# Parallelizing and De-parallelizing Elimination Orders

**Claudson Ferreira Bornstein**

August 13th, 1998

CMU-CS-98-159

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Bruce M. Maggs
Gary L. Miller
R. Ravi
John Gilbert, Xerox Parc

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy*



This research was supported by the National Science Foundation award No. CCR-9505472, by the National Science Foundation Young Investigator Award No. CCR-94-57766 and by the generous contributions of the NEC Research Institute and Sun Microsystems.

**Keywords:** Parallel elimination orders, fill and work minimization, nested dissection, min-degree, hybrid algorithms, chordal completion, LU decomposition, Gaussian elimination, chordal graphs, interval graphs, graph theory

# Abstract

The order in which the variables of a linear system are processed determines the total amounts of fill and work to perform LU decomposition on the system. We identify a trade-off between the amounts of fill and work for a given order and the parallelism inherent in that order. We present two algorithms: one that tries to parallelize sequential orders, and another that tries to produce low-fill orders, while, in the process, producing somewhat sequential orders.

The first algorithm takes a sequential order for a matrix and produces a parallel one with at most a constant factor more nonzeros and work. We also show that, for certain graphs, any parallel order requires an amount of additional fill that is a function of the amount of parallelism exhibited. The more parallel the order, the more fill it introduces.

We identified a particular "deficiency" of nested dissection that arises from the parallel nature of the orders it produces. Thus, when shifting our goal towards fill and work minimization, we choose to modify nested dissection to obtain a similar algorithm that produces orders that introduce less fill and work than a traditional nested dissection order would, but that are also less parallel than the orders that would be produced by the traditional nested dissection algorithm.

Our experimental work comparing this variant of nested dissection and a number of other publicly available ordering algorithms indicates that while a few of the algorithms produce comparable-quality orders, the minimum-degree algorithm stands out as the worst one. Contrary to common belief, the minimum-degree algorithm produces poor quality orders in terms of fill and work. Our variant of nested dissection compares favorably with state-of-the-art ordering algorithms, including implementations of nested dissection, minimum-degree and their hybrids.

*To my parents, Gilda and Claudio*
*my sisters, Claudia and Solange*
*and my little niece, Lara,*
*whom I love very much,*
*and can finally go home to.*

# Acknowledgements

I would like to thank all those who in any way have contributed to this thesis:

- My parents and sisters, whose love and support have always been present. And Lara, my little niece – she is so cute – I had to graduate to go home to my family and see her grow.
- My friends, who helped me endure the years of graduate school: Po-Jen, with whom I had many helpful discussions, and Henry, Tammy and Clara. My officemate Arup, who taught me a lot. Po, Arup and his wife Nita became my best friends through graduate school.
- Gary, Bruce and Ravi, with their intuition, insightful comments, and friendship. I had the pleasure of working with them for the last few years.
- Jayme and Edil, for their friendship and encouragement since my undergraduate days.
- Bruce Hendrickson and Ed Rothberg, who were very helpful in answering all my questions about Chaco and BEND, making it easier for me to include a more realistic comparison of various state-of-the-art algorithms in this thesis.
- Sharon Burks, always there, always smiling when I came to her with my problems, and Catherine Copetas, always with a smile to brighten those dark winter days.
- And all those who I might have forgotten to mention here.

# Contents

# Chapter 1

# Introduction

Among the most common problems to be solved in both operations research and scientific computing is that of finding a vector $x$ that satisfies a linear system of equations $Ax = b$, where $A$ is a known matrix and $b$ is a known vector. Systems of this sort arise in scientific computing when modeling physical systems, such as the flow of air over an airfoil, and in operations research when solving optimization problems, such as assigning production tasks to machines in a workshop. Often these systems are quite large, making their solution on even the most powerful computers challenging. Two common approaches to solving linear systems in practice are direct methods such as Gaussian elimination, and iterative methods such as conjugate gradient. This thesis focuses on Gaussian elimination.

A system of equations $Ax = b$ can be simplified if the matrix $A$ can first be decomposed into the product of two matrices $A = L \cdot U$, where $L$ is a lower triangular matrix, and $U$ is an upper triangular matrix. The solution, $x$, can then be found by first solving $Ly = b$ for $y$, and then solving $Ux = y$ for $x$. These two systems are easier to solve than the original system $Ax = b$ because when the known matrix is upper or lower triangular, a fast algorithm called back substitution can be applied. The factorization $A = L \cdot U$ can be found through Gaussian elimination. One advantage to solving linear systems through $L \cdot U$ decomposition in the scientific computing domain is that systems of the form $Ax = b$ are often solved repeatedly for a fixed matrix $A$ but with different matrices $b$. This scenario arises, for example, when using the popular finite-element method for solving a set of partial differential equations describing a physical system. In this case, the cost of factoring $A$ into $L \cdot U$ can be amortized against the cost of repeatedly solving the system.

The basic step in Gaussian elimination is to add or subtract a multiple of one row of the matrix $A$ to another row of $A$. In $L \cdot U$ decomposition, a multiple of the $i$th row of $A$ is subtracted from each of the following rows in order to eliminate the $i$th entry of each of these rows. In particular, for $j > i$, a multiple of $A_{ji}/A_{ii}$ of the $i$th row of $A$ is subtracted from the $j$th row so that the new value of $A_{ji}$ is $A_{ji} - (A_{ji}/A_{ii})A_{ii} = 0$. This operation is referred to as pivoting on $A_{ii}$, and the multiple $A_{ji}/A_{ii}$ is recorded as $L_{ji} = A_{ji}/A_{ii}$. Processing the $i$th row of $A$ corresponds to eliminating the $i$th variable $x_i$ from the system of equations. The rows of $A$ are processed in order, so that once the last row has been processed $A$ has been reduced to an upper triangular matrix $U$. At this point, $A = L \cdot U$.

When a multiple of the $i$th row of $A$ is subtracted from the $j$th row of $A$ in order to eliminate $A_{ji}$, the entries in row $i$ in all columns $k$ beyond the $i$th column are also subtracted from the corresponding entries in the $j$th row. When the entry $A_{ik}$ is non-zero, but the entry $A_{jk}$ is zero, we say that *fill* has occurred, because the new value of $A_{jk}$ is no longer non-zero. In general, fill is considered to be undesirable because it increases the amount of memory required to store the linear system and because there is a high correlation between the amount of fill and the total number of floating point operations (the *work*) required to complete the $L \cdot U$ factorization. Minimizing the amount of memory required is crucial because in practice the size of the largest system that can feasibly be solved is typically limited by the size of a computer's main memory. Unfortunately, the process of elimination may turn a sparse matrix $A$ into a dense matrix $U$.

The $L \cdot U$ decomposition algorithm, as described so far, allows no freedom in the choice of pivots, and hence provides no mechanism for avoiding fill. For the purposes of solving $Ax = b$, however, the order in which the rows of $A$ (and hence $b$) appear does not affect the solution to the system. Hence the rows of $A$ and $b$ can be permuted by a permutation matrix $P$, and then Gaussian elimination can be applied to the system $(PA)x = Pb$ rather than $Ax = b$. The permutation specifies the order in which the original rows of the matrix $A$ are to be processed. Different orders may create wildly different amounts of fill.

The matrix $A$ can be viewed as a graph. In particular, an $n \times n$ symmetric matrix $A$ can be viewed as an undirected graph consisting of $n$ vertices, $\{1, \ldots, n\}$, with an edge between vertices $i$ and $j$ if the matrix entry $A_{ij} = A_{ji}$ is non-zero. The operation of eliminating the $i$th variable $x_i$ then corresponds to the graph operation of removing vertex $i$ from the graph and adding edges between all of the neighbors of $i$ so that they form a clique [Par61, Ros70]. To see why such edges must be added, note that in pivoting on $A_{ii}$, a non-zero entry may be created at $A_{jk}$ for any $j$ such that $A_{ij} \geq 0$ (i.e., for any neighbor $j$ of $i$) and for any $k$ such that $A_{ik} \neq 0$ (i.e., for any neighbor $k$ of $i$). The newly added edges are called *fill edges* (or just *fill*). Minimizing either the fill or the work by performing symmetric permutations of symmetric matrices, that is, by re-ordering the rows and columns of the matrix, has been shown to be NP-hard [Yan81].

The Gaussian elimination algorithm discussed so far breaks down if it encounters a pivot $A_{ii}$ whose value is zero, since such a pivot cannot be used to eliminate non-zero elements in the following rows. To dodge this issue, we restrict our discussion to the class of *symmetric positive definite matrices*. If a matrix is symmetric positive definite, then during the course of Gaussian elimination, no matter how the variables are ordered, no pivot element $A_{ii}$ will have value zero. Symmetric positive definite matrices are not unusual in practice. Furthermore, there are a variety of approaches for dealing with a zero-valued pivot, including exchanging the row with another row, and replacing the pivot value with a small non-zero value (and later adjusting the final solution).

In this thesis we identify a trade-off between the amounts of fill and work for a given order and the parallelism inherent in that order. We present two algorithms: one that tries to parallelize sequential orders, and another that tries to produce low-fill orders, while, in the process, producing somewhat sequential orders.

## 1.1 Existing ordering heuristics

Over the years a number of algorithms for producing elimination orders with low fill and work have been proposed. The most popular of these are *minimum degree* [Mar57, TW67, Liu85] and *nested dissection* [Geo73, GL78, LRT79]. At each step, the minimum-degree heuristic selects a vertex with the smallest *degree* (number of neighbors), removes this vertex from the graph by pivoting on it (making its neighborhood a clique), and then looks for the next vertex with the smallest degree in the new graph. It proceeds in this fashion until all vertices have been selected. The orders produced are typically good. There exist, however, minimum-degree orders with significant amounts of fill. For example, minimum degree can produce orders with $\Omega(n^{\log_3 4})$ fill for the toroidal $n \times n$ mesh [BS90], while the optimal order for the same graph requires only $\Theta(n \cdot \log n)$ fill.

Nested dissection, on the other hand, examines the graph as a whole before ordering it. Unlike minimum degree, nested dissection orders the vertices of the graph backwards, that is, it begins by deciding which vertices should be eliminated last. Nested dissection works by selecting a *balanced separator*, i.e., a set of vertices that, when removed from the graph, partitions it into connected components each of which has at most a constant fraction of the total number of vertices in the graph. The vertices in the separator are placed last in the elimination order. Then nested dissection recursively orders each of the connected components until the whole graph has been ordered. Because the separator is eliminated last, and there are no edges connecting vertices in different connected components, no fill can be created between different components as they are eliminated. Hence, the various components can be eliminated either sequentially or in parallel without affecting the quality of the order in terms of both fill and work. For planar graphs and graphs with bounded genus [GT87, LRT79], and for graphs with bounded degree [AKR93], nested dissection has been shown to produce orders that have fill within a poly-logarithmic factor of the optimum.

Even though minimum-degree algorithms can produce elimination orders that introduce more fill than the worst-case nested dissection order, minimum degree is usually preferred to nested dissection, and is said, in practice, to produce orders with less fill. However, the "in practice" wisdom on this is changing. The currently accepted champion algorithms are hybrids of nested dissection and minimum degree [AL96, HR96] and benefit from the strengths of both methods but do not, however, provide any performance guarantees.

## 1.2 Parallel elimination

The performance of Gaussian elimination can be improved on parallel computers by generalizing the basic elimination step to allow for more than one vertex to be eliminated at a time. In order for such an operation to make sense, the vertices that are eliminated in one step must form an *independent set*, i.e., a set of vertices no two of which are adjacent. As an example, in the nested dissection algorithm a set of vertices consisting of one from each of the connected components constitutes an independent set, and hence can be eliminated in parallel. The *height* of a graph is the minimum number of parallel elimination steps needed to eliminate all of the vertices of the graph.

Finding the height of a graph is NP-hard [Pot88], but minimum-height orders may be found for specific classes of graphs. Aspvall and Heggernes [AH94] present an algorithm that finds elimination orders with minimum height for interval graphs, but the orders produced have not been analyzed in terms of fill. A family of chordal graphs for which any minimum-height order must produce fill that is more than a constant factor larger than the total number of nonzeros in a minimum-fill decomposition is presented in [Asp95]. Manne [Man91] shows how to produce minimum-height orders for trees with fill linear in the number of edges in the tree.

The height of an order is a measure of its parallelism. The more parallel an order, the smaller its height. Some graphs are inherently more sequential than others. For example, the vertices of a *clique* (a graph in which every pair of vertices is connected by an edge) must be eliminated sequentially, because a clique contains no independent sets of size greater than one. More generally, if a graph contains a clique, or if a clique is created during the course of elimination, then that clique must be eliminated sequentially. There are, however, alternative parallel algorithms for solving these dense linear systems of equations. This suggests modifying Gaussian elimination so that when a large dense subgraph is encountered, a different algorithm is used to eliminate it. In a single *stage* of such a hybrid algorithm, an independent set of cliques may be eliminated in parallel.

We begin our study by examining parallel orders for specific classes of graphs, namely interval graphs and chordal graphs. The purpose of the study was twofold. First, the rich structure of these graphs provides some insight into the problem of finding orders that minimize both fill and work. Second, because chordal graphs are precisely those graphs with zero-fill elimination orders, any graph $G$ along with the fill edges introduced by a given order is a chordal graph, called the *chordal completion* of $G$. This suggests that an algorithm designed to find a parallel order for a chordal graph can be applied to the chordal completion of a graph $G$ generated by some other (possibly sequential) ordering heuristic, and the resulting order, which may be more parallel, can then be applied to the original graph $G$.

Although zero-fill orders for chordal and interval graphs can be computed in linear time [RTL76], these orders do not necessarily have low height. Our goal was to obtain an algorithm that takes a chordal graph and produces an order with low height and with fill linear in the number of edges of the graph. Although zero fill is preferable, allowing linear fill increases the amount of space required for Gaussian elimination by only a constant factor.

We started by analyzing nested dissection on the classes of interval and chordal graphs. We showed that if the separators are required to partition the graph into components with no more than half the number of vertices in the graph, then nested dissection may introduce a super-linear amount of fill even on interval graphs. We also showed that by allowing a constant-factor imbalance in the size of the subgraphs generated by the choice of the separator, a specific nested dissection algorithm produces orders with linear fill and linear work for interval graphs.

The bounds we obtained for this nested dissection algorithm corroborate the common notion that allowing some imbalance in the size of the subgraphs produced by removing a separator can help produce better orders. With this imbalance, nested dissection is a suitable algorithm for producing parallel orders for interval graphs. Unfortunately, we also showed that there exist chordal graphs for which, even when allowing some constant-factor imbalance, the nested dissection algorithm will produce orders with super-linear fill.

There is a trade-off between the parallelism exhibited by an order and the amount of extra fill introduced by that order. In fact, by slightly reducing the amount of parallelism in an order, we obtained an algorithm that produces parallel orders for chordal graphs with linear fill. One technique for "sequentializing" the order is the use of *sentinels*. Sentinels are separators that help to sequentialize an order just enough to localize the fill within subgraphs of a chordal graph. The orders produced by our algorithm have linear fill and height within an $O(\log^2 n)$ factor of the optimal.

Our experience parallelizing and sequentializing elimination orders for chordal graphs led us down an interesting avenue. Nested dissection produces orders that are naturally parallel. This same parallelism is also responsible for some of the fill required by nested dissection orders. Thus, perhaps nested dissection could be improved by reducing the amount of parallelism in the orders it produces.

We have designed an algorithm that is a variant of nested dissection that produces orders with low fill but which are usually less parallel than orders produced by standard nested dissection algorithms. While still using separators to guide the ordering process, the algorithm does not necessarily assume that a separator should be ordered last, but only that it should be used to avoid fill between the different connected components it defines in the graph. That is, as long as the vertices within all but one of the components are ordered before the separator vertices, then the requirement that the separator vertices be ordered last no longer exists. Instead, the algorithm recurses on the subgraph formed by the last of the components along with the, as yet unordered, separator vertices. The actual algorithm is more involved, and cannot always decide which component should be ordered last. It sometimes reverts to regular nested dissection. This algorithm behaves very much like nested dissection, except that in certain cases when nested dissection misbehaves, this algorithm works better and produces low-fill orders. Unfortunately, have not yet been able to provide a theoretical analysis for this algorithm on general graphs. However we have proved that this algorithm produces zero-fill orders when applied to chordal graphs, which is not, however, the case for nested dissection.

### 1.2.1 Empirical results

We performed experiments with an implementation of this variant of nested dissection and state-of-the-art implementations of other ordering heuristics on matrices commonly used as benchmarks. We observed that nested dissection performed on average almost as well as a hybrid of minimum degree and nested dissection, the current champion algorithm. When given good enough separators, our modified version of nested dissection outperformed this hybrid algorithm by about 5 percent in terms of fill and 10 percent in terms of work on average, over the set of test matrices. It outperformed a minimum-degree ordering algorithm by about 20 percent in terms of fill, and 66 percent in terms of work on average, on the same benchmarks.

#### Nested dissection versus minimum degree

Even though we advocate nested dissection as a better approach for producing elimination orders than minimum degree, we cannot deny that, on certain graphs, minimum degree does generate better elimination orders in terms of both fill and work.

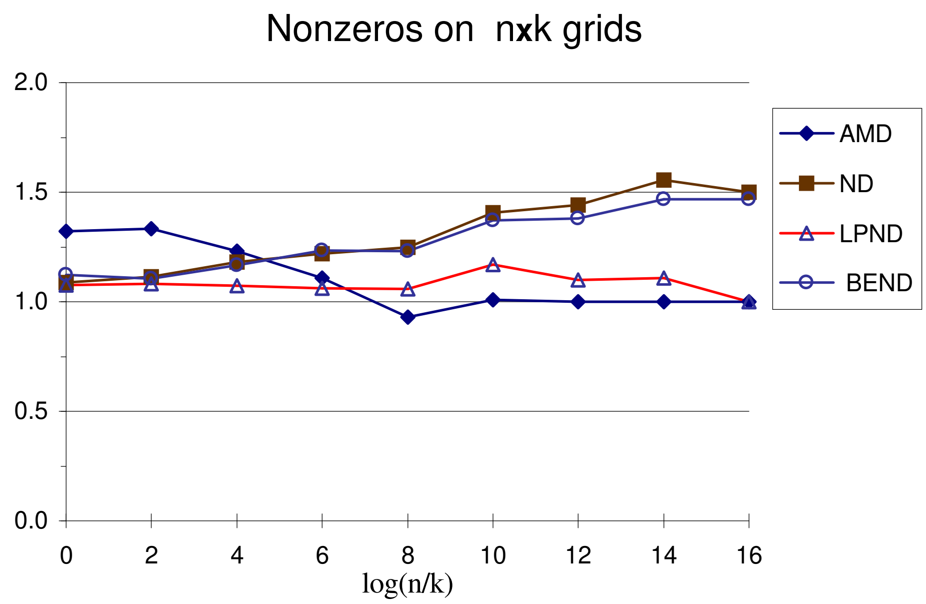



Figure 1.1: Number of nonzeros introduced by different elimination orders for $n \times k$ rectangular grids relative to a special-purpose order.

Figures 1.1 and 1.2 show the number of nonzeros and the amount of work for decomposing grids according to various ordering algorithms. In these experiments we used rectangular two-dimensional grid graphs of various aspect ratios, each of which had a total of $2^{16}$ vertices. Four ordering algorithms were applied to each graph, namely nested dissection (ND), the hybrid algorithm that we mentioned previously (BEND) [HR96, HR97], our "less parallel" variation of nested dissection (LPND) and a version of minimum degree (AMD) [ADD96]. We also applied a special-purpose algorithm that works only on two-dimensional grids, and uses diagonal separators as a basis for the ordering. The results for all orders were normalized to the results for this special-purpose order.

On graphs with large aspect ratios, the minimum-degree algorithm does best in terms of fill, but poorly in terms of parallelism. On these same graphs, the nested dissection and the hybrid algorithm produce orders that are substantially more parallel, but require more fill and work. For each of the orders produced by these algorithms we computed the minimum-height order that is equivalent to the order produced, in the sense that the new order produces the same fill edges. The heights of these minimum-height orders are shown in Figure 1.3, again, relative to the special-purpose order. We note that the LPND algorithm produces very sequential orders that are only a few percent more parallel than those produced by AMD. At least part of the additional fill experienced by the nested dissection and hybrid orders is a function of the additional parallelism in these orders.

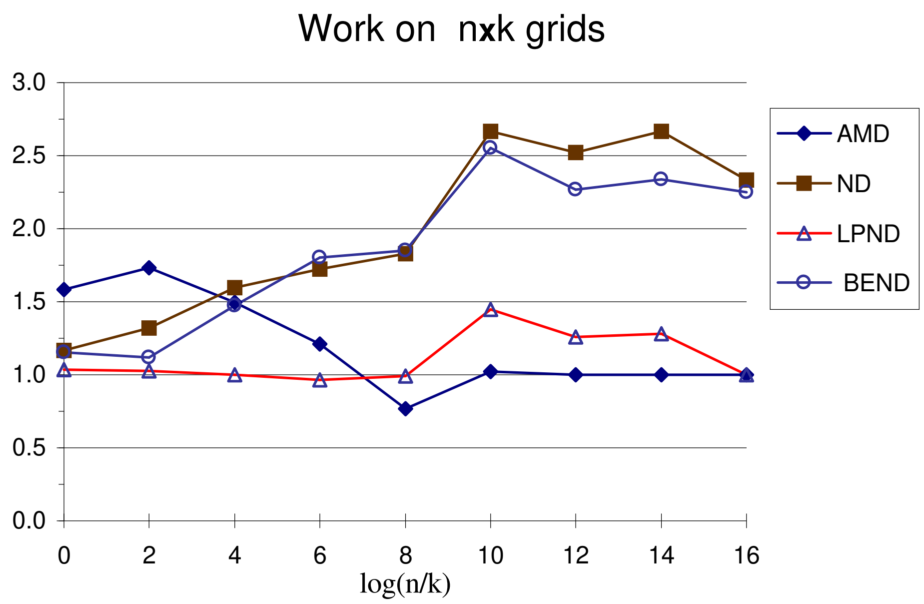



Figure 1.2: Work for different orders relative to a special-purpose order.

The following results stand out:

1. Contrary to popular belief, our results indicate that even though minimum-degree orders have low fill on graphs with large aspect ratios, in practice they are not very good for large graphs, such as the ones found in a number of sparse matrix applications. For smaller graphs, and in particular for small enough subgraphs of the input graphs, we often use minimum-degree as an ordering heuristic.

2. There exists a trade-off between exposing parallelism and producing low-fill orders. This trade-off can play a significant role when comparing orders that differ by only a few percent. We show that to obtain a parallel order for certain graphs we must allow some extra fill. Since nested dissection orders are highly parallel, this effect can sometimes make other ordering heuristics more attractive. By limiting the amount of parallelism that is exposed, we obtained a new algorithm that is a variant of nested dissection that, on our test cases, outperforms existing algorithms on average.

3. Given a sequential order, our parallelizing algorithm will produce a parallel elimination order with only a constant factor more fill and work than the initial order, while achieving a height within a factor of $O(\log^2 n)$ of optimum.

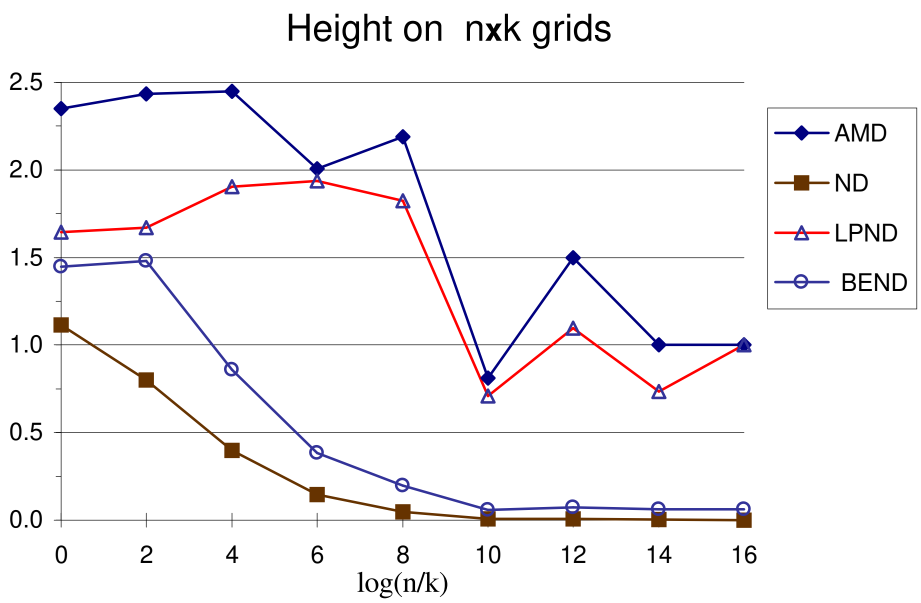



Figure 1.3: Height relative to a special-purpose order.

# Chapter 2

# Background and definitions

## 2.1 Gaussian elimination and LU decomposition

We start by reviewing Gaussian elimination and LU decomposition. Gaussian elimination solves a linear system of $n$ equations and $n$ variables by adding a multiple of the first equation to the remaining equations so as to remove the first variable from those equations, in what we call an elimination step. After the first elimination step we obtain a system of $n-1$ equations and $n-1$ variables, that can be processed in the same fashion. This method results in a system that corresponds to an upper triangular matrix, and can be easily solved by back-substitution.

Each of the elimination steps described above corresponds to multiplying the matrix representing the linear system by a matrix that is readily inverted. While multiplying this matrix by another matrix adds a multiple of the pivot row to a number of other rows, multiplying its inverse by a matrix subtracts that same multiple of the pivot row from the other rows. Thus, each one of these matrices and their inverses have the same non-zero pattern. These matrices have non-zeros in the diagonal and in the positions that correspond to each of the equations from which the variable is being eliminated in the current step.

The process known as $LU$ decomposition consists of "storing" the sequence of such elementary operations. If $E_1, E_2, \cdots, E_n$ are the elementary matrices corresponding to the Gaussian elimination of a matrix $A$, then the matrix $U$ obtained by Gaussian elimination can be written as $U = (E_n \cdot (E_{n-1} \cdots (E_2 \cdot (E_1 \cdot A)) \cdots))$. Thus we have $A = LU$, where $L = E_1^{-1} \cdot E_2^{-1} \cdots E_{n-1}^{-1} \cdot E_n^{-1}$ is a lower triangular matrix. The non-zero structure of $L$ corresponds to the non-zero structure of all the matrices $E_i$ added together, that is, $L$ can only have a non-zero in column $i$ row $j$ if at least one of the matrices $E_i$ also does. Each column of $L$ has non-zeros corresponding exactly to the non-zero entries on that column, below and including the diagonal, at the time the corresponding variable was eliminated.

## 2.2 Definitions

Entries in the matrix $L$ produced in the $LU$ decomposition of a matrix $A$ can be classified in two categories: *original* entries that correspond to non-zeros in $A$, and *fill* entries that correspond to new non-zeros introduced during the elimination process. The total number of floating-point operations to perform an $LU$ decomposition is called the *work* to decompose the matrix. The total amount of fill and work for a matrix decomposition depends on $A$, but also depends on the order of the equations and variables. Instead of decomposing the matrix $A$, we first find a suitable order for the rows and columns of the matrix. We can then either think of the decomposition process as choosing a pivot at each step according to the order, or we can permute the matrix, so that the elements are listed in the order they should be pivoted in. To maintain the symmetry of the matrix, we perform both row and column permutations, that is, given a permutation matrix $P$ for the matrix $A$ we actually decompose the matrix $PAP^T$. The order corresponding to $P$ is said to be an *elimination order*. When dealing with sparse matrices, it is interesting to consider the problem of finding the elimination order that minimizes either fill or work.

In order to understand and try to find low-fill and low-work orders, we model matrices as graphs, and LU decomposition as a graph operation, following Parter and Rose [Par61, Ros70]. Let $M = (m_{i,j})$ be a square $n \times n$ matrix. We associate with $M$ the graph $G$ with vertices $v_1, \cdots, v_n$ and edges $(v_i, v_j)$ iff $m_{i,j} \neq 0, i \neq j$. When $M$ is symmetric $G$ is an undirected graph.

Given a graph $G = (V, E)$ and a vertex $v$ in $V$ we call $N(v)$, the set of vertices adjacent to $v$ in $G$, the *neighborhood* of $v$ in $G$. When $G$ is unclear, we will refer to $N(v)$ as $N_G(v)$. A pair of vertices $v$ and $w$ are said to be *twins* if $N(v) \cup \{v\} = N(w) \cup \{w\}$. A *clique* of $G$ is a set of vertices any two of which are adjacent in $G$. A clique is said to be *maximal* if it is not contained in any larger clique. An *independent set* of vertices of $G$ is a set of vertices none of which are pairwise adjacent. An independent set is *maximal* if it is not contained in any larger independent set.

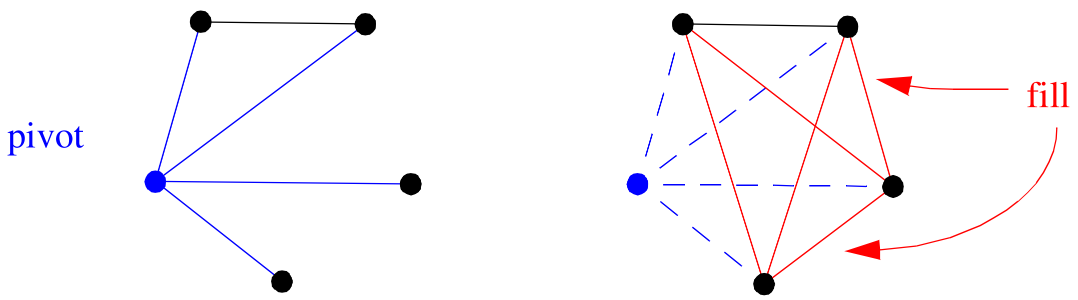



Figure 2.1: The elimination of the vertex corresponding to the pivot in a given step introduces fill edges between its neighbors.

Each step of Gaussian elimination on a symmetric matrix $M$ corresponds to finding the next variable and the corresponding vertex $v_i$ to be eliminated, adding edges to $G$ where necessary to make $v_i$'s neighborhood into a clique, and then removing $v_i$ from $G$. The vertex $v_i$ is said to have been *eliminated* from $G$. Any new edges introduced by the elimination of a vertex are called *fill* edges, or simply *fill*, and have a one-to-one correspondence with the fill entries in $L$. Figure 2.1

illustrates the process of eliminating a vertex.

Alternatively, we can think of the $LU$ decomposition process in terms of a different graph operation. In this case we do not remove the vertex being eliminated from the graph $G$, but rather mark it, just so it is not included in later steps in the neighborhoods of other vertices. This process augments $G$ into a supergraph $G^+$ obtained from $G$ by inserting all the fill edges into $G$. This augmented graph $G^+$ is referred to as the *filled graph*. Given two non-adjacent vertices $v_i$ and $v_j$ in $G$, there exists a fill edge $(v_i, v_j)$ in $G^+$ iff there exists a path from $v_i$ to $v_j$ going only through vertices $v_k$, $k < min(i, j)$, where $v_1, v_2, \cdots, v_n$ is the elimination order used to perform the LU decomposition [RTL76]. Two elimination orders are said to be *equivalent* if they produce the same filled graph.

An order $v_1, v_2, \cdots, v_n$ of the vertices of $G$ is a *perfect elimination order* if the elimination of the vertices of $G$ according to the order does not introduce any fill edges. A vertex $v$ is *simplicial* in a graph $G$ if $N(v)$ is a clique in $G$. Simplicial vertices are of special interest since the elimination of a simplicial vertex does not introduce any fill edges. An order $v_1, v_2, \cdots, v_n$ is *perfect* iff $v_i$ is simplicial in $G \setminus \{v_1, \cdots, v_{i-1}\}$.

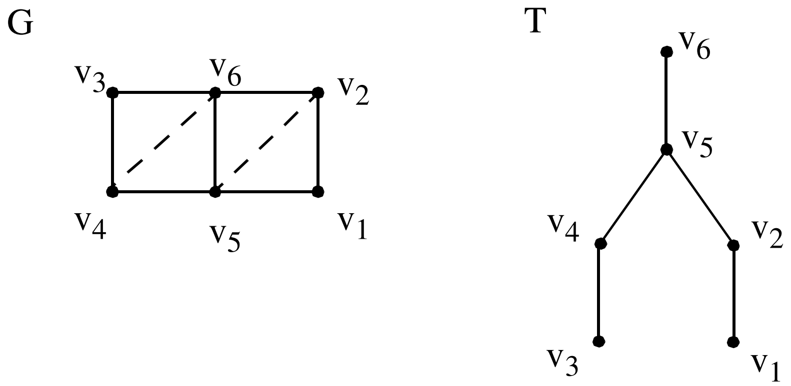



Figure 2.2: A graph $G$ and its elimination tree $T$ according to the order $v_1, v_2, \ldots, v_6$.

We can represent an elimination order by an *elimination tree*, i.e., a tree that encodes the actual precedence relations between the vertices of the graph. Given a graph $G$, and the graph $G^+$ obtained from $G$ by adding all the fill edges introduced by a given order $\pi$ of the vertices of $G$, the elimination tree of $G$ corresponding to $\pi$ is defined as follows: every vertex of $G$ is a vertex of the elimination tree and the parent of a vertex $v$ is the first of the neighbors of $v$ in $G^+$ that is ordered after $v$ in $\pi$. Figure 2.2 represents a graph $G$, and the graph $G^+$ obtained by adding the fill edges corresponding to the elimination order $v_1, v_2, \ldots, v_6$ to $G$. The fill is represented in the figure by the dotted edges. Figure 2.2 also shows the elimination tree $T$ of $G$ corresponding to the same elimination order.

Any vertices that do not have any children in the elimination tree can be eliminated in a single parallel step. The *height* of an elimination order is given by the height of the corresponding elimination tree plus one. The height of an order corresponds to the minimum number of steps to perform LU decomposition on the graph while respecting the precedence constraints imposed by the order. The *height* $H(G)$ of a graph $G$ is the minimum height over all elimination orders for that graph.

If we implement LU decomposition as a parallel algorithm, we can, at each step of the algorithm, eliminate an independent set of simplicial vertices. This parallel LU decomposition algorithm must eliminate vertices in a clique sequentially. We can also conceive a parallel decomposition algorithm that handles cliques as dense subgraphs. This new algorithm can take advantage of the sparsity of a matrix while applying a different, parallel dense decomposition algorithm to the dense sub-matrix corresponding to a clique, resulting in a very parallel algorithm. We define the number of *stages* of an order as the minimum number of parallel clique elimination steps to decompose a graph while respecting the precedence constraints imposed by the order.

The *reordering height* of a graph $G$ given an elimination order $\pi$ is the minimum height among all orders of $G$ equivalent to $\pi$. The *number of reordering stages* of a graph $G$ given an order $\pi$ is defined analogously. The reordering height of a graph given an order can be computed efficiently [JK82, LPP89], and so can the number of reordering stages of the graph.

A graph $G$ is said to be *chordal* if and only if every simple cycle with more than three vertices has a *chord*, that is, an edge connecting two non-consecutive vertices in the cycle. The class of chordal graphs is also the class of perfect elimination graphs [Dir61, Ros70].

Now consider the graph $G^+$ obtained from a graph $G$ by adding all fill edges introduced by some order. When eliminating $G^+$ according to the same order used to obtain $G^+$ form $G$, we observe that no new fill is created, and in fact $G^+$ is a perfect elimination graph. Thus, we also refer to the filled graph $G^+$ as a *chordal completion* of $G$. Chordal graphs have also been studied in other contexts, and have a number of interesting properties. Some NP-hard problems such as $k$-coloring and finding a maximum clique have linear time solutions for chordal graphs. Chordal graphs are also known by the names of triangulated graphs and rigid circuit graphs.

Chordal graphs have also been characterized as a particular class of intersection graphs. The *intersection graph* of a family $\mathcal{F}$ of sets $S_i$ is the graph obtained by associating a vertex $v_i$ with each set $S_i$, and edges $(v_i, v_j)$ whenever $S_i$ intersects $S_j$. Chordal graphs correspond to the intersection graphs of subtrees of a tree, i.e., each of the sets $S_i$ defining the intersection graph is a set of nodes that induces a connected subgraph of a tree. We call the tree in question a *skeleton* of the chordal graph. The skeleton along with the various subtrees forms a *tree representation* of the graph. A tree representation of a graph $G$ is said to be *minimal* if the associated skeleton has the minimum number of nodes possible. Gavril [Gav74] and Buneman [Bun74] showed that in a minimal tree representation there is a one-to-one correspondence between vertices of the skeleton $T$ and maximal cliques of $G$. Thus, a minimal tree representation of $G$ is called a *clique tree* of $G$.

Figure 2.3 exhibits a chordal graph $G$ with 8 vertices, numbered $v_1$ through $v_8$. The clique tree $T$ of $G$ (in this case unique) is also depicted in the figure. Each node of the clique tree corresponds to a maximal clique of $G$, namely the set of vertices of $G$ whose representative subtrees include that node of the skeleton $T$. Since all these subtrees include the particular node of the skeleton, the corresponding vertices in $G$ are connected and form a clique. For instance, the node $C_{128}$ in Figure 2.3 corresponds to the clique formed by $v_1$, $v_2$ and $v_8$ in $G$.

Each vertex of $G$ can appear in a number of different maximal cliques. The representative subtree for a vertex of $G$ is the subtree induced by the set of cliques that contain that vertex. For instance, the vertex $v_2$ is in the cliques $C_{128}$, $C_{2468}$ and $C_{234}$. These cliques induce the subtree $T_{v_2}$ that represents $v_2$ in the tree representation of $G$. Vertex $v_3$ on the other hand, is in only one clique, so that its representative subtree $T_{v_3}$ corresponds to the node $C_{234}$. These representative subtrees are represented in the figure slightly off from the skeleton tree, which is drawn with thinner lines.

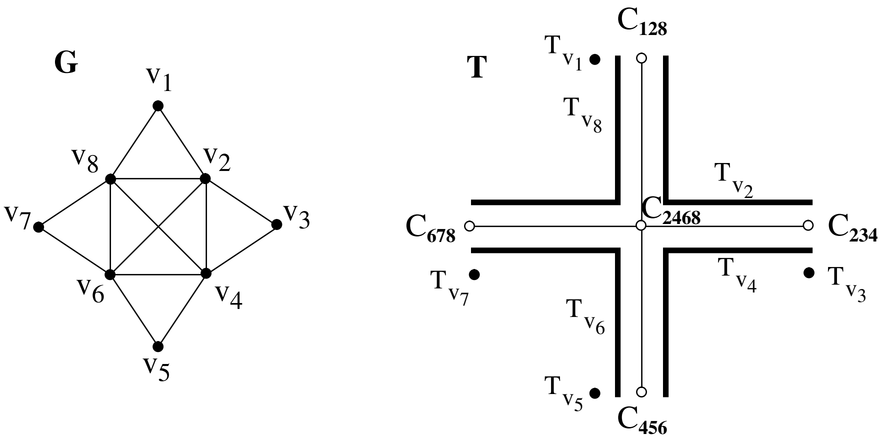



Figure 2.3: A chordal graph $G$ and its clique tree $T$. The nodes of $T$, labeled $C_{128}$, $C_{234}$, $C_{456}$, $C_{678}$ and $C_{2468}$, correspond to the maximal cliques of $G$.

Interval graphs constitute an important subclass of chordal graphs. These are chordal graphs that have paths for skeletons. In other words, an *interval graph* is the intersection graph of sub-paths of a path. The skeleton in question can be thought of as the real line, and the sub-paths as intervals, hence the name. A tree representation with a path for a skeleton is also called an *interval representation*. An *asteroidal triple* is an independent set of three vertices such that there is a path between each pair of vertices that avoids the neighborhood of the third. Interval graphs correspond to chordal graphs without asteroidal triples.

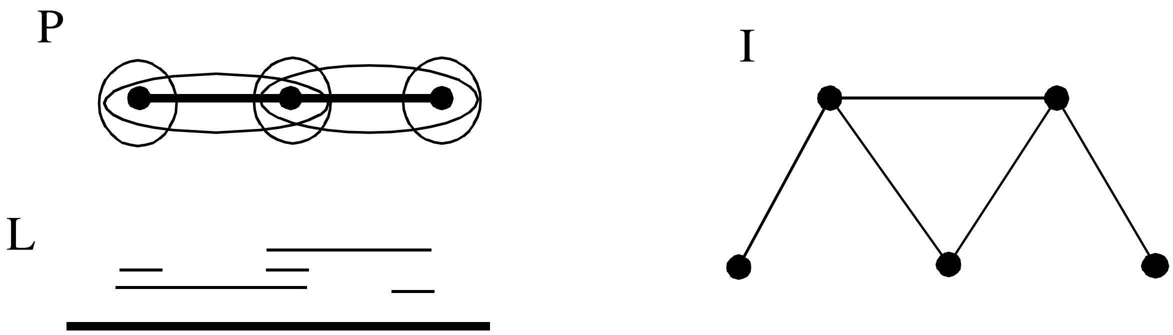



Figure 2.4: Representations of an interval graph $I$ as an intersection graph.

Figure 2.4 shows equivalent representations of the interval graph $I$. $I$ is represented as the intersection graph of subpaths of the path $P$ and as the intersection graph of intervals of the line $L$.

Throughout this thesis we refer to vertices in a graph, but we will usually use the term *node* to refer to vertices that correspond to sets of vertices in some other graph. For instance, we will refer to vertices of a chordal graph, but nodes of its skeleton, for each node corresponds to a clique in the chordal graph. Given a tree representation of a graph $G$, the subtree $T_v$ is said to *cover* a node/edge of the skeleton if that node/edge is in $T_v$. The *ply* of an edge $e$ of a skeleton is the number of distinct subtrees $T_v$ that cover $e$. A *terminal branch* of $T$ is a maximal path from a leaf $v$ to a node $w$ in $T$ that, except for $v$ and $w$, contains only nodes of degree two in $T$.

Let $G = (V, E)$ be a graph. We denote by $G \setminus v$ and $G \setminus S$ the subgraphs of $G$ induced by $V \setminus v$ and $V \setminus S$ respectively. Given a subgraph $H$ of $G$, the *boundary* $N_G(H)$ of $H$ in $G$ is the set of vertices in $G \setminus H$ that have neighbors in $H$. A *separator* $S$ of $G$ is a set of vertices such $G \setminus S$ consists of two or more connected components. A separator is *minimal* if it does not contain any smaller separators. It can be shown that a graph is chordal if and only if every minimal vertex separator is a clique.

An $\alpha$-*balanced separator* of $G$ is a set of nodes $S \subseteq V$ such that no connected component of $G \setminus S$ has more than $\alpha \cdot |V|$ vertices, for some $\alpha < 1$. An $\alpha$-*balanced separator tree* of $G$ is a tree whose nodes are $\alpha$-balanced separators of the subgraphs of $G$. The root of the separator tree corresponds to an $\alpha$-balanced separator $S$ of $G$, and its children correspond to $\alpha$-balanced separator trees of the connected components of $G \setminus S$. We also use the term *balanced separator* to refer to an $\alpha$-balanced separator, for some constant $\alpha$. A class of graphs satisfies an $f(n)$-*separator theorem* with constants $\alpha < 1$ and $\beta > 0$ if every graph $G$ with $n$ vertices in the class has an $\alpha$-balanced separator $S$ with no more than $\beta \cdot f(n)$ vertices. Planar graphs satisfy a $\sqrt{n}$-separator theorem with $\alpha = 2/3$ and $\beta = 2 \cdot \sqrt{2}$ [LT79].

# Chapter 3

# Related work

In this chapter we present some existing ordering heuristics and look at some of the many ways of representing chordal graphs. We also mention related work on interval graph completion and work on elimination orders with low height.

## 3.1 Fill minimizing heuristics

In this section we present a brief overview of some popular heuristics for producing orders with low fill.

### 3.1.1 Minimum deficiency

The number of edges that are needed to make the neighborhood of a vertex $v$ into a clique is the *deficiency* of $v$, also known as the *local fill* at $v$. This corresponds to the amount of fill introduced by the elimination of $v$ in $G$. The minimum deficiency heuristic [Mar57, TW67] is a greedy algorithm that tries to minimize the overall amount of fill introduced by the elimination process by, at each step, choosing to eliminate a vertex with minimum deficiency.

Every chordal graph has at least one vertex with zero deficiency (a simplicial vertex). Eliminating a simplicial vertex corresponds to removing that vertex from the graph since its neighborhood already forms a clique. The graph obtained is an induced subgraph of the initial graph and is thus chordal since any induced cycle of length greater than three in this new graph must have a chord (for it had one in the initial chordal graph). By repeatedly applying this argument we find that every chordal graph has a perfect elimination order, and that the minimum deficiency heuristic produces perfect elimination orders on chordal graphs.

In practice, the minimum deficiency heuristic is discarded in favor of other ordering heuristics, probably because of the large amount of time necessary to update the deficiency information as the ordering algorithm progresses. Recently, however, Rothberg and Eisenstat [RE98] pointed out that, in practice, the minimum deficiency heuristic produces elimination orders with low fill. In fact, in their experiments, the minimum-deficiency heuristic produced orders that were better than the ones produced by the minimum-degree heuristic, which is often used in practice. We describe the minimum-degree heuristic in the next section.

Rothberg and Eisenstat [RE98] propose approximate versions of the minimum deficiency heuristic, that they call *approximate minimum local fill* (AMF) and *approximate minimum mean local fill* (AMMF). These heuristics work by computing cheaper approximations of the deficiency of the vertices in the graph, thus avoiding the prohibitive cost of computing a minimum deficiency order, while still producing elimination orders with low fill.

### 3.1.2 Minimum-degree

The *minimum-degree* heuristic originated with the work of Markowitz in the late 50's and has undergone several enhancements in the years since. In its simplest form, the minimum-degree algorithm repeatedly finds a vertex of minimum degree and eliminates it. This very natural greedy algorithm works surprisingly well in practice.

A simple enhancement that does not affect the quality of the orders but makes for a more efficient algorithm, is the identification of twin vertices, i.e., vertices that have the same set of neighbors, into sets called *supervariables* or *supernodes*. After a vertex within a supernode is eliminated all other vertices within that same supernode become simplicial – for the elimination of that first vertex makes its neighborhood into a clique. If the vertex removed had minimum degree in the graph, then the remaining vertices in the same supernode must also have had minimum degree, and still do, since their degree is reduced by one when the vertex is removed, and the minimum degree in the graph was $d$, so that after the removal of one vertex it must be at least $d - 1$. Thus, supernodes can be treated as a single vertex in the graph, reducing the amount of work needed to compute an elimination order. All the heuristics that we describe can take advantage of supernodes.

To further reduce the running times for the minimum-degree algorithm, Liu proposed the multiple minimum-degree algorithm (MMD) [Liu85]. In this algorithm, an independent set of nodes with minimum degree is eliminated at each step, thus allowing for a smaller number of degree update steps, while producing a naturally parallel order.

Amestoy et al. [ADD96] proposed an approximate minimum degree algorithm (AMD). This algorithm uses estimates for the degree of the vertices instead of actually computing the exact degrees. The implementations of both the AMD and the MMD algorithms, state-of-the-art improvements upon the minimum-degree heuristic, produce similar quality orders.

Another enhancement present in both AMD and MMD is the use of external degrees for the supernodes. The external degree of a supernode is the degree of any vertex in that supernode minus the number of neighbors of that vertex that are in that same supernode. This alternate measure tries to compensate for the fact that after the first vertex is eliminated no additional fill is introduced by the elimination of the remaining vertices in the supernode.

With all these algorithmic improvements, minimum-degree orders can be computed in very little time. The orders produced usually have relatively low fill. Nonetheless, a minimum degree algorithm algorithm can produce orders with large amounts of fill. Berman and Schnitger [BS90] show that on a toroidal square mesh there are orders consistent with the minimum degree heuristic that exhibit $\Omega(n^{\log_3 4})$ fill, substantially more than the the optimal fill for toroidal square meshes which is $O(n \log n)$. However, there do exist minimum-degree orders for the toroidal square mesh with low fill. In any case, in the absence of a good tie-breaking strategy, a minimum-degree algorithm is likely to exhibit, to some extent, the same behavior described by Berman and Schnitger, namely the presence of large cliques in the chordal completion. We elaborate on their example in Section 3.2.

### 3.1.3 Nested dissection

*Nested dissection* was first proposed by George [Geo73] as a method for ordering vertices in a mesh. It was later generalized by George and Liu [GL78] to work on arbitrary graphs. Unlike the heuristics we have considered so far, nested dissection builds the elimination order in the reverse direction, that is, it finds a vertex separator and orders those vertices to be eliminated last. It then recurses on the connected components defined by the removal of the separator vertices. The various components can be ordered in any relative order (or even be interleaved), without affecting the work or fill for the order produced, since the separators stop any fill from forming between the components.

It is common to represent nested dissection orders as elimination or separator trees. A *separator tree* is the tree we obtain by grouping the vertices in the top level separator into a node, and then making its children correspond to the separator trees for the connected components obtained by the removal of the top level separator. A nested dissection order corresponds to any order in which every vertex is eliminated after its descendants in the separator tree.

Separator trees make for a convenient representation of elimination orders and the corresponding fill. A separator "blocks" the creation of fill between sibling subtrees, so that fill can only occur between vertices that have an ancestral relationship in the separator tree. As a component is eliminated, some fill is introduced between the vertices of the separator, which will typically become a clique. Thus, it is advantageous to keep separators small to limit the amount of fill produced. Local heuristics, such as minimum-degree and minimum local fill, cannot find good separators for the higher levels of the separator tree in general. By examining the graphs as a whole and finding small balanced separators, nested dissection produces orders that are provably good in terms of fill.

The first analysis for a variant of nested dissection for a class of graphs closed under subgraphs (that is, a class of graphs such that every subgraph of a graph in the class is also in the class), for which a $\sqrt{n}$-separator theorem holds, was given by Lipton, Rose and Tarjan [LRT79]. This variant of nested dissection, LRT, produces orders with $O(n \log n)$ fill on an $n$-node graph in these classes. Subsequently, Gilbert and Tarjan [GT87] analyzed the original nested dissection algorithm of George and Liu and showed that it also yields $O(n \log n)$ fill for planar graphs, graphs with bounded genus, and graphs with bounded degree that have $O(\sqrt{n})$ separators [LT80]. They also point out that this method does not work in general for graphs with $O(\sqrt{n})$ separators by constructing a counterexample, hence showing that the modifications in [LRT79] are essential in this case. Unlike George and Liu's algorithm, LRT includes the separators in the recursive calls, that is, if a separator $S$ divides the graph into components $A$ and $B$ then the original generalized nested dissection algorithm would recurse on $A$ and $B$, and would order the vertices in $S$ last. The LRT algorithm also orders the vertices in $S$ last, but recurses on $A \cup S$ and $B \cup S$, while taking note that in the recursive calls all the nodes in $S$ have already been ordered, and are only present to help find the appropriate separators.

Both versions of nested dissection [GT87, LRT79] produce orders with $O(n^{\frac{3}{2}})$ work on planar graphs. It is interesting to note that there are $n$-node planar graphs (square grids in particular) for which any elimination ordering has $\Theta(n \log n)$ fill and $\Theta(n^{\frac{3}{2}})$ work [HMR73].

Agrawal et al. [AKR93] gave the first approximation algorithms for elimination orders that simultaneously minimize the fill, height, and work, all within a polylogarithmic factor of optimal when the degree of the input graph is bounded. In general, their elimination orders have fill within a factor of $O(\sqrt{d} \log^4 n)$ of the minimum number of nonzeros (including fill and original edges), and height within a factor of $O(\log^2 n)$ of optimum where $d$ is the maximum degree of the graph. Their algorithm is the nested dissection algorithm using approximate minimum-sized balanced-node separators [LR88] to construct the recursive decomposition. If they were actually able to obtain minimum-sized balanced node separators their algorithm would produce an order with fill within a factor of $O(\sqrt{d} \log^2 n)$ of the minimum number of nonzeros, and height within a factor of $O(\log n)$ of the optimum.

Although the proof is not constructive, Gilbert [Gil87] showed that for any graph there exists a nested dissection order with fill within a factor of $O(d \log n)$ of optimal, where $d$ is the maximum degree of the vertices in the graph. In Section 4.2 we exhibit a graph for which a nested dissection algorithm that chooses minimal balanced separators produces orders that induce a factor of $\Omega(\sqrt{d \log n})$ more fill than the minimum number of nonzeros.

We note that in practice finding good separators accounts for the large running times of nested-dissection-based algorithms when compared to other heuristics. It is common practice to finish off a nested dissection order by applying a faster ordering algorithm to order small enough subgraphs. *Constrained minimum degree*, i.e., a minimum degree algorithm that orders vertices of a sub-graph according to increasing degrees when eliminated from a larger graph, is usually the heuristic of choice for these small graphs.

### 3.1.4 Hybrid algorithm

Once we have understood how the minimum-degree and nested dissection heuristics work and what are the advantages and drawbacks of each, it is natural to consider hybrid algorithms that take advantage of the best characteristics of each. Using a few levels of separators seems like the ideal remedy to control the the amount of fill introduced by minimum degree orders.

Recently, Hendrickson and Rothberg [HR96] and independently, Liu and Ashcraft [AL96] proposed hybrid algorithms that, in practice, produce orders that compare favorably with state-of-the-art nested dissection and minimum degree algorithms. Neither hybrid algorithm has known worst-case fill or work analyses.

We concentrate here on the hybrid algorithm of Hendrickson and Rothberg in [HR96, HR97], which, to our knowledge, is the current champion as far as elimination order algorithms go. Henceforth, we refer to Hendrickson and Rothberg's algorithm as *the* hybrid algorithm, or as they chose to refer to it, as the BEND algorithm.

The BEND algorithm works as follows. It finds a few levels of separators, just like nested dissection would. It sets those vertices aside for elimination at the end. It then finds a constrained minimum-degree order for each of the connected subgraphs formed. Finally, the algorithm re-orders the vertices that it set aside, i.e., the vertices within the separators that were initially found. This final step is accomplished by applying the AMD algorithm to the graph obtained by eliminating the vertices that have already been ordered, i.e., the usual nested dissection order among separator vertices is thrown away and a new order is computed.

This way, the algorithm is able to ignore the nested dissection assumptions as to what a good elimination order should be, while still avoiding fill between the various subgraphs defined by the separator vertices and keeping the boundaries of the subgraphs small. The resulting hybrid algorithm produces orders with very little fill in a small amount of time.

Liu and Ashcraft's algorithm is very similar to that of Hendrickson and Rothberg's although it is cast in a different light. Liu and Ashcraft's algorithm finds a set of vertices that partitions the graph in a large number of connected components at once, instead of finding multiple levels of separators that, taken together, achieve a similar effect.

## 3.2 Elimination orders

Elimination orders can be viewed in different lights. We discuss a few of these different views in this section.

### 3.2.1 Elimination trees

As we mentioned before, chordal graphs coincide with the class of graphs that have perfect elimination orders, so that finding an order that produces the minimum amount of fill corresponds to finding a chordal supergraph of an initial graph with the fewest possible number of edges. On the other hand, chordal graphs correspond to intersection graphs of subtrees of a tree. In this section we show how to construct a tree representation for a chordal completion of a graph based on a given elimination order.

In an elimination tree, a vertex precedes its tree ancestors in the elimination order and any order that respects the ancestral precedence constraints is equivalent to the initial one, in the sense that it produces the same chordal completion. By construction, the edges of a graph $G$ can only connect vertices that have an ancestral relationship in its elimination tree $T$.

Figure 3.1 represents a graph $G$ and its chordal completion according to the order $v_1$, $v_2$, $v_3$, $v_4$, $v_5$, $v_6$. The dotted edges represent the fill introduced during the elimination process. Figure 3.1 also depicts the elimination tree of $G$ according to the same order. Each vertex of the elimination tree corresponds to a vertex $v$ in the original graph, and can be thought of as corresponding to the maximal clique formed by $v$ and its neighbors at the time $v$ is eliminated. We also represent these

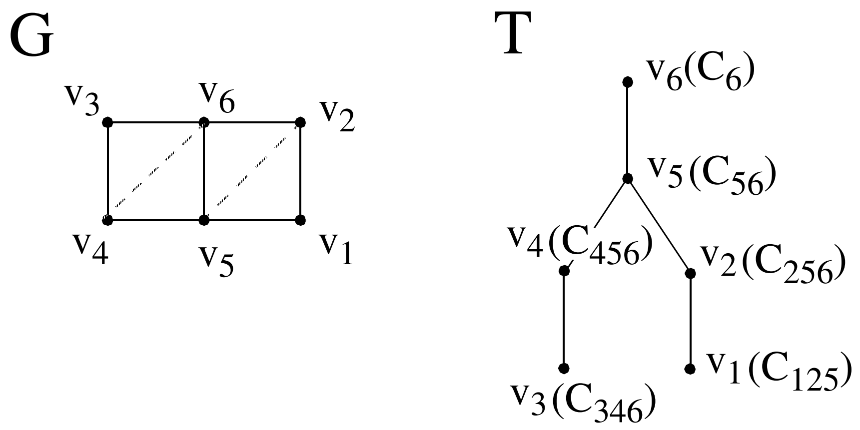



Figure 3.1: The chordal completion of $G$ and the corresponding elimination tree.

cliques in Figures 3.1 and 3.2. The label $C_{125}$ for instance, corresponds to a clique formed by vertices 1, 2 and 5 in the chordal completion of the graph $G$.

Figure 3.2 depicts two equivalent representations of the chordal completion of $G$; the chordal completion can be either viewed as the intersection graph of the subtrees $T_i$ in the figure, or as the graph obtained by connecting vertices whenever they both belong to one of the sets $C_{\{\}}$ that correspond to cliques in the chordal completion. This representation also requires a tree structure for the cliques, namely it requires that when we restrict our attention to the cliques that contain any specific vertex, they span a connected subgraph of a fixed tree. This subgraph is exactly the tree one would use to represent the vertex in the intersection-of-subtrees representation of the graph.

Both representations can be easily obtained from the elimination tree. Let $T$ be an elimination tree for $G$. Let $T_v$ be the maximal subtree of the elimination tree $T$ rooted at $v$ whose leaves are neighbors of $v$ in $G$, that is, the tree obtained as the union of all paths $v - w$ from the vertex $v$ to all neighbors $w$ of $v$ in $G$ that are descendants of $v$ in $T$. The elimination tree $T$ along with the subtrees $T_v$ form a tree representation of the chordal completion of $G$ corresponding to the order given. That is to say, the chordal completion of $G$ is the intersection graph of the subtrees $T_v$.

Note that the nodes labeled $C_6$ and $C_{56}$ in Figure 3.2 do not add to the representation, since they are superseded by $C_{256}$. Both nodes can be removed and replaced by an edge connecting $C_{456}$ directly to $C_{256}$. In general, any chordal graph has a minimal tree representation with a one-to-one correspondence between nodes and the maximal cliques of the chordal graph. Two nodes that are adjacent in the tree representation can be contracted when the set corresponding to the first contains the set corresponding to the second, since all adjacencies implied by the second are already implied by the first. A minimal representation (in which every node corresponds to a maximal clique) can always be obtained in this fashion.

For the sake of completeness, we also mention that a perfect elimination order can be obtained from a tree representation of a chordal graph $G$ as follows. Let $l$ be a leaf of the skeleton for the

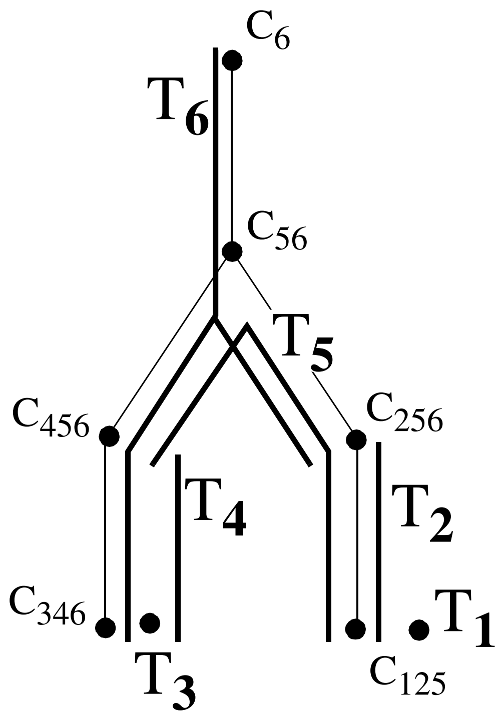



Figure 3.2: Tree representation of the chordal completion of the graph in Figure 3.1. The subtree $T_6$ corresponding to the vertex $v_6$ intersects all other subtrees, except for $T_1$, since after the fill edges are inserted $v_6$ is adjacent to all vertices of the graph except for $v_1$.

given tree representation. Either no representative subtree consists solely of $l$, in which case we can remove $l$ from the skeleton tree, or there exists at least one such subtree $T_v$. In this case, $T_v$ corresponds to a simplicial vertex and can be taken as the first vertex in the elimination order of $G$. We proceed by removing $T_v$ from the tree representation and iterating the same algorithm, until all vertices have been ordered. More than one elimination order can be produced in this fashion. In the tree representation in Figure 3.1, for instance, the order $v_3, v_4, v_6, v_5, v_1, v_2$ would produce the same chordal completion.

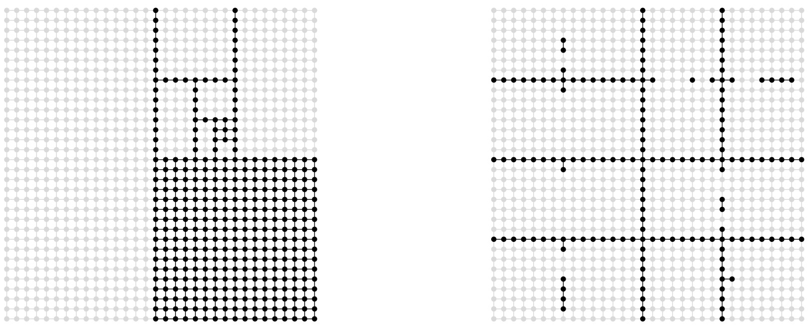
### 3.2.2 Graphs with "almost" planar representations

In this section we describe how elimination orders for certain graphs can be visualized in the plane. This is yet another way of representing the process of eliminating a graph and can provide some insight into different ordering heuristics.

We examine graphs that can be easily represented on a 2-D surface. These include planar graphs, but also non-planar graphs, such as a torus. To more easily visualize some of these graphs we will omit some of their edges so as to obtain a planar representation. For the sake of argument, we further assume that the faces that are internal in the planar representation correspond to cliques, that is, we assume that each face represents a clique formed by the vertices on the face. During the elimination process we will remove vertices and make their neighborhoods into cliques. As long as the vertex is not in the external face of the planar representation, its neighbors correspond to the set of vertices that share a face with it in the planar graph. By removing that vertex we effectively merge the various faces the vertex belongs to. Instead of adding the new fill edges, we maintain the planar representation, and think of every internal face as corresponding to a clique. In practice even if some small faces do not form cliques we can just consider them as forming cliques anyway, without changing our understanding of the elimination process by much. Any elimination order can be visualized in this somewhat imprecise fashion. This is essentially the same model used by George in the early 70's to describe and analyze nested dissection on square meshes [Geo73].

Figure 3.3(a) represents a stage in the elimination of a square $32 \times 32$ grid according to a nested dissection order. All of the vertices to the left of the center vertical line have already been eliminated at this step, but instead of being removed from the picture, they were represented in gray. All other vertices in gray have also been removed. From the progression of the elimination we notice that the center vertical line corresponds to the top-level separator. Subsequent separators were chosen by recursively cutting the graph approximately in half, horizontally and then vertically, as can be seen in the figure.

After the vertices to the left of the center vertical line are removed the vertices on that line "see" each other and form a clique. The vertices in the smaller faces that are created also form cliques.

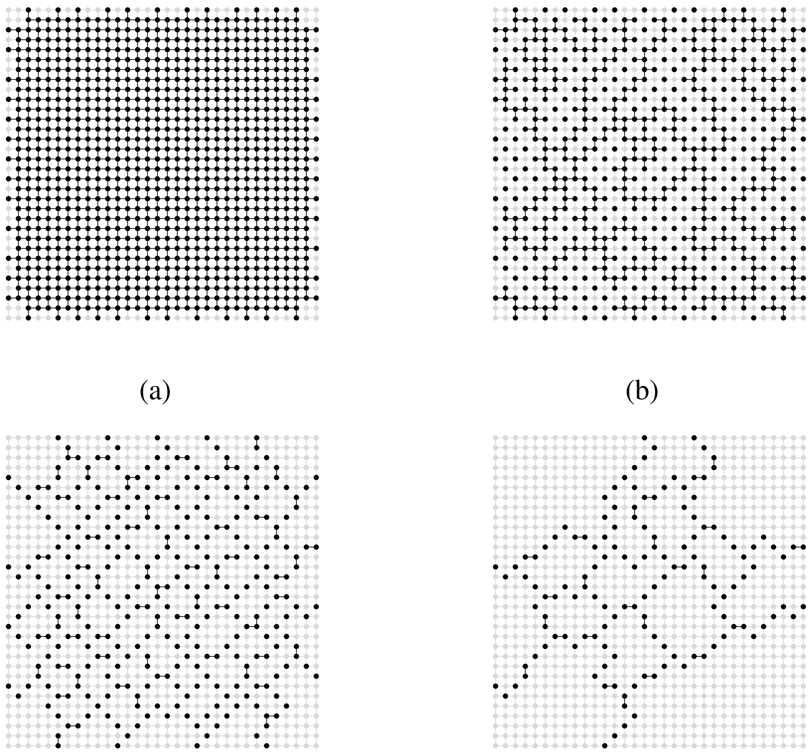


(a) (b)

Figure 3.3: Partial elimination according to a nested dissection order.

A global view of a nested dissection order is given by the corresponding separator tree. Figure 3.3(a) implicitly depicts one of the branches of the separator tree. Instead, we can visualize the elimination according to a different order, in which a few levels of leaf nodes of the separator tree are eliminated, as shown in Figure 3.3(b). If we do not have an explicit separator tree, a parallel order equivalent to a given nested-dissection order would achieve the same effect.

A minimum-degree order would look very different from Figure 3.3(a) but somewhat similar to Figure 3.3(b). Vertices along the boundaries of the grid would be eliminated first, and then a

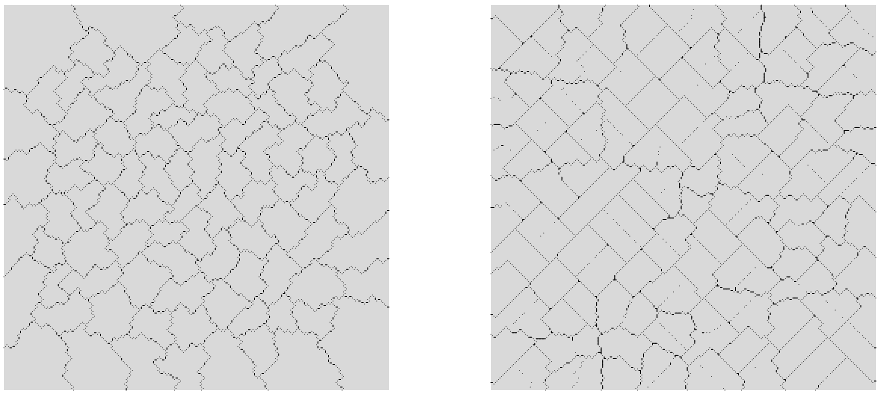


(a) (b) (c) (d)

Figure 3.4: Different points in the elimination of a square grid according to an AMD order.

large independent set of internal vertices would be eliminated. The minimum-degree order that would follow would likely eliminate vertices not necessarily close in the graph in successive steps. However, whenever a vertex is eliminated, its twins would be eliminated in the next few steps, for they would have minimum degree at that time. The faces in the pictorial representation would merge with neighboring faces and "grow" as the elimination progresses. Figure 3.4 depicts various stages of Gaussian elimination on a $32 \times 32$ grid, according to an order produced by AMD. When a vertex internal to the grid is eliminated, its four neighbors become a clique, which is not clearly indicated in the figure, for we did not include the corresponding diagonal edges. Nevertheless, we can see how the order progresses.

Figure 3.5 shows a point in a parallel order that respects the precedence constraints of an AMD order (a) and a point in a parallel order that respects the precedence constraints of a nested-dissection order (b) for a square $256 \times 256$ grid. In both cases about 5000 of the 65536 vertices still remain to be eliminated. The grid given as input to each ordering algorithm had its vertices listed in a random order. In this pictorial representation we can see that the faces created by the nested dissection order are smoother, that is, have contours that are closer to straight lines, than the faces from the AMD order.



(a) (b)

Figure 3.5: Partially eliminated views of a square grid. The order used in (a) was produced by a minimum-degree algorithm, while the one used in (b) was produced by a nested dissection algorithm.

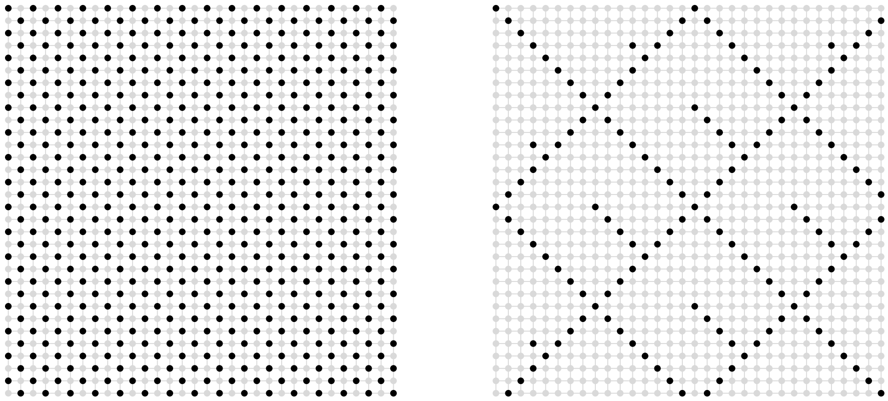
Berman and Schnitger [BS90] show that minimum-degree orders can produce very large faces. By carefully breaking ties among vertices with the same degree, they were able to construct a minimum-degree order for which the faces in our pictorial representation develop into fractals. The resulting order has $\Omega(n^{\log_3 4})$ fill and $\Omega(n^{1.5 \log_3 4})$ work, while optimal orders for the same graphs have $O(n \log n)$ fill and $O(n^{1.5})$ work. On the other hand, nested dissection is guaranteed to produce orders with fill and work within a constant of the optimal for these graphs.

The problem with minimum-degree orders lies in this fractaling effect. Berman and Schnitger pointed out that since minimum-degree is a local heuristic it cannot adequately control the "shape" of the faces. On small graphs, this is not much of an issue, but on larger graphs, long, irregular boundaries form between faces. These boundaries correspond to large cliques in the chordal completion, and thus to additional fill and work. Figure 3.5 accurately reflects the difference in quality between the two orders depicted; The AMD algorithm produced an order with 21 percent more fill and 62 percent more work than the nested dissection order. If we do not present the vertices of the input graph in a random order then both algorithms produce better orders. Local algorithms such as minimum-degree are more susceptible to the initial order of the vertices, while nested dissection, having a global view of the graph, should be less affected by the order.

**Nested dissection on square grids**

We can use the knowledge and intuition we gained from examining elimination orders in the previous sections to show how to obtain better elimination orders for square grids. This improves on the more traditional horizontal and vertical separators for square grids. This result was first presented in [BG73] but nonetheless is interesting to re-visit it.

Traditionally, nested dissection on square grids and toroidal grids uses vertical and horizontal separators, as depicted in Figure 3.3. The pictorial representation of the resulting elimination order can help guide us through an analysis, similar to the original one presented by George [Geo73], that shows that indeed the order produced for an $n \times n$ grid (or torus) has $7.75 \cdot n^2 \log n + O(n^2)$ fill.



(a) (b)

Figure 3.6: Steps in the elimination of a 32 × 32 grid according to a parallel nested dissection order with diagonal separators.

Looking at pictures such as the one in Figure 3.5 we notice an interesting fact. In a grid graph, the subgraph contained in a $k \times k$ square whose edges are horizontal and vertical contains $k^2$ vertices, while the subgraph contained in a $k \times k$ square whose edges are diagonal has about twice as many vertices. Figure 3.6 shows two steps in a nested dissection order obtained using diagonal separators instead of the traditional horizontal and vertical ones.

This extra factor of two in the ratio between the perimeter of these (square) regions squared and the number of vertices in the region, translates to an order that has a factor of 2 less fill than the traditional nested dissection on square grids.

Take an $n \times n$ square toroidal grid, and a nested dissection order for that grid using diagonal separators. Since a grid is a subgraph of the toroidal grid, the same analysis works for square grids. The first step of that order eliminates every other vertex of the graph, as can be seen in Figure 3.6(a). This step causes $O(n^2)$ fill to be introduced. The graph obtained can be thought of as composed of two $n/2 \times n/2$ square grids that have been rotated 45 degrees, and have had each $1 \times 1$ square made into a clique. But the same analysis that was used on square grids with horizontal and vertical separators can be used to show that $k \times k$ grids such as these can be eliminated with $7.75 \cdot k^2 \log k + O(k^2)$ fill. The total amount of fill for this nested dissection order is $2 \cdot 7.75 \cdot (n/2)^2 \log(n/2) + O(n^2)$, i.e., $3.875 \cdot n^2 \log n + O(n^2)$.

## 3.3 Interval graph completion

Interval graph completion, that is, finding a set of edges of minimum cardinality that when added to a graph make it into an interval graph, is closely related to the chordal completion problem. Just as with chordal completion, we measure how close a solution is to optimal with respect to the total number of edges in the interval completion.

Ravi et al. [RAK91] present an $O(\log^2 n)$ approximation algorithm for the interval graph completion problem. Even et al. [ENRS95] present an approximation algorithm that among other problems, solves the interval completion problem within an $O(\log n \cdot \log\log n)$ factor of optimal. Their approach consists of casting a solution in terms of a linear program that includes a set of constraints that try to ensure a good quality solution by imposing an average distance within sets of vertices, in what is called a *spreading metric*. The cost function is a lower bound on the cost of any solution for this problem, so that a solution within a certain factor of the minimum cost is also within that same factor of optimal. Rao and Richa [RR98] improve on their result by presenting an algorithm that uses the same cost function and obtains a solution within an $O(\log n)$ factor of the optimal.

## 3.4 Related work

### 3.4.1 Height

Computing an elimination ordering for a given graph with minimum height is NP-hard [Pot88], and remains so even if an additive error in the estimate of the height is allowed [BGHK95]. However, given a chordal graph there are efficient algorithms that find perfect elimination orders for the graph with minimum height [JK82, Liu89].

Pan and Reif give one of the first analyses of the parallel height of nested dissection orderings as well as how nested dissection can be used for solving the shortest-path problem in graphs [PR85, Pan93]. Bodlaender et al. [BGHK95] uses an approach similar to [LRT79] and [AKR93] to find elimination orders with bounds on the height and several related parameters. Both [AKR93] and [BGHK95] give elimination orders with height at most $O(\log^2 n)$ times the minimum possible, for any $n$-node graph. Numerous heuristics without performance guarantees are also known for the problem of finding a chordal completion with minimum height [Gea90, JK82, LL87, LPP89, Liu89, LM89].

Manne [Man91] shows how to produce optimal height orders for trees with fill linear in the number of edges in the tree. Aspvall [Asp95] presents a class of chordal graphs for which a perfect elimination order has height equal to the minimum possible height plus one, but for which every minimum height elimination order has super-linear fill. Aspvall and Heggernes [AH94] present a polynomial time algorithm that finds elimination orders with optimal height for interval graphs. We discuss the trade-offs between low-height and low-fill orders in the next chapters.

# Chapter 4

# Parallel Gaussian elimination

We begin our study by examining parallel orders for specific classes of graphs, namely interval and chordal graphs. The purpose of this study is twofold. First we note that any graph along with the fill edges introduced by a given order is a chordal completion of the graph. We are thus studying how we can, given a certain order (and the corresponding chordal graph), produce another order that is parallel and does not have too much additional fill or work. Second, the rich structure of these classes of graphs provides us with insight into the related problem of finding orders that minimize fill and work.

Efficient algorithms that find perfect elimination orders for chordal and interval graphs are known. However, perfect elimination orders might not be suitable for elimination in parallel. Our goal is to obtain an algorithm that will take a chordal graph and produce a parallel elimination order for that graph while introducing fill that is at most linear in the number of edges in the initial chordal graph. We start by showing that for interval graphs a certain nested dissection algorithm has this property. However, the same nested dissection algorithm can introduce a super-linear amount of fill on chordal graphs. We accomplish our goal by introducing the notion of sentinels. Sentinels are separators that sequentialize the orders just enough to localize the fill to within subgraphs. Our final algorithm obtains orders with $O(m)$ fill, work within a constant factor of optimal, and height within an $O(\log^2 n)$ factor of optimal, on chordal graphs.

In this and in the next chapters, we measure how close an order is to having optimal fill as a function of the total number of nonzeros in the minimum-fill elimination order of the graph. That is, when we say an order has linear fill, we really mean that the amount of fill it introduces is at most a constant times the total number of nonzeros in the minimum-fill elimination order of the graph which includes both fill and original entries.

## 4.1 Nested dissection on interval graphs

Nested dissection produces naturally parallel orders. Empirical results suggest that nested dissection produces better elimination orders if allowed to choose slightly imbalanced but smaller separators. In this section we help substantiate this belief by showing that a 1/2-balanced nested dissection algorithm working on an interval graph can produce orders with up to $\Omega(m \cdot \sqrt{\log n})$ fill, while a 2/3rds-balanced nested dissection always produces an order with linear fill for interval graphs.

Nested dissection on interval (and chordal) graphs has a simple interpretation in terms of the path (tree) representation of the graph. Consider one such representation. For non-trivial graphs every minimal separator is a set of vertices that correspond to the intervals covering some edge in the path representation (skeleton) of the graph. In these terms, each step in nested dissection can be thought of as selecting an edge of the skeleton. The corresponding separator is formed by the intervals that cover that edge and have not been included in previous separators. Some representative intervals might span a single node of the skeleton, and thus will not be selected in this process. Any such singletons are selected last, as leaves of the elimination tree. We cannot guarantee the existence of an $\alpha$-balanced separator that covers an edge of the skeleton, due to the existence of these intervals that only span a single node of the skeleton. By ordering these single node intervals beforehand, as first to be eliminated, and not changing the tree representation of the graph we can restrict ourselves to separators that cover an edge of the skeleton of the representation of the graph. These intervals must correspond to singletons in the graph, and their elimination causes no fill.

Throughout this section, we order separator trees of interval graphs so that an in-order traversal of the tree corresponds to a left-to-right traversal of the skeleton path of the graph. When necessary we will refer to an *ordered* separator tree to make it clear that we are considering a separator tree whose children are ordered as described here. In the lemmas that follow, we only consider non-trivial interval graphs, that is, we assume that the graphs in question are not complete.

### 4.1.1 Balanced nested dissection

We proceed to derive an upper bound on the total number of fill edges introduced by an $\alpha$-balanced nested dissection algorithm on an interval graph. We show this bound is tight for $\alpha = \frac{1}{2}$, but not for other values of $\alpha$.

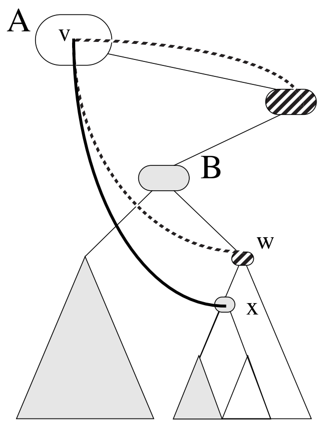



Figure 4.1: Fill among vertices in a separator tree of an interval graph.

Figure 4.1 shows part of a separator tree of an interval graph $I$. Each node in the tree corresponds to a clique separator of the subgraph induced by the subtree rooted at that node. The shaded subtrees correspond to vertices that are neighbors in $I$ of a given vertex $v$ contained in the node $A$, while the striped nodes contain vertices that have fill edges to $v$ in the chordal completion of $I$. The striped edge between $v$ and $w$ represents one such fill edge. A vertex has fill edges to $v$ if at least one of its descendants in the separator tree, say $x$, is adjacent to $v$, as represented by the solid edge between $v$ and $x$. Moreover, when that is the case, all vertices between $v$ and $x$ in the in-order traversal of the separator tree must also be adjacent to $v$ in $I$. When the separator tree is balanced, we can use the edges between these vertices and $v$ to account for the fill to $v$.

We define an *inner-path* of a node $A$ in an ordered binary tree as the path that starts with the edge to the left or right child of $A$, and goes all the way to the in-order predecessor or successor of $A$, respectively. We say that a fill edge between a vertex in $A$ and a vertex in a node in $A$'s inner-path is an *inner fill* edge. A fill edge between a vertex in $A$ and a vertex that is a descendant of $A$ in the elimination tree but that is not in $A$'s inner-path is called an *outer fill* edge.

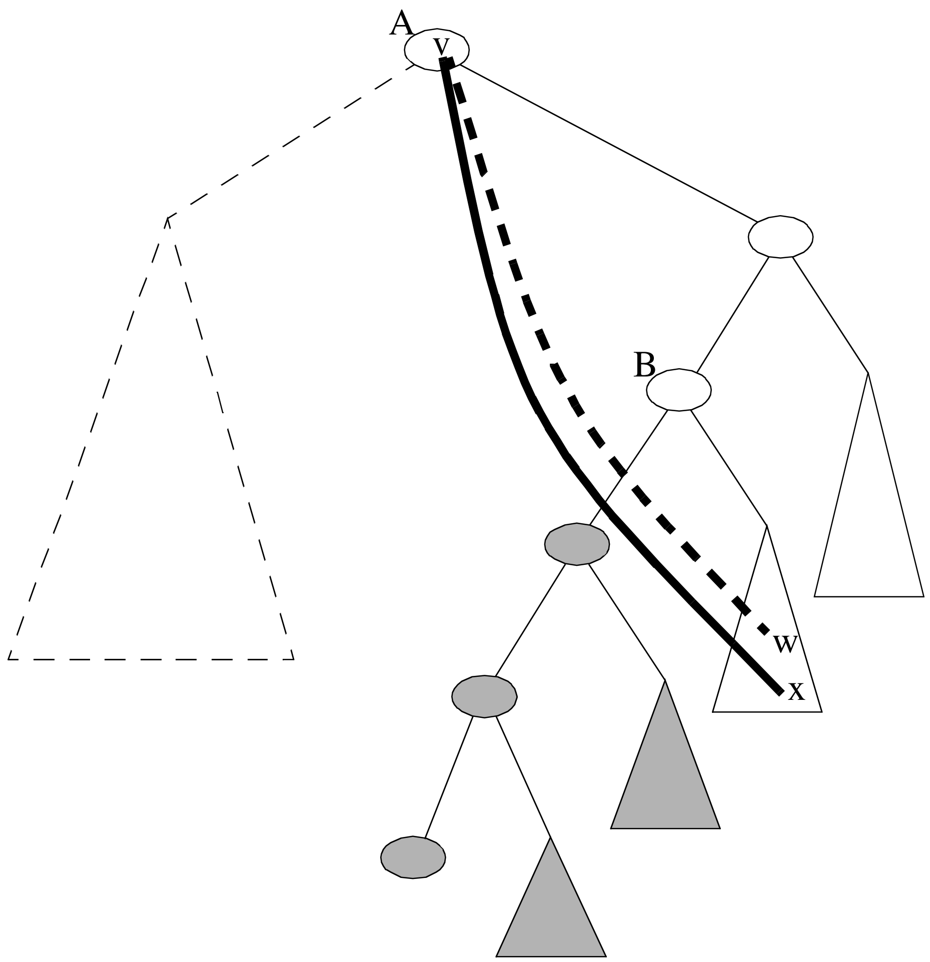



Figure 4.2: Right inner path of a node.

Figure 4.2 depicts a node $A$ and its right inner-path. A vertex $v$ in $A$ has an outer fill edge to vertex $w$. That implies the existence of a node $x$, also not in $A$'s inner-path, that is adjacent to $v$ and is a descendant of $w$ in the elimination tree. The existence of the edge between $v$ and $x$ implies that all vertices in the shaded area in Figure 4.2 are also adjacent to $v$. We can use this fact to prove the next lemma that states that the total amount of outer fill introduced by orders for interval graphs, given by an ordered balanced separator tree is $O(m)$.

**Lemma 1** *Let $I$ be a connected interval graph. The total amount of outer fill in an ordered $\alpha$-balanced separator tree of $I$ is at most $\alpha m/(1-\alpha)$.*

**Proof.** The lemma is proved via an amortized analysis of the total amount of outer fill introduced by the elimination order, in which we use existing edges to help account for all outer fill. We count each fill edge at its highest endpoint. Let $v$ be a vertex of $I$ and let $A$ be the separator node containing $v$. Let's examine the right subtree of $A$. The left one is analogous. Let $B$ be a node on $A$'s right inner-path, whose right subtree contains vertices that have fill to $v$. Then the vertices in $B$ and in $B$'s left subtree are all adjacent to $v$ in $I$, and no ancestor of $B$ has vertices in its right subtree with fill to $v$, for otherwise all vertices in $B$'s right subtree would have edges to $v$ to start with. Since the separator tree is $\alpha$-balanced, $v$ is adjacent to at least a fraction $1-\alpha$ of the vertices in the subtree rooted at $B$, and thus $d(v) \geq (1-\alpha)\cdot n'$, where $n'$ is the number of vertices in the subtree rooted at $B$. The total amount of outer fill from $v$ to its right subtree is at most $\alpha \cdot n'$ which is less than or equal to $\alpha \cdot d(v)/(1-\alpha)$. By applying this same argument to every vertex in the graph we can account for all outer fill and obtain a total of $\alpha \cdot m/(1-\alpha)$ outer fill edges. ■

Lemma 1 allows us to concentrate on inner fill. Now, consider one inner-path.

**Lemma 2** *Let $I$ be a connected interval graph, and let $V_0$ be a node in an ordered separator tree of $I$. Let $V_1, V_2, \ldots, V_k$ be the nodes in $V_0$'s right inner-path. The total amount of inner fill from $V_0$ to the vertices in its right inner-path is at most $O(\sqrt{k} \cdot \sum_{i=0}^{k} |V_i|^2)$.*

**Proof.** Let $n_i = |V_i|$. The total amount of inner fill from $V_0$ to vertices in its right inner-path is at most $n_0 \cdot (n_1 + \ldots + n_k)$. Let $n_1 + n_2 + \ldots + n_k = d$. We want to bound the amount of fill as a function of the sum of the squares of the $n_i$'s, which is on the order of the number of edges to $\cup V_i$, since each node $V_i$ forms a clique and thus has $n_i(n_i-1)/2$ edges. The case $n_i = 1$ is dealt with by noting that the graph is connected, so that each vertex has at least one edge incident to it. We are looking for the smallest number $x$ satisfying $n_0 \cdot d \leq x \cdot (\sum_{i=0}^{k} n_i^2)$. But $\sum_{i=1}^{k} n_i^2 \geq k(d/k)^2$. Let's define $f(x) = x \cdot n_0^2 - d \cdot n_0 + x \cdot d^2/k \geq 0$. For positive $x$, $f(x)$ is positive for large enough $n_0$. The function $f(x)$ is non-negative if and only if the second degree equation on $n_0$ has at most one real root. Thus $x$ must satisfy $d^2 - 4x^2 d^2/k \leq 0$, i.e., $x \geq \sqrt{k}/2$. Therefore the total amount of fill is at most $\frac{\sqrt{k}\cdot\sum_{i=0}^{k} n_i^2}{2}$. ■

A given separator is in at most 4 inner-paths: two starting at itself, one starting at its parent, and possibly another starting at some other ancestor. Since a balanced separator tree has $O(\log n)$ depth, Lemmas 1 and 2 give the following corollary:

**Corollary 1** *Let $I$ be a connected interval graph, with a balanced separator tree of clique separators. The total amount of fill induced by the order specified by the tree is $O(m \cdot \sqrt{\log n})$.*

**Proof.** According to Lemma 1, the total amount of fill is at most $O(m)$ plus the amount of fill in inner-paths. Apply Lemma 2 to all inner-paths. Each separator is the root of at most one left and one right inner-path and is in at most two other inner-paths. Since the tree is balanced, the largest inner-path has $O(\log n)$ length. All that remains to be shown is that the sum of the squares of the sizes of the separators in the tree is $O(m)$. But each separator is a clique, and each vertex occurs in only one separator. Thus, a separator with $n_i$ vertices has $n_i(n_i - 1)/2$ edges in $G$ that do not appear in any other separator. For $n_i > 1$, $n_i^2 \leq 2(n_i(n_i - 1))$. To handle the case $n_i = 1$ we note that the graph is connected and thus every vertex must be adjacent to at least one other vertex in the graph. The corresponding edge can be used to account for fill and we have the desired result. ■

### 4.1.2 Strictly balanced nested dissection

In this section we show that there exist interval graphs on which the nested dissection algorithm that chooses minimal 1/2-balanced separators produces an order with $\Omega(m \cdot \sqrt{\log n})$ fill, thus matching our upper bound.

We will build an example by constructing the appropriate separator tree. We start with a right inner-path and construct the example as follows: we insert vertices into each of the nodes of the inner-path so as to form cliques. We insert $n_0$ vertices to form the top clique, and $s$ vertices to form each of the cliques in the remaining nodes. Finally, we finish off the graph by inserting paths, each corresponding to a balanced subtree of the separator tree being constructed, in a bottom-up fashion, as depicted in Figure 4.3. We insert a single vertex between the root node's clique and the node that follows it in the in-order traversal of the inner path. We also attach a right subtree to each node in the inner-path. Each subtree corresponds to a 1/2-balanced nested dissection of a path with as many nodes as needed so as to make the subtree rooted at its parent node balanced. We insert a left subtree so as to also make the top level separator a 1/2-balanced separator. We could either use a mirrored version of the right subtree, or use a path with as many vertices as needed. The endpoints of the inserted paths are made adjacent to the vertices in each of the pre-existing separator nodes that "neighbor" it in this separator tree.

For $s > 1$, a 1/2-balanced nested dissection order that uses minimal separators must be such that its separator tree has the inner-path we started with as an inner-path, for those are the only choices for $1/2$-balanced minimal separators.

A graph built in this fashion, with $k + 1$ nodes in the initial inner-path has $n_0 + 2 \cdot N(k)$ vertices total, where $N(i)$ is the number of vertices in a subtree of height $i$, given by $N(0) = 1$; $N(i) = 2 \cdot N(i-1) + s$, $i \leq k$. If we make $k = \log_2 n_0$ then the total number of vertices $n$ in the graph is $O(n_0 \cdot s)$. The vertices in each of the paths inserted have at most 2 neighbors, except for possibly the first and last vertices of the path, which might have clique separators as their neighbors. But each of the vertices within each clique has at most two neighbors in paths, so that the total number of edges from these paths is $O(n)$. The cliques along the root's inner path have a total of $O(n_0^2 + k \cdot s^2)$ edges, so the graph has a total of $O(n_0^2 + n_0 \cdot s + k \cdot s^2)$ edges. An elimination

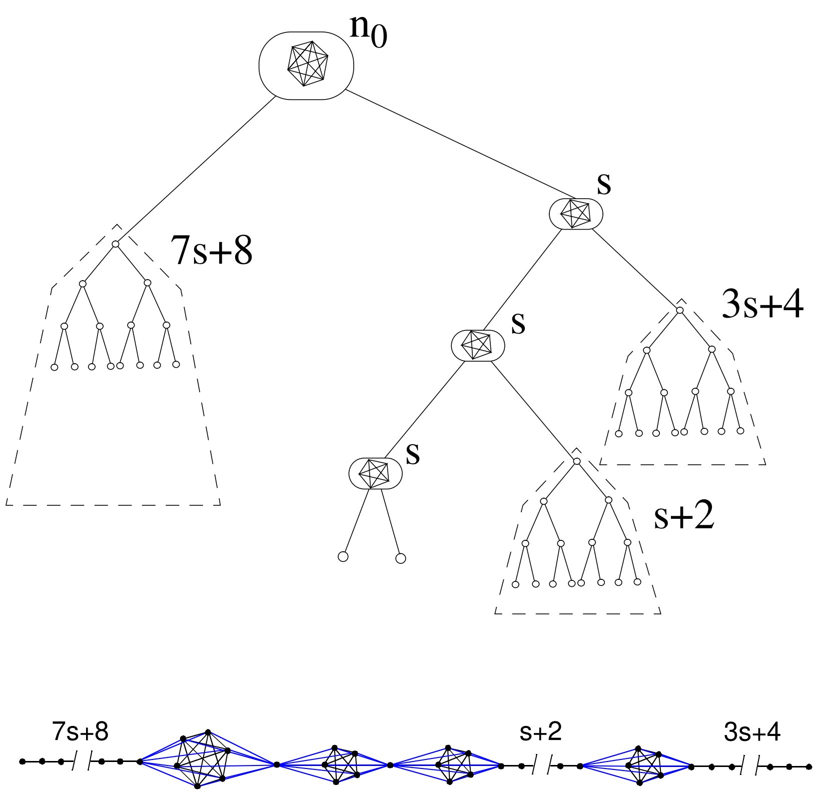



Figure 4.3: Separator tree and our example graph for $k = 3$.

order that removes the vertices of the root's inner path in the order prescribed by the separator tree will introduce edges from all vertices in the nodes along the root's inner path to the root, giving a total of $\Omega(n_0 \cdot s \cdot \log_2 n_0)$ fill. If we choose $s = n_0/\sqrt{\log_2 n_0}$, the total number of edges is $O(n_0^2)$, while the amount of fill is $\Omega(n_0^2 \cdot \sqrt{\log_2 n_0})$.

### 4.1.3 Adding a little freedom

We can change nested dissection to obtain an algorithm that produces orders with fill linear in the number of edges in the initial interval graph. Instead of insisting on 1/2-balanced separators, we can choose the smallest (with the minimum number of vertices) 2/3rds-balanced minimal separator, and recurse. Unlike the case for 1/2-balanced nested dissection, there no longer exists an interval graph for which this algorithm will incur more than linear fill. A separator will only be chosen as the root of the separator tree if at least a third of the vertices in the graph are adjacent to at least as many vertices as are present in the separator. That is to say, a large top level separator implies that the graph must have many edges and thus can "support" considerable fill. As our analysis shows, this is enough to ensure linear fill.

**Lemma 3** *Let $I$ be a connected interval graph with $n$ vertices, and let $p_0$ be the number of vertices in its smallest 2/3rds-balanced minimal separator. Then $I$ has at least $n/3$ vertices of degree $p_0$.*

**Proof.** The lemma is trivial if $I$ is a complete graph. Otherwise a skeleton path for $I$ contains at least one edge, and every minimal separator of $I$ corresponds to a set of intervals that cover some edge of its skeleton path. Moreover, given two distinct skeleton edges corresponding to one-third-two-thirds separators, every edge of the skeleton between those two edges also corresponds to one-third-two-thirds separators. If we scan the edges from left to right, there is a leftmost edge $l$ and a rightmost edge $r$, such that all edges of the skeleton between and including $l$ and $r$ correspond to one-third-two-thirds separators. There are less than $n/3$ vertices whose representative intervals are entirely contained to the left of $l$. Otherwise the edge to the left of $l$ would also correspond to a one-third-two-thirds separator. Analogously there are less than $n/3$ vertices whose intervals are entirely contained to the right of $r$, so that at least $n/3$ vertices correspond to intervals that cover at least one node in the path between and including $l$ and $r$. Every edge in that path has ply at least $p_0$, that is, is covered by at least $p_0$ intervals, thus concluding this proof. ■

The next lemma allows us to distribute enough credits to each vertex $v$ of $I$ to pay for the inner-fill from $v$ to all its ancestors. Let $l(v)$ be the level of a vertex $v$ in $T$.

**Lemma 4** *Let $I$ be a connected interval graph , and let $T$ be a separator tree obtained by applying a one-third-two-thirds nested dissection algorithm to $I$. Let $V_0, \ldots, V_{l(v)}$ be the nodes in the path from the root of $T$ to $V_{l(v)}$, the node of $T$ that contains $v$. Moreover, let $p_0, \ldots, p_{l(v)}$ be the number of vertices in $V_0, \ldots, V_{l(v)}$. Then we can assign $\mathcal{P}(v) = \max_{i \leq l(v)}(p_i)$ credits to each vertex $v$ in $I$ such that $\sum_{v \in V} \mathcal{P}(v)$ is $O(m)$.*

**Proof.** We present an amortized analysis. In our analysis, we distribute $O(1)$ credits per edge, i.e., each vertex $v$ in the graph initially has $O(d(v))$ credits to "spend", so that the total amount of credits distributed is $O(m)$. We will redistribute these credits.

We use Lemma 3 recursively. At the top level there are at least $n/3$ vertices with degree at least $p_0$, enough to distribute $p_0/3$ credits to each vertex in $I$. Since we want a total of $O(m)$ credits, we can actually distribute 3 credits per edge, and thus $p_0$ credits to each vertex.

The same vertices whose edges were used to help distribute credits at a given level might be used in again in the analysis. Assume by induction that down to level $l - 1$ we can distribute $\max_{i \leq (l-1)}(p_i)$ to every vertex in the subtree rooted at $V_l$, while using at most $3 \cdot \max_{i \leq (l-1)}(p_i)$ credits from any single vertex in that subtree.

Let $n_l$ be the number of nodes in the subgraph induced by $V_l$'s subtree. Applying Lemma 3 to this subgraph we find that at least $n_l/3$ vertices in that subgraph have degree at least $p_l$. By induction, we have used at most $3 \cdot \max_{i \leq (l-1)}(p_i)$ credits from any vertex in that subgraph. Thus, we can use an additional $3 \cdot (p_l - \max_{i \leq (l-1)}(p_i))$ credits from the $n_l/3$ vertices of degree at least $p_l$, which is enough to distribute a total of $\max_{i \leq (l)}(p_i)$ to each vertex in the subtree rooted at $V_l$. If $p_l - \max_{i \leq (l-1)}(p_i) \leq 0$ then the vertices already had enough credits and we do not need to distribute any additional credits. ■

**Lemma 5** *The one-third-two-thirds balanced nested dissection algorithm produces an order that has fill linear in the number of edges of the input interval graph $I$.*

**Proof.** Since the algorithm chooses balanced separators at each level, Lemma 1 allows us to only consider fill within the inner-paths of $I$'s separator tree. Every node $V_i$ in the separator tree can be the root of at most two inner-paths, and otherwise appears in at most two other inner-paths, one rooted at its parent, and one rooted at another of its ancestors. Let $p$ be the number of vertices in the largest of these two ancestors of $V_i$. Assigning $O(m)$ total credits as described in Lemma 4, each vertex in $V_i$ has potential at least $p$, and can thus "pay" for the at most $2 \cdot p$ upward fill to the two nodes on whose inner-paths $V_i$ lies. This accounts for all upward inner-fill, since an inner-fill edge must connect a vertex with a vertex in one of the nodes that is its descendant in an inner-path. All fill that remains unaccounted for is outer-fill, and by Lemma 1 a 2/3rds-balanced nested dissection has at most $O(m)$ outer-fill. ■

An even simpler proof that does not use Lemma 1 can be obtained for the nested dissection algorithm that includes the separator vertices in the recursive subgraphs, knowing, however, that those vertices have already been ordered last. The proofs are identical, except that, since the separators are included in the recursive calls, all the upward fill is to the vertices in whose inner-path the vertex lies, i.e., there is no outer fill. In this separator tree a vertex might appear multiple times, but is ordered at its highest occurrence in the tree, that is, in the node closest to the root in which it appears. Lemma 4 can be proved for this tree, which allows us to conclude the proof.

## 4.2 Parallel elimination orders for chordal graphs.

We proceed to show that although the one-third-two-thirds nested dissection we defined in the previous section produces orders with linear fill for interval graphs it can produce orders with super-linear fill for chordal graphs. We also show an algorithm that produces parallel orders that do achieve linear fill on chordal graphs, but are less parallel than nested dissection orders.

### 4.2.1 Nested dissection.

Even though the one-third-two-thirds nested dissection is guaranteed to produce orders with linear fill for interval graphs, it may create more than linear fill on chordal graphs. We proceed to show an example that demonstrates this fact.

Just as we did for the example in Section 4.1.2, we build a graph based on the desired separator tree. We start with what will end up being the inner-path of the root node of the nested dissection separator tree. The root separator is made of size $n_0$, and all other nodes in the initial inner-path are made of size $s$. Unlike what we did in Section 4.1.2, we do not build a binary tree. We make every node except the root have 3 children in the separator tree. Again, from bottom up, we add a path (whose representation in terms of a separator tree can be a balanced binary tree) with as many vertices as needed so as to make the subtrees have the same total number of vertices, just as we did in Section 4.1.2; but we also add a second path, with as many vertices as the first one, and make only one of the endpoints of the path adjacent to the vertices in the node (from the original inner-path) being considered. This extra path makes sure the node is the only choice for a one-third-two-thirds minimal separator. We then attach to the root node a number of copies, say $l$, of the graph we just attached to the root node.

The total number of vertices in this graph is bounded by $n = 4^k \cdot s \cdot l + n_0$, where $k$ is the height of the node path we started with. The number of edges in this graph is on the order of the number of edges within each separator node, plus $O(n)$, that is, $O(k \cdot s^2 \cdot l + n_0^2 + n)$. On the other hand we have at least $\Omega(l \cdot s \cdot k \cdot n_0)$ fill. For $k = (\log_4 n_0)/2$, $s = \sqrt{n_0}/\sqrt{\log_4 n_0}$, and $l = n_0$, we get a total of no more than $n = n_0^2/\sqrt{\log_4 n_0} + n_0$ vertices and $O(n_0^2)$ edges in the graph. The total amount of fill is $\Omega(n_0^2 \cdot \sqrt{n_0 \cdot \log_4 n_0}/2)$, a factor of $\Omega(\sqrt{n_0 \cdot \log_4 n_0})$ times the number of edges in a perfect elimination order. Note that $n_0$ is not $\Omega(n)$, but rather roughly $\Omega(d)$, where $d$ is the maximum degree of the graph, since $n_0$ is about the degree of the vertices in the root node.

### 4.2.2 A (less) parallel order with linear fill

In this section we show how to find an elimination order for a chordal graph $G$ by repeatedly applying an interval graph algorithm to branches of $G$'s skeleton. The resulting order has linear fill, but has an extra $O(\log n)$ factor in height when compared to the height of an order that a nested dissection algorithm would produce, bringing us to a factor of $O(\log^2 n)$ off the optimal height.

To more easily describe our algorithm, we need additional definitions. An edge of a skeleton tree $T$ is said to be an extremity of $T$ if one of its endpoints is a leaf. A path is said to be a terminal branch of $T$ if it is a maximal path containing a leaf of $T$, and all its internal vertices have degree 2 in $T$. Given a tree representation of a chordal graph $G$, with skeleton $T$ and representative subtrees $T_i$, we denote $P(l, r)$ the interval graph obtained by restricting $G$ to the path $P_{l,r}$ between the edges $l$ and $r$ (inclusive) of $T$; $P(l, r)$ is the interval graph whose representation consists of $P_{l,r}$ and representative subtrees $\{T_i \mid T_i \cap P_{l,r} \neq \emptyset\}$.

Let $G$ be a chordal graph, and let $T$ be the skeleton of a representation of $G$. Vertices that appear in a terminal branch, but also appear outside the branch are not to be ordered within that branch. We shall refer to these vertices as *depleted* in the terminal branch, meaning edges between vertices whose representative intervals lie entirely within a terminal branch to depleted vertices can help us account for fill within the terminal branch, but edges between pairs of depleted vertices cannot, for those edges can also be present in a number of other interval subgraphs that we will consider. The vertices of $G$ whose representative subtrees are entirely contained in some terminal branch of $T$ can be eliminated independently and in parallel with the vertices entirely contained in other terminal branches.

We say a vertex of $G$ is *pinned* at an edge of $T$ if its representative subtree covers that edge. Let $A$ be an ordering algorithm for interval graphs. Let $e$ and $f$ be two edges of the skeleton of an interval graph. Let $K_l(e, f)$ denote the graph obtained from $P(e, f)$ by removing all vertices pinned at $l$ as well as those that cover both skeleton edges $e$ and $f$. Let $r'$ be an extremity of $T$, and let $l$ be the other extremity of the terminal branch of $T$ that contains $r'$. To make it easier to describe our algorithm, imagine an artificial edge $r$ with zero ply, so that $r$ is the extremity of the terminal branch, instead of $r'$. The algorithm works by pruning all terminal branches of $T$ and then recursing on what is left of the skeleton tree $T$, until only a path is left for a skeleton. To the interval graph corresponding to the last path, we can apply algorithm $A$ directly. Each of the terminal branches are ordered using the algorithm that follows. All branches obtained in a given pruning step are processed in parallel, the vertices in the corresponding subgraphs being ordered before those in later pruning steps. A total of $O(\log n)$ pruning steps are needed to order the whole graph.

We apply the following algorithm to each terminal branch:

Kill$(T, l, r)$

> Mark the edge $r$. Traverse $P_{l,r}$ from $r$ to $l$ and mark the first edge of $P_{l,r}$ that is covered by an interval pinned at $l$. Keep scanning $P_{l,r}$ towards $l$, and marking the next edge that is covered by at least twice as many intervals pinned at $l$ as did the last marked edge. Call these *milestone* edges. Mark the edges adjacent to the milestones, which are closer to $r$ than the corresponding milestone, and call those *sentinels*. Also mark $l$.
> Let $k+1$ be the number of edges marked, and let $e_i$, $i \leq k$ be the $i$-th edge closest to $l$ that was selected, i.e., $e_0 = l$ and $e_k = r$. Remove the vertices pinned at $l$ from $P(l,r)$.
> For $i$ from 1 to k, order the vertices pinned at $e_i$ last among the remaining, unordered vertices of $P(l,r)$, and remove them from $P(l,r)$.
> Apply Homogenize to each of the subgraphs $K_l(e_i, e_{i+1})$.

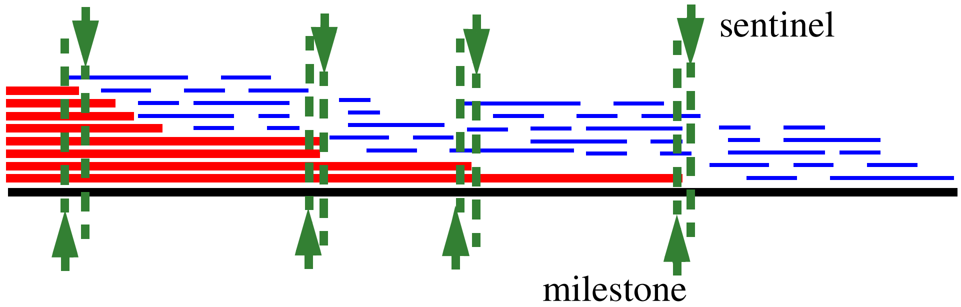



Figure 4.4: The Kill procedure orders separators that we call sentinels and milestones last. The subgraphs formed are then passed to the Homogenize procedure.

Figure 4.4 depicts the choice of sentinels and milestones on an interval graph. The intervals that cover the leftmost edge of the skeleton are depleted. The sentinels play an important role in the ordering algorithm. Along with the milestones, they impose a sequential step in the midst of the parallel order and ultimately allow us to control the amount of fill introduced. Because the vertices pinned at $e_i$ and $e_{i+1}$ are ordered by the kill procedure, we cannot directly apply a regular interval graph algorithm to the subgraphs $K_l(e_i, e_{i+1})$ and expect to obtain low fill. Instead, we apply the following algorithm to control the amount of fill that can be created to vertices whose intervals cover either $e_i$ or $e_{e+1}$. The algorithm we describe next divides an interval graph into somewhat homogeneous subgraphs, for the vertices in each of them have some minimum degree guarantee, namely every vertex in a subgraph obtained by the homogenize procedure has degree greater than half of the number of intervals pinned at either the left or the rightmost edge of the skeleton path that defines the subgraph.

Homogenize$(T, l', r')$

> Mark $l'$, and transverse $P_{l',r'}$ towards $r'$, marking the first edge whose ply is at most half the ply of the last marked edge. Repeat until $r'$ is reached. Apply the same algorithm from $r'$ to $l'$. Let $f_i$, $0 \leq i \leq k+1$, be the edges that were marked ordered from $l'$ to $r'$, including $l' = f_0$ and $r' = f_{k+1}$. Number the as of yet unnumbered vertices pinned at the various marked edges, in an arbitrary order, last among the vertices of $P_{l',r'}$. Apply the interval graph algorithm (A) to each of the subgraphs obtained by removing any vertices pinned at $f_i$ or $f_{i+1}$ from $P(f_i, f_{i+1})$.

Let $I$ be an interval graph with $n$ vertices and $m$ edges. Let's assume that when the interval graph algorithm $A$ is applied to $I$ it produces an order with height $O(h(n) \cdot H(I))$, fill $O(f(n) \cdot m)$, and work $O(w(n) \cdot W(I))$, for non-decreasing functions $f(n)$, $h(n)$ and $w(n)$, each greater than 1, where $H(I)$, $m$ and $W(I)$ correspond to the height of $I$, the number of edges in $I$, and the minimum amount of work to eliminate $I$, respectively. We can then prove the following lemmas about the chordal graph algorithm we have just described:

**Lemma 6** *Our chordal graph algorithm produces an order with height* $O(\log n \cdot (h(n) + \log n) \cdot H(G))$.

**Proof.** Only $\log n$ pruning steps are necessary to divide the entire graph into various interval graphs each of which are passed to the *kill* algorithm. That is so because by pruning all the existing leaves and corresponding terminal branches, the number of vertices of $T$ with degree not equal to two goes down by at least a factor of two.

The *kill* procedure marks at most $O(\log n)$ edges of a terminal branch of the skeleton, and orders the cliques covering each edge sequentially. Since the size of any existing clique is a lower bound on the height of any elimination order for a given graph, this part of the order has height $O(\log n \cdot H(G))$. It then orders the subgraphs in parallel, using Homogenize.

Each call to Homogenize marks up to $2 \cdot \log n$ edges of a skeleton path $P$. The total number of vertices pinned at the various edges is at most 4 times the ply at the endpoint of $P$ with largest ply, for the edges are chosen so that they have plies that decrease geometrically, by a factor of 2 at each time. The vertices pinned at the largest of the two endpoints of $P$ form a clique, and thus any elimination order for $G$ must have height at least as large as the number of such vertices. Thus Homogenize adds at most a constant times $H(G)$ to the height of the order.

Finally, the interval graph algorithm produces orders with height $O(h(n) \cdot H(G))$.

Each terminal branch is ordered with height $O((4 + h(n) + \log n) \cdot H(G))$, and since there are $\log n$ pruning steps, we get the desired bound. ■

**Lemma 7** *When applied to an interval graph $I$, with $n$ vertices and $m$ edges, and using the interval graph algorithm $A$, the Homogenize procedure produces an order with $O(f(n) \cdot m)$ fill.*

**Proof.** There is no fill between vertices in the different subgraphs to which $A$ is applied, regardless of what order the procedure $A$ produces, since there are separators between the various subgraphs. The total amount of fill within all those subgraphs is $O(f(n) \cdot m)$, since $f()$ is non-decreasing and any given vertex of $I$ can appear in only one subgraph.

Let $a$ and $b$ be the number of vertices pinned at each of the endpoints of the skeleton of $I$ used in the algorithm. Then at most $2 \cdot (a+b)$ vertices are selected by the Homogenize procedure. The fill between these vertices is at most $2 \cdot (a+b)^2$, which is within a constant factor of $(a(a-1)+b(b-1))/2$, the number of edges in the cliques corresponding to the two endpoints, when either $a$ or $b$ is larger than 1. Since the graph $I$ is connected, the case in which $a$ and $b$ are one can be handled by noting that every vertex has at least one edge since the graph is connected and the existing edge can be used to pay for a constant amount of fill. The only other source of fill is between the vertices in the various subgraphs ordered using $A$, and those vertices that were selected. But there is only fill to the vertices pinned at the two edges that delimit the skeleton path for a given subgraph. The homogenize algorithm ensures that the ply at every edge within that path, and thus the degree of the nodes in each subgraph, is at least half the number of vertices pinned at either extremity. Thus, every vertex in the subgraph has enough edges to pay for its fill to the vertices pinned at the two extremities of the skeleton path. Thus, the fill introduced is linear in the number of edges of $I$. ∎

Let $G$ be a chordal graph, with $n$ vertices and $m$ edges:

**Lemma 8** *Our chordal graph algorithm applied to G produces an order with $O(f(n) \cdot m)$ fill.*

**Proof.** There is no fill between vertices in the different $K_l(e_i, e_{i+1})$ subgraphs, since the vertices pinned at the various selected edges form separators between the subgraphs, and the vertices in the separators are eliminated after the vertices within each of the subgraphs.

Any vertices removed from $K_l(e_i, e_{i+1})$ because they were pinned at both $e_i$ and $e_{i+1}$ are already adjacent to all vertices of $K_l(e_i, e_{i+1})$, and thus have no fill to vertices in $K_l(e_i, e_{i+1})$.

If $e_i$ and $e_{i+1}$ are a milestone and its sentinel then they are already adjacent in $P_{l,r}$, and no fill is created between vertices pinned at those edges, for they were also adjacent to start with.

Let $L$ be the set of vertices of $P(l,r)$ pinned at $l$. The sources of fill are:

1. Fill within each of the subgraphs that were created;
2. Fill between vertices pinned at edges $e_i$ and $e_{i+1}$, not including those vertices in $L$;
3. Fill between vertices of $K_l(l', r')$ and $L$.

Fill of types (1) and (2) has been accounted for in Lemma 7.

We need to account for fill to vertices in $L$ (type 3). We accomplish this by using already existing edges from each vertex to a subset of $L$. The order imposed on the vertices pinned at the edges $e_i$ ensures that any vertex that has fill to a subset of the vertices in $L$ is adjacent in $G$ to at least half as many vertices in $L$, so that this component of the fill is linear in the number of edges from $K_l(l, r)$ to $L$.

No edge is reused in the accounting, so that there is no over-counting when we apply the same argument to the various terminal branches, in this and future pruning steps. Therefore, the total amount of fill is $O(f(n) \cdot m)$, proving the lemma. ■

If we combine the results from Section 4.1.3 on the one-third-two-thirds nested dissection algorithm with the results in this section, we get:

**Corollary 2** *Our chordal graph algorithm, using a one-third-two-thirds nested dissection algorithm on interval graphs, produces orders with $O((\log n)^2 \cdot H(G))$ height, and $O(m)$ fill.*

### 4.2.3 Work analysis

In this section we provide an analysis of the work to eliminate a graph according to an order produced by the linear fill algorithms we have presented so far. We show that the one-third-two-thirds nested dissection algorithm on interval graphs and our chordal graph algorithm both produce orders that are within a constant factor of optimal in terms of work. We proceed to prove these results.

We "charge" $d + 1 + d^2$ to eliminate a single vertex $v$ from a graph $G$, where $d$ is the degree of $v$ in $G$, for this is proportional to the total number of floating-point operations to perform the corresponding matrix operation. We use this charging scheme throughout this section whenever analyzing the work of a given elimination order.

Let $c_v$ be the size of the largest clique of $G$ that contains $v$:

**Lemma 9** *The total work necessary to perform Gaussian elimination on a graph $G$ is at least $W'(G) = \sum_v (\frac{c_v}{2} + \frac{(c_v-1)^2}{3})$.*

**Proof.** The base case, with a single vertex is trivial – it costs us 1 unit of work. Assume the lemma holds for graphs with less than $k$ vertices. Let $v$ be the first vertex to be eliminated in a minimum work elimination order of a graph $G$ with $k$ vertices. The lemma holds for $G - \{v\}$. Let $d$ be the degree of $v$ in $G$. For each of the neighbors of $v$, either the largest clique it is in has the same size in both $G - \{v\}$ and in $G$, or it decreases by 1 when $v$ is removed, in which case $v$ must be part of that clique. Let $w$ be a neighbor of $v$ in $G$ such that the largest clique that contains $w$ in $G - \{v\}$ has size $x$, while the largest clique containing $w$ in $G$ has size $x + 1$. Then $w$ contributes $1/2 + (x^2 - (x-1)^2)/3 = 1/2 + (2 \cdot x - 1)/3$ more to $W'(G)$ than to $W'(G - \{v\})$. But $x \leq d$ so that the contributions of all neighbors of $v$ to $W'(G)$ minus their contributions to $W'(G - \{v\})$ add up to at most $d/2 + d \cdot (2 \cdot d - 1)/3$. The contribution of the vertex $v$ to W'(G) is at most $(d+1)/2 + d^2/3$, so that $W'(G) - W'(G') \leq d + 1 + d^2$.

By induction, the amount of work necessary to perform Gaussian elimination in $G$ is at least $W'(G')$ plus $d + 1 + d^2$, the amount of work to eliminate $v$, and thus at least $W'(G)$. ■

The next lemma is just a simple observation that allows us to account for work in a constant number of parcels. It says that as long as we have at most a constant number $c$ of parts that can be accounted for separately, then all of them taken together can also be accounted for, by increasing the constants involved in the $O()$ notation. Let $\omega'(x) = x + 1 + x^2$.

**Lemma 10** *Let $G$ be a graph, and $v$ be a vertex of $G$. If $v$ has neighbors in at most $c$ sets of vertices $V_1$, $V_2$, ..., $V_c$ when $v$ is eliminated from $G$, then the work involved in eliminating $v$ from $G$ is at most $w'(x)$ where $x = |V_1 \cup V_2 \cup \ldots \cup V_c|$ which is $O(\omega'(|V_1|) + \omega'(|V_2|) + \ldots + \omega'(|V_c|))$.*

**Proof.** Follows from the facts that $|V_1 \cup V_2 \cup \ldots \cup V_c|$ is at most c times the maximum of $|V_1|$, $|V_2|$, ..., $|V_c|$ and $\omega'(cx) \leq c^2 \cdot \omega'(x)$ for $x \geq 0$. ■

What follows is almost identical to Lemma 7 and its proof. We account for the amount of work created by an order by usually showing that the amount of work to eliminate vertices within a subgraph is within a certain bound, and then showing that the vertices of the graph have degree, at the time the vertices are eliminated, which consists of the degree they had within that subgraph at the time they were eliminated plus either edges that existed in the original graph or fill edges. We show that the work associated with the existence of these edges is within a factor of amount of work associated with cliques in the original graph (Lemma 9), and then use Lemma 10 to add the various contributions to estimate the total amount of work for the given order.

Let $A$ be an interval graph algorithm that produces orders with $O(w(n) \cdot W(I))$ work for any interval graph $I$.

**Lemma 11** *When applied to an interval graph $I$, with $n$ vertices, and using the interval graph algorithm $A$, the Homogenize procedure produces an order with $O(w(n) \cdot W(I))$ work.*

**Proof.** We follow the steps of the proof of Lemma 7, and account for the work to eliminate each vertex. The amount of work to eliminate a vertex is $d^2 + d + 1$, where $d$ is the total number neighbors the vertex has at the time it is eliminated. Thus, we need to examine each vertex, and find the number of fill edges it has at the time it is eliminated.

Let $H$ be one of the subgraphs to which algorithm $A$ is applied, and let $n'$ be the number of vertices in $H$. The elimination of $H$ according to the order produced by $A$ requires at most $O(w(n') \cdot W(H))$ work. Consider fill edges between $H$ and vertices of $I$ not in $H$. The vertices of $H$ can only have fill edges to the vertices of $I \setminus H$ that are pinned at the two edges that delimit the skeleton path of $H$. By construction, the ply at every edge within $H$'s skeleton path is within a constant of $p$, the number of vertices pinned at the endpoints of the skeleton. Therefore, given any vertex $v$ in $H$, the number of neighbors of $v$ in $I \setminus H$ is within a constant of the largest clique of $H$ containing $v$. By Lemmas 9 and 10 the extra amount of work involved in the elimination of $H$ and the other subgraphs to which algorithm $A$ is applied as subgraphs of $I$ adds up to $O(w(n) \cdot W(I))$. Since the subgraphs are disjoint, the total amount of work to eliminate the subgraphs is also at most $O(w(n) \cdot W(I))$.

Finally, we need to account for the work to eliminate the vertices pinned at the various selected edges. Let $a$ and $b$ be the number of vertices pinned at each of the endpoints of the skeleton of $I$

used in the algorithm. Then there is a clique of size at least $max(a, b)$ in $I$, and even if we make all of the at most $2 \cdot (a + b)$ vertices that are selected by the Homogenize procedure into a clique, the total amount of work to eliminate these vertices is within $O(W(I))$. ■

**Lemma 12** *Our chordal graph algorithm produces an order with $O(w(n) \cdot W(G))$ work.*

**Proof.** Just as we did in the proof of the previous lemma, we consider the amount of work involved in eliminating each of the $K_l(e_i, e_{i+1})$ subgraphs, and then the work to eliminate the remaining vertices.

By Lemma 11, the work involved in eliminating each $K_l(e_i, e_{i+1})$ using an order produced by applying Homogenize to $A$ is $O(w(n') \cdot W(K_l(e_i, e_{i+1})))$, where $n'$ is the number of vertices in $K_l(e_i, e_{i+1})$.

There is no fill between vertices in the different $K_l(e_i, e_{i+1})$, since the vertices pinned at the various selected edges form separators between the subgraphs, and the vertices in the separators are eliminated after the vertices within each of the subgraphs.

Any vertices removed from $K_l(e_i, e_{i+1})$ because they were pinned at both $e_i$ and $e_{i+1}$ are already adjacent to all vertices of $K_l(e_i, e_{i+1})$ and form a clique. By Lemmas 9 and 10 the existence of these edges adds at most $O(w(n) \cdot W(G))$ more work as a result of the elimination process, when summed over all subgraphs $K_l(e_i, e_{i+1})$ ever created by the algorithm.

Let $L$ be the set of vertices of $P(l, r)$ pinned at $l$. Vertices in $L$ are not ordered at this step of the algorithm, but there are fill edges to those vertices. The Kill algorithm ensures that any vertex that has fill to a subset of the vertices in $L$ is adjacent to at least half as many vertices in $L$. Since $L$ is a clique, Lemmas 9 and 10, and the repeated application of this argument to all terminal branches ever ordered by the algorithm, imply that the total amount of work involved in eliminating $G$ is $O(w(n) \cdot W(G))$. ■

To show that our algorithm produces orders with work within a constant factor of optimum, we need to show that the one-third-two-thirds nested dissection produces orders with linear work.

**Lemma 13** *Let $I$ be a connected interval graph. The one-third-two-thirds nested dissection algorithm applied to $I$ produces an order with $O(W(I))$ work, i.e., w(n) = O(1) for this algorithm.*

**Proof.** To bound the amount of work involved in the elimination order, we need to count the number of edges out of each vertex at the time it is eliminated.

We examine fill by looking at the ordered balanced elimination tree created by the algorithm. Let's examine the fill from each vertex to vertices that lie in its ancestors in the separator tree. Let $v$ be a vertex in a node $V$ at level $k$ of the separator tree. Let $L$ and $R$ be, respectively, the left and right lowest ancestors of $V$ in the elimination tree. Any ancestor of $v$ to which it is adjacent to at the time it is eliminated must cover the edge of the skeleton corresponding to either $L$ or $R$.

Consider the fill from $v$ to vertices that cover $L$. Fill to vertices covering $R$ is analogous. The work caused by edges to vertices that are adjacent to $v$ and cover $L$ (and thus form a clique) adds up to at most $O(W(I))$ over all vertices of $I$, by Lemmas 9 and 10 and the fact that these are edges in $I$. Let $l$ and $l$' be the number of vertices in $L$ and the number of vertices that cover $L$ but are not adjacent to $v$ in $I$ and are not included in $L$, because they are part of higher-level separators, respectively.

All vertices in $L$'s left subtree must have edges (in $I$) to the $l$' vertices that cover $L$ and are in higher-level separators. Moreover, those $l$' vertices must form a clique in $I$. Since the subtree is balanced, this implies that at least a third of the vertices in the subtree rooted at $L$ are in cliques of size at least $l'$. No other subtree will use these cliques to account for work to a left ancestor node (they might also be used to account for a right ancestor) so that again, by Lemmas 9 and 10 we have a total of $O(W(I))$ work.

The last component of the work corresponds to fill to the $l$ vertices in the separator $L$. Again, these form a clique, and we can proceed just as in the proof of Lemma 4.

Let $p_i$ be the number of vertices in a node $V_i$ at level $i$ of the separator tree. At the top level there are at least $n/3$ vertices of $I$ that are adjacent to a clique of size at least $p_0$. At each recursive level let $n'$ be the number of vertices in a given subtree. Then, at least $n'/3$ of these vertices are adjacent to a clique of size at least $p_l$. This adds up to enough potential for each vertex within a subtree to pay for work related to a clique of size $\max_{i \leq l}(p_i)$, while maintaining the sum of the potentials of all vertices in $I$ within $O(W(I))$. Therefore, $v$ can pay for its work related to the $l$ vertices in $L$. By Lemma 10, we conclude the total amount of work induced by this order is indeed $O(W(I))$. ■

## 4.3 Empirical results

We implemented the linear fill and work algorithm described in the previous sections. We present here the results we obtained by applying that algorithm as a post-processing step to orders obtained using different heuristics.

The "bcsstk" matrices used in our experiments come from structural engineering problems, and were obtained from the Harwell-Boeing collection [IDL89], and from Timothy Davis's "University of Florida Sparse Matrix Collection" [Dav94] (the matrices were provided to Davis by Roger Grimes, at Boeing.) The nasasrb matrix models the structure of the NASA Langley shuttle rocket booster, while the "sf" matrices are used in the simulation of an earthquake in the San Fernando Valley [OS96]. The "g" matrices are $h \times w$ grids. The number of vertices and edges in each graph can be found in Table 4.1.

We applied our algorithm as a post-processing step to the orders produced by a version of the approximate minimum-degree heuristic (AMD) [1] [ADD96], to the nested dissection orders produced by METIS-3.0 [2] [KK95], and to the orders produced by the BEND algorithm obtained from Rothberg [HR96, HR97]. Given each order we applied our chordal algorithm to the corresponding chordal completion. The order obtained was then used as an elimination order for the original, non-chordal graph.

[1] code from ftp://ftp.cise.ufl.edu/pub/umfpack/AMD/
[2] code from ftp://ftp.cs.umn.edu/dept/users/kumar/metis/

| Matrix | vertices | edges |
|---|---|---|
| 3dtube | 45330 | 1584144 |
| bcsstk30 | 28924 | 1007284 |
| bcsstk31 | 35588 | 572914 |
| bcsstk32 | 44609 | 985046 |
| bcsstk33 | 8738 | 291583 |
| bcsstk35 | 30237 | 709963 |
| bcsstk36 | 23052 | 560044 |
| bcsstk37 | 25503 | 557737 |
| cfd1 | 70656 | 878854 |
| cfd2 | 123440 | 1482229 |
| g256x256 | 65536 | 130560 |
| g64x1024 | 65536 | 129984 |
| gearbox | 153746 | 4463329 |
| hex256 | 393216 | 2359296 |
| hex64 | 24576 | 147456 |
| hsct16k | 16152 | 376432 |
| nasasrb | 54870 | 1311227 |
| pwt | 36519 | 144794 |
| sf10 | 7294 | 44922 |
| sf5 | 30169 | 190377 |
| shuttle-eddy | 10429 | 46585 |
| struct3 | 53570 | 560062 |

Table 4.1: Number of vertices and edges in each test graph

Table 4.2 shows the fill produced by the various orders, and the amount of fill our post-processing order induces, relative to the amount of fill the original order induces, that is, we divide the amount of fill of the post-processed order by the amount of fill for the order used as an input to our algorithm, whether produced by AMD, BEND or Metis. The amount of fill includes entries that are in the original graph as well as any fill entries, bellow and including the diagonal.

Table 4.3 shows the amount of work involved in performing Gaussian elimination according to each of the initial orders and the amount of work for the post-processed orders relative to the initial orders.

The heights measured correspond to the reordering heights for the various orders and are presented in Table 4.4 while the number of reordering stages, that is, the number of parallel dense elimination steps, for the various orders are presented in Table 4.5.

Our results indicate that our algorithm usually produces orders that have a small amount of extra fill when compared to the chordal completion it starts with. In some cases, our post-processing actually produces small improvements in the number of non-zeros. Contrary to what we expected, for most graphs, the AMD orders were very parallel, thus making it harder for us to obtain significant improvements in the height or number of stages. It is interesting to notice that for grids with large aspect ratio the AMD orders cannot be directly parallelized. In those test cases, our orders induce a slightly lower number of non-zero entries than the nested dissection orders, and a small constant

| Matrix | AMD ($\times 10^3$) | Post fill | BEND ($\times 10^3$) | Post fill | Metis ($\times 10^3$) | Post fill |
|---|---|---|---|---|---|---|
| 3dtube | 26355 | 1.019 | 17906 | 1.002 | 18468 | 0.999 |
| bcsstk30 | 3854 | 1.108 | 3892 | 1.026 | 4439 | 0.993 |
| bcsstk31 | 5557 | 1.064 | 4183 | 1.015 | 4412 | 0.998 |
| bcsstk32 | 4987 | 1.033 | 5067 | 1.018 | 5731 | 0.996 |
| bcsstk33 | 2571 | 1.004 | 1880 | 1.015 | 2307 | 0.997 |
| bcsstk35 | 2732 | 1.038 | 2776 | 1.011 | 3132 | 0.992 |
| bcsstk36 | 2733 | 1.012 | 2555 | 1.022 | 3037 | 0.994 |
| bcsstk37 | 2799 | 1.033 | 2693 | 1.017 | 3146 | 0.987 |
| cfd1 | 37734 | 1.105 | 22386 | 1.011 | 22845 | 0.999 |
| cfd2 | 75008 | 1.014 | 39026 | 1.034 | 38937 | 1.000 |
| g256x256 | 1971 | 1.014 | 1675 | 1.006 | 1760 | 0.997 |
| g64x1024 | 1425 | 1.190 | 1351 | 1.026 | 1455 | 0.996 |
| gearbox | 48556 | 1.042 | 38089 | 1.012 | 38147 | 0.999 |
| hex256 | 54841 | 1.000 | 50091 | 1.001 | 45434 | 0.999 |
| hex64 | 2422 | 1.000 | 2080 | 1.006 | 2017 | 0.998 |
| hsct16k | 2680 | 1.018 | 2479 | 1.004 | 2817 | 0.993 |
| nasasrb | 11954 | 1.168 | 9765 | 1.051 | 10582 | 0.996 |
| pwt | 1592 | 1.011 | 1494 | 1.005 | 1389 | 0.996 |
| sf10 | 676 | 1.030 | 531 | 1.010 | 570 | 1.000 |
| sf5 | 5244 | 1.020 | 3873 | 1.083 | 3997 | 1.000 |
| shuttle-eddy | 327 | 1.123 | 330 | 1.062 | 368 | 0.998 |
| struct3 | 5093 | 1.013 | 4452 | 1.008 | 4574 | 0.992 |

Table 4.2: Amount of fill for each order and the corresponding post processed order

factor more nonzeros than the AMD orders. In these cases, the orders our algorithm produced are significantly more parallel then the original AMD orders, and only slightly less parallel than the nested dissection orders. The hybrid algorithm produces orders that are usually good in terms of both fill and height.

In the next chapter we provide some indication that this trade-off between low-fill and low-height orders is inherent to the chordal completion problem.

| Matrix | AMD ($\times 10^6$) | Post work | BEND ($\times 10^6$) | Post work | Metis ($\times 10^6$) | Post work |
|---|---|---|---|---|---|---|
| 3dtube | 30030 | 1.064 | 12740 | 1.005 | 12130 | 0.999 |
| bcsstk30 | 942 | 1.406 | 936 | 1.089 | 1179 | 0.993 |
| bcsstk31 | 2898 | 1.214 | 1159 | 1.045 | 1190 | 1.003 |
| bcsstk32 | 948 | 1.098 | 959 | 1.061 | 1288 | 0.996 |
| bcsstk33 | 1240 | 1.004 | 590 | 1.033 | 896 | 0.996 |
| bcsstk35 | 383 | 1.145 | 394 | 1.026 | 509 | 0.990 |
| bcsstk36 | 620 | 1.045 | 483 | 1.099 | 706 | 0.993 |
| bcsstk37 | 532 | 1.131 | 477 | 1.071 | 691 | 0.981 |
| cfd1 | 44520 | 1.278 | 13360 | 1.031 | 17930 | 1.000 |
| cfd2 | 136400 | 1.037 | 28710 | 1.130 | 34340 | 1.000 |
| g256x256 | 261 | 1.066 | 190 | 1.028 | 223 | 0.999 |
| g64x1024 | 85 | 1.909 | 83 | 1.123 | 98 | 0.998 |
| gearbox | 47020 | 1.189 | 22020 | 1.032 | 23330 | 1.000 |
| hex256 | 47860 | 1.000 | 41620 | 1.001 | 34960 | 1.000 |
| hex64 | 691 | 1.000 | 562 | 1.028 | 468 | 1.000 |
| hsct16k | 718 | 1.057 | 578 | 1.005 | 784 | 0.991 |
| nasasrb | 4771 | 1.724 | 2820 | 1.188 | 3548 | 0.993 |
| pwt | 172 | 1.039 | 140 | 1.014 | 112 | 0.985 |
| sf10 | 137 | 1.090 | 71 | 1.025 | 80 | 1.002 |
| sf5 | 2781 | 1.055 | 1275 | 1.358 | 1374 | 1.001 |
| shuttle-eddy | 17 | 1.514 | 18 | 1.237 | 23 | 0.998 |
| struct3 | 1091 | 1.049 | 706 | 1.031 | 769 | 0.993 |

Table 4.3: Amount of work for each order and the corresponding post processed order

| Matrix | AMD | Post height | BEND | Post height | Metis | Post height |
|---|---|---|---|---|---|---|
| 3dtube | 5041 | 0.867 | 3477 | 0.905 | 2655 | 1.000 |
| bcsstk30 | 2764 | 0.747 | 1800 | 0.833 | 1350 | 0.963 |
| bcsstk31 | 2285 | 1.021 | 1641 | 1.000 | 1291 | 0.997 |
| bcsstk32 | 2457 | 0.818 | 1774 | 0.931 | 1386 | 0.953 |
| bcsstk33 | 1792 | 1.000 | 1534 | 0.961 | 1250 | 0.999 |
| bcsstk35 | 1262 | 0.948 | 1084 | 0.989 | 970 | 0.939 |
| bcsstk36 | 1540 | 0.932 | 1255 | 1.042 | 1166 | 0.990 |
| bcsstk37 | 1333 | 1.065 | 1162 | 1.022 | 1232 | 1.000 |
| cfd1 | 7921 | 0.878 | 3438 | 1.003 | 3228 | 1.000 |
| cfd2 | 9494 | 1.000 | 5538 | 0.967 | 4068 | 1.000 |
| g256x256 | 1617 | 0.978 | 995 | 0.997 | 744 | 0.999 |
| g64x1024 | 2791 | 0.319 | 979 | 0.733 | 438 | 0.995 |
| gearbox | 5589 | 1.124 | 3899 | 1.000 | 3640 | 1.000 |
| hex256 | 4549 | 1.000 | 5423 | 1.002 | 4362 | 0.999 |
| hex64 | 1105 | 1.000 | 1226 | 1.076 | 1018 | 0.988 |
| hsct16k | 1917 | 0.942 | 1307 | 0.995 | 1163 | 1.000 |
| nasasrb | 4829 | 0.549 | 3360 | 0.725 | 1662 | 1.112 |
| pwt | 944 | 0.935 | 778 | 0.857 | 571 | 0.993 |
| sf10 | 793 | 0.974 | 703 | 0.983 | 557 | 1.000 |
| sf5 | 2341 | 1.000 | 1830 | 1.033 | 1523 | 0.998 |
| shuttle-eddy | 851 | 0.677 | 659 | 0.730 | 335 | 0.988 |
| struct3 | 1542 | 0.982 | 1440 | 1.002 | 1093 | 0.992 |

Table 4.4: Reordering height of each order

| Matrix | AMD | Post stages | BEND | Post stages | Metis | Post stages |
|---|---|---|---|---|---|---|
| 3dtube | 17 | 0.941 | 18 | 1.000 | 23 | 1.043 |
| bcsstk30 | 38 | 0.684 | 24 | 0.917 | 23 | 0.826 |
| bcsstk31 | 29 | 0.931 | 30 | 0.967 | 30 | 0.833 |
| bcsstk32 | 40 | 0.775 | 27 | 0.963 | 30 | 0.867 |
| bcsstk33 | 23 | 1.043 | 22 | 0.864 | 19 | 1.000 |
| bcsstk35 | 32 | 0.844 | 29 | 0.966 | 30 | 0.833 |
| bcsstk36 | 24 | 0.958 | 24 | 1.000 | 26 | 0.769 |
| bcsstk37 | 31 | 0.935 | 27 | 0.926 | 29 | 0.793 |
| cfd1 | 48 | 0.812 | 35 | 1.000 | 38 | 0.737 |
| cfd2 | 38 | 1.000 | 40 | 0.925 | 34 | 0.853 |
| g256x256 | 52 | 0.942 | 46 | 0.913 | 33 | 0.788 |
| g64x1024 | 109 | 0.413 | 47 | 0.809 | 31 | 0.774 |
| gearbox | 33 | 0.939 | 30 | 1.000 | 32 | 0.969 |
| hex256 | 16 | 1.000 | 22 | 1.000 | 37 | 0.730 |
| hex64 | 12 | 1.000 | 17 | 0.941 | 29 | 0.759 |
| hsct16k | 28 | 0.964 | 25 | 0.880 | 23 | 0.870 |
| nasasrb | 33 | 0.606 | 28 | 0.821 | 30 | 0.800 |
| pwt | 25 | 1.000 | 43 | 1.000 | 28 | 0.857 |
| sf10 | 24 | 0.958 | 31 | 0.903 | 32 | 0.812 |
| sf5 | 29 | 0.966 | 38 | 0.947 | 41 | 0.878 |
| shuttle-eddy | 39 | 0.667 | 26 | 0.769 | 21 | 0.857 |
| struct3 | 29 | 0.966 | 30 | 0.967 | 27 | 0.815 |

Table 4.5: Number of reordered stages

# Chapter 5

# Parallelism and fill minimization

It is interesting to look at the quality of the orders that nested dissection produces on chordal graphs, as we did in Chapter 4. The analysis of nested dissection on chordal graphs indicates that even if we had an oracle that could provide us with the minimal separators that form cliques in the minimum-fill solution, nested dissection would still produce orders with an amount of fill that is more than a constant times the size of the minimum-fill completion of the graph.

In this chapter we show that there exist graphs for which any parallel elimination order must have at least $\Omega(n)$ extra fill. We also present an algorithm that is a variation of nested dissection that tries to obtain low fill orders that are less parallel than regular nested dissection orders. When applied to chordal graphs this new algorithm produces zero fill. Our algorithm is very similar to nested dissection, but does not have the performance guarantees in terms of fill, height and work that make nested dissection so attractive from a theoretical point of view, except when applied to chordal graphs. Our experiments show this algorithm outperforms, on average, previous state-of-the-art implementations of other ordering heuristics, including minimum-degree, nested dissection and a hybrid of minimum-degree and nested dissection.

## 5.1 Studying height

The lemmas that follow help us analyze parallel elimination orders, and allow us to change a given order into a more parallel one without introducing any fill.

**Lemma 14** *(Jess and Kess [JK82]) The set of simplicial vertices of a graph $G$ consists of disconnected cliques.*

Based on this fact, Jess and Kess proposed an algorithm to produce elimination orders with low height, namely to recursively eliminating at each step of the algorithm a maximal set of independent simplicial vertices. Liu later proved that this algorithm indeed produces orders that achieve the reordering height on chordal graphs, as stated in the next lemma.

**Lemma 15** *(Liu [Liu89] ) Let $G$ be a chordal graph. Then a perfect elimination order for $G$ with minimum height can be obtained by applying the algorithm of Jess and Kess described above. The height corresponds to the number of maximal independent sets the algorithm eliminates.*

An order for a graph can be used for any of its subgraphs. We thus obtain the following lemma:

**Lemma 16** *(Manne [Man91]) Let $H$ be a subgraph of a graph $G$. Then the height of $H$ is less than or equal to the height of $G$.*

The notion of optimal height of a graph allows for fill to be introduced into the graph. However, we can still eliminate a maximal independent set of simplicial vertices at each step, without compromising the final height of the order. In general, however, some non-simplicial vertices might be included in the maximal independent set to be eliminated in a given step of a parallel elimination order. This is indicated in the lemmas that follow.

**Lemma 17** *Let $G$ be a graph and let $\pi$ be an order for the vertices of $G$ with height $h$ that introduces $f$ fill and $w$ work. There exists an order in which any set of independent vertices that are simplicial in $G$ are eliminated in the first parallel elimination step, the amount of fill is no larger than $f$, the amount of work is no larger than $w$, and the height is no larger than $h$.*

**Proof.** It suffices to show that a simplicial vertex $v$ can be added to the first step or can replace a vertex to which it is adjacent in the first step of the elimination.

If $v$ and $w$ are two adjacent simplicial vertices, and $w$ is in the first step of the elimination, then we can simply replace it by $v$, since $v$ and $w$ must be twins.

If a simplicial vertex $v$ is adjacent to $w$, and $w$ is not simplicial but appears in the first step of $\pi$, again we can replace $w$ by $v$. In this case, the neighborhood of $v$ is contained in the neighborhood of $w$, and thus the graph obtained by the elimination of $v$ is isomorphic to a subgraph of the graph obtained by the removal of $w$. This subgraph can be eliminated with no more height (by Lemma 16), fill or work than the graph obtained by the elimination of $w$ and the remaining vertices in the first step of the elimination, simply because it is a subgraph of that graph. The same argument is valid no matter how many vertices get replaced by simplicial neighbors.

The only case remaining is the case in which $v$ is simplicial but neither $v$ nor any of its neighbors are in the first step of the elimination order. Consider the first (future) step in which either $v$ or one of its neighbors is eliminated according to $\pi$. At that point, by the argument in the previous paragraph, we can insist that $v$ be eliminated in that step. Since none of $v$'s neighbors are eliminated before that step, we can move $v$ to any earlier elimination step. ■

The next lemma allows us to only consider elimination orders that eliminate twin vertices in consecutive steps.

**Lemma 18** *Let $v$ and $w$ be twin vertices in a graph $G$. Then given any elimination order for $G$ there exists another elimination order in which $v$ and $w$ are eliminated in consecutive steps and for which neither the fill, work or height exceed that of the original order.*

**Proof.** If $v$ and $w$ are twins, they remain so until either $v$ or $w$ is eliminated from $G$. Say $v$ is eliminated first. Once $v$ is eliminated, $w$ becomes a simplicial vertex, and by Lemma 17 there exists an order for the graph obtained by the elimination of $v$ with no more height, fill or work than the original order, and such that $w$ is eliminated in the first step of the order. ■

The next lemmas indicate that there exists an interesting trade-off in producing elimination orders that are optimized for height or fill, at least on some simple graphs.

**Lemma 19** *Let $P$ be a path with $n$ vertices. Any elimination order for $P$ with height $h$ requires at least $n - 2 \cdot h$ fill.*

**Proof.** Eliminating any vertex of $P$ produces a path one vertex shorter. Eliminating a leaf node produces no fill, while eliminating an internal node produces one fill edge. Since there are only two leafs at every step of the elimination process, at most $2 \cdot h$ leafs can be eliminated in $h$ steps. Therefore, at least $n - 2 \cdot h$ other vertices that are also to be eliminated during those $h$ steps introduce one fill edge each. ■

**Lemma 20** *Let $G$ be a graph, and let $H$ be an induced subgraph of $G$. If any elimination order for $H$ with height $h$ requires at least $f(h)$ fill, where $f(h)$ is a non-increasing function, then any elimination order for $G$ with height $h'$ requires at least $f(h')$ fill.*

**Proof.** Since $H$ is an induced subgraph of $G$, any elimination order of height $h'$ for $G$ is also an elimination order for $H$ and has height less than or equal to $h'$. Since $f$ is non-increasing, eliminating $H$ according to this order introduces at least $f(h')$ fill. Since $H$ is an induced subgraph of $G$, this same order must introduce at least $f(h')$ fill in $G$. ■

Now consider the following graph. Take a path of length $n$ and replace each node by a clique with $k$ vertices. Also introduce all possible edges between cliques that are in adjacent nodes of the path. Call this a $k$-path of length $n$.

**Lemma 21** *Any elimination order for the $k$-path of length $n$ with height $h$ introduces at least $k^2 \cdot (n - 2 \cdot h/k)$ fill.*

**Proof.** If we identify all the twin vertices of the $k$-path graph we obtain a path of length $n$, every node of which represents $k$ vertices of the original graph. According to Lemma 18 any order for the $k$-path can be modified so that when a vertex is eliminated all its twins are eliminated in successive steps. Thus an order with height $h$ for the $k$-path corresponds to an order of height $h/k$ for the $n$-node path graph. Every fill edge introduced in the elimination of the $n$-node path graph corresponds to $k^2$ fill edges in the original $k$-path elimination. By Lemma 19 any elimination order for the $k$-path has at least $k^2 \cdot (n - 2 \cdot h/k)$ fill. ■

It is interesting to note that the $k$-path graph is very similar to a graph created during the elimination of 2-dimensional grids with large aspect ratio. We triangulated a 5 $\times$ 100 rectangular grid and then ordered its vertices using the BEND algorithm. Figure 5.1 represents a step in the elimination of this graph according to the order produced by BEND. The vertices in grey have already been eliminated. The vertices that remain to be eliminated appear to be part of the top level separators that the BEND algorithm uses to partition the graph into smaller subgraphs (that are then ordered using minimum-degree.) At this particular point of the elimination it is not possible to distinguish between this order and a parallel nested dissection order. The partially filled graph obtained at this step is very similar to the $k$-path.

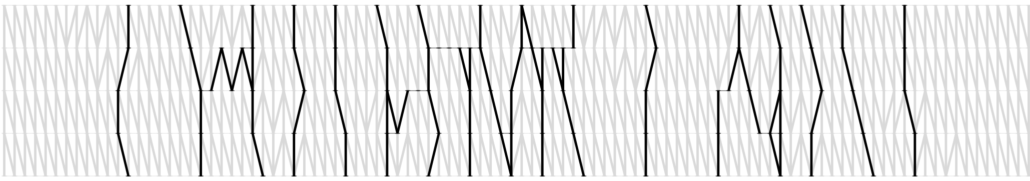


Figure 5.1: Partial elimination of a graph, according to a BEND order.

Consider the nested dissection ordering for a grid with width $n$ and height $k$, $k << n$. A likely nested dissection order would involve selecting roughly $n/k$ vertical separators with $k$ vertices each. The $k \times k$ subgraphs obtained this way would be recursively ordered, causing fill between consecutive separators. Each separator would become itself a clique. At this stage of the elimination process, the graph would correspond to a $k$-path of length $n/k$. Therefore by Lemma 21, after this point, any elimination order with low height would require an additional $\Omega(n \cdot k)$ fill.

This explains in part why nested dissection produces orders with higher amounts of fill than other ordering heuristics on such graphs. In an attempt to show this effect empirically, we performed an additional experiment. We took rectangular grids of different aspect ratios and computed an AMD and a nested dissection order for each graph. We then applied the parallelizing algorithm of the previous chapter as a post processing step to the AMD orders. We used a special purpose ordering algorithm for rectangular grids as the base for normalizing all the results. This algorithm is based on nested dissection and diagonal separators, but does not use the nested dissection order for the top levels of separators.

Although not conclusive, the results obtained corroborate the hypothesis that the trade-off between minimizing height and minimizing fill and work plays an important role in the comparison between minimum-degree and nested dissection orders. The parallel orders obtained required a little more fill and work than the corresponding nested dissection orders, as seen in Figure 5.2.

## 5.2 A less parallel nested dissection algorithm

Our goal was to redeem nested dissection by obtaining a new algorithm that would behave much like nested dissection, but would produce orders with less fill and work. We propose the following algorithm that takes a subgraph $G$ of a graph $H$ and the set $T = H \setminus G$ of vertices to be eliminated after the vertices of $G$ and produces an elimination order for the vertices of $G$. Henceforth, we call this algorithm the less parallel nested dissection algorithm (LPND).

The algorithm starts with an empty set $T$, and finds a minimal separator $S$ of $G$, and the con-

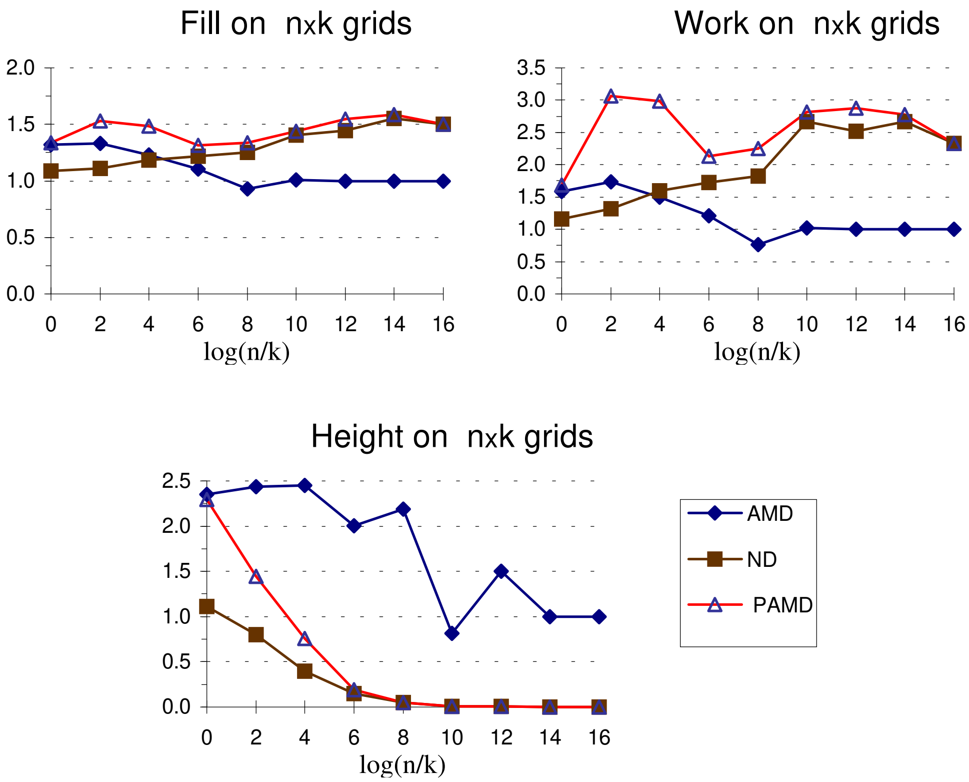



Figure 5.2: Comparison between orders produced by nested dissection (ND), minimum-degree (AMD), and the order obtained by applying the parallelizing algorithm of the previous chapter as a post processing step to the AMD order (PAMD).

nected components $G_k$ of $G\backslash S$. If a non-trivial minimal separator cannot be found then it must be the case that $G$ is a clique. In this case its vertices are ordered according to increasing degrees, that is, according to the number of neighbors each vertex has in $T$. At each step, if a non-trivial minimal separator can be found, the algorithm computes the number of neighbors that each component $G_k$ has in $T$. If at least two components have the maximum number of neighbors in $T$ over all components $G_k$, then the algorithm recurses on each component $G_k$ as a subgraph of $G_k \cup S \cup T$ (with a set $T' = T \cup S$), exactly like nested dissection would. Vertices within each component are numbered so as to be eliminated before the vertices in $S$, exactly like nested dissection would number them. Then the vertices of $S$ are ordered according to increasing degrees in the graph induced by $S \cup T$. If however, a subgraph $G_i$ has a number of neighbors in $T$ that is larger than the number of neighbors any other component has in $T$, then the algorithm treats $G_i$ and $S$ in a different manner. All components of $G \setminus S$ except for $G_i$ are recursively ordered, just as before. However, instead of recursing on $G_i$, the algorithm recurses on the subgraph $G_i \cup S$ of $G_i \cup S \cup T$, after inserting edges corresponding to the fill created by the elimination of the other components of $G \setminus S$. The vertices in $G_i \cup S$ are numbered after the vertices in the other components of $G \setminus S$.

We start by proving a lemma that says that, under certain conditions, the step of the LPND algorithm that orders the vertices of a clique does not introduce any fill.

**Lemma 22** *Let $C$ be a clique in a chordal graph $H$, such that $N_H(C)$ is also a clique. The vertices of $C$ can be eliminated from $H$ according to any minimum-degree order with zero fill.*

**Proof.** Since both $C$ and $N_H(C)$ are cliques we have to show that if $v$ and $w$ are vertices in $C$ such that $d(w) \geq d(v)$ then $w$ is adjacent to all neighbors of $v$, which implies that any vertex with minimum degree in $C$ is simplicial in $H$.

Assume by contradiction that this is not the case. Since $C$ is a clique, it must be the case that there exists a vertex $n_v \notin C$ that is adjacent to $v$ but not to $w$. Also, since $d(w) \geq d(v)$, there must exist a vertex $n_w \notin C$ that is adjacent to $w$ but not to $v$. But then $n_v$ and $n_w$ are both in $N_H(C)$, and thus are adjacent. That forms a chordless cycle of length four and contradicts the assumption that $H$ is chordal. ∎

The next lemma allows us to show that the LPND algorithm produces perfect elimination orders for chordal graphs.

**Lemma 23** *Let $G$ be an induced subgraph of a chordal graph $H$, such that $N_H(G)$ is a clique. Then the LPND algorithm orders the vertices of $G$ so that they can be eliminated from $H$ with zero fill.*

**Proof.** If $G$ is a clique then it has no non-trivial minimal separators, and the algorithm orders the graph using minimum degree, so that we can apply Lemma 22 proving the result. Assume $G$ is not a clique. Then it has a minimal separator $S$ that will be used by the algorithm to partition $G$ into connected components $G_1, \ldots, G_n$, $n > 1$. Since $G$ is an induced subgraph of $H$ it is also chordal. Every minimal separator of a chordal graph is a clique, and thus $S$ is a clique.

If there is a component $G_i$ that has more neighbors in $N_H(G)$ than any of the other component of $G \setminus S$, then the algorithm will first order the vertices within the components other than $G_i$, and then order the vertices of $G_i \cup S$. By induction, we can assume that the order produced for $G_i \cup S$ will eliminate these vertices from $G_i \cup S \cup (H \setminus G)$ with zero fill, since the set of neighbors of $G_i \cup S$ in $H \setminus G$ is contained in $N_H(G)$, which is a clique by hypothesis. Next, we will show that if a component $G_k$ does not have more neighbors in $N_H(G)$ than any other component, then $N_H(G_k)$ is a clique. In this case $G_k$ can be eliminated with zero-fill by induction.

Without loss of generality, take a component $G_1$ of $G \setminus S$. Assume $N_H(G_1)$ is not a clique. Then there must exist vertices $s_1$ in $S$ and $t_1$ in $N_H(G)$ that are adjacent to vertices in $G_1$ but are not adjacent to each other. Since $S$ is a minimal separator, all other components $G_k$ of $G \setminus S$ have at least one vertex adjacent to $s_1$, for otherwise $s_1$ could be removed from $S$. Either $G_1$ is the component with the largest number of neighbors in $N_H(G)$ or there exists some other component $G_2$ with no less neighbors in $N_H(G)$ than $G_1$, so that some vertices of $G_2$ are, without loss of generality, either adjacent to $t_1$ or to a vertex $t_2$ in $N_H(G)$ that does not have neighbors in $G_1$.

In the first case, consider a shortest path between $s_1$ and $t_1$ going only through $s_1, t_1$ and vertices in $G_1$ and a shortest path between $s_1$ and $t_1$ going only thorough $s_1$, $t_1$ and vertices in $G_2$. There are no edges between vertices in $G_1$ and vertices in $G_2$. Since $H$ is chordal the cycle formed by these paths must have a chord. The only place where this chord can be is between $s_1$ and $t_1$, a contradiction.

If no vertex of $G_2$ is adjacent to $t_1$, then again take a shortest path between $s_1$ and $t_1$ in $G_1$. The vertex $t_2$ is adjacent to $t_1$, and either $t_2$ is also adjacent to $s_1$, in which case the shortest path in $G_1$ along with the path $s_1 - t_2 - t_1$ is chordless, a contradiction, or $s_1$ and $t_2$ are not adjacent. In this case we consider the cycle composed of the shortest path from $s_1$ to $t_1$ in $G_1$, along with the edge $(t_1, t_2)$ and a shortest path between $t_2$ and $s_1$ in $G_2$. This cycle is again chordless, a contradiction. This concludes the proof that the subgraphs $G_k$ are ordered with zero fill.

The only case that remains is when all components of $G \setminus S$ are eliminated first, and the set of vertices $S$ is left to be eliminated at the end (as nested dissection would do). But $S$ is a clique, and so is $N_H(S)$, since $N_H(G)$ is clique. Thus we can apply Lemma 22. ■

**Corollary 3** *The LPND algorithm produces a perfect elimination order when applied to a chordal graph.*

**Proof.** Let $H$ be a chordal graph. If $H$ is a clique, then any order will produce zero fill. Otherwise, the algorithm finds a minimal separator $S$. $S$ is a clique since $H$ is chordal. The algorithm orders the vertices of $S$ last. As long as all other vertices have been eliminated, the vertices in $S$ can be eliminated in any order with zero fill. The algorithm will recurse on each of the components of $H \setminus S$, and by Lemma 23 it will order each of these components with zero fill. ■

Unlike nested dissection, the algorithm we described will not necessarily generate orders with low height. In particular, given a path of length $n$, this algorithm will not introduce any fill edges so that any order it produces must have height at least $n/2$.

## 5.3 Experimental results

We implemented the algorithm described in Section 5.2 and performed experiments that compare the orders it produces with orders produced by state-of-the-art implementations of a number of other heuristics. We proceed to describe a few aspects of the implementation of the LPND algorithm.

Separators:

The algorithm only requires minimal separators, without any balance constraints. However, in the actual implementation the algorithm tries to obtain separators with a bounded amount of imbalance allowed, thus mimicking nested dissection.

Obtaining small balanced separators is an NP-hard problem on its own, but there are provably good approximation algorithms for finding separators as well as a number of algorithms that, in practice, produce very good separators. We side-stepped the issues associated with finding separators and use vertex separators produced by Chaco[1] [HL94, HL95], a graph partitioning algorithm by Hendrickson and Leland at Sandia Labs.

Chaco is a multilevel algorithm that works by coarsening the graph into smaller and smaller graphs that approximate the initial graph. The smallest such graph is partitioned. This partition is mapped to a partition of the next smallest graph, from which the coarse graph was

[1] http://www.cs.sandia.gov/CRF/chac.html

obtained. The process proceeds, as partitions are propagated back to finer and finer graphs. Every so many of these mapping steps, a variant of the Kernigan-Lin algorithm is applied as a local refinement method that improves the partitioning. Chaco uses this combination of heuristics to directly produce high quality edge and vertex separators. Parameters to Chaco affect our algorithm and the orders it produces. Among the parameters to Chaco is the permissible fraction of imbalance between the partitions to be produced. Chaco uses randomization, most noticeably in the coarsening steps. This makes our ordering algorithm also sensitive to the initial random seeds used. Even though Chaco produces good separators, we noticed that we obtain even better solutions at the cost of speed, by running Chaco more than once with different random seeds on the same input graph and then choosing the smallest separator produced.

Minimum degree:

Most descriptions of nested dissection do not prescribe any ordering for vertices within a given separator because these vertices usually end up forming a clique after the components into which the separator breaks the graph have been eliminated. It is natural however, to try to minimize the fill from vertices in a separator to vertices higher up in the separator tree. An important part of the LPND algorithm is the use of a constrained minimum-degree rule to order the vertices within separators and to order the vertices in subgraphs that could not be further divided. The constrained minimum-degree heuristic orders the vertices in the subgraphs by eliminating first the vertex of minimum degree among the vertices of the subgraph, while however considering the degrees measured in the whole graph.

We also used constrained minimum degree as the ordering heuristic for subgraphs that were smaller than some threshold. This widely adopted practice allows us to reduce the amount of time needed to produce our orders, while not affecting the quality of the orders produced by much.

Sorting recursive subproblems:

The step that differentiates the LPND algorithm from nested dissection involves choosing whether to treat all subgraphs defined by the separator identically, or to save one subgraph for last and merge it with the separator. As we have shown with Corollary 3, the choice of which subgraph should be left for last is clear when the graph being ordered is already a chordal graph. In practice we made the conditions under which the algorithm deviates from nested dissection more strict. Instead of requiring a component to have a number of neighbors outside the subgraph being ordered that is larger than the number of neighbors of any other component, in the implementation, we actually require that this component have at least some constant fraction more neighbors than any other component. This is a conservative measure that tries to avoid making bad decisions based on the imperfect information that is available during the computation by defaulting to the nested dissection behavior.

We also implemented a version of the Lipton, Rose and Tarjan (LRT) nested dissection algorithm that uses the same separator and minimum-degree algorithms that were used in implementing our own algorithm. In the tables that follow we report the results obtained for our LPND algorithm, our implementation of the LRT algorithm, as well as the AMD [2] algorithm [ADD96], the BEND

[2] code from ftp://ftp.cise.ufl.edu/pub/umfpack/AMD/

algorithm, obtained from Rothberg [HR96, HR97], and the nested dissection order produced by METIS-3.0 [3] [KK95]. As for the results reported for the approximate minimum deficiency algorithm (AMMF), these are quoted from [RE98], and are not available for all the test matrices. Any unavailable results are represented with a "-" in the tables that follow.

As described in the previous chapter, the "bcsstk" matrices used in our experiments come from structural engineering problems, and were obtained from the Harwell-Boeing collection, and from Timothy Davis's "University of Florida Sparse Matrix Collection" [Dav94] (the matrices were provided to Davis by Roger Grimes, at Boeing). The "cfd", gearbox and struct3 matrices were provided by Rothberg, while the nasasrb, pwt and shuttle-eddy matrices came from NASA. All these matrices can be obtained from the University of Florida Sparse Matrix collection. The "sf" matrices come from the simulation of an earthquake in the San Fernando Valley [OS96]. The "g" matrices are $n \times k$ grids, while the "hex" matrices are meshes of hexagons. The CAR, hsct16k, 50K, 172K and 178K matrices came from Olaf Storaasli at NASA Langey.

| Matrix | vertices | edges | c-vertices | $\|L\|/10^3$ | work/$10^6$ | height | stages | front |
|---|---|---|---|---|---|---|---|---|
| shuttle-eddy | 10429 | 46585 | 10363 | 330 | 18 | 659 | 26 | 143 |
| sf10 | 7294 | 44922 | 7294 | 531 | 71 | 703 | 31 | 296 |
| g64x1024 | 65536 | 129984 | 65536 | 1351 | 83 | 979 | 47 | 188 |
| pwt | 36519 | 144794 | 36515 | 1494 | 140 | 778 | 43 | 222 |
| g256x256 | 65536 | 130560 | 65536 | 1675 | 190 | 995 | 46 | 390 |
| bcsstk35 | 30237 | 709963 | 6611 | 2776 | 394 | 1084 | 29 | 430 |
| bcsstk37 | 25503 | 557737 | 7093 | 2693 | 477 | 1162 | 27 | 518 |
| bcsstk36 | 23052 | 560044 | 4351 | 2555 | 483 | 1255 | 24 | 536 |
| hex64 | 24576 | 147456 | 24576 | 2080 | 562 | 1226 | 17 | 814 |
| hsct16k | 16152 | 376432 | 7911 | 2479 | 578 | 1307 | 25 | 491 |
| bcsstk33 | 8738 | 291583 | 4344 | 1880 | 590 | 1534 | 22 | 677 |
| struct3 | 53570 | 560062 | 41644 | 4452 | 706 | 1440 | 30 | 446 |
| bcsstk30 | 28924 | 1007284 | 9289 | 3892 | 936 | 1800 | 24 | 579 |
| bcsstk32 | 44609 | 985046 | 14821 | 5067 | 959 | 1774 | 27 | 564 |
| bcsstk31 | 35588 | 572914 | 17403 | 4183 | 1159 | 1641 | 30 | 623 |
| sf5 | 30169 | 190377 | 30169 | 3873 | 1275 | 1830 | 38 | 829 |
| 50K | 49790 | 1253700 | 11224 | 7313 | 2235 | 2716 | 36 | 879 |
| nasasrb | 54870 | 1311227 | 24954 | 9765 | 2820 | 3360 | 28 | 620 |
| 3dtube | 45330 | 1584144 | 15909 | 17906 | 12740 | 3477 | 18 | 1607 |
| cfd1 | 70656 | 878854 | 70656 | 22386 | 13360 | 3438 | 35 | 1417 |
| CAR | 263574 | 6292129 | 61983 | 34352 | 13970 | 3824 | 40 | 1571 |
| gearbox | 153746 | 4463329 | 56175 | 38089 | 22020 | 3899 | 30 | 1846 |
| 172K | 172400 | 7229603 | 36062 | 35601 | 24700 | 4790 | 26 | 2073 |
| cfd2 | 123440 | 1482229 | 123440 | 39026 | 28710 | 5538 | 40 | 1754 |
| hex256 | 393216 | 2359296 | 393216 | 50091 | 41620 | 5423 | 22 | 3319 |
| 178K | 178044 | 6277614 | 145112 | 132100 | 183110 | - | - | - |
| sf2 | 378747 | 2509064 | 378747 | 188240 | 339500 | - | - | - |

Table 5.1: Matrices and statistics for a BEND order

[3] code from ftp://ftp.cs.umn.edu/dept/users/kumar/metis/

The number of vertices and edges in each graph/matrix can be found in Table 5.1. Under the label "c-vertices" this table lists the number of vertices in the compressed graph that is obtained by identifying each set of twin vertices into a supernode. We applied the BEND algorithm to order each of these matrices, and use the numbers obtained to normalize the results that are presented in the next tables. The results for the BEND order are also included in Table 5.1. The fill and work entries indicate the number of nonzeros in the matrix $L$ and the amount of work to decompose each matrix according to the BEND order. This and the remaining tables that we present are sorted according to increasing amounts of work. Finally, the height, stages, and front entries correspond to the reordering height, to the number of reordering stages, and to the front size for the BEND order.

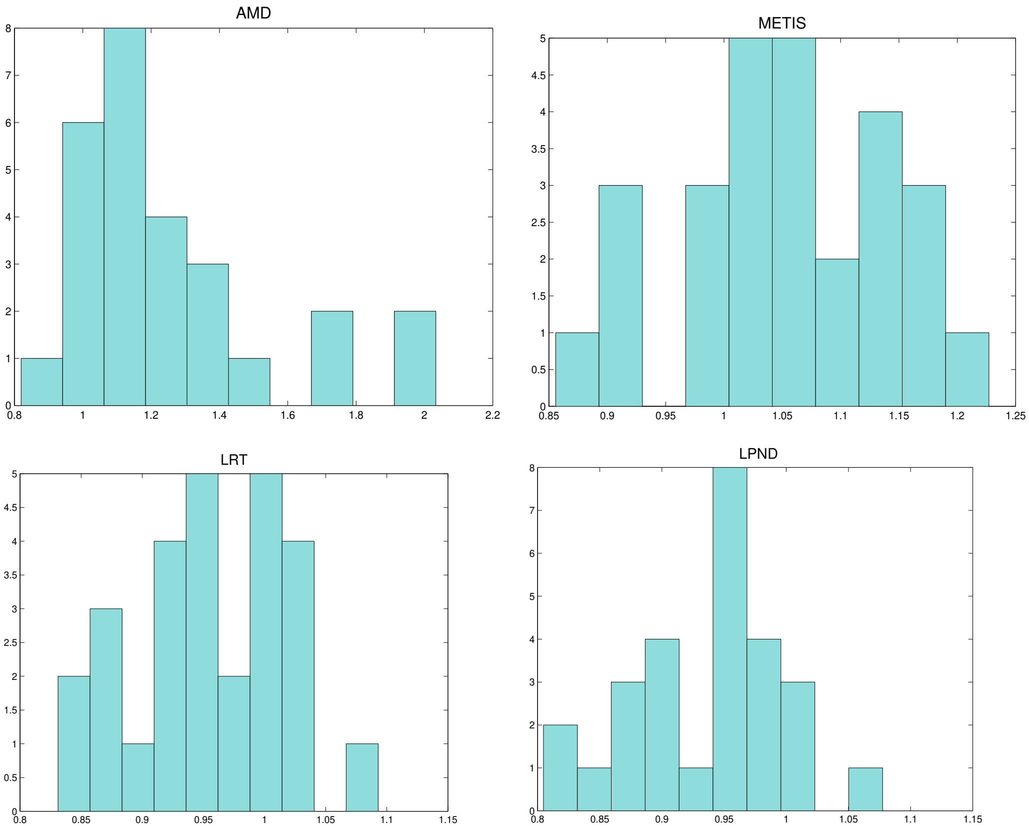



Figure 5.3: Histograms of the fill of each order for the various matrices relative to the BEND order. The $y$-axis indicates the number of matrices whose fill-ratio is in each bucket.

Tables 5.2 and 5.3 present the number of nonzeros (below and including the diagonal, i.e, the number of nonzeros in $L$) and work for decomposing each matrix according to the various orders produced. Both LRT and LPND used **two** calls to Chaco per separator. These results are presented as ratios with respect to the BEND order. Numbers smaller than 1 indicate an order that compares favorably with the BEND order. In an attempt to summarize the results in each table, we included the average and standard deviation of these normalized results, as well as the histograms in Figures 5.3 and 5.4.

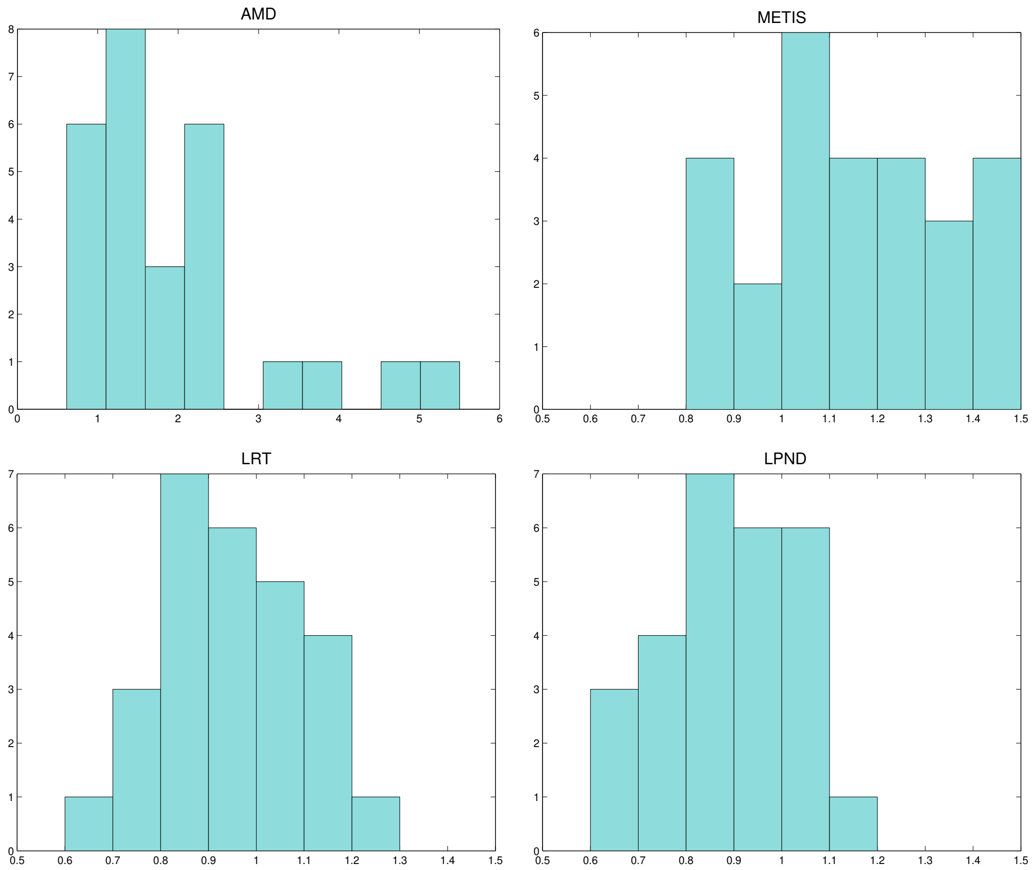



Figure 5.4: Histograms of the work of each order for the various matrices relative to the BEND order. The $y$-axis indicates the number of matrices whose work-ratio is in each bucket.

These results indicate that both LRT and LPND orders are on average slightly better than the BEND order. However, the differences observed are of only a few percent, making it difficult to draw any definite conclusions. The running times for both LRT and LPND are comparable and are significantly higher than those for the BEND algorithm. Table 5.4 compares the amount of time to produce the BEND and the LPND orders for the various matrices. It is unclear how much the BEND orders could improve if the algorithm were given more time to compute the orders. However, these results indicate that significant gains in terms of both fill and work might justify investing more time into obtaining good elimination orders.

The fill and work results for the AMD orders stand out. Although occasionally better in individual cases, the AMD algorithm produces orders that are significantly worse than the orders produced by the remaining algorithms. This is consistent with the results obtained by Hendrickson and Rothberg in [HR96], when they found that their implementation of the nested dissection algorithm was significantly better than the minimum-degree algorithms, and that the BEND algorithm was a few percent better than their own implementation of the nested dissection algorithm.

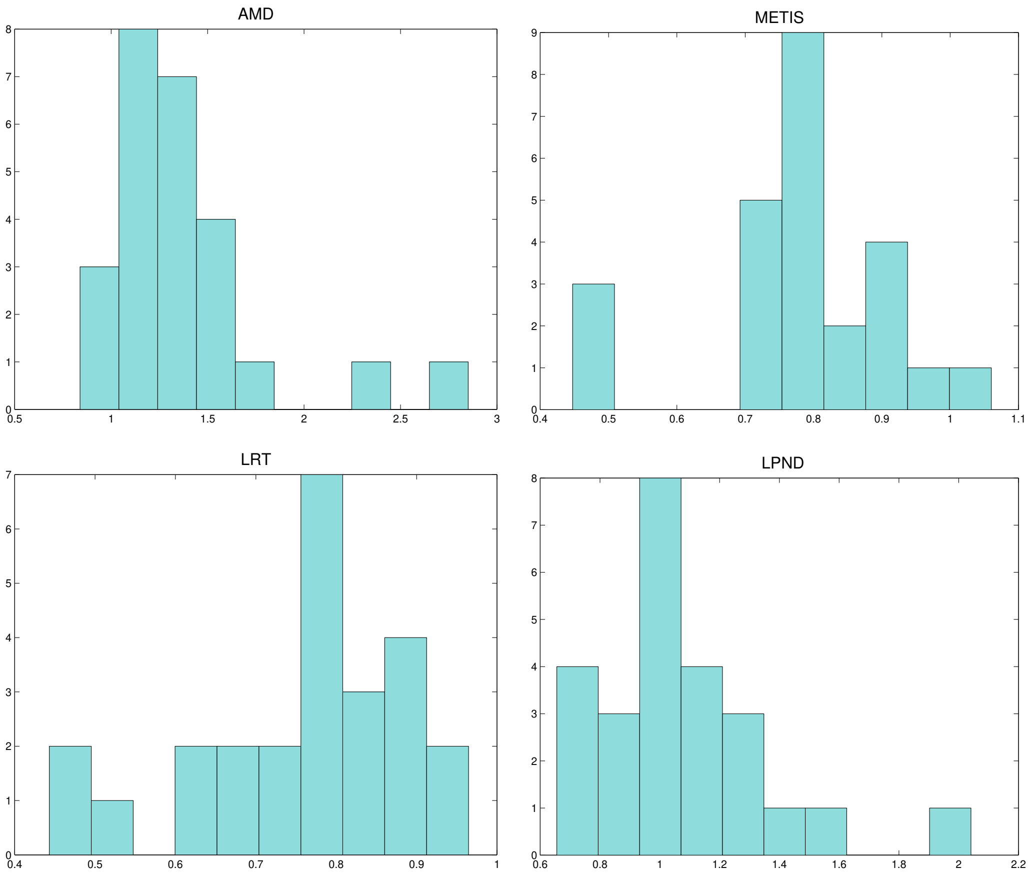



Figure 5.5: Histograms of the reordering height of each order for the various matrices relative to the BEND order. The $y$-axis indicates the number of matrices whose reordering-height ratio is in each bucket.

Tables 5.5 and 5.6 show the reordering height and number of reordering stages for each of the orders. The results for the AMMF order are not available. The results for the two largest cases are omitted, for they would take too long to compute. As expected, we can see in the tables that the orders produced by the LPND algorithm are less parallel than the corresponding LRT orders while the AMD orders are even less parallel than the LPND orders in terms of height, but in most cases are more parallel than the LPND orders in terms of the number of stages. This difference indicates that AMD orders are probably likely to produce larger cliques than the LPND orders, as corroborated by Table 5.7, which lists the maximum front size for each order (again, except for the AMMF orders).

We also performed an additional experiment using the 7 largest matrices in the test set. We ran the LPND algorithm 10 times using different random seeds (the numbers 1 through 10) and one call to Chaco per separator (LPND1) and another 10 times, with two calls to Chaco per separator (LPND2). In the first call the same seeds (1 through 10) were used, and in the second call the seed was obtained by applying a fixed affine function to the original seed. Figure 5.6 shows the average

| Ratio | AMD | AMMF | METIS | LRT | LPND |
|---|---|---|---|---|---|
| shuttle-eddy | 0.9910 | - | 1.1144 | 1.0139 | 0.9415 |
| sf10 | 1.2726 | - | 1.0737 | 1.0205 | 0.9905 |
| g64x1024 | 1.0553 | - | 1.0771 | 0.9686 | 0.8757 |
| pwt | 1.0656 | 1.1205 | 0.9295 | 0.8740 | 0.8754 |
| g256x256 | 1.1772 | - | 1.0512 | 0.9056 | 0.8986 |
| bcsstk35 | 0.9843 | 0.9645 | 1.1284 | 0.9944 | 0.9733 |
| bcsstk37 | 1.0395 | 1.0154 | 1.1682 | 0.9564 | 0.9567 |
| bcsstk36 | 1.0695 | 1.0375 | 1.1887 | 1.0145 | 1.0028 |
| hex64 | 1.1644 | - | 0.9699 | 0.9105 | 0.9103 |
| hsct16k | 1.0808 | - | 1.1361 | 1.0276 | 1.0080 |
| bcsstk33 | 1.3680 | 1.2757 | 1.2271 | 1.0929 | 1.0775 |
| struct3 | 1.1441 | 1.1185 | 1.0274 | 0.9468 | 0.9667 |
| bcsstk30 | 0.9903 | 0.8869 | 1.1407 | 1.0196 | 0.9102 |
| bcsstk32 | 0.9840 | 0.9541 | 1.1310 | 1.0135 | 0.9824 |
| bcsstk31 | 1.3286 | 1.0525 | 1.0549 | 0.9449 | 0.9575 |
| sf5 | 1.3541 | - | 1.0320 | 0.9595 | 0.9521 |
| 50K | 1.2086 | - | 1.1721 | 1.0003 | 0.9982 |
| nasasrb | 1.2242 | - | 1.0837 | 1.0086 | 0.9560 |
| 3dtube | 1.4719 | 1.5511 | 1.0314 | 0.9548 | 0.9548 |
| cfd1 | 1.6856 | 1.3106 | 1.0205 | 0.8597 | 0.8047 |
| CAR | 1.1028 | - | 1.0631 | 0.9769 | 0.9846 |
| gearbox | 1.2748 | 1.2614 | 1.0015 | 0.9227 | 0.9166 |
| 172K | 0.8196 | - | 1.0203 | 0.9249 | 0.9010 |
| cfd2 | 1.9220 | 1.6969 | 0.9977 | 0.9218 | 0.9411 |
| hex256 | 1.0948 | - | 0.9070 | 0.8420 | 0.8420 |
| 178K | 1.7301 | - | 0.9272 | 0.8650 | 0.8645 |
| sf2 | 2.0333 | - | 0.8558 | 0.8311 | 0.8306 |
| AVERAGE | 1.2458 | 1.1727 | 1.0567 | 0.9545 | 0.9360 |
| STDEV | 0.2959 | 0.2400 | 0.0911 | 0.0650 | 0.0618 |

Table 5.2: Fill relative to the BEND order

results normalized by dividing each average by the corresponding result for the BEND order. Each average is accompanied by an error bar corresponding to the normalized standard deviation over each set of 10 runs. These deviations are sometimes too small to show up in the figure. Again, these results indicate that the orders produced are on average slightly better than the ones produced by the BEND algorithm.

| Ratio | AMD | AMMF | METIS | LRT | LPND |
|---|---|---|---|---|---|
| shuttle-eddy | 0.9642 | - | 1.2855 | 1.1027 | 0.8422 |
| sf10 | 1.9313 | - | 1.1294 | 1.0865 | 0.9736 |
| g64x1024 | 1.0160 | - | 1.1778 | 0.9716 | 0.6271 |
| pwt | 1.2305 | 1.3761 | 0.8017 | 0.7194 | 0.7215 |
| g256x256 | 1.3736 | - | 1.1728 | 0.8519 | 0.8087 |
| bcsstk35 | 0.9721 | 0.9110 | 1.2905 | 1.0233 | 0.9642 |
| bcsstk37 | 1.1150 | 1.0106 | 1.4479 | 0.8705 | 0.8869 |
| bcsstk36 | 1.2833 | 1.1255 | 1.4609 | 1.0786 | 1.0075 |
| hex64 | 1.2301 | - | 0.8330 | 0.7139 | 0.7137 |
| hsct16k | 1.2415 | - | 1.3558 | 1.0901 | 1.0337 |
| bcsstk33 | 2.1035 | 1.6197 | 1.5193 | 1.2053 | 1.1730 |
| struct3 | 1.5455 | 1.4126 | 1.0895 | 0.9266 | 0.9841 |
| bcsstk30 | 1.0057 | 0.7022 | 1.2592 | 1.1182 | 0.7630 |
| bcsstk32 | 0.9885 | 0.9305 | 1.3432 | 1.1273 | 1.0396 |
| bcsstk31 | 2.5004 | 1.3932 | 1.0267 | 0.8645 | 0.9077 |
| sf5 | 2.1812 | - | 1.0776 | 0.8996 | 0.8824 |
| 50K | 2.2367 | - | 1.4872 | 1.0224 | 1.0336 |
| nasasrb | 1.6918 | - | 1.2582 | 1.1411 | 0.9199 |
| 3dtube | 2.3571 | 2.4563 | 0.9521 | 0.8673 | 0.8673 |
| cfd1 | 3.3323 | 1.7663 | 1.3421 | 0.8293 | 0.6635 |
| CAR | 1.6600 | - | 1.0014 | 0.9556 | 1.0193 |
| gearbox | 2.1353 | 2.0058 | 1.0595 | 0.9382 | 0.9260 |
| 172K | 0.6138 | - | 0.9425 | 0.7854 | 0.7186 |
| cfd2 | 4.7510 | 3.4315 | 1.1961 | 0.9871 | 1.0355 |
| hex256 | 1.1499 | - | 0.8400 | 0.6540 | 0.6540 |
| 178K | 3.8589 | - | 1.0334 | 0.9012 | 0.8941 |
| sf2 | 5.5022 | - | 0.8999 | 0.8315 | 0.8031 |
| AVERAGE | 1.9249 | 1.5493 | 1.1586 | 0.9468 | 0.8838 |
| STDEV | 1.1934 | 0.7459 | 0.2100 | 0.1441 | 0.1409 |

Table 5.3: Work relative to the BEND order

| Matrix | BEND(secs) | Slow-down: LPND/BEND |
|---|---|---|
| shuttle-eddy | 0.981 | 3.258 |
| sf10 | 0.962 | 3.534 |
| g64x1024 | 4.730 | 4.599 |
| pwt | 3.049 | 3.938 |
| g256x256 | 4.407 | 4.975 |
| bcsstk35 | 7.365 | 0.351 |
| bcsstk37 | 5.539 | 0.610 |
| bcsstk36 | 5.400 | 0.313 |
| hex64 | 2.805 | 3.646 |
| hsct16k | 3.322 | 2.079 |
| bcsstk33 | 2.133 | 1.915 |
| struct3 | 7.131 | 3.565 |
| bcsstk30 | 7.680 | 0.893 |
| bcsstk32 | 10.142 | 0.837 |
| bcsstk31 | 7.063 | 1.684 |
| sf5 | 3.491 | 5.916 |
| 50K | 11.594 | 0.465 |
| nasasrb | 11.817 | 1.805 |
| 3dtube | 12.485 | 0.988 |
| cfd1 | 13.385 | 8.632 |
| CAR | 42.983 | 0.850 |
| gearbox | 33.132 | 1.378 |
| 172K | 47.357 | 0.688 |
| cfd2 | 18.237 | 13.141 |
| hex256 | 41.574 | 6.313 |
| 178K | 54.393 | 9.461 |
| sf2 | 59.876 | 17.186 |

Table 5.4: Amount of time to compute BEND order, in seconds, and ratio between the time to compute the LPND order and the BEND order

| Ratio | AMD | METIS | LRT | LPND |
|---|---|---|---|---|
| shuttle-eddy | 1.2914 | 0.5083 | 0.4431 | 1.1897 |
| sf10 | 1.1280 | 0.7923 | 0.8990 | 0.9900 |
| g64x1024 | 2.8509 | 0.4474 | 0.5455 | 2.0409 |
| pwt | 1.2134 | 0.7339 | 0.6530 | 0.7558 |
| g256x256 | 1.6251 | 0.7477 | 0.8151 | 0.9980 |
| bcsstk35 | 1.1642 | 0.8948 | 0.8792 | 1.0563 |
| bcsstk37 | 1.1472 | 1.0602 | 0.8614 | 1.0258 |
| bcsstk36 | 1.2271 | 0.9291 | 0.9211 | 0.9793 |
| hex64 | 0.9013 | 0.8303 | 0.8051 | 0.8051 |
| hsct16k | 1.4667 | 0.8898 | 0.9082 | 1.2754 |
| bcsstk33 | 1.1682 | 0.8149 | 0.7640 | 0.7823 |
| struct3 | 1.0708 | 0.7590 | 0.6479 | 0.9868 |
| bcsstk30 | 1.5356 | 0.7500 | 0.7361 | 1.4728 |
| bcsstk32 | 1.3850 | 0.7813 | 0.7418 | 1.2272 |
| bcsstk31 | 1.3924 | 0.7867 | 0.7569 | 1.0049 |
| sf5 | 1.2792 | 0.8322 | 0.8066 | 1.0443 |
| 50K | 1.0670 | 0.7323 | 0.6116 | 0.6554 |
| nasasrb | 1.4372 | 0.4946 | 0.4726 | 1.2711 |
| 3dtube | 1.4498 | 0.7636 | 0.8332 | 0.8332 |
| cfd1 | 2.3040 | 0.9389 | 0.7833 | 1.5852 |
| CAR | 1.2644 | 0.7858 | 0.8413 | 1.1462 |
| gearbox | 1.4334 | 0.9336 | 0.9646 | 1.1749 |
| 172K | 0.9305 | 0.7931 | 0.7975 | 0.8929 |
| cfd2 | 1.7143 | 0.7346 | 0.6844 | 1.1685 |
| hex256 | 0.8388 | 0.8044 | 0.7640 | 0.7533 |
| 178K | - | - | - | - |
| sf2 | - | - | - | - |
| AVERAGE | 1.3714 | 0.7816 | 0.7575 | 1.0846 |
| STDEV | 0.4301 | 0.1380 | 0.1348 | 0.2999 |

Table 5.5: Reordering height relative to the BEND order

| Ratio | AMD | METIS | LRT | LPND |
|---|---|---|---|---|
| shuttle-eddy | 1.5000 | 0.8077 | 0.6538 | 1.4615 |
| sf10 | 0.7742 | 1.0323 | 0.8387 | 1.1290 |
| g64x1024 | 2.3191 | 0.6596 | 0.5319 | 2.0213 |
| pwt | 0.5814 | 0.6512 | 0.6977 | 0.7674 |
| g256x256 | 1.1304 | 0.7174 | 0.5435 | 0.7391 |
| bcsstk35 | 1.1034 | 1.0345 | 0.9655 | 1.1379 |
| bcsstk37 | 1.1481 | 1.0741 | 1.0000 | 1.1481 |
| bcsstk36 | 1.0000 | 1.0833 | 0.8333 | 0.9583 |
| hex64 | 0.7059 | 1.7059 | 1.0588 | 1.0588 |
| hsct16k | 1.1200 | 0.9200 | 0.7600 | 1.1200 |
| bcsstk33 | 1.0455 | 0.8636 | 0.8636 | 0.9545 |
| struct3 | 0.9667 | 0.9000 | 0.7667 | 1.0333 |
| bcsstk30 | 1.5833 | 0.9583 | 0.8750 | 1.6667 |
| bcsstk32 | 1.4815 | 1.1111 | 0.9630 | 1.7778 |
| bcsstk31 | 0.9667 | 1.0000 | 1.0000 | 1.2333 |
| sf5 | 0.7632 | 1.0789 | 1.0526 | 1.3158 |
| 50K | 0.7778 | 1.5000 | 0.8611 | 0.9722 |
| nasasrb | 1.1786 | 1.0714 | 0.6786 | 1.5714 |
| 3dtube | 0.9444 | 1.2778 | 1.0000 | 1.0000 |
| cfd1 | 1.3714 | 1.0857 | 0.7143 | 1.4857 |
| CAR | 1.4000 | 0.8500 | 0.8250 | 1.0500 |
| gearbox | 1.1000 | 1.0667 | 1.0667 | 1.3000 |
| 172K | 1.0385 | 1.2308 | 0.8462 | 1.1154 |
| cfd2 | 0.9500 | 0.8500 | 0.7000 | 1.4750 |
| hex256 | 0.7273 | 1.6818 | 1.0909 | 1.0909 |
| 178K | - | - | - | - |
| sf2 | - | - | - | - |
| AVERAGE | 1.1071 | 1.0485 | 0.8475 | 1.2233 |
| STDEV | 0.3642 | 0.2717 | 0.1617 | 0.3108 |

Table 5.6: Number of reordered stages relative to the BEND order

| Ratio | AMD | METIS | LRT | LPND |
|---|---|---|---|---|
| shuttle-eddy | 0.8462 | 1.1748 | 1.1049 | 0.7762 |
| sf10 | 1.5135 | 1.0980 | 1.1858 | 0.9155 |
| g64x1024 | 0.7766 | 1.0266 | 1.0160 | 0.6755 |
| pwt | 1.1351 | 1.1577 | 0.8649 | 0.8559 |
| g256x256 | 1.0974 | 1.1103 | 1.0179 | 1.0179 |
| bcsstk35 | 0.8814 | 1.0442 | 1.0977 | 0.9953 |
| bcsstk37 | 1.0174 | 1.2046 | 0.8301 | 0.9093 |
| bcsstk36 | 1.2705 | 1.0989 | 1.2425 | 0.9757 |
| hex64 | 0.9349 | 0.7850 | 0.7064 | 0.7064 |
| hsct16k | 1.2077 | 1.3910 | 1.2322 | 1.2688 |
| bcsstk33 | 1.3870 | 1.1152 | 1.0295 | 1.0295 |
| struct3 | 1.4507 | 1.5045 | 1.0067 | 1.0942 |
| bcsstk30 | 1.0294 | 1.2003 | 1.1796 | 0.8497 |
| bcsstk32 | 0.9521 | 1.2801 | 1.1915 | 1.2305 |
| bcsstk31 | 1.9695 | 0.9791 | 0.9310 | 0.9727 |
| sf5 | 1.5416 | 1.0567 | 0.8492 | 0.8480 |
| 50K | 1.8794 | 1.4710 | 0.9317 | 0.9807 |
| nasasrb | 1.5339 | 1.3339 | 1.3403 | 1.1016 |
| 3dtube | 1.3759 | 0.8693 | 0.8693 | 0.8693 |
| cfd1 | 2.0487 | 1.4912 | 1.0974 | 0.8772 |
| CAR | 1.3997 | 0.9236 | 1.0102 | 1.0993 |
| gearbox | 1.4588 | 1.1647 | 1.1712 | 1.1712 |
| 172K | 0.8997 | 0.8886 | 0.8886 | 0.8886 |
| cfd2 | 2.3176 | 1.3871 | 1.2229 | 1.1431 |
| hex256 | 0.9235 | 0.8566 | 0.6939 | 0.6939 |
| 178K | - | - | - | - |
| sf2 | - | - | - | - |
| AVERAGE | 1.3139 | 1.1445 | 1.0285 | 0.9578 |
| STDEV | 0.4106 | 0.2047 | 0.1721 | 0.1613 |

Table 5.7: Front sizes relative to the BEND order

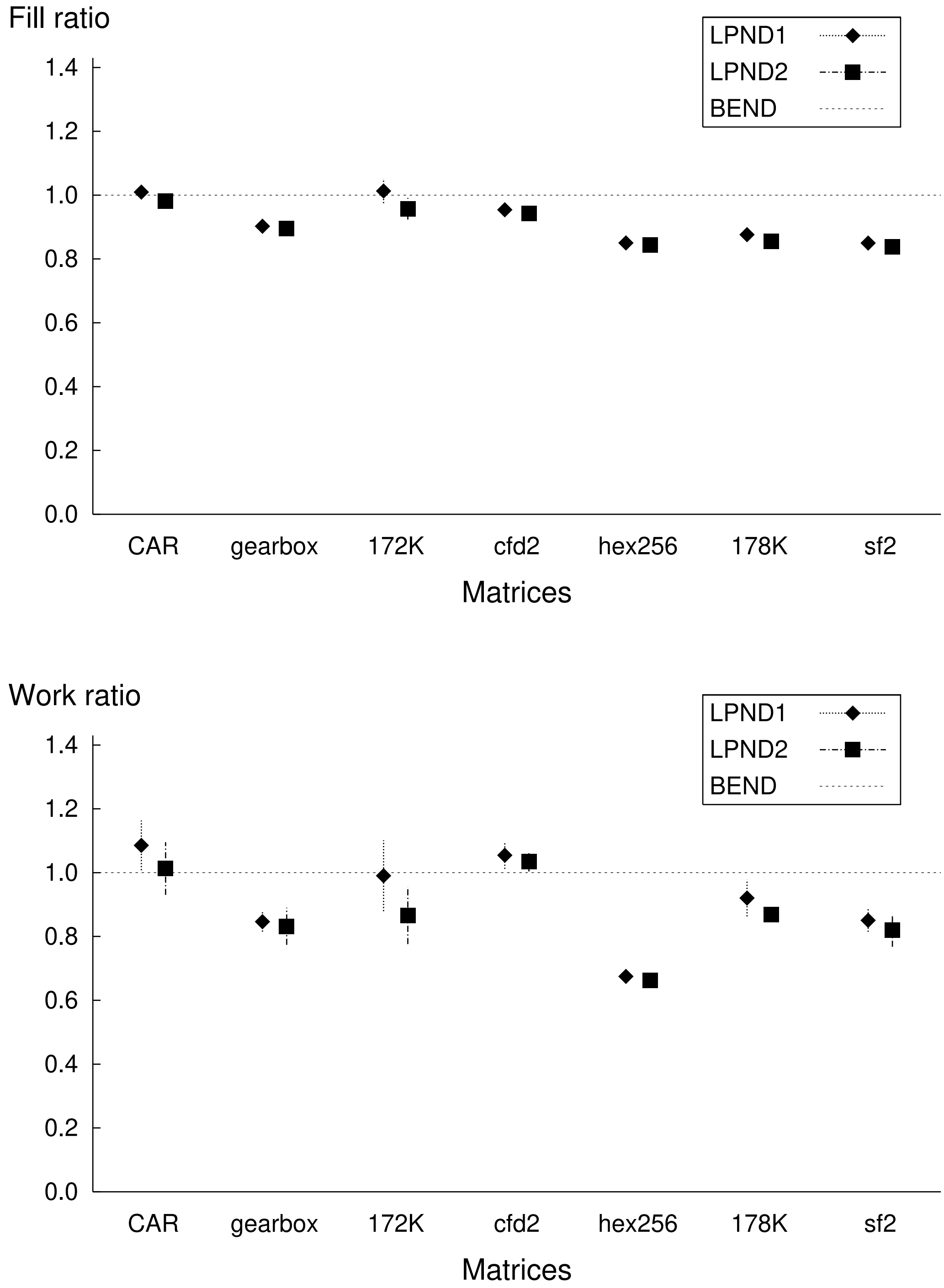



Figure 5.6: Average and standard deviation of 10 orders produced by the LPND algorithm with 1 and 2 calls to Chaco, relative to the orders produced by BEND.

# Chapter 6

# Final remarks

In this thesis we identify a trade-off between low-fill and low-height elimination orders and present two opposing ideas as to how to produce good elimination orders. We provide a parallelizing algorithm that takes an order and produces a parallel order with height within an $O(\log^2 n)$ factor of the optimal height of the chordal completion of the graph according to the original order. To obtain such parallel orders we incur some extra overhead in terms of both fill and work. We show that the parallel order obtained creates at most a constant factor more nonzeros and work than the original order.

On the other hand, we obtain an algorithm that produces low-fill orders that are fairly sequential. The sequential nature of the orders is the price we pay to obtain low fill. Even though this algorithm is based on nested dissection, we do not, as of yet, have any performance guarantees for this algorithm on general graphs. Our experiments show that the orders produced are competitive with the orders produced by the current-champion ordering algorithm (BEND). While the orders we produce usually require a little less fill, the BEND algorithm is usually faster in producing its orders. This indicates that if the BEND algorithm spent more time in producing its order, say, looking for better separators, it might produce even better orders.

While the first algorithm we presented tries to parallelize orders and the second one tries to produce fairly sequential ones, they both indicate that there exists a trade-off between producing low-fill elimination orders and producing elimination orders that are very parallel. It is interesting that, in order to produce parallel orders with only a constant factor more nonzeros than the chordal completion the algorithm starts with, an important step of the parallelizing algorithm was to introduce sentinels, which are nothing more than separators used to sequentialize the elimination of certain vertices in the graph. This is a compromise that allows us to produce very parallel orders while limiting the amount of fill introduced.

In a side remark, we observe that in our experiments, as well as in the experiments performed by Hendrickson and Rothberg in [HR96], a state-of-the-art minimum-degree algorithm performed poorly in comparison with nested dissection and the BEND algorithm. Our algorithm produced orders that on average required about half the amount of work, and about $3/4$-ths of the number of nonzeros the minimum-degree orders required.

A comparison of these ordering algorithms on square grids shows that the minimum-degree algorithm is the best on grids with large aspect ratio. These are graphs for which nested dissection and the BEND algorithm both produce very parallel orders, in contrast with the very sequential orders produced by minimum-degree. Minimum-degree orders can be computed in very little time. However, unlike the BEND algorithm, minimum-degree is unlikely to benefit much from any additional allotted running time. The comparison of the various ordering algorithms on graphs from a number of different areas reflects poorly on the quality of minimum-degree orders, and certainly justifies the additional time to compute better orders.

We list some areas that deserve further attention. Among other things, we would like to be able to obtain:

- a more comprehensive study of parallelism versus fill/work;
- tighter lower/upper bounds on the fill in chordal completions of grid graphs;
- non-trivial lower bounds on the size of the chordal completion of planar graphs and graphs in general;
- an analysis of the minimum-degree heuristic on chordal and interval graphs;
- ordering algorithms and analyses for asymmetric matrices.

# Bibliography


[ADD96] P. Amestoy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal of Matrix Analysis and Applications*, 17(4):886–905, 1996.

[AH94] B. Aspvall and P. Heggernes. Finding minimum height elimination trees for interval graphs in polynomial time. *BIT*, 34:484–509, 1994.

[AKR93] A. Agrawal, P. Klein, and R. Ravi. Cutting down on fill using nested dissection: provably good elimination orderings. In A. George, J. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 in the IMA Volumes in Mathematics and its Applications, pages 31–55. Springer–Verlag, 1993.

[AL96] C. Ashcraft and J. Liu. Robust ordering of sparse matrices using multisection. Technical Report ISSTECH-96-002, Boeing information and support services, 1996.

[Asp95] B. Aspvall. Minimizing elimination tree height can increase fill more than linearly. *Information Processing Letters*, 56:115–120, 1995.

[BG73] G. Birkhoff and A. George. Elimination by nested dissection. In J. F. Traub, editor, *Complexity of Sequential and Parallel Numerical Algorithms*, pages 221–269. Academic Press, 1973.

[BGHK95] Hans L. Bodlaender, John R. Gilbert, Hjálmtýr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18(2):238–255, March 1995.

[BS90] P. Berman and G. Schnitger. On the performance of the minimum degree ordering for Gaussian elimination. *SIAM Journal of Matrix Analysis and Applications*, 11(1):83–88, January 1990.

[Bun74] P. Buneman. A characterization of rigid circuit graphs. *Discrete Mathematics*, 9:205–212, 1974.

[Dav94] T. Davis. University of Florida sparse matrix collection. *http://www.cise.ufl.edu/~davis/sparse/*, 1994.

[Dir61] G. A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburgh*, 25:71–76, 1961.

[ENRS95] G. Even, J. Naor, S. Rao, and B. Schieber. Divide-and-conquer approximation algorithms via spreading metrics. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 62–71, October 1995.

[Gav74] F. Gavril. The intersection graph of subtrees of a tree are exactly the chordal graphs. *Journal of Combinatorial Theory B*, 16:47–56, 1974.

[Gea90] K. A. Gallivan and et al. *Parallel algorithms for matrix computations*. SIAM, 1990.

[Geo73] J. A. George. Nested dissection of a regular finite element mesh. *SIAM Journal of Numerical Analysis*, 10:345–363, 1973.

[Gil87] J. R. Gilbert. Some nested dissection order is nearly optimal. *Information Processing Letters*, 26:325–328, 1987.

[GL78] J. A. George and J. W. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal of Numerical Analysis*, 15:1053–1069, 1978.

[GT87] J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, 50:377–404, 1987.

[HL94] B. Hendrickson and R. Leland. The Chaco user's guide: Version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, 1994.

[HL95] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 International Conference on Supercomputing*, November 1995.

[HMR73] A. J. Hoffman, M. S. Martin, and D. J. Rose. Complexity bounds for regular finite difference and finite element grids. *SIAM Journal of Numerical Analysis*, 10:364–369, 1973.

[HR96] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. Technical Report SAND96-0868, Sandia National Labs, March 1996. To appear in SIAM J. Sci. Stat. Comput.

[HR97] B. Hendrickson and E. Rothberg. Effective sparse matrix ordering: Just around the BEND. In *Eight SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[IDL89] R. Grimes I. Duff and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 9:1–14, 1989.

[JK82] J. Jess and H. Kees. A data structure for parallel L/U decomposition. *IEEE Transactions on Computers*, 31:231–239, 1982.

[KK95] G. Karypis and V. Kumar. Metis. Unstructured graph partitioning and sparse matrix ordering system. *http://www.www.cs.umn.edu/~karypis/metis/metis.html*, 1995.

[Liu85] J. W. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 12:141–153, 1985.

[Liu89] J. W. Liu. Reordering sparse matrices for parallel elimination. *Parallel Computing*, 11:73–91, 1989.

[LL87] C. Leiserson and J. Lewis. Orderings for parallel sparse symmetric factorization. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 27–32. SIAM, Philadelphia, PA, 1987.

[LM89] J. W. Liu and A. Mirzaian. A linear reodering algorithm for parallel pivoting of chordal graphs. *SIAM Journal of Discrete Mathematics*, 2:100–107, 1989.

[LPP89] J. Lewis, B. Peyton, and A. Pothen. A fast algorithm for reordering sparse matrices for parallel factorization. *SIAM Journal on Scientific and Statistical Computing*, 10:1156–1173, 1989.

[LR88] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431. IEEE Computer Society Press, October 1988.

[LRT79] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis*, 16:346–358, 1979.

[LT79] R. J. Lipton and R. E. Tarjan. A planar separator theorem. *SIAM Journal of Applied Mathematics*, 36(2):177–189, April 1979.

[LT80] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9:615–627, 1980.

[Man91] F. Manne. An algorithm for computing an elimination tree of minimum height for a tree. Technical Report CS-91-59, University of Bergen, Norway, 1991.

[Mar57] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3:255–269, 1957.

[OS96] D. O'Hallaron and J. Shewchuk. Properties of a family of parallel finite element simulations. Technical Report CMU-CS-96-141, School of Computer Science, Carnegie Mellon University, 1996.

[Pan93] Victor Pan. Parallel solutions of sparse linear and path systems. In John Reif, editor, *Synthesis of Parallel Algorithms*, pages 621–678. Morgan Kaufmann, 1993.

[Par61] S. Parter. The use of linear graphs in Gaussian elimination. *SIAM review*, 3:364–369, 1961.

[Pot88] A. Pothen. The complexity of optimal elimination trees. Technical Report CS-88-16, Department of Computer Science, The Pennsylvania State University, 1988.

[PR85] Victor Pan and John Reif. Efficient parallel solution of linear systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 143–152, Providence,RI, May 1985. ACM.

[RAK91] R. Ravi, A. Agrawal, and P. Klein. Ordering problems approximated: single processor scheduling and interval graph completion. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, pages 751–762, July 1991.

[RE98] E. Rothberg and S. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal of Matrix Analysis and Applications*, 19(3):862–695, 1998.

[Ros70] D. Rose. Triangulated graphs and the elimination process. *J. Math. Anal. Appl.*, 32:597–609, 1970.

[RR98] S. Rao and A. W. Richa. New approximation techniques for some ordering problems. In *Proceedings of the 9th Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 211–218, January 1998.

[RTL76] D. Rose, R. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

[TW67] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *J. Proc. IEEE*, 55:1801–1809, 1967.

[Yan81] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal of Algebraic and Discrete Methods*, 2:77–79, 1981.