for $i \leq k < j$. We compute and use only the upper triangular part of the matrix *Opt*, since we only consider H_{ij} where $i \leq j$. Thus before computing *Opt* $[i, j]$, we must have computed rows $i+1, i+2, ..., m$, and elements $i, i+1, ..., j-1$ in row i. This explains the structure of the loops in the algorithm. It is then easy to see that the entry in *Opt* $[1, m]$ computed by $DynHeight(\mathcal{T}, Opt)$ is the same as the result returned by $Height(\mathcal{T}, 1, m)$

In order to find the time complexity of *DynHeight*, we note the following: computing A and updating the current maximal cliques can be done in $O(n)$ steps although we have chosen a less efficient way of doing this in our algorithm in order to make it easier to understand.

Thus the total amount of work needed to be done is:

$$O(m^2)+\sum_{i=1}^{m}\sum_{j=i}^{m}(O(n)+O(j-i)) = O(m^2)+O(nm^2)+O(m^3) = O(nm^2).$$

This gives us a time complexity of $O(nm^2)$. Note that since $n \geq m$ in a chordal graph, this time complexity is better than $O(n^3)$.

As with our previous algorithms in this thesis, when designing *DynHeight* we have concentrated on simplicity and tried to make it easy to understand. Our main goal was to achieve polynomial time complexity and not designing an algorithm as efficient as possible. Therefore, it might be possible to reduce the time bound on *DynHeight* by implementing it in a different way than we have done here. The algorithm may also be implemented in parallel to get a better time complexity.

Note also that we can find the actual elimination tree by slightly modifying this algorithm. This can be done by using another two dimensional array *Root* [1..m,1..m] to store the s-nodes in a separator tree for G corresponding to a minimum height elimination tree. The root separator in the separator tree for H_{ij} is stored in *Root* $[i, j]$. This way, the separator tree corresponding to a minimum height elimination tree can be found. We have to initialize the array *Root* to be empty in the beginning of the algorithm. The updating of the array *Root* which is

```
if min < Opt [i,j] then
    Opt [i,j] = min;
```

must be changed to

```
if min < Opt [i,j] then
    Opt [i,j] = min;
    Root [i,j] = $S'_k$;
end-if;
```

With this modification, the algorithm is able to provide the information needed to construct a minimum height elimination tree for G.

5.3 General graphs

In this section, we give a brief discussion on how we might use the Algorithm *DynHeight*, when the problem is to find a low elimination tree for a general graph G. In some cases, *DynHeight* can potentially give low elimination trees when used locally on subgraphs or globally on G or G^*. Some examples of how this can be done, are as follows:

1. Let G be a graph and let α be an elimination ordering on G. If $G^*_\alpha \in \Omega$, then we can use Algorithm *DynHeight* on the clique tree of G^*_α. This might result in a low elimination tree for G.

2. Let G be a graph and let $H \in \Omega$ be an induced subgraph of G. We can apply *DynHeight* locally on the clique tree of H. An elimination ordering on G that orders the vertices in H locally as suggested by *DynHeight*, might result in a low elimination tree for G.

3. Let G be a graph and let α be an elimination ordering on G. If G^*_α has an induced subgraph H^*_α, then we can apply *DynHeight* locally on the clique tree of H^*_α. An elimination ordering β on G that orders the vertices in H locally as suggested by*DynHeight*, and the rest of G as suggested by α might result in a lower elimination tree for G.

Consider the graphs in Case 1. Since G^*_α is in Ω, *DynHeight* finds a minimum height elimination tree for G^*_α. However, this elimination tree is not necessarily a minimum height elimination tree for G. Parts of G that could be separated before, may become cliques in G^*_α by addition of fill-in edges. Thus the fill-in edges may destroy separators in G, and therefore make it impossible for *DynHeight* to find the minimum elimination tree height for G. The example in Figure 5.9 illustrates this. Thus *DynHeight* does not guarantee finding a minimum height elimination tree for the graphs in Case 1, but it may be useful in finding low elimination trees.

For the graphs in Case 2, *DynHeight* finds a minimum height elimination tree for H. However, this ordering is local for H, and a global ordering for G that orders H locally by *DynHeight* does not necessarily result in a minimum height elimination tree. Also, it is not easy to know which subgraphs of G we should check.

Case 3 is a combination of Cases 1 and 2, and although it is not likely to find minimum height elimination trees this way, it may be possible to find low elimination trees.

Figure 5.9: a) G. b) $G^*_\alpha \in \Omega$, where α is a minimum fill-in ordering on G, and the resulting elimination tree which is a trail. c) A minimum height elimination tree ordering on G^*_α found by *DynHeight*. d) A minimum height elimination tree ordering on G.

As we can see, given a graph G, it is not easy to find every case on which *DynHeight* can be applied. However, *DynHeight* can be very useful to reduce the height of an elimination tree for G which is already given. The elimination tree can also indicate which subgraphs may be of interest with regard to reordering by *DynHeight*. We can start with an arbitrary elimination ordering on G, and see if we can reduce the height of the resulting elimination tree. If the elimination tree has induced subtrees that are trails, then it is likely that the height of these trails and possibly also the overall elimination tree height can be reduced. Let α be an elimination ordering on G, and let $T = T(G_\alpha)$ be the resulting elimination tree. If T has induced subgraphs that are trails, then it might be possible to locally reorder the vertices in these subgraphs to get a lower elimination tree for G. An example of this situation is shown in Figure 5.10. In this example, a minimum fill-in ordering α on G given, and the resulting elimination tree has three subtrees that are trails. When the subgraphs of G^*_α corresponding to these trails are examined, we find that one of these subgraphs both belongs to Ω and has more than one maximal clique. Then the height of this undergraph is reduced by *DynHeight*, and this results in a lower elimination tree.

Our approach is as follows: Given an elimination ordering α on G, do the following.

1. If $G^*_\alpha \in \Omega$ then apply *DynHeight* on the clique tree of G^*_α.
2. If $T(G_\alpha)$ has an induced subtree T' that is a trail, then check to see if the subgraph H^*_α of G^*_α induced by the vertices in T' is in Ω. If $H^*_\alpha \in \Omega$, apply *DynHeight* on on the clique tree of H^*_α.

With these suggestions on how the Algorithm *DynHeight* can be used on general graphs, we end this chapter.

Figure 5.10: a) G_α and the resulting elimination tree. b) G_β, where the marked vertices are locally ordered by *DynHeight*, and the resulting elimination tree.

Chapter 6

Conclusion

In this last chapter, we give a summary of the most central results achieved in this thesis and mention some open problems connected to our results.

6.1 Summary of results

Our own results were presented in Chapters 3, 4 and 5. We started in Chapter 3 by examining Nested Dissection orderings. We showed that for a subclass of graphs satisfying an n^ϵ-separator theorem, an elimination tree of height $O(n^\epsilon)$ can be found by using Nested Dissection. We also described a class of graphs on which Nested Dissection orderings produce elimination trees that are exponentially higher than optimal. The Nested Dissection orderings used here, choose separators so that none of the remaining components have more than $n/2$ vertices.

In Chapter 4, we worked on the problem of minimizing the fill-in size when a Nested Dissection ordering with minimal separators were given, without changing the elimination tree structure. We found out what portion of the fill-in can be reduced and showed that the problem decomposes into independent subproblems for each minimal separator. These subproblems were also shown to be NP-hard. We also presented a new heuristic for these subproblems and gave a time complexity analysis for this heuristic.

In Chapter 5, we concentrated on chordal graphs and finding minimum height elimination trees for chordal graphs by using clique trees. We gave a conceptionally simple algorithm for finding minimum elimination tree height for chordal graphs. The ideas behind this algorithm were used to develop a polynomial-time algorithm for a subclass of chordal graphs. We called this subclass Ω, and showed some properties on the graphs in Ω and their clique trees. We showed that the clique tree for a graph $G \in \Omega$ is a trail and that it is unique. These results were used to develop a polynomial-time algorithm that finds minimum height elimination trees for the graphs in Ω.

6.2 Open problems

We started Chapter 4 by giving a brief discussion on why we think low elimination trees might imply low fill-in. It has been conjectured that a minimum height elimination tree for a graph G does not produce more than linear amount of extra fill-in compared to minimum fill-in size for G. Manne shows this in [16] when G is a tree. However, this has not yet been shown for general graphs.

Algorithm *MinimumFillFirst* that was presented in Chapter 4 is a heuristic. We did not give any guarantee on how good results can be achieved by this algorithm. It might be interesting to compare this heuristic to established heuristics like Minimum Degree and Nested Dissection with regard to the amount of fill-in they produce.

In Chapter 5, we showed how minimum height elimination trees for chordal graphs can be found. Since chordal graphs have perfect elimination orderings, it is of interest to know how much fill-in can be introduced when chordal graphs are ordered to get minimum height elimination trees. The same question applies also to the graphs in Ω. It might be easier to show results about fill-in size on the graphs in Ω, since this is a more restricted class. We conjecture that the amount of fill-in introduced in these cases is linear compared to the number of already existing edges, but have not been able to give a proof for this.

It is also interesting to find subclasses Ω' of chordal graphs, where $\Omega \subseteq \Omega'$, for which there exist polynomial-time algorithms that compute minimum height elimination trees. We have not been able to find an extension of Ω for which the results of Section 5.2 are true. But other approaches different from ours, might lead to larger subclasses of chordal graphs for which minimum height elimination trees can be found in polynomial time.

Algorithm *MinTree* in Chapter 5, finds minimum elimination tree height for chordal graphs. A polynomial time algorithm for finding minimum height elimination trees for chordal graphs is not yet known. It is interesting to find such an algorithm, or to show that the problem of minimizing elimination tree height for chordal graphs is NP-hard. Since the number of minimal separators in a chordal graph G is bounded by the number of vertices in G, it is likely that this problem is easier than the problem of minimizing elimination tree height for general graphs.

Bibliography

[1] J. R. S. Blair and B. W. Peyton, *On finding minimum-diameter clique trees*, Tech. Rep. ORNL/TM-11850, Oak Ridge National Laboratory, Tennessee, 1991.

[2] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks, *Approximating treewidth, pathwidth, and minimum elimination tree height*, Tech. Rep. CSL-90-10, Xerox Palo Alto Research Center, 1991.

[3] G. A. Dirac, *On rigid circuit graphs*, Abh. Math. Sem. Univ. Hamburg, 25 (1961), pp. 71–76.

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

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman, 1979.

[6] A. George and J. W. H. Liu, *Computer Solutions of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[7] J. R. Gilbert, *Some nested dissection order is nearly optimal*, Information Processing Letters, 26 (1988), pp. 325–328.

[8] G. H. Golub and C. F. V. Loan, *Matrix Computations*, Johns Hopkins, 1989.

[9] H. Hafsteinsson, *Parallel Sparse Cholesky Factorization*, PhD thesis, Cornell University, 1988.

[10] C. W. Ho and R. C. T. Lee, *Counting clique trees and computing perfect elimination schemes in parallel*, Information Processing Letters, 31 (1989), pp. 61–68.

[11] J. A. G. Jess and H. G. M. Kees, *A data structure for parallel LU decomposition*, IEEE Transactions on Computers, C-31 (1982), pp. 231–239.

[12] J. G. Lewis, B. W. Peyton, and A. Pothen, *A fast algorithm for reordering sparse matrices for parallel factorization*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1146–1173.

[13] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[14] F. Manne, *Minimum height elimination trees for parallel Cholesky factorization*, Master's thesis, University of Bergen, Norway, 1989. In Norwegian.

[15] ——, *Reducing the height of an elimination tree through local reorderings*, Tech. Rep. 51, University of Bergen, Norway, 1991.

[16] ——, *An algorithm for computing a minimum height elimination tree for a tree*, Tech. Rep. 59, University of Bergen, Norway, 1992.

[17] S. Parter, *The use of linear graphs in Gauss elimination*, SIAM Review, 3 (1961), pp. 119–130.

[18] D. J. Rose, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, Graph Theory and Computing, (1972).

[19] D. J. Rose, R. E. Tarjan, and G. S. Leuker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Computing, 5 (1976), pp. 266–283.

[20] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Transactions on Mathematical Software, 8 (1982), pp. 256–276.

[21] R. E. Tarjan and M. Yannakakis, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, SIAM J. Computing, 8 (1984), pp. 566–579.

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