Partitioning and Ordering Graphs for Sparse Matrix Computations

Pinar Heggernes

Dr. Scient. Thesis

1996

Department of Informatics
University of Bergen
Norway

Acknowledgements

I am grateful to my advisor, Professor Bengt Aspvall, for all guidance, help, and encouragement he has given me. I have appreciated his advice, not only on the work with my thesis, but many times also on other questions. His commitment and concern have been a source of inspiration, and his ability to comfort and guide through difficult times of work has been invaluable. I thank the Research Council of Norway for financing the work on this thesis through grant 100622/410. Special thanks to Professors Torleiv Kløve and Tor Helleseth for encouraging me to to start on doctoral studies, and for their practical help in the beginning.

I would like to thank the people whom I have had the pleasure of working with on parts of this thesis in addition to my advisor. Jan Arne Telle has taught me many things, and our work together and our endless discussions have been most valuable. Pontus Matstoms has been a most inspiring person to work with, and I have enjoyed very much, and hope to continue, our cooperation in research and our friendship. Thanks to Jean Blair for interesting discussions while she was in Bergen, and for our joint research. I am indebted to Barry Peyton and John Gilbert for many valuable suggestions and comments, and useful discussions.

I want to thank all my friends and colleagues at the Department of Informatics, especially my fellow doctoral students and KaKeKlubben. Special thanks to Linda. I thank the Aspvall family for all their encouragement, friendship, and practical help. I am grateful to Aud and Sverre Heggernes for their affection and support, and for always being there to help in times of need. Gro deserves special thanks for willingly helping me out many times with child care in intensive work periods.

Without the love and encouragement of my parents, İnci and Mehmet Ataberk, I would never have come so far. Most of all, I thank them for giving me the freedom to choose my own ways since I was very young, and for always supporting my choices.

Finally, I want to thank the two most important people in my life, Frank and Hakan, for being my inspiration. Without their love and affection, life would be empty. I am grateful to Frank for his patience, support, words of comfort, and never-ending belief in me. To the two men of my life, I with love and gratitude dedicate this thesis.

Bergen, September 1996

PINAR HEGGERNES

Contents

Chapter 1

Introduction

Many problems that frequently arise in science and engineering require the solution of large and sparse systems of linear equations. Fields that often deal with such problems are, for example, oil reservoir simulation, meteorology, and computational fluid dynamics. We consider direct methods for solution of such systems.

In order to solve a large linear system efficiently, the structure of the system should be exploited. *Sparsity* is an important structure, and this thesis deals mainly with algorithms for exploiting the sparsity of linear systems in various ways in order to achieve efficient computation. The linear systems that concern this thesis appear in the form $Ax = b$, where A is a large and sparse matrix. A *sparse matrix* is a matrix where only a small portion of the elements are nonzero. Treating sparse matrices in the same way as dense matrices, often results in a waste of time and storage. Since many matrices related to real problems may be as sparse as with only 0.1% nonzeros, storing the whole matrix will require much more storage than necessary and, sometimes, available. Also when solving the system, only the nonzero elements of the matrix should be considered in order to save computation time.

In this thesis we work on two types of sparse matrices. *Symmetric and positive definite matrices*, and *rectangular matrices of full column rank*. Common for both these types of matrices, is that factorization is usually necessary for efficient solution of the related system. For symmetric and positive definite systems, the appropriate factorization method is Cholesky factorization, whereas for rectangular matrices of full column rank, we consider the method of QR factorization. In both cases, the matrices and the factorization process can be interpreted as graphs and graph algorithms. Problems related to efficient factorization can be formulated and solved as graph partitioning and ordering problems. This thesis deals with some of these problems.

Most of the important problems related to efficient factorization are NP-complete. These NP-complete problems are relevant to both sequential, and parallel factorization algorithms. It is therefore important to find heuristics that perform well generally, or find classes of graphs for which some of these problems become easy. These intractable problems might often have contradictory goals, which adds to the complexity of the whole problem of efficient factorization.

The problems mentioned above deal with partitioning or ordering graphs regarding some important criteria. In this thesis, we study several of these problems, and suggest heuristics and solution methods for general or restricted graph classes. Many graph partitioning problems that are not necessarily connected to sparse matrices, are also NP-complete. Some of these problems can be classified as *partitions into generalized dominating sets.* This thesis also includes a study of computational complexity of generalized dominating set problems. Some of the problems in this class have possible applications to sparse matrix computations.

In the rest of this chapter, we give a brief study in topics that are relevant for the thesis, and that are not explained in the coming chapters. The reader is assumed to be familiar with elementary standard graph and matrix terminology. More specialized terminology is explained whenever needed.

1.1 Theory of NP-completeness

Already during the 1950s and 60s, it was implicit from the work of several researchers that solvability in polynomial time is a desirable property of computational problems [102, 89, 16]. Edmonds [32, 33] gave an informal discussion of the class NP, and conjectured some problems not to be polynomially solvable. The foundations of the theory of NP-completeness were laid by Cook [19].

The class of NP-complete problems contains decision problems for which no polynomial time algorithm is known. The strength of this class is that, if one can find a polynomial time algorithm for one problem in this class, then the whole class of problems can be solved in polynomial time. This property provides a convenient way of proving that a given problem is NP-complete. We start by some definitions in order to give a formal explanation of NP-completeness. The results presented in this section are gathered from the book of Garey and Johnson [37], which is an excellent guide to the theory of NP-completeness.

A *decision problem* P, given an input x, is a problem to which the solution is either "yes" or "no". We let (P, x) denote this solution. For example, the *Hamiltonian cycle problem* asks, given a graph G as input, whether there is a cycle in G that visits each vertex exactly once. A *search problem* asks, not only whether the answer is "yes" or "no", but also, if the answer is "yes", for an example showing that the answer is "yes". The search problem for Hamiltonian cycle asks to find an actual cycle if the answer is "yes". If a decision problem asks about the existence of a structure with a certain *value*, then the corresponding problem of finding a structure with the "best" value is an *optimization problem.*

A *nondeterministic algorithm* corresponds to a nondeterministic Turing Machine.

Such an algorithm can *guess* an example of a "yes" answer, if the answer is "yes", and then *check* that the example actually corresponds to a "yes" answer. A nondeterministic algorithm *solves* a decision problem P, if it guesses a correct example and answers "yes" for all inputs x for which (P, x) is "yes", and never answers "yes" when the solution is "no". A nondeterministic algorithm that solves P operates in *polynomial time*, if the checking stage, which is deterministic, requires polynomial time for all "yes" inputs.

The class NP is the class of all decision problems that, under reasonable encoding schemes, can be solved by polynomial-time nondeterministic algorithms.

A decision problem P_1 is *polynomially reducible* to another decision problem P_2 if there exists a deterministic polynomial-time algorithm that converts each input x_1 for P_1 to an input x_2 for P_2, such that (P_1, x_1) is "yes" if and only if (P_2, x_2) is "yes".

A decision problem P_1 is defined to be *NP-complete* if $P_1 \in$ NP, and for all other decision problems $P_2 \in$ NP, P_2 is polynomially reducible to P_1.

From the definition of NP-completeness, it is clear that if any single NP-complete problem can be solved in polynomial time, then *all* problems in NP can be solved polynomially. If any problem in NP is provably intractable, then so are all NP-complete problems. Whether the NP-complete problems can be solved in polynomial time or not, is one of the most important theoretical open problems of our century. It is a general belief that they cannot be solved polynomially, although nobody has been able to prove this so far. Accordingly, in this thesis, we refer to NP-complete problems as intractable.

The following lemma, by Garey and Johnson [37], follows from the definitions and explanation above, and the transitivity of polynomial reduction.

Lemma 1.1 *If P_1 and P_2 belong to NP, P_2 is NP-complete, and P_2 is polynomially reducible to P_1, then P_1 is NP-complete.*

NP-completeness is only defined for decision problems. If a decision problem is NP-complete, then the corresponding search problem, or the optimization problem, is *NP-hard*.

1.1.1 NP-completeness proof techniques

From Lemma 1.1, we see that, in order to prove that a problem P_1 is NP-complete, we merely need to do the following:

1. Show that $P_1 \in$ NP, and

2. find a polynomial time reduction from a known NP-complete problem P_2 to P_1.

Although this looks simple, finding an appropriate NP-complete problem P_2, and finding a correct reduction, are often difficult tasks. The techniques used for proving NP-completeness vary widely, and it is not possible to illustrate them all. However, some general types of proofs occur frequently, and these techniques will be explained here briefly. We will see in the next chapter that some of these techniques are used to prove NP-completeness results for partition problems into generalized dominating sets.

Restriction An NP-completeness proof by restriction for a given problem P_1 consists of showing that P_1 contains a known NP-complete problem P_2 as a special case. Proof by restriction is the simplest, and probably the most frequently applicable proof technique.

Local replacement In proofs by local replacement, we take an input x_2 of the known NP-complete problem P_2, and we pick some aspect of x_2 to make up a collection of *basic units*. We obtain the corresponding input x_1 of the target problem P_1 by replacing each basic unit in x_2 with a different structure. Each replacement results in only a local modification of the whole structure.

Component design In proof by component design, the constructed input x_1 of the target problem P_1 can be viewed as a collection of components, each performing some function in terms of the given input x_2 of the known NP-complete problem P_2. These functions can be described as checking some properties, or making choices. The components are joined together such that there is a communication between the functions they perform. This type of proof design is often the most complicated.

1.1.2 Handling NP-complete problems

Although a certain problem is proven NP-complete, this problem can often be a generalization of a less elegant applied problem, and some of the details of the more restricted problem may be enough to make it polynomially solvable. Even though this might not be the case, for any general NP-complete problem, there still might be significant special cases that can be solved in polynomial time. In fact, the difficult inputs might be relatively rare, and might possess some properties that are easy to check, which may allow us to identify them beforehand. For problems that are generally NP-complete, and that have important applications in the general case, some large classes of inputs can be found for which the problem becomes easy. These classes can often be identified easily, and solved in polynomial time. It is therefore very important to analyze subproblems of NP-complete problems to find cases that can be solved in polynomial time. Analyzing subproblems may allow us to define a boundary between the easy and the intractable cases. Usually, there are subproblems whose NP-completeness is still an open problem. These subproblems will lie as a frontier between the cases for which intractability and polynomial-time solvability are proven.

When the problem is NP-complete for all interesting inputs, efforts can be made to find a usable algorithm for handling it. One approach in this direction seeks to obtain as much improvement over straightforward exhaustive search as possible. The idea is to generate partial solutions within a structured search format, and recognize partial solutions that cannot possibly be extended to general solutions, thereby eliminating parts of the search. Other approaches focus on alternative ways of organizing the search. *Dynamic programming* is such an approach.

For optimization problems that are NP-hard, polynomial time algorithms called *heuristics* can be designed. The goal of a heuristic is not to find an optimal solution, but rather to find a *good* solution. Heuristics are usually simple, and follow an obvious intuition to find good results. Sometimes performance guarantees can be given for heuristics, meaning that the solution found by the heuristic will never differ from optimal by more than a specified percentage.

1.2 Multiprocessor parallel computing

As computers become more and more powerful, the demand for even faster machines increases. Over the past decades dramatic rise in the computing speed was achieved. This trend will soon evidently come to an end, as today's sequential computers are already close to the limit implied by the speed of light in vacuum.

Parallelism represents the most feasible way to achieve faster computers. Parallel computers are developed from the idea that, if several instructions can be performed simultaneously, the computation time will decrease. Parallel computers are now commercially available throughout the world, and their declining cost makes them more and more attractive and practical.

Sparse matrix computations, and graph algorithms, are often suitable for parallelization. Some of the very important problems in matrix computations are related to finding efficient methods for parallel matrix factorization.

1.2.1 Parallel machine models

A multiprocessor parallel computer consists of several processors that work on different parts of a common problem. Each processor has either its own local memory, or the processors share a common memory space. For shared memory machines, all the processors read from, and write to, the common memory. Tie-breaking strategies are needed to handle the case of several processors attempting to access the same memory space simultaneously. Processors communicate through the common memory. Local memory machines provide separate local memory space for each processor. In order to exchange information, the processors have to communicate through communication channels. In most architectures, processors are interconnected in a network, and the network connections provide fast communication. Processors can also communicate with any other processor through a router, which is usually slower. The parallel computers that are most common, fall into one of the two following categories, first described by Flynn [35].

1. **SIMD** Single Instruction stream, Multiple Data stream.

2. **MIMD** Multiple Instruction stream, Multiple Data Stream.

SIMD computers have usually a large number of processors each performing the same set of instructions on different data. The processors are synchronized, hence in any time unit, a single operation is in the same state of execution on each processor. Processors can be set idle by using logical expressions. Communication between processors is easy on local memory SIMD machines, since the processors are synchronized.

MIMD computers have generally a small number of powerful processors each capable of performing different instructions. Each processor can run a different program on a part of the data. Communication is more complicated on local memory MIMD architectures, since sending and receiving has to occur simultaneously, and thus the processors have to be synchronized. More about parallel machine models, and parallel algorithms, can be found in several text books, like [2] and [88].

1.2.2 Parallel complexity

A parallel algorithm is considered efficient if it requires polylogarithmic parallel time, while the number of processors it uses is polynomial in the input size. The class of problems that can be solved by such efficient parallel algorithms is called the *class NC*, defined by Pippenger [84]. The polylogarithmic, $O(log^k n)$, time requirement rules out trivial parallelizations of polynomial sequential algorithms. The polynomial number of processors requirement is essential both from a practical and a theoretical point of view. A hyperpolynomial number of processors would be impractical, and would make the model too powerful with respect to Turing machine theory. For example, by making use of 2^n processors, all NP-complete problems could be solved by parallel algorithms in polynomial time.

All the problems in NC can be solved by sequential algorithms in polynomial time. Thus, NP-complete and NP-hard problems cannot be solved by efficient parallel algorithms, unless they can be solved polynomially by sequential algorithms. This shows once again the strength of the class of NP-complete problems, and how difficult these problems are. Even with very fast sequential, or parallel computers, these problems still remain practically intractable. Details about parallel and other complexity classes can be found in [82] and [14].

1.3 Sparse matrix computations

This thesis contains results that are related to finding a solution of $Ax = b$, where A is a large and sparse matrix. We consider following two types of large and sparse matrices:

1. The given matrix $A \in \mathbf{R}^{n\times n}$ is symmetric and positive definite.
2. The given matrix $A \in \mathbf{R}^{m\times n}$ is of full column rank, where $m > n$.

Using direct methods, in both cases, the matrix A, or a related matrix, has to be factorized before a solution can be computed. In the first case, the most common method to use is *Cholesky factorization.* In the latter case, the solution of the *full rank least squares problem* is required. Also the least squares problem is solved with help of matrix factorization. Several methods exist for the least squares problem; we consider solution by *QR factorization.*

1.3.1 Cholesky factorization

For symmetric, positive definite systems, there is always a unique solution. If $A \in \mathbf{R}^{n\times n}$ is symmetric and positive definite, then there exists a unique lower triangular matrix $L \in \mathbf{R}^{n\times n}$ with positive diagonal entries such that $A = LL^T$. Accordingly, solving $Ax = b$ can be done by solving the following two triangular systems: $Ly = b$ and $L^T x = y$.

There are several algorithmic ways to compute the Cholesky factor L. A method, which translates very easily to graph theory, is *outer product Cholesky.* This method is derived from the following partitioning:

$$A = \begin{pmatrix} \alpha & v^T \\ v & B \end{pmatrix} = \begin{pmatrix} \beta & 0^T \\ v/\beta & I_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 0^T \\ 0 & B - vv^T/\alpha \end{pmatrix} \begin{pmatrix} \beta & v^T/\beta \\ 0 & I_{n-1} \end{pmatrix}. \tag{1.1}$$

Here, $\beta = \sqrt{\alpha}$, and $\alpha > 0$ since A is positive definite. It can be shown that the $(n-1) \times (n-1)$ matrix $B - vv^T/\alpha$ is positive definite. Thus, if we have the Cholesky factorization $L_1 L_1^T = B - vv^T/\alpha$, then it follows that $A = LL^T$ with

$$L = \begin{pmatrix} \beta & 0^T \\ v/\beta & L_1 \end{pmatrix}.$$

The idea is then to perform repeatedly the reduction given in (1.1) on smaller submatrices.

1.3.2 Least squares solution via QR factorization

Let us now consider solution of $Ax = b$ for $A \in \mathbf{R}^{m\times n}$ with $m > n$. When there are more equations than unknowns, the system is called *overdetermined.* Usually an overdetermined system has no exact solution. This suggests minimizing the "distance" between Ax and b, $||Ax - b||_p$ for some suitable choice of p. The *least squares problem* asks for the solution of

$$\min_{x \in \mathbf{R}^n} ||Ax - b||_2.$$

If A has full column rank, then there is a unique least squares solution x_{LS}, and it solves the symmetric positive definite system

$$A^T A x_{\mathrm{LS}} = A^T b.$$

Thus x_{LS} can be found by performing a Cholesky factorization of $A^T A$. This method is called the method of *normal equations*. However, in Chapter 5 of this thesis, we will consider the solution of the least squares problem by QR factorization.

The QR factorization of A is the following decomposition

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}.$$

Here, $Q \in \mathbf{R}^{m \times m}$ is an orthogonal matrix, and $R \in \mathbf{R}^{n \times n}$ is upper triangular. If A has full column rank then R is unique. The QR factorization is usually performed by *Householder reflections* or by *Givens rotations*. We consider the Householder case. Householder QR factorization is explained in Chapter 5.

Suppose now that the QR factorization of A is computed such that

$$Q^T A = \begin{pmatrix} R \\ 0 \end{pmatrix}, \text{ and } Q^T b = \begin{pmatrix} c \\ d \end{pmatrix}. \tag{1.2}$$

Then, since the 2-norm is invariant under orthogonal transformation,

$$||Ax - b||_2^2 = ||Q^T Ax - Q^T b||_2^2 = ||Rx - c||_2^2 + ||d||_2^2.$$

Thus, when $\mathrm{rank}(A) = \mathrm{rank}(R) = n$, then x_{LS} is defined by the upper triangular system $Rx_{\mathrm{LS}} = c$. We therefore conclude that the full rank least squares problem can be readily solved once we have computed the QR factorization of A.

More details about Cholesky factorization, least squares, and QR factorization can be found in the book of Golub and Van Loan [52].

1.3.3 General sparse matrix factorization

In sparse matrix factorization, exploiting the sparsity is important. Therefore, the nonzero structures of the matrix and its triangular factor are usually determined before any numerical computation. Furthermore, the rows and the columns of the matrix A can be permuted to give better properties for the factorization. Different permutations of A give, for example, different number of nonzero elements in the resulting factor. How well A is suitable for parallel factorization is also decided by the permutation of A. We will shortly discuss these issues in more detail. Sparse matrix factorization is usually done in the following four steps:

1. **Ordering.**
    - *The symmetric, positive definite case.* Find a permutation matrix $P \in \mathbf{R}^{n \times n}$ such that the Cholesky factor L of PAP^T has better properties than the Cholesky factor of A. The system to be solved now becomes $(PAP^T)(Px) = (Pb)$.
    - *The rectangular, full rank case.* Find permutation matrices $P_1 \in \mathbf{R}^{m \times m}$, and $P_2 \in \mathbf{R}^{n \times n}$ such that the triangular QR factor R of P_1AP_2 has better properties than the corresponding factor of A. The system to be solved now becomes $(P_1AP_2)(P_2^T x) = (P_1 b)$. The matrix P_1 permutes the rows in A, whereas P_2 permutes the columns. In Chapter 5, we consider column permutations, thus $P_1 = I_m$.
2. **Symbolic factorization.** Determine the nonzero structure of the triangular factor, and decide a data structure to store the nonzero elements of A and of the triangular factor.
3. **Numerical factorization.** Insert the numerical values of the nonzero elements of A into the structure, and compute the numerical values of the nonzero elements in the triangular factor.
4. **Triangular solution.**
    - *The symmetric, positive definite case.* Solve $Ly = Pb$ and $L^T z = y$, and set $x = P^T z$.
    - *The rectangular, full rank case.* Solve $Rz = c'$, where c' is the upper part of $Q^T P_1 b$ corresponding to c in (1.2), and then set $x = P_2 z$.

The work related to sparse matrices in this thesis, deals with the first step, ordering. Our aim is to find good permutations of the given matrix in order to achieve more desirable properties for the factorization.

The size of the triangular factor of A is decided by the permutation of A. For QR factorization, row permutations of A have no effect on the number of nonzeros in R, whereas column permutations are essential. If element (i, j) of the factor is nonzero, whereas $a_{ij} = 0$, then a *fill* element has occurred. For sparse computations, it is desirable to keep the number of fill elements as small as possible. The number of nonzero elements in the factor is directly proportional to the storage requirements for it. This number is also closely related to the number of operations required in the factorization, especially for Cholesky factorization. For QR factorization, the situation is more complex, as will be explained in Chapter 5. Unfortunately, deciding a permutation that results in the minimum amount of fill is an NP-hard problem, as shown by Yannakakis [103].

Also for parallel computations, the permutation of A is important. A structure that provides information on the column dependencies of A is the *elimination tree*

[95, 71]. The height of the elimination tree represents a measure for the amount of work in parallel factorization. Therefore elimination trees of low height are desirable. Also the problem of finding a permutation that results in an elimination tree of minimum height is NP-hard, as shown by Pothen [85].

The ordering problem can easily be translated to graph theory, and graph theoretic tools can be very helpful to find good orderings.

1.3.4 The connection to graph theory

A symmetric, positive definite matrix $A \in \mathbf{R}^{n\times n}$ can be interpreted as a graph $G(A) = (V, E)$, with $V = \{v_1, v_2, ..., v_n\}$, and $E = \{(v_i, v_j) \mid a_{ij} \neq 0\}$. A symmetric permutation of A corresponds to a renumbering of the vertices of $G(A)$. The symbolic Cholesky factorization of A can be done by a graph algorithm called the *elimination game*. This algorithm will be explained in the coming chapters. The elimination game results in the filled graph $G(L) = (V, E^+)$ for the Cholesky factor L, where $E \subseteq E^+$. The edges $(v_i, v_j) \in (E^+ - E)$ represent the fill elements l_{ij}. The elimination tree of A can be defined directly from $G(L)$.

The graph interpretation of $A \in \mathbf{R}^{m\times n}$ with rank n, is by a *bipartite graph*. The graph of A is then $B(A) = (U, V, E)$, with $U = \{u_1, u_2, ..., u_m\}, V = \{v_1, v_2, ..., v_n\}$, and $E = \{(u_i, v_j) \mid a_{ij} \neq 0\}$. Row permutations of A corresponds to renumbering of the vertices in U, whereas column permutations correspond to renumbering of the vertices in V. Householder QR factorization translates also to a graph algorithm. The explanation of fill is a little more complicated in this case. In Chapter 5 these graph issues are explained in detail.

Since the minimum fill, and the minimum height elimination tree problems are both NP-hard, heuristics have been introduced to handle these problems. Two very famous heuristics are *minimum degree* and *nested dissection*. Both aim to reduce the size of the fill, and in addition, low height elimination trees are usually achieved by the latter.

1.4 Outline of the thesis

This thesis is based on six research papers. Instead of a portfolio of research papers with a common introduction, this thesis is organized more like a book, where the papers are presented as chapters. Each chapter includes a paper with some additional introduction, and sometimes an additional section, to bind the chapters together. Still, the chapters are independent, each with its own introduction and definitions, and thus can be read independently of each other. Since five papers are joint work with coauthors, definitions and terminology may differ slightly from chapter to chapter.

The papers presented in this thesis are the following, listed here in the order they appear in the thesis.

- **[58]** Pinar Heggernes and Jan Arne Telle, *Partitioning Graphs into Generalized Dominating Sets,* Proceedings of XV. International Conference of the Chilean Computer Society, 1995, pages 241-252.

- **[4]** Bengt Aspvall and Pinar Heggernes, *Finding Minimum Height Elimination Trees for Interval Graphs in Polynomial Time,* BIT 34, 1994, pages 484-509.

- **[5]** Bengt Aspvall, Pinar Heggernes, and Jan Arne Telle, *A Simple Cubic Algorithm for Computing Minimum Height Elimination Trees for Interval Graphs,* Reports in Informatics 103, 1995, University of Bergen, Norway.

- Pinar Heggernes, *Fill Minimization for Nested Dissection with Minimal Separators,* Based on some of the results presented by the author in [56]. (Preliminary version presented at the IMA Workshop on Direct Methods: Graph Theory Issues and Algorithms, Minnesota, October 1991).

- **[57]** Pinar Heggernes and Pontus Matstoms, *Finding Good Column Orderings for Sparse QR Factorization,* Technical Report LiTH-MAT-R-1996-20, 1996, Linköping University, Sweden. (To be presented at Second SIAM Conference on Sparse Matrices, Idaho, October 1996).

- **[8]** Jean R. S. Blair, Pinar Heggernes, and Jan Arne Telle, *Making an Arbitrary Filled Graph Minimal by Removing Fill Edges,* Proceedings of Fifth Scandinavian Workshop on Algorithm Theory, pages 173-184, Lecture Notes in Computer Science 1097, Springer Verlag, 1996.

Papers [4] and [5] are merged into a single paper in Chapter 3.

As this thesis is concerned with finding good graph orderings for sparse matrix factorization, we focus on the following three approaches.

- Reducing the height of the elimination tree.

- Reducing the number of fill edges.

- Reducing the number of operations required in the factorization.

We start with a more general study of NP-completeness of some graph partitioning problems in Chapter 2. Our aim is to find boundary values k for which partitioning the vertices of a graph into k sets becomes easy for some NP-complete graph partitioning problems. The problems that we study are called partitions into generalized dominating sets.

In Chapter 3, the minimum elimination tree height problem is studied. Since the problem is NP-hard for general graphs, we focus on finding classes of graphs for which the problem can be solved in polynomial time. Most of the previous research on this area has focussed on heuristics for general graphs [12]. Our intention has been to add to the few classes of graphs that have been shown to have polynomial algorithms. We use an important structure for chordal graphs, *clique trees*, and the method of dynamic programming.

Related to minimum height elimination trees, is the amount of extra fill introduced by minimum height orderings compared to minimum fill orderings. In Chapter 4, we show that it is possible to reduce the fill when an elimination tree of minimum height is given, and the structure of the elimination tree is to be preserved. Some nested dissection orderings include minimum elimination tree height orderings [74]. For such nested dissection orderings, we study the problem of minimizing the size of the resulting fill. This problem decomposes into local ordering problems for each chosen separator, and we show that the local problem is NP-hard.

Chapter 5 considers sparse QR factorization, and focuses on finding good column orderings in order to reduce the number of operations required by the factorization. Here, we use the method of designing heuristics for general graphs. We study equivalent reorderings, and show how these can reduce the operation count. The minimum degree algorithm, mentioned earlier, does not necessarily give a unique ordering. At each step of the algorithm, a vertex is chosen arbitrarily among vertices satisfying the minimum degree requirement. Resolving ties is very important for this algorithm. We propose tie-breaking strategies for the minimum degree heuristic that result in better operation count for QR factorization than standard minimum degree without tie-breaking.

In the last chapter of new results, the problem of reducing the number of fill edges for general graphs is visited. Our approach here is to find a minimal fill ordering, rather than a minimum fill ordering, which is NP-hard. There exist algorithms that find a minimal fill ordering of a given graph in polynomial time. However, minimal orderings are not necessarily close to minimum in general. Our algorithm considers any given filled graph, and picks away unnecessary edges from this graph to make it minimal. This way, a non-minimal ordering already resulting in low fill can be made minimal. A connection to equivalent reorderings is included in this version of the paper presented in Chapter 6.

Chapters 3, 4, and 6 are related to Cholesky factorization. Because of normal equations, there is a close relation between Cholesky and QR factors, which will be explained in Chapter 5. The thesis ends with a brief conclusion in Chapter 7.

Chapter 2

Partitioning Graphs into Generalized Dominating Sets

In Chapter 1, we emphasized the importance of finding bounds between NP-complete, and polynomially solvable subproblems of a generally NP-complete problem. In this chapter we study a class of graph partitioning problems described as partitions into generalized dominating sets. These problems concern deciding whether a (k, σ, ρ)-partitioning of a given graph exists, for a constant k and two sets σ and ρ. A (k, σ, ρ)-partition will be explained formally later, but informally, it means partitioning the vertices in the given graph into k disjoint sets which satisfy some properties described by the sets σ and ρ. For these problems, we give a systematic study of the values of k for which the problem is easy or NP-complete.

Load balancing problems related to parallel sparse matrix computations can often be translated to graph partition problems. Manne and Hafsteinsson show in [75] how the load balancing problem in sparse Cholesky factorization can be viewed as a weighted graph coloring problem. The following graph partitioning problem has applications to parallel sparse Cholesky factorization, and parallel sparse matrix dense vector multiplication: Partition the vertices in a given graph into k roughly equal parts, such that the number of edges connecting vertices in different parts is minimized [41, 61, 63, 86]. This problem is NP-hard. Some similar problems can be posed as (k, σ, ρ)-partition problems. For example, the $(k, \{0, 1, 2, ...\}, \{0\})$-partition problem asks whether the graph can be partitioned to k components with no edges in between. This problem can be solved in polynomial time. Another similar problem, the $(2, \{0, 1, 2, ...\}, \{1\})$-partition problem, asks whether the graph can be partitioned into 2 sets such that there is a perfect cut between the two sets. This problem is NP-complete.

This chapter presents the following research paper: [58] P. Heggernes and J. A. Telle, *Partitioning Graphs into Generalized Dominating Sets.*

2.1 Motivation and Overview

Several well-studied graph problems ask for a partition of vertices of a graph into subsets with a given property. For example, the Chromatic Number problem asks for a partition into the least number of independent sets. Even the fixed parameter version of this problem, vertex k-coloring, where we ask for the existence of a partition into k independent sets, is intractable (NP-complete) for any $k \geq 3$, while it is easy or trivial for smaller k, as shown by Garey and Johnson [37]. Similarly, Kratochvíl [62] shows that the K_k-cover problem asking for a partition into k perfect codes is NP-complete starting from $k \geq 4$, whereas the partition into perfect matchings problem is NP-complete already for $k \geq 2$, as shown by Schaefer [93].

These problems can all be defined as *Partitions Into Generalized Dominating Sets.* Generalized dominating sets, introduced by Telle in [99] and defined formally in the next section, are parameterized by two sets σ and ρ of nonnegative integers. Many well-studied vertex subset properties with applications in facility location and network communication can be expressed as (σ, ρ)-sets [17, 37, 55] (see Table 2.1). In this paper we present a systematic study of the complexity of partitioning vertices of a given graph G into k (σ, ρ)-sets, for varying values of the parameters k, σ, ρ.

Our focus has been on the vertex subset properties found in the literature, and on generalizations of these properties. In particular, for the vertex subset properties in Table 2.1, we find the tractability/intractability cutoff point k. Meaning that we give a polynomial-time algorithm deciding whether a graph has a partition into $k-1$ (σ, ρ)-sets, and an NP-completeness proof for the problem of deciding whether it has a partition into k (σ, ρ)-sets. See the table for these cutoff values. Several of our NP-completeness results are shown to hold for infinite classes of problems and/or for restricted graph classes, usually k-regular graphs. In addition to the three problems mentioned in the first paragraph, these partition problems have, to our knowledge, previously only been studied for dominating sets (domatic number) [37].

In the next section, we give some definitions and present some general facts on these partitioning problems. Section 2.3 contains the various NP-completeness reductions. In Section 2.4, we present the polynomial-time solvable problems, and we conclude with an open problem in Section 2.5.

2.2 Definitions and some facts

We first define the concepts used in our discussions. They concern mostly combinatorial graphs, *i.e.*, sets of vertices and edges together with the incidence relation between them.

A *graph* $G = (V, E)$ consists of a set of vertices V, and a set of edges $E \subseteq \{(u, v) \mid u, v \in V\}$. We consider only undirected graphs without self-loops. Given a vertex $v \in V$, we call the set of adjacent vertices its *neighborhood*, $N(v) = \{w \mid (v, w) \in E\}$.

σ	ρ	Standard terminology	cutoff
$\mathbb{N}$	$\{0,1\}$	Nearly Perfect set	∞ *
$\{0\}$	$\{0,1\}$	Strong Stable set or 2-Packing	4 *
$\{0\}$	$\{1\}$	Perfect Code or Efficient Dominating set	4 [62]
$\{0\}$	$\mathbb{N}$	Independent set	3 [37]
$\{0\}$	$\mathbb{P}$	Independent Dominating set	3 *
$\mathbb{N}$	$\mathbb{P}$	Dominating set	3 [37]
$\{0,1\}$	$\{0,1\}$	Total Nearly Perfect set	3 *
$\{0,1\}$	$\{1\}$	Weakly Perfect Dominating set	3 *
$\{1\}$	$\{1\}$	Total Perfect Dominating set	3 *
$\{1\}$	$\mathbb{N}$	Induced Perfect Matching	2 [93]
$\{1\}$	$\mathbb{P}$	Dominating Induced Matching	2 *
$\mathbb{P}$	$\mathbb{P}$	Total Dominating set	2 *
$\mathbb{N}$	$\{1\}$	Perfect Dominating set	2 *

Table 2.1: Some vertex subset properties expressed as (σ, ρ)-sets, with tractability/intractability cutoff value for the partitioning problem (∞ means tractable for all k values). We use an asterisk as reference if cutoff value proved in this paper.

The cardinality of $N(v)$ is called the *degree* of v. A graph, all of whose vertices have degree k, is called a *k-regular* graph. A *path* of length k in a graph $G = (V, E)$ is a sequence of distinct vertices, $v_0, v_1, \ldots v_k$, with $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$. If, in addition, $(v_0, v_k) \in E$, then we have a *cycle*. A graph is *connected* if there is a path between any two distinct vertices. In the *complete* graph K_k on k vertices there is an edge between every pair of vertices.

Let $\mathbb{N}= \{0, 1, ...\}$ and $\mathbb{P}= \{1, 2, ...\}$ denote the nonnegative and positive integers, respectively. A *partition* $V_1, V_2, ..., V_k$ of a set V satisfies $\bigcup_{i=1}^{k} V_i = V$, and $V_i \cap V_j = \emptyset$, $1 \leq i \neq j \leq k$.

Definition 2.1 *For $\sigma \subseteq \mathbb{N}$ and $\rho \subseteq \mathbb{N}$, a subset S of the vertices of a graph is a (σ, ρ)-set if $\forall v \in S : |N(v) \cap S| \in \sigma$ and $\forall v \notin S : |N(v) \cap S| \in \rho$.*

Definition 2.2 *A (k, σ, ρ)-partition of a graph G is a partition $V_1, V_2, ..., V_k$ of its vertices such that each V_i is a (σ, ρ)-set of G for $i = 1, 2, ..., k$.*

Partitioning a graph into (σ, ρ)-sets can also be viewed as a vertex labeling problem. A (k, σ, ρ)-partition of a graph G is a labeling of its vertices with k labels, such that for each vertex, the number of neighbors it has with its own label is in σ and the number of neighbors it has with each different label than its own is in ρ.

We now state some general facts on the existence of (k, σ, ρ)-partitions in a graph G. These facts follow from the definition of a (k, σ, ρ)-partition.

Fact 2.1 *A graph has a $(1, \sigma, \rho)$-partition if and only if the degree of every vertex is in σ.*

We will be assuming, quite naturally, that the parameter sets σ and ρ have polynomial-time membership tests. The problem of deciding whether a graph has a $(1, \sigma, \rho)$-partition is therefore easy for all values of σ and ρ.

Fact 2.2 *If p is a positive integer, then in any $(k, \sigma, \{p\})$-partition of a graph G, all partition classes V_i must have the same cardinality $|V_i| = |V(G)|/k$, for $i = 1, ..., k$.*

Fact 2.2 follows since for any two partition classes V_i and $V_j, i \neq j$, every vertex in V_i must have p neighbors in V_j, and vice-versa.

Fact 2.3 *For nonnegative integers s and r, if a graph G has a $(k, \{s\}, \{r\})$-partition then G is $(s + (k-1)r)$-regular.*

In the case of Fact 2.3, every vertex must have exactly s neighbors with the same label as itself, and it must have exactly r neighbors with each of the $k-1$ other labels.

Fact 2.4 *Given a graph $G = (V, E)$, if V and $\emptyset$ are both (σ, ρ)-sets, then G has a trivial (k, σ, ρ)-partitioning $V_1 = V$, and $V_i = \emptyset$ for $i = 2, ..., k$.*

An example of the situation described in Fact 2.4, can be obtained with $\sigma = \mathbb{N}$ and $0 \in \rho$.

Fact 2.5 *A graph G has a non-trivial $(k, \sigma, \{0\})$-partition if and only if G contains at least k components, and the degree of every vertex is in σ.*

The problem of deciding whether a graph has a $(k, \sigma, \{0\})$-partition is thus easy for all values of k and σ. The following lemma will be used in the proofs of some of the NP-completeness results of Section 2.3.

Lemma 2.1 *For any σ and ρ with $\max \sigma = s$ and $\max \rho = r$, where r and s are nonnegative integers, an $(s + (k-1)r)$-regular graph G has a (k, σ, ρ)-partition if and only if G has a $(k, \{s\}, \{r\})$-partition.*

Proof: If G has a $(k, \{s\}, \{r\})$-partition, then G has clearly a (k, σ, ρ)-partition since $s \in \sigma$, and $r \in \rho$.

If G has a (k, σ, ρ)-partition with $\max \sigma = s$ and $\max \rho = r$, then G has a vertex labeling with k labels such that each vertex has at most s neighbors with the same label as itself and at most r neighbors labeled with each of the $k-1$ other labels. Since G is $(s + (k-1)r)$-regular, each vertex v has exactly $s + (k-1)r$ neighbors, where s neighbors must be labeled with the same label as v, and r neighbors must be labeled with each of the other $k-1$ labels. This is equivalent to a $(k, \{s\}, \{r\})$-partition of G. ■

For all our results it will turn out that tractability of a (k, σ, ρ)-partition problem holds even if k is part of the input, while an intractability result always holds for k fixed. This is not surprising, as the number of partitions of n vertices into k classes is not a polynomial in n for any $k \geq 2$.

2.3 NP-completeness results

The values of σ and ρ used to describe the vertex subset properties listed in Table 1 are confined to $\{0\}, \{0,1\}, \{1\}, \mathbb{N}$ and $\mathbb{P}$. For $\rho = \{0\}$ we know from Fact 2.5 that the corresponding partition problems are easy. Table 2.2 gives a complete listing of all remaining combinations of σ and ρ among these values. We focus our study on the problems in this table. In this section, we give the NP-completeness proofs for the following (k, σ, ρ)-partition problems, which have not been proven before:

1. $(2, \mathbb{N}, \{1\})$, $(2, \mathbb{P}, \{1\})$, $(2, \{0,1\}, \mathbb{N})$ $(2, \{0,1\}, \mathbb{P})$, $(2, \mathbb{P}, \mathbb{P})$ and $(2, \{1\}, \mathbb{P})$.
2. $(k, \{1\}, \{1\})$, $(k, \{1\}, \{0,1\})$, $(k, \{0,1\}, \{1\})$, $(k, \{0,1\}, \{0,1\})$, $(k+1, \{0\}, \{0,1\})$ all on k-regular graphs (implying NP-completeness for general graphs), and $(k, \{0\}, \mathbb{P})$, all for $k \geq 3$.

These problems are marked with an asterisk in our tables. The entries in Table 2.2 are the cutoff values for the corresponding (k, σ, ρ)-partition problems. In order to show that these k-values are the cutoff points, polynomial-time Algorithms are given in Section 2.4 for partitioning into less than k (σ, ρ)-sets.

σ \ ρ	$\mathbb{N}$	$\mathbb{P}$	$\{1\}$	$\{0,1\}$
$\mathbb{N}$	∞^*	3	2^*	∞^*
$\mathbb{P}$	∞^*	2^*	2^*	∞^*
$\{1\}$	2	2^*	3^*	3^*
$\{0,1\}$	2^*	2^*	3^*	3^*
$\{0\}$	3	3^*	4	4^*

Table 2.2: A two dimensional cutoff table.

Before the NP-completeness proofs, we give definitions of three problems which do not appear in previous sections. These problems are all known to be NP-complete, and are used in the proofs of some of the theorems of this section.

Definition 2.3 (1-3SAT) *Given sets $S_1, ..., S_m$ each having three members, the One-In-Three Satisfiability problem asks whether there is a subset T of the members such that for each i, $|T \cap S_i| = 1$.*

Definition 2.4 (NAE-3SAT) *Given a collection C of clauses on a finite set X of variables where each clause contains three literals, the Not-All-Equal 3-Satisfiability problem asks whether there is a truth assignment for X that satisfies all the clauses in C such that no clause contains literals that are all true.*

The problems 1-3SAT and NAE-3SAT were shown to be NP-complete by Schaefer in [93].

Definition 2.5 (k-EC) *Given a graph G, the k-Edge-Coloring problem asks whether the edges of G can be colored with k colors such that no vertex is incident to two edges colored with the same color. If G is a k-regular graph, the question becomes whether each vertex is incident to k distinctly colored edges.*

This last problem was shown to be NP-complete for $k = 3$ by Holyer in [60], and for $k \geq 3$ by Leven and Galil in [65].

We now give the proofs of the NP-completeness results for the partition problems listed at the beginning of Section 2.3. Clearly, all of these problems are in NP, and as will be obvious, all of our reductions are polynomial-time reductions. We will not mention this explicitly in the proofs below.

Theorem 2.1 *The following (k, σ, ρ)-partition problems are NP-complete:*

(i) $(2, \mathbb{N}, \{1\})$ - *Perfect Matching Cut*

(ii) $(2, \mathbb{P}, \{1\})$ - *Partition Into 2 Perfect Dominating Sets Inducing No Isolated Vertices.*

Proof: (i) We use a reduction from 1-3SAT to $(2, \mathbb{N}, \{1\})$-partition. Let us consider an arbitrary instance of 1-3SAT, where $S_1, S_2, ..., S_m$ are the given sets, each containing three variables. We construct a graph $G = (V, E)$ such that G is $(2, \mathbb{N}, \{1\})$-partitionable if and only if the sets $S_1, S_2, ..., S_m$ are one-in-three satisfiable.

For each set $S_i = \{x, y, z\}$, we construct a subgraph G_i of G consisting of 10 vertices $x_i, y_i, z_i, a_i^1, a_i^2, ..., a_i^7$, as shown in Figure 2.1. Whenever two sets S_i and S_j share a variable x, there is an edge between x_i and x_j in G. G also contains a 4-cycle $[s, t_2, t_1, t_3, s]$ induced by the vertices s, t_1, t_2 and t_3 with $(s, a_i^1) \in E$, for $i = 1, ..., m$ (t_1, t_2, t_3 and s have no other neighbors). An example is shown in Figure 2.2.

Let G be $(2, \mathbb{N}, \{1\})$-partitionable. This is equivalent to a 2-labeling of the vertices of G such that each vertex has exactly one neighbor with the opposite label than itself. The vertices t_2 and t_3 must be labeled oppositely since they are the only neighbors of t_1. Equivalently, s and t_1 must also be labeled oppositely. Assume, without loss of generality, that t_1 is labeled with 1 and s with 0. Since s already has a neighbor (t_2 or t_3) with the opposite label, every other neighbor of s must be labeled with the same label as s. Thus a_i^1 is labeled with 0, for $i = 1, ..., m$. Consider

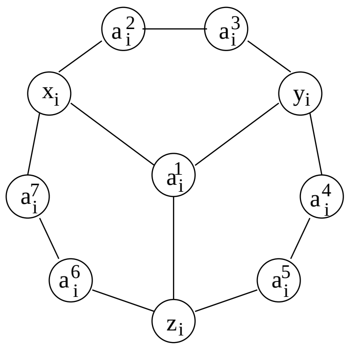


Figure 2.1: The subgraph G_i of G corresponding to $S_i = \{x, y, x\}$.

a subgraph G_i, where $S_i = \{x, y, z\}$ is the corresponding set. Apart from s, a_i^1 has neighbors x_i, y_i, z_i. Since both s and a_i^1 are labeled with 0, and a_i^1 must have exactly one neighbor with the opposite label, one of x_i, y_i and z_i must be labeled with 1, and the other two with 0. When these are labeled, the rest of the labeling of G_i is unique. This labeling is such that each of x_i, y_i and z_i has exactly one neighbor with the opposite label than itself within G_i. Therefore, any neighbor outside G_i of a vertex x_i must have the same label as x_i. The only such neighbors of x_i are the vertices x_j, where $x \in S_i \cap S_j$. If G has a valid labeling, then the desired set T for 1-3SAT is $T = \bigcup_{i=1}^{m}\{x \in S_i \mid x_i \text{ is labeled with } 1\}$. Now, each set S_i has exactly one member x such that the corresponding vertex x_i in G_i is labeled with 1. Also, for each other set S_j containing x, the corresponding vertex x_j must also be labeled with 1. Thus the sets $S_1, ..., S_m$ are one-in-three satisfiable.

Assume now that there exists a set $T \subset \bigcup_{i=1}^{m} S_i$, such that each set S_i has exactly one member in T. For each $x \in T$, and for each i such that $x \in S_i$, label the corresponding vertex x_i in G with label 1. For each $y \notin T$, and for each i such that $y \in S_i$, label the corresponding vertex y_i with 0. Label t_2 and t_3 oppositely, t_1 with 1, s with 0, and a_i^1 with 0, for $i = 1, ..., m$. Let us now consider a subgraph G_i corresponding to $S_i = \{x, y, z\}$. Assume that $z \in T$. Label a_i^2, a_i^3, a_i^5 and a_i^6 with label 1, and a_i^4 and a_i^7 with 0. This describes a valid labeling of all the vertices in G_i. Because of the symmetry of G_i, this labeling can easily be adapted for $x \in T$, or $y \in T$, instead of z (see Figure 2.2). Let us label all the subgraphs G_i as described. Since each subgraph has a valid labeling independent of the rest of G, and since external edges only go between vertices of same label, this describes a valid labeling of G. Thus G is $(2, \mathbb{N}, \{1\})$- partitionable.

(ii) Note that in G, every vertex has at least two neighbors. Therefore, G has a $(2, \mathbb{N}, \{1\})$-partitioning if and only if it has a $(2, \mathbb{P}, \{1\})$-partitioning. It is easy to see that the labeling described in **(i)** is indeed a $(2, \mathbb{P}, \{1\})$-partitioning for G. ∎

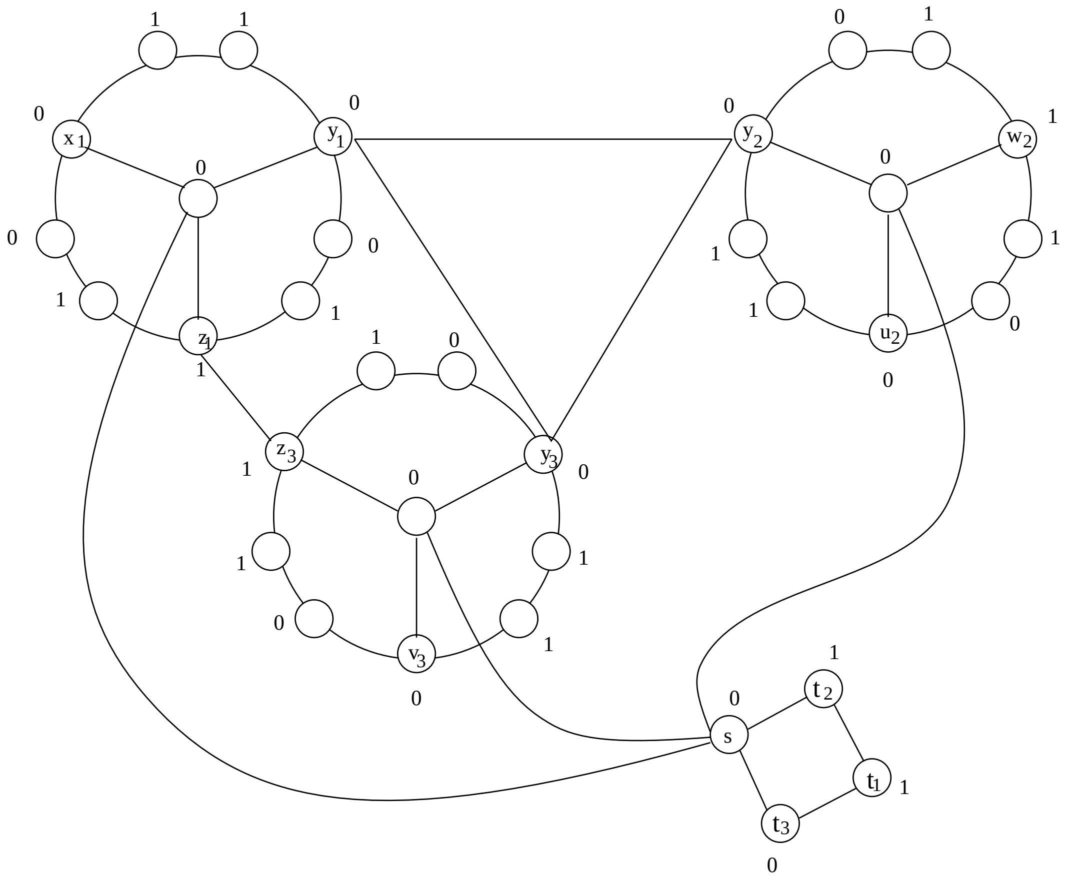


Figure 2.2: An example showing the constructed graph G, when the given sets are: $S_1 = \{x, y, z\}, S_2 = \{u, w, y\}$ and $S_3 = \{v, y, z\}$. The labeling corresponds to choosing $T = \{w, z\}$.

Theorem 2.2 *The following (k, σ, ρ)-partition problems are NP-complete:*

(i) $(2, \{0,1\}, \mathbb{N})$ - *Partition Into 2 Sets Inducing Isolated Vertices and Edges.*

(ii) $(2, \{0,1\}, \mathbb{P})$ - *Partition Into 2 Dominating Sets Inducing Isolated Vertices and Edges.*

Proof: (i) The reduction is from 1-3SAT to $(2, \{0,1\}, \mathbb{N})$-partition. Let $S_1, S_2, ..., S_m$ be the given sets in an arbitrary instance of 1-3SAT. We construct a graph $G = (V, E)$ such that G is $(2, \{0,1\}, \mathbb{N})$-partitionable if and only if the sets $S_1, S_2, ..., S_m$ are one-in-three satisfiable.

For each set $S_i = \{x, y, z\}$, there is a 4-clique K_i in G induced by the vertices x_i, y_i, z_i and a_i, as shown in Figure 2.3 a). For each variable x, there is an edge e_x in G. For each set S_i such that $x \in S_i$, the vertex x_i of K_i has edges to both endpoints of e_x, as shown in Figure 2.3 b). G also contains a 4-clique induced by the vertices s, t_1, t_2 and t_3 with $(s, a_i) \in E$, for $i = 1, ..., m$ (t_1, t_2, t_3 and s have no other neighbors). An example is shown in Figure 2.4.

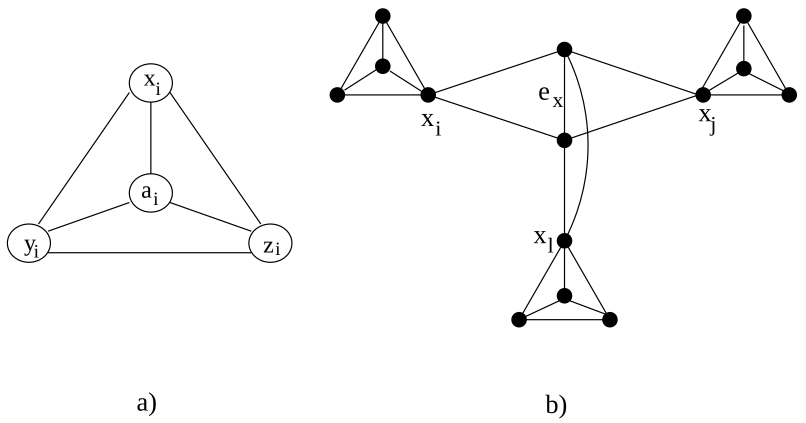


Figure 2.3: a) The clique K_i corresponding to the set $S_i = \{x, y, z\}$, and b) the subgraph concerning e_x when $x \in S_i \cap S_j \cap S_l$.

A $(2, \{0,1\}, \mathbb{N})$-partitioning of G is equivalent to a 2-labeling of the vertices such that each vertex has at most one neighbor with the same label as itself. Assume that G has such a labeling. Each 4-clique must be labeled such that two vertices are labeled with 0 and two are labeled with 1, otherwise some vertex in the clique would have two or more neighbors that are labeled the same as itself. Assume without loss of generality that s is labeled with 0. Then two of t_1, t_2 and t_3 are labeled with 1, and one with 0. Since s already has a neighbor with the same label as itself, every other neighbor of s must be labeled with the opposite label. Thus each a_i

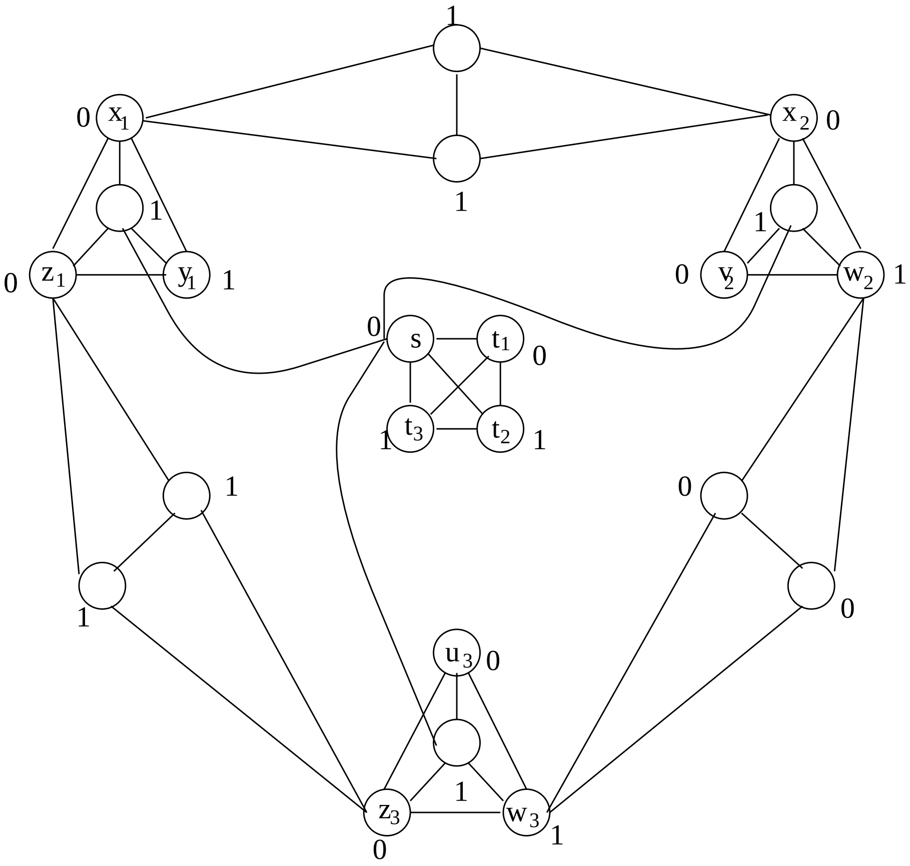


Figure 2.4: An example showing the constructed graph G, when the given sets are: $S_1 = \{x, y, z\}, S_2 = \{x, v, w\}$ and $S_3 = \{u, w, z\}$. The labeling corresponds to choosing $T = \{w, y\}$.

is labeled with 1, for $i = 1, 2, ..., m$. Let K_i be the 4-clique corresponding to the set $S_i = \{x, y, z\}$. Since K_i must have exactly two vertices with each of the labels, one of x_i, y_i and z_i must be labeled with 1, and two must be labeled with 0. Each vertex x_i has now one neighbor in K_i with the same label as x_i. Therefore, every other neighbor of x_i must be labeled with the opposite label. This implies that both endpoints of the edge e_x have the same label, and every other neighbor of these endpoints must be labeled with the same label as x_i. Now, the desired set T in 1-3SAT is $T = \bigcup_{i=1}^{m}\{x \in S_i \mid x_i \text{ is labeled with } 1\}$, and we can conclude that the sets $S_1, ..., S_m$ are one-in-three satisfiable.

Assume that the sets $S_1, ..., S_m$ are one-in-three satisfiable, and let T be the desired subset. Label a_i with 1, for $i = 1, 2, ..., m$, and label the rest of the vertices in each clique K_i with 1 if the corresponding variables belong to T, and with 0 otherwise. Since there is only one member in each set that can have its corresponding vertex labeled with 1, there are two vertices labeled with 1, and two with 0 in each K_i. Label s and t_1 with 0, and t_2 and t_3 with 1. For each edge e_x, label both endpoints with 0 if $x \in T$, and with 1 otherwise. Clearly, this is a labeling of G corresponding to a $(2, \{0, 1\}, \mathbb{N})$-partition.

(ii) Since each vertex in G has at least two neighbors, a $(2, \{0, 1\}, \mathbb{N})$-partition for G is equivalent to a $(2, \{0, 1\}, \mathbb{P})$-partition. The labeling described in **(i)** is indeed a $(2, \{0, 1\}, \mathbb{P})$-partitioning of G. ∎

Theorem 2.3 *The $(2, \mathbb{P}, \mathbb{P})$-partition problem (Partition Into Two Total Dominating Sets) is NP-complete.*

Proof: The reduction is from NAE-3SAT. Let X be the set of variables and C be the the set of clauses in an arbitrary instance of NAE-3SAT. We can assume that all literals appear in some clause, otherwise for each literal that does not appear, we can put it and its negation in an additional clause. This clause is then always true and contains at least one false literal. We construct a graph $G = (V, E)$ such that G has a $(2, \mathbb{P}, \mathbb{P})$-partition if and only if the variables in X can be assigned values true or false such that each clause in C has at least one literal that is true and at least one that is false.

For each variable $x \in X$, there are three vertices x, $\bar{x}$ and a_x in V, where $(x, \bar{x}) \notin E$, $(x, a_x) \in E$, and $(\bar{x}, a_x) \in E$. For each clause $c_i \in C$, there is a vertex $c_i \in V$, and c_i has edges to the vertices representing the literals in the clause c_i. This construction is shown in Figure 2.5.

A $(2, \mathbb{P}, \mathbb{P})$-partitioning of G is a 2-labeling of the vertices of G such that each vertex $v \in V$ has at least one neighbor with the same label as v, and at least one neighbor with the opposite label. Assume that G has such a labeling, and let the two labels be T and F. Since a_x has only two neighbors x and $\bar{x}$, the vertices x and $\bar{x}$ have opposite labels for each $x \in X$. Furthermore, each vertex c_i has at least one neighbor labeled with T, and at least one neighbor labeled with F. With $T =$

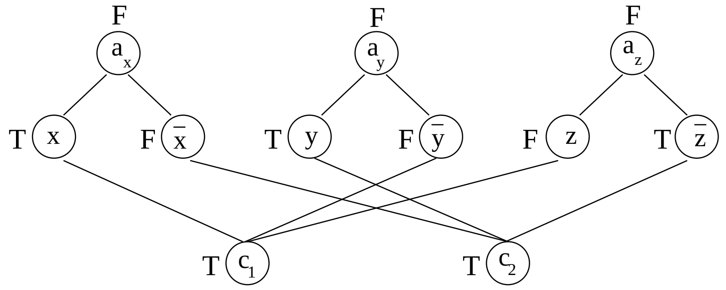


Figure 2.5: An example showing G when the given clauses are: $c_1 = (x, \bar{y}, z)$ and $c_2 = (\bar{x}, y, \bar{z})$.

true and $F =$ false, this describes a truth assignment for the variables in X that is consistent with the requirements of the NAE-3SAT problem. An example labeling is given in Figure 2.5.

Conversely, given that X has such a truth assignment, we will show how to achieve a 2-labeling of the vertices of G that corresponds to a $(2, \mathbb{P}, \mathbb{P})$-partitioning. Label each vertex a_x with F, for $x \in X$, and label each vertex c_i with T, for $c_i \in C$. Each vertex x and $\bar{x}$ is labeled with T if its corresponding literal is assigned true, and with F if its corresponding literal is assigned false. Clearly, this describes a valid $(2, \mathbb{P}, \mathbb{P})$-partitioning of G, and the proof is complete. ∎

Theorem 2.4 *The $(2, \{1\}, \mathbb{P})$-partition problem (Partition Into 2 Dominating Induced Matchings) is NP-complete on 3-regular graphs.*

Proof: The reduction is from the $(2, \{1\}, \mathbb{N})$-partition problem, which is also called *Two-Colorable Perfect Matching.* Schaefer shows in [93] that this problem is NP-complete on 3-regular graphs.

Let G be a $(2, \{1\}, \mathbb{N})$-partitionable 3-regular graph. Then the vertices of G can be labeled with two labels such that each vertex has exactly one neighbor labeled with the same label as itself. Since G is 3-regular, this also means that each vertex has exactly two neighbors labeled with the opposite label as itself. Thus G is also $(2, \{1\}, \mathbb{P})$-partitionable.

Conversely, a $(2, \{1\}, \mathbb{P})$-partition for any graph is clearly a $(2, \{1\}, \mathbb{N})$-partition for the same graph. Thus, a 3-regular graph G is $(2, \{1\}, \mathbb{N})$-partitionable if and only if G is $(2, \{1\}, \mathbb{P})$-partitionable, and we have shown that $(2, \{1\}, \mathbb{P})$-partition problem is NP-complete on 3-regular graphs. ∎

Theorem 2.5 *The $(k, \{1\}, \{1\})$-partition problem (Partition Into k Total Perfect Dominating Sets) is NP-complete on k-regular graphs for $k \geq 3$.*

Proof: The reduction is from k-EC on k-regular graphs for $k \geq 3$. Let $G = (V, E)$ be a k-regular graph. We construct a k-regular graph $G' = (V', E')$ such that G' has a $(k, \{1\}, \{1\})$-partitioning if and only if G has a k-edge-coloring.

For each vertex $v \in V$, there is a clique K_v in G' with k vertices, and $V' = \bigcup_{v \in V} K_v$. Each vertex of K_v has exactly one neighbor outside K_v corresponding to the edges incident to v in G. If $(u, v) \in E$, then in E', there is an edge between exactly one vertex of K_u and exactly one vertex of K_v. This construction is shown in Figure 2.6. Since G is k-regular, each clique K_v in G' has exactly k external edges incident to it, one incident to each of the k vertices in K_v. Thus G' is also k-regular. Any two cliques K_v and K_w in G' have at most one external edge connecting them.

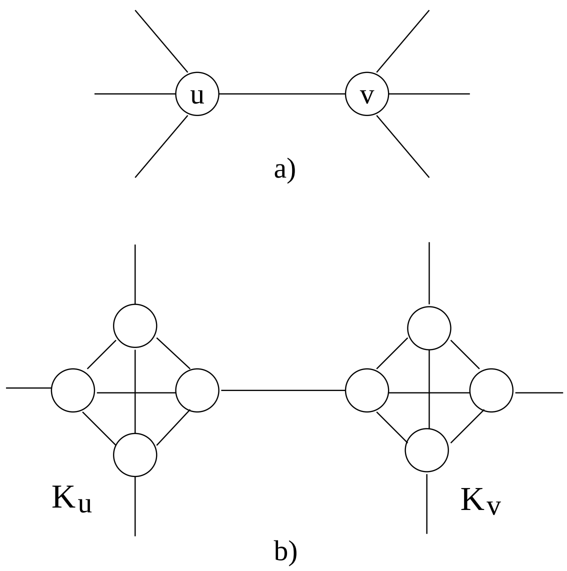


Figure 2.6: a) An edge in G, and b) the corresponding subgraph in G', where G is 4-regular.

A $(k, \{1\}, \{1\})$-partitioning of a graph is equivalent to a k-labeling of the vertices such that each vertex has exactly k neighbors each labeled distinctly with one of the k labels (one neighbor of its own label, and exactly $k-1$ other neighbors each labeled distinctly). Assume that G' has a k-labeling. The internal vertices of each clique K_v must be labeled with k different labels, otherwise some of the vertices in the clique would have at least two neighbors labeled with the same label. In addition, both endpoints of each external edge must be labeled with the same label, since each endpoint z has $k-1$ other neighbors labeled distinctly and differing from the label of z in the clique that z belongs to. An example is shown in Figure 2.7. For each external edge in G', if the endpoints are labeled with label i, then color the corresponding edge in G with color i. Since the vertices in each clique are labeled distinctly in G', each vertex in G will then be incident to k distinctly colored edges, and G is k-edge-colorable.

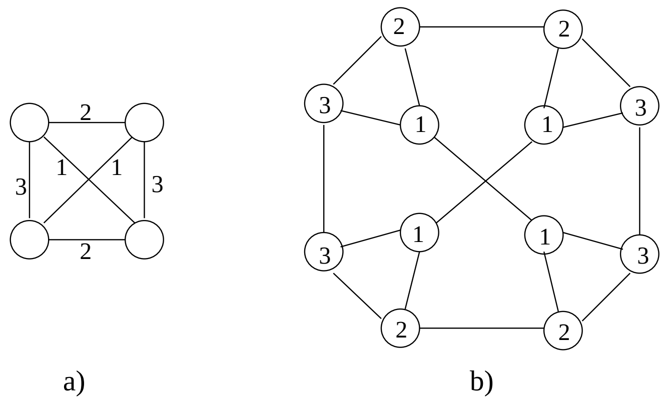


Figure 2.7: a) A 3-regular graph G which is 3-edge-colorable, and b) the vertex labeling of the corresponding graph G'.

If G is k-edge-colorable with colors $\{1, 2, ..., k\}$, then there exists a k-labeling of G' with labels $\{1, 2, ..., k\}$, such that for each edge colored with i in G, there is an external edge in G' both of whose endpoints are labeled with i, and where the internal vertices of each clique K_v are labeled with k distinct labels. Thus G' is $(k, \{1\}, \{1\})$-partitionable, and the proof is complete. ∎

Theorem 2.6 *The following (k, σ, ρ)-partition problems are NP-complete on k-regular graphs for $k \geq 3$:*

(i) $(k, \{0, 1\}, \{1\})$ - *Partition Into k Weakly Perfect Dominating Sets*

(ii) $(k, \{0, 1\}, \{0, 1\})$ - *Partition Into k Total Nearly Perfect Set s*

(iii) $(k, \{1\}, \{0, 1\})$ - *Partition Into k Nearly Perfect Sets Inducing Perfect Matching.*

Proof: Let $s = \max \sigma$, and $r = \max \rho$. For all these three problems, $s = r = 1$, and $s + (k - 1)r = k$. By Lemma 2.1, a k-regular graph G has a $(k, \{1\}, \{1\})$-partitioning if and only if it has a (k, σ, ρ)-partitioning where $\sigma \in \{\{0, 1\}, \{1\}\}$, and $\rho \in \{\{0, 1\}, \{1\}\}$. Thus Theorem 2.5 implies that all of these problems are NP-complete on k-regular graphs for $k \geq 3$. ∎

Theorem 2.7 *The $(k, \{0\}, \{0, 1\})$-partition problem (Partition Into k Strong Stable Sets) is NP-complete on $(k-1)$-regular graphs for $k \geq 4$.*

Proof: The reduction is from the $(k, \{0\}, \{1\})$-partition problem, which is also called *Partition Into k Perfect Codes.* Kratochvíl shows in [62] that this problem is NP-complete on $(k-1)$-regular graphs for $k \geq 4$. Let $s = \max \sigma$ and $r = \max \rho$. For these two problems, $s = 0$, $r = 1$, and $s + (k-1)r = k - 1$. Then by Lemma 2.1, a $(k-1)$-regular graph G has a $(k, \{0\}, \{1\})$-partitioning if and only if G has a $(k, \{0\}, \{0, 1\})$-partitioning. ∎

Theorem 2.8 *The $(k, \{0\}, \mathbb{P})$-partition problem (Partition Into k Independent Dominating Sets) is NP-complete for $k \geq 3$.*

Proof: The reduction is again from k-EC on k-regular graphs. Given a k-regular graph $G = (V, E)$, we construct a graph G' such that G' has a $(k, \{0\}, \mathbb{P})$-partitioning if and only if G is k-edge-colorable.

For each vertex $v \in V$, there is a k-clique K_v in G', and for each edge $(u, v) \in E$, there is a $(k-1)$-clique E_{uv} in G'. Let $(u, v) \in E$, then in G', there is exactly one vertex in K_u, and exactly one vertex in K_v that have edges to every vertex in E_{uv}, and no other vertex in G' has edges to the vertices in E_{uv}. This is depicted in Figure 2.8.

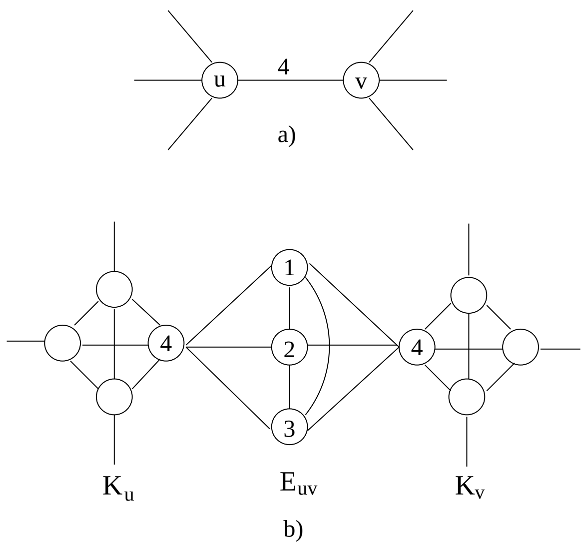


Figure 2.8: a) An edge in G, and b) the corresponding subgraph of G', where G is 4-regular.

Now, a $(k, \{0\}, \mathbb{P})$-partitioning of a graph is equivalent to a k-labeling of the vertices such that each vertex has no neighbors labeled the same as itself, and at least one neighbor labeled with each of the other $k-1$ labels. Vertices in each k-clique must be labeled with k distinct labels, otherwise at least one vertex v in the cliques would have neighbors that are labeled the same as v. Since each vertex of G' appears in a k-clique, this is equivalent to a proper k-coloring of the vertices of G'. Consider Figure 2.8; if the vertex connecting K_u to E_{uv} is labeled with i, then the vertex connecting E_{uv} to K_v must also be labeled with i since the $(k-1)$-clique in between requires all of the remaining $k-1$ labels. In G, we can color the edge (u, v) with color i. With an argument equivalent to the one in the proof of Theorem 2.5, we can conclude that G is k-edge-colorable if and only if G' is $(k, \{0\}, \mathbb{P})$-partitionable. ∎

2.4 Polynomial time results

The polynomial-time solvable (k, σ, ρ)-partition problems listed in Table 2.2 fall in four classes, corresponding to a cutoff value of 2, 3, 4 or ∞. For each (σ, ρ)-set we show that deciding whether a graph has a partition into k classes, where k is less than the cutoff value, is easy.

For the problems with cutoff value 2 this indeed follows as stated in Fact 2.1. The problems with cutoff value ∞ either have the property that a trivial partition with $V_1 = V$ and $V_i = \emptyset, i = 1, \ldots, k$ is a k-partition (see Fact 2.4), or they have $\rho = \{0\}$ and are easy by Fact 2.5.

For the problems with cutoff value 3 or 4, the properties that must be checked to decide whether a graph has a partition of size smaller than the cutoff value are summerized in Table 2.3. We explain the first two of these problems and their solutions further, whereas we leave the rest to be studied in the same way by the reader. Here we assume that the given graph is connected, if it is not, the same argument can be applied on each connected component. For the discussion below, we let P_j denote the set of all simple paths of length j, and C_j the set of all simple cycles of length j.

Let us first consider the $(2, \{1\}, \{1\})$-partition problem. This is equivalent to finding a 2-labeling of the vertices of the given graph G such that each vertex v has exactly one neighbor labeled with the same label as v, and exactly one neighbor labeled with the opposite label. Since the degree of each vertex in G must then be 2, G must be a cycle. The desired labeling is possible if and only if the number of vertices in the cycle is a multiple of 4. Thus we can check in polynomial time whether G is a $4m$-cycle.

The $(2, \{1\}, \{0, 1\})$-partition problem asks for a 2-labeling of the given graph G such that each vertex v has exactly one neighbor labeled with the same label as v, and at most one neighbor labeled with the opposite label. Since the degree of each

(k, σ, ρ)	(k, σ, ρ)-partitionable $(m \geq 1)$
$(2, \{1\}, \{1\})$	C_{4m}
$(2, \{1\}, \{0, 1\})$	$\{P_{2m-1},\, C_{4m}\}$
$(2, \{0, 1\}, \{1\})$	$\{P_{2m-1},\, C_{4m}\}$
$(2, \{0, 1\}, \{0, 1\})$	$\{P_m,\, C_{4m}\}$
$(2, \{0\}, \mathbb{P})$	Bipartite
$(3, \{0\}, \{0, 1\})$	$\{P_m,\, C_{3m}\}$

Table 2.3: For each row, G has the (k, σ, ρ)-partition on the left if and only if G or its connected components belong to the class on the right.

vertex in G must then be at most 2, G must be either a cycle or a path. If G is a cycle, then it must be a $4m$-cycle as above. Otherwise, G must be a path, and since each vertex has exactly one neighbor with the same label as itself, there must be an even number of vertices in the path. Thus we must check whether $G \in \{P_{2m-1}, C_{4m}\}$ for $m \geq 1$.

2.5 Closing remarks

We have presented a systematic study of the complexity of partitioning graphs into generalized dominating sets, with emphasis on those generalized dominating sets which are found in the literature. If we look more broadly at the general class of (k, σ, ρ)-partition problems, we believe that they will all fall in one of the four classes corresponding to a cutoff value of 2, 3, 4 or ∞. We may mention that for graphs of bounded treewidth there are linear-time algorithms solving any (k, σ, ρ)-partition problem [100].

Most problems with cutoff 3 or 4 are easily solvable for partitions into 2 or 3 classes either because such partitions exist only if the graph has maximum degree 2, or because $\sigma = \{0\}$ and a partition into 2 classes exists if and only if the graph is bipartite and the degree of every vertex is in ρ. The exception to this rule is the $(2, \mathbb{N}, \mathbb{P})$ problem, partitioning a graph into two dominating sets, which can always be done by taking one subset to be a maximal independent set, easily found by a greedy algorithm.

For problems with cutoff ∞, it might be of interest to consider only non-trivial partitions, *i.e.*, allow only non-empty partition classes. Some problems, like $(k, \mathbb{N}, \mathbb{N})$ are still easy, since every subset of vertices in a graph is a $(\mathbb{N}, \mathbb{N})$-set.

Let us look at the $(k, \mathbb{P}, \mathbb{N})$-partition problem into non-empty classes $V_1, ..., V_k$ and show that it can be solved for all k using a maximum matching of the graph G. Clearly, G cannot have any isolated vertices. Let $M = \{e_1, e_2, ..., e_m\}$ be a maximum matching of G. If $m \geq k$, then we place in V_i the endpoints of the edge e_i, for $i = 1, ..., k - 1$. The endpoints of the edges $e_k, e_{k+1}, ..., e_m$ are placed in V_k.

Since there are no isolated vertices in G, every other vertex in the graph must be a neighbor of some of the vertices that are already placed in the partition sets. For each such vertex v, place v in any one of the partition sets that contains a neighbor of v. Now, all the vertices in G are partitioned into the sets $V_1, ..., V_k$, and each set contains at least two vertices where each vertex has at least one neighbor in the same set as itself. Note also that, given a non-trivial $(k, \mathbb{P}, \mathbb{N})$-partition for G, a matching with k or more edges can be found by picking an edge from each partition set. Thus G has a non-trivial $(k, \mathbb{P}, \mathbb{N})$-partitioning if and only if G has a matching with k or more edges. Since a maximum matching can be found in polynomial time, this describes a polynomial-time algorithm for the (non-trivial) $(k, \mathbb{P}, \mathbb{N})$-partition problem.

We have not been able to resolve the complexity of the (non-trivial) $(k, \mathbb{N}, \{0, 1\})$ and $(k, \mathbb{P}, \{0, 1\})$-partition problems and will therefore close the paper with the the following open problem.

Open Problem: What is the complexity of deciding whether the vertices of a given graph G can be labeled by two labels such that each vertex has at most one neighbor of a different label than its own, and there is at least one vertex with each label?

Chapter 3

Finding Minimum Height Elimination Trees for Interval Graphs

In this chapter, we consider the problem of finding minimum height elimination trees, which can be viewed as a graph partitioning problem. As earlier discussed, for graph problems that are generally NP-hard, it is important to find either parameters of the problem, or classes of graphs, for which the problem can be solved in polynomial time. We have seen in the previous chapter that it is interesting to find cutoff values for NP-hard graph problems. In this chapter, we will consider the method of finding classes of graphs for which a generally NP-hard problem becomes polynomially solvable. For an arbitrary NP-hard graph problem, a usual approach in this direction is to consider classes of graphs for which polynomial algorithms are already known. We can then try to expand these classes finding new superclasses that have polynomial time algorithms, or prove that the problem is difficult for a superclass. This way, a border similar to the cutoff values can be found between classes of graphs for which the problem becomes easy and remains difficult.

For sparse matrix factorization, two problems that are very important, are the minimum fill, and the minimum elimination tree height problems. Both these problems are NP-hard for general graphs. A very interesting class of graphs concerning sparse matrix computations, is the class of chordal graphs. For this large class of graphs, finding minimum fill orderings is easy. However, it is to date not known whether computing minimum elimination tree height for chordal graphs is NP-hard or not. The paper presented in this chapter, considers two subclasses of chordal graphs, and shows that the problem becomes polynomial for these classes. We start by a restricted class, which we call Ω, and then we expand this class to the class of interval graphs, which is a well-known class with practical applications, and which contains Ω as a subset.

This chapter presents results which are, with some refinements, based on the following two papers:

[4] B. Aspvall and P. Heggernes, *Finding Minimum Height Elimination Trees for Interval Graphs in Polynomial Time.*

[5] B. Aspvall, P. Heggernes, and J. A. Telle, *A Simple Cubic Algorithm for Computing Minimum Height Elimination Trees for Interval Graphs.*

We would like to mention that [4] is an extension and improvement of some of the results presented by Heggernes in [56]. The authors of [4] would like to thank Barry Peyton, and the referees, for valuable suggestions.

3.1 Introduction

Many problems that frequently arise in science and engineering, require the solution of large and sparse systems of linear equations. These systems are often in the form $Ax = b$, where A is a symmetric, positive definite matrix. Solving such a system via *Cholesky factorization* involves computing the Cholesky factor L of A, where $A = LL^T$, and then solving the triangular systems $Ly = b$ and $L^T x = y$. If A is sparse, then it is often useful to do some preprocessing on its associated graph $G(A)$. Various parameters of this graph, like the elimination tree height, determine, for example, how efficiently the system can be solved in parallel.

As parallel computers become more and more available, they are being widely used for solving sparse linear systems. To get the most benefit from using parallel computers, the system to be solved must allow a high level of parallelism. The height of the elimination tree indicates the amount of work that can be done in parallel during Cholesky factorization. A low elimination tree means that more work can be done in parallel, reducing the total amount of time needed.

The problem of minimizing the elimination tree height for general graphs is NP-hard. Several heuristics for reducing the elimination tree height for general graphs have been proposed. In this paper, we use the idea behind one of these heuristics, nested dissection, to develop an $O(n^3)$ algorithm that finds minimum height elimination trees for interval graphs. Interval graphs are useful for modeling real world problems, and they find applications in several research areas.

Aspvall and Heggernes showed in [4] that the problem can be solved for interval graphs in polynomial time. The first version of their paper was published as a technical report in March 1993. Before this result, little was known about polynomial time solvable classes of graphs apart from trees [94]. Later, when Deogun, Kloks, Kratsch, and Müller [22] solved the vertex ranking problem for permutation graphs in time $O(n^6)$, they mentioned that the problem for interval graphs could be solved in time $O(n^3)$. In [5], Aspvall, Heggernes, and Telle revisited the problem, and presented an $O(n^3)$ algorithm that solves the problem for interval graphs. This paper gathers the results from [4] and [5] with some slight changes and improvements. It should also be mentioned that, for any fixed k, an optimal vertex ranking of graphs with treewidth at most k can be found in polynomial time, as shown by Bodlaender et al. [11].

The problem of finding the *height* of a minimum elimination tree of a graph is equivalent to the problem of finding the vertex ranking *number* of a graph. An optimal vertex ranking does not by itself provide enough information to compute an elimination tree of minimum height. However, given also the input graph, a minimum height elimination tree can be constructed rather efficiently. On the other hand, an optimal vertex ranking can readily be found directly from an elimination tree of minimum height. We refer the reader to [22] for details on the vertex ranking problem.

In this paper, we give a simple cubic algorithm for finding minimum height elimination trees for interval graphs, along with some implementation details, so that complete implementation should be straightforward. Furthermore, we show that after an $O(n^2)$ preprocessing, updating the necessary information on the minimal separators, which are needed in the nested dissection approach, can be done in constant time for each separator. This opens for a possible algorithm of lower time complexity if the number of subproblems that can be treated in constant time is lower than $\Theta(n^3)$.

This paper is organized as follows. The necessary background is introduced in Section 3.2. In Section 3.3, we develop a simple and illustrative, but exponential-time algorithm for finding minimum height elimination trees for chordal graphs. This algorithm is modified in Section 3.4 by using dynamic programming to give a polynomial-time algorithm for a rather restricted subclass of chordal graphs, which we call Ω. In Section 3.5, a polynomial-time algorithm is developed for interval graphs as a natural extension of Ω. Section 3.6 deals with the time complexity issues of the algorithms. Finally, some concluding remarks are given in Section 3.7.

3.2 Definitions

The reader is assumed to be familiar with standard graph notation. In this section, we introduce the necessary definitions concerning the elimination process, elimination trees, chordal graphs and clique trees.

3.2.1 Symbolic factorization and elimination trees

Let A be an $n \times n$, symmetric, positive definite matrix. The *associated graph* of A is a graph $G(A) = (V, E)$, where $V = \{v_1, ..., v_n\}$, and $E = \{(v_i, v_j) \mid a_{ij} \neq 0\}$. We will be working on $G(A)$ instead of A throughout this paper. The graph of the Cholesky factor of A, $G(L + L^T)$ is called the *filled graph* of A and is denoted by $G^+(A)$.

The filled graph $G^+ = (V, E^+)$ is found from G by the following process, which is called the *elimination game:* Repeat the following three steps until there are no more vertices in G. 1) Pick a vertex v with least subscript in G. 2) Add edges between the neighbors of v that are not already adjacent. 3) Delete v from G. These added edges are called the *fill* edges. The filled graph G^+ is obtained by adding to G all the fill edges that occur during the elimination. The fill edges correspond to the non-zero elements l_{ij} of L, where $a_{ij} = 0$.

The *elimination tree* T of G is defined as follows. Vertex v_j is the parent of vertex v_i in T if $j = \min\{k > i \mid (v_i, v_k) \in E^+\}$. The elimination tree provides information on column dependencies in the Cholesky factor. Low elimination trees imply less dependency and thus more parallelism. The *height* of the elimination tree is one less than the number of vertices on a longest path from the root to a leaf. Liu gives a thorough examination of elimination trees in [71].

The order in which the vertices of G are eliminated, i.e. the order in which the vertices of G are numbered, is called an *elimination ordering* on G and corresponds to a symmetric permutation of A. Different elimination orderings yield different filled graphs and elimination trees. A *perfect elimination ordering* is an ordering that results in no fill edges. Finding orderings that minimize the size of fill, and finding orderings that minimize the elimination tree height are NP-hard problems, as shown respectively by Yannakakis in [103], and by Pothen in [85].

In this paper, we assume that the graphs we are working with are connected. The results can easily be extended to include disconnected graphs. 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 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 no proper subset of S is a v-w separator. Every minimal separator in G is a minimal v-w separator for some v and $w \in V$.

Several heuristics aim to find orderings that result in low elimination trees. One such heuristic that we will be using in this paper is *nested dissection*, which does the following: pick a separator S in G, number the vertices in S arbitrarily from $n - |S| + 1$ to n, for each component in $G - S$ repeat this process recursively until the components cannot be separated further. By choosing the separators carefully, the height of the elimination tree may be reduced as Manne has shown by the following theorem in [74].

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

It should, however, be noted that some nested dissection orderings with minimal separators may also give elimination trees that are exponentially higher than minimum, as shown by Heggernes [56]. By a nested dissection ordering with minimal separators, we mean, in this paper, that the separators chosen at each step of the algorithm are minimal. No assumptions are made on the size of the resulting components. Theorem 3.1 provides a basis for the algorithms and results that will be presented later in the paper.

3.2.2 Chordal graphs and clique trees

A *clique* in G is a complete subgraph of G. A clique C is *maximal* if no other clique in G contains C. Since cliques are complete subgraphs, a clique can be identified solely by the vertices in it. In this paper, we treat cliques as sets of vertices. Thus a clique C in a graph G is a set of vertices that together induce a complete subgraph of G.

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 [36] that chordal graphs are exactly those graphs with perfect elimination orderings. Thus the chordality of a graph G can be established by finding a perfect elimination ordering for G. Tarjan and Yannakakis present a simple linear algorithm for this in [98], which they call *maximum cardinality search.* All filled graphs are chordal. 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 edges. Dirac shows in [23] that a chordal graph is either a complete graph, or has at least two non-adjacent simplicial vertices. Induced subgraphs of chordal graphs are also chordal. The following lemmas show some more results on chordal graphs. Lemma 3.1 is from [23], and Lemma 3.2 is shown by Rose in [91].

Lemma 3.1 *G is a chordal graph if and only if every minimal v-w separator in G is a clique.*

Lemma 3.2 *For every clique C in a chordal graph, there exists a perfect elimination ordering that eliminates the vertices in C last.*

Let G be a chordal graph and let $\mathcal{V}_G = \{C_1, C_2, ..., C_m\}$ be the set of maximal cliques in G. The following theorem by Gavril [38] will help us define a clique tree for G.

Theorem 3.2 *A graph G is chordal if and only if there exists a tree $\mathcal{T} = (\mathcal{V}_G, \mathcal{E})$ that satisfies the following property: for every vertex v in G, the set of maximal cliques containing v induces a subtree of $\mathcal{T}$.*

Any such tree $\mathcal{T}$ is called a *clique tree* of G. The *clique graph* of G is a graph $\mathcal{G} = (\mathcal{V}_G, \mathcal{E})$, where $\mathcal{E} = \{(C_i, C_j) \mid C_i \cap C_j \neq \emptyset,\ C_i, C_j \in \mathcal{V}_G\}$. Each edge (C_i, C_j) in $\mathcal{E}$ is a nonempty set of vertices, equal to $C_i \cap C_j$. The weight of an edge (C_i, C_j) is $|(C_i, C_j)|$. An example is shown in Figure 3.1. A clique tree of G can also be identified by the following theorems by Bernstein and Goodman [7], and by Blair and Peyton [9].

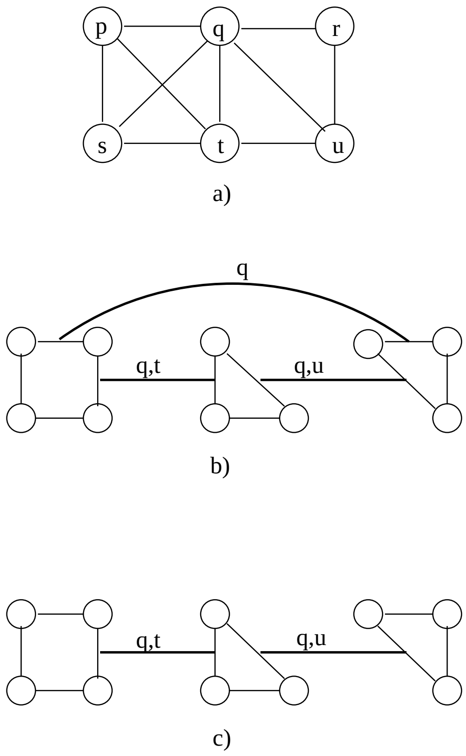


Figure 3.1: a) A chordal graph G. b) The clique graph of G. c) The clique tree of G, which in this case is unique.

Theorem 3.3 *A tree $\mathcal{T} = (\mathcal{V}_G, \mathcal{E})$ is a clique tree of G if and only if $\mathcal{T}$ is a maximum-weight spanning tree of the clique graph of G.*

Theorem 3.4 *A tree $\mathcal{T} = (\mathcal{V}_G, \mathcal{E})$ is a clique tree of G if and only if, for every pair of distinct cliques C_i and $C_j \in \mathcal{V}_G$, 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 3.1 follows directly from Theorem 3.4.

Corollary 3.1 *A tree $\mathcal{T} = (\mathcal{V}_G, \mathcal{E})$ is a clique tree of G if and only if, for every pair of distinct cliques C_i and $C_j \in \mathcal{V}_G$, the intersection $C_i \cap C_j$ is contained in every edge on the path connecting C_i and C_j in $\mathcal{T}$.*

As can be seen from the previous definitions and theorems, the clique graph of G is unique, whereas Ho and Lee show in [59] that G may have exponentially many clique trees. It is shown in [23] that the number of maximal cliques in a chordal graph G is less than or equal to the number of vertices in G. Thus $|\mathcal{V}_G| \leq |V|$.

There is a close relationship between clique trees and separators of G, which can be seen in the following results. Lemma 3.3 and Corollary 3.2 are from [59], and are quoted without proofs.

Lemma 3.3 *Let G be a chordal graph and let $\mathcal{T}$ be a clique tree of G. Every edge (C_i, C_j) in $\mathcal{T}$ is a minimal v-w separator in G, and for every minimal v-w separator S in G there is at least one edge $(C_i, C_j) = S$ in $\mathcal{T}$.*

Corollary 3.2 *If a chordal graph G has m distinct 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 3.2, the number of minimal v-w separators in G is less than or equal to $n - 1$. Recall that both the edges and the vertices in a clique tree of G are cliques in G.

In the next section, we will use the results presented here to find minimum height elimination trees for chordal graphs. For more detailed information on clique trees, see the survey by Blair and Peyton [10].

3.3 An algorithm for chordal graphs

In this section, we show how we can find a minimum height elimination tree for a chordal graph G by using a clique tree of G. Since minimum height elimination tree orderings are not always minimum fill orderings, we will not be producing perfect elimination orderings in general. Thus we are interested in finding minimum height elimination trees without restricting the amount of fill. The recursive algorithm presented in this section is not of polynomial-time complexity, but will serve to illustrate how clique trees can be used for this purpose. 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 of G.

A linear-time algorithm for solving Problem 1 can be found in [10]. 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 of 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 of G can then be found by trying out every minimal separator S as the top separator, and finding minimum height elimination trees for the components in $G - S$ recursively.

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 3.4 *Let G be a chordal graph and $\mathcal{T}$ a clique tree of G. Let H be any induced subgraph of G. Any minimal v-w separator S_H in H is a subset of at least one edge in $\mathcal{T}$.*

Proof: Since induced subgraphs of chordal graphs are also chordal, H is a chordal graph. Let S_H be a minimal v-w separator in H. By Lemma 3.1, S_H is a clique. Since S_H is a minimal v-w separator, by Lemma 3.3 there exist at least two maximal cliques C_1 and C_2 in H such that $S_H = C_1 \cap C_2$. Now, $C_1 \subseteq K_1$ and $C_2 \subseteq K_2$ for two distinct maximal cliques K_1 and K_2 in G. To see this, remember that H is induced and C_1 and C_2 cannot be subsets of one single maximal clique in G, since there are vertices in C_1 and C_2 that are not neighbors. Clearly, K_1 and K_2 are vertices in the clique tree $\mathcal{T}$ and $S_H \subseteq K_1 \cap K_2$. Thus, by Corollary 3.1, S_H is a subset of each edge on the path between K_1 and K_2 in $\mathcal{T}$. ∎

As we can see by Lemma 3.4, the clique tree provides all the information that is needed by each step of a nested dissection algorithm. Given a chordal graph G and a clique tree $\mathcal{T}$ of G, the minimal separators in each subgraph that we must consider, are edges in $\mathcal{T}$.

Before we can continue, we also need to show the relationship between subtrees of $\mathcal{T}$ and induced subgraphs of G. Let S be a minimal v-w separator in G and let $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ be a clique tree of G. We define $\mathcal{T} - S$ to be the forest obtained by removing the vertices in S from every maximal clique and every edge in $\mathcal{T}$. The edges and maximal cliques that become empty are removed from $\mathcal{T}$. Since there is at least one edge $(C_i, C_j) = S$ in $\mathcal{T}$, $\mathcal{T} - S$ is disconnected. Let $\mathcal{T}' = (\mathcal{V}', \mathcal{E}')$ be a component of $\mathcal{T} - S$ with $\mathcal{V}' = \{C'_{i_1}, ..., C'_{i_k}\}$, and $\mathcal{E}' = \{S'_{i_1}, ..., S'_{i_{k-1}}\}$. Then $\mathcal{T}'$ corresponds to a connected induced subtree $\mathcal{T}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ of $\mathcal{T}$ with $\mathcal{V}_1 = \{C_{i_1}, ...C_{i_k}\}$ and $\mathcal{E}_1 = \{S_{i_1}, ..., S_{i_{k-1}}\}$, where $C'_{i_j} \subseteq C_{i_j}$ for $1 \leq j \leq k$, and $S'_{i_j} \subseteq S_{i_j}$ for $1 \leq j < k$.

Let $G' = (V', E')$ be a component of $G - S$ such that $V' = \{v \mid v \in C'_{i_j}, 1 \leq j \leq k\}$. Such a component must exist by Theorem 3.2. Notice that $\mathcal{T}'$ is not necessarily a proper clique tree of G', since there may be cliques in $\mathcal{T}'$ that are no longer maximal after the removal of the vertices in S. But as we have seen in Lemma 3.4, minimal separators in G' are subsets of minimal separators in G. They will therefore be edges of $\mathcal{T}'$, which thus includes all the information we need about the minimal separators of G'. By Theorem 3.4, a clique that is not maximal in $\mathcal{T}'$ must be a subset of one of its neighbors. A proper clique tree of G' can be found by merging each clique in $\mathcal{T}'$ that is not maximal with its neighbor that is a superset.

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 3.5 *We can find a minimum height elimination tree of 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 of 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 $|C| - 1$. As will be clear from the rest of the proof, for our purposes, it is convenient to define that an empty tree has height -1.

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 of a chordal graph having $k + 1$ maximal cliques for a chordal graph G. Let S be an edge in $\mathcal{T}$ that is a minimal separator in G. By the induction hypothesis, for each component $\mathcal{T}'$ in $\mathcal{T} - S$, we know how to find a minimum height elimination tree of the subgraph of G corresponding to $\mathcal{T}'$. We find an elimination tree T for G in the following way: Let S be 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 of 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. ∎

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*, which is shown in Figure 3.2, returns the minimum elimination tree height for a chordal graph G, given a clique tree $\mathcal{T}$ of G.

In the proof of Theorem 3.5, 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 by Algorithm *MinTree* although we also consider the edges in $\mathcal{T}$ that are not minimal separators. In fact, the minimum height elimination tree found by the algorithm may contain separators that are not minimal.

In Algorithm *MinTree*, the proper clique tree for each component $\mathcal{T}'$ is not computed. By the same argument as above, this is not necessary since each $\mathcal{T}'$

```
Algorithm MinTree (T: Clique tree): integer;

begin
    if T has only one maximal clique C then
        MinTree = |C| - 1
    else
        min = ∞;
        for each edge S in T do
            max = -1;
            for each component T' in T - S do
                height = MinTree(T');
                if max < height then
                    max = height;
            end-for;
            if min > |S| + max then
                min = |S| + max;
        end-for;
        MinTree = min;
    end-if;
end;
```

Figure 3.2: Algorithm *MinTree*.

includes all the information we need and more.

The time complexity of 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 polynomial-time algorithms for subclasses of chordal graphs. Such an algorithm for a subclass of chordal graphs, which we call Ω, is presented in the next section. Our main goal is to design an efficient algorithm for interval graphs. The results in the next section will build a bridge from general chordal graphs to interval graphs, and illustrate how dynamic programming can be used to produce polynomial-time algorithms.

3.4 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 3.1 *A graph G belongs to Ω if G is chordal and has at most two maximal cliques that contain simplicial vertices.*

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 in polynomial time 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 of 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.

We now present some results about the graphs in Ω and their clique trees, which we will use to develop an algorithm similar to $MinTree$.

Lemma 3.5 *A graph $G \in \Omega$ if and only if every perfect elimination ordering on G results in an elimination tree that has either one or two leaves.*

Proof: *(if)* 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. This however, results in an elimination tree that has at least three leaves.

(only if) 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 possible only when G has at least three non-adjacent simplicial vertices. But then G must have at least three maximal cliques with simplicial vertices. ∎

We now define a special type of graph that will be used throughout the rest of this paper.

Definition 3.2 *A trail is a tree $T = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$, and $E = \{(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)\}$.*

We continue by showing in Theorem 3.6 that a graph $G \in \Omega$ has a unique clique tree that is a trail. The following lemma will help us to prove Theorem 3.6.

Lemma 3.6 *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, 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 3.1 that $(C_i, C) \subseteq (C, C')$. Thus, the vertices in $C - (C, C')$ 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 $C - (C, C')$. ∎

Theorem 3.6 *Let G be a graph in Ω. Then G has a unique clique tree $\mathcal{T}$ that is a trail.*

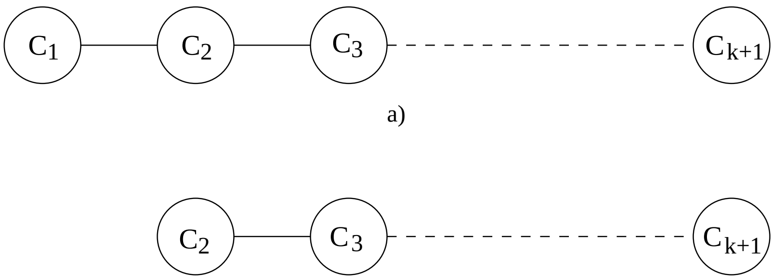
Proof: Let $\mathcal{T}$ be a clique tree of G. By the definition of Ω and Lemma 3.6, $\mathcal{T}$ has at most two leaf cliques. If G has only one maximal clique, then this is the only clique in $\mathcal{T}$, and $\mathcal{T}$ is therefore unique. If G has two or more maximal cliques, then any clique 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 3.3 a). Let G' be an induced subgraph of G that 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 3.1, in $\mathcal{T}$, (C_1, 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 3.3 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 verify 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'.



a)

b)

Figure 3.3: $\mathcal{T}$ and $\mathcal{T}'$ in the proof of Theorem 3.6.

In order to show Statement 1, we note that in $\mathcal{T}$, (C_1, 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, C_2) were contained in some other maximal clique, then by Corollary 3.1 (C_2, C_3) would contain (C_1, 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 $C_i, 3 \leq i \leq k$, 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, 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. ∎

Graphs that do not belong to Ω may also have clique trees that are trails. One of the several clique trees for a chordal graph G may be a trail although G does not belong to Ω. Also, a graph $G \notin \Omega$ may have different clique trees that are all trails.

Let G be a graph in Ω. Theorem 3.6 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 one of the leaf cliques in $\mathcal{T}$ be numbered 1. 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 3.6, when the simplicial vertices in C_1 are eliminated, the only neighbor of C_1 becomes a leaf clique with simplicial vertices. We call this neighbor C_2, and continue by eliminating the simplicial vertices in C_2. Whenever the simplicial vertices in a leaf clique C_i are eliminated, the only neighbor of C_i becomes a leaf clique with simplicial vertices in the remaining clique tree. We can then number this neighbor as C_{i+1}, and continue by eliminating the simplicial vertices in C_{i+1}. Let us start with C_1 and repeat this process until the remaining clique tree is empty. This elimination ordering α results in an elimination tree with one leaf, and implies a natural linear ordering of the maximal cliques in G.

In the rest of this section, we assume that the maximal cliques of G are numbered from 1 to m as described above. This argument can also 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. By Lemma 3.2, any maximal clique on the path from C_1 to C_m can be eliminated last in a perfect elimination ordering.

This discussion about perfect elimination orderings is included here only in order to define a natural linear ordering of the cliques in $\mathcal{T}$. As mentioned before, the orderings that minimize the elimination tree height, are generally not perfect elimination orderings. Thus the algorithms we present in this paper do not generally produce perfect elimination orderings. 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 of G. The number of maximal cliques in G will always be denoted by m, unless specified otherwise.

Lemma 3.7 *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 3.1, S is a clique, and by Lemma 3.2, 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. This implies the existence of at least three maximal cliques in G with simplicial vertices, and thus contradicts the fact that $G \in \Omega$. ∎

Lemma 3.8 *Let G be a graph in Ω and let $\mathcal{T}$ be the clique tree of G. Then there do not exist two distinct edges S and S' in $\mathcal{T}$, such that $S' \subseteq S$.*

Proof: Suppose there exist two distinct edges S and S' for which $S' \subseteq S$. Let $S = (C_1, C_2)$, and $S' = (C_1', C_2')$. Without loss of generality assume the cliques occur in the following order:

$$..., C_1, C_2, ..., C_1', C_2', ...$$

Possibly, $C_2 = C_1'$. Consider the tree obtained by removing the edge (C_1', C_2'), and adding the edge (C_2', C_1). Since $S' \subseteq S$, the new tree has weight no smaller than that of the original; moreover, the weight of the new tree cannot be larger, else the original tree would not be a clique tree. It follows that the new tree is a clique tree of G. The clique C_1' is a leaf in the new tree and hence by Lemma 3.6, C_1' has simplicial vertices. With at least three cliques with simplicial vertices, G cannot belong to Ω. ∎

The following corollary follows directly from Lemma 3.8.

Corollary 3.3 *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 3.3 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 3.3 that the separators provided by the clique trees, are exactly the minimal separators that we need.

Definition 3.3 *Let G be a graph in Ω, and $\mathcal{T}$ the clique tree of G. For all i, j, where $1 \le i \le j \le m$, we define H_{ij} to be the subgraph of G induced by the set of vertices internal to maximal cliques $C_i, C_{i+1}, ..., C_j$. In other words, H_{ij} is the graph induced by the vertices in the following set: $(C_i \cup ... \cup C_j) - (S_{i-1} \cup S_j)$, where S_k denotes the separator (C_k, C_{k+1}), for $1 \le k < m$, and $S_0 = S_m = \emptyset$.*

Lemma 3.9 *Let G be a graph in Ω, and $\mathcal{T}$ the clique tree of G. For all i, j, where $1 \le i \le j \le m$, $H_{ij} \in \Omega$.*

Proof: Let $1 < i \le j < m$ (the cases where either $i = 1$ or $j = m$ will follow readily from the proof). By Lemma 3.7, removing S_{i-1} from G divides the remaining graph into two connected components. These components are $H_{1,i-1}$ and H_{im}. We will first prove by contradiction that H_{im} belongs to Ω. Assume that $H_{im} \notin \Omega$. Then H_{im} must have at least three maximal cliques with simplicial vertices. Let i' be the least integer greater than i such that $C_{i'} - S_{i-1}$ is a maximal clique in H_{im}. Let $C'_k = C_k - S_{i-1}$, $k = i', \ldots, m$. By Lemma 3.6, the leaf cliques $C'_{i'}$ and C'_m of H_{im} contain simplicial vertices. Additionally, by our assumption, there must exist a maximal clique C'_l, where $i' < l < m$, such that C'_l has a simplicial vertex x in H_{im}. A simplicial vertex can only occur in one maximal clique, so the adjacency set of x in H_{im} is the clique $C'_l - \{x\}$. Now, either x is simplicial (belonging only to C_l) also in G, or S_{i-1} contains some neighbors of x, whose removal makes x a simplicial vertex. In the latter case, x must belong to every C_k, for $i \leq k \leq l$. This contradicts the fact that $x \notin C'_{i'}$. Therefore, x must be simplicial also in G, and since $G \in \Omega$, we have arrived at a contradiction. Thus H_{im} belongs to Ω. (It can be shown similarly that $H_{1,i-1}$ also belongs to Ω.) To show that $H_{ij} \in \Omega$, assume that $i' \leq j$. (If $j < i'$, then H_{ij} is empty and thus belongs to Ω.) We remove $S'_j = S_j - S_{i-1}$ from H_{im}, dividing the remaining graph into the two connected components H_{ij} and $H_{j+1,m}$. By the same argument as above, H_{ij} (and $H_{j+1,m}$) belong to Ω. ∎

By Lemma 3.9, 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 3.7 *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 of 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 -1. Otherwise, the elimination tree height is one less than the number of the vertices in H_{ij}, and the elimination tree consists of the vertices in H_{ij}, and has only one leaf.

We assume now that the induction hypothesis is true, and show how this enables us to find a minimum height elimination tree of 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 $\mathcal{T}'$ to be the top separator in the elimination tree. Let $S = (C'_p, C'_q)$ be this edge, where C'_p and C'_q are two consecutive cliques in $\mathcal{T}'$. By Lemma 3.7, 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 of H_{ij}. Further, let $T_S(H_{ij})$ be the elimination tree of H_{ij} with S as the top separator, and with $T_{min}(H_{ip})$ and $T_{min}(H_{qj})$ as the two subtrees that hang from S. This is illustrated in Figure 3.4. 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. ∎

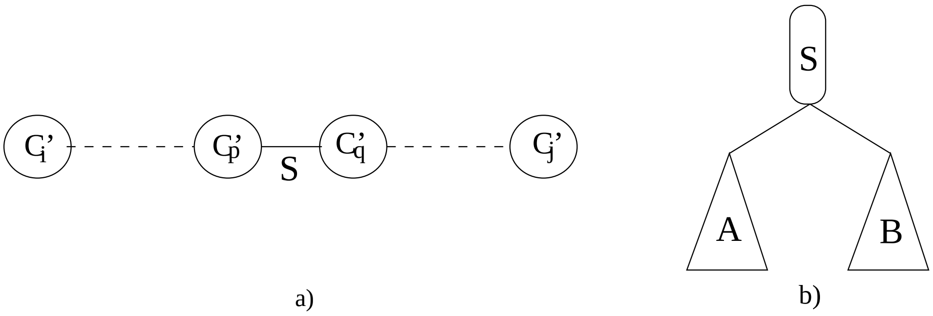


Figure 3.4: a) $\mathcal{T}'$. b) $T_S(H_{ij})$, where $A = T_{min}(H_{ip})$, and $B = T_{min}(H_{qj})$.

We now present Algorithm *Height* that computes the minimum elimination tree height for a graph in Ω as described in the proof of Theorem 3.7. For simplicity, we compute only 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 $S_i = (C_i, C_{i+1})$. We define $Simp(C_i)$ to be the set of simplicial vertices in clique C_i of $\mathcal{T}$. According to the definition of Ω, $Simp(C_1)$ and $Simp(C_m)$ are nonempty, whereas $Simp(C_i)$ is empty for $2 \leq i \leq m - 1$. The algorithm returns the height of a minimum height elimination tree for $H_{fromc,toc}$, where $1 \leq fromc \leq toc \leq m$. Algorithm *Height* is shown in Figure 3.5.

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

begin
  for $i = fromc$ **to** $toc - 1$ **do**
    $S'_i = S_i - (S_{fromc-1} \cup S_{toc})$;
  if $fromc = toc$ **then**
    $Height = |Simp(C_{fromc})| - 1$;
  else
    $opt = \infty$;
    for $i = fromc$ **to** $toc - 1$ **do**
      $min = |S'_i| + \max(Height(\mathcal{T}, fromc, i), Height(\mathcal{T}, i+1, toc))$;
      if $min < opt$ **then**
        $opt = min$;
    end-for;
  end-if;
  $Height = opt$;
end;

Figure 3.5: Algorithm *Height*.

Theorem 3.8 *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 3.7 and Algorithm *Height*. ∎

Although 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, which is called *DynHeight*, 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 of H_{ij}. Algorithm *DynHeight* is shown in Figure 3.6.

As we can see in Algorithm *DynHeight*, 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

```
Algorithm DynHeight (T : Clique tree);

begin
    for i = 1 to m do                    (* Initialization*)
        for j = 1 to m do
            Opt[i, j] = ∞;
            Root[i, j] = ∅;
        end-for;

    for i = m downto 1 do                (* Main loop*)
        Opt[i, i] = |Simp(C_i)| − 1;
        for j = i + 1 to m do
            for k = i to j − 1 do
                S'_k = S_k − (S_{i−1} ∪ S_j);
                height = |S'_k| + max(Opt[i, k], Opt[k + 1, j]);
                if height ≤ Opt[i, j] then
                    Opt[i, j] = height;
                    Root[i, j] = S'_k;
                end-if;
            end-for;
    end-for;

end;
```

Figure 3.6: Algorithm *DynHeight*.

in the algorithm. It is then easy to see that the entry in $Opt[1, m]$ computed by $DynHeight(\mathcal{T})$ is the same as the result returned by $Height(\mathcal{T}, 1, m)$.

We leave the time complexity analysis of this algorithm to later. The complexity is clearly polynomial, and we will see in Section 3.6 that the complexity of finding minimum height elimination trees for graphs in Ω is $O(n^3)$. In Section 3.6, detailed complexity analysis of an algorithm that finds minimum height elimination trees for a superclass of Ω is given, along with an explanation on how separators can be updated efficiently.

Algorithm *DynHeight* finds only the elimination tree height. We can find the actual elimination tree by using the information provided by this algorithm in the array *Root*. 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 in linear time. The actual elimination tree follows readily from this separator tree.

The idea of dynamic programming will be used in the next section to present a polynomial-time algorithm for an important superset of Ω.

3.5 Interval graphs

In Section 3.3, we showed that *any* clique tree of a chordal graph G could be used to find a minimum height elimination tree of G. In Section 3.4, we showed that if the clique tree is *unique* and a trail, where *no* interior clique has simplicial vertices, then dynamic programming can be used to develop a polynomial-time algorithm. In this section, we combine these two results and show how to find minimum height elimination trees in polynomial time for graphs that have *a* clique tree that is a trail *without any restrictions* on simplicial vertices. Such a class will obviously be a superclass of Ω and a subclass of chordal graphs.

By Theorem 3.5 any clique tree of G can be used as long as we try every possible ordering of the minimal separators in G. If there exists a clique tree $\mathcal{T}$ for G that is a trail, then the maximal cliques in G can be numbered from left to right as they appear in $\mathcal{T}$. This gives a linear ordering of the maximal cliques in G, and thus dynamic programming can easily be used. It turns out that the class of graphs that have this property is indeed the well-known class called interval graphs. We first give a definition of this class.

Definition 3.4 *A graph $G = (V, E)$ is an interval graph if there is a mapping I of the vertices in G into sets of consecutive integers such that for each pair of vertices $v, w \in V$ the following is true: $(v, w) \in E \Leftrightarrow I(v) \cap I(w) \neq \emptyset$.*

The following theorem, here stated in terms of clique trees, was first proved by Gilmore and Hoffman [51] (see also [53], page 172).

Theorem 3.9 *A graph $G = (V, E)$ is an interval graph if and only if there exists a clique tree $\mathcal{T}$ for G, where $\mathcal{T}$ is a trail.*

Booth and Lueker give a linear-time algorithm for testing whether a graph is an interval graph in [13]. Their algorithm is constructive so that the set $I(v)$ for each vertex v in G is found in linear time and thus is of linear cardinality. Therefore the construction of a clique tree that is a trail can be done efficiently.

Let G be an interval graph, S a minimal separator in G, and $\mathcal{T}$ a clique tree of G that is a trail. We now present Algorithm *Interval* that finds minimum elimination tree height for G. We will use the same method as in Algorithm *DynHeight*, but this time the connectivity of each subgraph has to be checked. Let I_{ij} be defined for interval graphs in the same way as H_{ij} was defined for the graphs in Ω. As we have done in Algorithm *DynHeight*, we organize the computation such that when we are working with I_{ij}, the minimum elimination tree height for each I_{kl}, $i \leq k \leq l \leq j$, is already computed.

Golumbic shows in [53] that the interval graph property is hereditary. Thus each I_{ij} is also an interval graph. The main difference between subgraphs I_{ij} and H_{ij} is that each I_{ij} is not necessarily connected, but the connected components of I_{ij} are interval graphs. Since by Theorem 3.2 the connected induced subgraphs of G correspond to connected induced subtrees of $\mathcal{T}$, we need first to check whether the subgraph $\mathcal{T}'$ of $\mathcal{T}$ that corresponds to I_{ij} is connected or not. If not, I_{ij} cannot have an elimination tree. In order to handle this case, we introduce the notion of *elimination forests*. Another important difference between Ω and interval graphs, is that H_{ii} is empty, whereas I_{ii} need not be empty, for $1 < i < m$.

Definition 3.5 *An elimination forest for a disconnected graph G is a forest where each connected tree is an elimination tree of a connected component of G. The height of an elimination forest is the height of a highest connected elimination tree in the forest. An elimination forest of minimum height consists of elimination trees of minimum height for each of the connected components of G.*

If I_{ij} is not connected, then at least one of the sets S'_k is empty. In this case we want to find an elimination forest of minimum height for I_{ij}. An elimination forest F with connected components $T_1, ..., T_p$, can be regarded as having a dummy top separator S which is empty, and where $T_1, ..., T_p$ are the children of S. Thus we can treat an empty set S'_k as a separator of size 0; an *empty separator*, and try it as the top separator for I_{ij}.

When trying S'_k, which might be empty, as the top separator, the minimum elimination tree heights of I_{ik} and $I_{k+1,j}$ are found in $Opt[i,k]$ and $Opt[k+1,j]$, which are already computed. If one or both of these subgraphs are disconnected then the corresponding values in the table *Opt* are the heights of elimination forests of minimum height.

If I_{ij} is disconnected, then choosing an empty separator S'_k as the top separator must yield the minimum height. This will also give a correct elimination forest with S'_k as the dummy top separator. (Choosing a separator in a component of I_{ij} that has the highest minimum height elimination tree, as the top separator for I_{ij}, may also give the minimum height for I_{ij}. But this will not result in a valid elimination tree, since I_{ij} must have an elimination forest which is disconnected.) Therefore, when we find out that I_{ij} is disconnected, we choose an empty separator S'_k as the top separator.

Algorithm *Interval* is shown in Figure 3.7. The maximal cliques and the edges in $\mathcal{T}$ are assumed to be numbered as for Algorithm *DynHeight*.

Algorithm *Interval* computes the minimum elimination tree height of an interval graph. It may be confusing that we choose empty sets to be separators, but this is just symbolic. If the top separator is empty, then we have an elimination forest. Otherwise the graph must be connected and has an elimination tree. The elimination tree defined by the resulting array *Root* at the end of the algorithm is a binary

Algorithm *Interval* ($\mathcal{T}$: Clique tree);

```
begin

    for i = 1 to m do                    (* Initialization*)
        for j = 1 to m do
            Opt[i,j] = ∞;
            Root[i,j] = 0;
        end-for;

    for i = m downto 1 do                (* Main loop*)
        Opt[i,i] = |Simp(C_i)| - 1;
        for j = i + 1 to m do
            for k = i to j - 1 do
                |S'_k| = |S_k - (S_{i-1} ∪ S_j)|;
                height = |S'_k| + max(Opt[i,k], Opt[k+1,j]);
                if height ≤ Opt[i,j] then
                    Opt[i,j] = height;
                    Root[i,j] = k;
                end-if;
                if |S'_k| = 0 then
                    k = j;               (*I_ij is disconnected - jump out of the loop *)
            end-for;
    end-for;

end;
```

Figure 3.7: Algorithm *Interval*.

separator tree where some of the separators are empty sets. To find the separators of the actual elimination tree, we must recursively do the following starting from the top separator. If a separator S in the elimination tree is empty then we set $parent(leftchild(S)) = parent(S)$ and $parent(rightchild(S)) = parent(S)$. Since G is assumed connected, we have always a top separator that is nonempty, and thus we have a connected elimination tree for G. Therefore, we have a valid base case, and for each new separator S to consider as we go down the elimination tree, we know that none of the ancestors of S are empty.

In Algorithm *DynHeight*, the table *Root* actually contains the vertices to be placed in the elimination tree, and the tree can be built in linear time in a top-down fashion, as explained in Section 3.4. Now in Algorithm *Interval*, we only compute the sizes of the separators and not which vertices they contain. *Root* only contains the names (subscripts) of the separators that are chosen. Therefore, the construction is a little more complicated. Again we build the tree in a top-down fashion. The vertices of G are marked when they are placed in the elimination tree, and each new separator to be placed in the tree must check which of its vertices are already in the elimination tree. Thus the construction of the elimination tree can be done in time $O(n^2)$.

This difference between *DynHeight* and *Interval* is due to complexity issues. In the next section, we will give a time complexity analysis for Algorithm *Interval*. Clearly, since Ω is a subclass of Interval graphs, this time complexity also applies for the graphs in Ω. The following result is already clear from Algorithm *Interval*, and the related discussion.

Theorem 3.10 *A minimum height elimination tree of an interval graph can be found in polynomial time.*

It turns out that interval graphs are among the most useful mathematical structures for modeling real world problems. Applications of interval graphs can be found in several areas like mathematics, chemistry, biology and archeology. In the final section we will see the role of interval graphs in sparse factorization. We will also look at the place of Ω among the subclasses of interval graphs. Golumbic gives a detailed study of interval graphs in [53].

3.6 Time complexity issues

3.6.1 Set update

In this section we look at how we can compute $|S'_k| = |S_k - (S_{i-1} \cup S_j)|$ efficiently. For each pair $(i, j), i < j$, $|S'_k|$ is computed for $i \leq k \leq j-1$ by Algorithm *Interval*. If the actual sets S'_k were to be computed, this would take $O(n^4)$ time. But the algorithm only needs the sizes of the minimal separators in order to be able to compute the minimum elimination tree height. We will now see that after a preprocessing of $O(n^2)$ on all the pairs of separators (S_i, S_j), $i < j$, computing a value $|S_k - (S_i \cup S_j)|$ can be done in constant time when k is between i and j.

Lemma 3.10 *Let G be an interval graph, and let $S_1, S_2, ..., S_{m-1}$ be the minimal separators of G numbered as they appear in a clique tree that is a trail. For $1 \leq i \leq k \leq j < m$, $|S_k - (S_i \cup S_j)| = |S_k| - |S_i \cap S_k| - |S_k \cap S_j| + |S_i \cap S_j|$.*

Proof: From set theory we have the following result for general sets:
$|A - (B \cup C)| = |A| - |A \cap B| - |A \cap C| + |A \cap B \cap C|$. From Theorem 3.4, we can conclude that $(S_i \cap S_j) \subseteq S_k$. Thus, $S_i \cap S_k \cap S_j = S_i \cap S_j$. From this the lemma follows with $A = S_k$, $B = S_i$, and $C = S_j$. ∎

As we can see from Lemma 3.10, $|S_k - (S_{i-1} \cup S_j)|$, for any triple $(i, k, j), i \leq k \leq j$, can be computed in constant time if $|S_k|, |S_{i-1} \cap S_k|, |S_k \cap S_j|$ and $|S_{i-1} \cap S_j|$ are already computed. We will show that in time $O(n^2)$, $|S_i \cap S_j|$ can be computed for all pairs (i, j), $1 \leq i < j < m$.

In order to explain how to find the sizes of the intersections between all pairs of minimal separators in G, we need the following definitions. Let $\mathcal{T}$ be a clique tree of G that is a trail. To each vertex v in G, we associate a number $right(v)$ which is the number of the rightmost maximal clique in $\mathcal{T}$ that contains v. That is, $right(v) = max\{i \mid v \in C_i\}$. Let the vertices in each minimal separator S_i be sorted by their $right$ values. The vertices are renamed so that $S_i = \{v_1^i, v_2^i, ..., v_{n_i}^i\}$, where $right(v_k^i) \leq right(v_{k+1}^i)$, $1 \leq k < n_i$.

For each S_i, $i = 1, 2, ..., m-2$, we want to find $|S_i \cap S_j|$ for $i < j < m$. For a vertex v_k^i in S_i, we know by Theorem 3.4 that if $right(v_k^i) > j$ then $v_k^i \in S_j$, since $S_j = C_j \cap C_{j+1}$. Let k be the smallest index for which $right(v_k^i) > j$. Then we have $right(v_{k-1}^i) \leq j$, and thus $v_{k-1}^i \notin S_j$. In this case, by the same theorem, we can conclude that $v_1^i, ..., v_{k-1}^i \notin S_j$, and $v_k^i, ..., v_{n_i}^i \in S_j$. The preprocessing algorithm is given in Figure 3.8.

Considering the time complexity of Algorithm *Preprocess*, notice that k is initiated outside the j-loop, and the number of steps in this loop is $O(n_i + (m-1-i))$. The total number of steps for Algorithm *Preprocess* is then $O(\sum_{i=1}^{m-2}(n_i + (m-1-i)) = O(n^2)$, since $n_i \leq n$ and $m \leq n$, where n is the number of vertices in G.

Regarding Algorithm *Interval*, the sizes of all the minimal separators and of $Simp(C_i)$ for all the maximal cliques can easily be computed within this time bound. Thus the total amount of work needed to be done in Algorithm *Interval* is:

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

Since $m \leq n$ for chordal graphs, this time complexity is no worse than $O(n^3)$.

Algorithm *Preprocess;*

begin

> **for** $i = 1$ **to** $m-2$ **do**
> > $k = 1;$
> > **for** $j = i+1$ **to** $m-1$ **do**
> > > **while** $k \leq n_i$ **and** $right(v_k^i) \leq j$ **do**
> > > > $k = k+1;$
> > >
> > > $|S_i \cap S_j| = n_i - k + 1;$
> >
> > **end-for;**
>
> **end-for;**

end;

Figure 3.8: Algorithm *Preprocess.*

3.6.2 Can we do better?

As we have seen, the overhead work in Algorithm *Interval* is done in time $O(n^2)$. What increases the time complexity is the number of minimal separators that we have to run through for each subproblem I_{ij}.

In dynamic programming algorithms, a linear factor can often be removed from the time bound if the problem possesses some monotonicity property. When computing $Opt[i,j]$ and $Root[i,j]$, Algorithm *Interval* runs through every k between i and j. Assume that the following monotonicity property holds: There exists an integer k, such that $Root[i,j-1] \leq k \leq Root[i+1,j]$, and setting $Root[i,j] = k$ gives the minimum height for I_{ij}. In this case, the total time is reduced to $O(n^2)$ when the innermost loop only runs for $k = Root[i,j-1]$ to $Root[i+1,j]$.

Our algorithm finds an arbitrary minimum height elimination tree or forest for each subproblem. With this approach, the monotonicity property mentioned above does not hold for interval graphs or for the class Ω. The lack of monotonicity in the sizes of the separators S'_k, $i \leq k < j$, seems to be an obstacle. We still suspect and hope that at least one of these classes possesses a similar property. For example, can one, by restricting the attention to minimum height elimination trees with some additional requirements, obtain a desirable monotonicity property? We have not been able to prove such a result so far.

3.7 Concluding remarks

The main contribution of this paper has been to introduce a polynomial algorithm for finding minimum height elimination trees for interval graphs. In this last section, we give a couple of interesting results on interval graphs and the class Ω that might help to understand how widely applicable these graphs are. We start by looking at the matrices that correspond to interval graphs and see what interval graph property means for Cholesky factorization of such matrices.

3.7.1 Dense profile graphs

Let $G = (V, E) = G(A)$ for a symmetric, positive definite matrix A. One method that aims to reduce the fill size in Cholesky factorization of A is the *bandwidth method.* For each vertex v_i in V, let $f(v_i) = \min\{j \mid (v_i, v_j) \in E\}$, and $b(v_i) = i - f(v_i)$. The *bandwidth* of G is defined as $b = \max\{b(v_i) \mid v_i \in V\}$. The bandwidth method finds a band of width $2b+1$ about the main diagonal of A outside of which all the entries are zero, and performs Cholesky factorization within the band.

The problem of finding an ordering of the vertices in G that results in minimum bandwidth for G is NP-hard [81]. Various heuristics exist for finding orderings with small bandwidth. Cuthill and McKee give an algorithm in [20] for this purpose.

An extension of the bandwidth method is the *profile method.* The *envelope* of G is $\{(v_i, v_j) \mid 0 < i - j \leq b(v_i)\} = \{(v_i, v_j) \mid f(v_i) \leq j < i\}$. The *profile* of G is $|envelope(G)|$. The profile method assumes that all the non-zero entries in A are within an envelope of varying width about the main diagonal. As with the bandwidth method, the problem of finding an ordering that results in minimum profile for G is NP-hard.

A graph G, for which there is an ordering such that all edges within the envelope are present, is defined by Tarjan in [96] as a *dense profile graph.* That is, G is a dense profile graph if and only if there is an ordering of the vertices in G such that if $(v_i, v_j) \in E$ and $i \leq k \leq j$ then $(v_k, v_j) \in E$. The following interesting result is from [96].

Theorem 3.11 *G is a dense profile graph if and only if G is an interval graph.*

This result helps us to visualize what the corresponding matrices of interval graphs look like. An example is given in Figure 3.9. Also, any graph G can be ordered so that the associated graph G' of the envelope of G is an interval graph. Then Algorithm *Interval* can be used to find a minimum height elimination tree of G'. Of course, this will not necessarily be a minimum height elimination tree of G, but if the profile of G is small, then the elimination tree found will probably be close to the optimal one for G.

We now continue by looking at a known subclass of interval graphs called proper interval graphs and examining the relationship between this subclass and Ω.

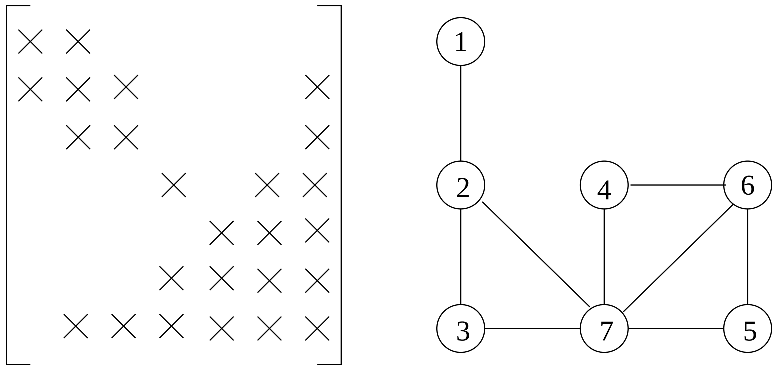


Figure 3.9: A dense profile matrix and its associated graph.

3.7.2 Proper interval graphs

The following theorem by Roberts [90] defines proper interval graphs.

Theorem 3.12 *G is a proper interval graph if and only if G is an interval graph containing no induced copy of* $K_{1,3}$.

The complete bipartite graph $K_{1,3}$ is shown in Figure 3.10 a). We will now show that the class Ω is not comparable to the class of proper interval graphs.

Theorem 3.13 *There exist proper interval graphs that are not in* Ω*, and there exist graphs in* Ω *that are not proper interval graphs.*

Proof: Let G be as shown in Figure 3.10 b). Clearly, $G \in \Omega$. But the black vertices in G induce $K_{1,3}$, so G cannot be a proper interval graph. Thus $\Omega \not\subseteq$ proper interval graphs. Let now H be as shown in Figure 3.10 c). Since H is an interval graph and does not contain an induced copy of $K_{1,3}$, H is a proper interval graph. But H is not in Ω since H has more than two maximal cliques with simplicial vertices. Thus proper interval graphs $\not\subseteq \Omega$. ∎

Theorem 3.13 shows that Ω defines a subclass of interval graphs that is neither a subset nor a superset of proper interval graphs. This is an interesting result since it shows that Ω is a non-trivial subclass of interval graphs.

3.7.3 Open problems

Chordal graphs are a very important class of graphs in sparse Cholesky factorization. We have seen that all filled graphs are chordal and that all chordal graphs have perfect elimination orderings. It would be very interesting to know whether

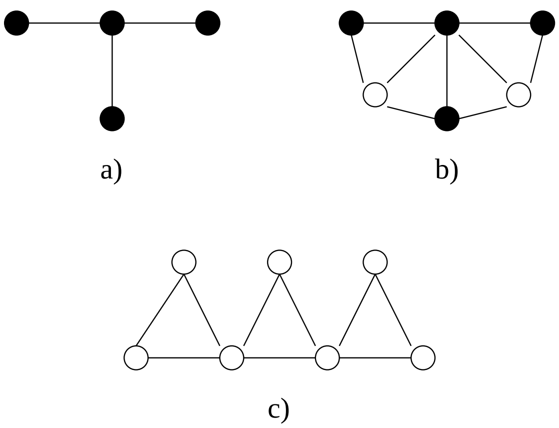


Figure 3.10: a) $K_{1,3}$ b) G in the proof of Theorem 3.13. c) H in the proof of Theorem 3.13.

minimum height elimination trees of chordal graphs can be obtained in polynomial time.

It would also be interesting to know how much fill is introduced when graphs are reordered according to the algorithms that we have introduced. Manne [74] shows that, for a tree G, a minimum height elimination tree of G does not produce more than linear amount of extra fill compared to minimum fill size for G. We have not been able to find a linear bound for the amount of fill for any of the classes that appear in this paper. Even the class Ω, which is the most restricted, seems to possess many of the properties of general graphs that make it difficult to prove a linear bound. For general graphs, it was conjectured that orderings resulting in minimum height elimination trees, do not introduce more than a linear amount of fill. However, Aspvall [3] shows that minimum height elimination tree orderings can result in non-linear fill for general graphs.

In this paper we have given a simple $O(n^3)$ algorithm for finding minimum height elimination trees for interval graphs. An interesting open problem is whether it is possible to develop an algorithm that is significantly better than $O(n^3)$. Our algorithm uses the method of dynamic programming, and there exist techniques for increasing the efficiency of dynamic programming algorithms. We have given a discussion on how one might possibly reduce the time complexity of this particular algorithm. Unfortunately, we have to leave as an open problem to show some appropriate monotonicity property for interval graphs.

Chapter 4

Fill Minimization for Nested Dissection with Minimal Separators

In the previous chapter, we gave a polynomial-time solution for the minimum elimination tree height problem on interval graphs. For elimination trees of minimum height, it is interesting to know how much fill the corresponding ordering produces compared to a minimum fill ordering. For general graphs, the amount of extra fill introduced by minimum height elimination tree orderings is not known. But we know that it can be more than linear [3].

In this chapter, we show that, given a minimum height elimination tree, it is possible to reduce the fill introduced by the corresponding ordering, while still preserving the structure of the elimination tree. For every graph, there exists a minimum height elimination tree which results from a nested dissection algorithm that chooses minimal separators at each step. We study the problem of minimizing the size of the fill for a given nested dissection ordering with minimal separators (not necessarily a minimum height ordering). We show that this problem is NP-hard.

This chapter is based on some of the results presented by Heggernes in [56]. A preliminary version was presented at the IMA Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms, October 1991, Minnesota, USA.

4.1 Introduction

In sparse Cholesky factorization, $A = LL^T$, it is important to find symmetric permutations of the matrix A, such that the Cholesky factor of the permuted matrix is as sparse as possible. Considering the associated graphs of these matrices, the edges in the filled graph $G^+ = G(L + L^T)$ that are not in the original graph $G = G(A)$ are referred to as the fill. Thus orderings that produce small fill are desirable. Minimizing the size of fill is an NP-hard problem [103]. For parallel computations, an important structure is the elimination tree, defined by Schreiber [95], which gives information about the column dependencies in the Cholesky factor. In order to reduce the amount of work in parallel factorization, elimination trees of low height should be computed. Finding a permutation of A that results in an elimination tree of minimum height is also an NP-hard problem [85].

For general graphs, Aspvall [3] shows that minimizing the elimination tree height might increase fill more than linearly compared to minimum fill. For trees, minimum height elimination trees result in at most linear fill, as shown by Manne [74]. We do not know of similar results for other graph classes. In this paper, we show that the amount of fill can possibly be reduced when an elimination tree ordering is already given, and the structure of the elimination tree is to be preserved.

Manne [74] shows that there always exists a nested dissection ordering with minimal separators of 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 components resulting from the removal of a minimal separator. The only requirement is that the separators chosen at each step are minimal. Nested dissection orderings do not define uniquely how the vertices in each separator should be numbered. We will show that the ordering of the vertices in each separator is important for the amount of fill introduced by the ordering.

For any given nested dissection ordering with minimal separators (not necessarily producing a minimum height elimination tree), we define a structure called the separator tree corresponding to the resulting elimination tree. What defines the separator tree and the structure of the elimination tree, is the choice of separators by the given nested dissection ordering. We show that the local ordering of the vertices in each separator does not affect these structures. However, this local ordering does affect the size of the fill produced by the given ordering. Given a nested dissection ordering with minimal separators, we consider the problem of minimizing the fill resulting from this ordering, by reordering each chosen separator and each remaining component locally. Furthermore, we show that in order to solve the global minimization problem, the reordering of each minimal separator can be done independently of the other separators. Finally, we show that the local reordering problem is NP-hard, implying that it is NP-hard to minimize fill for a given nested dissection ordering.

Since a nested dissection ordering numbers the vertices in each chosen separator arbitrarily, the problem we consider can be seen as a tie-breaking problem. It has previously not been usual to consider the issue of tie-breaking for nested dissection in this context. Traditionally, tie-breaking involved in nested dissection algorithms has concerned only the choice of separators [50, 47, 74].

4.2 Definitions

For a given graph $G = (V, E)$, an *elimination ordering* α is a function that numbers the vertices of G according to any permutation of $\{1, 2, ..., |V|\}$. The graph G ordered by α is denoted by G_α, and the vertex that is numbered i by α is denoted by v_i.

For a vertex v in G, the set $adj_G(v)$ is the set of neighbors of v in G, whereas $madj_{G_\alpha}(v)$ is the set of neighbors of v that have higher numbers than that of v according to α. If $S \subset V$, then $adj_G(S) = \{w \notin S \mid v \in S \text{ and } (v, w) \in E\}$. The definition of $madj_{G_\alpha}(S)$ is analogous.

The *filled graph* G_α^+ is obtained from G_α by the following *elimination game*: Let $G' = G_\alpha$. For $i = 1, 2, ..., |V|$, add edges to G' such that $adj_{G'}(v_i)$ induces a clique in G', and remove v_i from G'. The added edges are called the *fill edges*. The filled graph G_α^+ is a result of adding, to G_α, all the fill edges resulting from the elimination game. Note that $madj_{G_\alpha^+}(v_i)$ induces a clique in G_α^+ for each v_i. If $G_\alpha^+ = G_\alpha$, then α is called a *perfect* elimination ordering. When α is clear from the context, we will use only G and G^+.

A graph G is *chordal* if, on every cycle of length at least four, there is an edge between two nonconsecutive vertices of the cycle. Chordal graphs are exactly the class of graphs with perfect elimination orderings [36]. All filled graphs are chordal [91]. A *simplicial* vertex in G is a vertex whose neighbors induce a clique in G. The elimination of a simplicial vertex does not cause any fill edges. Every chordal graph is either complete, or has two nonadjacent simplicial vertices [23]. Chordality is a hereditary property.

Given G and α, the resulting *elimination tree* is a rooted tree, where v_i is the parent of v_j if $i = \min\{k \mid v_k \in madj_{G_\alpha^+}(v_j)\}$. If v_s and v_t lie on the same path between a leaf and the root, and $s > t$, then v_s is an *ancestor* of v_t, and v_t is a *descendant* of v_s. The complete subtree of T rooted at v is denoted by $T[v]$. For disjoint subtrees $T[u]$ and $T[v]$, and for any two vertices $y \in T[u]$ and $z \in T[v]$, Schreiber [95] shows that (y, z) is not an edge in G_α^+. Thus elimination of vertices in a subtree does not affect the vertices in other disjoint subtrees.

A *separator* in G is a set of vertices whose removal from G disconnects the graph. A separator S is *minimal* if no proper subset of S is a separator in G.

A *Nested Dissection ordering with Minimal Separators* (an NDMS-ordering) chooses a minimal separator S of the given graph G, and numbers the vertices in S arbitrarily with numbers from $\{|V| - |S| + 1, |V| - |S| + 2, ..., |V|\}$. Then this process is repeated recursively for each connected component of the subgraph of G induced by $V - S$, until the remaining components are cliques. The vertices in the remaining cliques are numbered analogously. We will show that the chosen minimal separators correspond to cliques in the filled graph. Accordingly, in the resulting elimination tree, each chosen minimal separator induces a path, where the first chosen separator appears at the top, and branching occurs only between separators. For every graph G, there exists an NDMS-ordering that results in an elimination tree of minimum height [74]. However, it is NP-hard to find such orderings [85].

Given any NDMS-ordering α of G, we define the *separator tree* of G_α to be a rooted tree whose nodes are the separators chosen by α. The relative placement of the nodes of the separator tree follows the placement of the chosen separators in the corresponding elimination tree. The leaves of the separator tree correspond to the remaining cliques of G when all chosen separators are removed. The nodes of a separator tree are thus sets of vertices from G, and are called *s-nodes*. An example is given in Figure 4.1. The original graph G_α is given in 4.1 a), and the resulting filled graph is given in Figure 4.1 b). Note that α is an NDMS-ordering with $\{e, f, g\}$ as the chosen minimal separator. The resulting elimination tree, which is of minimum height for G, is shown in Figure 4.1 c), and the corresponding separator tree is indicated by dashed lines.

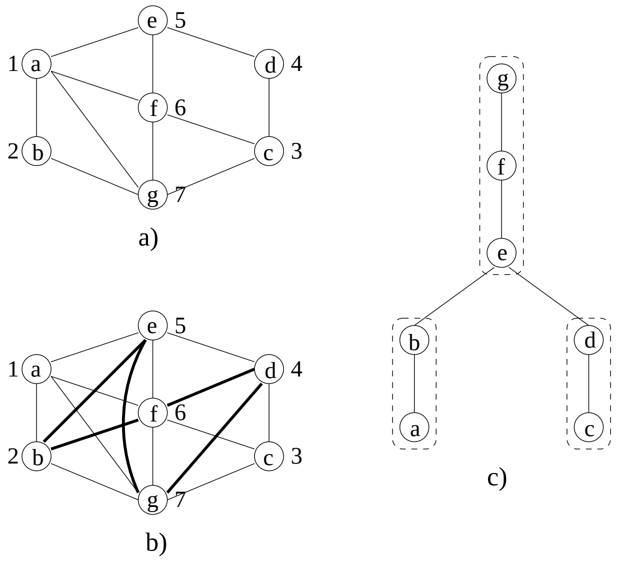


Figure 4.1: a) An ordered graph G_α, where the numbers indicate the elimination ordering α on G. b) The filled graph G_α^+. c) The resulting elimination tree, where the s-nodes of the separator tree are highlighted.

4.3 A local minimization problem for each s-node

We consider the problem of reordering the vertices in each chosen separator, and in each remaining clique, of a given NDMS-ordering to minimize the resulting fill. By the definition of separator trees, this is equivalent to reordering the vertices in each s-node of the corresponding separator tree. We start this section by showing in an example that the fill can indeed be reduced for a given separator tree. Consider the graph G_α shown in Figure 4.1. In Figure 4.2, another elimination ordering of the same graph is given. This ordering corresponds to a local reordering of the vertices of the s-nodes in the separator tree. The global NDMS-ordering of the s-nodes is not changed. The resulting elimination tree structure, and the separator tree are the same as for G_α. However the number of fill edges is reduced by more than half.

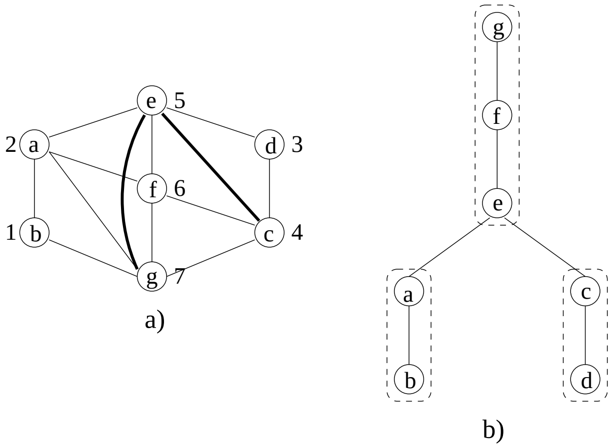


Figure 4.2: a) The filled graph G^+_β that results from a reordering β of the graph G in Figure 4.1. c) The resulting elimination tree, where the separators are highlighted.

Let α be a given NDMS-ordering on G, and let T and ST be the resulting elimination and separator trees, respectively. In the following discussion, we will show how the local reordering of the vertices in an s-node of ST affects the separator tree and the fill introduced. In the rest of this paper, unless otherwise stated, G, T, and ST will be as described above, G being initially ordered by an NDMS-ordering. Results cited from other authors are valid for all elimination orderings, whereas our results only concern NDMS-orderings. For our first proof, we need the following result from Liu [71].

Lemma 4.1 *For any elimination ordering α on G, the set $adj_G(T[v]) \cup \{v\}$ induces a clique in G^+.*

Now, we show some results on the s-nodes in ST.

Lemma 4.2 *Each s-node in ST induces a clique in the filled graph G^+.*

Proof: Since the leaf s-nodes correspond to subgraphs of G that cannot be separated further, these subgraphs are cliques in G, and hence in G^+. 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 therefore lowest in T among the vertices in S), and let y be a child of x in T. 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 $S \subseteq adj_G(T[y])$, and by Lemma 4.1, S induces a clique in G^+. ∎

In the next lemma, we show formally that the local ordering of the vertices in each chosen separator of an NDMS-ordering, does not affect the resulting separator tree.

Lemma 4.3 *Changing the relative ordering of the vertices in an s-node locally, does not result in any changes in ST or in the structure of T.*

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. Therefore, the separator tree remains the same. It follows from Lemma 4.2 that the vertices in an s-node S induce a path in the elimination tree regardless of their relative ordering local to S. Accordingly, the elimination tree structure does not change because of such a local reordering. ∎

Lemma 4.3 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 can be reduced by such a reordering. We divide the fill that is caused by the elimination of vertices in S, into four disjoint groups:

1. Fill that occurs within S.

2. Fill that occurs within another s-node.

3. Fill that occurs between other s-nodes.

4. Fill that occurs between S and other s-nodes.

Some of this fill occurs regardless of the local ordering of the vertices in an s-node. It is clear from Lemma 4.2 that changing the relative ordering of the vertices in S cannot result in any changes in fill of Type 1 or 2. In the following discussion, we will study fill edges of Type 3 and 4.

First we need the following theorem from Rose, Tarjan, and Lueker [92]. This result gives necessary and sufficient conditions for fill edges to occur in general.

Theorem 4.1 *For any elimination ordering α on G, the edge (v_i, v_j) is an edge of G^+ if and only if (v_i, v_j) is an edge of G, or there exists a path $v_i = v_{k_1}, v_{k_2}, ..., v_{k_{l+1}} = v_j$ in G, such that $k_m < \min(i, j)$, for $2 \leq m \leq l$.*

Lemma 4.4 *Changing the relative ordering of the vertices in an s-node S locally, does not result in any changes in the fill that occurs between other s-nodes (Type 3).*

Proof: We need only to consider those s-nodes that lie on the path between S and the root s-node. This is because whatever lies below S is eliminated before S, and the s-nodes that lie in disjoint subtrees cannot be affected by the elimination of S. Let S_i and S_j be two s-nodes that are ancestors of S in the separator tree, and let v_i and v_j be two vertices, where $v_i \in S_i$, and $v_j \in S_j$. By Theorem 4.1, (v_i, v_j) is an edge of the filled graph if and only if (v_i, v_j) is already an edge of G, or there exists a path $v_i = v_{k_1}, v_{k_2}, ..., v_{k_{l+1}} = v_j$, such that all subscripts in $\{k_2, ..., k_l\}$ are less than $\min(i, j)$. If (v_i, v_j) is not an edge of G and such a path does not exist, then a reordering of the vertices in S cannot create such a path, since all the vertices in S are numbered lower than $\min(i, j)$, following the definition of elimination and separator trees. By the same argument, if such a path does exist, then it will exist also after a reordering of the vertices in S. ■

The only vertices whose elimination may result in fill edges from S to higher numbered vertices, are the vertices in S and in the descendants of S in ST. It should now be clear from the previous lemmas that, by reordering S locally, we can only reduce the number of fill edges between S and its higher numbered neighbors in G^+, that are caused by the elimination of vertices within S. Thus we can restrict our attention to fill edges of Type 4, particularly the fill edges between S and $madj_{G^+}(S)$. In the example given in Figure 4.2, we have seen that such a reordering can reduce the number of fill edges. A close study of the same example shows that only fill edges of Type 4 are affected. Consequently, the vertices in the root s-node of ST can be ordered arbitrarily, since their elimination cannot create fill to other s-nodes.

Lemma 4.4 and the discussion above intuitively 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, given the separator tree, the s-nodes can be reordered in an arbitrary order, or in parallel.

For a given s-node S, we will call the set $(madj_{G^+}(S) - madj_G(S))$ the *fill neighbors of S*. We will see in Lemma 4.5 that we do not have to consider the fill neighbors of S when we reorder S, since these will have edges in G^+ to all the vertices in S regardless of the relative ordering of the vertices in S. For the proof of Lemma 4.5, we need the following theorem from Liu [68].

Theorem 4.2 *Let α be any elimination ordering on G, and let $i > j$. The edge (v_i, v_j) is in G^+ if and only if (v_i, v_j) is an edge of G, or v_j is an ancestor of some node v_k in T, where (v_i, v_k) is an edge of G.*

Lemma 4.5 *If there is a fill edge in G^+ from a vertex v_j in an s-node S to a higher numbered vertex v_i, and this edge is caused by the elimination of a descendant of S in ST, then every vertex in S has an edge in G^+ to v_i.*

Proof: Let S be any s-node in ST that is not a leaf. Since there is a fill edge between v_i and v_j caused by a vertex v_k in a descendant s-node of S, there must be an edge in the filled graph between v_k and v_i. Then by Theorem 4.2, either (v_i, v_k) is an edge in G, or there must be a vertex v_l which is a descendant of v_k in T $(i > j > k > l)$ such that (v_i, v_l) is an edge in G. By the same theorem, every vertex in S must also have a fill edge to v_i in G^+, since the vertices in S are all ancestors of v_k in T. ∎

The set $madj_{G^+}(S)$ is constructed by adding to $madj_G(S)$ all the fill neighbors of S, and the fill neighbors of S are decided by the elimination of the descendants of S. The elimination of S does not add any new vertices (fill neighbors) to $madj_{G^+}(S)$, but it may add new fill edges between S and $madj_{G^+}(S)$. Our aim is to minimize the number of such fill edges. By Lemma 4.5, the fill neighbors of S have edges to all the vertices in S in the filled graph, regardless of the local ordering of the vertices in S. Therefore, no further fill can be introduced, and no fill can be reduced between S and the fill neighbors of S by reordering the vertices in S. We can thus restrict our minimization problem to minimizing the number of fill edges between S and $madj_G(S)$ introduced by the elimination of the vertices in S. Between S and $madj_G(S)$, there may be fill edges created prior to the elimination of S. However, if any vertex v in $madj_G(S)$ has a fill edge to a vertex in S created by the elimination of descendants of S, then by Lemma 4.5, v has edges in the filled graph to all the vertices in S. Hence no fill can be added or reduced between S and v by reordering the vertices in S. Therefore, we can exclude such vertices from the neighborhood of S that we will consider. These vertices are identified by Theorem 4.2; they are the vertices in $madj_G(S)$ that also belong to $madj_G(C)$, for a descendant C of S in ST.

Let $adj'(S) = madj_G(S) - \cup\{madj_G(C) | C \text{ is a descendant of } S\}$. From the discussion above, it is clear that our minimization problem can be restricted to the edges between S and $adj'(S)$. Furthermore, prior to the elimination of S, no fill edges exist between these two sets. Thus, before we start reordering S, the only edges between S and $adj'(S)$ we need to consider, are the original edges in G between these sets. Since all s-nodes are disjoint, this will allow us to reorder the vertices in an s-node independently of the other s-nodes.

By Lemma 4.2, S is a clique in the filled graph prior to its elimination. We have seen that a local reordering of S does not affect the fill within $adj'(S)$. Clearly, the edges between the vertices of $adj'(S)$ have no effect on the size of the fill introduced between S and $adj'(S)$ by the elimination of S. Thus, we can regard $adj'(S)$ as a clique in the subgraph of G^+ that we work with. (Actually, it can be shown that $adj'(S)$ is indeed a clique in the filled graph). In order to give a formal description of our local reordering problem, we need the following definitions.

Definition 4.1 *Given S, we define G_S to be the bipartite graph $(S, adj'(S), E)$, where E contains only those edges in G that are between the vertices in S and their neighbors in $adj'(S)$. (See Figure 4.3 b).)*

Definition 4.2 *Let B=(P,Q,E) be a bipartite graph. We define C(B) 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(B). (See Figure 4.3 c).)*

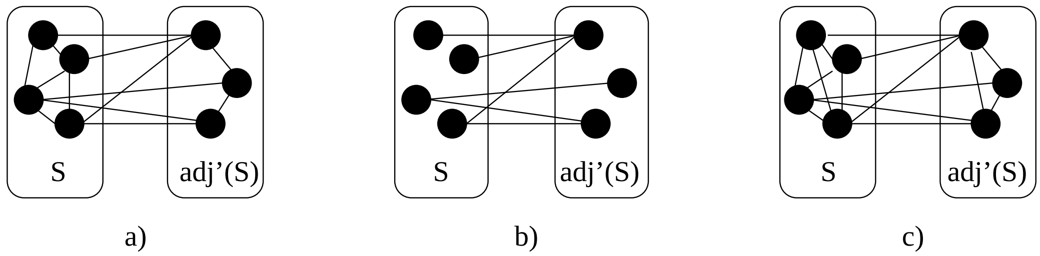


Figure 4.3: a) The subgraph of G induced by the vertices in S and $adj'(S)$. b) G_S. c) $C(G_S)$.

Note that the graph $C(G_S)$ is completely determined by the edges in the original graph G, and does not depend on the other s-nodes in ST. The following algorithm *Reorder* describes how a graph G can be reordered to reduce fill, given a separator tree for G. It also gives a formal 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;*
    Δ: Find an ordering of the vertices in S that minimizes the fill introduced
      by their elimination in $C(G_S)$;
    mark *S;*
  end-while;
end;

Theorem 4.3 *Algorithm Reorder solves the problem of minimizing the fill resulting from a given NDMS-ordering described by its corresponding separator tree.*

Proof: Follows from the lemmas and the discussion above. ∎

We have shown that the problem of minimizing the fill for a given NDMS-ordering decomposes into smaller subproblems Δ, which can be solved independently for each s-node. The independence of the subproblems is important since it implies that the s-nodes can be studied in any order. This is a desirable property for several reasons. First, it means that, during the nested dissection algorithm, each minimal separator can be locally ordered to minimize fill, either as soon as it is chosen, or at a later step. Global fill minimization for the resulting NDMS-ordering will be provided when all s-nodes are locally ordered. Second, if the local reordering problem is solved by a postprocessing after an NDMS-ordering is given, then parallel computation can be utilized to reorder several s-nodes simultaneously. In the next section, we show that Δ, although more restricted and structured than the general minimum fill problem, is still NP-hard.

4.4 The local minimization problem Δ is NP-hard

In this section we are going to show that the problem Δ, described in the previous section, is NP-hard. In order to show this, we need some results from Yannakakis who shows in [103] that the general minimum fill problem is NP-hard. Lemma 4.6 and Theorem 4.4 are from [103], and are stated without proofs.

Definition 4.3 *A bipartite graph (P,Q,E) is a chain graph if the neighborhoods of the vertices in P form a chain; i.e., there is a bijection* $\pi : \{1, 2, ..., |P|\} \leftrightarrow P$ *such that* $adj(\pi(1)) \supseteq adj(\pi(2)) \supseteq ... \supseteq adj(\pi(|P|))$. *(Then the neighborhoods of the vertices in Q form also a chain.)*

Lemma 4.6 *Let B be a bipartite graph. The graph C(B) is chordal if and only if B is a chain graph.*

Theorem 4.4 *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.*

Lemma 4.6 and Theorem 4.4 show that the general minimum fill problem is NP-hard for a special class of graphs, implying that it is also NP-hard for general graphs. In order to prove that Δ is NP-hard, we need to show the following result on this special graph class.

Lemma 4.7 *Let* $B = (P, Q, E)$ *be a bipartite graph. Then* $C(B)$ *is chordal if and only if* $C(B)$ *has a perfect elimination ordering that eliminates all the vertices in* P *before the vertices in* Q.

Proof: *(if)* Since $C(B)$ has a perfect elimination ordering, it must be chordal.

(only if) Since $C(B)$ is chordal, it has a perfect elimination ordering. We need to show that there exists a perfect elimination ordering of $C(B)$ which eliminates the vertices of P before the vertices of Q. We know by Lemma 4.6 that B is a chain graph. Let $p = |P|$ and let π be an ordering of P such that $adj_B(\pi(1)) \supseteq ... \supseteq$

$adj_B(\pi(p))$. The neighborhood of $\pi(p)$ in $C(B)$ is $adj_{C(B)}(\pi(p)) = adj_B(\pi(p)) \cup (P - \pi(p))$. This is a clique in $C(B)$, since P and Q are cliques, and each vertex in P has edges to all the vertices of Q that are neighbors of $\pi(p)$. Thus $\pi(p)$ is simplicial in $C(B)$, and can be eliminated first without introducing fill. When $\pi(p)$ is eliminated, we can find a new simplicial vertex in P (for example $\pi(p-1)$) in the same way, since the chain property is hereditary. Accordingly, if the vertices in P are eliminated in the following order, no fill will be introduced: $\pi(p), \pi(p-1), ..., \pi(1)$. Thus we can eliminate all the vertices in P before the vertices in Q without introducing any fill between P and Q. When all the vertices in P are eliminated, the resulting subgraph of $C(B)$ induced by Q is a clique, and thus can be eliminated without introducing any fill. The described ordering is a perfect elimination ordering of $C(B)$ that orders the vertices in P before the vertices in Q. ■

Theorem 4.5 *Δ is NP-hard.*

Proof: For any given bipartite graph $B = (P, Q, E)$, we will construct a bipartite graph $D = (T, U, E')$ such that:

1. $C(D)$ is an *instance* of, i.e. a valid input for, the problem Δ, and
2. B is a chain graph if and only if $C(D)$ has a perfect elimination ordering that eliminates the vertices in T before the vertices in U.

We must ensure that the graph $C(D)$ is a valid input for the problem Δ. For Δ, the bipartite graph G_S is defined by the edges in G, for an arbitrary s-node S of G, where G is an arbitrary general graph. The only restriction that G_S has, compared to general bipartite graphs, is that $adj'(S)$ has no isolated vertices in G_S. Thus given B, we must construct D such that U has no isolated vertices.

Given $B = (P, Q, E)$, let I be the set of isolated vertices in Q. Construct $D = (T, U, E')$, where $T = P$, $U = Q - I$, and $E' = E$. Thus U has no isolated vertices in D.

By Lemmas 4.6 and 4.7, B is a chain graph if and only if $C(B)$ has a perfect elimination ordering that eliminates the vertices in P before the vertices in Q. Since no vertex of P has edges to the vertices of I in $C(B)$, $adj_{C(B)}(P) = adj_{C(D)}(T)$. Therefore, any perfect elimination ordering of the vertices of P in $C(B)$ is a perfect elimination ordering of the vertices of T in $C(D)$. Accordingly, $C(B)$ has a perfect elimination ordering that eliminates the vertices in P first if and only if $C(D)$ has a perfect elimination ordering that eliminates the vertices in T first. Thus B is a chain graph if and only if $C(D)$ has a perfect elimination ordering that orders the vertices in T first.

By Theorem 4.4, it is NP-hard to find the minimum number of edges that the elimination of T introduces in $C(D)$. Since $C(D)$ is an instance of Δ, this implies that Δ is NP-hard. ■

We have shown that the local ordering problem for each s-node is an NP-hard problem. This implies that the global problem of minimizing the fill resulting from a given NDMS-ordering is also NP-hard.

4.5 Conclusion

In this paper, we studied the problem of minimizing fill while preserving the global ordering of the chosen minimal separators in a given NDMS-ordering. We showed that the solution of this problem decomposes into independent local minimization problems for each separator. Finally, we proved that finding an optimal local reordering of each separator is an NP-hard problem, implying that it is NP-hard to minimize fill for a given NDMS-ordering.

Although the local problem studied in this chapter is NP-hard, we have shown that each separator in the separator tree can be reordered independently of the other separators. If the separator tree consists of only small s-nodes, then the s-nodes can be reordered optimally within reasonable time. For parallel computations, this means that several s-nodes can be reordered simultaneously.

Regarding the NP-hardness proof of the previous section, some interesting observations can be done. Both the NP-completeness proof of the minimum fill size problem by Yannakakis [103], and the NP-completeness proof of the minimum elimination three height problem by Pothen [85], utilize the same construction from a bipartite graph $B = (P, Q, E)$ to the graph $C(B)$ as explained in the previous sections. Let $|P| = p$, $|Q| = q$, and $|E| = e$. The matrices corresponding to graphs of type $C(B)$ have the following structure (see also Figure 4.3 c) for comparison):

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

Here, $A_{11} \in \mathbf{R}^{p\times p}$ and $A_{22} \in \mathbf{R}^{q\times q}$ are completely dense, and $A_{12} = A_{21}^T$, where $A_{12} \in \mathbf{R}^{p\times q}$ has e elements. The mentioned NP-completeness proofs show that, for the graphs of matrices which can be partitioned as described, the problem minimizing the elimination tree height, the problem of minimizing the size of the fill, and our problem Δ are NP-hard.

For given NDMS-orderings, the reorderings we studied preserve the structure of the elimination tree, and change the size of the fill. In the next chapter, an opposite approach will be explained for any given initial ordering. We will, among other things, consider certain reorderings that preserve the size of the fill while possibly changing other properties of the filled graph, like the elimination tree structure. We mentioned that the problem we considered in this chapter can be regarded as a tie-braking problem for the nested dissection algorithm. Tie-breaking is a very important problem for the minimum degree algorithm. In the next chapter, we will consider tie-breaking strategies for the minimum degree algorithm, with respect to reducing the number of operations required by sparse QR-factorization.

Chapter 5

Finding Good Column Orderings for Sparse QR Factorization

In Chapter 3, we considered the problem of minimum elimination tree height. The elimination tree structure is also very important in multifrontal QR factorization, as will be explained in this chapter. We will in this chapter concentrate on reducing the number of operations required by sparse QR factorization. The operation count is partly dependent on the size of the fill and the structure of the elimination tree. However, as we will show, the efficiency of sparse QR factorization is also dependent on other very important measures: the size of the intermediate fill, and the size and the structure of the frontal matrices. Traditionally, a suitable column ordering for sparse QR factorization of a full rank matrix $A \in \mathbf{R}^{m \times n}$ with $m \geq n$, has been found through symmetric sparsity analysis of the graph $G(A^T A)$. In this chapter, we show that, because of the properties of intermediate fill and frontal matrices, methods studying directly the structure of the bipartite graph of A, $B(A)$, should be used. All necessary graph terminology will be explained. We propose heuristics for choosing good equivalent reorderings, and strategies for tie-breaking in the minimum degree algorithm in order to achieve low operation count in sparse QR factorization.

This chapter presents an extended refinement of the following research paper: [57] P. Heggernes and P. Matstoms, *Finding Good Column Orderings for Sparse QR Factorization*. The authors would like to thank John R. Gilbert for useful discussions in Bergen, and for his valuable comments. We are also grateful to Bengt Aspvall, Åke Björck, and our universities, for making our trips between Bergen and Linköping possible.

5.1 Introduction

Least squares problems play an important role in a wide range of scientific applications. Parameter estimation in mathematical modeling is one important example. Many applications lead to sparse linear least squares problems. The coefficient matrix is then assumed to be large with relatively few nonzero entries. The introduction of multifrontal methods have made direct methods based on sparse QR factorization attractive and competitive to previously recommended alternatives, as explained by Matstoms [78]. In this paper, we consider the QR factorization,

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix},$$

of a *large* and *sparse Strong Hall* matrix $A \in \mathbf{R}^{m \times n}$ of *full column rank*, where $m \geq n$. The column ordering on A is decisive for the efficiency of the factorization. Researchers have concentrated on finding column orderings that result in as sparse R as possible. George and Heath [40] have observed that a suitable column ordering of A can be found by symmetric fill-reducing analysis of A^TA. Efficient methods, like minimum degree and nested dissection, exist for fill reduction in symmetric factorization. Moreover, it is shown by George and Liu [43] that minimum degree analysis of A^TA can be done directly on A, without having to compute A^TA. Efficient standard methods can thus be used to compute column orderings that normally give low fill in R.

In this paper, we focus on column orderings for sparse QR factorization, and we show, by examples, that reducing the fill in R is not enough to achieve efficient factorization. Other factors, like the size and the structure of intermediate fill, and the size and the structure of the frontal matrices for multifrontal factorization, are also important for the number of operations required by the factorization process. Even for traditional sparse Householder QR factorization, the size and the structure of intermediate fill can vary considerably among column orderings that result in the same number of nonzero elements in R. In order to consider other criteria than the amount of fill in R, studying the structure of A^TA is not enough. A lot of information about A is lost when A^TA is formed, and this information from A is important in order to decide good column orderings for sparse QR factorization. Traditionally, the reason for doing a fill-reducing ordering of A^TA directly on A, has been solely to reduce the computation time. As we will see, another important reason is that, in this way, the information from A can be used in addition to the information from A^TA. This is necessary, for example, when trying to minimize the amount of intermediate fill, or the size of the frontal matrices.

Following the criteria mentioned above, we first concentrate on finding column orderings that reduce the number of operations required by sparse QR factorization, without increasing the number of nonzeros in R. Equivalent reorderings have this property, and they are described formally later in this paper. Given any initial ordering, presumably a fill-reducing ordering, of the matrix A, equivalent reorderings can result in better column orderings than the initial ordering of A. Numerical tests show that considerable improvements, with respect to the number of operations, can be achieved by good equivalent reorderings. We propose heuristics for finding good equivalent reorderings, and explain the ideas behind these heuristics. These heuristics are based on the structural information from A. Numerical results, comparing these heuristics to the initial ordering, and to the best random equivalent reordering, are presented. George and Ng [45] show a connection between structural Householder QR factorization and structural LU factorization. For this purpose, it is interesting to find good orderings not only for the multifrontal algorithm, but also for the traditional Householder algorithm. We therefore design and compare our heuristics regarding both these algorithms.

As opposed to the two-step approach of a good initial ordering with a proceeding equivalent reordering, we also consider a one-step approach directly applied on the original matrix A. Our focus is now on minimum degree analysis of A^TA done directly on A. In the minimum degree algorithm, a vertex of minimum degree is chosen at each step of the algorithm, and ties occur when there are several vertices having the minimum degree at each step. The produced ordering is dependent on how ties are resolved. The standard minimum degree algorithm resolves ties arbitrarily. However, the structure of A is important to consider when designing good tie-breaking strategies. This paper proposes methods for tie-breaking in minimum degree, which utilize the structure of A to produce column orderings that are suitable for sparse QR factorization. We compare the orderings produced by our tie-breaking strategies with orderings resulting from standard minimum degree, with respect to the number of operations required by the multifrontal and the traditional Householder QR factorization algorithms. Our numerical tests indicate that the gain from these tie-breaking strategies is very good, and higher than the gain from equivalent reorderings.

This paper is intended to give a solid background in QR factorization and multifrontal methods. Section 5.2 contains a summary of recent research in this area, along with a discussion about the importance of A compared to A^TA in sparsity and efficiency analysis. Equivalent reorderings, and the relation to graph theory are explained in Section 5.3, and a more specific study of equivalent reorderings and QR factorization is found in Section 5.4. Tie-breaking strategies for minimum degree in QR factorization are explained and discussed in Section 5.5. Finally, Section 5.6 contains concluding remarks, open problems, and plans for future research.

The numerical results presented in this paper are achieved through numerical tests in MATLAB versions 4.2a and 4.2c. Throughout the paper, simple MATLAB notations are used, which the reader is assumed to be familiar with. Table 5.1 shows the test matrices we use for our numerical experiments. Some of the matrices are from the Harwell-Boeing collection (Duff, Grimes and Lewis [26]). The others are artificially made by concatenating two Harwell-Boeing matrices. The matrices that are marked with an asterisk do not have the assumed Strong Hall Property, which will be explained in Section 5.2.

Matrix	m	n	nnz	Description
ABB313	313	176	1557	Sudan survey data
ASH219	219	85	438	Geodesy problem - Holland survey data
ASH331	331	104	662	Geodesy problem - Scotland survey data
ASH608	608	188	1216	Geodesy problem - England survey data
ASH958	958	292	1916	Geodesy problem - U. K. survey data
WL1033	1033	320	4732	Gravity-meter observations
WL1850	1850	712	8758	Gravity-meter observations
WL1252*	1252	320	5170	WL1033 , ASH219
WL1364*	1364	320	5394	WL1033 , ASH331
WL1641*	1641	320	5948	WL1033 , ASH608
WL1991	1991	320	6648	WL1033 , ASH958
WL2808	2808	712	10674	WL1850 , ASH958

Table 5.1: Test matrices used throughout this paper. Number of rows is m, number of columns is n, and number of nonzero elements is nnz for each matrix.

5.2 Sparse QR factorization

Although the method of normal equations is probably the most common method for solving dense linear least squares problems, from a numerical and computational point of view, Householder QR factorization is clearly preferable. Only well-conditioned problems can generally be satisfactorily solved by the normal equations. The situation is, however, less clear in the sparse case, where also sparsity aspects, in particular the introduction of fill, must be taken into account. Excessive fill leads to performance degradation in terms of increased execution time and storage requirements.

Among direct methods for sparse least squares, the main alternatives are the normal equations, the augmented system method, and different variants of QR factorization. Methods based on QR factorization have, during the last decade, become highly interesting and recommended for sparse least squares problems (Matstoms [78]). We should keep in mind that, until the early 80s, QR factorization was rejected for general sparse problems (Duff and Reid [27]). The main explanation for why QR factorization has become useful, is the multifrontal technique. This technique makes it possible to use Householder transformations efficiently also for sparse matrices. Another alternative is using Givens rotations; they can, in a very flexible way, eliminate selective nonzero entries, and should therefore be the natural choice for sparse matrices. Householder transformations are, however, from a computational point of view more attractive. First, dense QR factorization by Givens rotations is more costly than QR by Householder transformations. Second, methods using Givens rotations rely on a suitable row ordering of the matrix. A good row ordering may be difficult and expensive to find. Third, Householder transformations are more easily vectorized.

In this section, we show how the multifrontal technique, originally proposed for symmetric sparse linear systems by Duff and Reid [28], can be used for sparse QR factorization. Let $A \in \mathbf{R}^{m \times n}$ be a large and sparse matrix of full rank. The described multifrontal algorithm uses Householder transformations to compute the upper triangular factor $R \in \mathbf{R}^{n \times n}$ in

$$Q^T A = \begin{pmatrix} R \\ 0 \end{pmatrix}. \tag{5.1}$$

The orthogonal factor $Q \in \mathbf{R}^{m \times m}$ is assumed not to be stored. In the solution of least squares problems, $\min_x ||Ax - b||_2$, the vector $Q^T b$ may be required, but is assumed to be computed during the factorization. In practice, this is implemented by simply merging the right-hand side vector b as an extra column to the right of A. Orthogonal transformations of A are then implicitly applied also to b. It should be mentioned that an alternative method based on QR factorization exists, making least squares problems possible to solve without using Q. In this case, an initial solution is computed by the semi-normal equations, $R^T R x = A^T b$, and then improved by one step of iterative refinement in fixed precision. Repeated least squares problems with a fixed matrix A, but with different right-hand sides, can thus be solved without refactorization.

If A has full column rank, then the symmetric matrix $A^T A$ is positive definite and has a unique lower triangular Cholesky factor L, $A^T A = LL^T$. From the uniqueness of L, it is easily shown that the triangular factor R in (5.1), except for possible sign differences of the rows, equals the Cholesky factor L^T of $A^T A$. In particular, the two matrices R and L^T have the same nonzero pattern. This relation has some important theoretical and practical implications, that will be used throughout the paper.

The above observation suggests that the sparsity pattern of R can be predicted by symbolic Cholesky factorization of $A^T A$. A priori estimation of the fill makes it, for example, possible to store R in a static data structure. Another application is to compute structural information to be used for the subsequent numerical factorization. Sparsity analysis of $A^T A$ may, however, strongly overestimate the fill. A dense row in an otherwise sparse matrix A, for example, makes $A^T A$ structurally full, although R and L may be sparse. Thus, further structural analysis becomes useless. One example is the following matrix A, where Q is the identity matrix, and

R equals the matrix A itself:

$$A = \begin{pmatrix} x & x & x & \dots & x \\ & x & & & x \\ & & x & 0 & x \\ & & & \ddots & \vdots \\ & & & & x \end{pmatrix}. \tag{5.2}$$

Tighter nonzero predictions of R can be obtained by symbolic Givens rotations or by Householder transformations of A. In these cases, fill is structurally introduced according to the following propositions:

Proposition 5.1 *(Fill in sparse Householder QR factorization)*
In the elimination of column k, each column in the remaining unreduced part of the matrix having structurally nonzero inner product with column k, takes the sparsity pattern of the union of itself with column k.

Proposition 5.2 *(Fill in sparse Givens QR factorization)*
The sparsity patterns of two rows involved in a Givens transformation are replaced by the union of their nonzero patterns.

George and Heath [40] show that Givens prediction is always at least as good as symbolic Cholesky factorization. The same is true for symbolic Householder prediction, as shown by Manneback [76]. Altogether this can be summarized as follows:

$$\text{struct(R)} \subseteq \left\{ \begin{array}{l} \text{pred(Householder on } A) \\ \text{pred(Givens on } A) \end{array} \right\} \subseteq \text{pred(Cholesky on } A^T A).$$

The matrix $A \in \mathbf{R}^{m \times n}$ and its bipartite graph have the *Strong Hall Property* (SHP) if every $m \times k$ submatrix, for $1 \leq k < n$, has at least $k+1$ nonzero rows. Coleman, Edenbrandt, and Gilbert [18] show that, if A has the SHP, then symbolic Cholesky factorization, and also symbolic Givens and Householder factorizations, correctly predict the nonzero structure of R. By a reordering scheme studied by Dulmage and Mendelsohn [29, 30, 31], a general matrix can be permuted into block triangular form with diagonal blocks having the SHP. In the solution of a least squares problem, where the coefficient matrix has this block triangular form, only the diagonal blocks need to be factorized. It follows that in practical applications, it can always be assumed that the considered matrix has the SHP. This is also an assumption made throughout this paper. Dulmage-Mendelsohn decomposition of the matrix given in (5.2), which does not have the SHP, leads to 1×1 diagonal blocks.

Let P_r and P_c be row and column permutations of A. Since $(P_r A P_c)^T (P_r A P_c) = P_c^T (A^T A) P_c$, it is clear that the row ordering P_r of A has no mathematical influence on R. It should, however, be stressed that the row ordering may indeed be important for the intermediate fill; in particular if Givens rotations are used. Intermediate fill here refers to temporary nonzero entries that are introduced, but later annihilated, by the algorithm. Such fill has no influence on the final result, but may increase both the execution time and the storage requirements. Moreover, we conclude that the column ordering P_c plays the same role for the fill in R as a symmetric permutation in sparse Cholesky factorization does. The column ordering must therefore be chosen with great care.

In [40], George and Heath utilize the above relation between QR and Cholesky factorization. They observe that the problem of finding a good column ordering for QR factorization of A can be replaced by the corresponding symmetric permutation problem for sparse Cholesky factorization of A^TA. Thus, well-known and efficient heuristics for symmetric matrices, like the minimum degree algorithm, can be used. In their proposed algorithm, George and Heath form the nonzero structure of A^TA, pass it to a symmetric ordering heuristic, and return to the numerical factorization step where the computed column ordering of A is used. This observation and a proposed efficient scheme for sparse QR factorization by Givens rotations, make the paper by George and Heath an important starting point for today's algorithms. References on the way from the paper of George and Heath to modern multifrontal methods include Liu [69], George and Liu [42], Lewis et al. [67], Puglisi [87], and Matstoms [78].

There are essentially three important features characterizing the multifrontal method for sparse QR factorization. First, flexibility in the elimination order of the columns is provided by the underlying elimination tree. This tree gives information about structural dependencies between the columns in A. This information is of particular interest in a parallel setting, where independent columns can be eliminated in parallel. By elimination of columns, we formally mean elimination of the nonzero subdiagonal elements in the column. Second, the elimination of columns in A is performed as a sequence of *dense* subproblems. Standard Householder transformations, implemented by efficient dense linear algebra, can thus be employed and high performance can be expected on, for example, vector architectures. Third, the multifrontal algorithm avoids a lot of intermediate fill. This effect is illustrated in Figure 5.1, where the total amount of memory used in each elimination step is shown for the multifrontal method and for an algorithm based on the traditional Householder technique.

In the traditional Householder algorithm, columns of A are systematically eliminated from left to right. Exactly one Householder transformation is used for each column eliminated in A. After the first k elimination steps, the partially eliminated matrix equals

$$Q_k Q_{k-1} \dots Q_1 A = \begin{pmatrix} R_{11} & R_{12} \\ 0 & A^{(k)} \end{pmatrix},$$

where $\begin{pmatrix} R_{11} & R_{12} \end{pmatrix} \in \mathbf{R}^{k\times n}$ is the first k rows of R and $A^{(k)}$ is the matrix that remains

Figure 5.1: Total number of stored nonzero entries in each elimination step.

to be factorized. Now, let us assume that there are h_k nonzero entries in the first column of $A^{(k)} \in \mathbf{R}^{(m-k)\times(n-k)}$. In the next elimination step, a Householder vector $u_k \in \mathbf{R}^{m-k}$ is formed, such that the Householder transformation $I - 2u_k u_k^T$ eliminates the last $h_k - 1$ nonzero elements in the first column of $A^{(k)}$. The vector u_k is zero in all elements but the h_k nonzero positions of the leading column in $A^{(k)}$. Given the Householder vectors $u_i, i = 1, ..., n$, the *Householder matrix* $H = [u_1, ..., u_n]$ is the concatenation of these vectors. Since u_i is zero in position $1 : (i-1)$, the matrix H is lower trapezoidal.

Following the above procedure, large and sparse matrices give long and sparse Householder vectors u_i. This is, however, easily avoided by sorting each active matrix $A^{(k)}$ by rows, in such a way that rows with leading nonzero element in the first column are permuted first in the matrix. Dense Householder vectors, and dense linear algebra, is then used. The submatrix with row dimension h_k in $A^{(k)}$, that actually is involved in the elimination, corresponds to what we will later define as a frontal matrix.

Let us now look more carefully on the multifrontal algorithm. Given a fixed column ordering of A, it is clear that columns in general must be eliminated from left to right. Fill is otherwise introduced in already processed columns. However, sparsity typically makes sets of columns structurally independent and possible to eliminate in any order, or in parallel. If two columns, i and $j > i$, of A are structurally independent, then the elimination of column i will not touch the nonzero elements in column j, and vice versa. The two columns can thus be eliminated in an arbitrary order.

It is easily shown that the columns i and j are structurally independent if and only if $r_{ij} = 0$. A compact way of describing all column dependencies is by an *elimination tree*. It is a rooted tree with n nodes and is defined from the nonzero structure of R, such that node p is the parent of node i if and only if

$$p = \min_j \{j > i \mid r_{ij} \neq 0\}.$$

In the elimination tree, node i corresponds to the elimination of column i in A. Let i and $j > i$ be two nodes in the tree and let $T[j]$ denote the subtree rooted at node j. Schreiber [95] shows that $r_{ij} = 0$ if $i \notin T[j]$. It follows that the columns corresponding to nodes in disjoint subtrees are independent, and can be eliminated in any order. Note, however, that r_{ij} might be zero although $i \in T[j]$. It is clear that elimination trees of low height, implying a large extent of branching and disjoint subtrees, are desirable for parallel computations. In the multifrontal method, the matrix A is completely factorized when the traversal of the elimination tree is completed, and each node, corresponding to the elimination of a column in A, is visited. Nodes can be visited in any order, as long as child nodes are visited before their parents. This is called a *topological traversal* of the elimination tree.

Since the elimination tree is obtained by symbolic prediction of R, with possible overestimation of the fill, the tree may be higher than the tree resulting from the correct R. The tree is, however, always correct if the matrix A, as we assume, has the SHP, since the predicted R is then correct. For the matrix given in (5.2), the correct elimination tree is a star-like tree of height two, where all leaves, but one, have the root as their parent. The elimination tree of the R predicted by the Cholesky factorization of $A^T A$ is, however, a path of length $n-1$, where node i has node $i+1$ as its parent, for $1 \leq i < n$.

To eliminate the i^{th} column of A, a *frontal matrix* F_i is formed by a contribution A_i from the matrix A, and *update matrices* U_{c_k} from the child nodes $c_k, k = 1, ..., t$, of node i,

$$F_i = \begin{pmatrix} A_i \\ U_{c_1} \\ \vdots \\ U_{c_t} \end{pmatrix}. \tag{5.3}$$

Here, A_i is the set of rows in A with leading nonzero elements in column i. To quickly access these rows, the matrix is initially sorted by rows. The resulting permuted matrix can be written in the form

$$A = \begin{pmatrix} A_1 \\ \vdots \\ A_n \end{pmatrix},$$

where

$$A_k = (0, \ldots, 0, \bar{a}_{kk}, \ldots, \bar{a}_{kn}) \in \mathbf{R}^{m_k \times n}$$

represents the $m_k \geq 0$ rows with leading nonzero entry in column k. The vector $\bar{a}_{kj} \in \mathbf{R}^{m_k \times 1}$ denotes the j^{th} column in this block row. By definition, $\bar{a}_{kk}$ is structurally nonzero in all components. Notice that there may be columns k in which no rows have their leading nonzero entries. For certain values of k, m_k may therefore be zero and the corresponding block row of A empty. The update matrices $U_{c_k}, k = 1, ..., t$, representing block rows in the active part of the partially factorized matrix, are defined and made available by the elimination of previous columns in A. Given these matrix contributions, the new frontal matrix is formed by the row merge operation (5.3).

The i^{th} column of A is eliminated by the elimination of the first column in F_i,

$$Q_i^T F_i = \begin{pmatrix} r_{ii} & r_{ii_1} & \dots & r_{ii_s} \\ 0 & & & \\ \vdots & & U_i & \\ 0 & & & \end{pmatrix}. \tag{5.4}$$

Then the i^{th} row of R, containing nonzero elments in columns $i, i_1, ..., i_s$, and the update matrix U_i of node i itself are computed. As soon as the update matrices of all children of a node are computed, the method can recursively continue upwards until the root node of the tree is reached and the factorization is finished.

To facilitate the use of dense linear algebra, we notice that identically zero columns in F_i remain zero after the application of orthogonal transformations. Such columns can therefore be removed in advance, making the remaining matrix dense. In practice, this modification is implemented by the use of *local* and *global* column indices. The column indices given in (5.4) refer to the global column indices of A.

Instead of eliminating only the first column of F_i (*restrained* elimination), also the remaining columns can be eliminated. A complete QR factorization of the frontal matrix is then computed, where the resulting triangular factor defines the i^{th} row in R and the update contribution U_i. By such *complete* elimination, the update contributions become zero below the main diagonal (upper triangular form). The update matrices are otherwise dense rectangular matrices.

A natural question is now whether complete or restrained elimination should be used. From a local point of view, it is clear that the number of operations used for a certain node is minimized if only the first column is eliminated. Then the primary task is fulfilled, and there is no local reason for further elimination. However, we will see that complete elimination significantly reduces both the operation count and the storage requirements. The latter can be seen in Figure 5.1, where "Multifront" corresponds to complete elimination and "Traditional Householder" to restrained elimination. It should be emphasized that the use of complete elimination is one of the most important differences between multifrontal and traditional Householder QR factorization. In fact, the multifrontal algorithm with restrained elimination is, from a computational point of view, equivalent to the traditional Householder algorithm. Thus, from now on, the multifrontal algorithm will always assume complete elimination.

The most important point is that complete elimination normally reduces the number of rows in the computed update contributions. If the the frontal matrix has m_F rows and n_F columns, then the corresponding update matrix has $\min(m_F, n_F) - 1$ rows when complete elimination is used, and $m_F - 1$ rows otherwise. The upper triangular update form, obtained by complete elimination, can be utilized in the frontal factorization, such that the zero entries below the diagonal are not explicitly annihilated. This is made by row-wise permutation of the frontal matrices into block upper triangular form. Numerical experiments by Matstoms [78] show that the total operation count can be reduced by half if the block triangular form is properly used.

The main reason for using complete elimination is to minimize the amount of intermediate fill. Following Proposition 5.1, fill in the Householder method is propagated as the union of nonzero entries in the involved columns. By always keeping the update matrices as low as possible, we also minimize the amount of propagated fill. Intermediate fill increases the size of the frontal matrices, making them more costly to store and to factorize. The amount of memory used is, however, more dependent on the storage of the update matrices. The rectangular update matrices from restrained elimination require much more space than the upper triangular matrices obtained by complete elimination.

The multifrontal algorithm computes the QR factorization by systematically visiting the nodes in the elimination tree. Each node is associated with a frontal matrix, formed by a contribution from the original matrix A and by update matrices of the children nodes. Factorization of the frontal matrices defines a row in R and the update matrix of the node itself. It follows that update matrices temporarily must be held in memory from the point when they are computed until the parent node is reached. To minimize the amount of temporary update storage, during the topological traversal, parent nodes should be visited as soon as possible after their first child node has been visited. This can be achieved by a *postordering* of the nodes, which orders the nodes such that node j always has node $j-1$ as one of its children. A postordering of the nodes in the elimination tree, and of the columns in A, does not affect the fill in R, as explained in the next section. Postorderings can be conveniently computed by a *depth-first search* of the elimination tree, and they are a special case of topological orderings.

A number of important modifications of the described multifrontal algorithm have been proposed. The most significant improvement is *node amalgamation*, where adjacent nodes in the elimination tree are combined into *supernodes*. Such nodes correspond to the elimination of more than one column in A. Node amalgamation can, under certain conditions, be made without increasing the amount of fill. Less overhead and reduced number of floating point operations give dramatic reductions in the total execution time. A detailed study of multifrontal QR factorization can be found in Puglisi [87] and Matstoms [79].

5.3 Equivalent reorderings

In this section and the next, we describe column reordering strategies for A in order to obtain better properties for sparse QR factorization than the original column ordering of A. This section starts with a brief explanation of our purpose and approach, followed by necessary definitions and terminology. The connection between sparse matrix factorization and graph theory is given here, along with the definition of and results on equivalent reorderings. The next section is more specifically about column reordering strategies for sparse QR factorization.

The QR factorization of A is usually computed after an initial fill-reducing ordering on A to get low fill in R. Minimum degree is a popular fill-reducing method which has a fast implementation and which gives good results in general. As we have seen earlier, R is essentially identical to the Cholesky factor of A^TA. As suggested first by George and Heath [40], the minimum degree analysis is based on A^TA to produce low fill in the resulting R. Thus, minimum degree and other fill-reducing heuristics based on A^TA, consider only the amount of fill in R. For QR factorization, however, the number of operations required in the factorization is not only dependent on the size of R, but also on the size of intermediate fill, and the size and the structure of the frontal matrices for the multifrontal method. Since $m \geq n$, the importance of the latter mentioned properties increases as the difference between m and n increases. Important information about these properties is lost when only A^TA is considered. For good column reorderings, the information from A should also be used in addition to A^TA. Our purpose is to reduce the number of operations required in sparse QR factorization, without introducing any new fill in R. Certain column reorderings of A that preserve the size of R are referred to as equivalent reorderings, and will shortly be defined formally. Our approach will be to find, among equivalent reorderings, one that tries to minimize the number of operations in sparse QR factorization, studying the structure of A.

We will use graph theory to describe equivalent reorderings. The reader is assumed to be familiar with standard graph terminology, but some of the terms especially related to sparse factorization are now explained. Let $M \in \mathbf{R}^{n\times n}$ be a sparse, symmetric and positive definite matrix, and let L be its Cholesky factor, so that $M = LL^T$. The graph $G = (\{1,2,...,n\},E) = G(M)$ is a simple graph where the vertices are numbered from 1 to n, and where each vertex corresponds to a row/column of M. For each pair of vertices i and j, the edge (i,j) is in the graph if and only if $m_{ij} \neq 0$. The *filled graph* of G is the graph $G^+ = G^+(M) = G(L+L^T)$. For triangular matrices, we will simply use $G(L)$ or $G(L^T)$ instead of the correct term $G(L+L^T)$. When M and L are clear from the context, we will only use G and G^+. The filled graph G^+ can be obtained from G without computing L, with help of the following algorithm by Parter [83], called the *elimination game*:

for $i = 1 : n$,
    add edges to make all neighbors of vertex i pairwise adjacent;
    delete vertex i from the graph;
end

The graph G^+ is obtained by adding to G all the new edges suggested by this algorithm. These edges are called *fill edges* and correspond to fill elements in L. What is done in one step of this loop, is called *eliminating* vertex i. The i^{th} *elimination graph* G_i is the resulting graph after the elimination of vertex i. Thus, $G_0 = G$, and $G_n = \emptyset$. An *elimination ordering* α on G is a renumbering of the vertices of G. We use G_α to denote the renumbered graph, and G_α^+ to denote the resulting filled graph. A *perfect elimination ordering* of G is an elimination ordering which results in no fill. Rose [91] has shown that all filled graphs are chordal. Chordal graphs are exactly the class of graphs which have perfect elimination orderings. A *simplicial* vertex in a chordal graph is a vertex whose neighbors induce a clique. Thus the elimination of a simplicial vertex does not produce any fill edges. Every chordal graph that is not a complete graph, has at least two nonadjacent simplicial vertices, and chordality is a hereditary property. Accordingly, a perfect elimination ordering of a chordal graph can be found by eliminating a simplicial vertex at each step of the elimination game. These results on chordal graphs are due to Dirac [23], and Fulkerson and Gross [36].

Let us show these graph issues on an example. Figure 5.2 shows a symmetric matrix M and its Cholesky factor L^T. The graph $G = G(M)$ and the filled graph $G^+ = G^+(M)$ are also shown. In G, vertex 1 is simplicial, thus no fill is introduced when it is eliminated first. The elimination of vertex 2 introduces the fill edge connecting vertices 3 and 4. After the elimination of 2, the remaining graph in the elimination process is complete, and all three of the vertices 3, 4 and 5 are simplicial. Thus no further fill is introduced. The filled graph G^+ is chordal, and $\alpha = [1, 5, 4, 2, 3]$ and $\beta = [1, 2, 3, 4, 5]$ are two of its perfect elimination orderings. As defined by Rose, Tarjan and Lueker [92], a *minimal fill ordering* α on a graph G is an ordering such that no proper subgraph of the resulting filled graph G_α^+ is both chordal and includes G as a subgraph. Note that the initial ordering of $G(M)$ in the example of Figure 5.2 is a minimal fill ordering.

An elimination ordering α on $G = G(M)$ corresponds to a symmetric permutation of the matrix M. For each ordering α, there can be found a permutation matrix P such that $G_\alpha = G(P^T M P)$. In MATLAB notation, P can be expressed as $P = I_n(:, \alpha)$, where I_n is the $n \times n$ identity matrix.

Liu [70] gives the following definition for *equivalent reorderings*.

Definition 5.1 *An elimination ordering α on G is an equivalent reordering of G if G_α^+ is isomorphic to G^+, that is, the new ordering α produces a filled graph isomorphic to the original filled graph of G.*

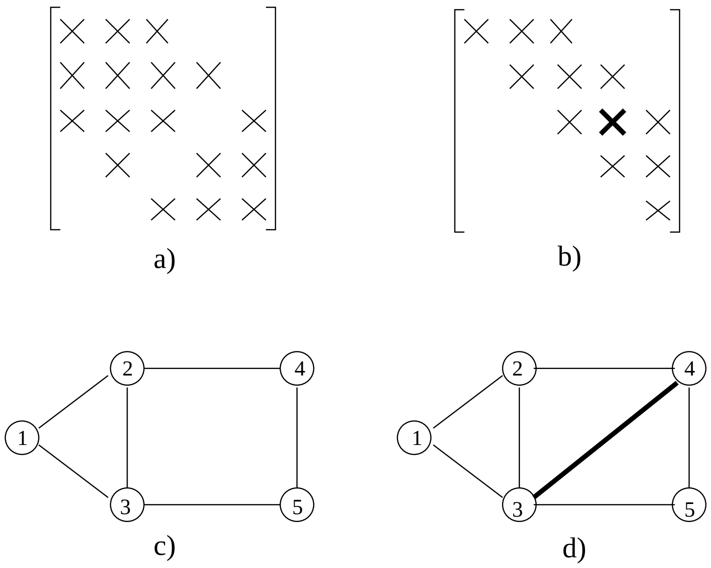


Figure 5.2: a) A symmetric matrix M ($M = A^T A$, where A is the nonsymmetric matrix shown in Figure 5.3). b) The Cholesky factor L^T of M. c) The graph $G(M)$. d) The filled graph of M, $G^+(M) = G(L)$.

For matrices, an equivalent reordering of M is a symmetric permutation of M that results in the same fill elements in the Cholesky factor. The Cholesky factor of the permuted matrix is also permuted, therefore the nonzero patterns of the factors of M and $P^T MP$ are not the same. But the two factors have the same number of nonzero elements, since their graphs are isomorphic.

Our study on equivalent reorderings is based on the following lemma.

Lemma 5.1 *Let G be a graph and $F = G^+$ be its filled graph. Let α be a perfect elimination ordering on F. Then G^+_α is a subgraph, or a proper subgraph of F_α.*

Proof: Since F is chordal, it has a perfect elimination ordering α such that $F^+_\alpha = F_\alpha$. Elimination of the graph G_α cannot produce any fill edges that are not in F_α, otherwise these edges must also be introduced in the elimination of F_α, contradicting that α is a perfect elimination ordering on F.

To see that G^+_α can be a proper subgraph of F_α, consider a case where G is a chordal graph and F is a complete graph as a result of a poor initial ordering on G. (An example is a star graph where the center vertex is numbered first.) The perfect elimination orderings of F are thus simply all permutations of $\{1, 2, .., n\}$. Among these permutations, there is at least one ordering α which is also a perfect elimination ordering of G since G is chordal and has the same vertex set as F.

Applying α on G results in zero fill, and the produced filled graph G_α^+ is a proper subgraph of F_α. ∎

Note that F need not be complete for G_α^+ to be a proper subgraph of F_α. If F is a result of an arbitrary elimination ordering on G, then it is likely that perfect elimination orderings α of F, applied on G, will give G_α^+ that is a proper subgraph of F_α.

Lemmas 5.2 and 5.3 are from Liu [70].

Lemma 5.2 *If α is an equivalent reordering on G, then α is a perfect elimination ordering on the filled graph G^+.*

From Lemmas 5.1 and 5.2. we see that equivalent reorderings of G constitute a subset of perfect elimination orderings of its filled graph G^+.

Lemma 5.3 *Let G be initially ordered by a minimal fill ordering. A reordering on G is equivalent if and only if it is a perfect elimination ordering of the filled graph G^+.*

The graph G in Figure 5.2 is initially ordered by a minimal fill ordering. Therefore, in this example, all perfect elimination orderings of the filled graph G^+ are equivalent reorderings on G.

Corollary 5.1 *Let $F = G^+$ be a filled graph. A reordering on F is equivalent if and only if it is a perfect elimination ordering on F.*

Proof: Since F is a filled graph, it is already numbered such that $F^+ = F$. Therefore, this is a perfect elimination, hence minimal fill, ordering on F, and Lemma 5.3 applies with $F^+ = F$. ∎

Thus for all filled graphs, the set of equivalent reorderings and the set of perfect elimination orderings coincide.

Returning to matrix notation, an equivalent reordering of M is a perfect elimination ordering of $G(L)$, whereas a perfect elimination ordering of $G(L)$, applied on M, might result in a factor with fewer nonzero elements than in L. As pointed out by Liu [70], if M is initially ordered by a fill-reducing ordering, factors resulting from perfect elimination orderings of $G(L)$ applied on M, will usually not differ considerably from the original factor L of M with respect to the amount of fill. Because of this, and to be able to consider a larger class of orderings, we will concentrate on the equivalent reorderings of L, i.e. perfect elimination orderings of $G(L)$, rather than on the equivalent reorderings of M. From now on, we use equivalent reordering to refer to one on the filled graph $G(L)$. *In the rest of this paper we will sometimes misuse the term equivalent reordering. Since L is unique for M, equivalent reorderings of M will be used to mean equivalent reorderings of L, or equivalently, perfect elimination orderings of $G(L)$.*

Although equivalent reorderings usually result in the same filled graph, the structure of the resulting elimination tree is not necessarily the same. The filled graph is reordered, and hence the sparsity pattern of the new resulting Cholesky factor is changed, although its graph might be the same as before. Following the definition by Tarjan [97], a *topological ordering* of the elimination tree is an ordering that numbers the children nodes before their parent node. Therefore, topological orderings preserve the elimination tree. Postorderings, mentioned in Section 5.2, are a subset of topological orderings. Liu [68] observes that any topological ordering of the elimination tree corresponds to an equivalent reordering of the matrix M. However, because of the discussion above, the opposite is not true. All the leaves of the elimination tree are simplicial vertices in the filled graph. But there may be more simplicial vertices in the filled graph than there are leaves in the elimination tree. Accordingly, it can be shown that the number of equivalent reorderings of the graph is greater than or equal to the number of topological orderings of the elimination tree. The class of topological orderings of the elimination tree is a subset of the class of elimination tree preserving orderings, which is again a subset of equivalent reorderings of the initial matrix/graph. These relations are summarized by Liu [70].

Let us return to QR factorization (5.1) of A. If L is the Cholesky factor of A^TA, then $G(R) = G(L)$ since the nonzero structures of R and L^T are the same. In this case, equivalent reorderings of A refer to perfect elimination orderings of $G(R) = G(L)$. A symmetric permutation of A^TA implies a column permutation on A, since $P^T(A^TA)P = (AP)^T(AP)$. Consequently, we use *equivalent column reorderings* to denote the equivalent reorderings on R, which are the same as perfect elimination orderings on $G(R)$.

Graph theory is also used to represent nonsymmetric matrices. A *bipartite graph* $B = (U, V, E)$ is a graph with two vertex sets U and V, and for each edge (i, j) in E, $i \in U$ and $j \in V$. The bipartite graph $B(A)$ of a nonsymmetric matrix A is defined as follows: $U = \{1, 2, ..., m\}$ is the set of rows of A, $V = \{1, 2, ..., n\}$ is the set of columns of A, and for each nonzero entry a_{ij}, there is an edge (i, j) in $B(A)$. The assumed Strong Hall Property can also be explained by graph terminology. The bipartite graph $B(A)$ has the SHP if, for every proper subset W of the column vertex set V, the vertices in W have altogether at least $|W| + 1$ neighbors in U. For traditional Householder QR factorization, the elimination game on the bipartite graph $B = B(A)$ is defined by the following algorithm (Manneback [76]), which we call *the (Householder) bipartite elimination game*:

for $i = 1 : n$,
  let S be the set of all neighbors of column vertex i; $(S \subseteq U)$
  let T be the set of all neighbors of the vertices in S; $(T \subseteq V)$
  add edges to make all vertices in S adjacent to all vertices in T;
  delete row-vertex i and column-vertex i from the graph;
end

(A similar process is described for Givens transformations by George, Liu, and Ng [44].) The *filled bipatite graph* B^+ is obtained by adding to B all the edges suggested by this algorithm. The *bipartite elimination graph* B_i is the resulting graph after the elimination of column and row i. Thus, $B_0 = B$, and B_n consists of $m - n$ isolated row vertices. A connection between the set T and the neighborhood of vertex i in $G(A^TA)$ should be noted. For $i = 1$, the set T is the set of all neighbors of vertex i in $G(A^TA)$ and $G(R)$. For $i > 1$, the set T is the set of all higher numbered neighbors of i in $G^+(A^TA) = G(R)$. For $1 \leq i \leq n$, the set T is also the set of all neighbors of vertex i in the elimination graph $G_{i-1}(A^TA)$. For the bipartite elimination game algorithm to work, the matrix must have only nonzero entries in its main diagonal. The rows of a Hall matrix can always be permuted to result in a zero-free diagonal [24, 54, 73]. The *total fill* involved in the elimination process is the total of all the edges added during this algorithm. The fill edges (i, j), where $i \leq j$, correspond to fill appearing in A^TA and R. The number of nonzero entries in A^TA is constant for all column or row orderings of A. The number of nonzero elements in R is, however, dependent on the column ordering of A. Strictly speaking, the fill edges (i, j), where $i > j$, are called *intermediate fill* since they are later annihilated during the factorization, to give only zero entries below the main diagonal in R. *However, our interpretation of intermediate fill will be slightly different in the rest of this paper.* The fill which is added to $G(A^TA)$ to get $G(R)$ always appears in the upper triangle. This fill is dependent on the column ordering. Where the rest of the fill appears is dependent on the row ordering of A. Manneback mentions in [76] p. 29, that row orderings do not change the amount of total fill in the Householder bipartite elimination game. But row orderings do decide how much of the fill appears above the main diagonal, and how much below. *Since our work does not consider row orderings , we will onwards use intermediate fill to refer to the total fill except the fill introduced between $G(A^TA)$ and $G(R)$.* Figure 5.3 shows an example of this process. Intermediate fill is shown in dotted line style, whereas the fill in R is shown by thick lines. Consider also the symmetric matrix M in Figure 5.2. These two examples are related such that $M = A^TA$, where A is the matrix in Figure 5.3. Thus the Cholesky factor L^T of M and the QR factor R of A have the same nonzero structure. The structure of R is the upper triangular part of the filled matrix shown in Figure 5.3 c). The Householder matrix H has the same lower trapezoidal structure as the lower trapezoidal part of the filled matrix.

When A is initially ordered by a fill-reducing analysis based on A^TA, equivalent column reorderings may result in a slightly smaller R. However, the main importance of equivalent reorderings is that they can result in less number of operations in the factorization. Equivalent reorderings generally result in considerable differences in elimination tree height, amount of intermediate fill, and size and structure of frontal matrices in the factorization of A. Experimental results, presented in the next section, show that improvements, with respect to the number of operations, are highly probable by an arbitrary equivalent reordering. These results also show

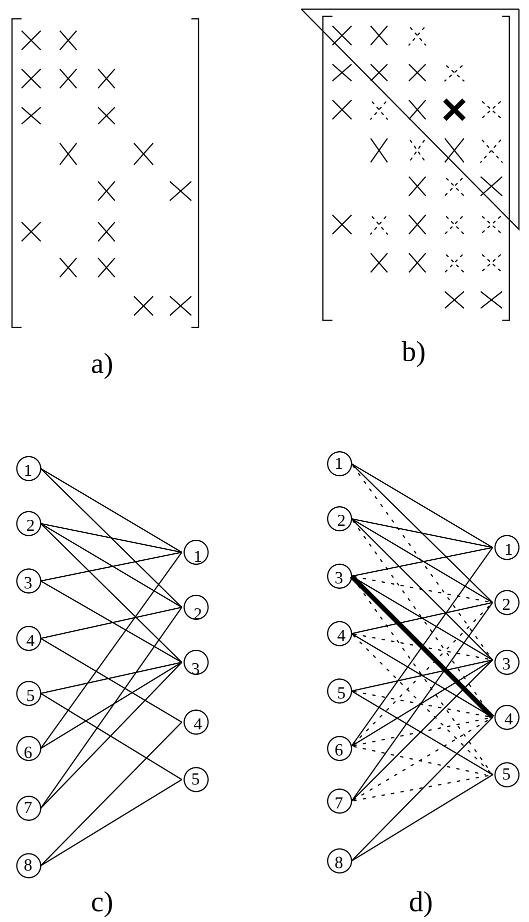


Figure 5.3: a) A rectangular matrix A. b) The filled matrix of A showing total fill occurring during the Householder QR factorization. The upper triangular part corresponds to R. b) The bipartite graph $B(A)$. c) The filled bipartite graph, where the dotted edges correspond to intermediate fill, and the thick edge shows fill in R.

that the best equivalent reorderings can be significantly better than the original fill-reducing ordering. Thus, with intelligent equivalent reordering strategies, the rate of improvement can be increased considerably. In the next section, we will concentrate on how equivalent reorderings can reduce the number of operations required by the multifrontal and by the traditional Householder QR factorization algorithms.

We know that efficient sparse QR factorization by Householder transformations requires a method like the multifrontal scheme with *complete* factorization of the frontal matrices. The traditional Householder algorithm introduces too much intermediate fill, and therefore can not be used in practice. However, an interesting connection to sparse LU factorization (Gilbert [46], Gilbert and Ng [49]) makes the traditional Householder algorithm still interesting in real applications. The main result, by George and Ng [45], is presented in Theorem 5.1 using our notation.

Theorem 5.1 *Let $A \in \mathbf{R}^{n \times n}$ be a nonsingular matrix with nonzero diagonal. Suppose A is factorized by Gaussian elimination with row interchanges as*

$$A = P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} U,$$

and define L to be the unit lower triangular matrix whose i^{th} column is the i^{th} column of L_i. Then

$$B(L+U) \subseteq B^+(A).$$

The filled graph $B^+(A)$ is the bipartite graph of the filled matrix of A. The upper triangular part of this filled matrix corresponds to the upper triangular factor R, whereas the remaining lower trapezoidal part corresponds to the Householder matrix H. It follows from the theorem that the nonzero structures of L and U, independently of row permutations of A, are contained in the nonzero structures of H and R, respectively. Although our aim, with respect to efficient sparse QR factorization, is to find good column orderings for the multifrontal method, because of the discussion above, we also study ordering heuristics for the traditional Householder algorithm in order to achieve a sparse H.

5.4 Equivalent column reorderings for sparse QR factorization

We start this section by an example showing that the amount of intermediate fill can be changed considerably by equivalent column reorderings on the matrix to be factored. Our example matrix A is the first matrix given in Figure 5.3. The matrix $A^T A$ is shown in Figure 5.2 a). A close study of the graph $G(A^T A)$ shows that the initial ordering of A is a fill-reducing ordering on $A^T A$, namely a minimum degree ordering. We have already seen that, in this example, the mentioned ordering is also a minimal fill ordering. Let us now consider an equivalent column reordering on A. That means, we want a perfect elimination ordering of the filled graph of Figure 5.2 d). As noted earlier, $\alpha = [1, 5, 4, 2, 3]$ is such an ordering, and G_α^+ is isomorphic to G^+. Let us now reorder the columns of A to get a new matrix $B = A(:, \alpha)$, and then reorder the rows of B such that the elements on the main diagonal are all nonzero. The new matrix B is shown in Figure 5.4 a). In Figure 5.4 b) the filled matrix resulting from the traditional Householder algorithm is shown. We see that the number of nonzeros in the upper triangular part is the same as in the factor R of A, whereas the amount of intermediate fill is considerably reduced.

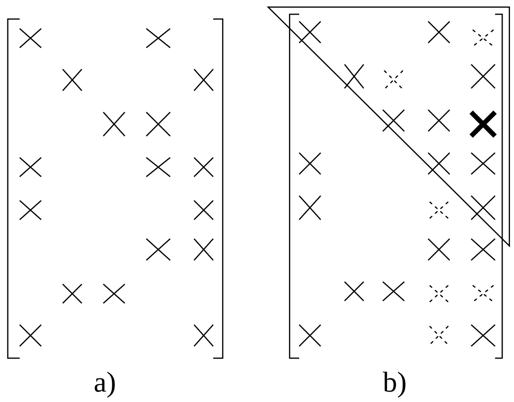


Figure 5.4: a) The resulting matrix B after an equivalent column reordering on matrix A of Figure 5.3. The rows are ordered to give only nonzeros in the main diagonal. b) The resulting filled matrix, where the upper triangular part represents the factor R' of B.

Studying the filled matrices of Figure 5.3 b) and Figure 5.4 b), some interesting observations about the occurrence of fill can be made. Since B is a result of an equivalent reordering on A, the number of nonzero entries in R' of B is always bounded above by the number of nonzeros in R. In addition, in this example, A is originally ordered by a minimal fill ordering. Therefore, the number of *nonzeros* in the upper triangular part of the filled matrix is *constant* for all equivalent reorderings of A. As can be seen from the figures, the amount of *fill* in upper triangle is *not constant*. It should be noted, however, that the number of fill edges added to $G(B^T B)$ to get $G(R')$ is constant, namely one, for every equivalent reordering on A in this example. The amount of intermediate fill can vary considerably for equivalent reorderings. What amount of this fill appears in the upper triangle as a part of $B^T B$, and what amount of it appears below the main diagonal, is dependent on the row ordering. The storage requirements by traditional Householder QR factorization are proportional to the number of nonzeros in the filled matrix, which is equal to the number of edges in the filled bipartite graph $B^+(A)$. This number, and accordingly, the size of the intermediate fill, is closely related to the number of operations. Thus for traditional Householder QR factorization, an important aim should be to find equivalent reorderings that reduce the amount of intermediate fill, or the number of nonzeros in the Householder matrix H. We will shortly see that the situation is more complex for the multifrontal QR factorization algorithm.

We now leave traditional Householder QR temporarily, and continue with some results on the multifrontal QR factorization. For the presented numerical results on the multifrontal method, our experiments are done with the multifrontal algorithm implemented in MATLAB by Matstoms [78]. As mentioned in Section 5.2, the introduction of intermediate fill is quite different in the multifrontal method. If we were to define a *multifrontal bipartite elimination game*, the difference from the Householder bipartite elimination game would be that several row vertices can be removed from the graph at each step of the multifrontal game. Remember that at step i, one column and $h-d+1$ rows are eliminated, where the frontal matrix corresponding to column i has h rows and d columns. Because of this, in the multifrontal method, the resulting filled bipartite graph $B^+(A)$ is different, and usually suffers less fill, than in the Householder bipartite elimination game described in Section 5.3. In the multifrontal method, other factors than the size of the intermediate fill are also important for efficient factorization. The results we now present, concern the intermediate fill and the number of operations in the multifrontal QR factorization algorithm. For our numerical experiments, we order the columns of the given matrix A by a fill-reducing analysis on A^TA. The method we have chosen is the *colmmd* function of MATLAB.

We started Section 5.3 by mentioning that the number of operations in multifrontal QR factorization is not only dependent on the number of nonzeros in R. Now we want to illustrate this point by an example. Figure 5.5 shows that various number of operations can be achieved for fixed $nnz(R)$ by equivalent reorderings. The same tendency is observed for all our test matrices, which is expressed in Observation 5.1.

Observation 5.1 *An equivalent reordering that reduces the number of nonzeros in R, does not necessarily reduce the number of operations required by the multifrontal factorization.*

As a consequence of Observation 5.1, we decide to concentrate also on other criteria when we want to find good equivalent reorderings. The example in Figure 5.6 shows that, for the multifrontal method, the amount of intermediate fill alone is neither enough to decide good orderings. From this we conclude that, as expected, the number of operations is dependent both on the size and the structure of the intermediate fill, and on the size and the structure of the frontal matrices, in addition to fill in R. Note, however, that for the example in Figure 5.6, an equivalent reordering that reduces the amount of intermediate fill will actually reduce the num-

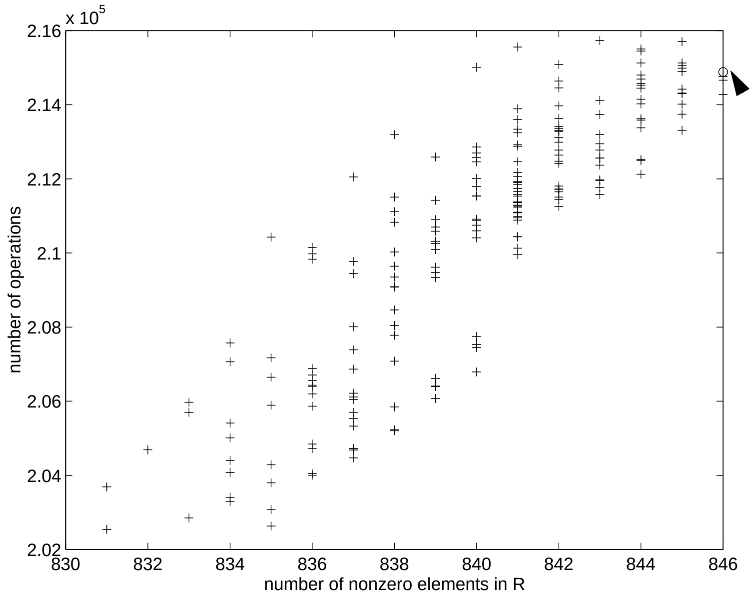


Figure 5.5: The columns of matrix ASH331 are ordered by the *colmmd* function of MATLAB. The chart shows the relation between the number of operations required by the multifrontal QR factorization and the size of $nnz(R)$ for 200 equivalent reorderings. Each equivalent reordering is marked with a plus. The initial *colmmd* ordering is shown by a circle.

ber of operations required in the factorization, compared to the initial fill-reducing ordering.

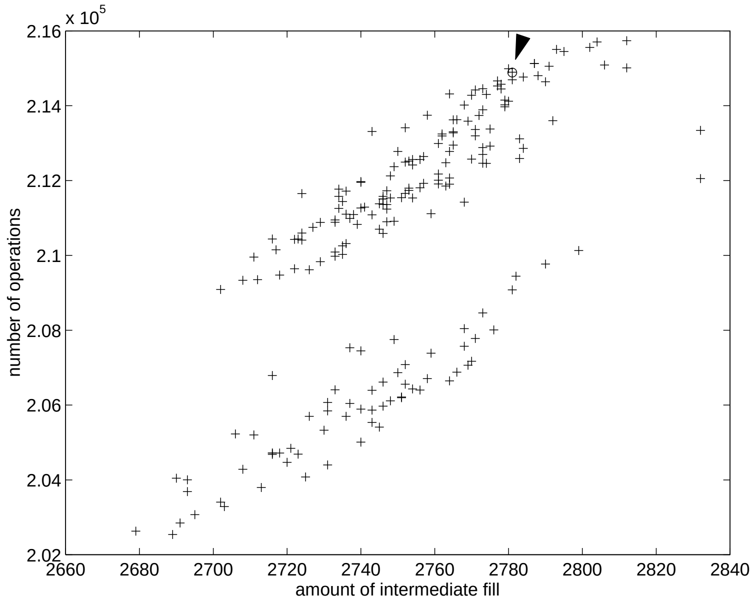


Figure 5.6: The columns of matrix ASH331 are ordered by the *colmmd* function of MATLAB. The chart shows the relation between the number of operations required by the multifrontal QR factorization and the size of intermediate fill for 200 equivalent reorderings. Each equivalent reordering is marked with a plus. The initial *colmmd* ordering is shown by a circle.

Observation 5.2 *An equivalent reordering that reduces the amount of intermediate fill, does not necessarily reduce the number of operations required by the multifrontal factorization.*

Observation 5.3 *The number of operations required by an equivalent reordering of A in the multifrontal method, is dependent on the number of nonzeros in R, the size and the structure of the intermediate fill, and the size and the structure of the frontal matrices resulting from the reordering.*

Our goal is to find, among equivalent column reorderings, one that reduces the number of operations in the multifrontal factorization. We want this ordering to be as close as possible to the best equivalent reordering with respect to the number of operations. We concentrate on the following two criteria for the solution of this problem:

- Size and structure of the frontal matrices.
- Size and structure of total fill.

In our heuristics, we consider perfect elimination orderings of $G(R)$. We start by examining all the simplicial vertices in $G(R)$. One of these is chosen as number 1, and is eliminated from the graph. We use the structural information from A to make this choice. At each step k, all simplicial vertices of the remaining graph are considered. Which one of these is chosen as number k depends on the heuristic, and we concentrate on the following alternatives. Choose a vertex such that:

1. Size of the k^{th} frontal matrix is minimized.
2. Number of rows with leading entry in column k of A, is minimized.
3. Number of operations to factorize the k^{th} frontal matrix is minimized.

For each of our test matrices, at least one of these heuristics produces an ordering close to the best of the 200 random equivalent reorderings tested.

Heuristic 1 The idea behind this heuristic is to locally minimize the size of the frontal matrix created at each step. The value minimized is simply the number of rows times the number of columns of the k^{th} frontal matrix. The number of rows of the frontal matrix k is the number of rows in A that have their leading entry in column k of A, in addition to the number of rows in the update matrices for column k. We approximate this number by only counting the number of rows with leading entry in column k of A. The number of columns in the frontal matrix is the number of nonzeros in row k of the resulting R.

Heuristic 2 This heuristic minimizes the number of rows with leading entry in column k of A at each step k. The idea is to try to locally minimize the fill introduced at each step of the bipartite elimination game. The fewer nonzeros there are in the leading column, the less chance there is that fill is created as a result of these nonzeros. This is of course not completely true, because the occurrence of fill is dependent on the amount of original nonzero entries and previously created fill in this space. Clearly, minimizing fill locally does not necessarily result in a global minimization of total fill, but as a heuristic, this approach works well. Heuristic 2 has also another interpretation, it attempts to minimize locally the number of rows in the frontal matrix k.

Heuristic 3 This heuristic is similar to Heuristic 1, but minimizes the number of operations required to factorize the k^{th} frontal matrix. Since the frontal matrices are considered dense, the factorization requires approximately $ops(k) = 2n_k^2(m_k - n_k/3)$

operations for frontal matrix k, where m_k is the number of rows and n_k is the number of columns of frontal matrix k. The number m_k is approximated here as in Heuristic 1. At each step k, a column is chosen to minimize $ops(k)$.

All these three heuristics concentrate on the multifrontal algorithm. For this algorithm, we have also tried another alternative: We formed the elimination tree, and considered the update matrices coming from the children in the elimination tree of each vertex to obtain the exact number of rows in the frontal matrix k. We then used this to minimize the number of Householder operations to factorize the k^{th} frontal matrix. But this attempt did not generally give as good results as Heuristics 1, 2, and 3.

The results from numerical tests, comparing Heuristics 1, 2, and 3 to the initial column ordering, and to the random equivalent reorderings, for the multifrontal algorithm are given in Table 5.2.

Matrix	colmmd	eq. reorderings		heuristics		
		min	max	1	2	3
ABB313	430282	389354	429971	386395	375949	375436
ASH219	111798	109924	116357	109011	109011	109011
ASH331	214887	202539	215738	196292	193442	193442
ASH608	574185	524108	575705	521700	548536	548536
ASH958	1071796	920063	1072839	1062447	921211	1053683
WL1033	689017	668106	865285	782113	781870	656637
WL1252	4443569	4222301	4853924	4199075	4215067	4197257
WL1364	5074931	4952360	5442450	4982849	4928433	4963352
WL1641	14203097	14169857	16914612	14527772	13985161	14527772
WL1850	3742639	3469173	3704666	3439713	3535497	3438114
WL1991	63352154	62899480	64543723	62141096	61887380	62201316
WL2808	180635063	174184039	179673640	175131384	172559760	173969244

Table 5.2: Each entry in the table shows the number of flops achieved in the multifrontal factorization. The columns of each matrix are first ordered by *colmmd*. Then 200 equivalent reorderings are tested for each row of the table, and the minimum and the maximum values are given along with the results from Heuristics 1, 2, and 3.

Studying the results presented in Table 5.2, we see that it is not easy to decide on the best heuristic method. At least one of the heuristics always produces a better ordering than the initial fill-reducing ordering, regarding the number of operations required by the multifrontal factorization. Only for one test matrix was the best heuristic "beaten" by the best random permutation. However, the heuristic that gives the best result, is not the same for all matrices. Except for two of the test matrices, all heuristics produce better orderings than the initial ordering. For the best results, around 10% improvement is achieved, compared to the initial ordering.

For ease of interpretation, we give a graphic representation of the results of Table 5.2 in Figure 5.7. The results are normalized by the *colmmd* ordering. The numbers along the x-axis represent the test matrices in the order they are listed in our tables. The numbers along the y-axis show how many times more operations the *colmmd* ordering requires than a given heuristic.

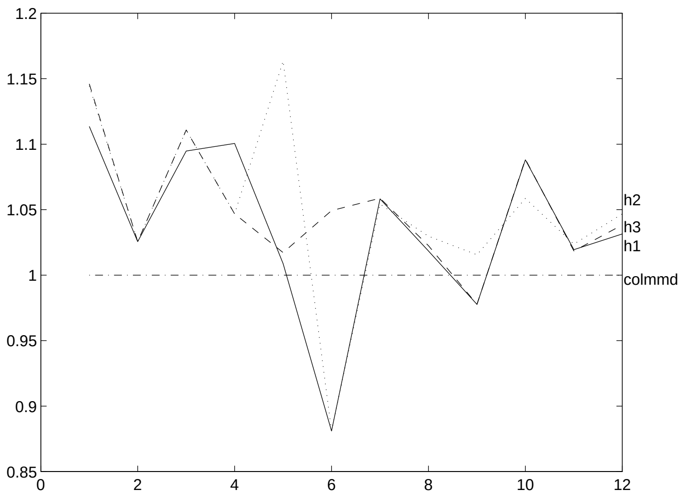


Figure 5.7: Graphic representation of the results of Table 5.2. Heuristics 1, 2, and 3 are shown in respectively solid, dotted and dashed line styles.

Although the multifrontal algorithm is superior for computational purposes, because of the relation to LU factorization, the traditional Householder algorithm is still important. We now present two more heuristics that aim to reduce the size of the total fill introduced both in the multifrontal method and the traditional Householder method.

Heuristic 4 At step k, choose a simplicial vertex of the remaining graph such that the amount of fill introduced in the bipartite elimination graph as a result of the elimination of column k of A is minimized. This is a direct attempt to minimize fill locally. At each step, the amount of fill that the elimination of column k introduces is computed for every simplicial vertex k. This amount of fill is approximated for each k as $|S| \cdot |T| - nnz(A(S,T))$, where S and T are as defined in the bipartite elimination game. Since A is filled after each elimination step, the exact amount would have to consider the updated matrix $A^{(k)}$ rather than the original matrix A.

Heuristic 5 At step k, choose a simplicial vertex of the remaining graph such that the number of nonzeros in the union of column k of A and all earlier eliminated columns is minimized. This is also an attempt to minimize locally the fill resulting from the elimination of column k. The desired effect is to choose a column whose nonzero elements are mainly in the positions of the nonzero elements of the previously eliminated columns. This way, we hope that most of the fill that would result from the elimination of column k is already introduced by previously eliminated columns, reducing the fill introduced at step k.

For the multifrontal algorithm, both of these attempts gave very similar results as Heuristic 2. Remember that Heuristic 2 also attempts to minimize the size of the total fill. For the traditional Householder algorithm, however, Heuristic 5 gave almost exactly the same results as Heuristic 2, whereas Heuristic 4 gave different results and turned out to be competitive to the other heuristics for several test matrices. We have also tested our heuristics, and random equivalent orderings, on the traditional Householder algorithm. Because of the slowness of this algorithm, we tested only 10 random orderings for each matrix. The results are given in Table 5.3. A graphic representation of these results analogous to Figure 5.7 is given in Figure 5.8.

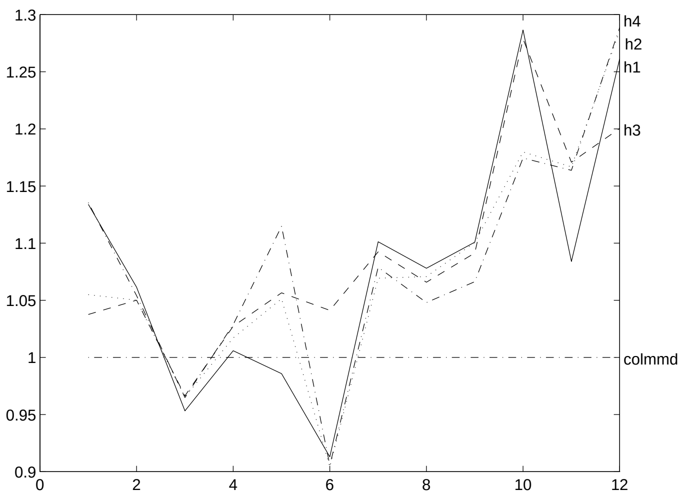


Figure 5.8: Graphic representation of the results of Table 5.3. Heuristics 1, 2, 3, and 4 are shown in respectively solid, dotted, dashed, and dash-dotted line styles.

Matrix	colmmd	eq. reorderings	
		min	max
ABB313	988655	934455	963335
ASH219	261323	242119	267059
ASH331	539540	526491	578354
ASH608	1835662	1709033	1767927
ASH958	4438783	3963993	4282513
WL1033	3141514	3326010	3602089
WL1252	8890684	8207077	9005466
WL1364	10473678	9552596	10344194
WL1641	22501539	20432749	23294883
WL1850	20067971	16626744	17242186
WL1991	63802057	57313800	65008903
WL2808	162841784	120904812	132572554

Matrix	heuristics			
	1	2	3	4
ABB313	871827	937210	952865	870643
ASH219	246201	248846	248846	247921
ASH331	566035	558901	558295	559174
ASH608	1825013	1804840	1786941	1784510
ASH958	4502867	4221928	4201131	3981945
WL1033	3440816	3474382	3017202	3469009
WL1252	8073458	8314081	8138825	8240455
WL1364	9715547	9781379	9826531	9994180
WL1641	20439288	20445502	20617927	21102045
WL1850	15598425	17008660	15693253	17086452
WL1991	58859649	54683557	54492602	54824176
WL2808	129108048	126648329	135660479	126363521

Table 5.3: Each entry in the table shows the number of flops achieved in the traditional Householder factorization. The columns of each matrix are first ordered by *colmmd*. Then 10 equivalent reorderings are tested for each row of the table, and the minimum and the maximum values are given along with the results from Heuristics 1, 2, 3, and 4.

Like for the multifrontal algorithm, also for the Householder algorithm, it is not possible to decide which of the three heuristics 1, 2, and 3 is the best in general. Among these three heuristics, the heuristic that performs best for each matrix is not the same for the multifrontal and the traditional Householder algorithms. For the traditional Householder algorithm, Heuristic 4 is clearly a winner for the ABB, and the ASH matrices. However, for the other matrices, it is not uniformly comparable to the other heuristics. The gain from equivalent reorderings turns out to be slightly better for the traditional Householder case. Generally, around 10%, and up to 20% for some of the larger matrices, of improvement in the operation count is achieved by the best reorderings. Except for one matrix, at least one heuristic always gives a better result than the initial *colmmd* ordering. Except for three of the test matrices, all heuristics produce better orderings than the initial ordering.

We end this section with a brief discussion on the initial fill-reducing ordering. The *colmmd* function of MATLAB, which we used as our initial fill-reducing ordering, aims to produce a minimum degree ordering of $A^T A$ directly from A. However, this function does not produce a strict minimum degree ordering. Some approximations are done to compute a good ordering efficiently. The parameters of this function can be set from *default* to *tight* so that a strict minimum degree ordering is produced. Tests with *tight colmmd* showed that the gain from equivalent reorderings may become more restricted, although improvements are achieved also here. For some matrices, equivalent reorderings of a *default colmmd* gave better results than *tight colmmd* with or without proceeding equivalent reordering. The reader can compare the results of Tables 5.2 and 5.3 to the results from *tight colmmd* presented in Tables 5.6 and 5.8. This effect is in accordance with the example shown in Figure 5.5. An ordering that initially minimizes $nnz(R)$ need not be the best ordering for sparse QR factorization. Equivalent reorderings can only reduce and not increase the amount of fill in R. Observation 5.1 indicates that the best operation count and R with smallest fill need not coincide. Thus starting with a slightly more filled R than the best possible R might give better results regarding equivalent reorderings and achieving best operation count.

5.5 Tie-breaking strategies for the minimum degree algorithm

In the previous section, we studied the problem of reducing the cost for sparse QR factorization, by finding good *equivalent* column reorderings of the matrix A. The initial ordering was assumed to be *any* fill-reducing ordering, computed, for example, by the minimum degree algorithm. By numerical experiments, we showed both that equivalent reorderings, in general, may give a great variation in the operation count, and that our proposed heuristics, in most cases, were able to find good reorderings. The restriction to equivalent reorderings guarantees that the resulting matrix R is, at least, as sparse as the factor produced by the initial ordering. Heuristics could therefore be designed solely with respect to the operation count or intermediate fill, without fear for increased fill in R.

This section deals with the same fundamental problem: find a column ordering of A that primarily gives low fill in R, but also reduces a measure related to sparse QR factorization; for example the operation count. Our idea is to modify the minimum degree (MD) algorithm, such that *ties* are resolved with respect to intermediate fill, the operation count for Householder QR factorization, or the total length of the Householder vectors used. The last measure equals the number of nonzero entries in the Householder matrix H, defined in Section 5.2. Ties refer to multiple solutions of the local minimization problem, where a column of *minimum degree* should be picked. The choice of column to be eliminated next is not well-defined by the MD heuristic. By resolving ties with respect to the above criteria, the resulting triangular matrix R should be sparse, while at the same time the factorization cost is reduced compared to arbitrary tie-breaking.

The minimum degree algorithm (Tinney scheme 2) by Tinney and Walker [101] is a well-known and often used *symmetric* fill-reducing method. It is a greedy heuristic, and is a symmetric version of the Markowitz algorithm (Markowitz [77]) for unsymmetric matrices. Suppose that the first k columns of the symmetric matrix $M \in \mathbf{R}^{n\times n}$ have been eliminated. In the partially eliminated matrix,

$$M_k = \begin{pmatrix} R_1^{(k)} & R_2^{(k)} \\ 0 & M^{(k)} \end{pmatrix},$$

the first block row contains the first k rows of the Cholesky factor R of M. The submatrix $M^{(k)} \in \mathbf{R}^{(n-k)\times(n-k)}$ is the active part that remains to eliminate and transform into upper triangular form. In each step, the MD algorithm chooses the next column to *modify* the least number of elements in the *next* active submatrix. Let $n_i^{(k)}$ be the number of nonzero entries in the i^{th} row and column of $M^{(k)}$, then the number of modified elements in $M^{(k+1)}$ equals $(n_i^{(k)}-1)^2$. Hence, the next column is chosen such that $n_i^{(k)}$ is minimized. In terms of the elimination game, introduced in Section 5.3, this is equivalent to choosing a vertex in $G(M^{(k)})$ of minimum degree at each step. The graph $G(M^{(k)}) = G_k(M)$ is the resulting graph after the k^{th} step of the elimination game on $G(M)$. Since the number of operations required for eliminating column i is proportional to $[n_i^{(k)}]^2$, this criterion can be interpreted also as a local minimization of the operation count.

An alternative to the above local criterion, minimizing the number of modified elements, is the *minimum fill* criterion (Tinney scheme 3). In this case, the column to be eliminated next is chosen such that the number of *new* nonzero entries is minimized. A comparison of the minimum degree and minimum fill criteria by Duff et al. [25], shows that the difference in practice is often marginal, while the minimum fill alternative is considerably more expensive in the updates.

The minimum degree method and its implementation have received much attention during the last decades. Significant improvements in efficiency have been achieved. An excellent survey of the minimum degree algorithm and ideas for its efficient implementation is given by George and Liu [43].

Let us now return to sparse QR factorization, and consider how the minimum degree algorithm can be used for computing a fill-reducing column ordering of A. Following George and Heath [40], the most natural approach is to form the structure of A^TA, and pass it to symmetric minimum degree analysis. Because of symmetry, it is sufficient to compute the upper triangular part of A^TA. Since this part is contained in the structure of R, the memory space reserved for R can be used temporarily. Extra memory should not be required. This is, however, not true in implementations where computed rows of R overwrite eliminated elements of A.

A better alternative, proposed by George and Liu [43], and implemented in MATLAB by Gilbert et al. [48], is to operate directly on the matrix A. Then the nonzero structure of A^TA need not be explicitly formed, and savings in efficiency can be expected. We show that an even more important argument is that analysis on A makes it possible to include structural properties of A. Such important structural information is lost when the analysis is based on A^TA.

In Householder QR factorization, we start with the initial matrix $A^{(0)} = A$ and systematically eliminate columns, such that the triangular factor R is finally obtained. Let Q_i denote the i^{th} orthogonal transformation. The matrix obtained after k elimination steps equals

$$Q_kQ_{k-1}\dots Q_1A = \begin{pmatrix} R_{11} & R_{12} \\ 0 & A^{(k)} \end{pmatrix},$$

where $(R_{11} \quad R_{12}) \in \mathbf{R}^{k\times n}$ contains the first k rows of the final factor R. The next orthogonal transformation is given by

$$Q_{k+1} = \begin{pmatrix} I_k & 0 \\ 0 & I - 2u_ku_k^T \end{pmatrix},$$

where I_k is the identity matrix of order k and the Householder vector u_k is chosen such that the first column of $A^{(k)}$ is eliminated.

Let $B(A^{(k)}) = (U^{(k)}, V^{(k)}, E^{(k)})$ be the bipartite graph of $A^{(k)}$. The bipartite elimination game, described in Section 5.3, defines the successive elimination graphs $B(A^{(k)}) = B_k(A)$ for $1 \le k \le n$. In each elimination step k, and for each column vertex j in $B(A^{(k)})$, let us define two sets, $S_j^{(k)}$ and $T_j^{(k)}$. The definition of these is analogous to the definition of S and T in the bipartite elimination game: $S_j^{(k)}$ is the set of row indices in column j of $A^{(k)}$, and $T_j^{(k)}$ is the set of all columns having nonzero intersection with the rows in $S_j^{(k)}$.

Since $T_j^{(k)}$ equals the set of nonzero columns in the k^{th} row of R, it follows that local fill minimization of R is achieved, in each elimination step, by choosing a column to eliminate such that $|T_j^{(k)}|$ is minimized. This is, however, equivalent to symmetric minimum degree analysis of $A^T A$. To see this, remember that $T_j^{(k)}$ is the set of all neighbors of vertex j in the elimination graph $G(M^{(k)})$, where $M = A^T A$, as mentioned in Section 5.3. We refer to the above method as *column* minimum degree.

The column version, operating directly on A, has the same properties as the symmetric version using $A^T A$. Dense rows in A cause, for example, problems also in the column version. This is, at first, a bit surprising, since important structural information about A is lost when $A^T A$ is formed. The explanation is, however, that the column version only looks on the union of column indices in the active rows, which is precisely the information used when the structure of $A^T A$ is formed.

A well-known problem in the minimum degree algorithm, is the question of tie-breaking. In a single elimination step there is often more than one vertex of minimum degree. Although such vertices solve the local optimization problem equally well, they normally lead to different global solutions. It turns out that the amount of fill in the resulting matrix R strongly depends on the way ties are resolved. If the minimum degree vertex is chosen as the first vertex in the set of local solutions, then the tie resolution is completely determined by the initial ordering of the matrix. In an experiment in MATLAB, we generated a symmetric 200×200 sparse random matrix with density 0.02. The built-in minimum degree algorithm was used on 1000 symmetric random permutations of this matrix. Figure 5.9 shows the variation in nonzero count for the resulting factors.

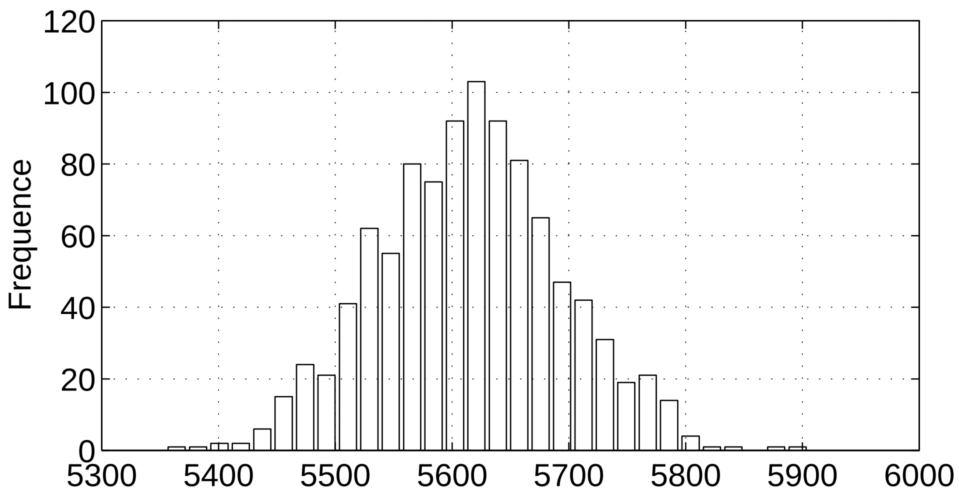


Figure 5.9: Distribution of $nnz(R)$ under minimum degree analysis of 1000 random permutations of a symmetric random matrix $M \in \mathbf{R}^{200\times 200}$.

The resolution of ties is today an open problem. An efficient and practical tie-breaking strategy would improve the generally good performance of the method, and in particular make the method more stable. To reduce the effect of the initial ordering, George and Liu [43] propose the use of preorderings. Before the minimum degree algorithm is applied, the matrix is first ordered by a method less sensitive for the initial ordering. The authors notice significant improvements on a regular grid test problem when using this preordering technique. In their case, the preordering was made by the reverse Cuthill-McKee algorithm (RCM) [20, 39, 72],

$$M \overset{RCM}{\longrightarrow} \tilde{M} = PMP^T \overset{MD}{\longrightarrow} \bar{M} = Q\tilde{M}Q^T.$$

The main reason for minimizing the fill, is to reduce the cost for computing and using the matrix factorization. The complexity of sparse algorithms depends, in general, on the number of nonzero entries in the result, rather than on the matrix dimension of the input matrix. The situation is, however, slightly different in sparse QR factorization. In this case, the operation count and the amount of memory used is rather determined by the total number of nonzero entries in the filled matrix A, as mentioned in Section 5.4. Each of these elements is either from the matrix itself or it is a part of the total fill fill. Therefore, the number of elements in the triangular factor as well as the intermediate fill must be minimized.

To solve this problem, we modify the minimum degree algorithm, such that the computed ordering both gives low fill in R and low intermediate fill. This is accomplished by the following general strategy:

> Use the minimum degree method to reduce fill in R, but resolve possible ties with respect to sparse QR factorization, by studying the structure of A.

In our setting, tie-breaking with respect to sparse QR factorization includes either the *operation count* or the *total fill* in the bipartite elimination. We also test our results regarding the *intermediate fill*, the *number of nonzeros in* R, and the *sparsity of the Householder matrix* H. These measures, all related to the *cost* for sparse QR factorization, are strongly connected.

Compared with random tie-breaking, as used in most minimum degree implementations, the proposed strategy will probably not produce more fill in R. Since, for example, fill in R and the total fill tend to be connected, it is rather likely that this strategy leads to less fill in R than random tie-breaking.

Following the general idea, we now describe two different tie-breaking strategies. Suppose that the minimum degree algorithm in a certain step encounters t columns (ties) $(c_1, c_2, ..., c_t)$ of the minimum degree d_j. The number of columns $|T_{c_i}^{(k)}|$ in $A^{(k)}$, involved in the elimination of any of these columns, is constant and equals the common degree d_j. As mentioned before, this number also equals the number of nonzero entries in the k^{th} row of R. However, the number of rows involved in the elimination depends, in general, on the column c_i to be eliminated. Involved rows are the rows of the k^{th} frontal matrix, i.e. rows in A with leading nonzero entry in the column to be eliminated, together with rows in the update matrices of the child nodes. This is true also for the traditional Householder algorithm, but then the definition of frontal matrices is different. In this case, the k^{th} frontal matrix is given by the nonzero columns in the submatrix defined by the rows in the active matrix $A^{(k)}$ with a nonzero element in the first columns. The *sets* of involved rows and columns are denoted $S_{c_i}^{(k)}$ and $T_{c_i}^{(k)}$, respectively. In terms of these notations, we consider the two tie-breaking strategies defined in Table 5.4.

Strategy	Local objective function
1	$\lvert S_{c_i}^{(k)}\rvert$
2	$\lvert S_{c_i}^{(k)}\rvert \cdot \lvert T_{c_i}^{(k)}\rvert - nnz(A^{(k)}(S_{c_i}^{(k)}, T_{c_i}^{(k)}))$.

Table 5.4: Local objective functions in the considered two tie-breaking strategies.

Here, $nnz(A(S_{c_i}^{(k)}, T_{c_i}^{(k)}))$ denotes the number of nonzero entries in the next frontal matrix. Similar to the minimum degree heuristic and the alternative minimum fill version, the purpose in Strategy 1 is to minimize the number of operations required in the next step, while Strategy 2 minimizes the fill introduced by the next elimination step. We notice that the operation count $\lvert S_{c_i}^{(k)}\rvert \cdot \lvert T_{c_i}^{(k)}\rvert$ can be minimized solely with respect to the row dimension $\lvert S_{c_i}^{(k)}\rvert$, since the ties all have the same degree.

In traditional sparse Householder QR factorization, the columns of A are eliminated from left to right, such that exactly one Householder transformation is used for each column. The only difference compared to the corresponding dense algorithm, is that rows with a nonzero element in the column to be eliminated are moved together in a frontal matrix. Similar to the multifrontal method, we then get dense subproblems. This algorithm corresponds to the update procedure of the bipartite graph described above. In particular, each elimination step reduces the size of the active matrix by one row and one column.

In the multifrontal algorithm, a *sequence* of Householder transformations is used for the elimination of a single column in A. Instead of eliminating only the first column of each frontal matrix, complete elimination is used, such that the frontal matrices become upper triangular. The size of the active matrix $A^{(k)}$ may therefore be reduced with more than one row. The number of columns is, however, always reduced by one.

The relationship between LU factorization and traditional Householder factorization, given in Theorem 5.1, makes tie-breaking methods based on the traditional Householder algorithm interesting. Closely related to tie-breaking with respect to the operation count in QR factorization, is tie-breaking with respect to fill in H. A sparse H reduces the upper bound for the fill in LU factorization with partial pivoting. Since the row dimension $|S_{c_i}|$ of the frontal matrix associated with column c_i, is the length of the Householder vector to be used, Strategy 1 can be seen as a local method for minimizing the number of nonzero elements in H.

By the presented two tie-breaking strategies and two update procedures, we have altogether four alternative minimum degree implementations. These are summarized in Table 5.5.

Update procedure	Strategy 1	Strategy 2
Traditional Householder	T_1	T_2
Multifront	M_1	M_2

Table 5.5: The consider tie-breaking strategies and elimination procedures.

To compare the considered tie-breaking strategies, we have implemented four versions of the minimum degree algorithm as *m-files* in MATLAB. The previously presented set of test matrices from the Harwell-Boeing collection was used to compare the operation count and intermediate fill. In Table 5.6, we compare the operation count for *multifrontal* QR factorization under the different tie-breaking strategies to the *tight colmmd* ordering of MATLAB, which produces a strict minimum degree ordering. The results for *default colmmd* were given in Table 5.2 of Section 5.3, and the reader can thus compare the results of Table 5.6 also with *default colmmd*. We conclude that Strategy 2 is the best. Compared with *tight colmmd*, it reduces the operation count with up to 20%. The results of Table 5.6 are also presented graphically in Figure 5.10, in the same way as in Section 5.3.

The choice of update procedure, M or T, is less important for the resulting ordering. Although only minor differences are noticed, multifrontal updates tend to be superior to traditional Householder updates. Strategy 2 under multifrontal update, M_2, is in general very stable. With one exception, it always gives a good reduction compared with *tight colmmd*.

Our main goal is to minimize the operation count for sparse QR factorization. This can be done by breaking ties either with respect to the operation count directly, or with respect to total fill. We believe that low total fill, in general, leads to low operation count. The sparsity of H and intermediate fill are closely connected, because of the relationship between H and the lower trapezoidal part of the filled matrix of A explained earlier. Table 5.7 compares intermediate fill resulting from *tight colmmd* to strategy M_2, for the multifrontal algorithm. We conclude that the proposed strategy, in almost all cases, reduces the intermediate fill as well as the fill in R for the multifrontal algorithm. The relative reduction of the intermediate fill

Matrix	tight colmmd	tie-breaking			
		T_1	T_2	M_1	M_2
ABB313	300333	254656	254287	254571	254571
ASH219	66190	82799	75928	82676	82527
ASH331	188370	160485	160260	170897	170500
ASH608	532812	445708	442155	464340	429123
ASH958	793318	814998	752251	802115	709836
WL1033	732439	691041	653789	618437	619511
WL1252	4027343	3574958	3291321	3569156	3337027
WL1364	5834049	5373185	5199605	5342485	5166734
WL1641	13164543	11450903	12385206	11012698	10707118
WL1850	3589752	3258542	3261045	3255772	3274698
WL1991	41831054	43752476	40541055	40046878	40657588
WL2808	135554156	144759635	117885825	149977522	110817212

Table 5.6: Each entry in the table shows the number of flops achieved in the multifrontal factorization in each of the cases where the matrix is column permuted by *tight colmmd* of MATLAB and by minimum degree with our tie-breaking strategies.

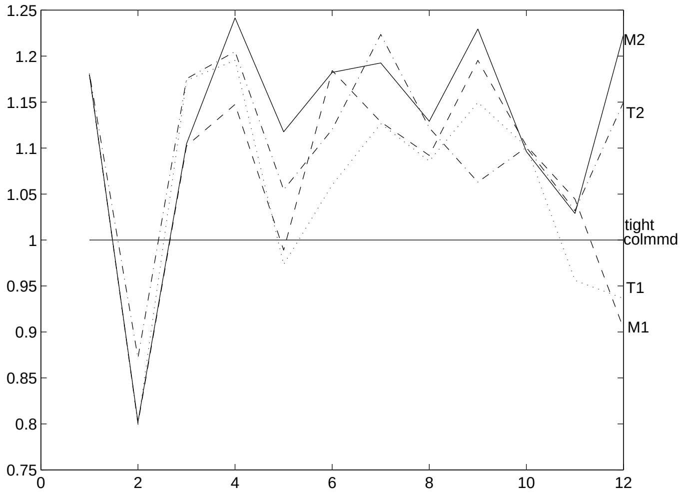


Figure 5.10: Graphic presentation of the results of Table 5.6. Tie-breaking strategies, T_1, T_2, M_1, and M_2 are shown in respectively dotted, dash-dotted, dashed, and solid line styles.

does not differ much from the relative reduction of the operation count.

Matrix	tight colmmd		tie-break M_2	
	intermed. fill	nonzeros in R	intermed. fill	nonzeros in R
ABB313	2693	1683	2525	1630
ASH219	1254	514	1376	544
ASH331	2594	803	2483	784
ASH608	5751	1694	5419	1630
ASH958	8966	2617	8687	2578
WL1033	8511	2607	6816	2572
WL1252	22683	3819	17821	3722
WL1364	27679	4225	23102	4152
WL1641	44435	5640	35779	5412
WL1850	23366	7504	21987	7372
WL1991	71166	9492	66441	9406
WL2808	126568	20420	107984	19533

Table 5.7: The tie-breaking method M_2 is here compared to *tight colmmd* regarding the amount of intermediate fill and the number of nonzeros in the resulting factor R, in the multifrontal method.

Finally, we consider how the tie-breaking strategies perform when used for sparse QR factorization by the traditional Householder algorithm. The operation count for orderings computed by *tight colmmd* in MATLAB, and by the four strategies, are presented in Table 5.8. Related results for *default colmmd* of MATLAB were given in Table 5.3 of Section 5.3. In this case, the choice of update procedure, T or M, seems to play a more important role than the choice of tie-breaking strategy, 1 or 2. T-strategies, based on traditional Householder algorithm, give the best results. However, T_2 is superior to T_1.

A graphical presentation of the results of Table 5.8 is given in Figure 5.11. From this presentation, it is clear that for our test matrices, Strategy T_2 always gives better results than the initial *tight colmmd* ordering. Except for one matrix, it always gives operation count lower than, or close to, those resulting from other strategies. Comparing Tables 5.6 and 5.8, we see that the gain from tie-breaking is even higher for the traditional Householder algorithm than for the multifrontal algorithm. This can possibly be explained by the fact that there is usually more intermediate fill involved in the traditional Householder method, and therefore there might be more to gain by trying to reduce it.

The Householder matrix H from traditional Householder plays an important role

Matrix	tight colmmd	tie-breaking			
		T_1	T_2	M_1	M_2
ABB313	862456	802100	790677	814532	808998
ASH219	198175	206009	197679	221467	221083
ASH331	525767	442207	441927	490492	491819
ASH608	1882747	1468051	1448897	1625482	1543707
ASH958	4260804	3576794	3499592	3934391	3476401
WL1033	3259344	2936409	2928559	2948021	2945344
WL1252	8465255	7154759	6625171	7163477	6839454
WL1364	12241484	9289946	9288404	9969582	9663951
WL1641	22873841	17356585	18077448	17074184	17074184
WL1850	20463821	14842492	14809787	14956083	15347495
WL1991	52763733	44912713	44778023	44238047	46199981
WL2808	139087318	108000550	99155210	118688482	106923782

Table 5.8: Tie-breaking strategies compared to *tight colmmd* regarding the number of operations in traditional Householder QR factorization.

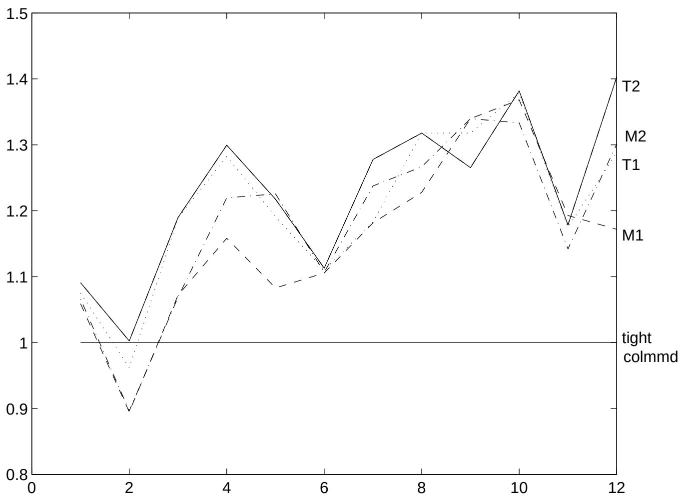


Figure 5.11: Graphic presentation of the results of Table 5.8. Tie-breaking strategies, T_1, T_2, M_1, and M_2 are shown in respectively dotted, solid, dashed, and dash-dotted line styles.

as an upper bound for the fill in sparse LU factorization. In Table 5.9, we compare the number of nonzero entries in H when *tight colmmd* and strategy T_2 are used. In this case, the proposed strategy gives a reduction with about 15%.

Matrix	tight colmmd	T_2
ABB313	7252	6846
ASH219	2557	2620
ASH331	6091	5301
ASH608	14890	12422
ASH958	30618	26326
WL1033	17190	14870
WL1252	42173	36590
WL1364	53422	45067
WL1641	88516	75101
WL1850	56695	50828
WL1991	159023	146162
WL2808	268360	248886

Table 5.9: The tie-breaking method T_2 compared with *tight colmmd* regarding the number of nonzero entries in the Householder matrix H (traditional Householder).

5.6 Concluding remarks

In this paper, we have studied two approaches for finding good column orderings for sparse QR factorization:

1. Starting with an initial fill-reducing ordering, find a more suitable equivalent reordering, by studying the structure of A.
2. Resolve ties in minimum degree on A^TA with respect to the structure of A.

For equivalent reorderings, we have proposed heuristic methods that generally reduce the number of operations considerably. For each test matrix, at least one of the heuristics gives, in each case, up to 10% improvement compared with the initial fill-reducing ordering.

Even more useful for practical applications, is tie-breaking in minimum degree. The idea of resolving ties in minimum degree with respect to QR factorization, turns out to be very fruitful. By incorporating the proposed tie-breaking strategies in the symmetric minimum degree analysis on A^TA, we reduced the cost for sparse QR factorization of A with up to 20%. This is perhaps one of the most interesting results of this paper.

In the paper, we also focussed on the importance of studying the structure of A, in addition to $A^T A$, when trying to find good column orderings for QR factorization of A. This is an important problem for both multifrontal and traditional Householder QR factorization.

Some theoretical questions related to equivalent reorderings remain as open problems. We conjecture, and would like to prove, that the following two problems are NP-hard:

1. Find an equivalent reordering that minimizes the amount of total fill in the bipartite elimination game.

2. Find the best equivalent reordering in terms of the operation count for sparse QR factorization.

Our future research plans in this field include a study of the above complexity issues. In this paper, we have not concentrated on the efficient implementation of the proposed heuristics. We have rather been interested in the produced numerical results. The same is true for the tie-breaking methods. The implementation details of these heuristics are definitely subject to further research. Since we use simple MATLAB codes for our experiments, testing large matrices is time-consuming. Extended numerical comparisons, on larger matrices, require a more efficient code.

Finally, a very important and interesting problem is to find an efficient column ordering algorithm, suitable for sparse QR factorization, that works directly on A.

Chapter 6

Making an Arbitrary Filled Graph Minimal by Removing Fill Edges

In this chapter we study the problem of reducing fill. Finding minimum fill orderings is NP-hard for general graphs, whereas it is easy for chordal graphs. In Chapter 3, we used the approach of finding classes of graphs that have polynomial solutions while considering the generally NP-hard minimum elimination tree height problem. In this chapter, our approach will be in the direction of suggesting methods for general graphs. Instead of *minimum* fill, we consider the polynomially solvable problem of finding *minimal* fill orderings. Usually, an efficient heuristic is used to produce a low fill ordering of a given graph. In addition, one is often interested in a minimal fill ordering, that is, no proper subgraph of the resulting filled graph can *both* be chordal *and* contain all the original edges. We give an algorithm that removes fill edges from a given filled graph until it becomes minimal.

As we mentioned in the previous chapter, perfect elimination orderings of a minimal filled graph, applied on the original graph, all result in the same filled graph, which is an advantage of minimal orderings. For an arbitrary filled graph, both its minimalizations by the method we present in this chapter, and its equivalent reorderings applied on the original graph, result in subgraphs of the initial filled graph. Therefore, we find it useful to explain the connection to equivalent reorderings in this chapter.

This chapter presents a revised and extended version of the following research paper: [8] J. R. S. Blair, P. Heggernes, and J. A. Telle, *Making an Arbitrary Filled Graph Minimal by Removing Fill Edges.* The authors are indebted to Barry Peyton for suggesting the problem of finding a minimal chordal supergraph from a non-minimal chordal supergraph.

6.1 Introduction

For any graph G and an ordering α of its vertices, there is an associated set of edges called the *fill* that, when added to G, results in a chordal graph $(G;\alpha)$, which is formally defined in Section 6.2. The goal of finding orderings of the vertices that produce low fill has been studied by researchers in several areas of computer science, like the solution of sparse symmetric systems of linear equations [41, 66, 80, 91, 92], data-base management systems [6, 98], knowledge-based systems [34, 64], and computer vision [15]. This problem still remains as an important research problem.

In a central 1976 paper, Rose, Tarjan, and Lueker [92] gave an algorithm which finds a *minimal* fill ordering of a graph G in $O(ne)$ time, where n and e are respectively the number of vertices and edges in G. To date, this algorithm, which is called LEX-M, has the best known running time for finding a minimal chordal supergraph of an arbitrary graph. An efficient parallel algorithm for the same problem is given by Dahlhaus and Karpinski [21]. In 1981, Yannakakis [103] showed that finding the *minimum* fill for an arbitrary graph is NP-hard. Already long before this result, several heuristics had been proposed for finding elimination orderings producing small fill. The two most famous and practically useful methods are called *minimum degree* and *nested dissection* (see [41] for a survey). For graphs of bounded maximum degree, it is shown by Agrawal et al. [1], that nested dissection based on near-optimal vertex separators gives fill which is within a polylogarithmic factor of the minimum fill.

We consider the more restricted problem of, given a graph G and an arbitrary ordering α, finding a graph M that is both a minimal chordal supergraph of G, and a subgraph of the filled graph $(G;\alpha)$. We also find a related minimal ordering β so that $(G;\beta) = M$. Minimal orderings are desirable since any perfect elimination ordering of the resulting filled graph, when applied to the original graph, produces the same filled graph. In other words, if β is a minimal ordering of a graph G then for any perfect elimination ordering γ of $(G;\beta)$ we have $(G;\gamma) = (G;\beta)$. This property does not hold for elimination orderings in general. The problem we consider is motivated by the following two facts: 1. Minimal orderings are not necessarily close to minimum in general, and lower fill is usually achieved by practical fill-reducing heuristic algorithms like minimum degree and nested dissection. 2. These famous heuristic algorithms usually produce non-minimal fill. Therefore, a suitable approach in order to get a minimal low fill ordering of a graph G, is to first find a non-minimal low fill ordering α produced by a heuristic algorithm, and then run our algorithm which removes redundant fill until the remaining fill is minimal.

The main contributions of this paper are two-fold: First, our Theorem 6.2 shows that by considering the fill edges in the reverse order to that in which they were introduced, each edge need only be considered as a candidate for removal once. This property is significant since, in general, the removal of a fill edge can both add to and delete from the set of remaining "removable edges." Second, using this result, we develop an $O(f(e+f))$ time algorithm that, given both G and α, will produce an ordering β such that the filled graph $(G;\beta)$ is minimal and is a subgraph of $(G;\alpha)$. Here, f and e are respectively the number of fill edges in $(G;\alpha)$ and the number of edges in G. Furthermore, the time complexity of our algorithm depends on structural properties of the filled graph, and we suspect that it might be in many cases better than the worst-case bound of $O(f(e+f))$.

The paper is organized as follows. Section 6.2 formally defines terms used in the paper. Section 6.3 characterizes redundant fill edges. The new algorithm and its proof of correctness are found in Section 6.4 with an analysis of the time complexity. A connection to equivalent reorderings is given in Section 6.5. Section 6.6 contains concluding remarks and some questions raised by our results.

6.2 Definitions and Notation

We start with some standard graph terminology. We consider undirected, simple graphs. For a graph G, the vertex and edge sets are denoted by respectively $V(G)$ and $E(G)$. $N_G(v)$ is the set of neighbors of v in G. 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 $v_1v_2, v_2v_3, ..., v_{k-1}v_k$. We also use $v_1 \to v_k$ to denote a path from v_1 to v_k. A *cycle* is a path whose first and last vertices are the same. 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. For a set S of vertices in G, the subgraph of G induced by S is denoted by $G[S]$. The graph $(G-S)$ is the graph $G[V(G)-S]$. For a set K of edges in G, the graph $(G-K)$ is the result of removing the edges in K from G. A *supergraph* of G is a graph which contains G as a subgraph.

For a graph G with $|V(G)| = n$, an *elimination ordering* of G is a bijection $\alpha : V(G) \leftrightarrow \{1, 2, ..., n\}$. For ease of presentation, we will also refer to α as a sequence, $\alpha = v_1, v_2, ..., v_n = \alpha^{-1}(1), \alpha^{-1}(2), ..., \alpha^{-1}(n)$. Thus v_i denotes the vertex v such that $\alpha(v) = i$.

Associated with α is a sequence of supergraphs of G, defined as follows: $G_0 = G$, and for $1 \le i \le n$, G_i is the graph obtained by adding edges to G_{i-1} so that all vertices in $N_{G_{i-1}}(v_i) \cap \{v_{i+1}, ...v_n\}$ become pairwise adjacent. This step is called the *elimination* of vertex v_i, and the whole process of obtaining G_n from G is called the *elimination process*. Note that vertices are not removed from the graph: $V(G) = V(G_i)$, $1 \le i \le n$. Usually, the elimination process and the graphs G_i are defined such that vertex v_i is removed after its elimination (thereby the term *elimination*). However, for our purposes and algorithm, the non-shrinking graphs G_i defined here are more appropriate. The new edges added are called *fill edges*, and $F_i = E(G_i) - E(G_{i-1})$ is the set of fill edges created by the elimination of v_i. We use C_i to denote the resulting clique induced by v_i and its higher numbered neighbors. The graph G_n is the *filled graph* of G for elimination ordering α. We also use $(G;\alpha)$ to denote the filled graph G_n. All filled graphs are chordal.

An elimination ordering α on G is *minimal* if the resulting $(G;\alpha)$ is a *minimal chordal* supergraph of G. In other words, no strict subgraph of $(G;\alpha)$ containing G is chordal, equivalently, for no ordering γ is $(G;\gamma)$ a strict subgraph of $(G;\alpha)$. Then $(G;\alpha)$ is also referred to as a *minimal filled graph* of G. An elimination ordering α on G is *minimum* if no other elimination ordering β on G can produce a filled graph with with fewer edges than $(G;\alpha)$. Minimum orderings are, of course, minimal.

A vertex is *simplicial* in a graph G if its neighbors induce a clique. Note that v_i is simplicial in $G_{i-1}[\{v_i, ..., v_n\}]$ if and only if $F_i = \emptyset$. Every chordal graph that is not a complete graph, has at least two simplicial vertices. An elimination ordering on G is *perfect* if no fill edges are created, that is, if $F_i = \emptyset$ for $1 \leq i \leq n$. Chordal graphs are exactly the class of graphs that have perfect elimination orderings.

6.3 Greedy Removal of Redundant Fill Edges

In this section we develop some properties of fill edges related to the order of their introduction, with the goal of finding an algorithm to remove redundant fill edges. The following result from [92] gives another characterization of fill edges.

Lemma 6.1 *Let G be a graph and α an elimination ordering of its vertices. Then uv is an edge of $(G;\alpha)$ if and only if $uv \in E(G)$ or there exists a path $u = u_0, u_1, ..., u_{k+1} = v$ in G such that, for $1 \leq i \leq k$, we have $\alpha(u_i) < \min\{\alpha(u), \alpha(v)\}$.*

The same paper also gives the following alternative characterization of minimal elimination orderings.

Theorem 6.1 *Let G be a graph and α an elimination ordering of its vertices. Then α is a minimal elimination ordering if and only if each fill edge is the unique chord of a 4-cycle in $(G;\alpha)$.*

We will need the following corollary.

Corollary 6.1 *Let G be a graph and $\alpha = v_1, ..., v_n$ be an elimination ordering of its vertices. If a fill edge $uv \in F_i$, created by elimination of v_i, is the unique chord of a 4-cycle in $(G;\alpha)$, then uv is the unique chord of a 4-cycle u, v_i, v, x, u in $(G;\alpha)$ with $\alpha(x) > i, \alpha(u) > i$ and $\alpha(v) > i$.*

Proof: Let u, y, v, x, u be a 4-cycle of $(G;\alpha)$ in which uv is the only chord. Since uv is created by elimination of v_i we must have $\alpha(u) > i$, $\alpha(v) > i$, $\alpha(y) \geq i$ and $\alpha(x) \geq i$. If either $y = v_i$ or $x = v_i$ we are done. If this is not the case, note that either yv_i or xv_i is not an edge of $(G;\alpha)$ since otherwise we would have xy a fill edge. As u and v are both neighbors of v_i, we can find a 4-cycle as stated in the corollary. ∎

Observe that Corollary 6.1 and Theorem 6.1 can be used to detect whether a given elimination ordering is minimal.

Definition 6.1 *Let G' be a chordal supergraph of a graph G. A* candidate *edge is an edge $uv \in (E(G') - E(G))$ which is not the unique chord of any 4-cycle in G'.*

Since removing a single edge from a chordal graph results in either a chordal graph or a graph with a chordless 4-cycle, it is clear that removing any candidate edge cannot destroy chordality. The following result is implicit in [92].

Proposition 6.1 *A non-minimal chordal supergraph G' of a graph G has at least one candidate edge. Any candidate edge can be removed without destroying chordality.*

This result immediately leads to an algorithm for finding a minimal filled graph by removing fill edges from a filled graph: repeatedly remove candidate edges until no candidate edges remain. Unfortunately, the set of candidate edges changes as edges are removed, with new candidate edges being introduced and old candidate edges ceasing to be candidates. Therefore, if we are to remove the candidate edges in an arbitrary order, each fill edge might have to be checked several times for candidacy. This can be illustrated by the simple example shown in Figure 6.1.

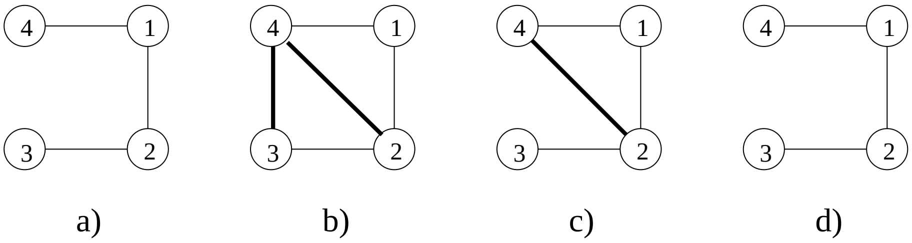


Figure 6.1: An example illustrating the non-monotonicity property of candidate edge removal in arbitrary order.

The original graph G, which is a simple path, and the ordering on G are shown in Figure 6.1 a). The filled graph is given in b). Looking at b), the fill edge (2,4) is not a candidate for removal, but (3,4) is. However, after the removal of (3,4), as shown in c), the fill edge (2,4) becomes a candidate, and can be removed to get a minimal filled graph equal to the original graph as shown in d). Thus in this simple example, the edge (2,4) had to be examined twice for candidacy.

This non-monotonicity property of the set of candidate edges seems to be the main obstacle in obtaining an efficient implementation of this algorithm. As we show below, we can partially avoid this by processing fill edges in the reverse fill order of that in which they were introduced. In particular, the following strategy will produce a minimal chordal supergraph:

for $i = n$ **downto** 1 **do**
    while there is a candidate edge in F_i **do** remove it from the filled graph;

We first give a lemma and then a theorem that will provide the basis for the correctness of this strategy.

Lemma 6.2 *Let $\alpha = v_1, ..., v_n$ be an elimination ordering of a graph G. Let M_i be both a subgraph of G_n and a minimal chordal supergraph of G_{i-1}. Then M_i has a perfect elimination ordering $\gamma = w_1, w_2, ..., w_n$ with $(G;\gamma) = M_i$ and $v_k = w_k$ for $k = 1, ..., i-1$.*

Proof: Since both G_{i-1} and G_n are associated with $\alpha = v_1, ..., v_n$, we have, for $k = 1, ..., i-1$, the vertex v_k simplicial in both the graphs $G_{i-1}[\{v_k, v_{k+1}, ..., v_n\}]$ and $G_n[\{v_k, v_{k+1}, ..., v_n\}]$. Note also that v_k has the same neighbors in both these graphs.

Hence v_k is also simplicial in $M_i[\{v_k, v_{k+1}, ..., v_n\}]$ since M_i is a supergraph of G_{i-1} and a subgraph of G_n. For chordal graphs, it is well-known that the elimination ordering resulting from repeatedly eliminating a vertex which is simplicial in the graph induced by non-eliminated vertices, is a perfect elimination ordering. We can therefore find a perfect elimination ordering γ of M_i with $\gamma = w_1, ..., w_n$ and $v_k = w_k$ for $k = 1, ..., i-1$.

We now show that $(G;\gamma) = M_i$. Eliminating $v_1, ..., v_{i-1}$ in this order in G or G_{i-1} will in either case give us G_{i-1}. Since M_i is a minimal chordal supergraph of G_{i-1}, we must have $(G_{i-1};\gamma) = M_i$. Hence, $(G;\gamma) = M_i$ since the initial sequence of γ is $v_1, ..., v_{i-1}$. ∎

We can now state the main theorem of this section.

Theorem 6.2 *Let $\alpha = v_1, ..., v_n$ be an elimination ordering of a graph G. Let M_i be both a subgraph of $(G;\alpha)$ and a minimal chordal supergraph of G_{i-1}. For any graph M which is both a subgraph of M_i and a chordal supergraph of G, we have $(E(M_i) - E(G_{i-1})) \subseteq E(M)$.*

Proof: Observe that $E(G) \subseteq E(G_{i-1}) \subseteq E(M_i) \subseteq E(G;\alpha)$ and $E(G) \subseteq E(M) \subseteq E(M_i)$. By Lemma 6.2, we know that M_i has a perfect elimination ordering $\gamma = w_1, w_2, ..., w_n$ with $v_k = w_k$ for $k = 1, ..., i-1$, and we also know that $(G_{i-1};\gamma) = M_i$. The statement in the theorem says that if the edge uv of M_i is a fill edge created, in the process giving $(G_{i-1};\gamma) = M_i$ from G_{i-1}, by the elimination of some w_j with $i \leq j \leq n$, then uv must be an edge of M.

We prove this by contradiction. Assume that uv is the latest introduced fill edge violating the condition. In other words, uv is created in the process giving

$(G_{i-1};\gamma) = M_i$ by $w = w_j$, uv is not an edge of M, and every other fill edge of this elimination process, created by any w_k with $k > j$, is an edge of M. Since M_i is a minimal chordal supergraph of G_{i-1}, and uv is a fill edge created by w, by Theorem 6.1 and Corollary 6.1 there must exist a vertex x which in M_i is not adjacent to w but is adjacent to both u and v, with $\gamma(u) > \gamma(w)$, $\gamma(v) > \gamma(w)$ and $\gamma(x) > \gamma(w)$.

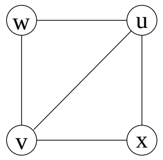


Figure 6.2: The subgraph of M_i induced by the vertices w, u, x, and v.

We will show that there exists in $M, whichisasubgraphof M_i$, a path from w to x whose internal vertices have γ-order less than w and x. We will thereby arrive at a contradiction since, with γ being a perfect elimination ordering of M_i, such a path would imply, by Lemma 6.1, that wx is an edge of M_i. But w and x are not adjacent in M_i.

Since neither uv nor wx are edges in the chordal graph M, at least one of wu, ux, xv and vw is not an edge of M. But M is a supergraph of G, so any of wu, ux, xv and vw which is not an edge of M must be a fill edge in the elimination process $(G;\gamma)$ giving M_i. By assumption, uv is the latest fill edge in this elimination process which is not in M, so any of wu, ux, xv and vw which is not an edge of M, must be a fill edge of M_i created earlier than uv. (Note that it could not be created at the same time as uv, since uv is a result of eliminating w.) Therefore, by Lemma 6.1 there must exist paths $w \to u, u \to x, x \to v$, and $v \to w$ in G, where $E(G) \subseteq E(M)$, whose internal vertices (if any) are earlier in the γ-order than w and thus also earlier than u, x, v. Consider the combined paths $u \to w \to v$ and $u \to x \to v$. If these two paths intersect in an internal vertex y then we find a path $w \to y \to x$ giving the desired contradiction. Otherwise, consider the shortest path $u \to w' \to v$ in M from u to v using only vertices on the $u \to w \to v$ path and the shortest path $u \to x' \to v$ in M from u to v using only vertices on the $u \to x \to v$ path. These paths must have an internal vertex since uv is not an edge of M, and the vertices on these paths therefore induce a subgraph of M containing a k-cycle with $k \geq 4$. Since M is chordal, there is a chord on this cycle and this chord must connect an internal vertex y' in $u \to x' \to v$ with an internal vertex y'' in $u \to w' \to v$. But then we find a path $w \to y' \to y'' \to x$ giving the desired contradiction. (Note that this argument allows for $w = y'$ or $x = y''$.) ∎

From Theorem 6.2, it follows that the fill edges in F_i that are not removed while examining F_i, will never become candidates for removal at later steps while examining F_j for $j = i - 1, ..., 1$. We are now ready to give a full description of the algorithm in the next section.

6.4 The Algorithm

In this section, we develop an algorithm that, given a graph G and an elimination ordering α of G, finds a graph M which is both a minimal chordal supergraph of G and also a subgraph of $(G; \alpha)$.

After computing $G_n \equiv (G; \alpha)$, and finding C_i and F_i for $i = 1, ..., n$ from G_n in a straightforward manner, our algorithm proceeds as follows: The algorithm has n iterations. Initially, we set $M = G_n$. Starting from $i = n$ and going backwards, at each iteration i, redundant fill edges in F_i introduced by the elimination of v_i are removed. By Theorem 6.2, we know that the remaining edges of F_i need not be considered for removal at later iterations. The algorithm has n iterations for simplicity, but actually $n - 3$ would suffice since F_n and F_{n-1} are empty, and F_{n-2} is either empty or singleton, and can always be removed.

The full algorithm is given in Figure 6.3. The subroutine that checks an edge $uv \in F_i$ for its candidacy for removal is called *CandidateEdge*, and is based on Corollary 6.1. The subroutine LEX-M is used to decide which of the candidate edges in F_i can be removed and which must stay to preserve chordality. W_i is the subgraph of C_i on which we run LEX-M to find out which edges in $Candidate(i)$ are necessary. The original LEX-M was introduced by Rose, Tarjan and Lueker in [92], and finds an elimination ordering resulting in a minimal chordal supergraph of a given graph. Our version of LEX-M(W_i) returns the set $KeepFill(i) \subseteq Candidate(i)$ of fill edges whose addition to W_i produces a minimal chordal supergraph of W_i. Note that $W_i \cup Candidate(i)$, if nonempty, is a clique and a subgraph of C_i.

Since the resulting graph M is a minimal chordal supergraph of G, every perfect elimination ordering β of M gives $(G; \beta) = M$. A linear-time algorithm for finding a perfect elimination ordering of a chordal graph is given in [92], and is called LEX-P. Figure 6.4 illustrates how our algorithm processes a graph on 7 edges, with details of the calls of LEX-M in Figure 6.5. We first prove correctness of the algorithm and then consider its time complexity.

Theorem 6.3 *Algorithm MinimalChordal on input G and $\alpha = v_1, ..., v_n$ finds a graph M which is both a minimal chordal supergraph of G and a subgraph of $(G; \alpha)$.*

Proof: Denote the graph M at the beginning of iteration i of the main loop of the algorithm by M_{i+1}, and at the end of iteration i by M_i. Since $E(M_i) \subseteq E(M_{i+1})$ and M_{n+1} is initialized to $G_n \equiv (G; \alpha)$ it is clear that each M_i is a subgraph of $(G; \alpha)$. In addition, the algorithm has the loop invariant: "The graph M_i is a minimal chordal supergraph of G_{i-1}." We show this by reverse induction on i from $n + 1$ to 1. The

Algorithm $MinimalChordal(G, \alpha)$;
Input: A graph G and an elimination ordering $\alpha = v_1, ..., v_n$ of G.
Output: 1. A chordal graph M which is both a minimal chordal supergraph of G and a subgraph of the filled graph $(G; \alpha)$.
2. A minimal elimination ordering β of G s.t. M is the filled graph $(G; \beta)$.

```
begin
    Find (G; α) and C_i, F_i for i = 1, 2, ..., n;
    M = (G; α);
    for i = n downto 1 do
        Candidate(i) = ∅;
        Incident(i) = ∅;
        for all edges uv ∈ F_i do
            if CandidateEdge(uv, i, M) then
                Candidate(i) = Candidate(i) ∪ {uv};
                Incident(i) = Incident(i) ∪ {u, v};
            end-if;
        if Candidate(i) ≠ ∅ then
            W_i = C_i[Incident(i)] − Candidate(i);
            KeepFill(i) = LEX-M(W_i);
            M = M − (Candidate(i) − KeepFill(i));
        end-if;
    end-for;
    return M and β = LEX-P(M);
end;
```

Function $CandidateEdge(uv, i, M)$: **boolean**;
Input: An edge $uv \in F_i$ and the graph M.
Output: Returns **true** if uv is a candidate to be removed from M, **false** o.w.

```
begin
    cand = true;
    for each neighbor x of u do
        if α(x) > i and xv ∈ E(M) and xv_i ∉ E(M) then
            cand = false;
    return cand;
end;
```

Figure 6.3: Algorithm *MinimalChordal* and Function *CandidateEdge*.

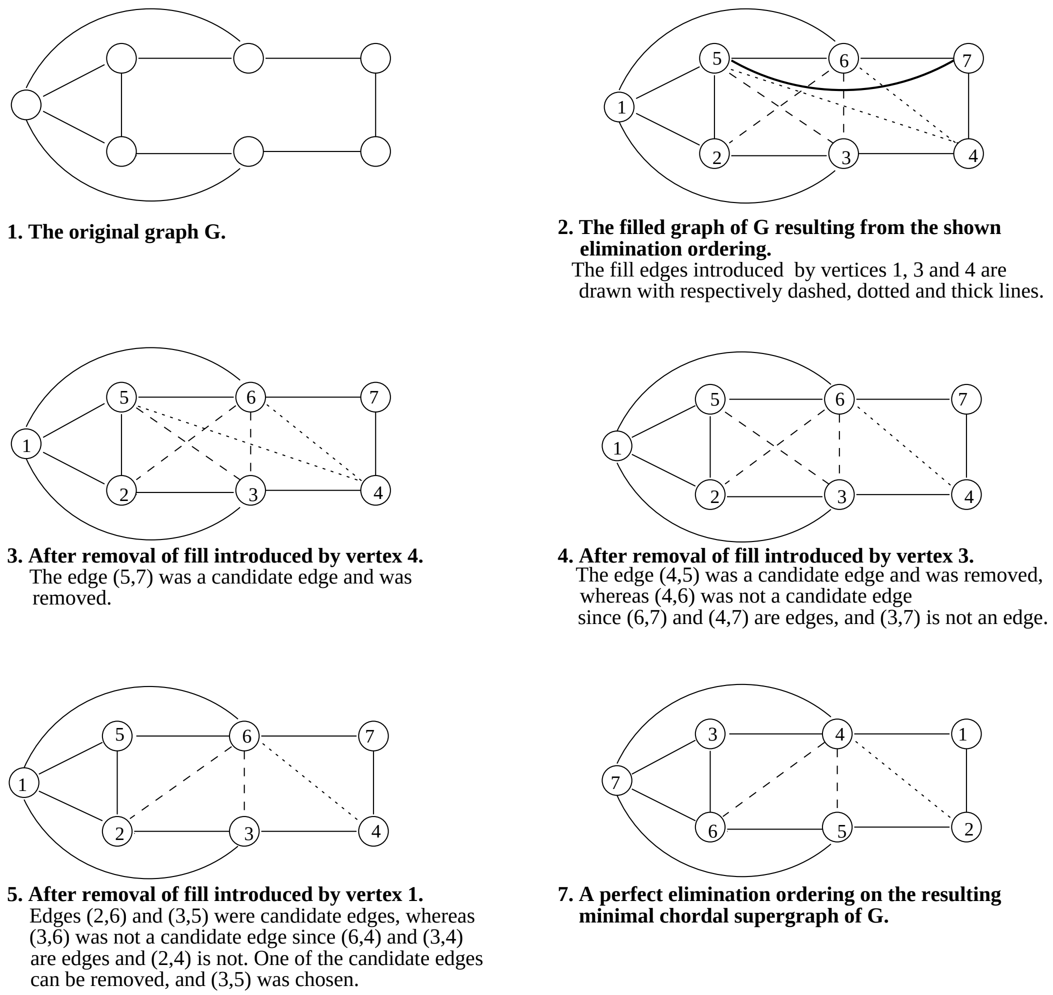


Figure 6.4: An example illustrating the algorithm. The calls of LEX-M in Steps 4 and 5 are shown in detail in Figure 6.5.

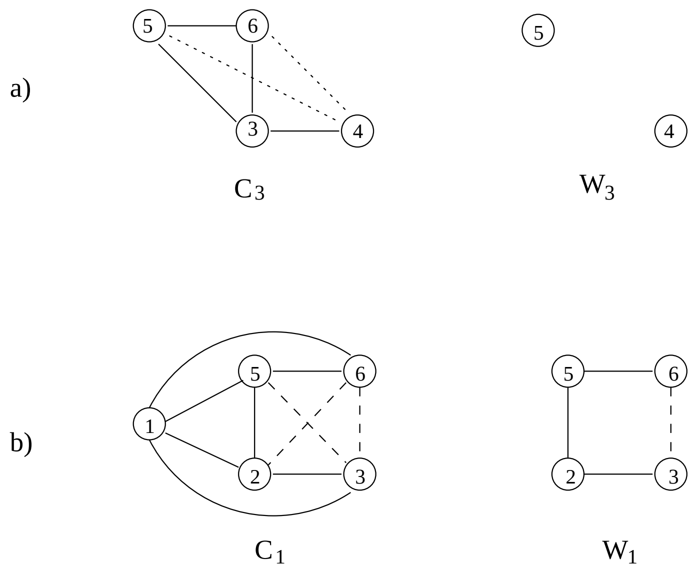


Figure 6.5: Steps 4 and 5 of the previous example (Figure 6.4) shown in detail. a) $Candidate(3) = \{(4,5)\}$, and $Incident(3) = \{4,5\}$. The subgraph W_3 is defined as $C_3[Incident(3)] - Candidate(3)$. LEX-M($W_3$) returns no fill, hence $(4,5)$ is removed from M. b) $Candidate(1) = \{(2,6),(3,5)\}$, and $Incident(1) = \{2,3,5,6\}$. LEX-M($W_1$) returns $(2,6)$ as fill, thus only $(3,5)$ can be removed from M as redundant fill.

loop invariant is clearly true initially for the graph $M_{n+1} = G_n$. The edges $E(M_{i+1})$ are of four types:

(1) edges belonging to the original graph G.

(2) fill edges that were introduced before the elimination of v_i which will be considered for removal at later iterations.

(3) fill edges that were introduced after the elimination of v_i which have not been removed at earlier iterations.

(4) fill edges belonging to F_i.

The graphs G_i and G_{i-1} both contain all the edges of types 1 and 2, so these edges must belong to any chordal supergraph of G_{i-1}. Since M_{i+1} is a subgraph of G_n and a minimal chordal supergraph of G_i we know by Theorem 6.2 that no edges of Type 3 can be removed from M_{i+1} and still give a chordal supergraph of G, where $E(G) \subseteq E(G_{i-1})$.

Hence, we need only show that (A) M_i is chordal and that (B) any edge uv of F_i which remains in M_i is not a candidate for removal. Consider Case (B) first. By Theorem 6.1 it suffices to show that any such edge uv is the only chord of a 4-cycle in M_i. There are two cases: either (B.1) $uv \in Candidate(i)$ or (B.2) not.

(B.1) At the beginning of iteration i, the neighbors among $\{v_{i+1}, ..., v_n\}$ in M_{i+1} of vertex v_i induce a clique C_i, and the fill edges F_i are all contained in this clique. The edges in F_i that are candidates for removal are identified as $Candidate(i)$, and the vertices which are incident to at least one candidate edge, are stored in $Incident(i)$. We set W_i to be the subgraph of C_i induced by the vertices in $Incident(i)$ minus the edges in $Candidate(i)$. Candidate edges of F_i may be removed after the call of LEX-M(W_i), which gives us a set $KeepFill(i) \subseteq Candidate(i)$ of fill edges that when added to W_i gives a minimal chordal supergraph of W_i. The edges in $(Candidate(i) - KeepFill(i))$ are then removed from M_{i+1} to give us M_i. An edge $uv \in KeepFill(i)$ is guaranteed, by the correctness of algorithm LEX-M and Theorem 6.1, to be a unique chord of a 4-cycle in the graph $W_i \cup KeepFill(i)$ and therefore also in M_i, since $W_i \cup KeepFill(i)$ is an induced subgraph of M_i.

(B.2) If uv is found not to be a candidate in the call $CandidateEdge(uv, i, M_{i+1})$, then this is because there exists a vertex x which in M_{i+1} is a neighbor of both u and v but is not a neighbor of v_i. Note that the 4-cycle v_i, u, x, v, v_i contains only edges of Type 1, 2 or 3 and uv is therefore a unique chord of this 4-cycle also in M_i.

(A) It remains to show that M_i is chordal, which we do by contradiction. Let S be the set of vertices on a shortest chordless cycle of length at least 4 of M_i. We know that there must exist at least two vertices a, c in S such that $ac \in (Candidate(i) - KeepFill(i))$, since $M_{i+1} = M_i \cup (Candidate(i) - KeepFill(i))$ is chordal. Also S must contain at least one vertex $b \notin V(C_i)$ since the graph $M_i[V(C_i)]$ is chordal ($M_i[V(C_i)]$ contains the minimal chordal supergraph of W_i computed by LEX-M and the vertices in $(V(C_i) - Incident(i))$ which are incident to every other vertex in $V(C_i)$.) Moreover, every vertex $x \in S$ must have $\alpha(x) \geq i$ since M_i contains all fill edges $F_1, ..., F_{i-1}$, and the cycle induced by S contains no chords. We can therefore find a path $a, b_1, ..., b_k, c$ in $M_i[S]$ with $k \geq 1$, a and c in $V(C_i)$, $ac \in (Candidate(i) - KeepFill(i))$, and $b_j, 1 \leq j \leq k$, not in $V(C_i)$. But then we have a $(k+3)$-cycle $a, b_1, ..., b_k, c, v_i, a$ in M_{i+1}, which ac is the only chord of since none of b_j is in $V(C_i)$. If $k = 1$, then this contradicts that $ac \in Candidate(i)$. If $k \geq 2$, then $a, b_1, b_2, ..., b_k, c, a$ induce a chordless $(k+2)$-cycle in M_{i+1}, contradicting that M_{i+1} is chordal.

The loop invariant therefore holds as claimed. Note that this establishes correctness of the algorithm as upon termination the returned graph M_1 is both a minimal chordal supergraph of $G_0 = G$ and a subgraph of $G_n \equiv (G; \alpha)$. ■

Theorem 6.4 *Let $\alpha = v_1, v_2, ..., v_n$ be an elimination ordering of a connected graph G, with $n = |V(G)|$, $e = |E(G)|$ and $f = \sum_{i=1}^{n} |F_i|$. The time complexity of Algorithm $MinimalChordal(G, \alpha)$ is $O(f(e+f))$.*

Proof: Computing G_n (see [98]) and the call of LEX-P (see [92]) both take time linear in the size of the filled graph. From G_n, the sets F_i and C_i can be computed in time

$O(nf)$ by simply examining every fill edge for every vertex. The algorithm has n iterations. At each iteration i, *CandidateEdge* is called $|F_i|$ times, and LEX-M is called once. The time complexity of $CandidateEdge(uv, i, M)$ is $O(|N_G(u)|) = O(n)$. The time complexity of LEX-M(W_i) is $O(|V(W_i)||E(W_i)|)$, (see [92]). Thus for the whole algorithm, we get:

$$O\Big(\sum_{i=1}^{n}(|V(W_i)||E(W_i)| + |F_i|n)\Big)$$

By the definition of W_i, $|V(W_i)| = |Incident(i)| \leq 2|F_i|$. Clearly, $|E(W_i)| \leq (e+f)$. Thus, the time complexity of LEX-M(W_i) becomes $O(|F_i|(e+f))$. Since all the sets F_i are disjoint, $\sum_{i=1}^{n} |F_i|n = O(fn)$, and $\sum_{i=1}^{n} |F_i|(e+f) = O(f(e+f))$. Since G is connected, $n \leq (e+f)$, and the overall time complexity is $O(f(e+f))$ ■

We would like to emphasize that the time complexity $O(f(e+f))$ seems to be an overestimate in many cases, and we suspect that the sum in the proof of Theorem 6.4 might often be less than this bound. A brief discussion about this is included in the concluding section.

6.5 Minimal fill and equivalent reorderings

In Chapter 5, we referred to perfect elimination orderings of the filled graph as equivalent reorderings. We adopt this convention also in this section. Let a graph G be ordered by an elimination ordering α. An equivalent reordering β, applied on G, results in a filled graph $(G; \beta)$ that is a subgraph of the original filled graph $(G; \alpha)$, as shown in Lemma 5.1. Our algorithm *MinimalChordal* returns a minimal filled graph of G that is also a subgraph of $(G; \alpha)$. Thus both equivalent reorderings, and the minimal fill orderings that are computed by Algorithm *MinimalChordal*, result in filled graphs that are both supergraphs of G and subgraphs of $(G; \alpha)$. This observation suggests that there is a relation between equivalent reorderings, and minimal fill orderings resulting in a subgraph of $(G; \alpha)$. One might, for example, think that the latter is a subset of the first. However, it turns out that these two classes of orderings are incomparable. Furthermore, as we will see by the following lemma, the resulting respective classes of filled graphs are neither comparable.

Lemma 6.3 *Let G be a graph and α an elimination ordering on G. There exist equivalent reorderings β such that the filled graph $(G; \beta)$ is not a minimal filled graph, and there exist minimal fill orderings γ such that the minimal filled graph $(G; \gamma)$ is a subgraph of $(G; \alpha)$, but no equivalent reordering results in a filled graph isomorphic to $(G; \gamma)$.*

Proof: We will prove this by giving a graph G with an original ordering α, and showing the existence of orderings β, and γ satisfying the lemma. Consider the graph shown in Figure 6.6 a). Let $\alpha = b, d, c, a, e$. The filled graph $(G; \alpha)$ is shown in Figure

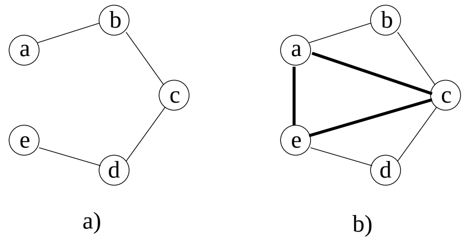


Figure 6.6: The graph in the proof of Lemma 6.3.

6.6 b). All equivalent reorderings, that is, perfect elimination orderings of $(G;\alpha)$ must either start with vertex b or d, since b and d are the only simplicial vertices of $(G;\alpha)$. Thus, any equivalent reordering β is either $b, \ldots$ or $d, \ldots$. The minimal filled graph of G that is a subgraph of $(G;\alpha)$ is, in this example, unique, and can be found by removing the fill edges in the following order: $\{ae\}, \{ce\}, \{ac\} = F_3, F_2, F_1$. This minimal filled graph is isomorphic to the original graph G. Therefore, the corresponding minimal fill ordering must start with one of the vertices a or e since these are the only simplicial vertices of G. Thus, the desired minimal fill ordering γ is either $a, \ldots$ or $e, \ldots$. We see that γ cannot agree with β. Furthermore, a filled graph resulting from any equivalent reordering β must have at least one fill edge from $\{ac, ce\}$, whereas a minimal filled graph of G that is a subset of $(G;\alpha)$ has no fill edges. ∎

Although the mentioned classes of orderings are not comparable, the following connection between them is clear.

Lemma 6.4 *Given G and α, any filled graph $(G;\beta)$ resulting from an equivalent reordering β contains a minimal filled graph $(G;\gamma)$ as a subgraph, where $(G;\gamma)$ is a subgraph of $(G;\alpha)$.*

Proof: Each $(G;\beta)$ is a subgraph of $(G;\alpha)$, and $(G;\beta)$ is either minimal, or edges can be removed from it to make it minimal. ∎

Corollary 6.2 *The smallest (in terms of the number of edges) minimal filled graph of G that is a subgraph of $(G;\alpha)$, is no larger than the smallest filled graph resulting from an equivalent reordering.*

In Section 6.1, we suggested that a minimal low fill ordering can be found by first using a fill-reducing heuristic, like minimum degree, and then running algorithm *MinimalChordal* on the resulting filled graph. In Section 5.3 of the previous chapter, we mentioned that if the given graph is ordered by a fill-reducing heuristic, then equivalent reorderings will not usually give considerable reduction in the amount of fill. However, Lemma 6.4 and Corollary 6.2 indicate that the gain, with respect to fill reduction, from minimalizing a fill-reducing ordering is larger than the gain by equivalent reorderings.

6.6 Conclusion

As mentioned in Section 6.1, minimal low fill orderings are desirable, but minimal fill is not necessarily low fill. Practical algorithms like minimum degree and nested dissection usually produce low fill orderings, but not minimal fill orderings. Therefore we assumed a filled graph (preferably with low fill) already given and considered the problem of removing fill edges to produce a minimal filled graph (with lower fill).

We have given a strategy to remove candidate fill edges so that each fill edge is checked for candidacy exactly once, avoiding backtracking. The time complexity of our algorithm is dependent on the number of fill edges we start with. In some cases this number is already low, in other cases it is high and our algorithm may remove many redundant fill edges. The upper bound given on the time complexity is met only if the subgraphs $W_1, W_2, ..., W_n$ overlap heavily so that the calls of LEX-M involve some large part of the graph several times. However, from examples we have studied, it seems that these subgraphs usually do not overlap very much. In the example of Figure 6.5, we see that the subgraphs W_i do not overlap on edges at all.

From the results given in this paper the following questions arise:

Does Theorem 6.2 have other interesting implications?

Can the time complexity of our algorithm be expressed as a function of the number of fill edges removed only?

Can the time complexity of our algorithm be improved if we assume that the given elimination ordering α was produced by the minimum degree algorithm? What if α was produced by the nested dissection algorithm? Assuming a particular algorithm might make it possible to show some properties of the resulting filled graph that help to prove a lower time complexity bound.

Does our algorithm have an implementation which is fast enough for practical purposes? Can the same idea of removing edges in the reverse order be used to design a more efficient algorithm without the use of LEX-M? Note that any such algorithm must compete with LEX-M. In order to see this, consider any graph G, and add a new vertex to G which is adjacent to every other vertex of G. An elimination ordering that eliminates this new vertex first, will cause a filled graph that is complete. In our implementation, minimalizing this filled graph is equivalent to running LEX-M on the original graph G or a subgraph of G. This is because all the fill edges are introduced at the same step, and every fill edge is, in this case, a candidate edge since every 4-cycle in a complete graph has two chords.

In our algorithm, we first find a minimal filled graph, and then we find a perfect elimination ordering of this graph to give a minimal fill ordering of the original graph. An interesting theoretical question is whether a minimal fill ordering can be computed simultaneously as the minimal filled graph is computed. For our algorithm, this is not of practical importance, since perfect elimination orderings are found in linear time by LEX-P. For other algorithms using the same order of removal, or more efficient implementations of our algorithm, it might be interesting. We have not been able to find a method for doing this. When candidate edges from $F_n, F_{n-1}, ..., F_i$ are removed, the remaining graph M_i has a perfect elimination ordering which is equal to the original ordering on G for $v_1, v_2, ..., v_{i-1}$, as shown by Lemma 6.2. However, the set of higher numbered vertices, which increases in cardinality after each step, has to be renumbered after each removal step of the algorithm. Thus perfect elimination orderings for smaller subgraphs must be found in each step. If these subgraphs overlap, which we suspect they may, then the practical gain from such a result will be small.

Finally, it would be interesting to know what percentage of the fill edges can be removed by minimalizing the fill in practice. The previous section ended with a discussion about this. From the examples that we studied, we believe that considerable reductions in the size of the fill are possible. Practical tests on a large number of graphs are necessary to confirm this. If the given ordering is an arbitrary ordering, and not necessarily a fill-reducing ordering, then considerable reduction in the fill size is highly probable.

Chapter 7

Conclusion

The aim of this thesis has been to study NP-hard graph partitioning and ordering problems related to sparse matrix factorization, and to suggest heuristic methods, or find classes of graphs for which the considered problem can be solved polynomially.

In Chapter 2, we studied the problem of partitioning graphs into generalized dominating sets, and showed cutoff values for each considered problem, giving a systematic complexity analysis of (k, σ, ρ)-partition problems for various values of the sets σ and ρ. Among the intentions of that chapter, two were to give a general study of NP-completeness of graph problems, and to provide examples of NP-completeness proofs.

Graph problems directly related to sparse matrix factorization were studied in Chapters 3, 4, 5, and 6. We considered graph problems related to minimizing one of the three very important measures: Elimination tree height, number of operations, and the amount of fill.

In Chapter 3, we gave a polynomial-time algorithm for finding minimum height elimination trees for interval graphs. Our approach, to handle the general NP-hardness of the minimum elimination tree height problem, has been first to find a rather restricted graph class for which the problem can be solved polynomially. After showing several results on this class, it was almost straightforward to expand it to the class of interval graphs. A close study of how the clique tree data structure should be updated in order to achieve cubic time complexity was also included in this chapter.

Since minimum height elimination tree orderings can cause a large amount of extra fill compared to fill-reducing orderings, in Chapter 4, we studied the problem of reducing the fill for a given elimination tree. In particular, for nested dissection orderings with minimal separators, we showed that minimizing fill while preserving the corresponding separator tree is an NP-hard problem.

The elimination tree is an important structure both for parallel Cholesky factorization, and for multifrontal QR factorization. In Chapter 5, we considered sparse

QR factorization, and methods to reduce the number of operations required by multifrontal and traditional Householder QR factorization algorithms. Our approach has been to design new heuristics, and to suggest tie-breaking strategies for the already famous minimum degree heuristic. We pointed out the importance of considering the original matrix A, rather than only sparsity analysis of $A^T A$, which has been the usual approach. Numerical results concerning operation count, fill, and intermediate fill were presented as results of our algorithms.

Reducing the amount of fill has been the topic of Chapter 6. The NP-hard problem of minimizing the fill was handled by finding *minimal* fill rather than *minimum*. Minimal fill problem has been previously solved in polynomial time. However, our approach has been different than the previous approaches. An arbitrary minimal fill ordering of a given graph might cause more fill than fill introduced by, for example, minimum degree or nested dissection. Therefore, we posed a “sandwich” problem, where we found a minimal filled graph which was a subgraph of an already given filled graph. This way, non-minimal filled graphs with low fill can be supplied to our algorithm, which then reduces the fill to minimal.

7.1 Selected open problems

Several open problems are mentioned in the previous chapters. These open problems can be grouped as follows:

1. Deciding the computational complexity of a considered problem, by proving its NP-completeness or NP-hardness, or by finding a polynomial algorithm that solves it.

2. Showing whether or not the shown upper bound for time complexity of a presented algorithm can be improved, by finding a lower bound or a new upper bound.

3. Designing new algorithms.

For problems of Type 1, the most interesting is perhaps the complexity of finding minimum height elimination trees for chordal graphs, mentioned in Chapter 3. Results concerning the time complexity bounds of the algorithms presented in Chapters 3 and 6 would would be solutions to problems of Type 2. Regarding Type 3, a very interesting approach would be a new column ordering heuristic that operates directly on A as mentioned in Chapter 5.

7.2 Future research directions

In each previous chapter, we gave a brief discussion on possible future research plans related to the problems discussed in the chapter. It is our belief that interesting further research can be carried out in all of the studied topics. In this section, we point out two main directions of future research, and briefly compare them regarding their potential for interesting and important new results.

The two basic sparse methods studied in this thesis are the sparse Cholesky factorization, and the sparse QR factorization. For both these methods, graph theory is useful to develop good algorithms for efficient factorization. Graph theory helps to define and visualize structures and algorithms simply, and also helps researchers with non-numerical background to participate in the solution of these problems.

Sparse Cholesky factorization, and related graph theory issues have been subject to extensive research over the past decades. The minimum fill, and the minimum elimination height problems have important parallels also in other research fields within graph theory. One such connection, the vertex ranking problem, was mentioned in Chapter 3. This makes the field even more interesting to research on. However, evidently, the potential for finding new results of practical and general interest decreases as the "age" and the number of researchers of the field increases.

Sparse QR factorization has become attractive mostly after the introduction of multifrontal methods, and thus is a newer research field. Due to the relation to Cholesky factorization through normal equations, QR factorization has "inherited" sparsity analysis methods for Cholesky factorization. However, the important difference between these factorization algorithms, intermediate fill, advocates designing sparsity analysis methods especially suitable for QR factorization. We believe that there is a great potential for achieving important new results in this area.

Appendix A

Abstracts of Research Papers

Partitioning Graphs into Generalized Dominating Sets

Pınar Heggernes Jan Arne Telle

Department of Informatics
University of Bergen, Norway
pinar@ii.uib.no, telle@ii.uib.no

Abstract

We study the computational complexity of partitioning the vertices of a graph into generalized dominating sets. Generalized dominating sets are parameterized by two sets of nonnegative integers σ and ρ which constrain the neighborhood $N(v)$ of vertices. A set S of vertices of a graph is said to be a (σ, ρ)-set if $\forall v \in S : |N(v) \cap S| \in \sigma$ and $\forall v \notin S : |N(v) \cap S| \in \rho$. The (k, σ, ρ)-partition problem asks for the existence of a partition $V_1, V_2, ..., V_k$ of vertices of a given graph G such that V_i is a (σ, ρ)-set of G, for $i = 1, 2, ..., k$. We study the computational complexity of this problem as the parameters σ, ρ, and k vary.

Finding Minimum Height Elimination Trees for Interval Graphs in Polynomial Time

Bengt Aspvall Pınar Heggernes

Department of Informatics
University of Bergen, Norway
bengt@ii.uib.no, pinar@ii.uib.no

Abstract

The elimination tree plays an important role in many aspects of sparse matrix factorization. The height of the elimination tree presents a rough, but usually effective, measure of the time needed to perform parallel elimination. Finding orderings that produce low elimination trees is therefore important. As the problem of finding minimum height elimination tree orderings is NP-hard, it is interesting to concentrate on limited classes of graphs and find minimum height elimination trees for these efficiently. In this paper, we use clique trees to find an efficient algorithm for interval graphs which make an important subclass of chordal graphs. We first illustrate this method through an algorithm that finds minimum height elimination trees for chordal graphs. This algorithm, although of exponential time complexity, is conceptionally simple and leads to a polynomial-time algorithm for finding minimum height elimination trees for interval graphs.

A Simple Cubic Algorithm for Computing Minimum Height Elimination Trees for Interval Graphs

Bengt Aspvall Pınar Heggernes Jan Arne Telle

Department of Informatics
University of Bergen, Norway
bengt@ii.uib.no, pinar@ii.uib.no, telle@ii.uib.no

Abstract

In parallel sparse Cholesky factorization, finding elimination orderings that produce elimination trees of low height is important. The problem of finding the minimum elimination tree height for a graph is NP-hard for general graphs. This problem is equivalent to the problem of finding the vertex ranking number of a graph. The problem can be solved in polynomial time for special classes of graphs. Aspvall and Heggernes presented a polynomial time algorithm for solving the problem for interval graphs, in 1993. This algorithm is $O(n^4)$. Later, when Deogun, Kloks, Kratsch and Müller solved the vertex ranking problem for permutation graphs in time $O(n^6)$, they mentioned that the problem for interval graphs could be solved in time $O(n^3)$. In this paper we present a simple $O(n^3)$ algorithm for finding minimum elimination tree height for interval graphs. In addition, our algorithm gathers enough information so that an actual elimination tree of minimum height can be found within the same time bound.

Fill Minimization for Nested Dissection with Minimal Separators

Pınar Heggernes

Department of Informatics
University of Bergen, Norway
pinar@ii.uib.no

Abstract

In sparse Cholesky factorization, finding elimination orderings that produce small fill is very important and various heuristics have been proposed. For parallel computations, good orderings should produce elimination trees of low height. Finding optimal fill orderings and finding optimal height orderings are NP-hard problems. A class of nested dissection orderings with minimal separators, has been shown to contain optimal height orderings. We show that for such orderings, the problem of computing minimum fill decomposes into independent subproblems for each separator. These subproblems are shown to be NP-hard. However, for small separators, they can be solved optimally in reasonable time.

Finding Good Column Orderings for Sparse QR Factorization

Pınar Heggernes
Department of Informatics
University of Bergen, Norway
pinar@ii.uib.no

Pontus Matstoms
Department of Mathematics
Linköping University, Sweden
pomat@mai.liu.se

Abstract

For sparse QR factorization, finding a good column ordering of the matrix to be factorized, is essential. Both the amount of fill in the resulting factors, and the number of floating-point operations required by the factorization, are highly dependent on this ordering. A suitable column ordering of the matrix A is usually obtained by minimum degree analysis on A^TA. The objective of this analysis is to produce low fill in the resulting triangular factor R. We observe that the efficiency of sparse QR factorization is also dependent on other criteria, like the size and the structure of intermediate fill, and the size and the structure of the frontal matrices for the multifrontal method, in addition to the amount of fill in R. An important part of this information is lost when A^TA is formed. However, the structural information from A is important to consider in order to find good column orderings. We show how a suitable equivalent reordering of an initial fill-reducing ordering can decrease the number of operations required by multifrontal QR factorization, without increasing the number of nonzeros in R. Heuristics that produce good equivalent reorderings are proposed and explained. We also propose tie-breaking strategies for the minimum degree algorithm, which produce orderings that require considerably less number of operations in both multifrontal and traditional Householder QR factorization, than the standard minimum degree algorithm without tie-breaking.

Making an Arbitrary Filled Graph Minimal by Removing Fill Edges

Jean R. S. Blair   Pınar Heggernes   Jan Arne Telle

US Military Academy
West Point, NY, USA
dj4693@dean.usma.army.mil

Department of Informatics
University of Bergen, Norway
pinar@ii.uib.no, telle@ii.uib.no

Abstract

We consider the problem of removing fill edges from a filled graph G' to get a minimal chordal supergraph M of the original graph G; thus $G \subseteq M \subseteq G'$. We show that a greedy strategy can be applied if fill edges are processed for removal in the reverse order of their introduction. For a filled graph with f fill edges and e original edges, we give a simple $O(f(e+f))$ algorithm which solves the problem and computes a corresponding minimal elimination ordering. We believe that in practice the runtime of our algorithm is usually better than the worst-case bound of $O(f(e+f))$.

Bibliography

[1] A. Agrawal, P. Klein, and R. Ravi, *Cutting down on fill using nested dissection: provably good elimination orderings*, in Sparse Matrix Computations: Graph Theory Issues and Algorithms, J. A. George, J. R. Gilbert, and J. W. H. Liu, eds., Springer Verlag, 1993, pp. 31–55. IMA Volumes in Mathematics and its Applications, Vol. 56.

[2] S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, 1989.

[3] B. Aspvall, *Minimizing elimination tree height can increase fill more than linearly*, Information Processing Letters, 56 (1995), pp. 115–120.

[4] B. Aspvall and P. Heggernes, *Finding minimum height elimination trees for interval graphs in polynomial time*, BIT, 34 (1994), pp. 484–509.

[5] B. Aspvall, P. Heggernes, and J. A. Telle, *A simple cubic algorithm for computing minimum height elimination trees for interval graphs*, Tech. Rep. 103, University of Bergen, Norway, 1995. Reports in Informatics.

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

[7] P. A. Bernstein and N. Goodman, *Power of natural semijoins*, SIAM J. Comput., 10 (1981), pp. 751–771.

[8] J. R. S. Blair, P. Heggernes, and J. A. Telle, *Making an arbitrary filled graph minimal by removing fill edges*, in Algorithm Theory - SWAT '96, R. Karlsson and A. Lingas, eds., Springer Verlag, 1996, pp. 173–184. Lecture Notes in Computer Science 1097.

[9] 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.

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

[11] H. L. Bodlaender, J. S. Deogun, K. Jansen, T. Kloks, D. Kratsch, H. Müller, and Z. Tuza, *Rankings of graphs*, Proceedings of the 20^{th} Workshop on Graph Theoretic Concepts in Computer Science, Springer Verlag, (1995), pp. 106–120. Lecture Notes in Computer Science 903.

[12] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks, *Approximating treewidth, pathwidth, frontsize and shortest elimination tree*, J. Algorithms, 18 (1995), pp. 238–255.

[13] K. S. Booth and G. S. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[14] D. P. Bovet and P. Crescenzi, *Introduction to the Theory of Complexity*, Prentice Hall, 1994.

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

[16] A. Cobham, *The intrinsic computational difficulty of functions*, Proceedings of the 1964 Congress on Logic, Mathematics, and the Methodology of Science, (1964), pp. 24–30.

[17] E. J. Cockayne, B. L. Hartnell, S. T. Hedetniemi, and R. Laskar, *Perfect domination in graphs*, J. Combin. Inform. System Sci., 18 (1993), pp. 136–148.

[18] T. F. Coleman, A. Edenbrandt, and J. R. Gilbert, *Predicting fill for sparse orthogonal factorization*, J. Assoc. Comput. Mach., 33 (1986), pp. 517–532.

[19] S. A. Cook, *The complexity of theorem-proving procedures*, in Proceedings of Third Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, 1971, pp. 151–158.

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

[21] E. Dahlhaus and M. Karpinski, *An efficient parallel algorithm for the minimal elimination ordering of an arbitrary graph*, Proceedings FOCS, (1989), pp. 454–459.

[22] J. S. Deogun, T. Kloks, D. Kratsch, and H. Müller, *On vertex ranking for permutation and other graphs*, Proceedings of the 11^{th} Annual Symposium on Theoretical Aspects of Computer Science, 775 (1994), pp. 747 – 758.

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

[24] I. S. Duff, *On algorithms for obtaining a maximum transversal*, ACM Trans. Math. Software, 7 (1981), pp. 315–330.

[25] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, London, 1986.

[26] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Softw., 15 (1989), pp. 1–14.

[27] I. S. Duff and J. K. Reid, *A comparison of some methods for the solution of sparse overdetermined systems of linear equations*, J. Inst. Maths. Applic., 17 (1976), pp. 267–280.

[28] ——, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.

[29] A. L. Dulmage and N. S. Mendelsohn, *Coverings of bipartite graphs*, Canad. J. Math., 10 (1958), pp. 517–534.

[30] ——, *A structure theory of bipartite graphs of finite exterior dimension*, Trans. Roy. Soc. Canada, 53, III (1959), pp. 1–13.

[31] ——, *Two algorithms for bipartite graphs*, J. Soc. Indust. Appl. Math., 11 (1963), pp. 183–194.

[32] J. Edmonds, *Optimum branchings*, J. Res. National Bureau of Standards, Part B, 17B (1966-1967), pp. 233–240.

[33] ——, *Systems of distinct representatives and linear algebra*, J. Res. National Bureau of Standards, Part B, 17B (1966-1967), pp. 241–245.

[34] R. E. England, J. R. S. Blair, and M. G. Thomason, *Independent computations in a probablistic knowledge-based system*, Tech. Rep. CS-90-128, Department of Computer Science, The University of Tennessee, Knoxville, Tennessee, 1991.

[35] M. J. Flynn, *Very high speed computing systems*, Proceedings of the IEEE, 54 (1966), pp. 1901–1909.

[36] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.

[37] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability*, W. H. Freeman and Co., 1978.

[38] F. GAVRIL, *The intersection graphs of subtrees in trees are exactly the chordal graphs*, J. Combin. Theory Ser. B, 16 (1974), pp. 47–56.

[39] J. A. GEORGE, *Computer implementation of the finite element method*, Tech. Rep. STAN-CS-208, Stanford University, USA, 1971.

[40] J. A. GEORGE AND M. T. HEATH, *Solution of sparse linear least squares problems using Givens rotations*, Linear Algebra Appl., 34 (1980), pp. 69–83.

[41] J. A. GEORGE AND J. W. H. LIU, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[42] ——, *Householder reflections versus Givens rotations in sparse orthogonal decomposition*, Linear Algebra Appl., 88/89 (1987), pp. 223–238.

[43] ——, *The evolution of the minimum degree ordering algorithm*, SIAM Review, 31 (1989), pp. 1–19.

[44] J. A. GEORGE, J. W. H. LIU, AND E. G. NG, *Row ordering schemes for sparse Givens transformations, I. Bipartite graph model*, Linear Algebra Appl., 61 (1984), pp. 55–81.

[45] J. A. GEORGE AND E. G. NG, *Symbolic factorization for sparse Gaussian elimination with partial pivoting*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 877–898.

[46] J. R. GILBERT, *Private communication.*

[47] ——, *Some nested dissection order is nearly optimal*, Information Processing Letters, 26 (1987/88), pp. 325–328.

[48] J. R. GILBERT, C. MOLER, AND R. SCHREIBER, *Sparse matrices in MATLAB: Design and implementation*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 333–356.

[49] J. R. GILBERT AND E. G. NG, *Predicting structure in nonsymmetric sparse matrix factorization*, in Graph Theory and Sparse Matrix Computation, J. A. George, J. R. Gilbert, and J. W. H. Liu, eds., Springer Verlag, 1993.

[50] J. R. GILBERT AND R. E. TARJAN, *The analysis of a nested dissection algorithm*, Numer. Math., 50 (1987), pp. 377–404.

[51] P. C. Gilmore and A. J. Hoffman, *A characterization of comparability graphs and of interval graphs*, Canadian Journal of Mathematics, 16 (1964), pp. 539–548.

[52] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, 1989. Second edition.

[53] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[54] P. Hall, *On representatives of subsets*, J. Lond. Math. Soc., 10 (1935), pp. 26–30.

[55] S. T. Hedetniemi and R. Laskar, *Bibliography on domination in graphs and some basic definitions of domination parameters*, Discrete Mathematics, 86 (1990).

[56] P. Heggernes, *Minimizing fill-in size and elimination tree height in parallel Cholesky factorization*, Master's thesis, University of Bergen, Norway, 1992.

[57] P. Heggernes and P. Matstoms, *Finding good column orderings for sparse QR factorization*, Tech. Rep. LiTH-MAT-R-1996-20, Linköping University, Sweden, 1996. To be presented at the Second SIAM Conference on Sparse Matrices, Idaho, October 1996.

[58] P. Heggernes and J. A. Telle, *Partitioning graphs into generalized dominating sets*, in Proceedings of XV International Conference of the Chilean Computer Society, University of Chile, Santiago, 1995, pp. 241–252.

[59] 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.

[60] I. Holyer, *The NP-completeness of edge-coloring*, SIAM J. Comput., 10 (1981), pp. 718–720.

[61] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, Tech. Rep. CS-95-035, University of Minnesota, Minneapolis, 1995.

[62] J. Kratochvíl, *Perfect codes in general graphs*, tech. rep., Academia Praha, 1991.

[63] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Cummings, 1994.

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

[65] D. Leven and Z. Galil, *NP-completeness of finding the chromatic index of regular graphs*, J. Algorithms, 4 (1983), pp. 35 – 44.

[66] 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. 1156–1173.

[67] J. G. Lewis, D. J. Pierce, and D. K. Wah, *Multifrontal Householder QR factorization*, Tech. Rep. ECA-TR-127-Revised, Boeing Computer Services, Seattle, WA, 1989.

[68] J. W. H. Liu, *A compact row storage scheme for Cholesky factors using elimination trees*, ACM Trans. Math. Software, 12 (1986), pp. 127–148.

[69] ——, *On general row merging schemes for sparse Givens transformations*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 1190–1211.

[70] ——, *Equivalent sparse matrix reorderings by elimination tree rotations*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 424–444.

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

[72] J. W. H. Liu and A. H. Sherman, *Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices*, SIAM J. Numer. Anal., 13 (1975), pp. 198–213.

[73] L. Lovász and M. Plummer, *Matching Theory*, North-Holland, 1986.

[74] F. Manne, *Load Balancing in Parallel Sparse Matrix Computations*, PhD thesis, University of Bergen, Norway, 1993.

[75] F. Manne and H. Hafsteinsson, *Efficient sparse cholesky factorization on a parallel SIMD computer*, SIAM J. Sci. Comput., 16 (1995), pp. 934–950.

[76] P. Manneback, *On some numerical methods for solving large sparse linear least squares problems*, PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium, 1985.

[77] H. M. Markowitz, *The elimination form of the inverse and its application to linear programming*, Management Science, 3 (1957), pp. 255–269.

[78] P. Matstoms, *Sparse QR factorization in MATLAB*, ACM Trans. Math. Software, 20 (1994), pp. 136–159.

[79] ——, *Sparse QR Factorization with Applications to Linear Least Squares Problems*, PhD thesis, Linköping University, Sweden, 1994.

[80] T. Ohtsuki, L. K. Cheung, and T. Fujisawa, *Minimal triangulation of a graph and optimal pivoting ordering in a sparse matrix*, J. Math. Anal. Appl., 54 (1976), pp. 622–633.

[81] C. H. Papadimitriou, *The NP-completeness of the bandwidth minimization problem*, Computing, 16 (1976), pp. 263–270.

[82] ——, *Computational Complexity*, Addison-Wesley, 1994.

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

[84] N. Pippenger, *On simultaneous resource bounds*, Proceedings of 20th IEEE Symposium on the Foundations of Computer Science, (1979), pp. 307–311.

[85] A. Pothen, *The complexity of optimal elimination trees*, Tech. Rep. CS-88-13, Pennsylvania State University, 1988.

[86] ——, *Graph partitioning algorithms with applications to scientific computing*, in Parallel Numerical Algorithms, D. E. Keyes, A. H. Sameh, and V. Venkatakrishnan, eds., Kluwer Academic Press, 1995.

[87] C. Puglisi, *QR Factorization of Large Sparse Overdetermined and Square Matrices with the Multifrontal Method in a Multiprocessing Environment*, PhD thesis, Institut National Polytechnique de Toulouse, France, 1993.

[88] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987.

[89] M. O. Rabin, *Mathematical theory of automata*, Proceedings of 19^{th} Symp. of Applied Math., (1966), pp. 153–175.

[90] F. S. Roberts, *Indifference graphs*, in Proof Techniques in Graph Theory, F. Harary, ed., Academic Press, 1969, pp. 139 – 146.

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

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

[93] T. J. Schaefer, *The complexity of satisfiability problems*, ACM Symposium on Theory of Computing, 10 (1978), pp. 216–226.

[94] A. A. Schäffer, *Optimal node ranking of trees in linear time*, Information Processing Letters, 33 (1989/90), pp. 91–96.

[95] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Trans. Math. Software, 8 (1982), pp. 256–276.

[96] R. E. Tarjan, *Graph theory and Gaussian elimination*, in Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, 1976, pp. 3 – 22.

[97] ——, *Data Structures and Network Algorithms*, SIAM, 1983.

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

[99] J. A. Telle, *Complexity of domination-type problems in graphs*, Nordic J. Comput., 1 (1994), pp. 157–171.

[100] ——, *Vertex Partitioning Problems: Characterization, Complexity and Algorithms on Partial k-Trees*, PhD thesis, University of Oregon, 1994.

[101] W. F. Tinney and J. W. Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proceedings of the IEEE, 55 (1967), pp. 1801–1809.

[102] J. von Neumann, *A certain zero-sum two-person game equivalent to the assignment problem*, in Contributions to the Theory of Games 2, H. W. Kuhn and A. W. Tucker, eds., Princeton University Press, 1953, pp. 5–12.

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