APPROXIMATELY OPTIMAL ELIMINATION ORDERINGS FOR SPARSE MATRICES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Wee-Liang Heng

August 1998

APPROXIMATELY OPTIMAL ELIMINATION ORDERINGS FOR SPARSE MATRICES

Wee-Liang Heng, Ph.D.
Cornell University 1998

Many problems in engineering and science require the solution of a sparse system of linear equations $Ax = b$, where A is a symmetric positive definite matrix. The system is typically solved via Gaussian elimination on A, which results in nonzero entries (known as *fill-in*) being introduced into the matrix. By a judicious reordering of the rows and columns of A, it is often possible to reduce the amount of fill-in greatly, and consequently, the space and time taken to solve the system.

Finding an ordering that minimizes fill-in is an NP-complete problem. In practice, heuristics are used to find good orderings. These heuristics, however, do not have any guarantees on the quality of the ordering produced. In contrast, by using a divide-and-conquer approach with approximately optimal balanced vertex separators, Agrawal, Klein and Ravi recently gave the first approximation algorithm for optimal elimination orderings.

This thesis describes an extensive computational study of the Agrawal-Klein-Ravi algorithm and its underlying subroutines, and also proposes and investigates improvements to that algorithm. We introduce a new cost measure for the separators used to decompose a problem, and also devise a novel way of forming the subproblems and combining the solutions to these subproblems. These new approaches are empirically compared to existing ones in the literature, and fare well against them. The computational bottleneck in all the divide-and-conquer approaches is the construction of approximately optimal balanced vertex separators. We demonstrate that substantial running time improvements can be obtained by aggressively optimizing the implementation of the construction.

Biographical Sketch

Wee-Liang Heng graduated *summa cum laude* from the University of California at Berkeley in 1990, in Computer Science and Mathematics. He received an M.Sc. from Berkeley a year-and-a-half later. Prior to joining Cornell in 1994, he worked on computer security in Singapore. Upon completion of his Ph.D., he will begin work at ESRI in Southern California.

To my wife, Stella, for her love, support and encouragement

Acknowledgements

I would like to thank God for His blessings, and for sustaining and watching over me.

I am grateful to my advisor Éva Tardos for her endless patience and guidance. Her confidence in me helped me through a difficult process. She was especially understanding and accommodating of my wish to graduate promptly, and was generous with advice at all times. Her deep insights and positive criticisms have been invaluable to my research. I cherish the opportunity of having worked with her.

Thanks also go to David Shmoys and Claire Cardie, for serving on my committee. Their constructive suggestions have been useful.

I leave with many happy memories of Ithaca and time spent with friends made here. I especially enjoyed the many stimulating conversations and fiercely fought chess matches with Marcos Aguilera. His scholarship and intellect have been inspirational, and I wish him all the best in his future endeavors. Thanks also to Siang-Huat Goh, Kim-Chuan Toh and Wei-Tsang Ooi for being excellent dining companions. Being with them evoked many wonderful memories of home. My officemates Brian Willem, Neal Glew and Nawaaz Ahmed shared in my struggles and joys, and were a constant source of comfort.

This research was partially supported by the ONR grants N00014-96-1-0050 and N00014-92-J-1989, and the NSF grants CCR-9700163 and DMI-9157199. Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not reflect the view of these agencies.

I would also like to thank my parents and family for their love and care. The lessons which my mother Moi-Hong Goh and my late father Choon-Thiam Heng taught me have helped me in my studies and will guide me through life.

Finally, I must express my deepest gratitude to my wife, Stella, for her constant encouragement and belief in me. It was difficult for us to be apart these few years, but she nonetheless kept faith in me and gave me the freedom to pursue my dream. This thesis would not have been possible without her sacrifices.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many problems in engineering and science require the solution of a sparse system of linear equations $Ax = b$, where A is a symmetric positive definite matrix. The system is typically solved via Gaussian elimination on A, which results in nonzero entries (known as *fill-in*) being introduced into the matrix. By a judicious reordering of the rows and columns of A, it is often possible to greatly reduce the amount of fill-in, and consequently the space and time taken to solve the system.

Finding an ordering that minimizes fill-in is an NP-complete problem. In practice, heuristics are used to find good orderings. These heuristics, however, do not have any guarantees on the quality of the ordering produced. In contrast, by using a divide-and-conquer approach with approximately optimal balanced vertex separators, Agrawal, Klein and Ravi [2] recently gave the first approximation algorithm for optimal elimination orderings. This thesis describes an extensive computational study of their algorithm and its underlying subroutines, and also proposes and investigates improvements to that algorithm.

This chapter is organized as follows. We first motivate and define the elimination ordering problem for sparse matrices in Section 1.1 and give examples of sparse matrices in Section 1.2. We then show how elimination can be formulated in graph theoretic terms in Section 1.3. In Section 1.4.1, we look at the most widely used heuristic for finding good elimination orderings. The class of nested dissection methods, which is the focus of this thesis, is introduced in Section 1.5, and the subroutines used are discussed in Section 1.6 and 1.7. Finally, we give an outline of the remainder of the thesis in Section 1.8, and establish in Section 1.9 some terminology that is frequently used in the thesis.

1.1 The Elimination Ordering Problem

1.1.1 Solution of Sparse Systems of Equations

The solution of a sparse system of linear equations

$$Ax = b$$

arises in many problems in engineering and science. For example, such systems appear in computational circuit design, linear programming, and the solution of partial differential equations ([77]). Here A is an n-by-n real symmetric positive definite[1] matrix where most of the entries are zero, and x and b are n-dimensional vectors. The system can be solved by performing Gaussian elimination on A and b, thereby reducing it to upper triangular form:

$$Ux = b'$$

where U is an upper triangular matrix with unit diagonal. Due to the positive definiteness of A, it is not necessary to pivot, i.e. interchange rows and/or columns, during the elimination to ensure numerical stability. The vector x is easily obtained by back substitution.

It is more typical however to factor A. It can be shown ([40]) that A has a unique factorization of the form $A = LDL^T$ where L is a lower triangular matrix with unit diagonal, D is a diagonal matrix with positive entries and L^T is the transpose of L. The factorization is related to Gaussian elimination. Indeed, the upper triangular matrix U above is identical to L^T. A closely related factorization is the *Cholesky* factorization $A = GG^T$, where G is a lower triangular matrix, and is equal to $LD^{1/2}$. With the factored form, we obtain x by backsolving in sequence $Lz = b$, $Dy = z$ and $L^T x = y$. This has the advantage of having to factor A only once in the case where the linear system is solved for many different right-hand side vectors b.

In general, the matrix L^T is not as sparse as the upper triangular half of A. During the process of elimination, the introduction of nonzeros, called *fill-in*, occurs. For example, consider the following symmetric positive definite matrix A and its associated factor L^T:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & & & & & & \\ 1 & & 4 & & & & & \\ 1 & & & 8 & & & & \\ 1 & & & & 16 & & & \\ 1 & & & & & 32 & & \\ 1 & & & & & & 64 & \\ 1 & & & & & & & 65 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & 1 & -1 & -1 & -1 & -1 & -1 & -1 \\ & & 1 & -1 & -1 & -1 & -1 & -1 \\ & & & 1 & -1 & -1 & -1 & -1 \\ & & & & 1 & -1 & -1 & -1 \\ & & & & & 1 & -1 & -1 \\ & & & & & & 1 & -1 \\ & & & & & & & 1 \end{bmatrix}$$

We see that L^T is a full upper triangular matrix.

However, since the matrix A is symmetric positive definite, the same is true if we reorder its rows and columns in a symmetric fashion. By applying a judicious ordering prior to factorization, we can often reduce the fill-in. Returning to our

[1] There are several equivalent definitions of positive definiteness [40]; the simplest is that $z^T A z > 0$ for all n-dimensional vectors z.

example, if we first reverse the ordering of the rows and columns of A, then *no* fill-in occurs during elimination:

$$\begin{bmatrix} 65 & & & & & & & 1 \\ & 64 & & & & & & 1 \\ & & 32 & & & & & 1 \\ & & & 16 & & & & 1 \\ & & & & 8 & & & 1 \\ & & & & & 4 & & 1 \\ & & & & & & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & & & & & & & \frac{1}{65} \\ & 1 & & & & & & \frac{1}{64} \\ & & 1 & & & & & \frac{1}{32} \\ & & & 1 & & & & \frac{1}{16} \\ & & & & 1 & & & \frac{1}{8} \\ & & & & & 1 & & \frac{1}{4} \\ & & & & & & 1 & \frac{1}{2} \\ & & & & & & & 1 \end{bmatrix}$$

Iterative Versus Direct Methods

Instead of using Gaussian elimination for solving a linear system of equations $Ax = b$, one can use an iterative approach. A typical iterative method starts with an approximation $x^{(0)}$ to the desired solution x, and computes a sequence of iterates $x^{(1)}$, $x^{(2)}$, ..., which converges to x. In theory, an infinite number of iterations is required to obtain x, but in practice, one uses some termination criterion to determine when the current iterate is sufficiently close enough to x. On the other hand, in the absence of roundoff errors, direct methods, such as Gaussian elimination, produce the solution to the system after a finite number of arithmetic operations.

The question as to which class of methods is better cannot be answered in general, and depends strongly on the problems to be solved. We refer the reader to [33] for a comparison of iterative and direct methods,

1.1.2 Definition of the Elimination Ordering Problem

We see from the example in the previous section that the ordering of the rows and columns of a matrix can have a drastic effect on the amount of fill-in. Fill-in is important because it affects both the amount of storage and the number of arithmetic operations needed for the factorization of a matrix. We therefore consider in this thesis the following problem, which has been the focus of considerable research in the field of sparse matrix computations:

> *The Elimination Ordering Problem.* Find an ordering of the rows and columns of the matrix A that yields a factorization with low fill-in.

In matrix terms, we are interested in finding a permutation P of the rows (and columns) of A such that the factorization of PAP^T has low fill-in. Premultiplying A by P has the effect of reordering its rows, while postmultiplying it by P^T reorders its columns.

In the analysis of the quality of an elimination ordering, it is typical to consider *fill* instead of fill-in. The fill generated by the factorization of PAP^T is defined to be the nonzero entries of L, and so the amount of fill measures the storage requirement for the factorization. Note that the fill is simply the nonzero entries of PAP^T together with the fill-in.

Besides trying to minimize the fill/fill-in, another common criterion for a good elimination ordering is that it directly minimizes the number of arithmetic operations performed during factorization. The operation count is easily expressed in terms of the nonzero structure of the columns of L. Let d_i denote the number of nonzero entries in the i^{th} column of L that are below the diagonal. Regarding division and subtraction as multiplication and addition operations respectively, we have the following operation count.

Theorem 1.1 (Rose [74]) *The LDL^T decomposition requires $\sum_{i=1}^{n-1} d_i(d_i+3)/2$ multiplication operations and $\sum_{i=1}^{n-1} d_i(d_i+1)/2$ addition operations.*

A third performance criterion is *elimination height*, which is the number of steps needed in parallel Gaussian elimination. In a typical parallel implementation, we eliminate multiple variables at each step. These variables must be independent for the elimination step to be valid, i.e. the variables must not be related by any equation in the system. We will precisely define elimination height in Section 1.3.

We now examine more closely the assumptions underlying the elimination ordering problem. The first is the symmetric positive definiteness of the matrix A. Since A is symmetric positive definite, the same is true of PAP^T for any permutation P. Thus during the factorization of PAP^T, no pivoting, i.e. row and/or column interchanges, is needed to maintain numerical stability against roundoff errors ([40]). The second assumption is that exact numerical cancellation does not occur. Specifically, we assume that any nonzero entries, including those in the matrix PAP^T and the fill-in that occurs, remain nonzero in the course of factorization. This assumption is reasonable since in actual computation, we use finite rather than exact real arithmetic.

With the noncancellation assumption and the absence of pivoting during factorization, we see that the sparsity structure of L is completely determined by the sparsity structure of A. So we can predict where fill-in will occur independently of the actual numerical values of the nonzero entries of A. The result also holds in the case of nonsymmetric sparse matrices, which may or may not have symmetric sparsity structure, but for which pivoting during factorization is not needed to maintain numerical stability. In that case however, there is a need to store both the lower and upper triangular halves of the factorization since they are not related. We observe also that if numerical cancellation *did* occur during factorization, then our prediction of fill-in is still valid but is pessismistic: any fill-in is limited to the predicted entries, but some of those entries may actually be zero.

In contrast, for general sparse systems, we cannot predict where fill-in occurs, since pivoting is dynamically performed during elimination, and the choice of pivots depends on the numerical values of the matrix entries. We can in fact think of the permutation P as a *static* pivoting that we apply to reduce fill-in. In general, pivoting to control fill-in and pivoting to maintain numerical stability are competing objectives; for symmetric positive definite systems, the conflict does not arise as there is no need for the latter use of pivoting.

1.1.3 Cholesky Factorization in Practice

From the previous discussion, we see that the solution process for the symmetric positive definite system $Ax = b$ can be conveniently divided into the following four steps. We work exclusively with the Cholesky factorization in this section.

1. Find a good ordering P.
2. (Symbolic factorization) Determine the nonzero positions in G for the Cholesky factorization $PAP^T = GG^T$.
3. (Numerical factorization) Set up static storage for G and compute it.
4. (Back solving)[2] Solve $Gy = Pb$ and $G^T(Px) = y$.

Separating the factorization process into two steps (symbolic and numerical) allows us to use a *static* storage scheme during numerical factorization. Without this prior knowledge of the fill structure for G, we would have to dynamically allocate storage for G during numerical factorization. Such dynamically allocated storage schemes, however, have very high book-keeping overheads, which have asymptotic time bounds greater than that for the arithmetic operations performed during numerical factorization. Note that the symbolic factorization can be efficiently computed in time $O(m)$, where m is the number of nonzero entries in G. In addition, the back solution in step 4 is easily performed in $O(n^2)$ time, where n is the order of the system. As for computing the ordering, the time taken depends on the ordering algorithm used. For the ordering heuristics described in Section 1.4, the ordering can be computed in a small fraction of the time taken for numerical factorization. In contrast, the theoretically motivated divide-and-conquer ordering algorithm in Section 1.5.3 is computationally intensive, and its computation time overwhelms that used for numerical factorization.

In many instances, such as in structural analysis or in the simulation of some time-dependent phenonema, the system $Ax = b$ is solved for many different right-hand side vectors b, in which case the cost of the first three steps can be amortized over the many solutions. Similarly, if the system is repeatedly solved for different matrices A having the *same* sparsity structure but not the same numerical entries, then steps 1 and 2 need only be performed once as a preprocessing step.

There are several approaches to computing the Cholesky factorization in practice. We will not consider these approaches or their implementations in this thesis. The interested reader may refer to the books by George and Liu [33] and by Duff

[2]Since P and P^T are inverses of each other, we can write the permuted system as $(PAP^T)(Px) = Pb$, or in factored form, $(GG^T)(Px) = Pb$.

et al [20], and to the paper by Liu [66] for sequential implementations and the survey article by Heath *et al* [43] on parallel Cholesky factorization.

We also point out that in high-performance implementations of Cholesky factorization on modern machine architectures, such as shared memory architectures, there are other considerations besides fill, operation count and elimination height. An important concern is the nonzero block structure of the matrices, and how the sizes of those blocks relate to the memory hierarchy, in particular, to the sizes of the memory caches ([16]). We do not consider these challenging but somewhat machine dependent issues in this thesis, but restrict our focus to the classical concerns of fill, operation count and elimination height.

1.2 Examples of Sparse Matrices

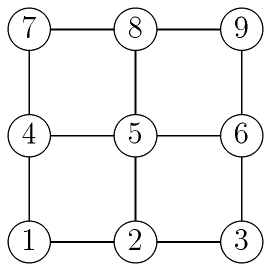
1.2.1 The Harwell-Boeing Sparse Matrix Collection

The Harwell-Boeing sparse matrix collection [18] [19] is a set of standard test matrices arising in the solution of linear systems, and in least squares and eigenvalue problems. The matrices come from actual applications in more than 20 scientific and engineering disciplines, including structural analysis, partial differential equations, finite element analysis and econometrics, and they have been used to evaluate elimination ordering algorithms ([64] [63] [2]).

The collection comprises both symmetric and nonsymmetric matrices, and they range in order from 9 to 44,609. Although these matrices are small by modern computing standards, in which matrices with millions of rows and columns are typical, they suffice for the theoretically motivated but computationally intensive elimination ordering algorithms that we empirically evaluate in this thesis.

1.2.2 An Example of a Finite Element Matrix

As an important illustration of how sparse matrices arise, we consider a (very) simplified two-dimensional finite element mesh problem on a regular square grid. We divide the unit square $[0,1] \times [0,1]$ into h^2 small squares, each of side length $1/h$, as shown below for $h = 2$.



Each of the small squares is a *finite element*. We can place *finite element nodes* on the "corners," edges or interior of each element, and we can associate to each node one or more variables x_i. In the above example, we have nodes at each "corner" and one (numbered) variable associated with each node. From a finite element mesh M, we derive a finite element matrix A of order equal to the number of variables as follows. The matrix entry a_{ij} is nonzero if and only if the variables x_i and x_j are associated with nodes of the same finite element. For the simple 3×3 grid above, the nonzero structure of the corresponding matrix using the given variable numbering is shown below:

$$\begin{array}{cc}
 & \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} &
\left[\begin{array}{ccccccccc}
\star & \star & & \star & \star & & & & \\
\star & \star & \star & \star & \star & \star & & & \\
 & \star & \star & & \star & \star & & & \\
\star & \star & & \star & \star & & \star & \star & \\
\star & \star & \star & \star & \star & \star & \star & \star & \star \\
 & \star & \star & & \star & \star & & \star & \star \\
 & & & \star & \star & & \star & \star & \\
 & & & \star & \star & \star & \star & \star & \star \\
 & & & & \star & \star & & \star & \star
\end{array}\right]
\end{array}.$$

Although the matrix for the small example does not look sparse, it should be clear that the matrices for much larger grids would be sparse, since each finite element node belongs only to a small number of finite elements. In general finite element problems, the mesh structures used may be in three dimensions and are much more complicated and less regular than the square grid above. We refer the reader to the book by Zienkiewicz and Taylor [90] for a detailed treatment of finite element methods.

1.3 A Graph Theoretic Formulation of Elimination

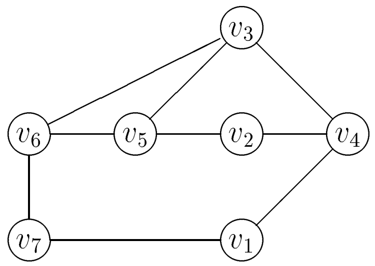
We illustrate in this section a formulation of Gaussian elimination in graph theoretic terms. The formulation provides a convenient way for describing elimination ordering algorithms and also gives insight into the elimination process. The use of graph theory to study elimination was first suggested by Parter [70], who analyzed the special case when the associated graph is a tree. A detailed study was conducted by Rose [74], who gave a graph-theoretic account of Gaussian elimination and showed the equivalence between matrices which possess an ordering that generates no fill-in and chordal graphs, a class of widely studied graphs.

Let $A = (a_{ij})_{1 \leq i,j \leq n}$ be an $n \times n$ symmetric (positive definite) matrix. We can naturally associate an undirected graph $G = (V, E)$ where $V = \{v_1, \ldots, v_n\}$ with A as follows: $v_i v_j$ is an edge of G if and only if $i \neq j$ and $a_{ij} \neq 0$. We call G the *graph* of A. Observe that the graph does not capture the diagonal entries of A. This is not necessary as these entries are always positive and remain so throughout elimination, because of the positive definiteness of the matrix.

A *vertex ordering* of G is a one-to-one mapping $\pi : V \leftrightarrow \{1, \ldots, n\}$. We denote by $G_\pi = (V, E, \pi)$ the ordered graph associated with π. Each ordered graph corresponds to a permuted version of A. In fact G_π corresponds to PAP^T where the permutation matrix P has a one in row $\pi(v_i)$ of column i, and zeros elsewhere.

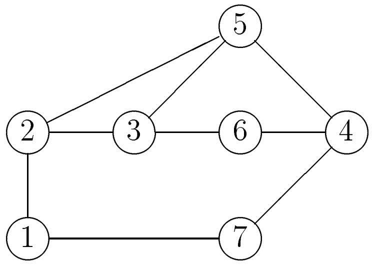
To motivate the graph-theoretic interpretation of the elimination process, we consider the following 7×7 matrix A whose nonzero structure and associated graph are shown.

$$A = \begin{bmatrix} \star & & & \star & & & \star \\ & \star & & \star & \star & & \\ & & \star & \star & \star & \star & \\ \star & \star & \star & \star & & & \\ & \star & \star & & \star & \star & \\ & & \star & & \star & \star & \star \\ \star & & & & & \star & \star \end{bmatrix}$$



Consider the permutation P which reverses the ordering of the rows and columns of A. This gives an ordering π of the vertices in the sequence $v_7, v_6, \ldots, v_1$. In the diagram below, we show the permuted matrix and also the graph G^1 where each vertex is now referred to by its ordering number.

$$PAP^T = \begin{bmatrix} \star & \star & & & & & \star \\ \star & \star & \star & & \star & & \\ & \star & \star & & \star & \star & \\ & & & \star & \star & \star & \star \\ & \star & \star & \star & \star & & \\ & & \star & \star & & \star & \\ \star & & & \star & & & \star \end{bmatrix}$$

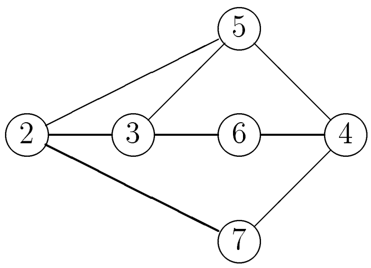


Graph G^1

The first step of elimination involves subtracting multiples of the first row of the (permuted) matrix from rows 2 and 7, so as to make the entries below the diagonal in column 1 all zero. Since row and column 1 are no longer of interest, we show below rows/columns 2 through 7 of the resulting matrix, where the circled entries are the fill-in. The graph G^2 for the submatrix is also given.

$$\begin{array}{c c} & \begin{array}{cccccc} 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{array}{c} 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} & \begin{bmatrix} \star & \star & & \star & & \otimes \\ \star & \star & & \star & \star & \\ & & \star & \star & \star & \star \\ \star & \star & \star & \star & & \\ & \star & \star & & \star & \\ \otimes & & \star & & & \star \end{bmatrix} \end{array}$$

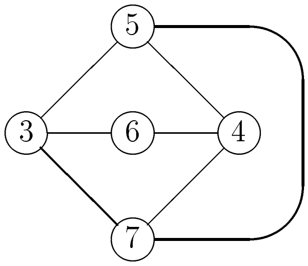
After 1 step



Graph G^2

We see that G^2 is obtained from G^1 by the addition of the edge $2-7$, which corresponds to the fill-in in positions (2,7) and (7,2), and by the deletion of the vertex 1 and its incident edges. Note that vertices 2 and 7 are the only neighbors of vertex 1 in G^1, and that in going from G^1 to G^2, we have made the neighbors of vertex 1 into a clique. In the second step of elimination, we subtract multiples of the second row from the remaining rows. The diagram below shows the resulting submatrix for rows/columns 3 through 7 and its graph G^3.

$$\begin{array}{c c}
 & \begin{array}{ccccc} 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{array}{c} 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} &
\left[\begin{array}{ccccc}
\star & & \star & \star & \otimes \\
 & \star & \star & \star & \star \\
\star & \star & \star & & \otimes \\
\star & \star & & \star & \\
\otimes & \star & \otimes & & \star
\end{array}\right]
\end{array}$$



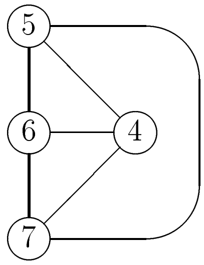
After 2 steps

Graph G^3

Now, G^3 differs from G^2 by the addition of the edges $3-7$ and $5-7$, and by the deletion of vertex 2 and its incident edges. The vertices 3, 5 and 7 are neighbors of vertex 2 in G^2, and we have turned them into a clique.

To reinforce the idea, we proceed with the third step of elimination.

$$\begin{array}{c c}
 & \begin{array}{cccc} 4 & 5 & 6 & 7 \end{array} \\
\begin{array}{c} 4 \\ 5 \\ 6 \\ 7 \end{array} &
\left[\begin{array}{cccc}
\star & \star & \star & \star \\
\star & \star & \otimes & \star \\
\star & \otimes & \star & \otimes \\
\star & \star & \otimes & \star
\end{array}\right]
\end{array}$$



After 3 steps

Graph G^4

Vertex 3 has three neighbors (vertices 5, 6 and 7) in G^3 and they now form a clique in G^4.

The above example suggests the following definition. For an ordered graph G_π, we define the sequence of elimination graphs $G_\pi^1, \ldots, G_\pi^n$, where G_π^i is a graph on the vertices numbered i and higher, recursively as follows. The graph G_π^1 is just G, while G_π^{i+1} is obtained from G_π^i by adding edges to make the neighbors of $\pi^{-1}(i)$ in G_π^i into a clique, and by the deletion of $\pi^{-1}(i)$ and its incident edges. The edges introduced into G_π^{i+1} correspond to the fill-in from step i of elimination.

We define the *filled elimination graph* G_π^* for the ordering π to be the graph on V whose set of edges is the union of those in the elimination graphs $G_\pi^1, \ldots, G_\pi^n$. Note that G_π^* describes exactly the nonzero entries below the diagonal in the factor

L for the factorization $PAP^T = LDL^T$, where P is the permutation of the matrix A corresponding to π.

Using G_π^*, we can precisely define the *elimination height* associated with the ordering π, which is the number of steps needed in a parallel elimination of the permuted matrix PAP^T. Consider two vertices v and w. If these vertices are adjacent in G_π^*, then the higher numbered vertex cannot be eliminated before the lower numbered one. So if we direct the edges in G_π^* from lower numbered vertices to higher numbered ones, the elimination height is the number of vertices in a longest directed path. An alternative characterization of elimination height using *elimination trees*, which are structures that express the dependencies among the columns of L, is given in [65].

The problem of finding an elimination ordering with minimum fill (or fill-in) is easily stated in the graph theoretic formulation. We say that α is a *minimum elimination ordering* if the number of edges in G_α^* is no more than the number of edges in G_π^* for any ordering π of G. Yannakakis [89] showed that the problem of finding a minimum elimination ordering is in fact NP-complete. The graph theoretic model is also easily extended to the case of nonsymmetric Gaussian elimination without row or column interchanges by using a directed graph to represent the nonzero entries in the nonsymmetric matrix. Rose and Tarjan [75] examined algorithmic aspects of elimination on directed graphs, and they also showed that the corresponding minimum elimination ordering problem is NP-complete.

1.4 Heuristics for Good Elimination Orderings

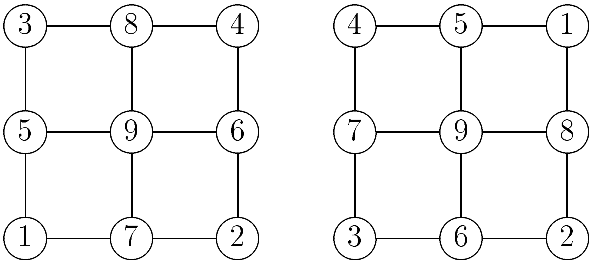
In solving the sparse system $Ax = b$, we exploit the presence of zero entries in the coefficient matrix A by avoiding any computation with them. This leads to two classes of sparse elimination methods: the general sparse methods which exploit all the zero entries of A, and the band or envelope methods, which exploit only those zeros outside a set of positions of A. In Section 1.4.1, we present the most widely used approach to finding good elimination orderings for general sparse methods.

Although band and envelope methods are not as efficient as general sparse methods, they are attractive because the programs and data structures used for such methods are relatively simpler than general schemes, and the storage and computational overhead is also much lower. We refer the reader to [33] for a discussion of the Cuthill-McKee and the reverse Cuthill-McKee algorithms, two classical ordering algorithms that attempt to reduce the envelope of the matrix.

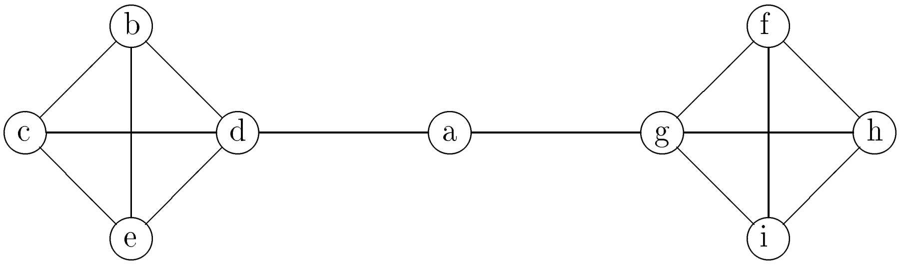
1.4.1 The Minimum Degree Algorithm

The *minimum degree algorithm* is the most widely used general purpose heuristic for finding an elimination ordering, and has been utilized in several sparse matrix packages, including the Harwell MA27 code [21], the Waterloo SPARSPAK package [35] and the Yale sparse matrix package [22]. Its implementation has been extensively studied, and there have been many important and effective enhancements made ([34]); the time taken to compute the ordering is typically only a few percent of the time required in factorization. The variant based on *multiple elimination* by Liu [63] yields the fastest implementation to date.

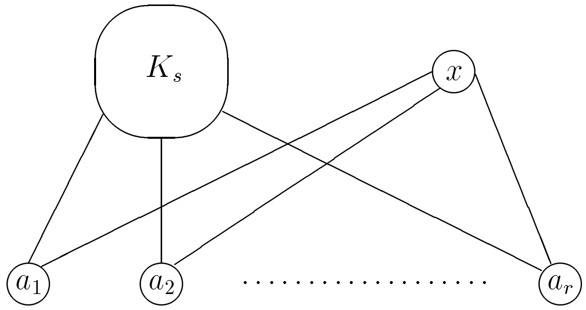
At each step, the algorithm selects a vertex of minimum degree in the elimination graph and eliminates it. Ties are resolved arbitrarily when there is more than one vertex of minimum degree. For example, two orderings produced by the algorithm for the 3×3 grid described in Section 1.2.2 are shown below.



In practice the heuristic gives good orderings, although it lacks any theoretical guarantees. Rose [74] gives two pathological cases for the minimum degree algorithm. For the graph below, the algorithm selects a for elimination and creates an edge between d and g. However the graph has an elimination ordering with no fill-in since it belongs to a special class of graphs known as *chordal* graphs, which have been shown to possess such special orderings [74]. We will examine the relationship between elimination and chordal graphs in Section 4.1.



The second example shows that the minimum degree algorithm can be arbitrarily bad for graphs that do not possess a no fill-in ordering. In the graph below, $2 \leq r \leq s$ and K_s is a clique on s vertices. Each of the vertices a_i is adjacent to every vertex of K_s and x. Vertex x has minimum degree $(= r)$, and eliminating it produces an elimination graph which is a clique on $r + s$ vertices. The ordering produced by the minimum degree algorithm therefore introduces $\frac{1}{2}r(r-1)$ edges. However, the ordering obtained by numbering the a_i first creates only s edges.



In [34], George and Liu performed an experimental study of the effect of tie-breaking on the performance of the minimum degree algorithm. They used the finite element matrix associated with the 180×180 grid; this matrix has order 32,400 and has 128,522 nonzero entries below the diagonal. By performing many runs with ties randomly broken, they found a 50% variation in the amount of fill-in and more than 100% variation in the operation count for the factorizations associated with the orderings produced. The performance of the minimum degree algorithm is therefore surprisingly sensitive to the tie-breaking strategy employed. The multiple elimination scheme of Liu [63] provides a form of tie-breaking by eliminating in each step a *maximal* set of minimum degree nodes.

A related but not as widely used heuristic is the *minimum fill-in* (also called *minimum deficiency*) algorithm ([86]) in which at each elimination step, the vertex which gives the minimum fill-in is chosen. As in the case of the minimum degree algorithm, pathological cases exist for the minimum fill-in algorithm (Rose [74]).

1.5 Nested Dissection

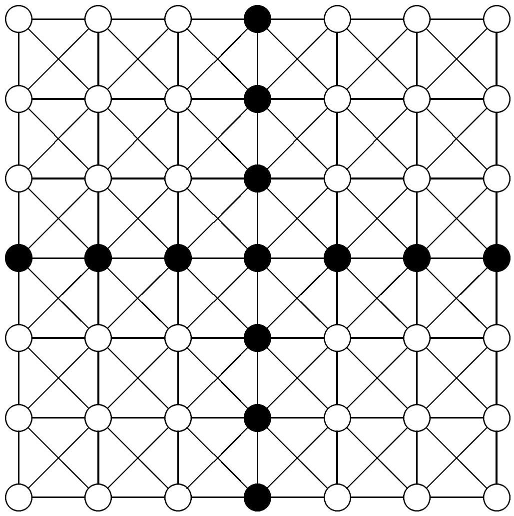
The ordering algorithms that we described in the previous section are heuristic in nature, with no theoretical guarantees on the quality of the ordering produced. In contrast, the method of nested dissection that we shall look at in this section has been shown to be asymptotically optimal on certain classes of graphs. The optimality guarantees are on the fill and operation count associated with the ordering produced. We first examine the motivating case of a square grid in Section 1.5.1 and then state some results for nested dissection on special classes of graphs in Section 1.5.2. Finally we present in Section 1.5.3 the variant given by Agrawal, Klein and Ravi [2], which is the first approximation algorithm for optimal elimination orderings, and which forms the focus of the computational study in this thesis.

1.5.1 Nested Dissection on Square Grids

The method of *nested dissection* was introduced by George [31] in the context of the regular finite element mesh described in Section 1.2.2. Recall that we subdivide the unit square $[0,1] \times [0,1]$ into h^2 small squares, each of which has side length

$1/h$, and that finite element nodes are placed on each "corner." To each finite element node we associate a variable x_i.

The finite element matrix associated with the mesh has a nonzero entry in position (i, j) if x_i and x_j are variables associated with nodes of the same small square. A moment's thought shows that the graph of the finite element matrix is the original mesh where each node is a vertex and in which the diagonals of the small squares have been drawn in. For example, the graph of the 7×7 grid is shown below:



In the above graph, we have shaded the nodes lying on the central horizontal and vertical "axes." Note that the removal of these nodes disconnects the graph into four connected quadrants. Consider a vertex ordering where the shaded vertices are numbered last, and let v and w be vertices in different quadrants. Note that these two vertices are not adjacent. From the graph theoretic interpretation of elimination in Section 1.3, we see that as the elimination proceeds, the vertices v and w become joined to other vertices in their respective quadrants and shaded boundaries. The only common neighbors that v and w can have are the shaded vertices on their common boundary. Since these shaded vertices are eliminated *after* v and w, there will never be any edge introduced between v and w during the elimination process. In essence, the shaded vertices prevent any interaction between vertices in different quadrants, and the elimination process on each of the quadrants proceeds *independently.* By recursively applying the idea on the quadrants, we obtain a dissection of the original mesh which is nested. A slightly different and inferior ordering in the case where more than one variable is associated with each finite element node is given in [30].

George's nested dissection ordering allows the Cholesky factorization to be computed in $O(n^{3/2})$ time and $O(n \log n)$ space, where n is the order of the finite element matrix. The ordering is asymptotically optimal in that symmetric factorization requires $\Omega(n^{3/2})$ time for any ordering of the finite element nodes. In contrast, the usual row-by-row banded numbering scheme results in a factorization requiring $O(n^2)$ time and $O(n^{3/2})$ space.

1.5.2 Nested Dissection on Special Classes of Graphs

The set of shaded nodes in the previous section is known as a *vertex separator*, which is a set of vertices whose removal disconnects the graph into two (or more) components.[3] On certain classes of graphs, there are separator theorems which guarantee the existence of a separator of a certain size, and whose removal divides the graph approximately in half. For example, in a planar graph on n vertices, a separator of size at most $2\sqrt{2n}$ can be found whose removal disconnects the graph into two pieces, each of which has no more than $\frac{2}{3}n$ vertices ([62]). Using that separator theorem, Gilbert and Tarjan [39] analyzed the performance of the nested dissection algorithm on planar graphs, and showed that the ordering gave $O(n \log n)$ fill and $O(n^{3/2})$ operation count. They also proved identical fill and operation count bounds for two-dimensional finite element graphs.

In [61], Lipton, Rose and Tarjan generalized George's nested dissection algorithm by requiring that the vertices in the separator be *included* in the recursive subproblems that are numbered. For their variant of the nested dissection algorithm, Lipton, Rose and Tarjan proved asymptotic bounds identical to those of Gilbert and Tarjan in the case of planar graphs. They also gave fill and operation count bounds for other classes of graphs with separator theorems.

We will give a fuller treatment of the results stated here in Chapter 4.

1.5.3 Nested Dissection on General Graphs

The results of Gilbert and Tarjan [39] and Lipton, Rose and Tarjan [61] in the previous section are restricted to graphs with special separator structures. In general, not all graphs are guaranteed to have small balanced separators. However, Leighton and Rao [57] have given a construction for balanced separators, i.e. separators whose removal divide the graph roughly in half, such that the size of the separators is guaranteed to be no more than a logarithmic factor larger than the smallest balanced separator. We will consider this construction in the next section. By using the approximately optimal balanced separators from the Leighton-Rao construction in George's nested dissection algorithm, Agrawal, Klein and Ravi [2] obtained an algorithm that is applicable to general graphs, and for which they were able to prove performance guarantees. For concreteness, we outline the Agrawal-Klein-Ravi algorithm below.

> *Agrawal-Klein-Ravi Ordering Algorithm* ([2]). Let $G = (V, E)$ be a graph with n vertices that we wish to number in the range $[a, b]$, where $b = a + n - 1$. If $n = 1$, we number the single vertex a. Other-

[3] A related concept is an *edge separator*, which is a set of edges whose removal disconnects the graph; we work exclusively with vertex separators in this thesis.

wise, we find an approximate $\frac{2}{3}$-balanced[4] vertex separator X using the Leighton-Rao construction. We number the vertices in the separator arbitrarily from $b-|X|+1$ to b. The rest of the vertices are numbered as follows. Let $G-X$ have k connected components $A_1, \ldots, A_k$ with $n_1, \ldots, n_k$ vertices. We recursively number the graph A_i in the range $[a+\sum_{j=1}^{i-1} n_j,\ a-1+\sum_{j=1}^{i} n_j]$ for each $i=1,\ldots,k$.

In contrast to earlier nested dissection algorithms, the analyses that Agrawal, Klein and Ravi gave for the quality of their ordering were with respect to the best possible ordering. They showed that for graphs of bounded degree the fill, multiplication count and elimination height of their ordering is within a polylogarithmic factor of the corresponding minimum quantities. They also gave a fill bound for graphs of unbounded degree. We state their results on bounded-degree graphs below.

Theorem 1.2 (Agrawal, Klein & Ravi [2]) *The elimination ordering found by the Agrawal-Klein-Ravi algorithm simultaneously minimizes elimination height to within an $O(\log^2 n)$ factor, fill to within an $O(\sqrt{d}\log^4 n)$ factor, and the operation count to within an $O(d\log^6 n)$ factor of the respective optimal quantities. In the guarantee, d denotes the maximum number of nonzero entries in any row or column of the $n \times n$ coefficient matrix.*

The same algorithm for finding an elimination ordering with approximately optimal elimination height was also given independently by Bodlaender *et al* [10], but no analysis of the performance bound on the fill or operation count was given there.

Recently, Natanzon, Shamir and Sharan [24] gave an approximation algorithm for the minimum *fill-in* problem. Their algorithm produces an ordering whose fill-in is at most $8f^2$, where f is the minimum fill-in for the given graph. The Natanzon *et al* algorithm is incomparable to the Agrawal *et al* algorithm. On graphs with low minimum fill-in, the Natanzon *et al* ordering is clearly better in terms of fill (or fill-in), while on graphs with high minimum fill-in, the Agrawal *et al* ordering is superior.

1.6 Construction of Balanced Separators

Approximately optimal balanced vertex separators play the key role in the elimination ordering algorithm of Agrawal, Klein and Ravi that we described in the previous section. Their usefulness, however, is extensive and not just limited to finding good elimination orderings. In general, such separators are desirable in divide-and-conquer algorithms as their "balancedness" ensures that the recursive subproblems on the remaining components are of roughly the same size. At the same time, the approximate optimality of such a separator controls the extent

[4] We will define the balance of a separator in Section 1.6.

of interaction between it and the subproblems. This separator-subproblem interaction usually introduces additional cost when solutions to the subproblems are combined together to form a solution to the original problem. Leighton and Rao [57] were the first to give a construction for approximately optimal separators (both edge and vertex versions). Since then, they and many other researchers have used these separators to develop polylogarithmic-times-optimal recursive approximation algorithms for many graph problems, including minimum VLSI area layout, minimum cut linear arrangement, minimum feedback arc set [58], optimal linear and higher-dimensional arrangement [42], and single-processor scheduling [73].

We will consider in detail in Chapter 3 the Leighton-Rao balanced separator (also known as balanced cut[5]) construction. In this section, we give an overview of the construction and its underlying subroutines, and also state its performance guarantees. To do so, we first describe the balanced cut problem. We define a vertex cut of a graph G to be a 3-way partition (A, B, C) of the vertex set, with the property that there are no edges between vertices in A and vertices in B. C is the set of cut vertices, and its removal divides up the graph into the subgraphs induced by the vertices in A and by the vertices in B. Let n be the number of vertices in the graph. We say that the cut is β-balanced, for some balance factor $\beta \geq \frac{1}{2}$, if $\max(|A|, |B|) \leq \beta n$, i.e. the larger piece contains at most βn vertices.[6] The cost of the cut is $c(C)$, where c is some capacity function on the vertices of the graph. An *optimal* β-balanced cut is a β-balanced cut of lowest cost.

The problem of finding an optimal β-balanced cut is NP-complete, even when all the vertices have unit capacity [57]. The balanced cut algorithm of Leighton and Rao [57] constructs a β-balanced cut whose cost is approximately optimal. The cost of the constructed cut is however not measured relative to the cost of an optimal β-balanced cut, but is instead compared to that of an optimal cut for a lower balance factor.

Theorem 1.3 (Leighton & Rao [57]) *Let $\beta > \beta' \geq \frac{1}{2}$ and $\beta \geq \frac{2}{3}$. There exists a polynomial time algorithm that finds a β-balanced vertex cut whose cost is within an $O(\log n)\frac{1}{\beta - \beta'}$ factor of the cost of an optimal β'-balanced vertex cut, where n is the number of vertices in the graph.*

It is an open question as to whether there exists a polynomial time approximation algorithm which finds a β-balanced cut whose cost is within an $O(\log n)$ factor of that of an optimal β-balanced cut.

The cut problem is closely related to an all-pairs multicommodity flow problem as follows. Suppose we wish to send a common amount of flow between each pair of vertices in the graph, subject to the total flow through a vertex not exceeding its capacity.[7] Then the total amount of flow through the vertices in a cut depends roughly on the sizes of the two components, and is limited by the capacity of

[5]We will clarify in Section 1.9 our usage of the interchangeable terms separators and cuts.

[6]Our definition of β-balancedness differs slightly from that given by Leighton and Rao [57]. Their construction and proof of approximate optimality, however, is easily adpated to our definition, and this will be discussed further in Section 3.6.1.

[7]We will precisely define the total flow through a vertex in Section 2.1.

the cut vertices. The ratio of the cut capacity to the product of the component sizes therefore bounds the common amount of flow that can be sent. Leighton and Rao give a method for finding an approximately smallest ratio cost cut, and aggregate these ratio cost cuts to form the desired balanced cut. The computational bottleneck in their construction is the solution of the dual of the associated multicommodity flows. This issue will be addressed next.

1.7 Solution of Concurrent Flows

As we saw in Section 1.6, the Leighton-Rao cut construction uses the solution to the linear programming dual of a special multicommodity flow, known as a *concurrent flow*. Multicommodity flows have a rich history and diverse applications, and have been used, for example, in routing in communication and transportation networks. We refer the interested reader to the comprehensive book by Ahuja, Magnanti and Orlin [3] on the theory, algorithms and applications of multicommodity flow. The solution of the concurrent flow problem is the focus of Chapter 2. In this section, we briefly describe the solution process.

The concurrent flow problem was first introduced by Shahrokhi and Matula [80]. In an instance of the (vertex-based) concurrent flow problem, we are given a graph where the vertices have capacities, and a set of commodities with associated demands. The objective is to maximize a common fraction of the demands that can be shipped from the source to the sink of each commodity, without violating the capacity constraints. When all the vertices have the same capacity, an equivalent objective is to minimize the *congestion* of the flow, which is defined to be the maximum flow through a vertex.

The dual to a concurrent flow problem essentially assigns lengths to the vertices in the graph, with an objective of maximizing the sum of the shortest path distances between the sources and sinks of the commodities. Both the concurrent flow problem and its dual can be formulated as linear programs with a polynomial number of variables and inequalities. They can therefore be solved to optimality in polynomial time by linear programming methods. However, as we shall see in Section 2.5.1, the solution time is very high, even for small problem instances.

We instead solve the concurrent flow problem and the dual problem approximately, using the approximation algorithm of Klein, Plotkin, Stein and Tardos [52]. Our implementation of their algorithm and computational experience with it are described in Chapter 2. A concurrent flow (respectively dual length function) is defined to be ϵ-optimal, for some error tolerance ϵ, if its objective value is within a $(1+\epsilon)$ (respectively $(1-\epsilon)$) factor of the optimal objective value. Since the optimal objective values of the concurrent flow and its dual are equal by linear programming duality, we can establish that a concurrent flow and a dual length function are both ϵ-optimal if they are within a $(1+\epsilon)$ factor of each other.

There are several parameters present in the concurrent flow approximation algorithm, and they have to be carefully chosen in an implementation. We describe in Section 2.4 an extensive tuning process that we performed, which yielded impressive improvements in running time.

1.8 Contribution and Outline of Thesis

This thesis describes an extensive computational study of the Agrawal-Klein-Ravi algorithm of Section 1.5.3 and the underlying balanced separator and concurrent flow subroutines introduced in Section 1.6 and 1.7. We propose and investigate improvements to the Agrawal-Klein-Ravi algorithm, and consider simplifications to the Leighton-Rao balanced separator construction. In addition, we show that the concurrent flow approximation algorithm, which is the computational bottleneck in the implementation, can be made to run substantially faster through aggressive optimization. To date, there has been only one other complete implementation of the Agrawal-Klein-Ravi algorithm [2].

In the previous sections, we have given a top-down overview of the elimination ordering problem and the Agrawal-Klein-Ravi algorithm for finding good orderings. Our implementation of the algorithm proceeded of course in a bottom-up fashion, and the arrangement of the thesis will follow this development.

An outline of the rest of the thesis and the specific contributions made follows. We give a detailed account of our implementation of the concurrent flow approximation algorithm of Klein, Plotkin, Stein and Tardos [52] in Chapter 2. In particular, we devise in Section 2.3.5 a surprising simplification of Dijkstra's algorithm for shortest paths in the case where distances are based on vertex rather than edge lengths. We also discuss at length in Section 2.4 the empirical process by which we tuned the implementation to the inputs that it will be used for. Further experimental results for the implementation are given in Section 2.5, including a comparison in Section 2.5.1 showing that the implementation is two-or-more orders of magnitude faster than linear programming methods.

In Chapter 3, we focus on the Leighton-Rao balanced separator construction and the associated sparse cut construction. Unlike the treatment of the sparse vertex cut problem in the literature, we do not reduce the problem to the directed edge cut problem. Instead, we present in Section 3.2 a construction that works directly on the input graph, and which is considerably simpler than its edge-based counterpart. We also propose a further simplification to the Leighton-Rao sparse cut construction in Section 3.3, and carefully evaluate these constructions in Section 3.4. In Section 3.6, we evaluate the performance of the Leighton-Rao balanced separator construction. We then introduce in Section 3.7 *balanced low-deficiency separators*, which are balanced separators with a nonstandard cost measure, and evaluate the use of simulated annealing for finding such separators.

Chapter 4 focusses on the class of nested dissection methods for finding elimination orderings. We introduce in Section 4.5 a new elimination ordering problem, and show that it can be reduced to the standard problem. This reduction forms the framework of a novel variant of nested dissection, which we call *restricted nested dissection*, that we shall develop in Section 4.6. We empirically evaluate the various nested dissection algorithms in Section 4.7. The results show that restricted nested dissection is superior to other forms of nested dissection. We also present results on using balanced low-deficiency separators for nested dissection in place of approximately optimal balanced separators.

1.9 Notation and Terminology

We shall use the standard notation $G = (V, E)$ for a graph, where V is the set of vertices and E is the set of edges. Most of the graphs we examine will be undirected and connected. The number of vertices is always denoted by n, and the number of edges by m. An edge between vertices u and v is written as uv. Note that uv and vu refer to the same edge.

The terms (vertex) cut and separator have both been widely used in the literature to describe a set of vertices whose removal disconnects a graph. The cut terminology is usually employed in the study and construction of such dual objects in relation to their primal multicommodity flows. In contrast, such sets are typically referred to as separators when they are used in the design and analysis of divide-and-conquer algorithms. We shall follow the common usage of these terms in the literature. Accordingly, in Chapters 2 and 3, the cut terminology is used, while in Chapter 4, we switch to the separator terminology.

Chapter 2

Vertex-Based Concurrent Flows

The multicommodity flow problem involves shipping several commodities subject to capacity constraints. In this chapter, we consider an optimization version of the multicommodity flow problem, known as the concurrent flow problem, in which the objective is to maximize a common fraction of the demand shipped for each commodity. The concurrent flow problem is a component of many important approximation algorithms, including the sparsest cut construction of Leighton and Rao [57] that we shall examine in Chapter 3. For more uses of the concurrent flow problem, we refer the reader to the survey article by Shmoys [81].

We first define in Section 2.1 the concurrent flow problem where vertices have capacity constraints. In the rest of the chapter, we focus on the vertex-based variant of the approximation algorithm of Klein, Plotkin, Stein and Tardos [52] for the unit-capacity concurrent flow problem. We describe the underlying ideas of the algorithm in Section 2.2, and discuss details of the implementation of the algorithm in Section 2.3. In Section 2.4, we focus on the extensive effort spent on tuning the implementation via the appropriate selection of some implementation parameters. Finally, we compare the performance of the concurrent flow implementation to that of linear programming methods in Section 2.5, and also present and analyze experimental results of running the algorithm on many input graphs.

2.1 Vertex-Based Concurrent Flows

The multicommodity flow problem involves shipping several commodities from their sources to their sinks. More precisely, an instance of the multicommodity flow problem consists of an undirected graph $G = (V, E)$ and a set of k commodities $\{d(i), (s_i, t_i) : 1 \le i \le k\}$, where s_i and t_i are the *source* and *sink* vertices of commodity i, and $d(i)$ is an integral *demand*. We shall refer to s_i and t_i as the *terminals* of commodity i, and assume that they are distinct. We also denote the total demand $\sum_i d(i)$ by D, and the number of vertices and edges in G by n and m respectively. Note that uv and vu refer to the same undirected edge.

The total amount of flow that can be shipped in the network is restricted by capacity constraints on the edges and/or vertices, depending on the application.

The elimination ordering algorithm that we are interested in uses the balanced vertex separator construction of Leighton and Rao [57], which in turn requires the solution of a special multicommodity flow problem where the vertices are capacitated. We therefore restrict our attention to flow problems where only the vertices have capacity constraints. We denote the capacity of a vertex $v \in V$ by $c(v)$.

Let $\mathcal{P}_i$ be the collection of all simple flow paths from s_i to t_i, for each $1 \le i \le k$. A flow f_i in G for the commodity i assigns to each path $P \in \mathcal{P}_i$ a nonnegative real value $f_i(P)$, which is the flow along P. The value of the flow f_i is thus $\sum_{P \in \mathcal{P}_i} f_i(P)$. The flow through an edge uv from commodity i is given by

$$f_i(uv) = \sum \{f_i(P) : P \in \mathcal{P}_i \text{ and } uv \in P\},$$

and the total flow through the edge is $f(uv) = \sum_i f_i(uv)$.

It is also necessary to define the flow through a vertex. There are several ways to do this, and we pick a definition that is suited to the construction of balanced vertex separators. In our definition, flow that originates or terminates at a vertex contributes only half of its value, while flow that is routed through a vertex is counted fully. More precisely, the flow $f_i(v)$ through a vertex v from commodity i is defined by:

$$f_i(v) = \begin{cases} \displaystyle\sum_{P \in \mathcal{P}_i : v \in P} f_i(P) & \text{if } v \ne s_i, t_i \\ \displaystyle\frac{1}{2} \sum_{P \in \mathcal{P}_i} f_i(P) & \text{if } v = s_i, t_i \end{cases}$$

and the total vertex flow is $f(v) = \sum_i f_i(v)$. An alternative, but less symmetric, definition of vertex flow was given in [53], in which flow originating at a vertex v is not counted as being routed through v, but flow terminating at v is.

The definition of vertex flow may at first seem unusual, but there is in fact a simple relation between the flow through a vertex and the flow along the edges incident to it. Each path $P \in \mathcal{P}_i$ contributes $f_i(P)$ flow through intermediate vertices on it but only $\frac{1}{2} f_i(P)$ flow through its endpoints s_i and t_i. In particular, summing the paths in $\mathcal{P}_i$ shows that the flow $f_i(s_i)$ and $f_i(t_i)$ through the terminals of commodity i must necessarily be $\frac{1}{2} d(i)$. One interpretation for the unequal weighting is that there is only one edge on a path $P \in \mathcal{P}_i$ incident to each of s_i and t_i, but there are two edges incident to any intermediate vertex of P, which gives rise to the following observation: *the flow through vertex v due to path P is half the sum of flow of P on edges incident to v.* Using the definitions of edge flows $f_i(uv)$ and $f(uv)$ defined earlier, we see that for a vertex v,

$$f_i(v) = \frac{1}{2} \sum_{uv \in E} f_i(uv) \ \ \forall \ \ 1 \le i \le k \quad \text{and} \quad f(v) = \frac{1}{2} \sum_{uv \in E} f(uv).$$

Thus vertex flows are just scaled accumulations of associated edge flows.

A *feasible* multicommodity flow f in G consists of a collection of flows f_i from s_i to t_i of value $d(i)$ for each commodity i, such that the total flow through each vertex does not exceed its capacity. In general, an instance of the multicommodity flow problem may not have a feasible flow, and it is more useful instead to consider an optimization version of the problem. The version that we shall use is known as the *concurrent flow problem*, and was first defined by Shahrokhi and Matula [80]. In the concurrent flow problem, the objective is to find the maximum possible value z such that there is a feasible multicommodity flow with scaled demands $z \cdot d(i)$ for each commodity i. We call z the *throughput* of the multicommodity flow. By regarding the flow along each flow path as a variable, we can formulate the concurrent flow problem as a linear program:

$$\begin{array}{rcll} \max & z & & \\ \sum_{P \in \mathcal{P}_i} f_i(P) & = & z \cdot d(i) & \forall\ 1 \le i \le k \\ f(v) & \le & c(v) & \forall\ v \in V \\ f_i(P) & \ge & 0 & \forall\ P \in \mathcal{P}_i,\ \ 1 \le i \le k \\ z & \ge & 0 & \end{array}$$

An equivalent formulation of the concurrent flow problem is obtained by relaxing the capacity constraints and minimizing the *congestion* $|f| = \max_v f(v)/c(v)$ of a flow that satisfies the commodity demands. Note that the congestion of a flow f gives the minimum scaling of the capacities for f to be feasible. It is easily verified that the optimal throughput is the reciprocal of the optimal congestion. The associated linear program is given below, where the objective λ upper bounds the congestion, and is equal to the congestion when the linear program is solved.

$$\begin{array}{rcll} \min & \lambda & & \\ \sum_{P \in \mathcal{P}_i} f_i(P) & = & d(i) & \forall\ 1 \le i \le k \\ f(v) & \le & \lambda c(v) & \forall\ v \in V \\ f_i(P) & \ge & 0 & \forall\ P \in \mathcal{P}_i,\ \ 1 \le i \le k \\ \lambda & \ge & 0 & \end{array}$$

It is useful to consider the dual to this linear program, for the dual objective value gives a lower bound on the optimal primal objective value, and is essential in constructing approximation algorithms for solving the concurrent flow problem. We assign dual variables $l(v)$ to the vertices and variables $\hat{d}(s_i, t_i)$ to the commodities. These variables correspond to the total vertex flow inequalities and the demand equations in the concurrent flow linear program. The dual linear program is as follows:

$$\begin{array}{rcll} \max & \sum_{i}^{k} d(i)\hat{d}(s_i,t_i) & & \\ \sum_v c(v)l(v) & = & 1 & \\ \hat{d}(s_i,t_i) & \leq & \frac{1}{2}(l(s_i)+l(t_i)) + \sum_{v\in P: v\neq s_i,t_i} l(v) & \forall\ P\in\mathcal{P}_i,\ 1\leq i\leq k \\ l(v) & \geq & 0 & \forall\ v\in V \\ \hat{d}(s_i,t_i) & \geq & 0 & \forall\ 1\leq i\leq k \end{array}$$

We can interprete the dual variables $l(v)$ as lengths on the vertices, and regard the dual LP as involving distances between the sources and sinks of the commodities as follows. For a path P, let $INT(P)$ be the set of all vertices on P except its endpoints. We wish to define the length of P in terms of the lengths of the vertices on P. This is accomplished by having the vertex lengths *induce* edge lengths, in which the induced length of an edge vw is half the sum of the lengths of its endpoints: $l(vw) = \frac{1}{2}(l(v)+l(w))$. Using the standard definition of the length of a path as being the sum of the lengths of the edges on it, it follows that the length of a path P from s_i to t_i is given by:

$$l(P) = \frac{1}{2}(l(s_i)+l(t_i)) + \sum_{v\in INT(P)} l(v).$$

Note that each vertex on P contributes its entire length to the path length, except for the endpoints which only contribute half their lengths. In addition, the length of a degenerate single-vertex path v is defined to be zero.

With the vertex-based definition of path length, we see that for a commodity i, the variable $\hat{d}(s_i,t_i)$ in the dual linear program is upper bounded by the length of each flow path for i. So when the dual linear program is solved, $\hat{d}(s_i,t_i)$ will be equal to the shortest-path distance between s_i and t_i. The dual objective then is to maximize the demand-weighted sum of shortest path distances between commodities.

In the rest of this chapter, we shall focus exclusively on the special case of the concurrent flow problem where all the vertices have unit capacities. For this special case, the congestion $|f|$ of a feasible flow is simply the maximum flow through a vertex, and a valid vertex length function for the dual problem is one where the sum of the vertex lengths is one. We make a simple observation on the congestion of a flow for the unit-capacity concurrent flow problem. Recall that D denotes the total demand of all commodities.

Lemma 2.1 *Let f be a feasible flow for a unit-capacity concurrent flow problem. Then $D/n \leq |f| \leq D$.*

Proof. Observe that the total amount of flow in the network is D, i.e. the sum of flow over all flow paths is D. Since each flow path $P \in \mathcal{P}_i$ contributes $f_i(P)$

flow through vertices on P that are not terminals of commodity i and contributes $\frac{1}{2}f_i(P)$ flow through s_i and t_i, we see that the total contribution over all flow paths to any vertex v is at most $\sum_i \sum_{P\in\mathcal{P}_i} f_i(P)$. Thus $|f| \leq D$. In addition, the flow path $P \in \mathcal{P}_i$ contributes at least $f_i(P)$ to $\sum_v f(v)$ since s_i and t_i are distinct by assumption. Summing over all flow paths gives $D \leq \sum_v f(v)$, and hence $|f| \geq \sum_v f(v)/n \geq D/n$. ■

An equivalent, but non-linear, formulation for the dual of the unit-capacity concurrent flow problem drops the normalization of the sum of vertex lengths to 1, and replaces the objective function by $\max \sum_i d(i) \cdot \hat{d}(s_i, t_i) / \sum_v l(v)$. The new objective function scales down the original by the sum of vertex lengths, and has the same optimal value as the original. We denote by $|l|_1$ the sum of the vertex lengths, which is no longer 1. We shall use this equivalent dual formulation throughout the rest of the chapter.

2.2 The Unit-Capacity Concurrent Flow Approximation Algorithm

The concurrent flow problem can be solved to optimality using linear programming methods. Although the formulation we gave above has an exponential number of variables due to the enumeration of flow paths, there are equivalent formulations that use only $O(mk)$ variables and $O(m+nk)$ constraints [80], giving a polynomial-time solution of the concurrent flow problem.

Much faster algorithms exist if only an approximately optimal flow is desired. Let ϵ be an error tolerance. We say that a feasible concurrent flow f is ϵ-optimal if $|f|$ is at most a factor $(1+\epsilon)$ more than the minimum possible $|f|$. Klein, Plotkin, Stein and Tardos [52] have given a fast approximation algorithm for the unit edge-capacity, which was subsequently implemented by Borger *et al* [11]. We describe in this and following sections an adaptation of their algorithm to the vertex-capacitated case. As for the general edge-capacity concurrent flow problem, Leighton *et al* [56] gave a fast approximation algorithm, which was later implemented by Leong, Shor and Stein [59].

The approximation algorithm of Klein, Plotkin, Stein and Tardos [52] uses a modification of the framework originated by Shahrokhi and Matula [80] to produce an ϵ-optimal concurrent flow in the case of unit edge capacities. It maintains feasible primal and dual solutions and reduces the current flow to ϵ-optimality by repeatedly rerouting flow from congested paths to less congested ones. Klein, Plotkin, Stein and Tardos give two implementations for the algorithm, one deterministic and the other randomized. In the case of fixed $\epsilon \leq 1/10$, i.e where ϵ does not vary with the given network, the deterministic version runs in time $O(m(k+m)(k+\log n)\log n)$ and the randomized version in expected time $O(m(k+m)\log^2 n)$, where the constants depend on ϵ.[1] The condition $\epsilon \leq 1/10$,

[1]The actual time bounds are slightly better but more complicated than those stated here (cf.

which is assumed by the algorithm, is not restrictive as larger error tolerances can always be replaced by the tighter tolerance of 1/10.

The algorithm is easily adapted to vertex-based concurrent flow by replacing edge-based quantities with their vertex-based counterparts and by using the vertex-based definition of path lengths instead. In the following sections, we briefly describe the adapted algorithm and its underlying ideas. We shall state some results to give insight into the algorithm, but will not prove them as the proofs are just vertex-based analogs of those in [52]. Note that all logarithms are taken to base e unless otherwise specified.

2.2.1 Optimality Conditions

Linear programming duality allows us to conclude when a flow is ϵ-optimal. Specifically, by the strong duality theorem and using the alternative form of the dual given at the end of Section 2.1, we know that a flow f is optimal if and only if there is a length function l for which $|f||l|_1 = \sum_i dist_l(s_i, t_i) \cdot d(i)$. Let f^* and l^* be optimal solutions, and consider a given flow f. If we can find a length function l for which

$$|f||l|_1 \leq (1+\epsilon) \sum_i dist_l(s_i, t_i) d(i), \tag{2.1}$$

then f is ϵ-optimal since

$$|f| \leq (1+\epsilon) \sum_i d(i) \frac{dist_l(s_i, t_i)}{|l|_1} \leq (1+\epsilon) \sum_i d(i) \frac{dist_{l^*}(s_i, t_i)}{|l^*|_1} = (1+\epsilon)|f^*|.$$

In this case, we say that the pair (f, l) is ϵ-optimal. Conversely, f is ϵ-optimal only if there is a length function l such that (f, l) is ϵ-optimal, as seen by taking $l = l^*$.

It is instructive to prove the weak duality result $|f||l|_1 \geq \sum_i dist_l(s_i, t_i) d(i)$ for our pair of linear programs. The proof makes evident the causes of gap between the primal and dual objective values:

$$\begin{aligned}
|f||l|_1 &\geq \sum_v f(v) l(v) \\
&= \sum_i \sum_v l(v) f_i(v) \\
&= \sum_i \left[\sum_{v \neq s_i, t_i} l(v) \sum_{P \in \mathcal{P}_i : v \in P} f_i(P) + \frac{1}{2}(l(s_i) + l(t_i)) \sum_{P \in \mathcal{P}_i} f_i(P) \right] \\
&= \sum_i \sum_{P \in \mathcal{P}_i} \left[\frac{1}{2}(l(s_i) + l(t_i)) + \sum_{v \in INT(P)} l(v) \right] \cdot f_i(P) \\
&= \sum_i \sum_{P \in \mathcal{P}_i} l(P) f_i(P) \\
&\geq \sum_i dist_l(s_i, t_i) \cdot d(i).
\end{aligned}$$

---

[52], Theorem 5.2).

From the proof, we see that the gap between $|f||l|_1$ and $\sum_i dist_l(s_i,t_i)d(i)$ is the sum of the gaps between $|f||l|_1$ and $\sum_v f(v)l(v)$ and between $\sum_i \sum_{P\in\mathcal{P}_i} l(P)f_i(P)$ and $\sum_i dist_l(s_i,t_i)d(i)$.

To bound the two gaps described above, Klein, Plotkin, Stein and Tardos introduce relaxed optimality conditions analogous to the complementary slackness conditions of linear programming. Let $\epsilon' = \epsilon/7$ throughout the rest of this chapter. The first relaxed optimality condition states that every vertex either has length that is a small fraction of the total length or is almost saturated:

$$\text{For each vertex } v \in V, \text{ either } l(v) \le \frac{\epsilon'}{n}|l|_1 \text{ or } f(v) \ge \frac{|f|}{1+\epsilon'}. \quad (R1)$$

The second condition concerns the flow along "long" paths. For a commodity i and a path $P \in \mathcal{P}_i$, we say that P is *ϵ-good* if

$$l(P) - dist_l(s_i,t_i) \le \epsilon' l(P) + \epsilon' \frac{|f|}{\min\{D, kd(i)\}}|l|_1$$

and *ϵ-bad* otherwise. Intuitively, a path is good if it is almost as short as the shortest (s_i,t_i) path or if its length is a small fraction of $|l|_1$. The second condition states that the flow on ϵ-bad paths is a small fraction of the sum $f \cdot l$:

$$\sum_i \sum_{P\in\mathcal{P}_i : P\ \epsilon-bad} f_i(P)l(P) \le \epsilon' \sum_i \sum_{P\in\mathcal{P}_i} f_i(P)l(P). \quad (R2)$$

Conditions (R1) and (R2) can be respectively used to bound the gap between $|f||l|_1$ and $\sum_v f(v)l(v)$ and between $\sum_i \sum_{P\in\mathcal{P}_i} l(P)f_i(P)$ and $\sum_i dist_l(s_i,t_i)d(i)$. When both conditions are satisfied, the given flow is ϵ-optimal, as stated without proof in the following results.

Lemma 2.2 (cf. [52], Lemma 3.1) *Suppose a concurrent flow f and a length function l satisfy relaxed optimality condition (R1). Then*

$$(1-\epsilon')|f||l|_1 \le (1+\epsilon')\sum_v f(v)l(v). \tag{2.2}$$

Theorem 2.3 (cf. [52], Theorem 3.2) *Suppose f and l satisfy the relaxed optimality conditions (R1) and (R2). Then f is ϵ-optimal.*

2.2.2 Generic Rerouting

The key operation in the concurrent flow approximation algorithm is the rerouting of flow to try to lower the congestion. The rerouting step is encapsulated in the core procedure Reduce, which takes an input flow and reduces its congestion below a given target value τ or produces a flow that is ϵ-optimal.

<u>Reduce(f, τ, ϵ, $\{\sigma_i : 1 \le i \le k\}$)</u>

$\alpha \leftarrow (7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7n\epsilon^{-1})$

while $|f| \geq \tau$ **and** f and l are not ϵ-optimal
    for each vertex v: $l(v) \leftarrow e^{\alpha f(v)}$
    find an ϵ-bad path P for some commodity i and a short path Q
        with the same endpoints as P
    reroute σ_i units of flow from P to Q
endwhile
return f

The procedure REDUCE requires that each flow path of a commodity i carries flow that is an integer multiple of its flow quantum σ_i. The basic step in REDUCE consists of selecting an ϵ-bad path P for some commodity i and rerouting some multiple of σ units of flow from P to a short path Q for commodity i, where the length function used is exponential in the vertex flow. Throughout the execution of REDUCE, the relaxed optimality condition (R1) is always satisfied.

Lemma 2.4 (cf. [52], Lemma 4.1) *If f is a concurrent flow and $\alpha \geq (7 + \epsilon)|f|^{-1}\epsilon^{-1}\log(7n\epsilon^{-1})$, then f and the length function $l(v) = e^{\alpha f(v)}$ satisfy relaxed optimality condition (R1).*

The effect of the iterative rerouting of flow is to gradually enforce relaxed optimality condition (R2). If (R2) becomes satisfied, then f and l are ϵ-optimal. Otherwise we can always find an ϵ-bad path to reroute flow away from. By imposing a technical granularity condition on the flow quanta σ_i:[2]

$$\sigma_i \leq \epsilon^2 \frac{\tau}{51\log(7n\epsilon^{-1})},$$

we can guarantee progress in that $|l|_1$ decreases significantly, and hence bound the number of iterations in REDUCE.

Lemma 2.5 (cf. [52], Lemma 4.2) *Suppose σ_i and τ satisfy the granularity condition. Then rerouting σ_i units of flow from an ϵ-bad path of commodity i to the shortest path with the same endpoints decreases $|l|_1$ by*

$$\Omega\left(\frac{\sigma_i}{\min\{D, kd(i)\}}|l|_1 \log n\right).$$

Theorem 2.6 (cf. [52], Theorem 4.3) *If, for every commodity i, σ_i and τ satisfy the granularity condition and $|f| = O(\tau)$ initially, then the procedure* REDUCE *terminates after*

$$O\left(\epsilon^{-1}\max_i \frac{\min\{D, kd(i)\}}{\sigma_i}\right)$$

iterations.

[2] In [52], the denominator has the factor 102 instead of 51. The larger factor arises from the inequality $\epsilon(7+\epsilon)/102 \leq \epsilon/7$ for $\epsilon \leq 1/10$ used in the proof of Lemma 4.2. In fact that inequality can be strengthened to $\epsilon(7+\epsilon)/51 \leq \epsilon/7$ for $\epsilon \leq 1/10$.

2.2.3 The Approximation Algorithm

Given a concurrent flow problem, the algorithm of Klein, Plotkin, Stein and Tarods first constructs an initial feasible flow and then repeatedly calls Reduce to halve the congestion until an ϵ-optimal flow is found.[3]

<u>Concurrent(G, ϵ, $\{d(i), (s_i, t_i) : 1 \le i \le k\}$)</u>

for each commodity i
    $\sigma_i \leftarrow d(i)$
    route $d(i)$ flow from s_i to t_i along a simple path
endfor
$\tau \leftarrow |f|/2$ **while** f is not ϵ-optimal
    for each i
        while σ_i and τ do not satisfy the granularity condition
            $\sigma_i \leftarrow \sigma/2$
        endwhile
    endfor
    call Reduce(f, τ, ϵ, σ)
    $\tau \leftarrow \tau/2$
endwhile

Before presenting running times for Concurrent, we first bound the number of calls to Reduce that are made, and thus show that Concurrent in fact terminates. Let f_0 and $\hat{f}$ respectively denote the initial and final flow. From Lemma 2.1, we know that $|f_0| \le D$ and $|\hat{f}| \ge D/n$. It follows that the congestion of the initial flow can only be reduced by at most a factor of n, and so at most $\log_2 n$ congestion-halving calls to Reduce are made.

Klein, Plotkin, Stein and Tardos give two different implementations of Reduce, one deterministic and the other randomized. For the case of constant ϵ, they show that the deterministic version takes time $O(k^* n \log n + m(\log n + k))$ and the randomized version takes expected time $O(\epsilon^{-1}(m + n \log n))$ for one rerouting step (cf. [52], Lemmata 4.5 and 4.7). Here k^* refers to the number of distinct sources among the commodities. They also show that the total number of rerouting steps over all calls to Reduce in Concurrent is $O(\epsilon^{-1} k \log n + \epsilon^{-3} m \log n)$. Combining these results give the following running times for Concurrent.

Theorem 2.7 ([52], Theorem 5.2) *For any constant $\epsilon > 0$, an ϵ-optimal solution for the unit-capacity concurrent flow problem can be found in $O(m(k + m)\log^2 n)$ expected time by a randomized algorithm and in $O(m(k + m)(\log n + k)\log n)$ time by a deterministic algorithm.*

[3] Klein, Plotkin, Stein and Tardos recommend the use of Concurrent for the case where ϵ is constant. If ϵ is less than a constant, they suggest the use of a companion algorithm which scales the error tolerance down to the desired value.

2.3 Implementation Details

In the following sections, we examine various aspects of the implementation of the concurrent flow algorithm. We first discuss representational issues in Sections 2.3.1 and 2.3.2, and then move on to computational issues involved in rerouting of flow, exponentiation, and shortest and longest paths in Sections 2.3.4 through 2.3.6. In particular, we shall see in Section 2.3.5 a surprising optimization arising from the vertex-based definition of path lengths which leads to a faster implementation of Dijkstra's algorithm for single-source shortest paths. Finally we examine in Sections 2.3.7 through 2.3.10 strategic issues involving the length parameter α, the routing of initial flow, rules for selecting commodities to examine in REDUCE, and when to test the terminating conditions of REDUCE. Some of the issues we discuss have been addressed in [52] and [11] in the context of edge-based flows, and we shall contrast our treatment with those in [52] and [11] in the appropriate sections.

2.3.1 Representation of Flow

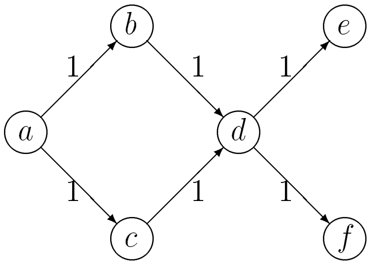
Klein, Plotkin, Stein and Tardos [52] give two implementations of the REDUCE subroutine, one deterministic and the other randomized, each with a different representation of multicommodity flow. The deterministic implementation represents multicommodity flow as a collection of directed flow graphs, whereas the randomized implementation uses a three-level data structure to store the flow paths with positive flow. While the randomized implementation has a better asymptotic running time than the deterministic version, the experience of Borger *et al* [11] shows that it spends too much examining commodities which do not have ϵ-bad paths and is inferior to the deterministic version in practice. We therefore implement the deterministic version of REDUCE.

The directed flow graphs in the deterministic implementation are defined as follows. We direct the flow paths of each commodity from its source to its sink, and consider only those flow paths with positive flow. For each vertex s, we combine into one directed graph, denoted by G_s, the flow paths of those commodities with s as their source vertex. More precisely, an edge is present in G_s if it is on a directed flow path for some commodity with source s, and the total flow through the edge is the sum of the flows on all such flow paths. We refer to G_s as the *directed flow graph* for s. Note that it is possible for G_s to have directed cycles; such cycles can be eliminated without affecting the flow graph, as will be shown later. In general, G_s is an edge-subgraph of the *directed version* H of G, where each undirected edge $\{u, v\}$ of G is replaced by the two directed edges (u, v) and (v, u) in H.[4]

A little thought shows that a flow graph might not be uniquely decomposable into its underlying flow paths. Indeed, consider the following scenario of two unit-

[4] Actually, Klein, Plotkin, Stein and Tardos [52] aggregate into directed flow graphs the directed flow paths of commodities having the same source vertex *and the same flow quantum.* In Section 2.3.3, we replace the individual flow quantum of each commodity by a uniform quantum, and so the sharper distinction of flow graphs vanishes.

demand commodities with the same source vertex a and sink vertices e and f, and suppose the flow graph for those commodities is the following:



This flow graph can be interpreted as sending one unit of flow along the path $a-b-d-f$ and another unit along $a-c-d-e$, or as being composed of the flow paths $a-b-d-e$ and $a-c-d-f$ instead. However, such ambiguities in the decomposition of a flow graph into constituent flow paths for its commodities are harmless. More precisely, the "correctness" of a flow graph lies in how flow is conserved at each vertex, and is independent of how the flow paths were aggregated, as stated below. The following result characterizes flow graphs by extending the flow conservation properties of 1-commodity flows.

Lemma 2.8 *Let $G=(V,E)$ be an undirected graph. Let $t_1,\ldots,t_r$ be the distinct sink vertices of r commodities, all of whom have the same source vertex s. Let $\hat{G}=(V,F)$ be an acyclic edge-subgraph of the directed version of G, and let $f(e)$ denote the positive flow on each edge e of $\hat{G}$. Then $\hat{G}$ represents a feasible multicommodity flow, i.e. it can be decomposed into a collection of flow paths for each commodity i satisfying its demand $d(i)$, if and only if the following constraints hold:*

(a) for each non-source non-sink vertex v, flow is conserved, i.e. $\sum_{uv\in F} f(uv)=\sum_{vw\in F} f(vw)$.

(b) the outgoing excess at s is $\sum_i d(i)$, i.e. $\sum_{sw\in F} f(sw)-\sum_{us\in F} f(us)=\sum_i d(i)$.

(c) the incoming excess at t_i is $d(i)$, i.e. $\sum_{ut_i\in F} f(ut_i)-\sum_{t_iw\in F} f(t_iw)=d(i)$.

It follows from the above lemma that we can convert a flow graph with cycles to one without cycles by reducing in sequence the flow through each cycle by the minimum flow on an edge of the cycle. Such reductions remove at least one edge from each cycle and still leave a flow graph since they do not change the flow conservation properties. We shall return to cycle cancellation in the context of computing longest paths in flow graphs in Section 2.3.6.

In fact the ambiguous decomposition of a flow graph adds flexibility during rerouting. After selecting a commodity i for potential rerouting, we check if a long path P from s_i to t_i in the flowgraph G_{s_i} is ϵ-bad. It need not be the case that flow for commodity i was actually routed along P previously, for we can decompose the flowgraph such that P actually carries flow for commodity i. Even more importantly, let Δ be the minimum flow on edges of P, and suppose $\Delta > d(i)$. This is possible if P is the initial part of flow paths from s to other sinks. We can then reroute up to Δ flow from P to some other (s, t_i) path Q, effectively rerouting flow for more than one commodity, and still have a feasible flow graph. This is because rerouting from P to Q does not change the incoming/outgoing excess at s and t_i nor at intermediate vertices along P and Q. The actual amount of flow to reroute will be considered in Section 2.3.3.

2.3.2 Numerical Considerations

In implementing any algorithm involving real quantities by floating point arithmetic, overflow and precision are two important considerations. The use of an exponential length function in REDUCE exacerbates the limitations of floating point arithmetic, and we examine in this section measures taken to ensure correct functioning of the concurrent flow algorithm. We also analyze the adequacy of IEEE double precision arithmetic for implementing the algorithm.

Coping with Overflow

The exponential length function $l(v) = e^{\alpha f(v)}$ easily overflows the range of IEEE double precision arithmetic.[5] In fact e^{710} exceeds the largest representable number. Using the value $\alpha = (7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7n\epsilon^{-1})$ given in Section 2.2.2, we see that at the more congested vertices, say v with $f(v) \geq \tau$, and with $\epsilon = 1/10$,

$$\alpha f(v) \geq (7+\epsilon)\epsilon^{-1}\log(7n\epsilon^{-1}) \geq 71\log(70n).$$

This last value exceeds 710 for $n \geq 320$. The solution is to downscale each vertex length by some constant, which we derive below. Note that scaling the lengths do not affect the rest of the algorithm; in fact we can think of the scaled lengths as an alternative length function. Throughout the rest of this section, l denotes the scaled vertex length function.

Let ∞ denote the largest representable real number, and let $e^{\alpha p}$ be the scaling factor that we divide lengths by. A close examination of REDUCE shows that the largest computed quantities occur in checking the ϵ-optimality inequality $|f||l|_1 \leq (1+\epsilon)\sum_i dist_l(s_i, t_i)d(i)$. Let $|l|_\infty = e^{\alpha|f|}/e^{\alpha p}$ denote the largest scaled vertex length. Recall from Lemma 2.1 that $|f| \leq D$. We can crudely bound both $|f||l|_1$ and $(1+\epsilon)\sum_i dist_l(s_i, t_i)d(i)$ by $2nD|l|_\infty$. It is then sufficient to have

$$e^{\alpha|f|}/e^{\alpha p} \leq \frac{\infty}{2nD},$$

from which

$$p \geq |f| - \frac{\log(\infty/2nD)}{\alpha}.$$

It is convenient, as we shall see in Section 2.3.4, to have p be an integral multiple of σ. Discretizing yields

$$p = |f| - \sigma\lfloor\log(\infty/2nD)/\alpha\sigma\rfloor$$

[5] We choose IEEE double precision arithmetic because it is widely available. We could instead use the less readily available quadruple precision arithmetic for its greater range and precision, but the measures described here render that unnecessary.

as a valid scaling parameter.

The use of length scaling to make the largest length-related computation fit within floating-point limits has a side effect on small vertex lengths. Very small vertex lengths which are previously representable will now *underflow* when divided by the relatively large scaling factor. This phenomenon is unavoidable since the range of IEEE double precision arithmetic cannot always accommodate the range of values generated by the exponential length function, and so some set of length values will be unrepresentable for each scaling factor. However even though a scaled vertex length $e^{\alpha f(v)}/e^{\alpha p}$ may underflow, its logarithm $\alpha f(v) - \alpha p$ is easily computable and useful (see Section 2.3.3), and we return its value instead. We shall see later that underflow in vertex lengths does not affect the correctness of REDUCE.

Adequacy of finite precision

The second numerical consideration involves the finite precision of floating point arithmetic, which in IEEE double precision is 53 bits. The problem arises in both the representation of lengths and of flow values. In the case of vertex lengths, the huge range of values of the exponential function is worrisome. For example, the value e^{100} is "noise" compared to e^{140}, and disappears when combined additively with the much larger number. However, from Lemma 4.9 of [52], we know that the length of a path can have an *absolute* rounding error of up to $(\epsilon'|f||l|_1)/(2D)$ without affecting the correctness of REDUCE. Therefore each vertex length can have a rounding error of $(\epsilon'|f||l|_1)/(2nD)$. By using the inequalities $|f| \geq D/n$ and $|l|_1 \geq |l|_\infty$, we obtain a more conservative absolute rounding error of

$$\frac{\epsilon'}{2n^2}|l|_\infty = \frac{\epsilon}{14n^2}|l|_\infty.$$

It follows that vertices with larger lengths have to be computed with greater relative accuracy. In particular, the relative rounding error for the longest vertex can be at most $\epsilon/(14n^2)$. So $\log_2(14n^2/\epsilon)$ bits suffice in the representation of lengths. For IEEE double precision arithmetic, this yields the bound $\log_2(14n^2/\epsilon) \leq 53$, which is satisfied for $n^2 < 10^{14}\epsilon$. Thus IEEE double precision arithmetic is adequate except for very large graphs or very small error tolerances.

On the flip side, the above analysis shows that vertices whose lengths are below $\epsilon|l|_\infty/(14n^2)$ can be completely omitted in length calculations. The vertices with underflow lengths fall into this category, and their lengths can be set to 0 without affecting the correctness of the algorithm.

As for the representation of flow values, limited precision might cause difficulties if the flow quanta become too small, as flow values have to be exactly represented, unlike vertex lengths. Recall that the flow quanta σ_i are halved at each call to REDUCE until ϵ-optimality is established. If this halving occurs too many times, then for some commodity i, σ_i might be very small compared to the total flow through each vertex or edge, and rerouting σ_i units of flow would not change the flow values numerically.

We bound the number of bits needed to represent flow values by bounding the number of bits respectively needed for the integral part and the fractional part of the flow values. From Lemma 2.1, we know that the flow through any vertex or edge is at most D, so at most $\lceil \log_2 D \rceil$ bits are needed for the integral part. The number of bits needed for the fractional part depends on how many times σ_i was halved, which occurs in the initialization within CONCURRENT and at each call to REDUCE in which τ is halved. From Section 2.2.3, we know that the number of such REDUCE calls is at most $\lceil \log_2(n|f_0|/D) \rceil \leq \lceil \log_2 n \rceil$, where f_0 is the initial flow. As for the initialization of σ_i within CONCURRENT, there are two cases. In the first case, a σ_i value of $d(i)$ already satisfies the granularity condition and no halving occurs. The total number of bits needed for flow values is at most $(\log_2 D + 1) + (\log_2 n + 1)$, which does not exceed 53 for $nD \leq 10^{15}$. In the second case, σ_i is halved from 1 until it drops below $\epsilon^2 |f|_0 / 102 \log(7n\epsilon^{-1})$. The number of such halvings is at most $\lceil \log_2(102 \log(7n\epsilon^{-1})/\epsilon^2 |f_0|) \rceil$, giving a total integral and fractional bit count of at most

$$
\begin{aligned}
&(\log_2 D + 1) + (\log_2 n + \log_2 |f_0| - \log_2 D + 1) + \\
&(\log_2 102 + \log_2 \log(7n\epsilon^{-1}) + 2 \log_2 \epsilon^{-1} - \log_2 |f_0| + 1).
\end{aligned}
$$

Simplifying the above expression produces the upper bound $10 + \log_2(n/\epsilon^2) + \log_2 \log(7n\epsilon^{-1})$, which does not exceed 53 for $n/\epsilon^2 \leq 10^{11}$. Thus IEEE double precision arithmetic is capable of exactly representing the flow values encountered in the concurrent flow algorithm for practically all input networks and error tolerances.

2.3.3 Multiple Rerouting

The REDUCE procedure described in Section 2.2.2 keeps flow quanta σ_i for each commodity, and reroutes σ_i units of flow for some commodity i at each step. In our implementation however, we keep a uniform flow quantum σ for all commodities, and also reroute a multiple of σ units of flow at each rerouting step. We discuss these changes in this section. The implementation of Borger *et al* [11] also rerouted a multiple of the flow quantum at each rerouting step, but they retained the individual flow quatum for each commodity.

The disadvantage of having a uniform flow quantum σ is that it should divide evenly into the individual flow quantum σ_i, making it potentially much smaller than the individual flow quantum σ_i. However, for the concurrent flow problems that we are interested in, the demand of each commodity is bounded by a small integer constant, and is almost always 1 ,and so the uniform flow quantum σ is only a constant fraction smaller than each individual quantum σ_i. The use of a uniform flow quantum is in fact useful in vertex length computations, as we shall see in Section 2.3.4.

Some words of caution are in order on how to choose the uniform flow quantum σ. Recall that the initial flow is constructed by routing the demand of each commodity along a simple path, and that the demands are integers. It would seem then that initializing σ to 1 and then halving it whenever necessary to maintain the technical granularity condition of Section 2.2.2 would be correct. However we are also interested in quantizing the total vertex flows, and this introduces a complication due to the definition that flow originating or terminating at a vertex only contributes half its value to that vertex, as contrasted to flow passing through the vertex. Consider some vertex v, and suppose that each flow path carries flow that is an integer multiple of σ. Each commodity i with v as either source or sink contributes $d(i)/2$ total flow to v, whereas each flow path *passing through* v contributes some multiple of σ units of flow to v. Thus the total flow through v is some integer multiple of $\frac{1}{2}$ plus some integer multiple of σ. For σ to evenly divide into this quantity, it follows that we should initialize it to $\frac{1}{2}$ instead of 1.

The second change we made to flow rerouting involves rerouting more than σ units of flow at each rerouting step. The optimal amount of flow to reroute is determined as follows. Recall that we use $|l|_1$ to measure progress from rerouting. Let P be an ϵ-bad path for some commodity i and Q be a short flow path for i, and let $A = P - Q$ and $B = Q - P$ respectively contain the vertices that occur only on P and Q. Note that neither the source s_i nor the sink t_i appear in A or B. Rerouting r units of flow from P to Q decreases the flow through each vertex in A by r and increases the flow through each vertex in B by r, while leaving unchanged the flow through all other vertices, including s_i and t_i. The decrease in $|l|_1$ due to the rerouting is

$$\Delta = [l(A) + l(B)] - [e^{-\alpha r} l(A) + e^{\alpha r} l(B)],$$

where $l(A) = \sum_{v \in A} l(v)$ and $l(B) = \sum_{v \in B} l(v)$ are the sum of the lengths of the vertices in A and B, and l refers to the length function before the rerouting. By setting the derivative of Δ with respect to r to 0, we find that rerouting

$$r^* = \frac{1}{2\alpha}(\log l(A) - \log l(B))$$

units of flow maximizes the decrease in $|l|_1$. The actual amount rerouted is the minimum of the nearest multiple of σ to r^* and the minimum flow on an edge along P.

It is possible that $l(B)$ is computed as zero, rendering it impossible to take its logarithm. This occurs due to the underflow of vertex lengths after scaling, as described in Section 2.3.2. However, the length computation also returns the logarithm of each vertex length. We upper bound $\log l(B)$ by $\log |B| + \max_{v \in B} \log l(v)$ in the cases where $l(B)$ computes to zero.

2.3.4 Speeding up Exponentiation

Experience has shown that the exponentiation present in vertex lengths can be a computational bottleneck. We replace exponentiation by the *binary method* of computing powers [54], where an integral power is computed through a sequence of multiplications and squarings. This is possible because of the quantization of flow, which allows the computation of $e^{\alpha f(v)} = (e^{\alpha\sigma})^{[f(v)/\sigma]}$ as an integral power of

$e^{\alpha\sigma}$. Throughout the execution of one REDUCE call, α and σ remain fixed, and so we can *precompute* the binary powers $e^{\alpha\sigma}$, $(e^{\alpha\sigma})^2$, $(e^{\alpha\sigma})^4$, $(e^{\alpha\sigma})^8$, …, and form $(e^{\alpha\sigma})^{[f(v)/\sigma]}$ as a product of the appropriate binary powers. The downward scaling of lengths by the factor $e^{\alpha p}$ in Section 2.3.2 is easily incorporated. Recall that we chose p to be an integral multiple of σ, say $p = \sigma p'$, $p' > 0$. The actual lengths computed are of the form

$$e^{-\alpha p} e^{\alpha f(v)} = (e^{\alpha\sigma})^{[f(v)/\sigma - p']}$$

which conforms to the exponentiation scheme just described. Care has to be taken when $f(v)/\sigma - p'$ is negative, in which case reciprocals are necessary and underflow might occur.

The above exponentiation scheme exploits the use of a uniform flow quantum σ for all commodity flows. If instead individual flow quanta σ_i were maintained for each commodity, we would still need a common quantum that is guaranteed to divide evenly into each total vertex flow in order to use the exponentiation scheme. This last observation was the motivation for the uniform flow quantum scheme.

Exponentiation can be totally avoided across REDUCE calls if the previous call did not conclude that α was too low. In such instances, and if the flow is not already ϵ-optimal, α is doubled and σ is halved, leaving $\alpha\sigma$ constant. The precomputed binary powers from the previous REDUCE call can then be used for the next call.

2.3.5 Shortest Paths Computation

The rerouting of flow for a commodity i in REDUCE requires finding a short flow path for i. This is accomplished by computing a shortest path from s_i to t_i in the undirected input graph G, which is a computationally expensive operation as compared to the rest of the rerouting step. We consider the more general problem of computing shortest paths from a single source to all vertices. We first briefly review Dijkstra's algorithm and then develop a surprising optimization that considerably simplifies Dijkstra's algorithm for vertex-based shortest paths and the data structures used to implement it. Finally we consider other enhancements to the shortest path implementation.

Dijkstra's Algorithm

Dijkstra's algorithm [15] is a greedy algorithm that computes shortest paths from a single source, denoted by s, in the case where edge/vertex lengths are nonnegative. It maintains a priority queue of distance estimates $dist[]$ for each vertex, and at each step the vertex v in the queue with the lowest estimate is removed, and the edges from v are used to update the distance estimates of its neighbors. In the absence of negative lengths, it can be shown that when v was removed, its distance estimate $dist[v]$ in fact gives the shortest path distance from s to v. We shall say that vertex v has *stabilized*. Note that Dijkstra's algorithm stabilizes vertices in an ascending order of their shortest path distances. We also say that a vertex v is *reached* if some path from s, but not necessarily a shortest, has been found. The algorithm also records the predecessor of each vertex in the paths chosen, so that the shortest paths can be recovered. For concreteness and for later comparison, we give below a pseudocode for Dijkstra's algorithm.

<u>DIJKSTRA(G, l)</u>

```
for each vertex v
    dist[v] ← ∞, pred[v] ← UNDEFINED
endfor
dist[s] ← 0
S ← ∅
initialize Q to empty priority queue
INSERT(Q, s, dist[s])
while Q ≠ ∅
    (v, d) ← EXTRACTMIN(Q)
    S ← S ∪ {v}
    for each neighbor w of v
        altlen ← d + 1/2(l(v) + l(w))
        if w ∉ S and altlen < dist[w] then
            if w is in Q then
                DecreaseKey(Q, w, altlen)
            else
                Insert(Q, w, altlen)
            endif
            dist[w] ← altlen, pred[w] ← v
        endif
    endfor
endwhile
```

The set S above contains the stabilized vertices, while Q is a priority queue that supports the following operations: inserting a vertex (INSERT), removing a vertex with the smallest distance estimate (EXTRACTMIN), and lowering the distance estimate of a vertex (DECREASEKEY). The fastest implementations of Dijkstra's algorithm use the Fibonacci heaps of Fredman and Tarjan [26] and run in time $O(m + n \log n)$. Historically, the development of Fibnonacci heaps was motivated by the observation that in Dijkstra's algorithm there are potentially many more DECREASEKEY operations than EXTRACTMIN, so reducing the amortized time of DECREASEKEY to $O(1)$ without increasing the amortized time of EXTRACTMIN would lead to asymptotically faster running times.

We give a graphical interpretation of Dijkstra's algorithm that will be useful later. It is well known that the set of computed shortest paths forms a shortest path tree rooted at s, with $pred[v]$ giving the parent of v in the tree. We can view the execution of Dijkstra's algorithm as an evolution of the shortest path tree. At any point in time, the tree is made up of vertices in S and associated edges given by the $pred[]$ pointers, while the priority queue Q contains vertices outside of S

that are adjacent to the leaves of the tree. We shall refer to those leaves as the *frontier* of the shortest path computation. Each step of Dijkstra's algorithm adds a non-tree vertex that is adjacent to the frontier and that has the lowest distance estimate to the tree, and then updates the distance estimates of the non-stabilized neighbors of v.

Vertex-Based Dijkstra's Algorithm

An unusual optimization exisits for Dijkstra's algorithm when path lengths are defined in terms of vertex lengths. Let w be some target vertex, and P be a path from s to t. Recall that the length of P is given by

$$l(P) = \frac{1}{2}(l(s) + l(w)) + \sum_{v \in INT(P)} l(v),$$

where $INT(P)$ is the set of interior vertices in P. It follows from the definition that a shortest path from s to w minimizes the sum of the lengths of its interior vertices. This exclusive dependence on interior vertices motivates the definition of a new path length $\vec{l}(P)$, which we shall call the *directed path length*:

$$\vec{l}(P) = \frac{1}{2}l(s) + \sum_{v \in P : v \neq s} l(v).$$

Note that the definition of directed path length effectively imposes a direction on the path P by weighting the starting vertex s and the ending vertex w differently. The two path length definitions are closely related: $\vec{l}(P) = l(P) + \frac{1}{2}l(w)$ for $w \neq s$. We naturally define the directed distance of a vertex v from s to be the smallest directed length of a path from s to v. We also define the directed length of the degenerate single-vertex path s as $\frac{1}{2}l(s)$ so that the above relation between directed and normal path lengths holds for degenerate paths.

The crucial observation is that the shortest path tree from s remains the *same* if we use directed path lengths instead of normal path lengths. This is because with directed path lengths, the shortest path still depends only on the sum of the lengths of the interior vertices, as before. Given the computed directed distance $dirdist[v]$ of a vertex v from s, we can recover its true distance $dist[v]$ by $dist[v] = dirdist[v] - \frac{1}{2}l(v)$. The directed path length, however, has the advantage in that extending a path P from s to some vertex v by an edge vw increase the directed path length by $l(w)$, whereas its normal path length increases by $\frac{1}{2}(l(v) + l(w))$, a quantity that depends on *both* v and w, instead of only on w. Let w be some vertex not adjacent to s. It follows that by using the directed path length, a shortest path from s to w can be constructed by appending the edge v^*w to a shortest path from s to v^*, where v^* is a neighbor of w with the lowest directed distance. In the context of Dijkstra's algorithm, this means that if a vertex w is first *reached* via the edge vw when the vertex v is removed from the priority queue, then the path from s to w via edge vw is in fact a shortest path from s to w. Whenever a vertex is first reached in Dijkstra's algorithm, it immediately stabilizes, i.e. its shortest path is immediately known. The properties of a vertex being reached and of a vertex stabilizing are identical. We can therefore rewrite Dijkstra's algorithm as follows:

<u>NewDijkstra(G, l)</u>

for each vertex v
    $dist[v] \leftarrow \infty$, $pred[v] \leftarrow$ UNDEFINED
endfor
$dirdist[s] \leftarrow \frac{1}{2}l(s)$
$S \leftarrow \{s\}$
initialize Q to empty priority queue
Insert(Q, s, $dist[s]$)

while $S \neq V$
    $(v, d) \leftarrow$ ExtractMin(Q)
    for each neighbor w of v
        if $w \notin S$ **then**
            $dirdist[w] \leftarrow d + l(w)$, $pred[w] \leftarrow v$
            $S \leftarrow S \cup \{w\}$
            Insert(Q, w, $dirdist[w]$)
        endif
    endfor
endwhile

The modified version of Dijkstra's algorithm based on directed path lengths is considerably simpler than that based on normal path lengths. Most importantly, there are no longer any DecreaseKey operations because of the immediate stabilization of vertices. This eliminates the advantage of a Fibonacci heap in implementing the priority queue, and we can use regular heaps instead to achieve the same running time $O(m + n \log n)$. The priority queue Q now keeps reached (and hence stabilized) vertices whose edges have not yet been examined, whereas previously it kept reached vertices that have not stabilized.

Note however that the shortest path computation proceeds differently in the two versions of Dijkstra's algorithm. Although directed and normal path lengths are closely related, using directed and normal distances to order vertices will result in different orderings, i.e. a vertex v whose directed distance is greater than that of a vertex w might have a smaller normal distance compared to w. So the set of stabilized vertices and the associated shortest path trees evolve differently in the two versions of Dijkstra's algorithm, although the final shortest path trees are identical.

We performed some trial runs on the two versions of Dijkstra's algorithm. The results showed that using directed path lengths reduced the running time of Dijkstra's algorithm by 40% to 65%.

Other Shortest Path Enhancements

There are other enhancements that we can make to the shortest paths computation. The most obvious one is to only compute the shortest path from the source s_i of a commodity i being examined until the sink t_i is reached instead of computing to all vertices. For some commodity selection rules that we shall consider in Section 2.3.9, commodities with the same source are examined in sequence in searching for one with an ϵ-bad path. In such instances, the information accumulated during the shortest path computation of one such commodity should be saved to avoid unnecessary recomputation when a shortest path is then immediately needed for another commodity with the same source. In general, it is useful to have an incremental shortest path scheme, which is initialized with a source vertex, and that subsquently accepts requests for shortest paths to selected vertices. A request for a vertex that has already been reached is easily handled using the stored information, while a request for an unreached vertex v extends the last computation till v is reached.

Klein, Plotkin, Stein and Tardos [52] describe two improvements for shortest path computations in the case where the error parameter ϵ is constant, but we shall see that they are useful in general. The first improvement is that longest path distances can be used as cutoff-values for the shortest path computation. Recall that shortest paths are always computed in conjunction with longest paths when trying to determine if a commodity i has an ϵ-bad path. The shortest path from s_i to t_i in the input graph G cannot be longer than its longest path from s_i to t_i in the *flowgraph* for commodity i. Thus the longest path length, denoted by $long(s_i, t_i)$, can be used to eliminate parts of the shortest path computation as follows.[6] We first discard any vertex v whose length is greater than $long(s_i, t_i)$ since it will not be on a shortest (s_i, t_i) path. In addition, if a reached vertex v has directed distance greater than $long(s_i, t_i)$, then we do not need to examine its neighbors since the path from s to v will not be an initial part of a shortest (s_i, t_i) path. Note that the cutoff value has to be adjusted when it is to be applied to the incremental shortest path computation scheme described above. Instead of using the $long(s_i, t_i)$ as the cutoff, we use the longest path length from s_i to any vertex in the flowgraph for i. This more tolerant threshold is no smaller than the longest path length for any commodity with the same source as i, and ensures that no useful information is discarded should a shortest path from s_i to some other sink is subsequently needed.

The second improvement described in [52] is that of computing approximate shortest paths by discretizing the vertex lengths. In [52], it is shown that in trying to reroute flow from a longest path P of some commodity i to a flow path Q for i, Q need not be a shortest path, but its length can instead be up to $\epsilon' l(P)/2$ longer than that of a shortest path. We use the more conservative error bound $\epsilon' l(P)/4$, and round each vertex length up to the nearest integral multiple of $\epsilon' l(P)/4n$, to

[6] The actual cutoff value is slightly higher than $long(s_i, t_i)$ to accommodate roundoff errors.

obtain a discretized length function

$$\hat{l}(v) = \frac{\epsilon' l(P)}{4n} \left\lceil \frac{l(v) \cdot 4n}{\epsilon' l(P)} \right\rceil.$$

A shortest path from s_i to t_i using the length function $\hat{l}$ is at most $\epsilon' l(P)/4$ longer than a shortest path using the original length function l, and we can therefore compute shortest paths using $\hat{l}$. The path lengths that are encountered during the computation are multiples of $\epsilon' l(P)/4n$, permitting the use of an integer-valued heap.[7] We shall compare the performance of approximate shortest paths to exact shortest paths in Section 2.4.2.

In the case where ϵ is constant, we can use Dial's implementation [17] of Dijkstra's algorithm, where we replace the priority queue by a sophisticated array-like structure indexed by possible path lengths. The running time for Dial's implementation is $O(m + n\epsilon^{-1})$, while using a heap to implement the priority queue gives a running time of $O(m + n \log n)$ for Dijkstra's algorithm. However, initial profiling showed that Dial's implementation is slower than using an integer-valued heap, and so we did not further pursue that approach.

One other implementation decision concerns the branching factor of the heap used to implement the priority queue. Instead of using binary heaps, we could use 3-ary or even 4-ary heaps, with a tradeoff between the height of the heap and the number of comparisons to child nodes at each level during an INSERT or EXTRACTMIN operation. We ran some test cases using binary, 3-ary and 4-ary heaps. As expected, the 3-ary heaps performed the worst due to heap index calculations involving division by 3, which cannot be implemented by a right shift, as in the case of division by 2 or 4. For both real-valued heaps (used in exact shortest paths) and integer-valued heaps (used in approximate shortest paths), there was no significant difference between binary heaps and 4-ary heaps, and so we chose the simpler binary heaps for the implementation.

2.3.6 Longest Paths Computation

Klein, Plotkin, Stein and Tardos [52] observe that if a commodity i has an ϵ-bad path (with positive flow), then its longest flow path in the flowgraph for i must be ϵ-bad. We describe in this section the ideas behind a simple dynamic programming algorithm mentioned in [52] for computing longest paths in *acyclic flow graphs* and optimizations for this algorithm, and also discuss cycle-cancellation algorithms.

Consider the flowgraph G_s for commodities with the same source s, and assume that G_s is acyclic. By the definition of G_s as the aggregation of directed flow paths from s to sinks, we know that there are no incoming edges into s. So we can compute a topological ordering $\pi : V \to \{1, 2, \ldots, n\}$ of the vertices with $\pi(s) = 1$, i.e. for each edge vw in G_s, $\pi(v) < \pi(w)$. Observe that for a vertex w, any path from s to w in G_s can only pass through vertices ordered before w. In addition, a

[7] We also discretize the length $\frac{1}{2}l(s)$ by the same divisor, so that all directed path lengths are indeed multiples of $\epsilon' l(P)/4n$.

longest path from s to w can be found by appending the edge vw for some vertex v which has w as a neighbor to a longest path from s to v. By processing vertices in their topological order, we ensure that when the edge vw is encountered, a longest path from s to v has already been computed, and we can then decide whether the path from s to w via v is longer than the path we have seen so far for w. Formally the dynamic programming algorithm is as follows. We let $pred[v]$ denote the predecessor of v in a longest (s, v) path, and let $pathlen[v]$ be the length of such a longest path.

$\underline{\text{LONGESTPATH}(G_s,\, l)}$

compute a topological ordering for G_s
for each v
    $pred[i] \leftarrow$ UNDEFINED, $pathlen[v] \leftarrow$ UNDEFINED
endfor
$pathlen[s] = 0$
for $i \leftarrow 1, \ldots, n$
    $v \leftarrow \pi^{-1}(i)$
    for each w on adjacency list of v
        $altlen \leftarrow pathlen[v] + \frac{1}{2}(l(v) + l(w))$
        if $pred[w]$ is UNDEFINED **or** $altlen > pathlen[w]$ **then**
            $pred[w] \leftarrow v$, $pathlen[w] \leftarrow altlen$
        endif
    endfor
endfor

It is clear that the dynamic programming algorithm runs in time $O(n+m)$, for a constant amount of work can be attributed to each vertex and edge. However, initial experience showed that the flow graphs tend to be sparse in that they have very few edges, typically less than 10% of the edges of the underlying undirected input graph, and correspondingly few vertices have incident edges. We shall present statistics for the flow graphs in Section 2.5.2. In other words, only a small fraction of the vertices are reachable from s in the flow graph. We can therefore greatly reduce the execution time by computing a topological ordering and running the dynamic programming algorithm only on those reachable vertices.

The dynamic programming algorithm just described requires that the flow graph be acyclic in order to admit a topological ordering. However the rerouting of flow might create cycles in a flow graph, which would have to be removed. Klein, Plotkin, Stein and Tardos [52] use the fast and sophisticated cycle cancellation algorithm of Sleator and Tarjan [82] which runs in time $O(m \log n)$ to derive time bounds for one rerouting step. In practice, however, flow rerouting rarely introduces cycles, as we shall see in Section 2.5.2, and simple invocations of depth-first search suffice to detect and cancel cycles. As in the computation of the topological ordering, care is taken to ensure that no unnecessary time is spent on processing vertices that are not reachable from s.

It is well known [15] that depth-first search (DFS) can be used to classify the edges of a directed graph into three types: *tree edges*, which are edges that form the DFS tree; *forward edges*, which are edges that go from a vertex to a grandchild or lower descendant in the tree; and *back edges*, which are edges that go from a vertex to an ancestor. Given a directed flow graph, we run DFS starting from s to all reachable vertices, keeping a record of any back edges discovered, but ignoring any forward edges. If there are no back edges, the flow graph is already acyclic. Otherwise, each back edge wv is contained in the cycle C formed by it and the path from v to w in the DFS tree. We eliminate C by reducing the minimum flow along an edge in C to zero, and perform the cancellation process for all back edges. Note that it is possible that when a back edge is processed, the cycle that contained it might no longer exist in the flow graph if some of the tree edges forming the cycle have already been removed. After processing all the back edges, the flow graph might still have cycles caused by the forward edges that were ignored during DFS. We therefore repeat the DFS and back edge cycle elimination process until no back edges are found during DFS.

2.3.7 Initialization and Scaling of α

The concurrent flow approximation algorithm maintains a dual length function l that is dependent on the primal flow f. The length $l(v)$ of a vertex v is given by $l(v) = e^{\alpha f(v)}$ where α is a parameter that is determined at the beginning of each call to REDUCE. In Section 2.2.2, a value for α was given that bounds the gap in one of the inequalities relating the primal and dual objective functions. However, that value might be unnecessarily high. A high α results in a larger range of vertex lengths, leading to a greater incidence of underflow when lengths are scaled, and more instances where the best rerouting amount described in Section 2.3.3 has to be estimated. We develop in this section alternative lower values for α. These values, however, do not always guarantee a small gap in the desired primal-dual inequality, and we consider strategies for scaling α when it is determined to be invalid. Borger *et al* [11] used a similar strategy where they increase α by 0.25 whenever necessary. Their initial value of α was however not explicitly specified.

We first consider the conditions on α that are imposed to make the procedure REDUCE work. Recall that in Section 2.2.2, we assign to α the value $(7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7n\epsilon^{-1})$ at the start of each REDUCE call. We shall refer to this value as the *safe value for* α, since it ensures that relaxed optimality condition (R1) is always satisfied, which in turn guarantees that $|f||l|_1$ and $\sum_v f(v)l(v)$ are relatively close:

$$(1-\epsilon')|f||l|_1 \leq (1+\epsilon')\sum_v f(v)l(v). \tag{2.3}$$

In addition, the proof of Lemma 2.5 requires that $\alpha\sigma \leq (7+\epsilon)\epsilon/51$, which is true for the safe value of α and σ satisfying the technical granularity condition. To ensure that the bound on $\alpha\sigma$ is maintained, we will not choose a value for α that exceeds its safe value.

Alternative Values for α

Instead of having an α value that satisfies inequality (2.3) via its satisfaction of relaxed optimality condition (R1), we can pick an α value that satisfies the inequality without forcing it to fulfill (R1). We derive in the next lemma a smaller value of α that satisfies inequality (2.3) for the given flow f. Let n^* be the number of vertices whose flow is equal to the congestion $|f|$, and let $r = n - n^*$. We assume that $r > 0$, for otherwise (2.3) holds at equality.[8]

Lemma 2.9 *Let z^* be the unique solution to the equation $ze^z = r/(n^*e)$. If*

$$\alpha = \frac{1+\epsilon'}{2\epsilon'}\frac{z^*}{|f|},$$

then the length function $l(v) = e^{\alpha f(v)}$ will satisfy inequality (2.3).

Proof. Rewrite inequality (2.3) as

$$\frac{1-\epsilon'}{1+\epsilon'}|f| \leq \frac{1}{|l|_1}\sum_v f(v),$$

and observe that the above condition can be strengthened by making it hold for *any* flow in which n^* vertices have flow $|f|$ and the remaining vertices each has less than $|f|$ flow, as shown below:

$$\frac{1-\epsilon'}{1+\epsilon'}|f| \leq \min_{0\leq v_i<|f|, 1\leq i\leq r}\left[\frac{1}{n^*e^{\alpha|f|}+\sum_i e^{\alpha v_i}}(n^*|f|e^{\alpha|f|}+\sum_i v_i e^{\alpha v_i})\right]. \tag{2.4}$$

Making the change of variables $w_i = \alpha(|f| - v_i)$, we can simplify (2.4) to

$$\frac{1-\epsilon'}{1+\epsilon'}|f| \leq |f| - \frac{1}{\alpha}\max_{\vec{0}<\vec{w}\leq|\vec{f}|}\frac{\sum_i w_i e^{-w_i}}{n^*+\sum_i e^{-w_i}},$$

or equivalently,

$$\max_{\vec{0}<\vec{w}\leq|\vec{f}|} H(\vec{w}) \leq \frac{2\epsilon'}{1+\epsilon'}\alpha|f|, \tag{2.5}$$

where $H(\vec{w}) = \sum_i w_i e^{-w_i}/(n^* + \sum_i e^{-w_i})$. It can be shown that H has a unique global maximum $\vec{w^*}$ over $\Re^r$, where $\vec{w^*} = w^*(1,\ldots,1)$ and w^* is the unique solution to $(w-1)e^{w-1} = \frac{r}{n^*e} > 0$. In addition, $H(\vec{w^*}) = w^* - 1$. For $\alpha = \frac{1+\epsilon'}{2\epsilon'}\frac{w^*-1}{|f|}$, the right hand side of inequality (2.5) is equal to $H(\vec{w^*})$, and so (2.5) is satisfied.

In summary, the desired value of α is given by $\frac{1+\epsilon'}{2\epsilon'}\frac{z^*}{|f|}$, where z^* is the unique solution to $ze^z = \frac{r}{n^*e}$. □

We now show that the α-value in the above lemma is indeed smaller than the safe value $(7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7n\epsilon^{-1})$. Note that in the above lemma, $z^* > 0$ and that

[8] In the case where $r = 0$, we can choose any non-negative value for α. We arbitrarily use the value given by Lemma 2.9 for $r = 1$. This value of α corresponds to a small perturbation of the given flow which results in exactly one vertex with flow lower than the congestion.

ze^z is increasing for $z \geq 0$. At $z = \log(7n\epsilon^{-1})$, we have $ze^z = \log(7n\epsilon^{-1})7n\epsilon^{-1} > n > r/(n^*e)$, and so $z^* < \log(7n\epsilon^{-1})$. This upper bound on z^* shows that the safe value exceeds the value given in Lemma 2.9:

$$(7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7n\epsilon^{-1}) > \frac{7+\epsilon}{\epsilon}\frac{z^*}{\tau} \geq \frac{1+\epsilon'}{\epsilon'}\frac{z^*}{|f|} > \frac{1+\epsilon'}{2\epsilon'}\frac{z^*}{|f|}.$$

In practice the value from Lemma 2.9 is still too high. We derive an alternative value for α, but which is not guaranteed to satisfy inequality 2.3, by assuming that all vertices which do not have the maximum flow $|f|$ have the next highest vertex flow, denoted by $|f|_2$. As before, let n^* be the number of vertices with flow $|f|$, and let $r = n - n^*$. With the added assumption, inequality (2.3) expands to

$$(1-\epsilon')|f|[n^*e^{\alpha|f|} + re^{\alpha|f|_2}] \leq (1+\epsilon')[n^*|f|e^{\alpha|f|} + r|f|_2 e^{\alpha|f|_2}].$$

Simplifying the above gives

$$\alpha \geq \frac{1}{|f| - |f|_2}\log\left(\frac{r}{n^*}\left[\frac{|f|-|f|_2}{|f|}\frac{1+\epsilon'}{2\epsilon'} - 1\right]\right). \tag{2.6}$$

We choose α equal to the right hand side of 2.6. In the event that the right hand side is undefined due to a non-positive argument to the logarithm or is non-positive, we revert to the value of α given in Lemma 2.9.

A Scaling Scheme for α

We now consider the effect of using the smaller values of α that we have just described. Since the smaller values no longer ensure that relaxed optimality condition $(R1)$ holds, it is possible that as flow is rerouted, the modified flow and the length function based on the current choice of α no longer satisfy (2.3). When this happens, we increase α by scaling it by a predetermined factor γ until inequality (2.3) is restored. The scaling process will terminate since the results of Section 2.2.2 show that (2.3) holds whenever α reaches or exceeds its safe value.

The changes that must be made to the concurrent flow algorithm to accommodate the α-scaling scheme are as follows. Recall that τ, the target flow congestion, is halved each time Concurrent calls Reduce. Thus the safe value $\alpha = (7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7n\epsilon^{-1})$ doubles at each call to Reduce. We move the initialization of α out of Reduce, and Reduce now takes α as an input and could return with an indication that the given α is no longer sufficiently large to make inequality 2.3 for the modified flow. The modified Reduce procedure is shown below, with the changes made to flow quanta and flow rerouting amount as described in Section 2.3.3 is shown below.

Reduce(f, τ, ϵ, σ, α)

while $|f| \geq \tau$
  for each vertex v: $l(v) \leftarrow e^{\alpha f(v)}$

> **if** f and l are ϵ-optimal**then**
>> **return** f and message "ϵ-optimal"
>
> **elseif** (2.3) is not satisfied **then**
>> **return** f and message "low α"
>
> **else**
>> find an ϵ-bad flow path P and a short path Q for some commodity
>> reroute a multiple of σ units of flow from P to Q
>
> **endif**

endwhile
return f and message "congestion reduced"

The modified CONCURRENT algorithm first computes an initial flow and sets α to one of the alternative values given earlier. The algorithm then calls REDUCE repeatedly, scaling α upwards if REDUCE concludes that the current value of α is too low. Note that it should never happen that REDUCE concludes that α is too low if α reaches or exceeds its safe value. In practice, the values of α encountered are significantly lower than their safe values. We give below the modified CONCURRENT algorithm with the flow quanta and rerouting changes of Section 2.3.3 incorporated.

CONCURRENT(G, ϵ, $\{d(i), (s_i, t_i) : 1 \leq i \leq k\}$)

for each commodity i
    route $d(i)$ flow from s_i to t_i along a simple path
end for
$\tau \leftarrow |f|/2$
$\sigma \leftarrow \frac{1}{2}$
$\alpha \leftarrow$ value from Lemma 2.9 or (2.6)
$\alpha_{safe} \leftarrow (7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7n\epsilon^{-1})$
pick a scaling factor $\gamma > 1$
while f is not ϵ-optimal
    while σ and τ do not satisfy the granularity condition
        $\sigma \leftarrow \sigma/2$
    endwhile

    call REDUCE(f, τ, ϵ, σ, α)
    if REDUCE returned "congestion reduced" **then**
        $\tau \leftarrow \tau/2$, $\sigma \leftarrow \sigma/2$
        $\alpha \leftarrow \alpha * 2$, $\alpha_{safe} \leftarrow \alpha_{safe} * 2$
    elseif REDUCE returned "low α" **then**
        if $\alpha = \alpha_{safe}$ **then** exit with error
        $\alpha \leftarrow \alpha * \gamma$
    endif
endwhile

The scaling factor γ is chosen to be either $\sqrt{2}$ or 2. We can in fact have much lower scaling factors or instead increase α by an additive constant as was chosen by Borger *et al* [11] each time it is determined to be too low. We examine in Section 2.4.4 the effects of α-scaling on the performance of the concurrent flow algorithm.

2.3.8 Routing of Initial Flow

The concurrent flow algorithm constructs an initial feasible flow for the given problem by routing $d(i)$ units of flow along a simple path from the source to the sink of each commodity i. We consider in this section choices for the initial flow paths, and other enhancements to the routing of the initial flow.

A simple way to select the initial flow path for a commodity i is to use a minimum-hop path, i.e. a path with the fewest number of vertices on it. Such a path can be computed using breadth-first search from s_i. Another more sophisticated way would be to pick a shortest path from s_i to t_i, using a length function that changes as flow for each commodity is routed. We choose an exponential length function that is similar to that used in REDUCE. This approach can be pushed further by routing only a fraction of the demand $d(i)$ of each commodity at each step and then updating the length function, thus potentially allowing flow to be spread out more evenly. We have experimented with routing all, $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{8}$ of the demand along each shortest path.

The alert reader will realize that the sophisticated shortest-path based initial routing scheme essentially uses the ideas behind rerouting in REDUCE. It is conceivable then that "smarter" initial flows would be less amenable to improvement by rerouting, and we shall examine this effect in Section 2.4.5.

2.3.9 Commodity Selection Rules

The time bounds given by Klein, Plotkin, Stein and Tardos [52] for an iteration of REDUCE is worst-case in that it assumes that all commodities have to be examined before a commodity with an ϵ-bad longest path is found. In practice, the worst case seldom occurs, and we give in this section some selection rules for deciding which commodities to examine. The performance of these rules will be compared in Section 2.4.3. We shall refer to a commodity that has an ϵ-bad longest path as an ϵ-bad commodity, and one that does not as ϵ-good. Recall that k denotes the number of commodities.

The simplest and fairest commodity selection rule is round-robin, in which each commodity is tried in turn. We assume that the commodites are ordered by their indices. At the first rerouting step of each REDUCE call, we check commodities $1, 2, \ldots, k$ in sequence for ϵ-badness until one such commodity is found, and then rerouting is performed. Each subsequent rerouting step looks for an ϵ-bad commodity starting from the commodity *ordered after* the one rerouted in the previous step, and wrapping around to commodity 1 when the last commodity k has been examined. The starting point for commodity searching is reset to 1 at each call to REDUCE. By ordering commodities so that those with the same source are grouped together, the round-robin selection scheme works well with the incremental shortest path computation described in Section 2.3.5.

A second commodity selection rule is different from round-robin in that it selects commodities randomly. To exploit the savings from incremental shortest path computation, this *random-source sequential-sink* approach first randomly selects a source vertex, and then tries in sequence each commodity with that source vertex for ϵ-badness. If none are found, another source vertex is randomly chosen and the process repeated.

A third commodity selection rule involves weighting the commodities and randomly selecting a commodity according to its weight. This weighted-random scheme was introduced by Borger *et al* [11], and is implemented by storing the weights in a tree, allowing both updating of weights and selection of commodities in $O(\log k)$ time. The tricky part of this rule is the choice of weights. Essentially we want the weight assigned to a commodity to reflect how much it decreased the potential function $|l|_1$ each time it was selected for rerouting, but there are some difficult situations. For example, it is not clear how to assign weight to a commodity that resulted in a large decrease in the potential function the last time it was chosen, but is ϵ-good this time round. Borger *et al* [11] assign to each commodity i a weight equal to some small constant plus the difference between the longest and shortest path length for i the last time it was examined. We use a more complicated weighting scheme involving a decaying average of the decrease in potential function each time the commodity was examined.

It is tempting to carry the weighted commodity idea further and choose the commodity with the highest weight for consideration in rerouting. This is easily accomplished by using a heap instead of a tree to hold the commodity weights. However this greedy rule does not guarantee that each commodity will eventually be considered, unless special care is taken to gradually increase the weight of all other commodities. For example, in one of our initial experiments using this rule, an infinite loop was created by two ϵ-good commodities alternatively being chosen. We therefore abandoned the greedy rule.

2.3.10 Testing of Stopping Criteria

The enhanced REDUCE procedure from Section 2.3.7 has three possible outcomes: it either concludes that the current flow f is ϵ-optimal, or that the congestion $|f|$ has dropped below the target value τ, or that the supplied value of α no longer satisfies inequality (2.3). Checking that the congestion $|f|$ is at most τ is an inexpensive $O(n)$ operation, as is verifying if α still makes inequality (2.3) holds.

$$(1-\epsilon')|f||l|_1 > (1+\epsilon')\sum_v f(v)l(v),$$

We can therefore test those two terminating conditions after each rerouting step.

The ϵ-optimality condition

$$|f||l|_1 \leq (1+\epsilon)\sum_i dist_l(s_i,t_i)d(i)$$

is however much more computationally intensive to verify. Its verification involves n shortest path computations to determine the shortest path distances for all commodities, and therefore takes time $O(n(m+n\log n))$ We reduce the total time spent on checking this condition by only verifying it when necessary. As long as the flow is not ϵ-optimal and the other two terminating conditions are not met, there will be an ϵ-bad path for some commodity. However, when the flow becomse ϵ-optimal, ϵ-bad paths might no longer exist. So we only check the ϵ-optimal terminating condition whenever we are unable to find a commodity with an ϵ-bad path after trying many commodities. At present, we trigger the check if $20n$ commodities have been tried without success, where n is the number of vertices in the input graph. Borger *et al* use a similar triggering scheme where the trigger point is instead proportional to the number of commodities.

2.4 Tuning the Implementation

In the previous sections, we examined various aspects of the implementation of the concurrent flow algorithm. Some implementation decisions were, however, left open. We empirically explore in this section those unresolved issues and select an implementation setting that is good for the typical problems that we shall encounter. We then show that the implementation is superior to linear programming solution methods in Section 2.5.1, and give further detailed statistics on the computed ϵ-optimal flow in Section 2.5.2. Due to the length of this section, the reader may wish to skip it on a first reading. Some of the insights and statistics from the tuning process are repeated in Section 2.5.

2.4.1 Testing Methodology

There are several implementation decisions that were left open in Section 2.3. They are:

- when to check for ϵ-optimality in the routine Reduce,
- type of shortest path computation (exact or approximate),
- rule for selecting a commodity in searching for ϵ-bad paths,
- scaling factor for the length parameter α,
- and the routing of initial flow.

We refer to a particular combination of options for the implementation parameters above as a *setting* of the implementation, and our goal is to find a good setting.

Test Cases

The best setting is, of course, dependent on the inputs that we run the algorithm on, since it is unlikely that there is a setting that is optimal for all instances of the concurrent flow problem. We therefore restrict our attention to the concurrent flow instances that we shall encounter. Recall from Section 1.6 that the elimination ordering algorithm employs a recursive divide-and-conquer approach based on vertex cuts, and that the construction of vertex cuts requires the solution to instances of the concurrent flow problem. At the top level of the recursive process, the input graphs of the concurrent flow instances from the vertex cut construction procedure are exactly the inputs to the elimination ordering algorithm, viz. graphs of symmetric positive definite matrices.[9] We choose 20 matrices from the standard Harwell-Boeing collection of sparse matrices [18] [19], with the smallest matrix having 96 vertices and the largest 1454. As for the commodity specification, the vertex cut construction creates for each vertex v of the input graph a small constant number of unit-demand commodities with v as the source vertex and with randomly chosen sinks.[10] We adopt that approach to generate one set of commodities for each input matrix, with each vertex being the source of 3 commodities. Finally, we set the optimality error parameter ϵ to 0.1, which is the value used in the vertex cut construction. Thus we have 20 typical instances of the concurrent flow problem for evaluating the various settings of the implementation. We believe that there is nothing special about choosing ϵ to be 0.1 or having 3 (instead of some higher constant) commodities per source vertex, and so our conclusion on the best setting is unlikely to change if those problem parameters were altered.

Table 2.1 summarizes the characteristics of the test cases that we use in tuning the implementation. For each matrix, we list the number of vertices and edges in its associated graph, as well as the number of commodities, the average degree of a vertex, and the hop diameter of the graph. The latter quantity is the maximum number of edges in a shortest path between any two vertices in the graph, and gives an indication of how far apart vertices are in the graphs. We see that the test matrices have a wide range of average vertex degrees and diameters, and constitute a sufficiently diverse set of inputs for tuning the implementation.

Difficulties with Checking for Optimality

Before developing a framework for comparing various implementation settings, we first consider in detail the detection of ϵ-optimality of the concurrent flow and dual lengths constructed by the concurrent flow algorithm. We shall see that there are several difficulties associated with checking for optimality in practice, and that

[9] At lower recursion levels, the inputs graphs to the concurrent flow algorithm are vertex-induced subgraphs of the graphs of symmetric matrices, possibly with additional edges introduced. We do not include these subgraphs in our tests for they are dependent on the execution of the elimination ordering algorithm, and accounting for all possible subgraphs creates a huge set of test cases.

[10] Randomly generated commodities with the same source and sink are aggregated into a single commodity with higher demand.

Table 2.1: Test matrices used in tuning the implementation.

Matrix	n	m	k	Deg	Diam
CAN96	96	336	281	7.0	8
CAN445	445	1682	1334	7.6	12
CAN715	715	2975	2139	8.3	14
BCSSTK05	153	1135	457	14.8	12
BCSSTK06	420	3720	1258	17.7	16
BCSSTK19	817	3018	2448	7.4	86
BCSPWR05	443	590	1326	2.7	20
BCSPWR06	1454	1923	4355	2.6	36
DWT193	193	1650	578	17.1	7
DWT419	419	1572	1253	7.5	22
DWT878	878	3285	2633	7.5	40
NOS4	100	247	297	4.9	13
NOS5	468	2352	1402	10.1	11
NOS7	729	1944	2184	5.3	24
LSHP265	265	744	794	5.6	24
LSHP406	406	1155	1216	5.7	30
LSHP778	778	2247	2330	5.8	42
WILL199	199	660	592	6.6	5
FS541.1	541	2466	1621	9.1	2
BP1000	822	4635	2462	11.3	6

these difficulties preclude the possibility of having practical optimality checks when comparing implementation settings.

The fundamental problem with checking for ϵ-optimality is that such checks require k shortest path computations to evaluate the dual objective value and are computationally expensive as compared to a single rerouting step. Therefore it is too costly to test for optimality at every step; it was estimated that a single run on a large test matrix would take more than 30 hours in such a situation. However, having infrequent checks may result in many more reroutes being made than are necessary to achieve ϵ-optimality, and the observed number of reroutes and the running time may be significantly higher than the true number of reroutes and running time needed for an ϵ-optimal flow. Such a high uncertainty in the observed number of reroutes and the running time is unacceptable in the context of tuning the implementation, for it renders meaningless any comparison of implementation settings. A practical optimality checking scheme has to strike a balance between the two opposing concerns. We assessed the feasibility of practical optimality checks by evaluating a scheme where we tested for optimality periodically, after every $20n$ reroutes, and whenever k ϵ-good commodities have been examined in succession while looking for an ϵ-bad commodity to reroute. It turned out that there were some matrices which required only $20n$ or $40n$ reroutes to achieve ϵ-optimality, i.e. only one or two optimality checks were performed. For those cases, the timing uncertainty could be arbitrarily high (for matrices needing $20n$ reroutes) or up to 50% (for matrices needing $40n$ reroutes). Even when optimality was tested for more frequently, after every n reroutes and whenever $k/10$ ϵ-good commodities have been examined in succession, there were still 9 matrices (out of 20) for which 10 or fewer optimality checks were performed, with an associated uncertainty of 10% in the measured running time. We concluded that any practical optimality checking scheme will introduce significant timing uncertainties on some of the inputs.[11]

Idealized Optimality Checking

We shift our focus to comparing the various implementation parameters in an idealized optimality checking scenario, where the algorithm terminates soon after ϵ-optimality is reached and where the time spent on optimality checks is not counted. Only when the best setting of implementation parameters is determined do we then worry about checking for optimality in practice.[12] One idealized approach would be to first accurately determine the number of reroutes using a "binary search" process where we repeatedly halve the interval between optimality checks and also avoid optimality checks in the initial part of the run by using information from

[11] We rejected the option of discarding the matrices with low number of reroutes as we felt that they were representative of some of the inputs that we would run the concurrent flow algorithm on.

[12] It is plausible that the best way of checking for optimality would vary with the choice of implementation setting, giving yet another reason why it would not be feasible to compare implementation settings with practical optimality checking in place.

the previous run to get a lower bound on the number of reroutes needed. Once we know the number of reroutes, we can perform a run where all optimality checks are turned off and which terminates after the (pre)determined number of reroutes, and use that time in comparing implementation settings.

The search process implicitly assumes that the convergence towards ϵ-optimality is smooth, i.e. if an optimality check after r reroutes does not indicate ϵ-optimality, then it is very unlikely that ϵ-optimality occurred after some earlier number of reroutes. A closer examination of the runs, however, shows that the assumption is false. At each optimality check, we record the primal objective value (i.e flow congestion) and the dual objective value (i.e. normalized sum of shortest path distances) of the current solution. Figure 2.1 shows the behavior of the primal and dual objective values in a typical instance as reroutes are made, where optimality is checked after every 93 reroutes.

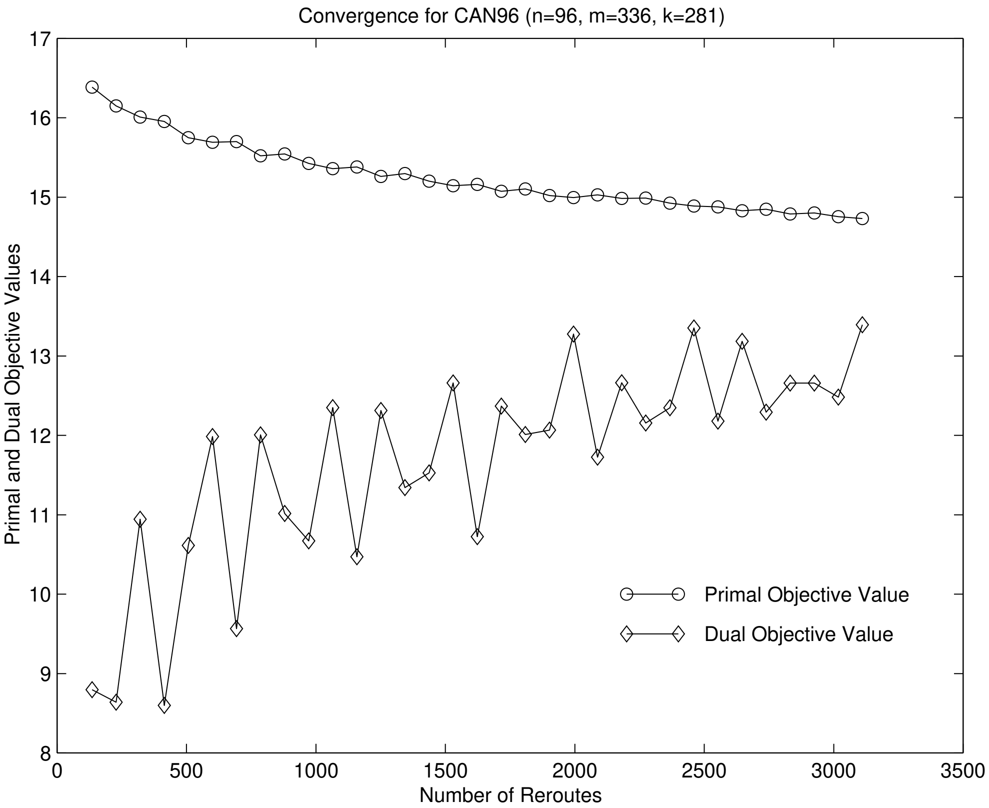


Figure 2.1: Convergence to Optimality for Matrix CAN96

From the figure, we see that although the primal objective value decreases quite smoothly as reroutes are made, the behavior of the dual is more erratic. There is no strong correlation between how close a primal solution and its associated dual length solution are to optimality. In fact the figure indicates that the quality of the dual objective value plays a bigger role in establishing ϵ-optimality than does the primal objective value. Borger *et al* [11] had observed a similar phenomenon in their evaluation of their implementation of the edge-based concurrent flow algorithm, where they found that decreasing the error parameter ϵ from 0.1 to 0.01 and 0.001 often led to much better dual solutions but hardly improved the primal solution.

We can "smooth" out the fluctuations in the dual solution by saving the best dual objective value evaluated so far and comparing that against the primal objective value during an optimality check. Such a scheme can potentially lead to an earlier detection of ϵ-optimality, and indeed does so for the preliminary runs, with the number of reroutes reduced by 5% to 10% in typical cases, and by up to 75% in the more extreme cases. However we still cannot exclude the possibility that the dual objective value might actually be *higher* at some point in the run where it was *not evaluated*, thus missing an opportunity to terminate the algorithm earlier. This last observation shows that the "binary search" process of accurately determining the number of reroutes needed is not reliable.

We therefore adopt another approach of performing idealized runs of the algorithm. The previous discussion shows that using the dual objective value to determine ϵ-optimality is not reliable due to fluctuations. However the primal objective value (the flow congestion) is much better behaved, and we can instead perform runs where there are no optimality checks and which stop once the flow congestion drops below a predetermined bound. This bound is determined as follows. Each run in the preliminary tuning effort concluded with a flow that is ϵ-optimal. For each matrix, we can take the maximum of the final flow congestion over all the runs, and this maximum value will be within $(1+\epsilon)$ of the optimal flow congestion, giving us the desired bound. In fact, any of the final flow congestion could serve as the bound, but taking the maximum ensures that we use the least restrictive value from the available data.

Comparing Implementation Settings

Having constructed a mechanism for terminating the algorithm, we can now compare the various implementation parameters. Since it is prohibitive to evaluate all possible implementation settings, we instead perform the following seqeuence of comparisons:

(1) Compare the various commodity selection rules in conjunction with exact and approximate shortest path computations, using an α-scaling factor of 2 and a min-hop initial flow.

(2) Using the best combination of commodity selection rule and shortest path computation method from (1), and a min-hop initial flow, compare the various α-scaling factors.

(3) Using the best combination of commodity selection rule and shortest path computation method from (1) and the best α-scaling factor from (2), compare the various initial flow routing methods.

After the best setting has been determined, we then revisit the issue of checking for optimality in practice.

The above sequence of comparisons assumes that the performance effects of the α-scaling factor and the routing of initial flow are independent of each other and of the commodity selection rule and the shortest path computation method. This assumption is reasonable due to a separation of level of "detail": the last two parameters affect the computation within a REDUCE call, whereas the α-scaling factor works across REDUCE calls, and the routing of initial flow can be considered as separate from the rerouting phase.

The cautious reader will no doubt question the tuning of the implementation in an idealized optimality checking situation. From a practical viewpoint, as we have previously discussed in detail, there is no meaningful way to compare the implementation parameters in a realistic framework with practical optimality checks, since there would be significant timing measurement uncertainities. Without a fuller understanding of the fluctuations in the dual objective value and the weak correlation between the quality of the primal and dual solution, even an idealized comparison that relies on the dual objective value for termination would be subject to uncertainty in the measurement of the number of reroutes needed. Our mechanism of precomputing a bound for the flow congestion and stopping the algorithm whenever that bound is reached results in much less uncertainty in the measured number of reroutes since the flow congestion decreases smoothly as reroutes are made. Although we are now reduced to evaluating only one facet (viz. the primal) of the convergence to ϵ-optimality of the concurrent flow algorithm, our simple termination mechanism is unlikely to be biased towards any implementation setting, and so using it in comparing settings is fair.

In the following sections, we evaluate in detail the various implementation parameters according to the comparison methodology proposed above. Note that all the commodity selection rules, with the exception of round-robin, are randomized. For settings involving randomized commodity selection, we run each test case 5 times and make comparisons using the average of the quantities obtained in those 5 runs. Whenever we refer to some quantity, eg. running time, for a randomized setting, we shall mean the average of that quantity over the 5 runs. All the runs for tuning the implementation are performed on a high-end 4-processor Sparc workstation.

2.4.2 Comparison of Exact and Approximate Shortest Path Computation

The first set of runs that we performed were for the various commodity selection rules in conjunction with exact or approximate shortest path computation, using an α-scaling factor of 2 and a min-hop routing of the initial flow, as described in the methodology from the previous section. For each commodity selection rule, we compare the performance of the versions using exact and approximate shortest path computations. We shall see that the exact shortest path version is almost always faster than the approximate version for *each* commodity selection rule, and that an approximate shortest path computation is slower than its exact counterpart due to the costly operation of discretizing the vertex lengths.

Comparison of Running Time

The various commodity selection rules that were described in Section 2.3.9 are as follows:

- round-robin (labelled RoundRobin),
- random-source sequential sink (labelled RandSeq),
- weighted-random selection, where the weight assigned to a commodity is related to the difference between its longest and shortest path lengths (labelled DistWtRand),
- and weighted-random selection, where the weight assigned to a commodity is related to the relative decrease in the potential function when it was rerouted (labelled PotWtRand).

For each commodity selection rule, we compute the time taken by the approximate shortest path version divided by the time taken by the exact shortest path version, and those ratios are plotted for the various test matrices in Figure 2.2.

In the figure, we have drawn two threshold lines at the ratios 0.8 and 1.25. For ratios that fall within this range, the running time of the slower version is no more than 25% greater than that of the faster version. We see that the exact version is faster for most of the cases. In fact there are only 15 cases (out of 80) where the approximate version is faster, of which 3 are more than 25% faster than the exact version. On the other extreme, there are 18 cases where the approximate version is more than 25% slower than the exact version. In addition, the exact version is faster across all commodity selection methods for 11 of the matrices (CAN715, BCSSTK06, BCSPWR05, BCSPWR06, NOS4, NOS7, LSHP275, LPSH406, LSHP778, FS541.1 and BP1000).

Comparison of Rerouting Quality

It is interesting to delve further into the performance differences between the two shortest path computation methods. Figure 2.3 compares the number of reroutes taken by the approximate version to that taken by the exact version.

As before we have drawn the threshold lines at the ratios 0.8 and 1.25. We see that the figure looks similar to the earlier one on running times, especially for the extreme cases. There are 29 cases where the approximate version had fewer reroutes, of which 7 had fewer than 80% of the reroutes of the exact version. On the other extreme, there are 10 cases where the approximate version had more than 25% extra reroutes than the exact version.

We can regard the number of reroutes taken by a particular implementation setting as a reflection of the quality of the reroutes that it makes, i.e. how much each reroute decreases the potential function used to measure progress. The figure shows

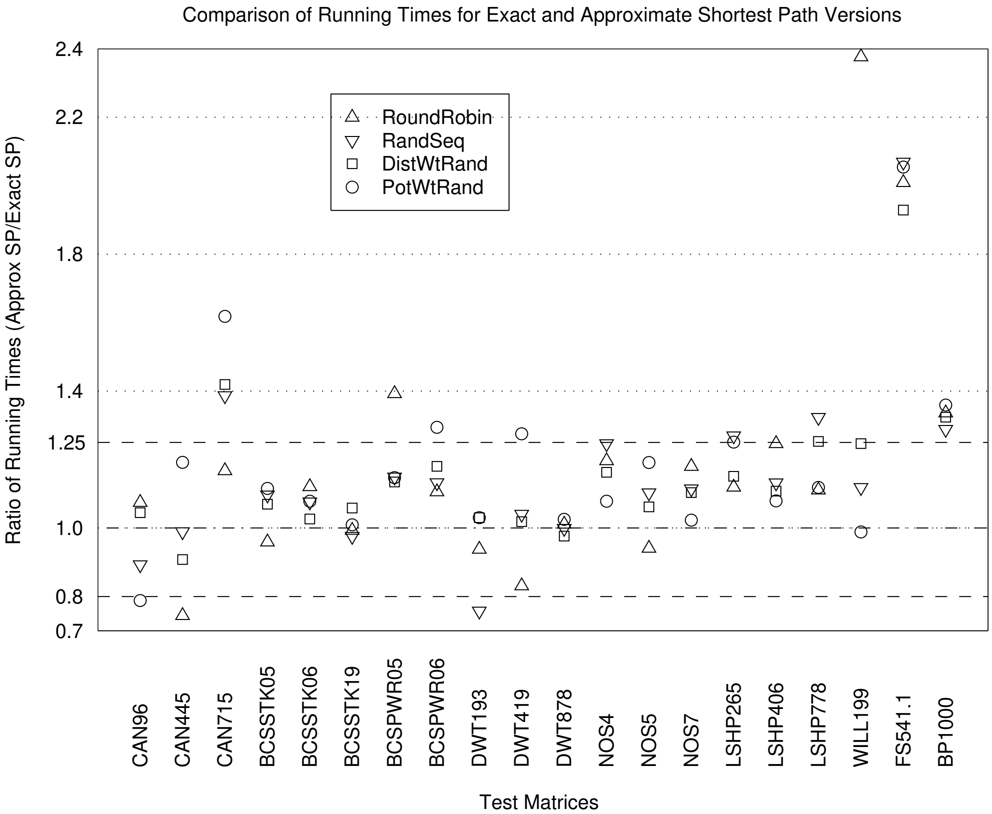


Figure 2.2: Ratio of time taken by approximate shortest path version to time taken by exact shortest path version for the various commodity selection rules.

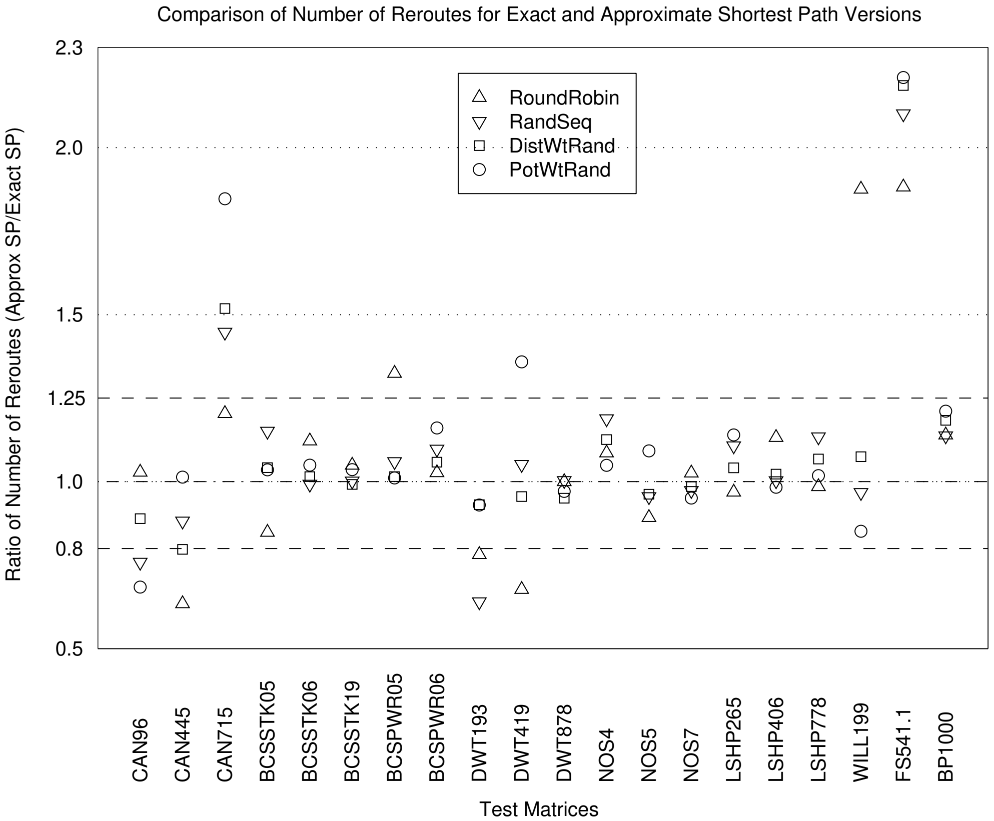


Figure 2.3: Ratio of number of reroutes taken by approximate shortest path version to number of reroutes taken by exact shortest path version for the various commodity selection rules.

that using approximate shortest paths often results in poorer rerouting quality, which can be attributed to the fact that flow is being rerouted from a longest path to an almost shortest path instead of to a shortest path.

The number of reroutes does not exactly correspond to the number of shortest path computations that are performed. This is because during each reroute, one or more commodities are examined according to the commodity selection rule, and evaluating each commodity involves one shortest path computation. In Figure 2.4, we compare the total number of commodities examined, or equivalently the number of shortest path computations performed, by the approximate shortest path version to that examined by the exact version.

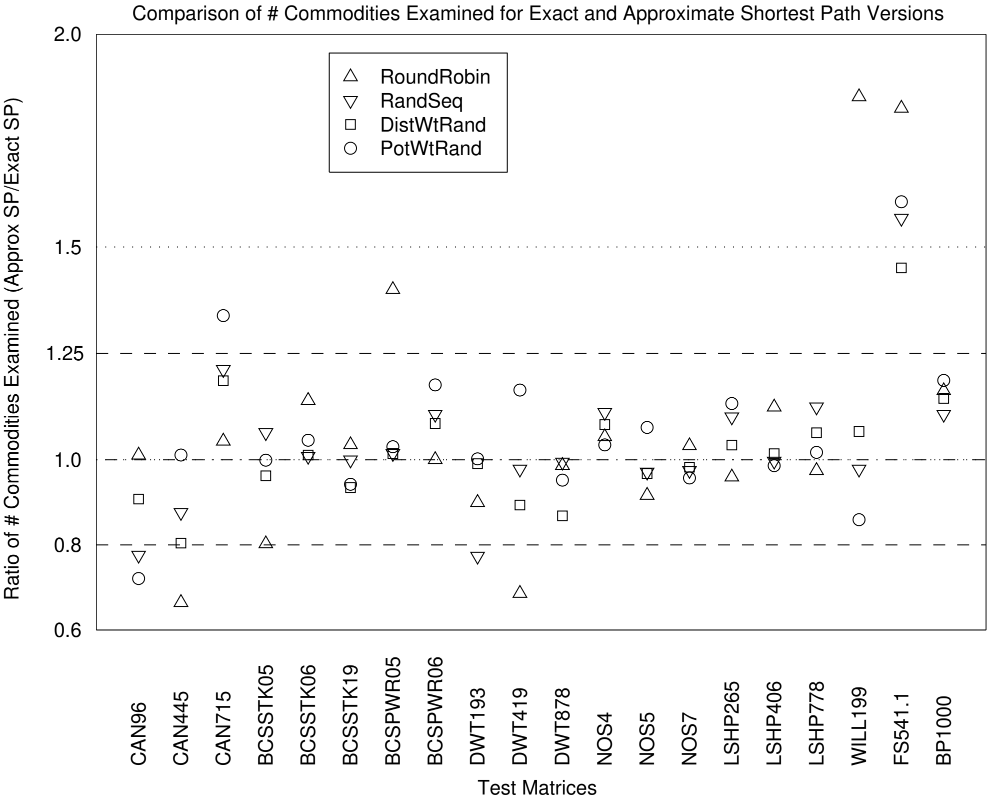


Figure 2.4: Ratio of number of commodities examined by approximate shortest path version to number of commodities examined by exact shortest path version for the various commodity selection rules.

Unlike the previous two comparisons, the exact version is only slightly better than the approximate version with respect to the number of shortest path computations performed. In close to half (34 out of 80) of the cases, there were fewer approximate shortest path computations than exact ones. The number of extreme cases is also quite balanced: there are 7 cases where the number of approximate shortest path computations was more than 25% higher than the number of exact ones, and 5 cases for the other extreme. This suggests that the higher running time for the approximate version is due to each approximate shortest path computation taking longer time than an exact computation.

Runtime Profiles of Shortest Path Computations

By profiling the implementation, we can measure the actual time spent in the shortest path subroutines. Recall from Section 2.3.5 that our shortest paths implementation is incremental, and that no unnecessary recomputation is performed if two successive shortest path computations are from the same source vertex. There are therefore two average shortest path computation times that we can measure. One is the time per sink vertex (i.e. per commodity examined) and the other is the time per source vertex (i.e. per shortest path tree constructed). Table 2.2 gives various timing quantities for the two shortest path methods for the runs where the round-robin commodity selection was used.[13] The first four columns give the name of the matrix, the number of vertices and edges in its associated graph, and the number of commodities. The next two columns give the average time in milliseconds taken per source and per sink vertex for the exact shortest path version, while the remaining columns give corresponding times for the approximate version, with relative comparisons to the exact version.

The table shows, somewhat surprisingly, that approximate shortest path computations always take longer than the exact version. For the approximate shortest paths, the time per source vertex is between 7% to 60% greater than that of the exact version, while the time per sink vertex is between 10% to 61% greater. We investigate further into the causes for the inferior performance of the approximate shortest path computation. Table 2.3 compares the heap operations for the exact and the approximate shortest path computations. The second and third columns give the average time in microseconds for the heap operations of insertion and deletion of the minimum element[14] and the fourth column lists the average number of heap operations per shortest path tree, for the exact shortest path computations. The remaining columns give corresponding quantities for the approximate version, with relative comparisons to the exact version.

From the table, we see that the heap operations are significantly faster in the approximate version (except for 2 instances in DelMin). This is to be expected since for the approximate version, we use an integer-valued heap to store the discretized distances, whereas in the exact version, we need a real-valued heap, and integer arithmetic operations are faster than floating-point arithmetic operations. The table also shows that the two versions perform almost the same number of heap operations, which indicates that their shortest path trees are of almost equal size. Thus the heap operations do not cause the approximate shortest path computation

[13] We only profiled the runs for the round-robin commodity selection rule. It is unlikely that the relative behavior of the approximate and exact shortest path subroutines would be different for other commodity selection rules.

[14] Recall that unlike the heaps needed in a typical implementation of Dijkstra's shortest path algorithm, our heaps do not have to support the DecreaseKey operation which allows distance estimates for vertices to be updated.

Table 2.2: Average time per source and sink vertex in exact and approximate shortest path computations.

Matrix	n	m	k	Exact SP time (ms) per		Approximate SP time (ms) per	
				source	sink	source	sink
CAN96	96	336	281	0.27	0.24	0.43 (+60%)	0.38 (+61%)
CAN445	445	1682	1334	1.58	1.39	2.06 (+30%)	1.79 (+29%)
CAN715	715	2975	2139	3.46	1.61	3.69 (+7%)	1.87 (+16%)
BCSSTK05	153	1135	457	0.54	0.31	0.66 (+22%)	0.39 (+25%)
BCSSTK06	420	3720	1258	2.08	1.85	2.48 (+19%)	2.17 (+18%)
BCSSTK19	817	3018	2448	3.00	1.60	4.66 (+56%)	2.52 (+58%)
BCSPWR05	443	590	1326	1.32	0.83	1.86 (+41%)	1.16 (+39%)
BCSPWR06	1454	1923	4355	4.56	2.68	7.04 (+54%)	4.14 (+55%)
DWT193	193	1650	578	0.85	0.46	1.03 (+22%)	0.56 (+23%)
DWT419	419	1572	1253	1.52	0.92	1.94 (+28%)	1.13 (+23%)
DWT878	878	3285	2633	3.95	1.73	5.90 (+50%)	2.58 (+49%)
NOS4	100	247	297	0.30	0.21	0.43 (+42%)	0.29 (+40%)
NOS5	468	2352	1402	1.83	1.49	2.08 (+14%)	1.63 (+10%)
NOS7	729	1944	2184	2.47	2.27	3.25 (+32%)	2.99 (+32%)
LSHP265	265	744	794	0.88	0.76	1.24 (+40%)	1.07 (+41%)
LSHP406	406	1155	1216	1.66	1.45	1.83 (+10%)	1.62 (+12%)
LSHP778	778	2247	2330	2.73	2.40	3.55 (+30%)	3.17 (+32%)
WILL199	199	660	592	0.55	0.45	0.79 (+43%)	0.64 (+42%)
FS541.1	541	2466	1621	1.97	1.41	2.84 (+44%)	2.02 (+44%)
BP1000	822	4635	2462	2.83	2.28	3.87 (+37%)	3.06 (+34%)

Table 2.3: Average times for heap operations and number of heap operations per exact and approximate shortest path trees.

Matrix	Exact Shortest Path			Approximate Shortest Path		
	time (μs) per		# heap	time (μs) per		# heap
	Insert	DelMin	ops	Insert	DelMin	ops
CAN96	1.6	1.7	96.4	0.9 (−44%)	1.7 (−5%)	94.1 (−2%)
CAN445	1.1	3.5	419.7	0.8 (−22%)	2.6 (−26%)	417.8 (−0%)
CAN715	1.1	3.0	917.3	0.8 (−29%)	2.3 (−23%)	867.7 (−5%)
BCSSTK05	1.2	1.5	180.9	0.5 (−63%)	2.2 (+41%)	161.5 (−11%)
BCSSTK06	1.2	3.4	410.2	0.9 (−29%)	2.4 (−31%)	413.4 (+1%)
BCSSTK19	1.2	2.7	1131.6	0.7 (−40%)	2.1 (−21%)	1054.8 (−7%)
BCSPWR05	1.4	2.2	573.4	0.6 (−60%)	1.8 (−15%)	549.9 (−4%)
BCSPWR06	1.3	2.5	1891.6	0.8 (−38%)	2.1 (−17%)	1753.9 (−7%)
DWT193	1.5	3.4	201.2	0.7 (−51%)	2.7 (−23%)	186.2 (−7%)
DWT419	1.4	2.6	502.7	0.6 (−61%)	1.9 (−29%)	488.6 (−3%)
DWT878	1.4	2.7	1233.7	0.8 (−42%)	2.5 (−9%)	1206.3 (−2%)
NOS4	1.1	1.3	109.1	0.9 (−24%)	1.6 (+20%)	106.8 (−2%)
NOS5	1.2	3.9	439.3	0.8 (−30%)	2.4 (−38%)	447.2 (+2%)
NOS7	0.9	3.4	712.0	0.7 (−22%)	2.5 (−28%)	704.0 (−1%)
LSHP265	1.2	2.5	280.2	0.8 (−28%)	1.8 (−26%)	277.0 (−1%)
LSHP406	1.3	3.9	428.9	0.8 (−36%)	1.9 (−51%)	421.1 (−2%)
LSHP778	1.0	2.8	822.1	0.8 (−21%)	2.2 (−22%)	812.3 (−1%)
WILL199	1.2	3.4	165.3	0.9 (−24%)	2.7 (−20%)	166.0 (+0%)
FS541.1	1.4	2.7	581.3	0.9 (−34%)	2.4 (−13%)	583.6 (+0%)
BP1000	1.5	4.7	625.3	1.0 (−35%)	3.3 (−30%)	637.2 (+2%)

to be slower (in fact the opposite).

The reason why the approximate shortest path computation is slower turns out to be rather unusual. As remarked above, the approximate version uses discretized distances, which result from discretizing the vertex lengths $l(v)$ by rounding them up as follows:

$$\hat{l}(v) = \frac{\epsilon' l(P)}{4n} \left\lceil \frac{l(v) \cdot 4n}{\epsilon' l(P)} \right\rceil .$$

In the above formula, $l(P)$ is the length of a longest path from the source vertex to the sink vertex of the commodity being examined. It turns out that this extra step of computing the discretized length function is relatively expensive since it involves floating-point to integer conversion. In fact, the time spent discretizing the lengths typically accounts for 20% of the total time in an approximate shortest path computation, and wipes out the performance gains from the faster integer-valued heap used.

Choosing Between Exact and Approximate Shortest Paths

We saw in the preceding discussion that the overall running time for the approximate version is usually higher than that for the exact version across all four commodity selection rules. In addition, the approximate version usually has lower rerouting quality, and the time for an approximate shortest path computation is always higher than that for an exact computation. The last observation is explained by the high cost of floating-point to integer conversion during the discretization of vertex lengths.

It is conceivable that the runtime performance of approximate shortest path computation can be improved by maintaining a set of discretized vertex lengths, thus avoiding the need to discretize all the vertex lengths at the start of each shortest path computation. The easiest way to do so would be to only rediscretize vertex lengths which have changed after a reroute. This solution has two complications.

The first complication is that the discretization unit, currently set at $\epsilon' l(P)/4n$, depends on the longest path length $l(P)$ of the commodity being examined. For the proposed scheme to work, the unit for discretization has to remain *fixed* over a sequence of reroutes so that past discretizations are still valid. We therefore have to use a unit that is no bigger than that derived from all possible longest path lengths, in order to ensure that the computed approximate shortest paths are within the error bounds prescribed in Section 2.3.5. Choosing an appropriate unit is tricky, and requires carefully lower bounding the longest path lengths over all commodities. A unit that is too low will most likely result in *integer* overflow when discretizing some vertex length because of the exponential range of vertex lengths. On the other hand, a unit that is too high and not based on an actual lower bound will not be valid when some commodity with a relatively short longest path is examined. At that point, the unit of discretization would have to be readjusted, forcing all vertex lengths to be rediscretized.

The second complication is the fact that the actual vertex lengths depend not only on the flow through the vertices, but also on the flow congestion, which is used to determine a scaling factor for controlling numerical overflow (cf. Section 2.3.2). Thus a reroute which changes the flow congestion would affect all the vertex lengths. The overflow control scheme would have to be changed to use a scaling factor derived instead from an upper bound on the flow congestion. Note that the upper bound has to be reasonably tight, for otherwise the scaling factor would be excessively large, and the companion problem of scaled vertex lengths underflowing would occur frequently. It might therefore be necessary to adjust the upper bound and rediscretize all the vertex lengths whenever the flow congestion has decreased significantly since the last adjustment.

Given the above complications and the attendant overflow/underflow scenarios, we decided not to improve the discretizing scheme to make the approximate shortest path computation competitive with its exact counterpart. Instead we chose exact shortest path computation over approximate computation for each of the four commodity selection rules.

2.4.3 Comparison of Commodity Selection Rules

Having concluded that the exact shortest path version is faster for each commodity selection rule, we now compare those selection rules for the runs when exact shortest paths are used. As stated earlier, we use an α-scaling factor of 2 and a min-hop routing for the initial flow.

Comparison of Running Time

Figure 2.5 compares the running time for the various commodity selection rules, which were described in Section 2.3.9. For each matrix and each selection rule, we compare its running time against the minimum achieved by the four rules. We adopt this relative comparison approach because it does not pre-bias toward a particular selection rule, and we can regard the minimum time as that of an "ideal" commodity selection rule. Note that for each matrix, one of the selection rules will have a relative running time of 1.0.

The matrices in Figure 2.5 are listed in ascending order of their minimum running times. Some of those minimum running times are indicated in the figure, eg. BCSSTK05 has the fastest running time of 0.502 seconds, while BP1000 is the slowest, at 300 seconds. Note also that the largest relative times (NOS4 and NOS5 for the PotWtRand rule) are not plotted to scale, but are instead given next to the plot.

We crudely evaluate the commodity selection rules by examining those cases which are 20% or more above the minimum running time. The 20% threshold is chosen arbitrarily; lower threshold values will require a more refined comparison of the four selection rules, while higher thresholds would omit many more cases for comparison. Using this criteria, we observe that among the 20 test matrices:

- For the round-robin selection rule, there are 10 matrices whose running times were more than 20% above the minimum time; of those 10, 4 (LSHP265,

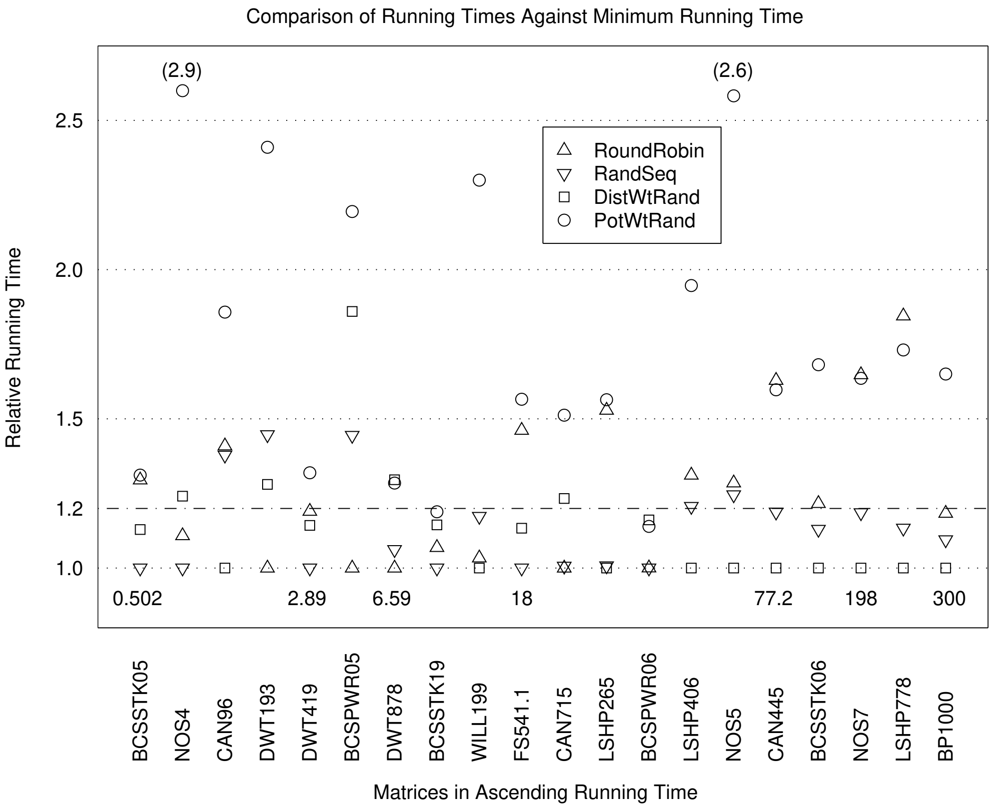


Figure 2.5: Ratio of time taken by each commodity selection rule to minimum time taken over all selection rules.

CAN445, NOS7 and LSHP778) had times that were more than 50% above the minimum. Also, there are 5 matrices (DWT193, BCSPWR05, DWT878, CAN715 and BCSPWR06) for which the round-robin rule had the minimum running time or was within 1% of it.

- For the random-source sequential-sink selection rule, there are 5 matrices whose running times were more than 20% above the minimum, of which none exceeded 50%. Also, there are 8 matrices (BCSSTK05, NOS4, DWT419, BCSSTK19, FS541.1, CAN715, LSHP265 and BCSPWR06) for which the RandSeq rule had the minimum time or was within 1% of it.

- For the distance-weighted random selection rule, there are 5 matrices whose running times were more than 20% above the minimum, of which 1 (BCSPWR05) exceeded 50%. Also, there are 10 matrices (CAN96, WILL199, LSHP265, LSHP406, NOS5, CAN445, BCSSTK06, NOS7, LSHP778 and BP1000) for which the DistWtRand rule had the minimum time, or was within 1% of it. Those 10 matrices include the matrices with the 7 highest running times.

- For the potential-weighted random selection rule, all but two (BCSSTK19 and BCSPWR06) of the matrices had running times more than 20% above the minimum, of which 15 exceeded 50%. In addition, the PotWtRand rule was not within 10% of the minimum running time for any of the matrices.

The above evaluation shows that the distance-weighted rule is justifiably the best among the four rules, especially for the matrices with higher running times, while the potential-weighted rule is clearly the worst. In addition the random-source sequential-sink rule is significantly better than the round-robin rule, and in fact it is faster for 14 of the 20 matrices, and has almost the same running time for 2 other matrices (CAN715 and BCSPWR06). However for 12 of the matrices, the running times for the two rules are always within 20% of each other. The comparison between these two rules somewhat supports the theoretical result of Radzik [72] proving that in the approximation algorithm of Leighton *et al* [56] for the multicommodity flow problem with general edge capacities, the randomized commodity selection scheme proposed there can be replaced by round-robin selection without increasing the *asymptotic* running time.

We next compare the total running time for the commodity selection rules in Table 2.4. In the table, we give for each rule the total time taken over all 20 matrices and also over the matrices with the 10 highest running times. For both aggregate running times, we see that the DistWt rule is the best, followed by the RandSeq and RoundRobin rules, and that the PotWt rule is the worst. We conclude that the DistWt rule is superior to the other three rules, and choose it for our implementation.

Table 2.4: Sum of running times for each commodity selection rule.

Commodity	Sum of Running Times (s)	
Selection Rule	All Matrices	Slowest 10 Matrices
RoundRobin	1542.3	1476.9
RandSeq	1214.7	1154.7
DistWt	1093.6	1028.9
PotWt	1812.4	1718.3

Comparison of Rerouting Quality

It is interesting to attempt to explain why the DistWtRand rule performs better than the other rules and to further compare the various rules. We first consider the number of reroutes needed by each selection rule to achieve ϵ-optimality. This is shown in Figure 2.6, where again we compare the number of reroutes for each rule against the minimum number over all four rules, and where the matrices are listed in ascending order of running times. For one case (NOS4 with the potential-weighted rule), the relative number of reroutes is not plotted to scale, but is instead indicated next to the plot.

We again evaluate the number of reroutes taken by each selection rule by setting a 20% threshold above the minimum. From the figure, we see that:

- The round-robin selection rule has 12 cases where the number of reroutes was more than 20% above the minimum, of which 9 exceeded 50%. Also, there are 4 cases (DWT193, BCSPWR05, DWT878 and CAN715) where it took the minimum number of reroutes or was within 1% of it.

- The random-source sequential-sink selection rule also has 12 cases where the number of reroutes was more than 20% above the minimum, but only 1 case (CAN96) where it exceeded 50%. In addition, there are 5 cases (BCSSTK05, NOS4, DWT419, BCSSTK19 and BCSPWR06) where it took the minimum number of reroutes or was within 1% of it.

- The distance-weighted rule has 1 case (BCSPWR05) where the number of reroutes was more than 20% above the minimum. In fact, for that case, the excess number of reroutes is more than 50%. In addition, there are 11 cases (CAN96, WILL199, FS541.1, LSHP265, LSHP406, NOS5, CAN445, BC-SSTK06, NOS7, LSHP778 and BP1000) where it took the minimum number of reroutes or was within 1% of it. These 11 cases include the matrices with the 7 highest running times.

- The potential-weighted rule has 17 cases where the number of reroutes was more than 20% above the minimum, of which 14 exceeded 50%, including 4 particularly bad cases (NOS4, DWT193, WILL199 and NOS5). Also there

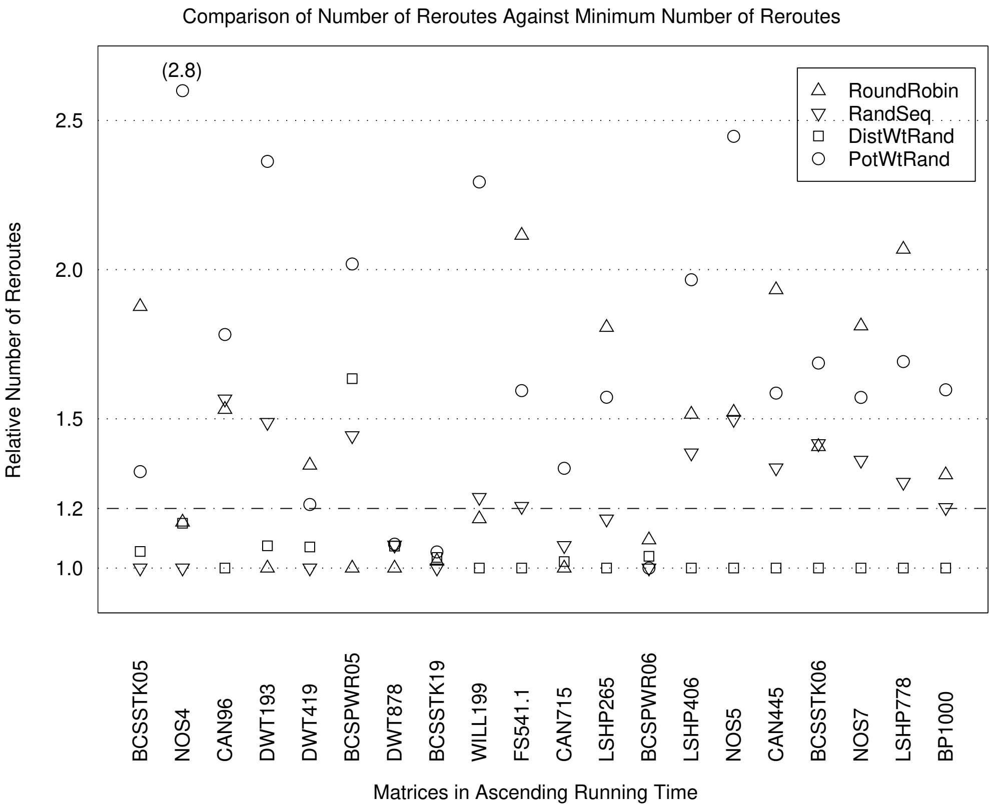


Figure 2.6: Ratio of number of reroutes taken by each commodity selection rule to minimum number of reroutes taken over all selection rules.

was only one instance (BCSPWR06) where it was within 1% of the minimum number of reroutes.

The comparison on the number of reroutes for the four rules is quite similar to that for their running times. As stated earlier, the number of reroutes reflects the quality of rerouting, and we see that the DistWtRand rule has the best rerouting quality and the PotWtRand the worst, with the round-robin and random-source sequential-sink rules having similar behavior. This shows that the use of the normalized difference between past longest and shortest path lengths as commodity weights in the DistWtRand rule quite accurately predicts the improvement that can be obtained from rerouting a commodity, and is superior to the uniform commodity selection that is essentially performed by the round-robin and RandSeq rules. Surprisingly, past changes in the potential function arising from rerouting a commodity, which is used to determine the weights under the PotWtRand rule, are not useful in estimating how much improvement would result from a future reroute of that commodity, as seen by the high number of reroutes taken by the PotWtRand rule. One possible reason is that the decrease in potential function from rerouting depends on the difference between the longest and shortest path lengths and also on the actual amount of flow rerouted, making past behavior less predictive of future behavior. We shall do a more careful correlation analysis for the two weighted rules later.

At each rerouting step, each commodity selection rule examines one or more commodities until one with an ϵ-bad longest source-sink path is found. Figure 2.7 shows the fraction of commodities examined that were actually rerouted, i.e. we plot the reciprocal of the average number of commodities examined at each rerouting step. We see a distinctive pattern in the fraction of commodities rerouted for each rule: the fraction for the PotWtRand rule is almost always higher than that for the DistWtRand rule (except for the matrices DWT878, BCSSTK19 and BCSPWR06), which in turn is almost always higher than those for the RandSeq and RoundRobin rules (except for BCSSTK05, DWT878 and FS541.1), which are typically close to each other. We see that the PotWtRand and DistWtRand rules are more effective than the other two rules in finding commodities with ϵ-bad paths, which further validates the choice of weights that are assigned to commodities under those rules.

The relatively higher fraction of rerouted commodities achieved by the DistWtRand rule, taken in conjuction with its typically lower number of reroutes as compared to the other rules, also shows that it typically examines fewer commodities overall than the other rules in each run. It is surprising though that the PotWtRand rule is more effective than the DistWtRand rule, suggesting that using the relative decrease in potential function as commodity weight leads to a more accurate prediction of which commodities are ϵ-bad, even though that choice of weight is a poorer predictor of how much improvement will result from a reroute.

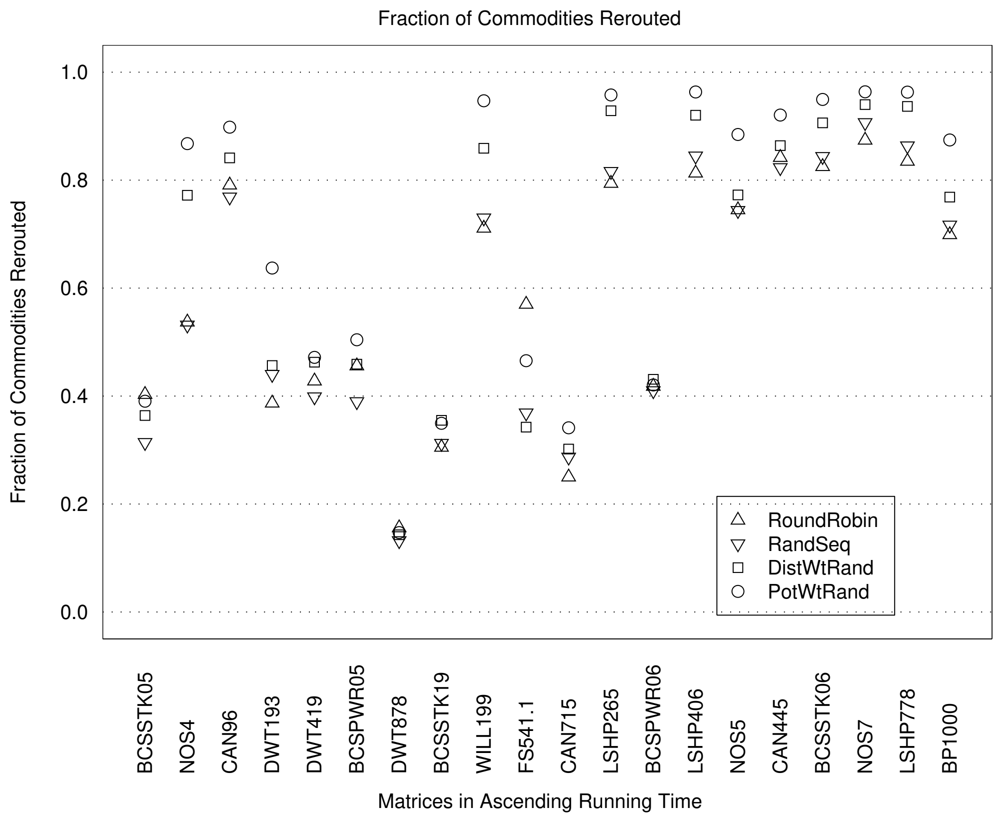


Figure 2.7: Fraction of commodities examined that were actually rerouted.

Statistical Correlations for the Weighted Selection Rules

For both the distance-weighted and potential-weighted selection rules, we instrument one run for each of the 20 matrices. Whenever a commodity is examined, we record the following information:

- its weight, which for the distance-weighted rule is the normalized difference between the longest and shortest path length the last time that commodity was examined, whereas for the potential-weighted rule it is a decaying average of relative decreases in the potential function from past reroutes of that commodity;
- the normlized difference between its current longest and shortest path lengths;
- whether the commodity was rerouted;
- and if the commodity was rerouted, the relative decrease in the potential function.

At the end of each run, we compute correlations between the various recorded quantities.

Correlation between old weights and decrease in the potential function. For the commodities which were rerouted, we computed the correlation between its old weight (when the commodity was selected) and the relative decrease in potential function from rerouting that commodity. It turns out that for the DistWtRand rule, the correlation was always positive, and ranged between 0.05 to 0.96, whereas for the PotWtRand rule, there were four negative correlations, ranging between -0.02 and -0.04, with the positive correlations varying between 0.002 to 0.14. We see that the correlation between the commodity weight and the drop in the potential function is much higher for the DistWtRand rule. The higher correlations show that the weights assigned by the DistWtRand rule more accurately predict the improvement that results from rerouting a commodity, and explains why the DistWtRand has much better rerouting quality than the PotWtRand rule, as we saw earlier.

Correlation between past and current (normalized) difference between longest and shortest path lengths. For the DistWtRand rule, we compute the correlation between the weight of a rerouted commodity and the difference between its longest and shortest path lengths before the reroute. Apart from a single insignificant correlation of 0.01, the correlations ranged from 0.27 to 0.96. Therefore the past difference between the longest and shortest path lengths of a commodity is a good indicator of future differences.

Correlation between old weights and actual reroutes. For the distance-weighted rule, the correlation between the weight of a commodity and the decision to reroute that commodity, i.e. whether that commodity was ϵ-bad, was negative for 18 of the 20 matrices. For the potential-weighted rule, the correlation was negative for all the matrices, and is usually smaller than the corresponding correlation for the distance-weighted rule. These negative correlations were probably due to the tendency of the weighted rules in selecting commodities that had been recently rerouted. Such commodities were usually not ϵ-bad.

Comparison of Average Rerouting Time

Our final comparison of the commodity selection rules is on the average time taken per reroute, i.e. the total rerouting time divided by the number of reroutes. This average time includes the time spent selecting and evaluating the average number of commodities per reroute, as well as the actual time spent on rerouting. Since the average number of commodities are not identical but quite close for all four selection rules, we have to exercise more care in comparing the average time per reroute.[15] In figure 2.8, we compare the average time per reroute for each selection rule against the minimum average time. Again we see a distinctive pattern. The minimum average time is always achieved by either the round-robin or random-source sequential sink rules, and in fact the average times for these two rules are always within 10% of each other, except for 4 cases (BCSSTK05, BCSPWR05, DWT878 and FS541.1) varying from 10.4% to 27.5%. As for the weighted random rules, their average times are always 10% or more than the minimum times and always higher than the round-robin or random-source sequential sink rules, even though these weighted rules generally examine fewer commodities per reroute than the other two rules. We conclude that the time taken by the weighted-random rules to select and evaluate a commodity is significantly higher than those of the unweighted rules, reflecting the additional time spent by the weighted rules to access and maintain a tree of commodity weights (cf. Section 2.3.9).

2.4.4 Comparison of α-Scaling Factors

We next evaluate several performance aspects of various α-scaling factors. The values we chose were $\sqrt{2}$ (≈ 1.414), $\sqrt[4]{2}$ (≈ 1.189). $\sqrt[8]{2}$ (≈ 1.091), $\sqrt[16]{2}$ (≈ 1.044) and $\sqrt[32]{2}$ (≈ 1.021).[16] Since we scale α multiplicatively, our choice of scaling factors essentially has a "doubling property," in that going from one factor to the next

[15] By profiling the runs, as we did for the shortest path evaluation, we would be able to decompose the average time per reroute into the commodity selection and evaluation component and the actual rerouting component. We did not do this because, as we shall see later, there is already a marked difference in the average time per reroute across the four selection rules, and so more precise timing is not needed.

[16] In the previous section, we had performed runs with the scaling factor 2, but those runs typically have the worst performance among the six scaling factors, and so we omit the results for that scaling factor in the subsequent discussion, except when presenting some miscellaneous statistics at the end of the section.

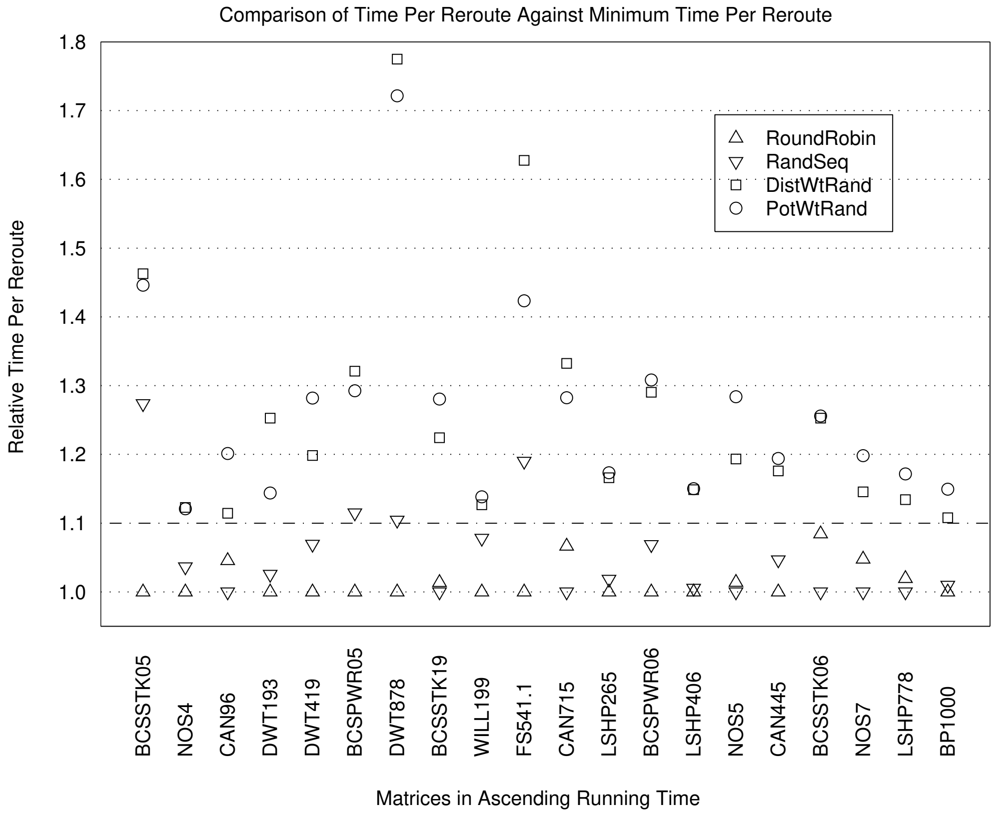


Figure 2.8: Ratio of average time per reroute for each commodity selection rule to minimum average time per reroute over all selection rules.

lower factor potentially doubles the number of scalings. We perform runs for the various scaling factors with the other implementation parameters set as follows: exact shortest paths, distance-weighted commodity selection rule, and min-hop initial routing. We shall see that the smaller scaling factors generally perform better than the larger ones, and that the overall difference between the three smallest factors is small. In addition, we also evaluate the scheme of Borger *et al* [11], where they raise α by incrementing it by 0.25 instead of scaling multiplicatively. Surprisingly, the additive scheme performs reasonably well, and is better than the multiplicative scheme in some cases, but overall it is inferior.

Unlike the previous section comparing the various commodity selection rules, where we saw that the performance of the selection rules varies widely, the differences in behavior for the scaling factors are usually small (typically under 20%). We choose a lower, and perhaps unreasonable, threshold of 10% above the minimum when evaluating various aspects of the performance of the scaling factors. However we shall be careful not to draw any specific conclusions favoring one scaling factor, but instead try to elicit any trends that may be present. The actual selection of the best scaling factor will be based on aggregating the running times over all the test cases instead of ranking the scaling factors based on performance in individual cases.

Role of α-Scaling Factor

Before presenting the performance data, we first review the constraints on α and the role played by the α-scaling factor. Recall from Section 2.2.2 that the purpose of the REDUCE routine is to half the current flow congestion or to detect ϵ-optimality. It does so by prescribing a value of α that satisfies a certain inequality following from the first relaxed optimality condition as long as the flow congestion has not been halved. However, as was discussed in Section 2.3.7, it is advantageous to use a much smaller value for α. Such a lower value however, might violate the desired inequality before the flow congestion has been halved, forcing REDUCE to exit prematurely. We then raise α by multiplying it by the scaling factor and re-enter REDUCE with the higher α value. Having a lower scaling factor makes α increase less quickly, resulting in a run where smaller values of α are used in computing the associated exponential vertex lengths.

Comparison of Running Time

As before, we first look at the running times for the various scaling factors, and those are given in Figure 2.9. The plot again gives the relative time, where we compare the running time of each scaling factor to the minimum running time over the five factors. In addition the matrices are listed in ascending running time, with some minimum running times indicated in the figure.

We evaluate the scaling factors by setting a threshold of 10%, from which we see that:

- For the scaling factor $\sqrt{2}$, there are 9 matrices whose running times were

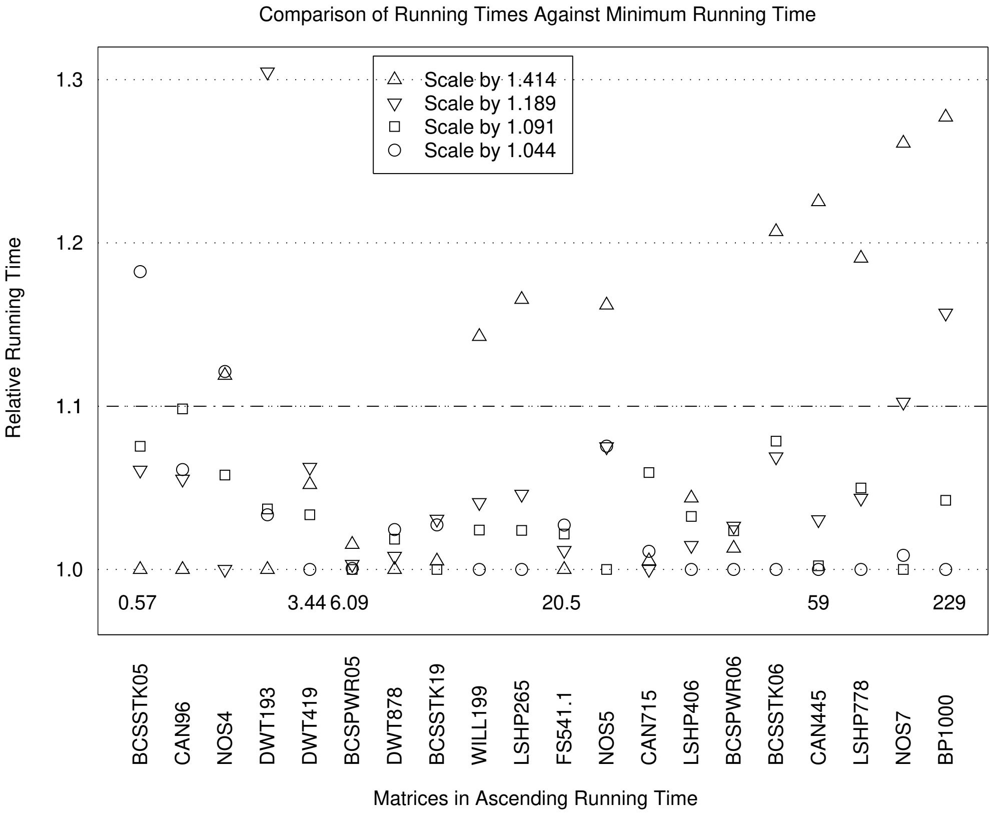


Figure 2.9: Ratio of time taken by each scaling factor to minimum time taken over all scaling factors.

10% or more above the minimum time. Also, there are 7 matrices (BCSSTK05, CAN96, DWT193, DWT878, BCSSTK19, FS541.1 and CAN715) which achieved the minimum time or were within 1% of it.

- For the scaling factor $\sqrt[4]{2}$, there are 3 matrices with running times 10% or more than the minimum. Also, there are 4 matrices (NOS4, BCSPWR05, DWT878 and CAN715) which achieved the minimum time or were within 1% of it.

- For the scaling factor $\sqrt[8]{2}$, only 1 matrix had running times 10% or more than the minimum. Also there are 5 matrices (BCSPWR05, BCSSTK19, NOS5, CAN445 and NOS7) which achieved the minimum time or were within 1% of it.

- For the scaling factor $\sqrt[16]{2}$, only two matrices had running times 10% or more than the minimum, but those matrices had low running times. Also there are 9 matrices (DWT419, BCSPWR05, LSHP265, LSHP406, BCSPWR06, CAN445, LSHP778, NOS7 and BP1000) which achieved the minimum running time or were within 1% of it.

- For the scaling factor $\sqrt[32]{2}$, there are 3 matrices with running times 10% or more than the minimum. Also there are 3 matrices (WILL199, BCSSTK06 and BP1000) which achieved the minimum running time or were within 1% of it.

The above evaluation shows that the running time generally improves as the scaling factor decreases. However the difference in the running time between the smaller scaling factors $\sqrt[8]{2}$, $\sqrt[16]{2}$ and $\sqrt[32]{2}$ is small.

Comparison of Rerouting Quality

We next look at the number of reroutes taken by each scaling factor to determine if there is a difference in their quality of rerouting. Figure 2.10 compares the number of reroutes for each scaling factor to the minimum number of reroutes over the four factors.

We see that the plot for the number of reroutes looks somewhat similar to that for the running times, especially with respect to the cases with higher ratios. More precisely, by setting a threshold of 10% as before, we observe that:

- For the scaling factor $\sqrt{2}$, there are 8 matrices whose number of reroutes is 10% or more above the minimum. Also there are 6 matrices (BCSSTK05, NOS4, DWT193, DWT878, BCSSTK19 and BCSPWR06) which achieved the minimum number of reroutes or were within 1% of it.

- For the scaling factor $\sqrt[4]{2}$, there are 6 matrices with 10% or more reroutes than the minimum, and 5 matrices (NOS4, DWT878, FS541.1, CAN715 and LSHP406) which achieved the minimum number of reroutes or were within 1% of it.

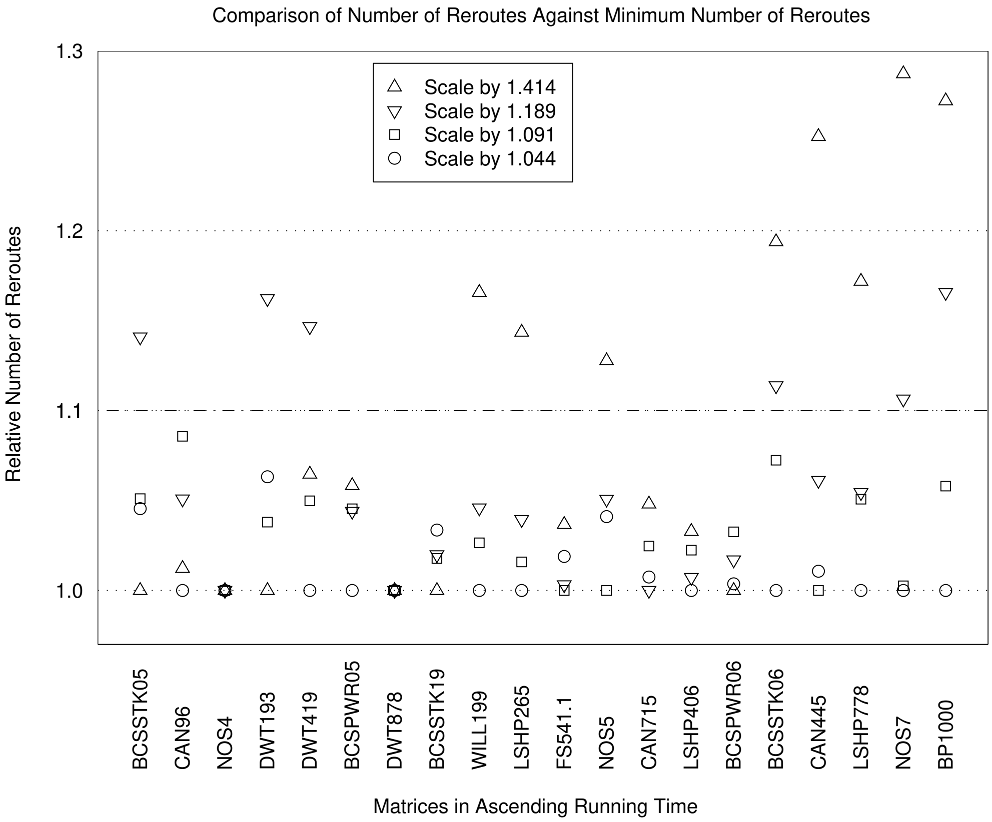


Figure 2.10: Ratio of number of reroutes taken by each scaling factor to minimum number of reroutes taken over all scaling factors.

- For the scaling factor $\sqrt[8]{2}$, only 1 matrix had 10% or more reroutes than the minimum. In addition, 6 matrices (NOS4, DWT878, FS541.1, NOS5, CAN445 and NOS7). achieved the minimum number of reroutes or were within 1% of it.

- For the scaling factor $\sqrt[16]{2}$, no matrix had 10% or more reroutes than the minimum. In addition, 12 matrices (NOS4, DWT193, BCSPWR05, DWT878, LSHP265, CAN715, LSHP406, BCSPWR06, BCSSTK06, LSHP778, NOS7 and BP1000) achieved the minimum number of reroutes or were within 1% of it.

- For the scaling factor $\sqrt[32]{2}$, there are 2 matrices with 10% or more reroutes than the minimum, and 7 matrices (CAN96, NOS4, DWT878, WILL199, LSHP265, BCSSTK06 and BP1000) which achieved the minimum number of reroutes or were within 1% of it.

We see that the quality of rerouting also improves as the scaling factor decreases. In fact the lower number of reroutes for the smaller scaling factors is the main reason why those factors have lower running times. This is supported by Figure 2.11, which compares the relative time per reroute for each scaling factor. From the figure, we see that the relative times per reroute are typically within 5% of the minimum, except for 12 cases (out of 100) that are between 5% to 10% above minimum, and 5 cases above 10%. Given that the relative times per reroute are similar for the four scaling factors, it follows that the running time is proportional to the number of reroutes taken.

The time per reroute depends on how many commodities are examined per reroute. In Figure 2.12, we plot the fraction of selected commodities which were actually rerouted. We see that the fraction of commodites rerouted, or inversely, the average number of commodities examined per reroute, is practically identical for most of the matrices and scaling factors. This indicates that the size of the scaling factor has little effect on the effectiveness of the distance-weighted commmodity selection rule, and therefore the relative time per reroute is not dependent on the scaling factor.

Why Lower Scaling Factors Have Better Rerouting Quality

We can in fact intuitively justify why the smaller scaling factors would have better rerouting quality. As remarked earlier, having a smaller scaling factor results in smaller values of α being used during a run. With a smaller α, the exponential vertex lengths grow less quickly with increasing vertex flow. During a rerouting step, flow is rerouted from a longest path to a shortest path, with the objective of maximizing the decrease in the potential function (which is the sum of vertex lengths). For a smaller α, the potential function does not decrease as quickly when flow is rerouted, thus encouraging more flow to be rerouted. We examine the relation between the amount of flow rerouted per step and the size of the scaling factor in Figure 2.13. In the figure, we compare the average amount of flow

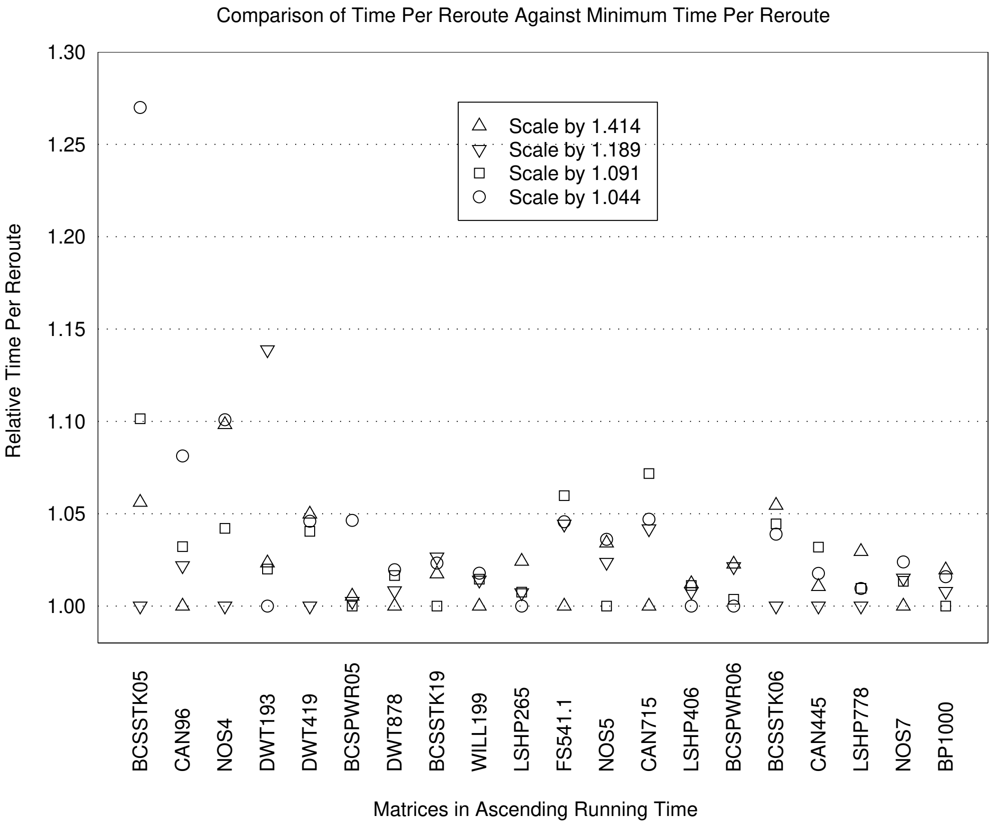


Figure 2.11: Ratio of average time per reroute for each scaling factor rule to minimum average time per reroute over all scaling factors.

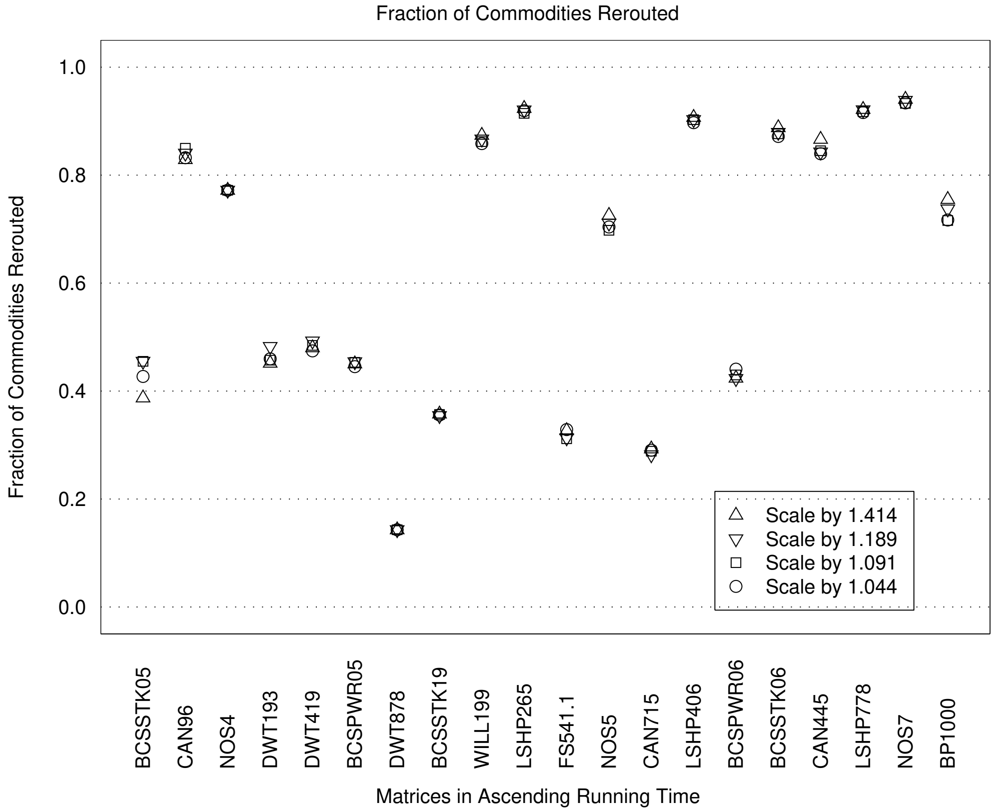


Figure 2.12: Fraction of commodities examined that were actually rerouted.

rerouted per step by each scaling factor to the minimum average amount. For all the matrices except BCSSTK05, DWT193, FS541.1, CAN715 and LSHP406, we see that the average flow rerouted per step is higher for the smaller scaling factors (those $\leq \sqrt[8]{2}$), agreeing with our intuitive analysis.

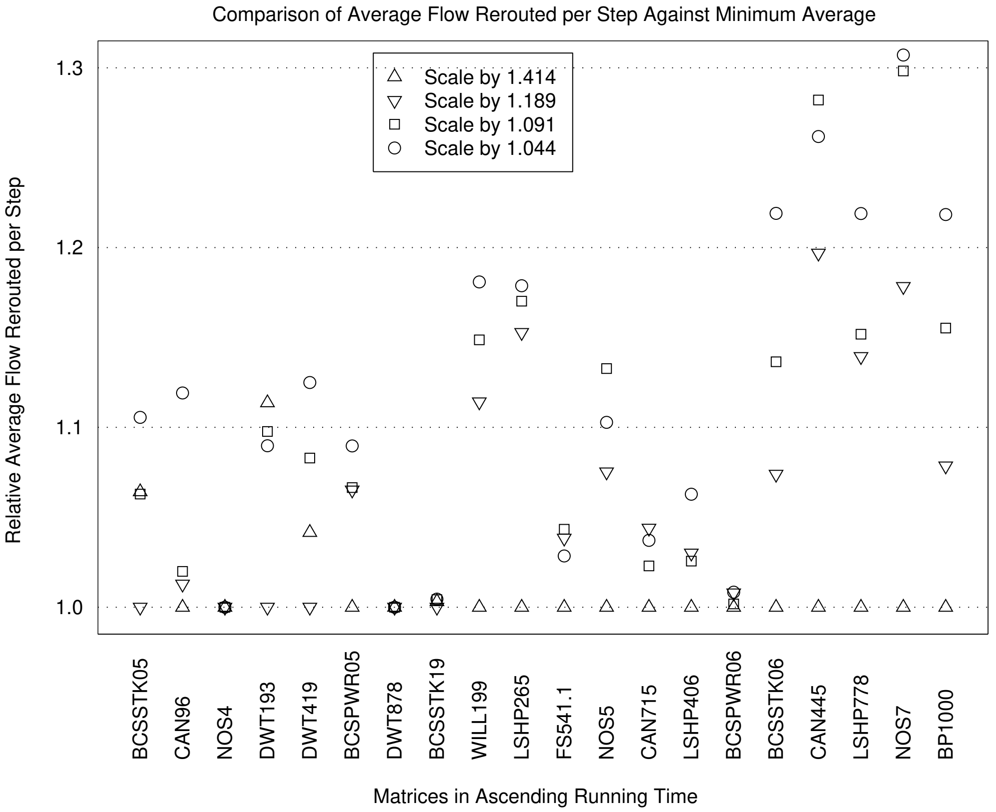


Figure 2.13: Ratio of average flow rerouted per step for each scaling factor to minimum average flow rerouted per step over all scaling factors.

Rerouting more flow at each step for a lower scaling factor, however, does not necessarily mean that fewer total reroutes are needed. If the total amount of flow rerouted increases disproportionately, then more reroutes would actually be necessary. We therefore focus on the total amount of flow rerouted by each scaling factor, which is the sum over all rerouting steps of the flow rerouted at each step. In Figure 2.14, we show the ratio of the total flow rerouted to the total demand for all commodities. We see that the total amount of flow rerouted ranges from 5% (DWT878) of the total demand to as much as 117% (BP1000). Strikingly, the figure shows that for each matrix, the different scaling factors rerouted almost the same total amount of flow. In fact the four scaling factors are always within 11% of each other, and in 91 of the 100 cases, within 5% of each other.

The empiricial evidence that smaller scaling factors reroute more flow at each step without rerouting significantly more flow overall explains why they have better rerouting quality.

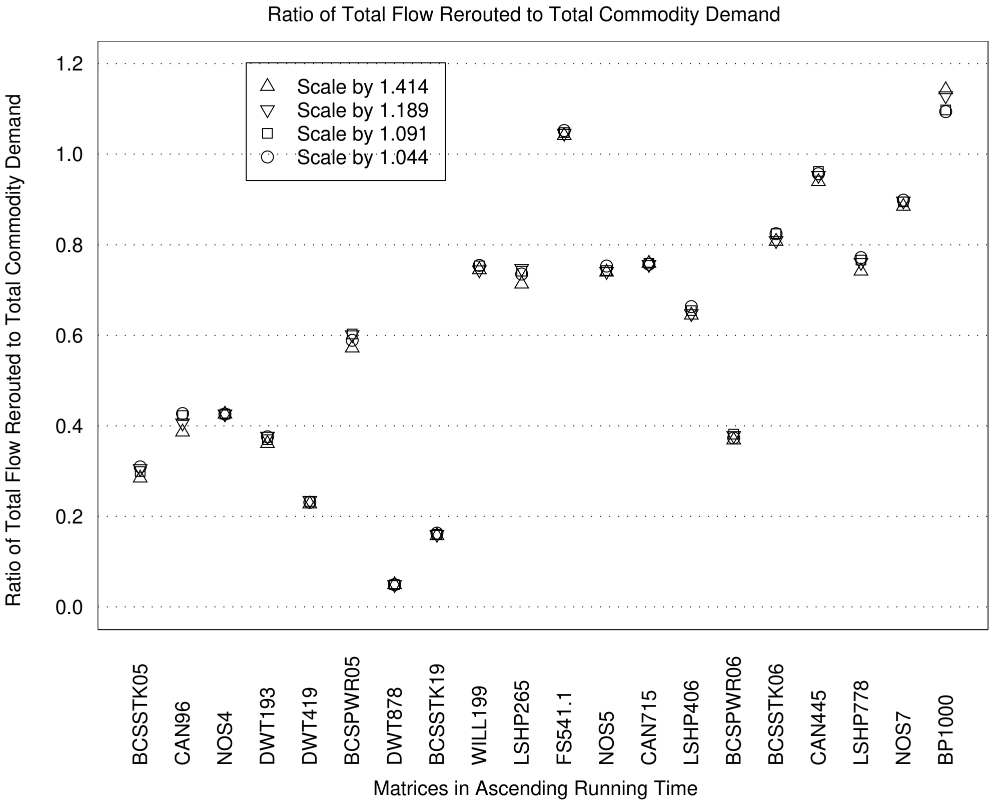


Figure 2.14: Ratio of total flow rerouted to total commodity demand for each scaling factor.

Table 2.5: Sum of running times for each scaling factor.

Scaling Factor	Sum of Running Times (s)	
	All Matrices	Slowest 10 Matrices
2	1093.6	1028.0
$\sqrt{2}$	973.1	911.5
$\sqrt[4]{2}$	879.3	819.9
$\sqrt[8]{2}$	835.7	777.5
$\sqrt[16]{2}$	815.7	757.8
$\sqrt[32]{2}$	834.2	775.6

We make a selection between the scaling factors by looking at the total running time over the 20 matrices and also over the slowest 10 matrices. The totals are presented for all the scaling factors in Table 2.5, where we have also included the times for the scaling factor $\sqrt[32]{2}$. We see that the factor $\sqrt[16]{2}$ has the lowest total running time for both measures, and that the overall total is lower than that for the factors $\sqrt[8]{2}$ and $\sqrt[32]{2}$ by about 2.5%. We therefore choose $\sqrt[16]{2}$ as the scaling factor.

Other Statistics For α-Scaling

Finally, we briefly present and comment on some statistics related to the number of scalings that occur and on the actual values taken on by α.

Number of α-scalings. The number of α-scalings for the factor $\sqrt{2}$ is between 1.7 to 2.2 times that for the factor 2. The corresponding ranges of ratios for the factors $\sqrt[4]{2}$, $\sqrt[8]{2}$, $\sqrt[16]{2}$ and $\sqrt[32]{2}$ with respect to the factor 2 are 2.8 to 4.2, 5.5 to 8.3, 10.7 to 16.4, and 21.6 to 31.9 respectively. These ranges agree well with the ideal ratios of 2, 4, 8 and 16 for the five factors.

Incidence of "unusable" values of α Recall that α has to be raised whenever it fails to satisfy some technical condition related to the current flow. It is possible that the new value of α still fails the technical condition, and is hence "unusable" in that no reroutes are performed for that value of α. The incidence of such "unsuable" values should be higher for smaller scaling factors, since the adjustment to α made by multiplying by the scaling factor is more likely to be too small. Indeed for our runs, the number of matrices (out of 20) for which "unusable" values do not occur is 16, 13, 10, 5, 4 and 3 respectively for the scaling factors 2, $\sqrt{2}$, $\sqrt[4]{2}$, $\sqrt[8]{2}$, $\sqrt[16]{2}$ and $\sqrt[32]{2}$. More strikingly, among matrices with "unusable" values of α, the percentages of α values that were "unusable" varies between 5.0% to 16.7%, 2.9% to 13.3%, 0.5% to 14.7%, 1.0% to 43.8%, 0.6% to 64.3% and 0.3% to 80.0% respectively for the scaling factors in decreasing order. A high incidence of "unusable" α value has some performance impact due to the wasted time spent in the initialization phase of REDUCE (including the computation of vertex lengths), and explains why the running time (cf. Table 2.5 of the algorithm does indefinitely decrease as the scaling factor shrinks.

Range of α. For the preferred scaling factor $\sqrt[16]{2}$, the initial values of α over the 20 matrices ranges from 0.0062 to 3.6, whereas the final values of α ranges from 0.065 to 8.3. In addition, the final value of α is between 1 to 900 times its initial value. We see that the range of α values and its growth varies widely with the matrix.

Comparison to prescribed value. For the preferred scaling factor $\sqrt[16]{2}$, the final value of α over the 20 matrices is 10.6 to 40.8 times smaller than that prescribed by the REDUCE routine. This confirms that the prescribed values are too large in practice, and that the idea of using a much smaller value of α is useful.

Comparison of final α-values. We compare the final values of α attained by each of the scaling factors $\sqrt{2}$, $\sqrt[4]{2}$, $\sqrt[8]{2}$, $\sqrt[16]{2}$ and $\sqrt[32]{2}$ to those obtained by the scaling factor 2. The ratios are 0.71 to 1.15, 0.59 to 1.07, 0.55 to 1.05, 0.55 to 1.04, and 0.55 to 1.00. Note that it is possible for the final value attained by the five lower scaling factors to exceed that attained by the highest scaling factor 2, since α has to satisfy an inequality that depends on the flow and the various scaling factors produced different final flows. The ranges of the ratios are close to ideal, with $\sqrt{2}/2 \approx 0.71$, $\sqrt[4]{2}/2 \approx 0.59$, $\sqrt[8]{2}/2 \approx 0.55$, $\sqrt[16]{2}/2 \approx 0.52$ and $\sqrt[32]{2}/2 \approx 0.51$.

Comparison with Additive α Scheme

Borger *et al* [11] used a different scheme for raising α in their implementation of the concurrent flow algorithm. Whenever α is detected to be too low, they increment it by an additive constant of 0.25. On the surface, such an additive scheme looks crude considering the wide range of α values encountered. It turns out, surprisingly, that the scheme performs reasonably well, as shown in Figure 2.15, where we plot the relative running time of the additive scheme and the multiplicative scheme with some of the scaling factors. From the figure, we see that the additive scheme has only 6 cases where its running time is greater than 10% above the minimum (better than the scaling factor $\sqrt{2}$ but worse than the other three factors).

As for the sum of running times, the additive scheme took 881.5 seconds for all the matrices, and 822.5 seconds for the slowest 10 matrices. Both of these times are about 8% higher than the corresponding times for the preferred scaling factor $\sqrt[16]{2}$ (cf Table 2.5, and so we empirically reject the additive scheme in favor of the scaling factor $\sqrt[16]{2}$.

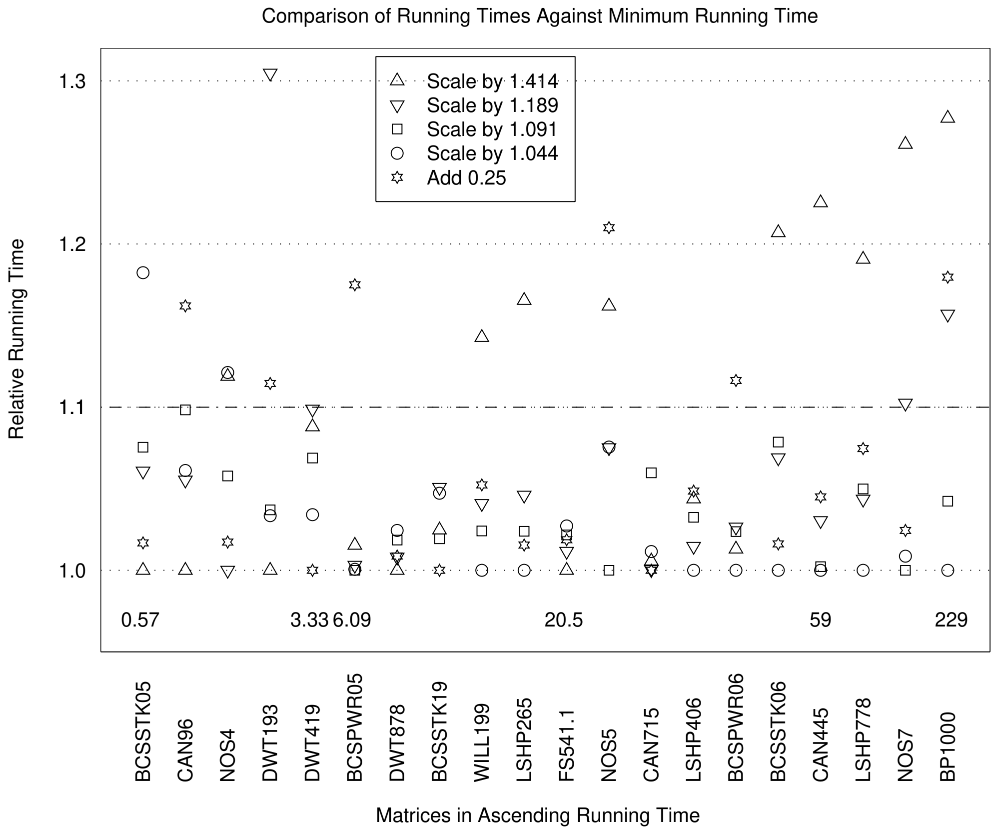


Figure 2.15: Ratio of time taken by additive scheme and each scaling factor to minimum time taken over all schemes.

2.4.5 Comparison of Initial Routings

The final implementation parameter that we shall consider is the routing of the initial flow before the first call to REDUCE. In all the previous tuning runs, the initial routing consisted of a min-hop path between the source and sink of each commodity. The use of a min-hop initial flow is standard, and it has been used in Shahrokhi and Matula [80] and also by Klein, Plotkin, Stein and Tardos [52]. We shall compare the min-hop initial routing to more sophisticated initial routings which use either 1, 2, 4 or 8 shortest paths to route the demand of each commodity. These routings were described in Section 2.3.8, and they use an exponential length function that is updated each time one more set of shortest paths have been routed for a source vertex. Following the testing methodology described earlier, we shall run each initial routing with the other implementation parameters set to: exact shortest path computations, distance-weighted commodity selection, and an α scaling factor of $\sqrt[16]{2}$.

Discussion on Shortest-Path Initial Routing

Before presenting empirical data for the various initial routings, we first review the shortest-path initial routing scheme and examine its relation to the rerouting scheme used by the concurrent flow algorithm. We shall see that the initial routing can be broadly interpreted as a form of round-robin rerouting, and therefore would produce an initial flow that is better than that from min-hop routing.

The shortest-path initial routing creates a feasible concurrent flow in phases, where in each phase, a fraction of the demand of each commodity is routed along some shortest path.[17] Specifically, for the p-shortest-path initial routing, we define a vertex length function

$$l_0(v) = e^{\log(\frac{\infty}{n})\frac{f(v)}{D}},$$

where ∞ is the largest representable real number, n is the number of vertices, D is the total commodity demand, and $f(v)$ is the current flow through vertex v.[18] Here $\log(\infty/n)/D$ plays the role of α, the parameter used in the length function during rerouting, and its value usually falls within the lower end of the range of values taken on by α during the concurrent flow algorithm, but can sometimes be higher.

Initially, there is no flow through any vertex; as flow paths are added to fractionally route the commodity demands, the flows through the vertices, and hence their lengths, increase. The actual routing of initial flow proceeds as follows:

<u>SHORTESTPATHROUTING</u>

for each vertex v: $f(v) \leftarrow 0$, $l_0(v) \leftarrow 1$

[17] For our implementation, we route along either one, two, four or eight shortest paths, so that the flow quantum is a negative power of 2 and flow values can be exactly represented.

[18] Note that the maximum length of a vertex is $e^{\log(\infty/n)} = \infty/n$, and so the length of a simple path cannot exceed ∞. Therefore the computation of path lengths cannot cause numerical overflow.

for $phase = 1, 2, \ldots, p$
    for each source vertex s
        for each commodity i with source s
            route $d(i)/p$ units of flow along shortest path from s to sink of i
        endfor
        update length function
    endfor
endfor

Note that the commodities are repeatedly routed in sequence, with commodities having the same source grouped together and sharing the same vertex lengths. This arrangement fully exploits the incremental shortest path computation scheme described in Section 2.3.5, and differs from the rerouting phase, in which the length function is updated after a commodity is rerouted.

The time taken by the shortest path initial routing is proportional to the number of shortest paths used, and can be quite high. When p shortest paths are used, the initial routing time is roughly equal to the time taken for pk reroutings. We shall give later actual figures for the fraction of total execution time spent in the initial routing.

The shortest-path-based initial routing, with its exponential-length function and its sequential routing of commodities, is reminiscent of the round-robin rerouting scheme used in the concurrent flow approximation algorithm. We can in fact draw a correspondence between a phase of the initial routing and one round of round-robin rerouting. The correspondence rests on the simple idea that scaling a concurrent flow by a constant β (i.e. multiplying all the edge and vertex flows by β) results in a feasible flow where the demand of each commodity has been multiplied by β. Therefore we can regard the flow after the completion of each phase of routing as a scaled version of a feasible flow. So in the case where two shortest paths are used, we can double the flow values for the first-phase flow to obtain a feasible flow for the original demands. The second phase can then be interpreted as rerouting half the demand of each commodity from the only active flow path to a shortest path. When more shortest paths are used however, the correspondence of initial routing to rerouting is less similar. Let $p > 2$ be the number of shortest paths used, and suppose $j > 1$ phases of initial routing have been performed. At that point, the flow is feasible for j/p of the demands of each commodity, and can be scaled by $(j+1)/j$ to obtain a feasible flow for $(j+1)/p$ of the demands. The routing of flow for commodity i in the $(j+1)^{\text{th}}$ phase can then be regarded as rerouting flow from *each* of the active flow paths for i onto a shortest path.

There are several other differences between shortest-path initial routing and round-robin commodity rerouting:

1. Only commodities with ϵ-bad longest paths are rerouted in each pass of round-robin, whereas flow is routed along a shortest path for each commodity during a phase of initial routing.

2. Rerouting involves transferring flow from a longest path to a shortest path,

whereas as just discussed, each phase of initial routing can be interpreted as transferring flow from all active flow paths to a shortest path.

3. The amount of flow rerouted is dynamic and is determined by the potential function and the shortest and longest paths of the selected commodity, while the amount of flow routed in each phase of initial routing is predetermined.

4. A phase of initial routing is not as "balanced" as one pass of round-robin in that the total flow in the network progressively increases within a phase. Therefore the routing of the later commodities within a phase is affected by the additional flow introduced in the routing of the earlier commodities.

5. As remarkd earlier, the parameters used in the length functions for initial routing and rerouting are different. In the initial routing length function, the multiplier applied to the vertex flow value is chosen to guard against numerical overflow, while the multiplier used for rerouting has to satisfy some technical relaxed optimality condition. In addition, the length function is updated after every reroute whereas in the initial routing, it is only updated after a set of commodities with the same source has been routed. So the initial length function is not as responsive to changes in vertex flows, but that is not of great concern as the network is less congested in the beginning and the goal of initial routing is to produce a resonably good initial flow.

6. Since the initial routing fully exploits the incremental shortest path computation setup but rerouting does not due to vertex lengths changing after each reroute, one phase of initial routing runs faster than one round of round-robin rerouting. In fact, measurements show that one phase of initial routing takes between 50% to 80% of the time of one round of round-robin rerouting.

We shall see later that initial routing using p shortest paths gives comparable behavior to initial routing using fewer shortest paths followed by an appropriate number of round-robin passes of commodity rerouting. So using shortest-path initial routing is similar to having a mixed commodity selection scheme where round-robin is used for the initial few passes, followed by the chosen commodity selection rule.

Comparison of Running Time

We now look at the running times for the various initial routings, and they are shown in Figure 2.16. As before, we plot the relative time, where we compare the running time of each initial routing to the minimum running time over all the initial routings. In addition the matrices are listed in ascending running time, with some running times indicated in the figure.

When a shortest-path-based initial routing is used, it turns out that some of the time, the initial flow is already ϵ-optimal, and so the initial flow congestion is below the target flow congestion that we use in terminating the algorithm. Such optimal initial flows are indicated in the figure by marking the appropriate symbol above

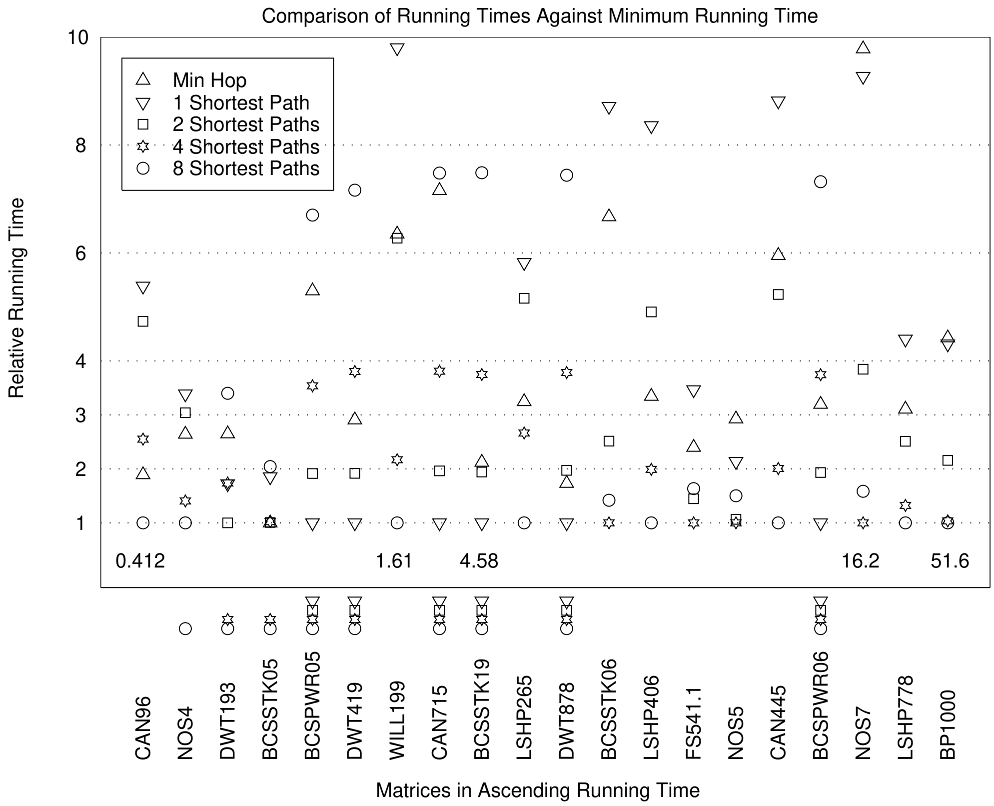


Figure 2.16: Ratio of time taken by each initial routing to minimum time taken over all initial routings.

the name of the matrix. Thus for the matrices BCSPWR05, DWT419, CAN715, BCSSTK19, DWT878 and BCSPWR06, the initial flows when 1, 2, 4 or 8 shortest paths are used are all ϵ-optimal. For the matrices BCSSTK05 and DWT193, the initial flows using 4 or 8 shortest paths are ϵ-optimal, while for NOS4, the initial flow using 8 shortest paths is ϵ-optimal. Observe that there is a natural nesting property among the shortest-path initial routings in that if using a certain number of shortest paths gives an ϵ-optimal initial flow, then using more shortest paths will preserve the ϵ-optimality.[19] We shall compare the initial flow congestions for the various initial routings later.

For a shortest-path-based initial routing, we can check if an appropriately scaled version of the flow after each phase has a flow congestion that is below the supplied target congestion, and hence terminate early with a partial but ϵ-optimal initial flow. The early termination scheme would benefit initial routings using several shortest paths on some problem instances, eg. the 8-shortest-path initial routing on those matrices where all shortest-path-based initial flows are ϵ-optimal. However, this scheme is an artifact of the idealized termination scheme used for our tuning runs, in which a known ϵ-optimal target flow congestion is supplied. In practice, checking if a scaled version of the flow after an initial routing phase is ϵ-optimal would involve evaluating the objective value of some dual solution, a step that is as computationally expensive as one phase. Moreover, there is no guarantee that a dual solution derived from any of the scaled partial initial flows would itself be ϵ-optimal, which is a necessary condition to establish the optimality of the flows. Indeed, experimentation on those matrices with ϵ-optimal initial flows shows that about three-quarters of the associated dual solutions are not ϵ-optimal, and so the ϵ-optimality of the partial initial flows will never be detected most of the time. We therefore do not check for early termination in the initial routings.

Since there is a wide variation in running times across the five initial routings, we only crudely compare the running times by counting the number of instances with relative running time greater than 4 or less than 2. We see that:

- For the min-hop initial routing, there are 7 matrices with relative running time greater than 4, and 3 matrices with relative running time less than 2.
- For the single-shortest-path initial routing, there are 9 matrices with relative running time greater than 4, and 8 matrices with relative running time less than 2.
- For the 2-shortest-paths initial routing, there are 5 matrices with relative running time greater than 4, and 10 matrices with relative running time less than 2.

[19] The nesting property is satisfied by our test cases, but does not hold in general. For example, we shall see later that for the matrix BP1000, the initial flow congestion when 8 shortest paths is used is slightly higher than that for 4 shortest paths, and so for some appropriately chosen error tolerance, the 4-shortest-path initial flow will be ϵ-optimal, but not the 8-shortest-path initial flow.

- For the 4-shortest-paths initial routing, there are no matrices with relative running time greater than 4, and 10 matrices with relative running time less than 2.

- For the 8-shortest-paths initial routing, only the 6 matrices with ϵ-optimal initial flows had relative running time greater than 4. In adddition, there are 12 matrices with relative running time less than 2.

Clearly, the two best initial routings are those using 4 or 8 shortest paths. While the 4-shortest paths initial routing does not have extremely high running times as compared to the 8-shortest routing, it has fewer instances of relatively low running times.

As expected, the proportion of overall time spent within the initial routing increases with the complexity of the routing. For min-hop initial routing, between 1.1% to 54.2% of the running time is spent in the initial routing. The wide variation is due to the fact that the quality of the initial flows varies greatly, as we shall see later, and so some initial flows require many more reroutes to achieve ϵ-optimality. For the shortest-path initial flows, the percentage of time spent in initial routing is as follows, where we have omitted those cases where the initial flow is already ϵ-optimal:

- For one-shortest-path initial routing, the percentage varies between 1.2% and 23.3%.

- For 2-shortest-path initial routing, the percentage varies between 3.6% to 84.3%.

- For 4-shortest-path initial routing, the percentage varies between 12.6% and 82.1%.

- For 8-shortest-path initial routing, the percentage varies between 52.6% to 97.4%.

We see that the initial routing using more shortest paths typically consumes a large fraction of the running time, and should not be regarded as a simple initialization process. As for the relation between the initial routing time and the number of shortest paths used, the behavior is also within expectation: routing using 2, 4 and 8 shortest paths roughly takes twice, four times and eight times as long as using one shortest path.

We next examine more closely the total running time relation between the various shortest-path-based initial routings. There are two distinctive patterns in the running times. For the six matrices (BCSPWR05, DWT419, CAN715, BCSSTK19, DWT878 and BCSPWR06) where all four shortest path versions were already ϵ-optimal, we see that the routing using 1 shortest path is the fastest, while using 2 shortest paths typically takes twice as long, using 4 shortest paths takes 3.5 to 3.8 times as long, and finally, using 8 shortest paths takes 6.7 to 7.5 times as long. This behavior is to be expected, since all the computation time in each

Table 2.6: Sum of running times for each initial routing.

Initial Routing	Sum of Running Times (s)	
	All Matrices	Slowest 10 Matrices
Min-Hop	815.7	741.3
1-Shortest Path	896.3	836.3
2-Shortest Paths	498.2	438.6
4-Shortest Paths	304.7	246.0
8-Shortest Paths	383.1	295.6

of those runs is spent only in the initial routings and is therefore proportional to the number of shortest paths used. For those instances, the shortest-path-based initial routings are very effective, and using more such paths is unnecessary from the perspective of reaching within 10% of optimality.

However for 8 of the other 14 matrices (CAN96, NOS4, WILL199, LSHP265, LSHP406, CAN445, LSHP778 and BP1000) we see a reverse pattern in the running times, with the 8-shortest-path version being the fastest, followed by the 4 and 2 shortest path versions, and the single shortest path version being the slowest. For those instances, routing on more shortest paths leads to initial flows which are faster to reroute to ϵ-optimality, and so the 8-shortest path version has the least overall running time despite spending more time on its initial routing.

As for the remaining 6 matrices (DWT193, BCSSTK05, BCSSTK06, FS541.1, NOS5 and NOS7), we see that the 1-shortest-path version is always slower than the 2-shortest path version, while the 8-shortest path version is always slower than the 4-shortest path version, and there is no pattern in the times for the 2-shortest and 4-shortest path versions.

We see that the effectiveness of the shortest path initial routing with respect to achieving ϵ-optimality varies with the matrix. Overall, however, it seems that the 4-shortest-path version is better since it has no extreme running times. This is more clearly seen in Table 2.6, where we give the total running time for each initial routing over all the 20 matrices and also over the 10 slowest matrices. We see that for both totals, the 4-shortest-path version is the best among the initial routing methods.

Comparison of Flow Congestion

We next compare the initial flow congestion for the various initial routings, and this is shown in Figure 2.17. In the figure we plot the ratio of the initial flow congestion to the target flow congestion used in terminating the computation. Note that several of the ratios for the min-hop initial routing fall outside the scale of the graph, and are instead plotted in the top part of the graph with the actual ratios indicated in parentheses next to the plot.

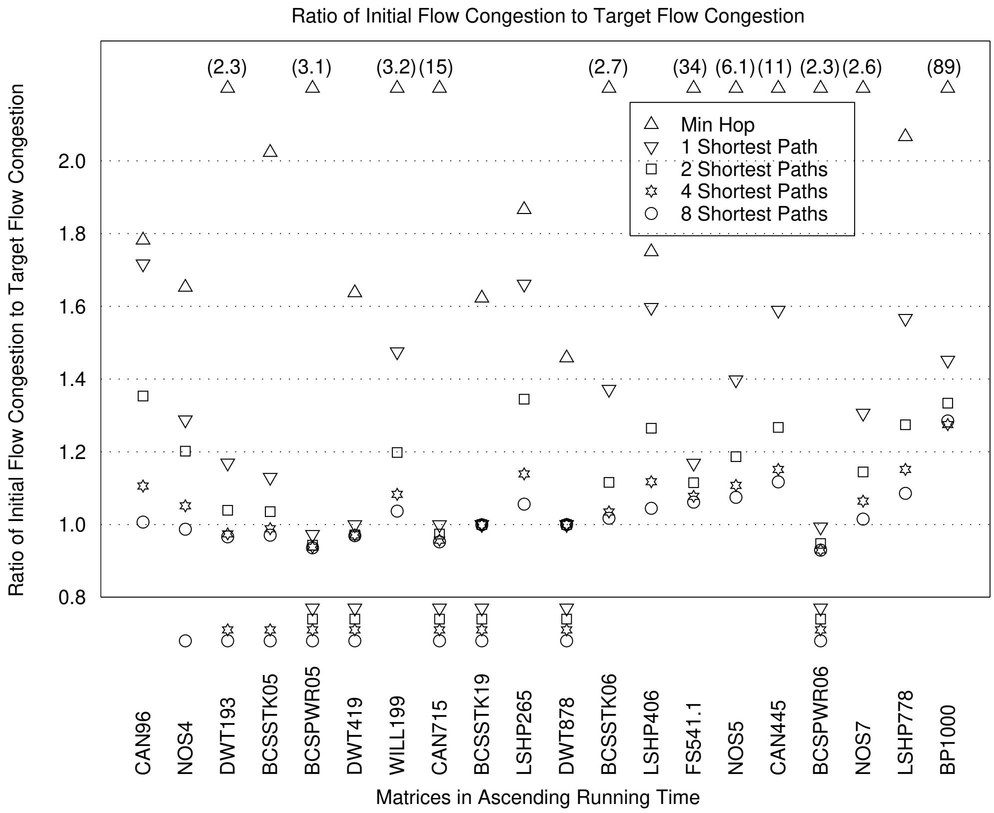


Figure 2.17: Ratio of initial flow congestion to target flow congestion for each initial routing.

We observe that the min-hop initial routing can be surprisingly bad, with one case where the initial congestion is 89 times the target congestion. In contrast, the shortest-path-based initial routings have initial congestions that never exceed 1.8 times the target congestion (nor exceed twice the optimal congestion). These empirical results are better than the theoretical $O(\log n)$ factor-from-optimal bound that Aspnes *et al* [6] gave for a similar routing scheme for virtual circuits. Additionally, the initial congestion of the five initial routings follow a fixed ordering, with the min-hop initial routing having the highest initial congestion, followed in sequence by the shortest-path-based initial routings with 1, 2, 4 and 8 paths. The only exception is for the matrix BP1000, where the 8-shortest-paths initial routing has a negligibly higher congestion than that using 4 shortest paths. The ordering of the congestion of the initial routings agrees with the intuition that the shortest-path based routings are better at "distributing" the flow over the graph, with more paths leading to more even distribution.

Note that in some cases, the initial congestion from the shortest-path initial routings is below the target congestion. This is due to the fact that the target congestion was obtained from the final congestion of past runs with an optimality tolerance of 0.1 ($\epsilon = 0.1$), and so the true optimal congestion can be as low as $\frac{10}{11}$ of the used target congestion.

As discussed at the beginning of this section, using additional shortest paths in the initial routing is similar to passes of the round-robin commodity selection scheme. We can in fact measure the effectiveness of using p shortest paths in the initial routing by comparing the initial flow congestion against that obtained when one (or 2) shortest path(s) is used in conjunction with $p-1$ (or $p-2$) passes of round-robin commodity rerouting. The comparisons we made were for 1-SP + 1-RR vs 2-SP, 1-SP + 3-RR vs 4-SP, 1-SP + 7-RR vs 8-SP, 2-SP + 2-RR vs 4-SP, and 2-SP + 6-RR vs 8-SP, where the abbreviations p-SP and p-SP + r-RR refer respectively to the initial routing with p shortest-paths and the initial routing with p shortest-paths followed by r passes of round-robin. It turns out that overall, the mixed versions had lower congestion than the corresponding plain initial routings, and that there were two extreme cases where the the congestion was typically less than 90% of that of the plain initial routings.

The comparison is, however, flawed as it does not take into account the running time of one pass of round-robin and one-phase of initial routing. As remarked earlier, the initial routing exploits the incremental shortest path computation scheme by using the same length function in computing the shortest paths for a set of commodities with the same source. The round-robin scheme does not completely enjoy this advantage even though it processes commodities with the same source in order, because the length function changes after each reroute. In fact measurements show that one phase of initial routing takes between 50% to 80% of the time for one pass of round-robin. We thus make another comparison where we reduce the number of round-robin passes in the mixed version so that the running time is comparable to that of the plain initial routing. With this adjustment, the mixed versions are in fact slightly worse than their corresponding plain initial routings, with the exception of the two extreme cases that previously occurred. We conclude therefore that there is no qualitative difference between additional shortest paths in the initial routing and an equivalent number of round-robin passes.

Comparison of Rerouting Quality

We further compare the various initial routings by looking at the number of reroutes and the total amount of flow rerouted. These are shown in Figures 2.18 and 2.19, where we express the number of reroutes as a fraction of the number of commodities, and the total flow rerouted as a fraction of the total commodity demand. We see that the number of reroutes always decreases as more shortest paths are used in the initial routing, as does the total flow rerouted. This again agrees with the intuition that using more shortest paths in the initial routing produces an initial flow that is closer to optimality, and therefore requires fewer reroutes to become ϵ-optimal.

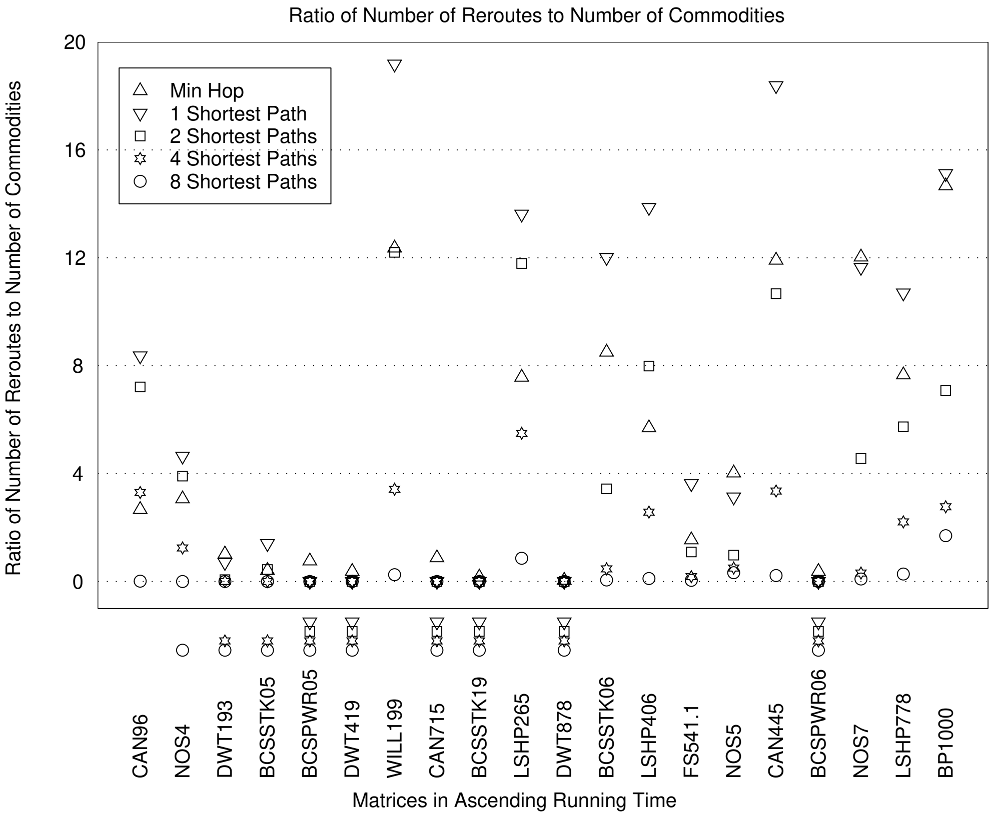


Figure 2.18: Ratio of number of reroutes to number of commodities for each initial routing.

The final statistic we look at is the fraction of commodites that were actually rerouted, and this is shown in Figure 2.20. Note that for those cases where the initial flow was already ϵ-optimal, we plotted the fraction of commodities rerouted as zero. We observe that there is significant variation in the fraction of commodites

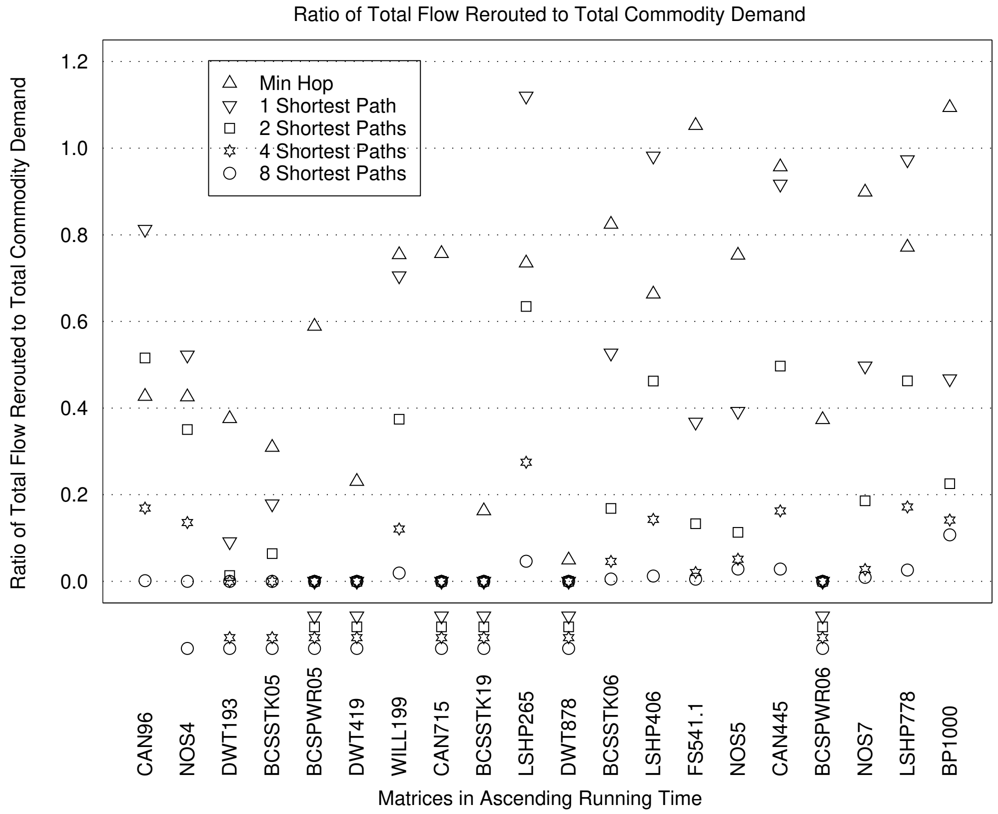


Figure 2.19: Ratio of total flow rerouted to total commodity demand for each initial routing.

across the five initial routings, unlike what we saw previously when evaluating the various α-scaling factors. It is almost always the case that the fraction decreases with the number of shortest paths used in the initial routing. This is not really that surprising since as a flow becomes closer to optimality, it is more difficult to reroute, and more commodities have to be examined at each rerouting step. Since using more shortest paths in the initial routing produces an initial flow that is closer to optimality, a lower fraction of commodities would end up being rerouted for those cases.

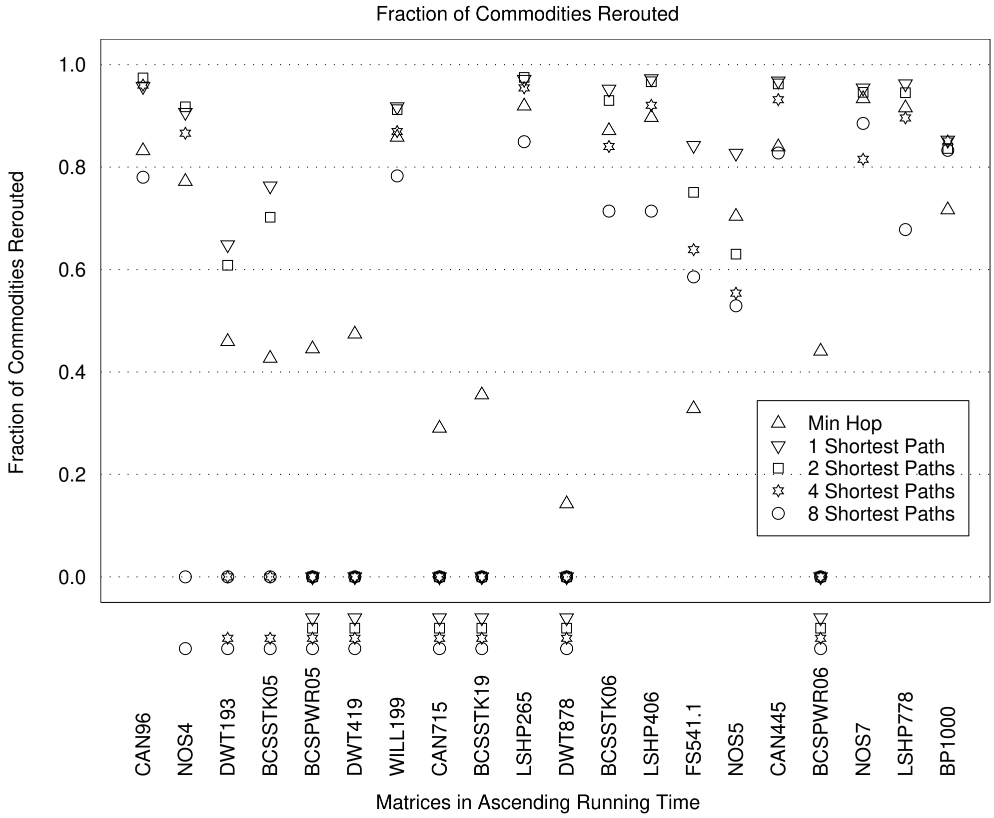


Figure 2.20: Fraction of commodities that were actually rerouted for each initial routing.

2.4.6 Optimality Checking for the Best Setting

The evaluation of the various implementation settings in the previous sections were based on runs employing an idealized optimality checking scheme, in which a run terminates when its flow congestion drops below a pre-determined user-supplied value. Such an idealized termination scheme was necessary due to the infeasibility of checking for optimality in a practical fashion when comparing implementation settings, as was discussed in detail in Section 2.4.1. We now revisit the issue of devising a practical way of checking optimality for the best setting (exact shortest path computation, distance-weighted commodity selection, α-scaling factor of $\sqrt[16]{2}$ and routing the initial flow using four shortest paths for each commodity). In particular, we shall consider various criteria for trigerring optimality checks, and compare the performance of the concurrent flow algorithm with practical optimality checks in place to that with idealized termination.

We first examine the framework for checking ϵ-optimality. Recall that ϵ-optimality is established in practice when the congestion of the current flow is at most $(1+\epsilon)$ times the objective value of some dual solution, which may not necessarily be the dual (length) solution associated with the current flow. However, we cannot evaluate the dual objective value too often as the evaluation is computationally expensive compared to a single rerouting step. We also saw in Section 2.4.1 that the objective value of the dual solution associated with the current flow is not monotone, i.e. the dual objective value at some earlier time can be higher than the current value. It is therefore prudent to save the best evaluated dual solution and compare its objective value against the flow congestion after each reroute.

To complete the framework, we have to specify when the dual objective value should be evaluated. There are two natural situations in which we would want to perform the evaluation. The first is periodically, after a certain number of reroutes have taken place since the last evaluation, for we expect overall progress towards achieving ϵ-optimality. The second situation is when some number of ϵ-good commodities have been examined in succession while looking for an ϵ-bad commodity to reroute; the difficulty in finding an ϵ-bad commodity is an indication that the concurrent flow is probably ϵ-optimal. We shall refer to the second type of check as the *ϵ-good commodity check*. In addition, when ϵ-optimality has been determined, we also evaluate the current dual solution (if it has not been evaluated by the other two criteria) since it can potentially be better than the best dual solution so far, and can give a lower achieved error bound.

For the best implementation setting, we saw in Section 2.4.5 that the initial flows for 8 of the 20 test matrices are already ϵ-optimal (determined by idealized optimality checking). The high incidence of ϵ-optimal initial flows suggests that it might be worthwhile to evaluate the objective value of the initial dual solutions. It turns out however, that 6 of the 8 initial dual objective values are not within a $(1+\epsilon)$ factor of the optimal dual objective value,[20] and so they cannot be used to establish the ϵ-optimality of *any* concurrent flow. For the two ϵ-optimal initial dual solutions (BCSSTK19 and DWT878), leaving out the evaluation of their objective values does not have significant performance impact, since in fact an evaluation of the dual value is triggered by the ϵ-good commodity check after fewer than 5 reroutes have taken place.

In our first assessment of the optimality checking framework, we perform 5 randomized runs (as before) for each of the 20 matrices, and set the periodic

[20] Note that this conclusion is made without knowledge of the optimal dual objective value z^*. Instead we found that the initial objective value z_0 is less than $z/(1+\epsilon)$ of some later objective value z, and so it must also be less than $z^*/(1+\epsilon)$.

check to after every $k/(10\epsilon)$ ($= k$ for $\epsilon = 0.1$) reroutes and the ϵ-good commodity check to $k/10$ ϵ-good commodities. Note that the period of the periodic check has a reciprocal dependence on ϵ, in accordance with Theorem 2.6 in Section 2.2.2, which gives the theoretical number of reroutes needed to halve the flow congestion. A summary of the results follows:

1. For 6 of the 8 matrices which have ϵ-optimal initial flows, ϵ-optimality was established by the first ϵ-good commodity check in each of the 5 runs, and no periodic checks were performed. This observation confirms our intuition that a concurrent flow is very likely to be ϵ-optimal when it is difficult to find an ϵ-bad commodity to reroute.

2. For the other 2 matrices (BCSSTK05 and DWT193) with ϵ-optimal initial flows, there were 1 or 2 periodic checks performed in each run. In 9 of the 10 runs, no ϵ-good commodity checks were performed, and optimality was established by the periodic checks. There was a single run which had one periodic check and also one ϵ-good commodity check, with optimality established when the latter check was performed. Overall, the maximum number of ϵ-good commodities examined during each reroute was less than $k/26$, with the exception of that single run that terminated with an ϵ-good commodity check.

3. For the 12 matrices that do not have ϵ-optimal initial flows, there were typically 4 to 5 periodic checks performed during each run, and up to 9 periodic checks, but no ϵ-good commodity checks. Overall, the maximum number of ϵ-good commodities examined during each reroute was less than $k/100$.

We see that for a matrix, both the relative maximum number of ϵ-good commodities examined in a reroute (i.e. as a fraction of k) and the type of optimality checks performed on it is related to the quality of its initial flow. The matrices without ϵ-optimal initial flows had less than $k/100$ ϵ-good commodities examined per reroute, while for the matrices with ϵ-optimal initial flows, that quantity was up to $k/26$ for 2 of the matrices, and exceeded $k/10$ for the other 6 matrices. We therefore refine the ϵ-good commodity check by arbitrarily lowering the trigger value to $k/60$ from $k/10$. As for the periodic checks, the following crude tradeoff analysis shows that we can decrease its frequency to every $2k$ reroutes without impacting the running time. The time taken to evaluate the dual objective value is roughly equal to the time for $k/2$ reroutes, taking into account the incremental shortest path computations that are performed and that on average each rerouting step computes slightly more than one shortest path. In a typical run with 4 to 5 periodic checks, eliminating half of those checks by raising the period from k to $2k$ would reduce the optimality checking time by the time taken by k reroutes, but might increase the number of reroutes by k since optimality is checked less frequently. For runs with higher number of periodic checks, the lower frequency potentially cuts down on the overall running time.

There are other alternatives for refining the optimality checking criteria. Instead of having a periodic check based on the number of reroutes, it is possible

Table 2.7: Comparison of performance of concurrent flow algorithm with practical optimality checks and with idealized termination.

Matrix	Time (s)		# Reroutes		Final	Final
	Actual	Ideal	Actual	Ideal	Error	Congestion
CAN715	20.2	13.4	507.2 (0.2k)	0.0	2.33%	−1.77%
BCSSTK05	1.4	0.7	305.2 (0.7k)	0.0	2.16%	−2.26%
BCSSTK19	20.8	17.2	3.4 (0.0k)	0.0	0.94%	−0.02%
BCSPWR05	5.0	4.1	12.2 (0.0k)	0.0	1.43%	−0.14%
BCSPWR06	50.8	43.2	15.2 (0.0k)	0.0	0.61%	−0.20%
DWT193	2.5	1.0	489.2 (0.8k)	0.0	3.34%	−2.57%
DWT419	6.1	4.5	94.6 (0.1k)	0.0	3.33%	−0.67%
DWT878	22.3	19.0	0.0 (0.0k)	0.0	0.04%	0%
CAN96	2.8	1.1	2554.2 (9.1k)	927.0	8.87%	−3.83%
CAN445	38.4	19.9	8673.2 (6.5k)	4475.8	9.82%	−1.18%
BCSSTK06	23.4	7.8	3938.0 (3.1k)	590.6	7.59%	−2.41%
NOS4	1.5	0.6	1177.0 (4.0k)	369.2	7.84%	−3.48%
NOS5	31.8	8.9	4486.4 (3.2k)	696.6	2.40%	−4.04%
NOS7	66.5	16.2	8822.8 (4.0k)	679.8	8.14%	−1.92%
LSHP265	18.3	12.7	6070.0 (7.6k)	4363.2	9.87%	−1.01%
LSHP406	35.3	15.7	7659.4 (6.3k)	3128.0	9.82%	−2.14%
LSHP778	86.5	52.9	9076.2 (3.9k)	5148.8	9.53%	−1.02%
WILL199	6.0	3.5	3354.4 (5.7k)	2026.0	9.26%	−0.80%
FS541.1	26.3	8.8	3260.6 (2.0k)	286.4	8.05%	−2.07%
BP1000	126.2	53.6	16896.2 (6.9k)	6811.0	9.69%	−1.72%

to have one based on the number of commodities examined so far. The difference is insignificant though, since it turns out that for the matrices whose ϵ-optimality was determined by the periodic check, the average number of commodities examined per reroute was at most 1.3, and so the number of commodities examined is always within a small constant factor of the number of reroutes. In addition, we can also have an adaptive periodic checking scheme where the period is adjusted based on how close the dual objective value is to the flow congestion when it was last evaluated. Such an adaptive scheme would probably be useful if there is a set of problem instances with a wide variation in the number of periodic checks being performed, which is not the case with our test matrices.

Table 2.7 summarizes the results of the runs with the final optimality checks in place, where the periodic checks occur after every $2k$ reroutes and the ϵ-good commodity check is triggered when $k/60$ ϵ-good commodities have been examined in succession.[21] The matrices in the table are separated into two groups, with the first eight matrices (CAN715 through DWT878) being those with ϵ-optimal initial

[21] To ensure that the trigger value does not drop to zero for small problem instances, we actually set it to $\max(2, k/60)$.

flows. The second and fourth columns give the average running time and the average number of reroutes (also expressed as a fraction of the number of commodities) taken over the 5 runs for each matrix, while the third and fifth columns give the running time and the number of reroutes when idealized optimality checking was used. The next-to-last column gives the average achieved error, which indicates how close the final primal and best dual solution are to each other, and which upper bounds the relative difference between the final primal/dual value and the optimal value. Finally the last column indicates how much lower the final congestion is with practical optimality checks in place as compared to the runs with idealized termination.

We see a marked difference in behavior of the concurrent flow algorithm with practical optimality checks for matrices with and without ϵ-optimal initial flows. For matrices with ϵ-optimal initial flows, a relatively small number of reroutes are needed before ϵ-optimality is detected by the ϵ-good commodity optimality check. In addition, the achieved error is much lower than the 10% error tolerance used. As for the other matrices, the number of reroutes is 1.4 to 13 times that taken when idealized termination was used, and optimality is established during the periodic check. Also, the achieved error is closer to the 10% error tolerance. The much higher number of reroutes reflects the observation made in Section 2.4.1 that the dual objective value is further from optimality as compared to the primal flow congestion, which was used in the optimality check for the idealized runs.

With the practical optimality checks in place, the running times are between 1.2 to 2.4 times those of the idealized runs for the matrices with ϵ-optimal initial flows, and between 1.4 to 4.1 for the other matrices. Note that the time taken to evaluate the dual objective value is roughly one-fifth of the time taken for the initial routing, and so an increase in 20% of the running time over the idealized version is inevitable. In fact for some of the matrices with ϵ-initial flows (BCSSTK19, BCSPWR05, BCSPWR06 and DWT878), the actual running time is close to the best possible, and in those instances, the final congestion is only slightly better than that obtained by the idealized runs. For the other matrices, the longer running time and higher number of reroutes produces a final flow whose congestion is between 0.7% to 4.0% lower than that from the idealized runs.

2.5 Experimental Results

We present in this section detailed results from running the concurrent flow approximation algorithm on several test matrices. The matrices are taken from the standard Harwell-Boeing collection of sparse matrices [18] [19], with the smallest matrix having 96 vertices and the largest, 1454. From each matrix, we create an instance of the concurrent flow problem using the graph of the matrix as the underlying network, and having for each vertex v, three unit-demand commodities with v as the source vertex and with randomly chosen sinks.[22] The concurrent

[22] Randomly generated commodities with the same source and sink are aggregated into a single commodity with higher demand.

flow instances are solved with the optimality error parameter ϵ set to 0.1.

In Section 2.5.1, we compare the performance of the concurrent flow approximation algorithm to standard linear programming methods. We shall see that solving the concurrent flow problem and its dual to optimality using linear programming takes too long, justifying the effort spent in implementing the concurrent flow approximation algorithm. We then examine in detail in Section 2.5.2 various statistics on the behavior of the approximation algorithm on the problem instances, and consider in Section 2.5.3 a minimum spanning tree heuristic of Shahrokhi for improving the dual solution.

2.5.1 Comparison of Approximation Algorithm to Linear Programming Methods

Instances of the concurrent flow problem can be solved to optimality using standard linear programming methods. We compare the performance of the approximation algorithm to that of CPLEX, a popular linear/integer program solver. We first give polynomial-sized formulations of both the concurrent flow problem and its dual, and then present the running times for solving the linear programs using the primal and dual simplex methods.

Polynomial-sized Linear Programming Formulations

The linear programming formulations described in Section 2.1 for the concurrent flow problem and its dual required the enumeration of all flow paths for the commodities, and hence have an exponential number of variables and/or inequalities. As indicated in that section, there exists equivalent formulations that are polynomial in the size of the problem instance. We give below one such formulation for the concurrent flow problem. The formulation has nonnegative variables $f_i(uv)$ for the flow of commodity i along the *directed* edge uv. Note that each undirected edge in the input graph $G = (V, E)$ is represented as a pair of directed edges, and so there is potentially flow of a commodity along both directions of an edge. The first set of inequalities concerns the total flow through a vertex and its relation to the flow congestion. The next three sets of inequalities express the flow conservation property at the nonterminals and at the source and sink of each commodity.

$$
\begin{array}{rcll}
 & \min & \lambda & \\
\sum_{i=1}^{k} \sum_{uv \in E} \frac{1}{2}(f_i(uv) + f_i(vu)) & \leq & \lambda & \forall \ u \in V \\
\sum_{uv \in E} f_i(uv) - f_i(vu) & = & 0 & \forall \ 1 \leq i \leq k, \ \ u \neq s_i, t_i \\
\sum_{s_i v \in E} f_i(s_i v) - f_i(v s_i) & = & d_i & \forall \ 1 \leq i \leq k \\
\sum_{t_i v \in E} f_i(t_i v) - f_i(v t_i) & = & -d_i & \forall \ 1 \leq i \leq k \\
\lambda & \geq & 0 & \\
f_i(uv) & \geq & 0 & \forall \ 1 \leq i \leq k, \ \ uv \in E
\end{array}
$$

It is possible instead to have only flow variables corresponding to an arbitrarily chosen direction of an undirected edge, but those variables will be unbounded in sign to accommodate a reversal in direction. The unbounded version is however slower to solve in practice.

The linear program can be reduced in size by aggregating the flow variables for commodities with the same source, akin to the use of directed flow graphs to represent the concurrent flow. The reduced linear program has $1 + 2mk^*$ variables and $n + nk^*$ inequalities, where k^* is the number of distinct commodity sources. For the actual problem instances that we are solving, where each vertex is the source for 3 commodities, the number of variables and the number of inequalities are $1 + 2mn$ and $n + n^2$ respectively.

We can also formulate the dual problem of minimizing the sum of shortest path distances between the terminals of the commodities using a polynomial number of variables and inequalities. The formulation is given below. Note that the given distance LP is *not* the dual of the flow LP above, but is equivalent to it. There is a length function l on the vertices, corresponding to the total vertex flow inequalities of the flow LP. In addition, there are distance variables $\hat{d}(s_i, u)$ which upper bound the distance (in terms of l) from the source s_i of commodity i to other vertices u in the graph. These distance variables correspond to the flow conservation equations. When the linear program is solved to optimality, the triangle inequalities ensure that $\hat{d}(s_i, u)$ indeed gives the distance from s_i to u.

$$
\begin{array}{rcll}
\max & & \sum_{i=1}^{k} d_i \hat{d}(s_i, t_i) & \\
\sum_{v \in V} l(v) & = & 1 & \\
\hat{d}(s_i, v) & \leq & \hat{d}(s_i, u) + \frac{1}{2}(l(u) + l(v)) & \forall\ 1 \leq i \leq k,\ \ uv \in E,\ \ v \neq s_i \\
l(v) & \geq & 0 & \forall\ v \in V \\
\hat{d}(s_i, s_i) & = & 0 & \forall\ 1 \leq i \leq k \\
\hat{d}(s_i, u) & \geq & 0 & \forall\ 1 \leq i \leq k,\ \ u \neq s_i
\end{array}
$$

Since distances between vertices are symmetric, we can potentially halve the number of distance estimate variables $\hat{d}(s_i, u)$ by having only one variable for each pair of vertices. The resulting linear program has at most $n + nk^*$ variables and $1 + 2mk^* - 2m$ inequalities. For the actual problem instances that we shall be solving, we can take full advantage of the symmetric distance estimate variables, and so the associated linear program has $n+n(n+1)/2$ variables and $1+2mn-2m$ inequalities.

Comparison of CPLEX and Approximation Algorithm

We solve some small instances of the concurrent flow problem using CPLEX, and also using the approximation algorithm with an error parameter of 0.01. Table 2.8 summarizes the characteristics of the problem instances. Columns 2 through 4 give the number of vertices, edges and commodities in the input network. The remaining columns present the size of the flow linear program and the dual distance linear program, indicating the number of rows, columns and nonzero entries in each linear program.[23]

We see that even for small problem instances, the associated linear programs can be quite large. It is not surprising then that solving the linear programs takes much longer than running the approximation algorithm, and this is shown in Table 2.9. The second and third columns give the time and number of reroutes taken by the concurrent flow algorithm to find a solution that is within 1% of optimal. The corresponding figures for CPLEX are given in columns 4 through 9, where we give the time and the number of iterations taken to solve the flow and distance linear program using the primal and/or dual simplex method. We see that solving the linear programs exactly can be much slower than obtaining an approximate solution for the small problem instances listed. For example, the primal simplex method on the distance LP for the problem NOS4 takes more than 500 times longer than the approximation algorithm. In addition, the primal

[23] The number of rows and columns are sometimes less than that given by the formulae earlier. This is due to elimination, substitution and aggregation operations performed by the LP presolver of CPLEX.

Table 2.8: Characteristics of concurrent flow problem instances.

Matrix	n	m	k	Flow LP # row	Flow LP # col	Flow LP # nz	Distance LP # row	Distance LP # col	Distance LP # nz
CAN24	24	68	69	600	3265	13080	3061	300	12196
BCSPWR02	49	59	146	1470	4803	17297	5606	1225	22410
BCSSTK01	48	176	136	2352	16897	67632	16369	1176	65344
CAN61	61	248	180	3782	30257	121085	29513	1891	117861
NOS4	100	247	297	9400	48701	193500	48660	5050	194489

Table 2.9: Comparison of concurrent flow algorithm to CPLEX.

Matrix	Approx Algo Sec	Approx Algo # Re-routes	Flow LP Primal Splx Sec	Flow LP Primal Splx Iters	Distance LP Primal Splx Sec	Distance LP Primal Splx Iters	Distance LP Dual Splx Sec	Distance LP Dual Splx Iters
CAN24	0.8	2061	2.4	1105	23.3	1482	5.0	477
BCSPWR02	0.2	30	6.2	905	124.3	4417	15.8	1440
BCSSTK01	25.1	68188	248.6	39461	788.7	8637	228.3	3637
CAN61	5.8	7977	109.3	9460	3075.3	15353	296.4	3062
NOS4	32.0	29740	2561.6	97705	18470.2	47324	2916.3	13607

simplex method on the distance LP is much slower than both the primal simplex method on the flow LP and the dual simplex method on the distance LP, although all three solution approaches produce optimal concurrent flows and optimal vertex lengths.

Recall from Section 1.6 that the concurrent flow problem occurs as a subproblem of the cut construction procedure of Leighton and Rao [57], but that what is actually used in that construction is the vertex length function. The approximation algorithm that we implemented delivers an ϵ-optimal vertex length function. As for the linear programming methods, both the primal simplex method on the flow LP and the dual simplex method on the distance LP have to be run to completion to obtain a *feasible* vertex length function, which would also then be optimal. The timings in Table 2.9 show that these two approaches are many times slower than the approximation algorithm. Alternatively, we can use the even slower approach of applying the primal simplex method on the distance LP, but not run it to completion. We then obtain a suboptimal vertex length function, but unlike the approximation algorithm, we have no error bound on how close it is to optimality. We conclude that is not practical to find a good vertex length function using linear programming methods, and that the effort spent in implementing the concurrent flow approximation algorithm was justified.

2.5.2 Experimental Results for the Concurrent Flow Approximation Algorithm

We ran the concurrent flow approximation algorithm on 33 matrices from the Harwell-Boeing sparse matrix collection [18] [19], with the commodity specification as described earlier, and with an error setting of 0.1. The implementation of the algorithm uses exact shortest path computation, a distance-weighted randomized commodity selection rule, an α-scaling factor of $\sqrt[16]{2}$ and an initial flow routing composed of four shortest paths for each commodity. In addition, ϵ-optimality is checked at each reroute by comparing the flow congestion to the best evaluated dual value. The dual objective value is evaluated after every $2k$ reroutes and whenever $k/60$ ϵ-good commodities have been examined in succession while searching for an ϵ-bad commodity to reroute. We shall refer to the two evaluation criteria as the *periodic check* and the *ϵ-good commodity check* respectively. These implementation settings were determined by the tuning process described in Sections 2.4.2 through 2.4.6.

For each input network, five randomized runs were performed on a high-end 4-processor Sparc workstation. The averages of the various statistics are reported in this section.

Before presenting statistics for the runs, we first look at the characteristics of the inputs to the algorithm, given in Table 2.10. For each matrix, we list the number of vertices and edges in its associated graph, as well as the number of commodities and the minimum, average and maximum degree of a vertex. In addition, the table contains information on the hop distance between two vertices, which is the minimum number of edges on a path between the vertices. The second-to-last column gives the average hop distance over all pairs of vertices, while the last column gives the diameter of the graph, which is the maximum number of edges in a shortest path between any two vertices. These two quantities measure the average and maximum "separation" of vertices in the graph.

We see that the matrices have widely varying characteristics. In particular, some matrices (eg. BCSSTK19 and NOS6) have very large average hop distances and diameters. The hop distances are in fact correlated with the average length of flow paths used by the concurrent flow algorithm, as we shall see later when examining the final concurrent flow.

Overall Performance of Algorithm

Table 2.11 summarizes the performance of the algorithm on the test matrices. The second through fourth columns give the number of vertices, edges and commodities in the input network, while the fifth column gives the total time taken in seconds, and the sixth column gives the time spent in constructing the initial routing, expressed in seconds and also as a fraction of the total running time. The second-to-last column shows the number of reroutes taken to achieve ϵ-optimality. Finally, the last column indicates the actual error achieved by the produced solution, i.e. it gives one less the ratio of the final flow congestion to the final dual value, which is an upper bound on how close the flow congestion and dual value are to the optimal value.

The matrices in the table are divided into two groups. The 12 matrices in the first group (CAN292 through ASH292) have initial flows that are already ϵ-optimal while the remaining 21 matrices do not have ϵ-optimal initial flows.[24] The behavior of the concurrent flow algorithm differs markedly between matrices with and without ϵ-optimal initial flows:

1. As expected, much fewer reroutes are needed for those instances with ϵ-optimal initial flows, and in those instances the achieved error is much less than the error tolerance of 10% that was used. An extreme example is DWT878, where the initial flow and the associated dual solution are practically optimal.

2. For runs where the initial flow was ϵ-optimal, the evaluation of the objective value of the dual was always triggered by the ϵ-good commodity check, and the ϵ-optimality of the solution was confirmed after one such evaluation. This suggests that initial flows that are already ϵ-optimal are difficult to reroute in that a relatively high number of commodities have to be examined when looking for a commodity to reroute.

3. For runs without ϵ-optimal initial flows, the evaluation of the dual objective value was always triggered by the periodic check, except for 2 ϵ-good com-

[24] The determination of the ϵ-optimality is done a posteori by comparing the initial flow congestion to the final flow congestion and dual values.

Table 2.10: Characteristics of input to the concurrent flow algorithm.

Matrix	n	m	k	Degree Min	Degree Avg	Degree Max	Hop Distance Avg	Hop Distance Diam
CAN96	96	336	281	5	7.0	8	4.4	8
CAN144	144	576	426	5	8.0	14	6.6	13
CAN187	187	652	557	4	7.0	9	8.1	16
CAN229	229	774	682	2	6.8	12	5.8	12
CAN292	292	1124	874	3	7.7	34	4.6	10
CAN445	445	1682	1334	3	7.6	12	5.9	12
CAN715	715	2975	2139	1	8.3	104	6.7	14
BCSSTK04	132	1758	393	14	26.6	45	2.2	5
BCSSTK05	153	1135	457	3	14.8	23	4.2	12
BCSSTK06	420	3720	1258	1	17.7	27	5.7	16
BCSSTK19	817	3018	2448	1	7.4	10	26.8	86
BCSSTM07	420	3416	1258	1	16.3	25	5.8	16
BCSPWR05	443	590	1326	1	2.7	9	9.3	20
BCSPWR06	1454	1923	4355	1	2.6	12	15.8	36
DWT193	193	1650	578	7	17.1	29	3.4	7
DWT209	209	767	625	3	7.3	16	5.0	12
DWT310	310	1069	925	3	6.9	10	13.0	39
DWT419	419	1572	1253	5	7.5	12	9.0	22
DWT592	592	2256	1775	3	7.6	14	12.8	37
DWT878	878	3285	2633	3	7.5	9	15.5	40
NOS4	100	247	297	1	4.9	6	5.0	13
NOS5	468	2352	1402	6	10.1	22	4.5	11
NOS6	675	1290	2023	2	3.8	4	18.9	58
NOS7	729	1944	2184	3	5.3	6	8.9	24
LSHP265	265	744	794	3	5.6	6	8.8	24
LSHP406	406	1155	1216	3	5.7	6	10.9	30
LSHP577	577	1656	1728	3	5.7	6	13.0	36
LSHP778	778	2247	2330	3	5.8	6	15.1	42
WILL199	199	660	592	2	6.6	13	3.1	5
ASH292	292	958	874	3	6.6	13	8.7	25
FS541.1	541	2466	1621	5	9.1	540	2.0	2
SHL200	663	1720	1986	2	5.2	439	2.7	6
BP1000	822	4635	2462	2	11.3	307	3.0	6

Table 2.11: Summary of concurrent flow runs.

Matrix	n	m	k	Time (s) Total	Time (s) Initial	# Reroutes	Achieved Error
CAN292	292	1124	874	2.5	2.0 (80%)	29.2	2.9%
CAN715	715	2975	2139	20.2	12.8 (64%)	507.2	2.3%
BCSSTK05	153	1135	457	1.4	0.7 (49%)	305.2	2.2%
BCSPWR05	443	590	1326	5.0	4.1 (81%)	12.2	1.4%
BCSPWR06	1454	1923	4355	50.8	42.8 (84%)	15.2	0.6%
BCSSTK19	817	3018	2448	20.8	17.5 (84%)	3.4	0.9%
DWT193	193	1650	578	2.4	1.0 (43%)	489.2	3.3%
DWT209	209	767	625	2.6	1.0 (40%)	562.2	1.8%
DWT310	310	1069	925	3.0	2.4 (81%)	31.2	1.3%
DWT419	419	1572	1253	6.1	4.5 (74%)	94.6	3.3%
DWT878	878	3285	2633	22.3	18.4 (83%)	0.0	0.0%
ASH292	292	958	874	2.6	2.0 (80%)	28.4	1.7%
CAN96	96	336	281	2.8	0.2 (8%)	2554.2	8.9%
CAN144	144	576	426	3.6	0.6 (16%)	1821.6	8.4%
CAN187	187	652	557	3.0	0.8 (28%)	1114.0	8.0%
CAN229	229	774	682	11.9	1.2 (10%)	5183.2	8.9%
CAN445	445	1682	1334	38.4	4.6 (12%)	8673.2	9.8%
BCSSTK04	132	1758	393	5.0	0.5 (11%)	2452.2	6.7%
BCSSTK06	420	3720	1258	23.4	5.4 (23%)	3938.0	7.6%
BCSSTM07	420	3416	1258	31.7	5.5 (17%)	5779.2	8.1%
DWT592	592	2256	1775	33.2	9.2 (28%)	3550.0	3.0%
NOS4	100	247	297	1.5	0.2 (16%)	1177.0	7.8%
NOS5	468	2352	1402	31.8	5.4 (17%)	4486.4	2.4%
NOS6	675	1290	2023	55.5	10.6 (19%)	6148.8	9.1%
NOS7	729	1944	2184	66.5	12.3 (18%)	8822.8	8.1%
LSHP265	265	744	794	18.3	1.6 (9%)	6070.0	9.9%
LSHP406	406	1155	1216	35.3	3.8 (11%)	7659.4	9.8%
LSHP577	577	1656	1728	56.3	7.5 (13%)	8471.2	9.4%
LSHP778	778	2247	2330	86.5	14.2 (16%)	9076.2	9.5%
WILL199	199	660	592	6.0	0.7 (12%)	3354.4	9.3%
FS541.1	541	2466	1621	26.3	7.2 (27%)	3260.6	8.0%
SHL200	663	1720	1986	54.5	9.2 (17%)	7727.6	8.0%
BP1000	822	4635	2462	126.2	13.2 (10%)	16896.2	9.7%

modity checks in the runs of the matrix BCSSTK04. The number of such evaluations was at most 5 for these runs. In addition, the number of ϵ-good commodities examined while looking for a commodity to reroute is at most $k/80$ (except for the anomalous runs of BCSSTK04), and is frequently much lower.

It is not surprising that a significant number (12 of the 33 matrices) of the initial flows are already ϵ-optimal, considering that four shortest paths were used to route each commodity in an initial flow. We had discussed in Section 2.4.5 the similarities between multiple shortest-path based initial routing and round-robin rerouting of commodities, where we saw that routing along one shortest path corresponds to rerouting one commodity. The initial routing therefore is comparable to 4 rounds of round-robin rerouting, and can take a substantial fraction of the total running time: 40% to 83% for initial flows that are ϵ-optimal, and 8% to 28% for those that are not.

It is interesting to see how well the actual number of reroutes agrees with that given by a theoretical analysis of the algorithm. From Theorem 2.6 of Section 2.2.2, $O(\epsilon^{-1}D/\sigma)$ reroutes are sufficient to halve the flow congestion, where D is the total commodity demand and σ is the common flow quantum that we use in our implementation. Note however that that bound is for the situation where only σ units of flow is rerouted at each step, unlike our implementation which reroutes an integer multiple of σ units of flow at each step. Nonetheless, we can regard that bound as a weak upper bound for our implementation.[25]

As we shall see later on, the ratio between the initial and final flow congestions for all runs is at most 1.3. Therefore, the expression $O(\epsilon^{-1}D/\sigma)$ upper bounds the total number of reroutes needed, and we expect the number of reroutes to be proportional to D/σ. In fact, the correlation coefficient between the number of reroutes and the quantity D/σ is 0.76, showing that the upper bound, although not derived for the multiple rerouting scheme, is still somewhat applicable.

Overall Rerouting Statistics

We look at some statistics on flow congestion and rerouting for an entire run, which is presented in Table 2.12. Column 5 gives the ratio of the total flow rerouted to the total commodity demand, while column 6 gives the ratio of the initial flow congestion to the final flow congestion. Again, we have divided the matrices into two groups based on the ϵ-optimality of their initial flows, as in Table 2.11.

We observe that the initial flow congestion is at most 1.3 times that of the final flow. Since we use an error tolerance of 0.1, it follows that the initial flow congestion is at most 43% above optimal, which indicates the effectiveness of the

[25]The $O(\epsilon^{-1}D/\sigma)$ bound on the number of reroutes is a worst-case bound for our multiple rerouting scheme, but we were unsuccessful in deriving better upper bounds due to the complication that the amount of flow we are trying to reroute at each step might exceed the flow on the path we are rerouting from. In that situation, it is difficult to estimate the improvement from the reroute because there is no simple relation between the amount of flow on a path and the total flow through the vertices on that path.

Table 2.12: Overall rerouting statistics.

Matrix	n	m	k	$\frac{\text{Flow Rerouted}}{\text{Total Demand}}$	$\frac{\text{Initial } \lvert f \rvert}{\text{Final } \lvert f \rvert}$
CAN292	292	1124	874	0.003	1.01
CAN715	715	2975	2139	0.014	1.02
BCSSTK05	153	1135	457	0.030	1.02
BCSPWR05	443	590	1326	0.001	1.00
BCSPWR06	1454	1923	4355	0.001	1.00
BCSSTK19	817	3018	2448	0.000	1.00
DWT193	193	1650	578	0.027	1.03
DWT209	209	767	625	0.028	1.01
DWT310	310	1069	925	0.001	1.00
DWT419	419	1572	1253	0.005	1.01
DWT878	878	3285	2633	0.000	1.00
ASH292	292	958	874	0.003	1.01
CAN96	96	336	281	0.323	1.15
CAN144	144	576	426	0.333	1.23
CAN187	187	652	557	0.073	1.02
CAN229	229	774	682	0.211	1.12
CAN445	445	1682	1334	0.244	1.17
BCSSTK04	132	1758	393	0.142	1.09
BCSSTK06	420	3720	1258	0.133	1.06
BCSSTM07	420	3416	1258	0.136	1.08
DWT592	592	2256	1775	0.090	1.06
NOS4	100	247	297	0.300	1.09
NOS5	468	2352	1402	0.117	1.15
NOS6	675	1290	2023	0.161	1.10
NOS7	729	1944	2184	0.106	1.09
LSHP265	265	744	794	0.338	1.15
LSHP406	406	1155	1216	0.249	1.14
LSHP577	577	1656	1728	0.280	1.16
LSHP778	778	2247	2330	0.241	1.16
WILL199	199	660	592	0.167	1.09
FS541.1	541	2466	1621	0.149	1.10
SHL200	663	1720	1986	0.191	1.20
BP1000	822	4635	2462	0.224	1.30

initial routing scheme. In addition, we see that even in runs where the initial flow is not ϵ-optimal, the amount of flow that had to be rerouted is at most slightly more than a third of the total commodity demand. This is much smaller than the upper bound of $\epsilon^{-1}D\log_2(|f_0|/|\hat{f}|) = 10D\log_2(|f_0|/|\hat{f}|)$ predicted by theory (cf. Theorem 2.6), where $|f_0|$ and $|\hat{f}|$ are the initial and final flow congestion and D is the total commodity demand. Nonetheless, the upper bound does describe the relationship between the ratio of the total flow rerouted to the total commodity demand and the ratio of the initial and final flow congestion. Indeed, the two quantities are correlated, with a coefficient of 0.86.

Final Flow Graph Statistics

We next examine the flow graphs for the final concurrent flow that is produced by the algorithm. A major issue that we shall explore in this and the next subsection is that of *fragmentation* of flow, i.e. the use of many paths to route the demand of a commodity. Recall that in our implementation of the concurrent flow algorithm, we use a common flow quantum σ for all commodities, and that each flow value is an integer multiple of σ (cf. Section 2.3.3). Fragmentation is a concern when σ is small relative to the unit commodity demands. Indeed the value of σ in our runs varied between 6×10^{-5} and 0.004, and was almost always less than 0.0005. It is possible in practice to increase σ, for the only constraint on σ is that its product with the length parameter α be less than some technical upper bound (cf. Section 2.2.2), and we have used a value of α that is typically 10 to 20 times smaller than that prescribed by Klein, Plotkin, Stein and Tardos [52]. There are several reasons however not to raise σ by the factor of 10 to 20 that is permitted. Using a higher σ complicates the implementation due to the fact that α is scaled upwards during a run, which therefore necessitates an occasional reduction in σ to avoid violating the technical upper bound. Furthermore, a larger σ results in cruder control on the optimal amount of flow to reroute at each step. More compellingly, we shall see later that the small σ values that we used do not cause fragmentation in practice.

To gather statistics on the flow paths used in the final concurrent flow, it is necessary to decompose the flow graphs for each set of commodities with the same source. As we discussed in Section 2.3.1, the decomposition of a flow graph into its constituent flow paths is not unique. We arbitrarily decompose a final flow graph by repeatedly removing from it a longest (in terms of number of vertices) path between the source and sink of the desired commodity. The statistics that we give are averages over all the extracted flow paths for the entire concurrent flow, and is unlikely to be biased by our method of decomposing flow graphs.

Number of flow paths per commodity. Using the method described above to decompose the final flow graphs into flow paths, we found that the average number of flow paths per commodity was between 2.5 and 12.2. For each run, there were commodities whose entire demand was routed along a single flow path. On the other extreme, for some of the runs, there were commodities which were routed on more than 50 flow paths. The low average number of flow paths per commodity corroborates the results of the experiments of Shahrokhi and Matula [80], where they observed that for all-pairs concurrent flow, the number of paths per commodity was at most 2.3. Given that the demand of each commodity was initially routed on four (not necessarily distinct) shortest paths, the low average number of flow paths per commodity also shows that the rerouting phase does not introduce many additional flow paths. We shall comment further on the last observation when we look at the flow statistics for each rerouting step.

Flow along flow paths. The average flow along a flow path varied between 0.08 and 0.41, or in relative terms, between $79\hat{\sigma}$ and $5700\hat{\sigma}$, where $\hat{\sigma}$ denotes the final flow quantum. The high average flow (recall that each commodity has a demand of 1) is due to the small number of flow paths needed to route each commodity. More significantly, there are few paths with small flow values. For each run, the 10th percentile for the flow values over all flow paths was at least $7\hat{\sigma}$, i.e. at most 10% of the flow path values were below $7\hat{\sigma}$. In fact for 27 of the 33 matrices, the 10th percentile was at least $24\hat{\sigma}$. As for the 20th percentile for flow path values, it was at least $14\hat{\sigma}$ for each of the runs, and exceeded $40\hat{\sigma}$ for 28 of the matrices. Finally, the median flow path value was never below $45\hat{\sigma}$, and was at least $99\hat{\sigma}$ for all but 2 of the matrices. The infrequent occurrence of flow paths with small flow values shows that the small flow quanta that we used in our runs did not adversely affect the distribution of flow among flow paths. More specifically, since the 10th flow value percentile typically exceeded $20\hat{\sigma}$, raising the flow quantum by the permitted factor of 10 to 20 would have had little effect on the frequency of flow paths with small flow values. Together with the low average number of flow paths per commodity, we conclude that our choice of flow quantum does not lead to fragmentation of the concurrent flow in practice.

Length of flow paths. The average number of edges on a flow path varied between 4.1 and 39 across the runs, while the maximum number ranged from 8 to 147. In relative terms, the average number of edges (or vertices) on a flow path does not exceed 9% of the numebr of vertices in the graph, and is typically less than 6%, while the maximum number is at most 25%. The low average flow path length is encouraging, since the concurrent flow algorithm does not specifically discourage flow paths with many edges while attempting to reduce the flow congestion via rerouting. Interestingly, the average edge-length of a final flow path is correlated with the average hop distance in the graph (with a correlation coefficient of 0.91). The correlation suggests that although exponential rather than uniform vertex lengths were used in the concurrent flow algorithm, the flow paths used tend to reflect the underlying connectivity structure of the input graph.

Size of final flow graphs. For each run, we computed the average number of non-isolated vertices and the average number of edges in a final flow graph and compared them against the number of vertices n and the number of edges m in the input graph. The average number of non-isolated vertices in a final flow graph varied between $0.035n$ and $0.42n$, while the minimum number was as low as $0.006n$ and the maximum number reached $0.92n$. In other words, each vertex in the input graph appeared as a non-isolated vertex in 3.5% to 42% of the final flow graphs, on the average. As for the number of edges in a final flow graph, the average ranged between $0.008m$ and $0.22m$, and the maximum did not exceed $0.49m$.[26] We see that on the average, each edge in the input graph is used in 0.8% to 22% of the final flow graphs.

The sparsity of the final flow graphs is a consequence of the fact that few active flow paths are typically needed to route each commodity, and that flow paths typically have a low number of edges, as we saw earlier. Additionally, the size of a flow graph is correlated with the average hop distance of the underlying graph, with correlation coefficients of 0.91 and 0.90 respectively for the average number of non-isolated vertices and the average number of edges. These correlations are related to that we saw earlier betweeen the average length of a final flow path and the average hop distance, since each flow graph is an aggregation of the small number of flow paths for 3 commodities.

We also collected statistics on the size of the intermediate flow graphs during a run, and they turn out to be similarly sparse. We exploit the sparsity of flow graphs by streamlining the computation of longest paths in those graphs to ignore the high number of isolated vertices.

Statistics for Rerouting Steps

We now consider several aspects of each rerouting step. Some of the statistics we present will help to explain why fragmentation did not occur in the final concurrent flows.

Number of commodities examined per reroute. Each rerouting step involves looking for a commodity with an ϵ-bad longest path to reroute flow from. For the matrices without ϵ-optimal initial flows, the average number of commodities examined per step was at most 1.6. However for the matrices with ϵ-optimal initial flows, the number was as high as 15.4. The results show that when a flow is not ϵ-optimal, it is easy to find a bad commodity to reroute, but that when ϵ-optimality is reached, there are much fewer bad commodities and it becomes more difficult to find them. In addition, they agree with our earlier discussion on optimality checks, where we saw that the evaluation of the dual objective value was always triggered by the ϵ-good commodity check in the runs with ϵ-optimal initial flow, and almost always by the periodic check for the other runs.

Amount of flow rerouted. As remarked in the earlier discussion on fragmentation, our implementation uses flow values which are an integer multiple of a

[26] Since the directed flow graphs are kept acyclic, it is more appropriate to express the number of edges in a flow graph as a fraction of m instead of $2m$.

common flow quantum σ. In addition, instead of rerouting only σ units of flow at each step, as described in Klein, Plotkin, Stein and Tardos [52], we reroute a multiple of σ units of flow to minimize some potential function. For the runs that we performed, the average amount of flow rerouted per step varied between 30σ and 480σ, while the maximum amount typically exceeded 1000σ. Furthermore, the average fraction of flow rerouted away from a longest path varied between 0.17 and 0.81 across the runs, and was typically around 0.45, while the average amount of flow left on the longest path was between 16σ and 4300σ, and was typically several hundred σ.

The high average rerouted flow reflects the practical utility of rerouting more than one quantum of flow. More importantly, the high average rerouted flow and the high average flow remaining on the longest path show that the rerouting phase typically does not produce flow paths with low flow values, and explains the observation we made earlier on the distribution of flow path values in the final concurrent flow.

Lastly, we observe that for some runs, the maximum amount of flow rerouted in a single step exceeds 1, which is the demand for each commodity. This is due to the aggregation of flow paths for commodities with a common source into a single flow graph, which admits the possibility of an active path carrying flow for more than one commodity, as was discussed in Section 2.3.1.

Congestion after rerouting. As indicated in the preceding paragraph, the amount of flow rerouted in each step is chosen with the purpose of minimizing a potential function, namely the sum of the lengths of the vertices. We saw in Section 2.2.2 (cf. Lemma 2.5 and Theorem 2.6) that the behavior of this potential function allows us to bound the number of iterations taken by a REDUCE call to halve the flow congestion. However, it is not necessarily the case that each reroute will actually decrease the flow congestion. In fact, for our test runs, typically 2% to 4% of the reroutes resulted in an increase in the flow congestion, with some instances where the percentage of such reroutes exceeds 10%.[27] The actual increase in the congestion is small though, and never exceeds 1%.

Complete reroutes. It is possible that the prescribed amount of flow to reroute (based on minimizing the potential function) can exceed the flow on the path P that we are rerouting from, and the rerouting step then takes all the flow away from P. In fact, this situation occurs for all but three of the test matrices, and between 7% to 59% of the reroutes result in all flow from one path being rerouted to another. The high incidence of such complete reroutes shows that the concurrent flow algorithm frequently replaces one active flow path by another and does not increase the number of active flow paths, and explains why there would be a small number of active flow paths per commodity, as we saw when examining the

[27] There was one matrix (CAN292) with an ϵ-optimal initial flow where the flow congestion always decreased after each of the small number of reroutes actually performed.

statistics on the final flow graphs.[28] It is also interesting to note that the average amount of flow rerouted in a complete reroute varies between 64σ and 5120σ across the runs, and typically exceeds 300σ, showing that complete reroutes do occur on flow paths with significant amount of flow.

Other Computational Statistics

Finally, we present some statistics on other computational aspects of the concurrent flow algorithm in practice.

Shortest path computation. The setup for our shortest path computation is incremental in that if two successive computations are from the same source vertex, we avoid any unnecessary recomputation to previously reached vertices (cf. Section 2.3.5). The incremental setup is especially beneficial in situations where the processing of commodities is by the order of their source vertices, eg. during initial routing, for then the shortest path computations from the same source are grouped together. However, since a randomized distance-weighted rule is used for commodity selection during rerouting, it is unlikely that two successive selected commodities would share the same source vertex. Indeed, for the runs we performed, less than 0.2% of successive commodities share the same source vertex, and each shortest path computation practically begins from a new source vertex.

Recall that the shortest path computation reaches vertices in order of their distance from the source vertex, and terminates when the target vertex is reached. It turns out that between one-half to three-quarters of the vertices in the input graph are reached on the average during a shortest path computation from a source s to a desired target t. Let ρ denote the number of vertices reached. A regression computation shows that ρ is highly correlated to the number of vertices n, with a correlation coefficient of 0.987, and the best linear fit is given by: $\rho = 0.553n$. Therefore the average running time of a shortest path computation is about half the worst-case running time.

Estimated shortest path lengths. As was described in Section 2.3.2, it is necessary to scale the exponential vertex length function that we use to avoid numerical overflow. Consequently. small vertex lengths might instead underflow and be set to 0. It is possible that a shortest path that we reroute flow onto might be composed entirely of vertices whose lengths underflowed, and so the length of the path has to be estimated (using the logarithms of the vertex lengths) in order to determine how much flow to reroute. This complication rarely happens in practice though. For half of the runs, it never occurred; while for the other half, the incidence of estimated shortest path lengths is at most 0.2% of the number of reroutes.

[28] Note that the significant tendency of the concurrent flow algorithm to replace one flow path by another would have been less readily observable if we only reroute σ units of flow at each step, for then several steps might be necessary to exhaust an active flow path.

Presence of cycles in flow graphs. As mentioned in Section 2.3.6, we did not use the sophisticated cycle-cancellation algorithms of Sleator and Tarjan [82] to cancel any cycles that might arise after rerouting in a flow graph, but instead relied on invocations of directed depth-first-search to detect cycles. That implementation decision was based on some early testing, which showed that cycles are infrequently created by rerouting. The set of runs that we performed somewhat corroborate the earlier observation, with the quality of the initial flow strongly correlated with the incidence of cycles in the flow graphs.[29] Note that it is possible for the initial flow graphs themselves to have cycles created by the use of four shortest paths for each commodity. Such cycles are detected and removed before rerouting begins.

For matrices with ϵ-optimal initial flows, it turns out that between 0% to 44% of the initial flow graphs had cycles. In addition, for the runs involving those matrices, cycles due to rerouting occurred after 1.2% to 45% of the reroutes. However for matrices without ϵ-optimal initial flows, more than half of the initial flows did not have cycles in the flow graphs, and for those that did, cycles appeared within 0.15% to 11% of the initial flow graphs. Morever, the incidence of cycles after rerouting varied between 0.06% and 11%.

The high percentage of cyclic flow graphs for some of the runs with ϵ-optimal initial flow graphs show that our earlier statistics on the presence of cycles in flow graphs were not completely valid. We therefore instrumented the performance of our DFS-based cycle cancellation routine to measure the effort involved in cancelling cycles. Recall from Section 2.3.6 that our routine uses a sequence of DFS computations to detect cycles caused by back edges in the associated DFS tree. For an acyclic flow graph, one DFS invocation would confirm that the graph is acyclic by the absence of back edges. For a cyclic flow graph, the first DFS call will return with back edges, and we cancel the cycles formed by the back edges and the DFS tree. However the modified flow graph may still have cycles involving *forward* edges, which were discarded during the DFS computation. A second DFS call is then made, which either verifies the acyclicity of the modified flow graph, or detects other cycles, resulting in further DFS calls.

For our runs, it turns out that at most 4 rounds of DFS were necessary in our cycle cancellation routine. In fact only 0.05% of the *cyclic* flow graphs presented to our cancellation routine required 4 DFS rounds to remove all cycles and confirm acyclicity, and at most one such instance occurred in any run. The overall incidence of cyclic flow graphs requiring 3 DFS rounds was 2.5%, but was as high 19% for one of the runs. Note that the percentage of 3-round and 4-round cyclic graphs are computed relative to the number of cyclic graphs, and the percentages would be much lower if they were measured relative to the number of flow graphs examined. Indeed, for any run, at most 1.8% of the flow graphs required 3 rounds of DFS to cancel cycles. The overall low fraction of multiple rounds of DFS needed to cancel cycles show that our simple cycle-cancellation routine is very effective, and that

[29] The earlier runs used a much less complicated initial routing where the entire demand of each commodity was routed along a min-hop path, and do not produce initial flows that are ϵ-optimal.

more sophisticated algorithms are unlikely to yield better practical performance.

Comparison of α to prescribed value. Recall that the exponential vertex length function $l(v) = e^{\alpha f(v)}$ depends on a parameter α which is used essentially to control the sensitivity of the length function to small changes in vertex flows. In Klein, Plotkin, Stein and Tardos [52], a value of α is prescribed that satisfies some technical condition throughout a Reduce call. However in our implementation, we use a smaller value that is obtained by some other formula (cf. Section 2.3.7), with the consequence that α might have to be adjusted upwards within a Reduce call. For our runs, it turns out that the final value of α is 8.9 to 78 times less than its prescribed value. This confirms that the prescribed value is too large in practice, and that the idea of using a much smaller value of α is useful. In addition, for 85% of the runs, it was not necessary to adjust α, but there were two matrices for which the final value of α was more than 10 times higher than its initial value.

Range of α values. For our runs, the initial value of α ranges between 0.09 and 9.4, while the final value of α ranges between 0.09 and 9.8, showing that α varies widely with the matrix. However the product of α and the flow congestion $|f|$, which determines the largest vertex length, varies less widely. The initial value of the product is between 9.5 and 174, while the final value is between 20 and 173.

2.5.3 Shahrokhi's Minimum Spanning Tree Heuristic

Shahrokhi [79] proposed a heuristic for improving the dual length function for *edge-based* concurrent flow. Given a length function l on the edges, the heuristic first computes a minimum spanning tree T on the input graph using the edge lengths as edge weights. Let e^* be an edge of minimum weight in T and let $(S, V \setminus S)$ be the vertex partition arising from the removal of e from T. It is well known [15] that the edge cut $(S, V \setminus S)$ in the input graph maximizes the minimum weight of an edge crossing a cut. Suppose there are r edges crossing this cut. The heuristic assigns each crossing edge a length of $1/r$, while all other edges are assigned zero length. It turns out that for the structured graphs that Shahrokhi [79] tested his concurrent flow algorithm on, the heuristic always improved the dual objective value. Subsequently, Borger *et al* [11] used the heuristic in their edge-based implementation of the Klein, Plotkin, Stein and Tardos [52] concurrent flow approximation algorithm. There they used the heuristic to construct an alternative dual solution to aid in testing for ϵ-optimality.

There is no direct vertex-based analog of minimum spanning trees that we can use for our vertex-based concurrent flows. However, we can construct a vertex equivalent of the special cut derived from the minimum spanning tree. Specifically, we want to find a vertex separator $S^* \in V$ (i.e. the removal of S^* disconnects the input graph) that maximizes the minimum length of a vertex in it. This special separator is easily found by removing from the graph the vertices in decreasing order of length until the graph becomes disconnected. The set S of removed vertices gives the desired vertex separator since any subset of it that does not contain the shortest vertex is not a separator of the graph. However, in general, S is not minimal in that a proper subset of it (but still containing the shortest vertex) is a separator. Consider the components remaining when S is removed from the input graph. We make S minimal by moving any vertex in it that is only adjacent to one component into that component.[30]

We evaluate the effectiveness of our analog to Shahrokhi's heuristic by applying it as a postprocessing step to the ϵ-optimal dual solution that was produced. For the set of test matrices that we used, the results were mixed. For 13 of the 33 test matrices, the alternative dual was significantly better than that produced by the concurrent flow approximation algorithm, typically halving the achieved error. In one extreme case, the new dual solution was *optimal*, having the same objective value as the final concurrent flow. However, for the other test matrices, the dual solutions derived from the heuristic were abysmal, with objective values ranging from 1/15 to 2/3 of that of the original dual solutions.

We see that the dual produced by the heuristic is sometimes significantly better than the original dual, and can be used in optimality checks to establish ϵ-optimality of the concurrent flow more quickly, as was used by Borger *et al* [11]. The enhanced optimality check approach is suitable if the primary object of interest is the concurrent flow produced. However as we described in Section 1.6 and also in Section 2.5.1, the concurrent flow algorithm is used to produce a good dual length function for use by the sparse cut construction procedure of Leighton and Rao [57]. Using the heuristic-based dual as a starting point for the Leighton and Rao sparse cut procedure is questionable, as that dual is already a cut and is likely to bias the output of the construction procedure towards itself. Instead, it is better to use the actual exponential length function produced by the algorithm for the Leighton and Rao procedure. We conclude that for our application of the concurrent flow algorithm, the heuristic is not appropriate both during testing for ϵ-optimality, where the modified algorithm might terminate with an alternate dual that is ϵ-optimal but an actual dual that is not, and also as a postprocessing step.

[30]If S separates the graph into more than two components, we can find a still smaller subset of it that only separates the graph into two components. We did not, however, apply this additional improvement.

Chapter 3

Sparse and Balanced Vertex Cuts

Small balanced cuts play an important role in the design and analysis of divide-and-conquer graph algorithms. Informally, a cut is "balanced" if its removal decomposes the graph into pieces of roughly equal size. Such a cut is desirable in divide-and-conquer algorithms as its balancedness ensures that the recursive subproblems on the remaining components are of roughly the same size. At the same time, its size controls the extent of interaction between it and the subproblems. This cut-subproblem interaction usually introduces additional cost when solutions to the subproblems are combined together to form a solution to the original problem.

For some classes of graphs, most notably planar graphs ([62]), balanced cuts with small size are known to exist. However, in general, most graphs do not have small balanced cuts. In fact, the problem of finding a balanced cut of minimum size on an arbitrary graph is NP-complete [57]. Leighton and Rao [57] gave the first algorithm for finding balanced cuts of approximately optimal size. Since then, they and many other researchers have used these cuts to develop polylogarithmic-times-optimal recursive approximation algorithms for many graph problems, including the elimination ordering algorithm that we shall study in Chapter 4. The Leighton-Rao balanced cut construction and their associated sparse cut construction form the subject of this chapter.

We first examine the intimate relation between cuts and commodity flows in Section 3.1, and then present the Leighton-Rao construction for sparse vertex cuts in Section 3.2. Unlike the treatment of vertex cuts in the literature, we do not reduce the vertex cut problem to the directed edge cut problem. Instead, the construction we present works directly on the input graph, and is considerably simpler than its edge-based counterpart. In Section 3.3, we consider a further simplification to the sparse cut construction. These constructions are empirically evaluated in Section 3.4. We also consider and evaluate in Section 3.5 the use of simulated annealing for finding sparse cuts.

In Section 3.6, we present the Leighton-Rao balanced cut construction and evaluate it against a balanced cut heuristic found in the sparse matrix package SPARSPAK [35]. Finally, in Section 3.7, we introduce a nonstandard cost measure for balanced cuts, and evaluate the use of simulated annealing for finding such balanced cuts.

3.1 Sparse Cuts and Multicommodity Flows

3.1.1 Edge Cuts and Multicommodity Flows

Single- and 2-Commodity Flows

We begin by briefly reviewing the classical max-flow min-cut results for single and 2-commodity flows. Let $G = (V, E)$ be an undirected graph, where each undirected edge e has a nonnegative capacity $c(e)$. In the single commodity flow problem, we are given a source vertex s and a sink vertex t, and the objective is to send as much flow as possible from s to t. Flow can go along both directions of an edge, subject to the constraint that the sum of flow along both directions of an edge cannot exceed its capacity. The dual of the flow problem is the edge cut problem. An edge cut $(A, \bar{A})$ is a partition of the vertices of the graph into two nonempty sets A and $\bar{A} = V \setminus A$. The capacity of the cut, denoted by $c(A, \bar{A})$, is the sum of the capacities of the edges crossing the cut, i.e.

$$c(A, \bar{A}) = \sum_{uv \in E: u \in A, v \in \bar{A}} c(uv).$$

We say that the cut separates s from t if exactly one of s and t is in A. Clearly the amount of flow that can be sent from s to t cannot exceed the capacity of any cut $(A, \bar{A})$ separating s from t, for the entire flow has to be routed through the edges in $(A, \bar{A})$. In particular, the *min-cut*, which is defined to be the minimum capacity among all cuts separating s from t, is an upper bound on the amount of the flow that can be sent. The celebrated result of Ford and Fulkerson [25] shows that the upper bound is in fact achievable: the max-flow is *equal* to the min-cut.

An analogous max-flow min-cut theorem exists for 2-commodity flows. In the 2-commodity flow problem, there are two source-sink pairs (s_1, t_1) and (s_2, t_2), and we wish to simultaneously send the first commodity from s_1 to t_1 and the second from s_2 to t_2, with the total amount of commodities through any edge not exceeding its capacity. For one optimization version of the problem, the objective is to maximize the total amount of commodities shipped.[1] In [45], Hu showed the following max-flow min-cut result for 2-commodity flows, which generalizes the result of Ford and Fulkerson: the maximum total amount of commodities that can be sent is equal to the minimum capacity among all cuts separating both s_1 from t_1 and s_2 from t_2.

Multicommodity Flows

In the multicommodity flow problem, we are given $k > 1$ commodities, each with its own source s_i, sink t_i and demand $d(i)$. For the feasiblity version of the problem,

[1]For the feasiblity version of the problem, we are also given demands $d(1)$ and $d(2)$ for each commodity, and wish to know if it is possible to simultaneously ship $d(1)$ units of commodity 1 and $d(2)$ units of commodity 2 without violating the capacity constraints.

the objective is to simultaneously route $d(i)$ units of commodity i from s_i to t_i, with the total amount of commodities through any edge not exceeding its capacity. Clearly not all instances of the multicommodity flow problem are feasible. It is often more useful to consider instead an optimization version of the problem, where the objective is to maximize the amount of commodities sent. There are various ways to define the max-flow for a multicommodity flow, and we choose a variant known as the *concurrent flow problem*, which was first introduced by Shahrokhi and Matula [80]. In the concurrent flow problem, the objective is to maximize a common fraction z of each commodity that is sent, i.e. the max-flow is the maximum fraction z such that there is a feasible flow for all the commodities with scaled demands $zD(i)$. The fraction z is also known as the *throughput* of the flow, and can exceed 1 in instances that are already feasible.

The corresponding cut problem for concurrent flows is defined differently from that for single- and 2-commodity flows. Instead of restricting our attention to edge cuts which separate the source and sink of each commodity, of which there might be none when there are many commodities, we consider each possible cut but use a cost function that reflects the commodities that are separated by the cut. Specifically, for a cut $(A, \bar{A})$, we denote by $d(A, \bar{A})$ the total demand across the cut:

$$d(A, \bar{A}) = \sum_i \{d(i) : \text{exactly one of } s_i \text{ and } t_i \text{ is in } A\}.$$

Note that at least $d(A, \bar{A})$ total flow has to be sent across the cut in the feasiblity version of the problem, and therefore the ratio $c(A, \bar{A})/d(A, \bar{A})$ limits the common fraction of the demand that can be sent across the cut in a feasible flow. We define the *min-cut* $\mathcal{S}$, also known as the *sparsest cut*, to be

$$\mathcal{S} = \min_{A \subset V} \frac{c(A, \bar{A})}{d(A, \bar{A})},$$

and we readily see that the min-cut is an upper bound on the max-flow. It is easy to verify that the min-cut definition generalizes that for the single commodity flow problem (assuming the single commodity has demand one).

Unlike the single- and 2-commodity flow problems, the max-flow of a concurrent flow might be lower than the min-cut. Consider for example, the 4-commodity concurrent flow instance in Figure 3.1, which is taken from Okamura and Seymour [68] (and also cited by Leighton and Rao [57]). In the example, each commodity has demand 1 and each edge has capacity 1. For each commodity, any path from its source to its sink has at least 2 edges. Since the total demand is 4 and the total edge capacity is 6, it follows that the max-flow is at most $6/(4 \cdot 2) = 3/4$. Indeed the max-flow is 3/4, which is achieved by sending for the commodities $i = 1, 2, 3$, 3/8 units of flow along the two 2-edge paths from s_i to t_i, and sending 1/4 unit of flow along the three 2-edge paths joining s_4 and t_4. However, we can verify by enumeration that the min-cut is 1 (eg. consider the cut with only s_1 on one side).

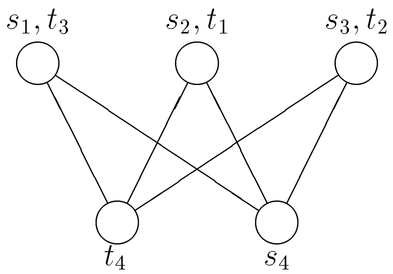


Figure 3.1: An instance of the concurrent flow problem where the max-flow is smaller than the min-cut.

The Uniform Multicommodity Flow Problem

The uniform multicommodity flow problem (UMFP) is a special case of the concurrent flow problem. There is a commodity for every pair of vertices, and the demands of all the commodities are the same and assumed to be one without loss of generality. Observe that the demand across a cut $(A, \bar{A})$ is simply $|A||\bar{A}|$, and so the ratio of the capacity of a cut to the demand across it simplifies to $c(A, \bar{A})/(|A||\bar{A}|)$. This last quantity is also commonly known as the *ratio cost* of the cut $(A, \bar{A})$. The min-cut of the UMFP is

$$\mathcal{S} = \min_{A \subset V} \frac{c(A, \bar{A})}{|A||\bar{A}|}.$$

For the UMFP, Leighton and Rao [57] have shown that the max-flow and min-cut can be separated by more than a constant. They constructed two families of UMFP instances, one where the underlying network is an expander graph and the other where it is not, in which the max-flow is an $O(\log n)$ *factor* smaller than the min-cut. An expander graph $G = (V, E)$ has the property that for some constant β, every subset U of V with at most half the vertices has $\beta|U|$ edges with exactly one endpoint in U.

Leighton and Rao also showed, however, that the gap cannot be bigger, by giving a construction for a cut whose ratio cost is $O(z \log n)$.

Theorem 3.1 ([57], Theorem 2) *For any uniform multicommodity flow problem,*

$$\Omega(\mathcal{S}/\log n) \leq z \leq \mathcal{S},$$

where z is the max-flow and $\mathcal{S}$ is the min-cut of the UMFP.

The Leighton-Rao cut construction and max-flow min-cut theorem for UMFPs can be generalized to product multicommodity flow problems, where the demand for any pair of vertices u and v is $\pi(u)\pi(v)$ for some function π on the vertices. Their results also extend to directed graphs, to flow problems where the flow paths are required to be short, and to vertex cuts via a standard reduction to directed flows. The last extension is of particular interest, for it forms the basis of the cut that we use in the divide-and-conquer elimination ordering algorithm.

By viewing graphs as geometric objects, Linial, London and Rabinovich [60] extended the Leighton-Rao max-flow min-cut theorem to arbitrary commodity settings. They also gave an algorithm for finding a nearly tight cut meeting the bound.

3.1.2 Vertex Cuts

In the previous section, we saw how small edge cuts were related to edge-capacitated multicommodity flows. A similar relationship exists between vertex cuts and vertex-capacitated flows, which we shall develop in this and the following section.

Let $G = (V, E)$ be a connected undirected graph where each vertex $v \in V$ has a positive integral capacity $c(v)$. A *vertex cut* (also known as vertex separator) of G is a partition (A, B, C) of V with $C \neq \emptyset$ such that the removal of C (and its incident edges) disconnects A from B. Equivalently, there is no edge between a vertex in A and a vertex in B. The capacity of the vertex cut (A, B, C) is defined to be $c(C)$, which is the sum of the capacities of the vertices in C.

A Transformation from Vertex Cuts to Edge Cuts

There is a well-known transformation for converting a vertex-cut problem on an undirected graph to an edge-cut problem on a directed graph. We briefly describe the transformation as it motivates the definitions that we shall use for the vertex cut problem. The transformation constructs a directed graph $G^* = (V^*, E^*)$ corresponding to our undirected graph, where

$$V^* = \{v | v \in V\} \cup \{v' | v \in V\}$$

and

$$E^* = \{(v, v') | v \in V\} \cup \{(u', v) | \{u, v\} \in E\}.$$

There is a pair of vertices v and v' in the directed graph for each vertex v in the undirected graph, and the pair is joined by an edge. In addition, each undirected edge uv gives rise to two directed edges (u', v) and (v', u). Capacities are assigned to each directed edge, with $c(v, v') = c(v)$ and $c(u', v) = \infty$.

The transformation induces a correspondence between vertex cuts in the undirected graph G and finite-capacity edge cuts in the directed graph G^*. Consider a directed edge cut $(U, V^* \setminus U)$ of G^*. The edge cut induces a partition (A, B, C) of V where

$$A = \{v \in V : v, v' \in U\}, \quad B = \{v \in V : v, v' \in V^* \setminus U\} \text{ and } C = V \setminus (A \cup B).$$

It is easy to see that if there are no infinite capacity edges crossing the cut $(U, V^* \setminus U)$ in either direction, then (A, B, C) is a vertex cut. Similarly, given a vertex cut (A, B, C), we can define a directed edge cut $(U, V^* \setminus U)$ with $U = \{v : v \in A \cup C\} \cup \{v' : v \in A\}$, and show that the edge cut has finite capacity.

Cost Functions for Vertex Cuts

Observe that in the transformation, we have $|U| = 2|A| + |C|$ and $|V^* \setminus U| = 2|B| + |C|$. Following the definition of the ratio cost of an edge cut, we define the ratio cost of a vertex cut (A, B, C) to be

$$r(A, B, C) = \frac{c(C)}{(|A| + \frac{1}{2}|C|)(|B| + \frac{1}{2}|C|)}.$$

An associated and more well-studied cost function is the *quotient cost*, which is defined to be

$$q(A, B, C) = \frac{c(C)}{\min\{|A| + \frac{1}{2}|C|, |B| + \frac{1}{2}|C|\}}.$$

Note that the quotient cost of a cut is always between a factor of $n/2$ and n of its ratio cost. Therefore, given one of the two costs, the other cost can be estimated to within a factor of 2.

There are two special sparse cuts that we can define. The vertex flux α is the minimum quotient cost over all cuts:

$$\alpha = \min_{(A,B,C)} q(A, B, C),$$

and the minimum cut $\mathcal{S}$ is the minimum ratio cost:

$$\mathcal{S} = \min_{(A,B,C)} r(A, B, C).$$

The cut that achieves the vertex flux is called the *vertex flux cut*, while the cut that achieves the minimum cut is called the *sparsest cut*. From the earlier remark on the factor-of-2 estimation of the quotient and ratio costs, we know that $n\mathcal{S}/2 \le \alpha \le n\mathcal{S}$.

In this chapter, we use the term sparse cut to generically refer to a cut with low quotient or ratio cost.

3.1.3 Relation between Vertex Cuts and Flows

We examine in this section two cases where the optimal objective value of a concurrent flow can be used to bound the minimum ratio cost of a vertex cut.

A Mixed Integer Program for the Minimum Ratio Cost Cut

We first formulate the problem of finding a vertex cut with (close to) minimum ratio cost as a mixed integer program with a nonlinear objective function. As we pointed out in the previous section, for the purpose of finding a minimum ratio cost cut, there is no loss of generality in restricting our attention to cuts which divide up the graph into only two components, and where each cut vertex is adjacent to both components.

A vertex cut (A, B, C) of a graph $G = (V, E)$ is represented by associating a 0–1 decision variable $l(v)$ to each vertex v, with $l(v)$ being 1 exactly when v is in

C. To measure the number of pairs of vertices separated by the cut, we interprete the decision variables as vertex lengths and use them to define a distance function between the vertices. Specifically, we let the vertex lengths induce edge lengths, in which the induced length of an edge vw is half the sum of the lengths of its endpoints: $l(vw) = \frac{1}{2}(l(v) + l(w))$. Using the standard definition of the length of a path as being the sum of the lengths of the edges on it, it follows that the length of a path $P = u - z_1 - \ldots - z_r - v$ $(r \geq 0)$ is given by:

$$l(P) = \frac{1}{2}(l(u) + l(v)) + \sum_i l(z_i).$$

Note that each vertex on P contributes its entire length to the path length, except for u and v, which only contribute half their lengths. The distance $dist_l(u, v)$ between two vertices u and v is the length of the shortest path between them.

Observe that if two vertices u and v are both in A (or B), then by definition, there is a path in G between them that does not go through vertices in C, and so $dist_l(u, v)$ is zero. In contrast, if one of u and v is in A and the other is in B, then any path between them uses at least one vertex in C. However, since each vertex in C is connected to both A and B, and distances between vertices in A (or B) are zero, it follows that in this case $dist_l(u, v)$ is exactly one.

In addition, the distance between two vertices in C is also exactly one, as seen by using a path between those two vertices that only goes through vertices in A. Lastly, if u is a vertex in A (or B) and v is a vertex in C, then $dist_l(u, v)$ must be $\frac{1}{2}$. Thus the sum of the distances $\sum_{\{u,v\} \subseteq V} dist_l(u, v)$ between all pairs of vertices is exactly

$$\begin{aligned} & |A||B| + \tfrac{1}{2}|C|(|C| - 1) + (|A| + |B|)|C| \\ = \quad & (|A| + \tfrac{1}{2}|C|)(|B| + \tfrac{1}{2}|C|) + \tfrac{1}{2}|C|(\tfrac{1}{2}|C| - 1). \end{aligned}$$

Note that the term $\frac{1}{2}|C|(\frac{1}{2}|C| - 1)$ is at least $-\frac{1}{4}$, while $(|A| + \frac{1}{2}|C|)(|B| + \frac{1}{2}|C|)$ is $\Omega(n)$. So for all practical purposes, we can assume that the sum of distances is a lower bound on $(|A| + \frac{1}{2}|C|)(|B| + \frac{1}{2}|C|)$.

Observe that the capacity $c(C)$ of the cut is equal to $\sum_v c(v)l(v)$. We therefore consider the nonlinear objective function

$$\sum_{\{u,v\} \subseteq V} dist_l(u, v) \;/\; \sum_v c(v)l(v),$$

which is at least

$$\frac{(|A| + \frac{1}{2}|C|)(|B| + \frac{1}{2}|C|)}{c(C)} = \frac{1}{r(A, B, C)}.$$

Thus the objective function upper bounds the reciprocal of the ratio cost of the cut. Since we wish to find a cut with minimum ratio cost, we maximize the objective function in the following mixed integer program:

$$
\begin{aligned}
\max \quad \sigma_{MIP} = \sum_{\{u,v\}\in V} & \hat{d}(u,v) \;/\; \sum_{v} c(v)l(v) \\
\hat{d}(u,w) \;&\leq\; \frac{1}{2}(l(u)+l(v)) + \hat{d}(v,w) && \forall\ \{u,w\}\subseteq V,\ \ uv\in E \\
l(v) \;&\in\; \{0,1\} && \forall\ v\in V \\
\hat{d}(u,v) \;&\geq\; 0 && \forall\ \{u,v\}\subseteq V
\end{aligned}
$$

The variables $\hat{d}(u,v)$ serve as lower bounds on the distances $dist_l(u,v)$. When the mixed integer program is solved, the triangle inequalities ensure that $\hat{d}(u,v)$ is equal to $dist_l(u,v)$. So the optimal objective value σ^*_{MIP} is at least $1/\mathcal{S}$.

LP Relaxation and Relation to Concurrent Flows

The mixed integer program is relaxed by allowing the "length" variables to take on arbitrary values between 0 and 1. We give below the resulting linear program, which we shall call the *distance sum* LP. Note that the normalization of the objective function is now achieved by requiring the denominator appearing in σ_{MIP} to be one.

$$
\begin{aligned}
\max \quad \sigma_{LP} = & \sum_{\{u,v\}\in V} \hat{d}(u,v) \\
\sum_v c(v)l(v) \;&=\; 1 \\
\hat{d}(u,w) \;&\leq\; \frac{1}{2}(l(u)+l(v)) + \hat{d}(v,w) && \forall\ \{u,w\}\subseteq V,\ \ uv\in E \\
l(v) \;&\geq\; 0 && \forall\ v\in V \\
\hat{d}(u,v) \;&\geq\; 0 && \forall\ \{u,v\}\subseteq V
\end{aligned}
$$

Clearly, $\sigma_{LP} \geq \sigma_{MIP}$. It is convenient to regard each pair of vertices as the terminals of a commodity. The distance sum LP is in fact the dual to an all-pairs concurrent flow problem with arbitrary capacities but with unit demands. By linear programming duality, the optimal objective value λ^* of the concurrent flow LP (which is precisely its optimal congestion) is equal to σ^*_{LP} and so it too upper bounds $1/\mathcal{S}$.

The concurrent flow formulation justifies a convenient assumption on the vertex capacities that will be useful later in proving the Leighton-Rao sparse cut construction. We first state the concurrent flow LP below, using the notation in Section 2.1. Recall that $f_i(v)$ denotes the flow through vertex v from commodity i, and that $f(v) = \sum_i f_i(v)$ is the total flow through v. In addition, flow that originates or terminates at a vertex contributes only half its value but flow that is routed through a vertex is counted fully. Since each vertex v is a terminal of at least one commodity, $f(v) \geq \frac{1}{2}$. In addition $f(v) \leq \frac{1}{2}n(n-1)$, which is the total demand.

$$\begin{array}{rcll} \min & \lambda & & \\ \sum_{P \in \mathcal{P}_i} f_i(P) & = & 1 & \forall\ 1 \le i \le k \\ f(v) & \le & \lambda c(v) & \forall\ v \in V \\ f_i(P) & \ge & 0 & \forall\ P \in \mathcal{P}_i,\ \ 1 \le i \le k \\ \lambda & \ge & 0 & \end{array}$$

Let c_{min} be the minimum capacity over all the vertices, and let v_{min} be a vertex that achieves the minimum capacity. For any feasible solution, we have $f(v_{min}) \le \lambda c_{min}$ and also $f(v_{min}) \ge \frac{1}{2}$, from which it follows that the objective λ is at least $1/(2c_{min})$. Moreover for any vertex v, $f(v) \le \frac{1}{2}n(n-1)$, and

$$\frac{f(v)}{\lambda} \le n(n-1) \cdot c_{min}.$$

We can therefore assume that $c(v) \le n(n-1) \cdot c_{min}$ for all v, and not strengthen the capacity utilization constraints nor change the optimal congestion.

Construction of Expander Graphs

The preceding LP bound on the minimum ratio cost of a cut is based on a concurrent flow with all-pairs demand. Such concurrent flows are computationally prohibitive to solve. By using a set of commodities whose demand graph is a constant low-degree expander, the concurrent flow is easier to solve. Furthermore, as we shall see later, the optimal objective value of the expander-based concurrent flow can also be used to bound (but not as strongly) the minimum ratio cost. We briefly review here how such expanders are constructed in practice.

Expander graphs have been widely studied and there are many closely related variants ([7] [9] [5] [13]). For our purpose, as we defined earlier, an undirected graph G is an expander with expansion rate β if for each nonempty proper subset $A \subset V$, the number of edges in the *edge cut* $(A, \bar{A})$ is at least $\beta \min(|A|, |\bar{A}|)$. There are several deterministic constructions for expander graphs ([4] [28] [71] [83]), but they are relatively complicated or only give graphs with certain number of vertices. For example, in the construction of Gabber and Galil [28], the number of vertices is the square of an integer. Such restrictions are hard to handle in our context, since the expanders are used in conjunction with the construction of sparse vertex cuts on graphs with an arbitrary number of vertices.

It is more common to use a randomized approach for constructing expanders. We say that a graph G is constant d-regular if each vertex has degree exactly d. A constant $2d$-regular expander is constructed by combining d perfect matchings as follows. Note that we allow multiple edges between vertices. Each perfect matching is with respect to an almost complete bipartite graph $H = (L, R, F)$, where L and R are copies of V, and F is the set of all edges between vertices in L and R, with the omission of the edges joining corresponding vertices, i.e. $F = \{(i,j) : 1 \leq i \neq j \leq n\}$. A perfect matching on F gives rise to a *derangement* on $\{1, \ldots, n\}$, a special kind of permutation where no element is mapped onto itself.

It can be shown ([5]) that with high probability, the resulting graph is an expander for some appropriately chosen expansion rate. Note that in a concurrent flow problem where the demand graph is a constant $2d$-degree expander, each vertex is the source of d commodities and also the sink of d commodities.

Bounds from Concurrent Flows with Expander Demand Graphs

We now show that the congestion of a concurrent flow where the demands are defined by a constant degree expander can also be used to bound the minimum ratio cost $\mathcal{S}$. To do this, it is equivalent to work with the distance sum LP. Let β be the expansion rate, and let (A, B, C) be some vertex cut with $|A| \leq |B|$. We first divide C into two subsets C_1 and C_2 with $|C_1| = \lfloor |C|/2 \rfloor$. Consider the *edge* cut $(A \cup C_1, B \cup C_2)$. We know that there are at least $\beta(|A| + |C_1|)$ edges of the demand graph crossing this cut, i.e. there are at least $\beta(|A| + |C_1|)$ commodities with one terminal in $A \cup C_1$ and the other in $B \cup C_2$. For a commodity with none or both terminals in C, we saw earlier that the distance between its terminals is one. For a commodity with only one terminal in C, the distance between its terminals is $\frac{1}{2}$. Therefore, the sum of distances between the terminals of all commodities is at least $\beta(|A| + |C_1|)/2$, and the optimal objective value σ^*_{LP} is at least

$$\frac{\beta(|A| + |C_1|)}{2c(C)} = \Omega\left(\frac{1}{q(A,B,C)}\right) = \Omega\left(\frac{1}{r(A,B,C)}\right).$$

Note that the bound is not as tight as the all-pairs bound.

Relating the Congestion of All-Pairs and Expander Demands

We saw that the optimal congestion of a concurrent flow with all-pairs demand and with expander-like demands can be used to bound sparsest cuts. In fact there is a simple relation between the optimal congestion for those two cases where the expander-like demands is constructed using the randomized approach described earlier.

Let σ^*_{AP} and σ^*_{EXP} be the optimal objective values respectively for the distance sum LP for the all-pairs and expander demand cases. Observe that for a pair of vertices u and v, the probability that they are the terminals of a commodity in the expander case is $2d/(n-1)$ since there is a probability of $d/(n-1)$ that v was picked when selecting the sinks for commodities with source vertex u, and similarly for commodities with source vertex v.

Consider an optimal length function l^* for the all-pairs case. Since l^* is not necessarily optimal for the expander demand case, the objective value of the expander demand case for the length function l^* is no more than σ^*_{EXP}. Taking expectations over this inequality, and using the preceding obseravation on the probability of a pair of vertices being the terminals of a commodity, we see that

$$\frac{2d}{n-1}\sigma^*_{AP} \leq E(\sigma^*_{EXP}).$$

3.2 The Leighton-Rao Sparse Vertex Cut Construction

We now present in detail the Leighton-Rao construction of an approximately optimal sparse vertex cut. The quotient cost of the cut produced is within an $O(\log n)$ factor of the minimum quotient cost α. As we remarked earlier, vertex cuts can be reduced to finite-capacity directed edge cuts, and so the sparse directed edge cut construction of Leighton and Rao [57] can be used to construct sparse vertex cuts. Unfortunately, the directed construction is complicated as it involves "inward" and "outward" distances due to the asymmetry of the directed graph. Instead of running the directed edge cut construction on the reduction of the input graph, we exploit the special properties of the reduction to simplify the directed construction, and then interprete the construction with respect to the actual input graph. We thus end up with a direct vertex-based construction. Since this direct construction has not been explicitly described in the literature, we will give a careful account of it in this section.

3.2.1 Stages of the Leighton-Rao Construction

Stage One of the Construction

The Leighton-Rao sparse vertex cut construction is divided into three stages. Let $G = (V, E)$ be the input graph, which is assumed to be connected. The first stage involves finding a suitable length function on the vertices. This is accomplished by solving the dual to a unit-demand concurrent flow problem on G, where there is a commodity for each pair of vertices, or where the demand graph is a constant degree expander. In the case where the vertices have unit capacities, the concurrent flow problem and its dual can both be approximately solved by our implementation of the vertex-based version of the approximation algorithm of Klein, Plotkin, Stein and Tardos [52], as was described in Chapter 2. Note that it is much faster to solve the concurrent flow problem when the demand graph is an expander since there are far fewer commodities ($O(n)$ as compared to $O(n^2)$).

Recall that for the (edge-based) single-commodity flow problem, there is an optimal dual solution where the set of edges with nonzero lengths define a minimum cut. The result is not true for general multicommodity flows. In fact, it is possible that an optimal vertex length function assigns nonzero lengths to all the vertices. It is unlikely, as Leighton and Rao [57] have observed, that a simple thresholding or rounding procedure can be applied to an optimal vertex length function to yield a sparse cut. Instead, the second and third stages of the Leighton-Rao construction

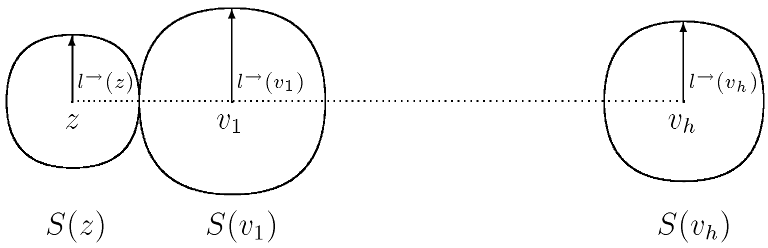


Figure 3.2: "Geometric" interpretation of a path $z \to v_1 \to \cdots \to v_h$

uses a discretized version of the vertex lengths to "grow" regions around randomly chosen vertices, and examines the cuts that are naturally associated with those regions. It will turn out that longer vertices will have a higher chance of being in the examined cuts, and hence in the sparse cut that is produced.

Discretized Vertex Length Function

Recall that the dual LP of the concurrent flow problem assigns to each vertex v a length $l(v)$ such that $\sum_v c(v)l(v) = 1$. The dual vertex lengths are continuous, and not directly suitable for the construction. We shall instead define a related integral length function on the vertices and associated integral path lengths. Throughout the rest of the Leighton-Rao construction, we shall exclusively use the integral vertex lengths and ignore their original dual lengths.

Let $s = \frac{1}{\sum_v c(v)}$. Since the Leighton-Rao construction grows regions around some vertex, we shall work with path lengths and distances that are measured with respect to that vertex. We define a discretized vertex length function $l^{\to}$, where $l^{\to}(v) = \max\{1, \lceil \frac{l(v)}{s} \rceil\}$. Let P be a *directed* path $z \to v_1 \to \cdots \to v_h$ ($h \geq 1$). We call z the origin vertex of P and v_h the destination vertex. The length of P is defined to be $l^{\to}(z) + 2\sum_{i=1}^{h} l^{\to}(v_i)$. Note that all the vertices on P except the origin vertex have their discretized lengths counted twice. The length of the degenerate single-vertex path z is defined to be $l^{\to}(z)$. We define the directed distance $dist^{\to}(z, v)$ from origin z to destination v to be the length of the shortest directed path from z to v. Note that the directed distances from z and the dual distances are related by: $dist(u, v) \leq \frac{1}{2}s(dist^{\to}(z, u) + dist^{\to}(z, v))$.

We give a pseudo-geometric interpretation of directed path lengths and distances that is useful in understanding these and later definitions. We can regard each vertex v as a sphere $S(v)$ of radius $l^{\to}(v)$, and a directed path $P = z \to v_1 \to \cdots \to v_h$ ($h \geq 1$) as laying out end-to-end the spheres of the vertices on P such that all the centers are collinear, as depicted in Figure 3.2. The directed length of P is then the Euclidean distance from the center of $S(z)$ to the furthermost point in $S(v_h)$. Note however that it is not possible to place the vertex spheres in a Euclidean space of any dimension such that for all paths simultaneously, the centers of the spheres of the vertices on each path are collinear. Our geometric interpretation only applies to a single directed path in isolation.

We can use the geometric interpretation to refine the notion of when a vertex v

is reached. Consider a shortest directed path $z \to \cdots \to w \to v$ from z to a vertex v. By definition, the distances from z to v and to its predecessor w are related by: $dist^{\rightharpoonup}(z,v) = dist^{\rightharpoonup}(z,w) + 2l^{\rightharpoonup}(v)$. Under the geometric interpretation of the path, the nearest point on the sphere $S(v)$ from z is at Euclidean distance $dist^{\rightharpoonup}(z,w)$, while the furthest point is at Euclidean distance $dist^{\rightharpoonup}(z,v)$. The remaining points on the sphere have intermediate Euclidean distances from z. We can therefore say that with respect to z, v is *almost reached* at directed distance $dist^{\rightharpoonup}(z,v) - 2l^{\rightharpoonup}(v)$, is *partially reached* at directed distances $dist^{\rightharpoonup}(z,v) - 2l^{\rightharpoonup}(v) + 1, \ldots, dist^{\rightharpoonup}(z,v) - 1$, and is *fully reached* at directed distance $dist^{\rightharpoonup}(z,v)$. Geometrically, more and more of the sphere $S(v)$ is "consumed" as v passes from the almost reached state through the partially reached states to the fully reached state. The gradual "consumption" of $S(v)$ is useful when we later grow regions from a center one unit of distance at a time and wish to assign a weight to the region.

Regions Around a Vertex

The Leighton-Rao construction produces an approximately optimal sparse cut by growing regions around vertices of the input graph. Intuitively, a region around a vertex z is the set of vertices within a certain given distance of z, and its (exterior) boundary is the set of vertices that are just beyond. The boundary acts as a "barrier" between the region and the other vertices in the graph, and gives rise to a vertex cut.

More formally, for a given root vertex z and an integer radius $r \geq 0$, we define the region $B(z,r)$ and its exterior boundary $\partial B(z,r)$ as follows:

$$\begin{aligned} B(z,r) &= \{v : dist^{\rightharpoonup}(z,v) \leq r\} \\ \partial B(z,r) &= \{v : dist^{\rightharpoonup}(z,v) - 2l^{\rightharpoonup}(v) \leq r\} \setminus B(z,r). \end{aligned}$$

$B(z,r)$ is simply the set of vertices within directed distance r of z. Recall from the discussion on directed path lengths that for a vertex $v \neq z$, $dist^{\rightharpoonup}(z,v) - 2l^{\rightharpoonup}(v)$ is the distance from z to its shortest path predecessor. Thus a vertex $v \neq z$ is in $\partial B(z,r)$ if it is not in $B(z,r)$ but its shortest path predecessor is. As for the vertex z, the definition places it in $\partial B(z,r)$ for radius $r < l^{\rightharpoonup}(z)$, and in $B(z,r)$ for radius $r \geq l^{\rightharpoonup}(z)$. We see that $\partial B(z,r)$ is the set of vertices that are almost or partially reached at distance r.

We can in fact interprete $\partial B(z,r)$ in terms of neighborhoods. For a set X, let $\Gamma(X)$ be the set of neighbors of X, i.e.

$$\Gamma(X) = \{v \in V : vw \text{ is an edge for some } w \in X\}.$$

For $B(z,r) \neq \emptyset$, we see that a vertex v is in $\partial B(z,r)$ exactly when it is not in $B(z,r)$ but is adjacent to a vertex in $B(z,r)$. Thus $\partial B(z,r) = \Gamma(B(z,r)) \setminus B(z,r)$. This shows that $(B(z,r), V \setminus (B(z,r) \cup \partial B(z,r)), \partial B(z,r))$ is in fact a vertex cut, provided that $\partial B(z,r) \neq \emptyset$. The last observation is degenerately true when $B(z,r) = \emptyset$, for then $\partial B(z,r) = \{z\}$.

Observe that a vertex v is in $\partial B(z,r)$ for the radii $r = dist^{\rightarrow}(z,v) - 2l^{\rightarrow}(v)$, $dist^{\rightarrow}(z,v) - 2l^{\rightarrow}(v)+1, \ldots, dist^{\rightarrow}(z,v)-1$. Thus v lies on the boundaries of $2l^{\rightarrow}(v)$ regions around z. We see that a longer vertex has a higher chance of being in a boundary, and hence being in a cut examined by the Leighton-Rao construction, in particular in the cut that is produced.

Weight of a Region

We now define a weight function on the regions centered at a vertex z. The weight $W(z,r)$ of the region $B(z,r)$ is defined inductively:

$$\begin{aligned} W(z,0) &= 0, \\ W(z,r+1) &= W(z,r) + c(\partial B(z,r)) \quad \text{for } r \geq 0. \end{aligned}$$

Each time the radius of the region increases by one, its weight increases by the total capacity of the vertices in the boundary of the region. A vertex v that appears in b contiguous boundaries among the boundaries $\partial B(z,0), \partial B(z,1), \ldots, \partial B(z,r-1)$ contributes a weight of $c(v)b$ to the weight $W(z,r)$. The maximum contribution of a vertex $v \neq z$ is $2c(v)l^{\rightarrow}(v)$, since v is contained in the boundaries of exactly $2l^{\rightarrow}(v)$ regions around z. With these observations, it is easy to verify inductively that for $r \leq l^{\rightarrow}(z)$, $W(z,r) = c(z)r$, and that for $r > l^{\rightarrow}(z)$,

$$W(z,r) = c(z)l^{\rightarrow}(z) + \sum_{v \in B(z,r): v \neq z} 2c(v)l^{\rightarrow}(v) + \sum_{v \in \partial B(z,r)} c(v)\{r - [dist^{\rightarrow}(z,v) - 2l^{\rightarrow}(v)]\}.$$

The equivalent expression for the weight function shows that the weight of a region is related to the vertices in the region and its boundary, and is best understood using the geometric interpretation of directed path lengths discussed earlier. In that interpretation, there is a sphere of diameter $2l^{\rightarrow}(v)$ corresponding to a vertex v. We can regard the sphere as having total weight $2c(v)l^{\rightarrow}(v)$, but without a fixed density distribution. In general, apart from the vertex z, a vertex v in the region $B(z,r)$ contributes the total weight of its sphere $S(v)$. However, the contribution of a vertex in the boundary $\partial B(z,r)$ depends on the extent to which it has been "consumed." For a boundary vertex v, the quantity $r - [dist^{\rightarrow}(z,v) - 2l^{\rightarrow}(v)]$ gives the length of the part of the diameter of $S(v)$ that is within Euclidean distance r of the center of the sphere $S(z)$. If v is almost reached, it contributes no weight to $B(z,r)$, while if v is partially reached, the contribution is proportional to the diameter that has been reached.

Observe that the weight of a region around z is at most $2\sum_{v \neq z} c(v)l^{\rightarrow}(v) + c(z)l^{\rightarrow}(z)$. We define the weight of the graph W_G to be the total weight of all the vertex spheres:

$$W_G = 2\sum_{v} c(v)l^{\rightarrow}(v).$$

The total weight of a region around any vertex is therefore less than W_G. Using the definitions $l^{\rightarrow}(v) = \max\{1, \lceil \frac{l(v)}{s} \rceil\}$ and $s = \frac{1}{\sum_v c(v)}$, and the condition $\sum_v c(v)l(v) =$

1 from the dual linear program, we can bound W_G using the total vertex capacities:

$$W_G \leq 2\sum_v c(v)\left(1+\frac{l(v)}{s}\right) \leq \frac{4}{s}.$$

Properties of Regions

We now state several properties of regions, their boundaries and their weights. These properties will be useful in the analysis in Section 3.2.2 of the Leighton-Rao construction. We assume a more general situation in which the graph G may not be connected. In that case, a vertex that is not reachable from z will be considered to have infinite directed distance. The definitions of $B(z,r)$, $\partial B(z,r)$ and $W(z,r)$ remain the same for the disconnected case.

Some of the properties listed below are just restatements of definitions or earlier observations, while the rest follow easily from the definitions. We let Z be the vertices in the connected component containing the vertex z, and W_Z be the total weight of the spheres of the vertices in Z, i.e. $W_Z = 2\sum_{v\in Z} c(v)l^{\rightarrow}(v)$. In the case that G is connected, Z is simply the vertex set V.

Properties of $B(z,r)$, $\partial B(z,r)$ and $W(z,r)$

1. $B(z,0)=\emptyset$ and $\partial B(z,0)=\{z\}$.
2. $B(z,r)\cup\partial B(z,r)\neq\emptyset \quad \forall\ r\geq 0$.
3. For $\partial B(z,r)\neq\emptyset$, $(B(z,r), V\setminus(B(z,r)\cup\partial B(z,r)), \partial B(z,r))$ is a vertex cut of G.
4. Let r_{max} be the largest directed distance of a vertex reachable from z. Then $B(z,r)=Z$ and $\partial B(z,r)=\emptyset$ for $r\geq r_{max}$.
5. $\partial B(z,r)=\emptyset \Leftrightarrow B(z,r)=Z$.
6. $W(z,0)=0$ and $W(z,1)=c(z)$.
7. $W(z,r)\leq 2\sum_{v\in B(z,r)\cup\partial B(z,r)} c(v)l^{\rightarrow}(v)\leq W_Z$.
8. $\sum_{r\geq 0} c(\partial B(z,r))\leq W_Z$.

Stage Two of the Construction

In the second stage of the construction, we examine a sequence of regions and use the cuts defined by them to get a low quotient cost. At the same time, we also produce a region of small radius which contains roughly half of the vertices.

Since the second stage involves removing regions from the graph, we extend our notation for regions to explicitly specify the underlying graph. We shall let H be a working copy of the graph. In fact H will always be a vertex-induced subgraph of the input graph $G=(V,E)$, but may be disconnected. Note that G

remains *fixed* throughout the construction. We denote by $B_H(z,r)$, $\partial B_H(z,r)$ and $W_H(z,r)$ respectively the region in H with center z and radius r, its boundary and its weight. As usual, we let n be the number of vertices in G.

Stage 2 begins with a crude quotient cost, which is repeatedly refined by calling a subroutine FINDHEAVYREGION that either returns a lower quotient cost or produces a region that contains roughly half of the vertices of the input graph. Eventually the desired heavy region is found.

STAGETWO(G, c, $l^{\rightarrow}$)

$\alpha_{min2} \leftarrow 2\min_v c(v)$
$(\bar{z}, \bar{r}) \leftarrow$ UNDEFINED
while $(\bar{z}, \bar{r})$ is UNDEFINED
    $(\alpha_{new}, (\bar{z}, \bar{r})) \leftarrow$ FINDHEAVYREGION$(G, l^{\rightarrow}, \alpha_{min2})$
    if α_{new} is not UNDEFINED **then** $\alpha_{min2} \leftarrow \alpha_{new}$
endwhile

The subroutine FINDHEAVYREGION attempts to lower the given quotient cost $\bar{\alpha}$ by examining a sequence of regions and checking if there is a region containing half the vertices whose weight expand sufficiently quickly, at a rate depending on $\bar{\alpha}$. The absence of such a heavy region indicates that the expansion factor is too large, and the collection of the regions examined will in turn produce a cut with a quotient cost smaller than $\frac{1}{2}\bar{\alpha}$. Note that the set L, which contains the center-radius pairs of the regions examined, need not be maintained by the subroutine, but is defined for use in the proof of correctness. Also the subroutine returns a pair consisting of a quotient cost and a center-radius pair, one of which is undefined. The regions constructed in the subroutine are defined with respect to the current graph H.

FINDHEAVYREGION(G, $l^{\rightarrow}$, $\bar{\alpha}$)

1. $H \leftarrow G$, $R \leftarrow \emptyset$, $\partial R \leftarrow \emptyset$, $L \leftarrow \emptyset$
2. Select some vertex z in the current graph, and let $r \leftarrow 0$
3. (*Optional step*) Let $\alpha^S = \dfrac{c(\partial S)}{\min\{|S| + \frac{1}{2}|\partial S|, n - |S| - \frac{1}{2}|\partial S|\}}$
   where $S = R \cup B_H(z,r)$ and $\partial S = \partial R \cup \partial B_H(z,r)$
   if $\alpha^S < \frac{1}{2}\bar{\alpha}$, **then return** $(\alpha^S$, UNDEFINED$)$
4. **if** $W_H(z, r+1) \geq (1+\epsilon)W_H(z,r)$ where $\epsilon = \frac{\bar{\alpha} n s}{33}$ **then**
       $r \leftarrow r+1$
       goto step 3
   endif
5. **if** $|B_H(z,r)| + \frac{1}{2}|\partial B_H(z,r)| \geq n/2$, **then return** (UNDEFINED, (z,r))
6. Update R, ∂R and L: $R \leftarrow R \cup B_H(z,r)$, $\partial R \leftarrow \partial R \cup \partial B_H(z,r)$,
   $L \leftarrow L \cup \{(z,r)\}$

Let $\alpha^R = \dfrac{c(\partial R)}{\min\{|R| + \frac{1}{2}|\partial R|, n - |R| - \frac{1}{2}|\partial R|\}}$
if $\alpha^R < \frac{1}{2}\bar{\alpha}$ **then**
    return (α^R, UNDEFINED)
else
    remove the vertices in $B_H(z, r) \cup \partial B_H(z, r)$ from the graph H
    goto step 2
endif

Note that we also implicitly keep track of the cut that gives the final quotient cost α_{min2}.

Stage Three of the Construction

In Stage 3, we grow the region $B_G(\bar{z}, r)$ one unit distance at a time *with respect to the input graph*, and examine the cuts given by the boundary of the region. Note that $B_G(\bar{z}, r)$ contains $B_H(\bar{z}, r)$.

StageThree(G, $\bar{z}$)

$\alpha_{min3} \leftarrow \infty$, $r \leftarrow 0$
while $B_G(\bar{z}, r) \neq V$
    $\alpha_B \leftarrow \dfrac{c(\partial B_G(\bar{z}, r))}{\min\{|B_G(\bar{z}, r)| + \frac{1}{2}|\partial B_G(\bar{z}, r)|, n - |B_G(\bar{z}, r)| - \frac{1}{2}|\partial B_G(\bar{z}, r)|\}}$
    if $\alpha_{min3} < \alpha_B$ **then** $\alpha_{min3} \leftarrow \alpha_B$
    $r \leftarrow r + 1$
endwhile

Let α_{min3} be the minimum quotient cost of the cuts observed at this stage, and let $\alpha_{obs} = \min(\alpha_{min2}, \alpha_{min3})$. We will show in the next section that α_{obs} is within an $O(\log n)$ factor of the minimum quotient cost. The output of the Leighton-Rao construction is the cut $(A_{obs}, B_{obs}, C_{obs})$ that achieved the quotient cost α_{obs}.

3.2.2 Analysis of the Leighton-Rao Construction

We now analyze the Leighton-Rao construction and show that the produced cut $(A_{obs}, B_{obs}, C_{obs})$ has a quotient cost that is indeed within an $O(\log n)$ factor of the minimum quotient cost, in the case where the associated concurrent flow had a commodity for each pair of vertices or where the demand graph was a constant-degree expander.

Proof of Correctness

Note that the partitions $(S, V \setminus (S \cup \partial S), \partial S)$ and $(R, V \setminus (R \cup \partial R), \partial R)$ in steps 3 and 6 respectively of the subroutine FindHeavyRegion are collections of regions defined on *different* underlying graphs. Inituitively, since a later region is formed from the graph remaining when the earlier regions and their boundaries have been removed, the boundaries of the regions together form a cut of G. We show this result more carefully below.

Lemma 3.2 *The partitions* $(S, V \setminus (S \cup \partial S), \partial S)$ *and* $(R, V \setminus (R \cup \partial R), \partial R)$ *are vertex cuts of* G.

Proof. We show inductively on the number of regions removed that $(R, V \setminus (R \cup \partial R), \partial R)$ is a vertex cut of G. The correctness of $(S, V \setminus (S \cup \partial S), \partial S)$ can be established in an identical fashion.

Consider the first time when step 6 is reached. At that point, the current graph H is the input graph G, and is therefore connected. Since FINDHEAVYREGION proceeded past step 5, we know that $B_H(z, r)$ is a proper subset of G, and therefore that $\partial B_H(z, r) \neq \emptyset$. Furthermore, by definition, all neighbors of vertices in $B_H(z, r)$ are either in $B_H(z, r)$ or $\partial B_H(z, r)$. After the update operation in step 6, $R = B_H(z, r)$ and $\partial R = \partial B_H(z, r)$, and thus $(R, V \setminus (R \cup \partial R), \partial R)$ is a vertex cut of G.

Now consider inductively the $(b+1)$st time that step 6 is reached, where $b \geq 1$. Let R_{old} and ∂R_{old} be the values of R and ∂R before the update operation. We want to show that

$$(R_{old} \cup B_H(z, r),\ V \setminus (R_{old} \cup \partial R_{old} \cup B_H(z, r) \cup \partial B_H(z, r)),\ \partial R_{old} \cup \partial B_H(z, r))$$

is a vertex cut, or equivalently, that there are no edges in G between (vertices in) $R_{old} \cup B_H(z, r)$ and (vertices in) $V \setminus (R_{old} \cup \partial R_{old} \cup B_H(z, r) \cup \partial B_H(z, r))$. By induction, $(R_{old}, V \setminus (R_{old} \cup \partial R_{old}), \partial R_{old})$ is a vertex cut of G. So there are no edges in G between R_{old} and $V \setminus (R_{old} \cup \partial R_{old})$, or more restrictively, between R_{old} and $V \setminus (R_{old} \cup \partial R_{old} \cup B_H(z, r) \cup \partial B_H(z, r))$. Observe that the current graph H is induced by the vertices in $V \setminus (R_{old} \cup \partial R_{old})$ on G. By definition, the neighbors *in* H of the vertices in $B_H(z, r)$ are in $B_H(z, r) \cup \partial B_H(z, r)$. Thus the neighbors in G of the vertices in $B_H(z, r)$ are in $R_{old} \cup \partial R_{old} \cup B_H(z, r) \cup \partial B_H(z, r)$. Therefore, there are no edges in G between $R_{old} \cup B_H(z, r)$ and $V \setminus (R_{old} \cup \partial R_{old} \cup B_H(z, r) \cup \partial B_H(z, r))$, as desired. ■

We next show that each invocation of the subroutine FINDHEAVYREGION in Stage 2 actually terminates, and hence Stage 2 halts.

Lemma 3.3 *Stage 2 halts.*

Proof. Let $c_{min} = \min_v c(v) > 0$. We initialized α_{min2} to $2c_{min}$, which is the minimum quotient cost of a cut $(V \setminus \{v\}, \emptyset, \{v\})$. By the definition of quotient cost, we know that $\alpha_{min2} \geq \frac{c_{min}}{n/2} = \frac{2c_{min}}{n}$. So α_{min2} can only be changed $\log n$ times and there are at most $1 + \log n$ invocations of FINDHEAVYREGION.[2] It therefore suffices to show that each invocation of FINDHEAVYREGION terminates.

[2]Suppose in the subroutine FINDHEAVYREGION, we replace the checks $\alpha^S \leq \frac{1}{2}\bar{\alpha}$ and $\alpha^R \leq \frac{1}{2}\bar{\alpha}$ respectively by $\alpha^S < \bar{\alpha}$ and $\alpha^R < \bar{\alpha}$. Then FINDHEAVYREGION returns whenever a lower quotient cost is found. However, there are a finite number of different quotient costs, and so FINDHEAVYREGION can only be invoked finitely many times, and Stage 2 will still terminate.

Consider an invocation of FindHeavyRegion. Note that for a center z, steps 3 and 4 can only be repeated a finite number of times since eventually $B_H(z,r) = H$ and the weight of the region no longer increases. Observe also that as long as the current graph H is not empty, we have that $B_H(z,r) \cup \partial B_H(z,r)$ is always not empty. Thus each execution of step 6 increases $|R| + \frac{1}{2}|\partial R|$ by at least $\frac{1}{2}$. We now show that $|R| + \frac{1}{2}|\partial R|$ cannot exceed $n/4$ and consequently H never becomes empty. Therefore step 6 is executed finitely many times, and FindHeavyRegion terminates.

Assume for the sake of contradiction that there is a first instance when $|R| + \frac{1}{2}|\partial R| \geq n/4$ *after* the update operation in step 6. Since $|B_H(z,r)| + \frac{1}{2}|\partial B_H(z,r)| < n/2$, we have $n/4 \leq |R| + \frac{1}{2}|\partial R| < 3n/4$, and so $n - |R| - \frac{1}{2}|\partial R| > n/4$. From step 6,

$$\frac{c(\partial R)}{\min\{|R| + \frac{1}{2}|\partial R|, n - |R| - \frac{1}{2}|\partial R|\}} \geq \tfrac{1}{2}\bar{\alpha}$$

which gives

$$c(\partial R) \geq \tfrac{1}{2}\bar{\alpha}\frac{n}{4} = \frac{n\bar{\alpha}}{8}.$$

By step 4, for each $(z,r) \in L$, $c(\partial B_H(z,r)) < \epsilon W_H(z,r)$, and so

$$c(\partial R) < \epsilon \sum_{(z,r)\in L} W_H(z,r) < \epsilon \sum_{(z,r)\in L} \;\sum_{v \in B_H(z,r)\cup\partial B_H(z,r)} 2c(v)l^{\rightarrow}(v) < \epsilon W_G,$$

which yields the desired contradiction

$$W_G > \frac{c(\partial R)}{\epsilon} \geq \frac{n\bar{\alpha}/8}{\bar{\alpha}ns/33} > \frac{4}{s}.$$

Therefore Stage 2 halts. ■

Bounds on Source-Sink Distances

We first show that the rapid expansion of the region $B_H(\bar{z},\bar{r})$ output from Stage 2 implies that its radius is "small".

Lemma 3.4 $\bar{r} = O(\log n/(\alpha_{min2}ns))$.

Proof. Observe that the region $B_H(\bar{z},\bar{r})$ is found during the last invocation of FindHeavyRegion, and that the parameter ϵ in that invocation is equal to $\alpha_{min2}ns/33$ for the final value of α_{min2}. Consider that last invocation, and let $c_{min} = \min_v c(v)$. Note that $\bar{r} \geq 1$ and that $\alpha_{min2} \leq 2c_{min}$ by the initialization step in Stage 2. By repeated execution of step 4, we know that

$$W_H(\bar{z},\bar{r}) \geq (1+\epsilon)^{\bar{r}-1} W_H(z,1) \geq (1+\epsilon)^{\bar{r}-1} c_{min}.$$

Thus $(1+\epsilon)^{\bar{r}-1} c_{min} \leq W_G \leq \frac{4}{s} = 4\sum_v c(v)$ and

$$\bar{r} = O\left(\frac{\log(\sum_v c(v)/c_{min})}{\log(1+\epsilon)}\right).$$

By assumption (cf. Section 3.1.3), $c(v) \le n(n-1)c_{min} \;\; \forall v$. So $\log(\sum_v c(v)/c_{min}) = O(\log n)$. In addition,

$$\epsilon = \frac{n\alpha_{min2}s}{33} = \frac{n\alpha_{min2}}{33\sum_v c(v)} \le \frac{n \cdot 2c_{min}}{33\sum_v c(v)} \le \frac{2}{33} < 1,$$

and we have $\log(1+\epsilon) = \Theta(\epsilon)$. ■

We now bound the sum of directed distances in G from the vertex $\bar{z}$ to all vertices.

Lemma 3.5 $\sum_v dist^{\rightarrow}(\bar{z}, v) = O(\frac{\log n}{s\alpha_{obs}})$.

Proof. Let $r_G = \min\{r : B_G(\bar{z}, r) = V\}$ and $r_{1/2} = \min\{r : |B_G(\bar{z}, r)| + \frac{1}{2}|\partial B_G(z, r)| \ge n/2\}$. Note that $r_{1/2} \le \bar{r}$. For $r_{1/2} \le r < r_G$, we have that

$$B_G(\bar{z}, r_{1/2}) \subseteq B_G(\bar{z}, r) \quad \text{and} \quad B_G(\bar{z}, r_{1/2}) \cup \partial B_G(\bar{z}, r_{1/2}) \subseteq B_G(\bar{z}, r) \cup \partial B_G(\bar{z}, r),$$

and so

$$n/2 \le |B_G(\bar{z}, r_{1/2})| + \tfrac{1}{2}|\partial B_G(\bar{z}, r_{1/2})| \le |B_G(\bar{z}, r)| + \tfrac{1}{2}|\partial B_G(\bar{z}, r)|.$$

So the minimum quotient cost α_{min3} observed in Stage 3 satisfies:

$$\begin{aligned}
\alpha_{min3} &\le \min_{r_{1/2} \le r < r_G} \frac{c(\partial B_G(\bar{z}, r))}{\min\{|B_G(\bar{z}, r)| + \frac{1}{2}|\partial B_G(\bar{z}, r)|, n - |B_G(\bar{z}, r)| - \frac{1}{2}|\partial B_G(\bar{z}, r)|\}} \\
&= \min_{r_{1/2} \le r < r_G} \frac{c(\partial B_G(\bar{z}, r))}{n - |B_G(\bar{z}, r)| - \frac{1}{2}|\partial B_G(\bar{z}, r)|}
\end{aligned}$$

For $\bar{r} \le r < r_G$, let $n_r = |V \setminus B_G(\bar{z}, r)| = n - |B_G(\bar{z}, r)|$. Then

$$\begin{aligned}
c(\partial B_G(\bar{z}, r)) &\ge \alpha_{min3}(n - |B_G(\bar{z}, r)| - \tfrac{1}{2}|\partial B_G(\bar{z}, r)|) \\
&\ge \alpha_{min3}(n - |B_G(\bar{z}, r)| - \tfrac{1}{2}(n - |B_G(\bar{z}, r)|)) \\
&= \tfrac{1}{2}\alpha_{min3} n_r.
\end{aligned}$$

Thus

$$\begin{aligned}
\sum_v dist^{\rightarrow}(\bar{z}, v) &\le n r_{1/2} + \sum_{r_{1/2} \le r < r_G} n_r \\
&\le n\bar{r} + \frac{2}{\alpha_{min3}} \sum_{r_{1/2} \le r < r_G} c(\partial B_G(\bar{z}, r)) \\
&\le O\left(\frac{\log n}{s\alpha_{min2}}\right) + \frac{2}{\alpha_{min3}} W_G \\
&= O\left(\frac{\log n}{s\alpha_{obs}}\right),
\end{aligned}$$

since $\alpha_{obs} = \min(\alpha_{min2}, \alpha_{min3})$ and $W_G \leq 4/s$. ■

The following lemma bounds the sum of shortest path distances (using the dual length function) for the commodities.

Lemma 3.6 *Suppose there are at most d terminals at each vertex. Then*

$$\sum_{i=1}^{k} dist(s_i, t_i) = O(d \log n/\alpha_{obs}).$$

Proof. For each source s_i and sink t_i, we can route a path between them using $\bar{z}$ as an intermediate point. So

$$\begin{aligned}\sum_{i=1}^{k} dist(s_i, t_i) &= \frac{s}{2}\sum_{i=1}^{k}(dist^{\rightarrow}(\bar{z}, s_i) + dist^{\rightarrow}(\bar{z}, t_i)) \\ &\leq \frac{sd}{2}\sum_{v} dist^{\rightarrow}(\bar{z}, v) \\ &= O\left(\frac{d \log n}{\alpha_{obs}}\right).\end{aligned}$$

■

In the case where the expected number of terminals at each vertex is d, Lemma 3.6 holds for the expected sum of shortest path distances for the commodities. Note that Lemmata 3.4, 3.5 and 3.6 do not rely on the optimality of the dual solution, nor on the number and location of the source-sink pairs.

Performance Guarantees

Finally, we prove that the cut produced by the Leighton-Rao construction has a quotient cost that is within an $O(\log n)$ factor of the minimum quotient cost, for two different classes of commodities.

Theorem 3.7 *Suppose the vertex lengths used in the construction were obtained from the dual to an all-pairs concurrent flow problem. Then $(A_{obs}, B_{obs}, C_{obs})$ is an $O(\log n)$-approximate flux cut, i.e. $q(A_{obs}, B_{obs}, C_{obs}) = O(\log n)\alpha$.*

Proof. Since there are $n-1$ terminals at each vertex, we have $\sum_{i=1}^{k} dist(s_i, t_i) = O(n \log n/\alpha_{obs})$ by Lemma 3.6. Recall from Section 3.1.3 that for this case, the optimal objective value D^* of the dual satisfies $D^* \geq \frac{1}{\mathcal{S}}$, and so the objective value $\tilde{D}$ of an ϵ-optimal dual solution satisifies $\tilde{D} = \Omega(1/\mathcal{S})$. Thus $\frac{n \log n}{\alpha_{obs}} = \Omega(\frac{1}{\mathcal{S}})$ and $\alpha_{obs} = O(\mathcal{S} n \log n) = O(\log n)\alpha$. ■

Theorem 3.8 *Suppose the vertex lengths used in the construction were obtained from the dual to a concurrent flow problem where the demand graph is an expander with constant degree d. Then* $(A_{obs}, B_{obs}, C_{obs})$ *is an* $O(\log n)$*-approximate flux cut, i.e.* $q(A_{obs}, B_{obs}, C_{obs}) = O(\log n)\alpha$.

Proof. Since d is $O(1)$, we have $\sum_{i=1}^{k} dist(s_i, t_i) = O(\log n/\alpha_{obs})$ by Lemma 3.6. Recall from Section 3.1.3 that for this case, the optimal objective value D^* of the dual satisfies $D^* = \Omega(1/\alpha)$, and so the objective value $\tilde{D}$ of an ϵ-optimal dual solution satisfies $\tilde{D} = \Omega(1/\alpha)$. Thus $\frac{\log n}{\alpha_{obs}} = \Omega(1/\alpha)$ and $\alpha_{obs} = O(\log n)\alpha$. ■

Recall that in our construction of expander graphs, the degree of each vertex is potentially unbounded, but that the expected degree of each vertex is constant. Theorem 3.8 is still applicable, and states that the *expected* quotient cost of the produced cut is within an $O(\log n)$ factor of the minimum.

3.2.3 Implementation and Analysis of Running Time

We briefly describe how the cut construction procedure can be implemented and analyze its running time. The main concern lies in efficiently updating the sets $B(z,r)$ and $\partial B(z,r)$ as the radius r increases. The key observation is that when the radius increases from a value, say r_0, to r_0+1, the sets $B(z,r)$ and $\partial B(z,r)$ change if and only if there is a vertex v whose directed distance $dist^{\rightarrow}(z,v)$ is r_0+1. In fact such a vertex v would have been in $\partial B(z,r_0)$, and v leaves $\partial B(z,r_0)$ and enters $B(z,r_0+1)$. At the same time, the neighbors of v which are not in $B(z,r_0+1)$ then enter $\partial B(z,r_0+1)$. More precisely, we update $B(z,r)$ and $\partial B(z,r)$ as follows:

<u>UPDATEREGION</u>

$U \leftarrow \{v \in \partial B(z,r_0) : dist^{\rightarrow}(z,v) = r_0+1\}$
if $U = \emptyset$ **then**
    $B(z,r_0+1) \leftarrow B(z,r_0)$, $\partial B(z,r_0+1) \leftarrow \partial B(z,r_0)$
else
    $B(z,r_0+1) \leftarrow B(z,r_0) \cup U$, $\partial B(z,r_0+1) \leftarrow \partial B(z,r_0) \setminus U$
    for each $v \in U$
        $\partial B(z,r_0+1) \leftarrow \partial B(z,r_0+1) \cup \{w : vw \in E \text{ and } w \notin B(z,r_0+1)\}$
    endfor
endif

It would seem that for a center z, the updating process could potentially be performed r_{max} times, where r_{max} is the maximum distance of a vertex from z. From the assumption $c(v) \leq n(n-1)c_{min}$ relating the vertex capacities to the minimum capacity c_{min}, we know that the sum of vertex capacities C is at most $n^2(n-1)c_{min}$. Using that bound, the dual LP relation $\sum_v c(v)l(v) = 1$, and the definition of the lengths $l^{\rightarrow}(v)$ in terms of the dual lengths $l(v)$, we can upper bound the sum of the discretized vertex lengths:

$$\sum_v l^{\rightarrow}(v) \leq \sum_v (1 + Cl(v)) = n + \sum_v \frac{C}{c(v)} c(v)l(v) \leq n + n^2(n-1) \sum_v c(v)l(v) \leq n^3.$$

So the maximum directed distance r_{max} is at most $2n^3$, and the updating process has to be potentially performed up to $2n^3$ times. However, we see from the updating procedure that the sets $B(z, r)$ and $\partial B(z, r)$ only change when a vertex is reached as the radius increases. Therefore instead of iterating over the possible radii in single unit increments, we only consider in increasing order radii that are equal to the directed distance of some vertex from z. Restricting the radii that we examine is not incompatible with the termination check

$$W(z, r+1) \geq (1+\epsilon)W(z, r)$$

on the incremental weight of a region in step 4 of FINDHEAVYREGION. By definition, $W(z, r+1) = W(z, r) + \partial B(z, r)$, and so the check compares $\partial B(z, r)$ against $\epsilon W(z, r)$. For two successive radii $r_1 < r_2$ equal to the directed distances of some vertices, $\partial B(z, r)$ remains the same for $r = r_1, ..., r_2 - 1$, while $W(z, r)$ grows linearly. Therefore the check $\partial B(z, r) \geq \epsilon W(z, r)$ fails for some $r = r_1, ..., r_2 - 1$ if and only if it fails at $r = r_2 - 1$, and the check need only be performed at specific radii rather than over all possible values of the radius.

With the restriction of the radii to actual directed distances, the region-growing process from a center z closely resembles a shortest path computation from z. In fact, the set $B(z, r)$ is just the set of vertices that has been reached from z, while the set $\partial B(z, r)$ is the vertices that are in the heap.[3] The updating phase corresponds to removing the smallest vertex from the heap and using it to extend the frontier of the shortest path computation. Care though has to be taken to ensure that if there are multiple minimum-valued vertices in the heap, then all these vertices are simultaneously removed from the heap and the frontier appropriately extended. We conclude then that the process of growing and processing regions from a single center can be implemented in time $O(m + n \log n)$, the asymptotic time for a single-source shortest path computation.

To determine the overall running time of the Leighton-Rao cut construction procedure, we have to determine how much time is spent on computation associated with each value of α_{min2} during Stage 2. We saw in the proof of correctness that there are at most $1 + \log n$ values of α_{min2} that are used during the algorithm. For each α_{min2} value, we remove a sequence of regions from the graph until a sufficiently large region is found or a much better quotient cost is encountered, potentially growing regions from all vertices. However, the process of growing and subsequently removing a region at a center z is in effect a truncated shortest path computation, which takes only time $O(m^* + n^* \log n)$ where m^* and n^* are the number of edges and vertices that are actually seen during that truncated

[3] Recall from Section 2.3.5 that the vertex-based path lengths allow us to conclude that when a vertex v is first reached from z during the shortest path computation, the path from z to v in fact is a shortest path from z to v.

computation. In the sequence of truncated shortest path computations from various centers, later computations do not encounter the vertices and edges processed by earlier computations since those entities have already been removed from the graph. Therefore, the time taken for the growing and removing a sequence of regions is at most that of *one* complete shortest path computation, and the time taken by Stage 2 is $O((m + n \log n) \log n)$. Since Stage 3 only involves growing regions from a single center, its time is dominated by that of Stage 2, and the Leighton-Rao procedure, excluding the time spent solving the concurrent flow, runs in time $O((m + n \log n) \log n)$.

For the concurrent flow problem instances that the Leighton-Rao procedure generates, the number of commodities is a small constant multiple of the number of vertices. From Theorem 2.7 in Chapter 2, the time taken to solve the concurrent flow is $O(m^2 n \log n)$, which dominates the time spent by the actual cut construction procedure itself. The bottleneck in the Leighton-Rao procedure is therefore in the determination of appropriate vertex lengths via the solution of concurrent flows.

3.3 Alternative Sparse Cut Construction

The Leighton-Rao sparse cut procedure constructs an approximate flux cut that is made up of a collection of one or more regions (either the cut $(S, V \setminus (S \cup \partial S), \partial S)$ from step 3 or the cut $(R, V \setminus (R \cup \partial R), \partial R)$ from step 6 of the subroutine FindHeavyRegion, or $(B(\bar{z}, r), V \setminus (B(\bar{z}, r) \cup \partial B(\bar{z}, r)), \partial B(\bar{z}, r))$ for some r in Stage 3). However, computational experience with the procedure shows that the constructed cut almost always consists of a single region, and this observation allows us to simplify the cut construction. Unfortunately, we are unable to prove that the resulting procedure finds a cut that is approximately optimal, but instead rely on an a posteriori check to confirm the approximate optimality.

The cuts encountered in the Leighton-Rao construction are formed by randomly selecting vertices and growing regions of increasing radius from those vertices. The selection and growing process is simplified by examining all possible regions and keeping the best cut defined by the boundary of a region.

DeterminsticLeightonRao(G, $l^{\rightarrow}$)

initialize $\bar{z} \leftarrow$ UNDEFINED, $\bar{r} \leftarrow$ UNDEFINED, $\alpha_{min} \leftarrow \infty$
for each $z \in V$
  $r \leftarrow 0$
  while $(B(z, r) \cup \partial B(z, r) \neq V)$
    let $\alpha_B = q(B(z, r), V \setminus (B(z, r) \cup \partial B(z, r)), \partial B(z, r))$
    if $\alpha_B < \alpha_{min}$ **then**
      $\bar{z} \leftarrow z$, $\bar{r} \leftarrow r$, $\alpha_{min} \leftarrow \alpha_B$
    endif
    $r \leftarrow r + 1$
  endwhile
endfor

The deterministic version of the Leighton-Rao procedure essentially runs Stage 3 over all vertices in the graph, keeping track of the best quotient cut defined by a region and its boundary. As we saw earlier, with an appropriate implementation, Stage 3 can be made to run in time $O(m + n \log n)$, and so the determinstic version spends $O(n(m + n \log n))$ time in its construction procedure. Therefore the bottleneck in the deterministic construction is still in the approximate solution of the dual to the related concurrent flow, which takes $O(m^2 n \log n)$ time, as is the case with the randomized construction.

Efforts to prove that the deterministic version finds a cut that is indeed within a $O(\log n)$ factor of optimal were unsuccessful. The main obstacle lies in the possibility of the Leighton-Rao procedure finding an approximate cut defined by a union of two or more regions. It is difficult to compare the quotient cost of such a multi-region cut against the best single-region cut, as we shall see next. Consider, for example, the case where the cut produced is $(B_G \cup B_H, V \setminus (B_G \cup \partial B_G \cup B_H \cup \partial B_H), \partial B_G \cup \partial B_H)$. Here B_G refers to the first region removed in step 6 of the subroutine FINDHEAVYREGION, while B_H is the second region, and is defined with respect to the subgraph H induced on G by the vertices $V \setminus (B_G \cup \partial B_G)$. The boundaries of B_G and B_H are respectively denoted by ∂B_G and ∂B_H. Assume for simplicity that $|B_G| + |B_H| + |\partial B_G| + |\partial B_H| \leq n/2$. Then the quotient costs in G of the cuts $(B_G \cup B_H, V \setminus (B_G \cup \partial B_G \cup B_H \cup \partial B_H), \partial B_G \cup \partial B_H)$ and $(B_G, V \setminus (B_G \cup \partial B_G), \partial B_G)$ are respectively $c(\partial B_G \cup \partial B_H)/(|B_G| + \frac{1}{2}|\partial B_G| + |B_H| + \frac{1}{2}|\partial B_H|)$ and $c(\partial B_G)/(|B_G| + \frac{1}{2}|\partial B_G|)$, while the quotient cost *in* H of the cut $(B_H, (V \setminus (B_G \cup \partial B_G)) \setminus (B_H \cup \partial B_H), \partial B_H)$ is $c(\partial B_H)/(|B_H| + \frac{1}{2}|\partial B_H|)$. Using the inequality

$$\frac{a+b}{c+d} \geq \min\left(\frac{a}{c}, \frac{b}{d}\right) \quad \text{for } a, b, c, d > 0,$$

we observe that the quotient cost of the multi-region cut cannot be smaller than both the quotient costs of the single-region cuts. Unfortunately, the cut $(B_H, (V \setminus (B_G \cup \partial B_G)) \setminus (B_H \cup \partial B_H), \partial B_H)$ in H *does not* correspond to any cut in G. So we cannot in general relate the quotient cost of a multi-region cut in G to the quotient cost of a single-region cut in G.

However, we were also unable to construct a family of graphs where the deterministic construction produces a cut that is worse than an $O(\log n)$ factor of optimal. The constructed deterministic cut is instead validated by performing an a posteriori check to show that it is no worse (except possibly by a constant factor) than that from the randomized Leighton-Rao procedure. Recall that the proof of the performance guarantee relies on the existence of a region $B(\bar{z}, \bar{r})$ in Stage 2 containing at least half the vertices, or more precisely, $|B(\bar{z}, \bar{r})| + \frac{1}{2}|\partial B(\bar{z}, \bar{r})| \geq n/2$, whose weight $W(\bar{z}, \bar{r})$ is at least $(1+\epsilon)^{\bar{r}-1}$. Here $\epsilon = (\alpha_{min2} ns)/33$ depends on the smallest quotient cost α_{min2} observed during Stage 2. We can test for the presence of such a region in the deterministic construction by recording the maximum value of the expression $\log(W(z, r))/(r - 1)$ over all regions with $|B(z, r)| + \frac{1}{2}|\partial B(z, r)| \geq n/2$, and subsequently checking that the maximum value is at least $\log(1 + \alpha_{min} ns/33)$

for the smallest quotient cost α_{min} observed. It turns out that for all the runs that we performed, the a posteriori check is always satisfied, and so the determinstic version of the Leighton-Rao procedure is a practical alternative to the original randomized version.

3.4 Experimental Results for the Leighton-Rao Sparse Cut Construction

We present in this section experimental results for the Leighton-Rao sparse cut construction. There are three related goals in our evaluation of the construction. The first is to compare the general version, described in Section 3.2, to the simplified version that we proposed in Section 3.3. The second goal is to determine if there is any dependence between the quality of the cut produced by the construction and the type of concurrent flow solved to obtain the vertex length function necessary for the construction. Lastly the third goal is to compare the results of the Leighton-Rao construction against those of a simplistic construction and also against theoretically derived lower bounds.

We describe the setup for the experiments in Section 3.4.1, including the types of concurrent flows that we solved. In Section 3.4.2, we present some results for the general Leighton-Rao construction. We shall see in Section 3.4.3 that the simplified version is always superior for our experiments. In Section 3.4.4, we describe a simple but effective cut refinement procedure that is easily incorporated into the Leighton-Rao construction, and examine the performance of the refined simplified Leighton-Rao construction. In particular, we shall look more closely at how the different vertex lengths used affect the quality of the produced cuts, and decide on a choice of length functions for use in practice. We then focus on the quality of the sparse cuts produced, by comparing them to a simplistic construction in Section 3.4.5, and to the theoretical bounds obtained from the associated all-pairs concurrent flow problem in Section 3.4.6. Finally, we present some miscellaneous statistics for the various constructions in Section 3.4.7.

3.4.1 Experimental Setup

For our evaluation of the Leighton-Rao sparse cut construction, we use the graphs of a set of sparse matrices and focus exclusively on the case where the vertices have unit capacities. These restrictions are reasonable for they hold in the scenario in which the construction is indirectly used, viz. the elimination ordering problem of Chapter 4. In the subsequent presentation, we will not make a distinction between a matrix and its graph. The set of 33 matrices that we use is from the standard Harwell-Boeing collection of sparse matrices [18] [19], and is identical to that used in evaluating the concurrent flow implementation in Section 2.5. Table 3.1 gives the number of vertices and edges in the matrices. For information on the vertex degrees and hop distances of the matrices, we refer the reader to Table 2.10 in Section 2.5.

Table 3.1: Sizes of the matrices used in evaluating the Leighton-Rao sparse cut construction.

Matrix	n	m	Matrix	n	m	Matrix	n	m
CAN96	96	336	BCSSTM07	420	3416	NOS6	675	1290
CAN144	144	576	BCSPWR05	443	590	NOS7	729	1944
CAN187	187	652	BCSPWR06	1454	1923	LSHP265	265	744
CAN229	229	774	DWT193	193	1650	LSHP406	406	1155
CAN292	292	1124	DWT209	209	767	LSHP577	577	1656
CAN445	445	1682	DWT310	310	1069	LSHP778	778	2247
CAN715	715	2975	DWT419	419	1572	WILL199	199	660
BCSSTK04	132	1758	DWT592	592	2256	ASH292	292	958
BCSSTK05	153	1135	DWT878	878	3285	FS541.1	541	2466
BCSSTK06	420	3720	NOS4	100	247	SHL200	663	1720
BCSSTK19	817	3018	NOS5	468	2352	BP1000	822	4635

Types of Dual Length Functions

Recall that Stage 1 of the Leighton-Rao sparse cut construction (cf. Section 3.2.1) involves solving the dual of a unit-demand concurrent flow problem to obtain a suitable vertex length function. The dual length function is then discretized and used to define the regions explored by the construction. Ideally, one would like to use the dual solution to an all-pairs concurrent flow problem for the construction, for then we obtain the tightest performance guarantee on the quotient cost of the cut produced, as was shown in Theorem 3.7. However, an all-pairs concurrent flow problem is costly to solve,[4] and in practice, we prefer to solve a concurrent flow problem where the demand graph is a low constant-degree expander. The resulting dual length function also allows us to obtain the same asymptotic performance guarantees on the quotient cost (cf. Theorem 3.8), albeit with higher constants hidden in the notation.

We therefore used various classes of dual length functions in our experiments to determine if they have any effect on the quality of the cut produced. The classes of length functions differ on the type of concurrent flow problem used and the accuracy to which it was solved. Specifically, we generated the following five classes, for a total of 31 dual length functions for each test matrix:

(i) We generated ten demand graphs, each composed of three matchings (i.e. permutations) on the vertices. Thus each vertex is both the source and sink of three commodities. As we saw in Section 3.1.3, these demand graphs are very likely to be expanders. We solved each of the associated concurrent flow

[4] In this section, we shall not be careful in stating that we only require the solution to the dual problem. This is because the concurrent flow implementation from Chapter 2 delivers both an ϵ-optimal concurrent flow and an ϵ-optimal dual length function.

problem using an error tolerance of 10%. We refer to these 10 dual vertex lengths as *0.1-optimal 3-matching* length functions.

(ii) We solved five of the concurrent flow problems from (i) using an error tolerance of 1%, obtaining five 0.01-optimal 3-matching length functions. The reason for not solving all 10 concurrent problems using the stricter error tolerance is that it would take too long to do so.

(iii) We generated ten demand graphs each composed of ten matchings. All ten associated concurrent flow problems were solved with an error tolerance of 10% to give ten 0.1-optimal 10-matching length functions. Five of the concurrent flow problems were also solved with an error tolerance of 1% to give five 0.01-optimal 10-matching length functions.

(iv) Finally, we solved the all-pairs concurrent flow problem using an error tolerance of 5% to obtain a 0.05-optimal all-pairs length function. We did not use an error tolerance of 1% for it required a very long solution time.

Similarity of the Dual Length Functions

As we saw in Section 3.1.3, the optimal congestion of an all-pairs concurrent flow problem can be used to lower bound the expected optimal congestion of a concurrent flow problem on the same underlying graph but with a demand graph given by random matchings. The four matching-based classes of length functions above can therefore be regarded as approximations to the dual length function of an all-pairs concurrent flow problem. We examine the closeness of the approximation by looking for similarities between the various length functions generated for each matrix. We remind the reader that each length function is normalized such that the sum of the vertex lengths is 1.

Before looking at the length functions, we point out two interesting aspects about the achieved errors for the various classes. The first is that although the all-pairs concurrent flow problems were solved with an error tolerance of 5%, the achieved errors were only at most 1.23%. The second is the behavior of the matrix DWT878, which had an achieved error of at most 0.045% for all of the concurrent flows solved. It turns out that the graph for that matrix has one articulation point, i.e. a vertex whose removal disconnects the graph. Each of the 31 dual length functions found for DWT878 practically specified a cut consisting of that single articulation point. Indeed that vertex was assigned a length of at least 0.9993, while the other vertices had lengths under 10^{-6}. Since the length function that assigns nonzero length to the articulation point and zero lengths to all other vertices is an *optimal* dual solution for the generated concurrent flow problems on this graph, it follows that the length functions produced are very close to optimal. Curiously, although some of the other matrices have articulation points in their graphs, the achieved errors for them are not as low as that for DWT878.

In general, the length functions from the matching-based classes are not almost identical to that from the all-pairs problem, as was the case with the matrix

Table 3.2: Matrices with repeated discretized length functions $l^{\rightarrow}$ among the 31 generated.

Matrix	Number of Distinct $l^{\rightarrow}$	Frequencies of Repeated $l^{\rightarrow}$
DWT878	1	31
DWT193	17	6 ; 4 ; 4 ; 2 ; 2 ; 2
BCSSTK05	19	5 ; 5 ; 3 ; 3
CAN715	22	10
DWT592	22	9 ; 2
ASH292	23	9
DWT209	24	3 ; 3 ; 3 ; 2
CAN292	26	6
BCSPWR05	27	5
CAN144	30	2
DWT310	30	2
DWT419	30	2
NOS5	30	2

DWT878. We can compare a length function l_{match} from a matching-based class to the all-pairs length function l_{ap} by computing the 1-norm of their difference: $\|l_{match} - l_{ap}\|_1 = \sum_v |l_{match}(v) - l_{ap}(v)|$. Note that the 1-norm difference is at most 2. For some matrices and classes of length functions, the 1-norm difference from the all-pairs length function was small (under 0.2). However, for the majority of the instances, the difference was substantially larger, with numerous cases where it exceeded 1. A large difference should not be taken as an indication that a matching-based length function is a poor approximation to its all-pairs counterpart. The reason is that a concurrent flow problem and its dual can have *multiple* optimal solutions, which may be very different from each other in the lengths assigned to specific vertices.

Since the Leighton-Rao construction does not use the dual length function directly but instead discretizes it, we now consider the discretized length function $l^{\rightarrow}$. Given the preceding futility in ascertaining the closeness of the length functions, we content ourselves with looking for duplicates among the 31 discretized length functions for each matrix. It turns out that for 20 of the 33 test matrices, *all* 31 discretized length functions were different from each other. We call these matrices the *unique-length* matrices. As for the other 13 matrices, which we shall refer to as the *repeated-length* matrices, there were varying degrees of repetition among their discretized length functions. We list the repeated-length matrices in Table 3.2. For each matrix, we give the number of distinct discretized length functions that it has and the frequencies of those discretized length functions that are repeated. For example, in line with earlier observations, all the discretized length functions for the matrix DWT878 are identical. In contrast, the matrix BCSSTK05 has 19 distinct discretized length functions among its 31, of which four are repeated, with frequencies of 5, 5, 3 and 3.

Our above comparison of discretized length functions is crude in that we do not identify duplicates within each class of length functions, or multiple incidences of the discretized all-pairs length function. It turns out that such finer distinctions are unnecessary, as we shall see in the experimental results for the various versions of the Leighton-Rao construction.

3.4.2 Results for the General Leighton-Rao Construction

We ran the general Leighton-Rao construction on the test matrices using the various classes of length functions described in the previous section. Since our implementation randomly chooses vertices to grow regions from, we perform multiple runs for each length function as follows:

- 10 runs for each of the 10 length functions in the 0.1-optimal 3-matching class,
- 10 runs for each of the 10 length functions in the 0.1-optimal 10-matching class,
- 20 runs for each of the 5 length functions in the 0.01-optimal 3-matching class,
- 20 runs for each of the 5 length functions in the 0.01-optimal 10-matching class,
- and 100 runs for the all-pairs length function.

Therefore for each matrix, a total of 100 runs were performed for each class of length functions.

Comparison for the Classes of Length Functions

We first compare the average quotient costs of the cuts produced by the general Leighton-Rao construction in Figure 3.3. In that figure, we plot for each matrix and each class of length functions the average quotient cost of the cuts obtained for that class relative to the minimum quotient cost found for that matrix. The matrices in the figure are divided into two groups. Those to the left of the dotted vertical line (DWT878 through NOS5) are the repeated-length matrices and are listed in the same order as in Table 3.2, while those to the right are the unique-length matrices. We see from the figure that the average quotient cost for the various classes are typically within 30% of each other, and that there is no one class that has a significantly lower average quotient cost than the others. In addition, we observe that the average quotient costs for the repeated-length matrices are usually closer to the minimum quotient cost found, as compared to the situation for the unique-length matrices. This is to be expected since the runs on those matrices are on a smaller set of unique discretized length functions.

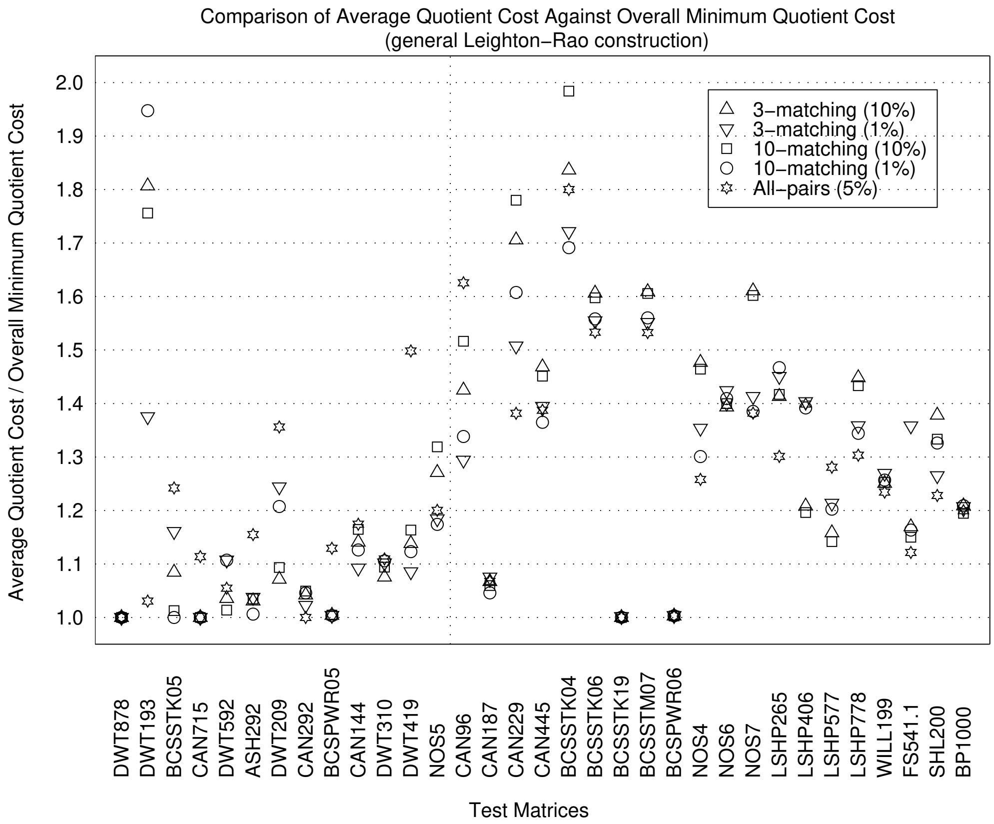


Figure 3.3: Comparison of the average quotient cost for each class of length functions to the minimum quotient cost over all classes, for the general Leighton-Rao construction.

Table 3.3: Types of cuts produced by the general Leighton-Rao construction.

Length Function Class	# Stage 2 Cuts		# Stage 3 Cuts
	1-Region	> 1-Region	
0.1-optimal 3-matching	802 (24%)	0	2498 (76%)
0.01-optimal 3-matching	904 (27%)	0	2396 (73%)
0.1-optimal 10-matching	774 (23%)	0	2526 (77%)
0.01-optimal 10-matching	908 (28%)	0	2392 (72%)
0.05-optimal All-pairs	832 (25%)	0	2468 (75%)

Classification of Cuts Found

We next classify the cuts that are produced. Recall from Section 3.2.1 that the produced cut is either found during Stage 2 or 3 of the construction. The cuts found in Stage 2 may be composed of multiple regions, while those found in Stage 3 are defined by a single region. Table 3.3 gives the number and percentage of single/multiple-region Stage 2 cuts and Stage 3 cuts produced. We observe that about three-quarters of the time, the cuts examined in Stage 3 improve upon those encountered in Stage 2. More strikingly, there were *no instances* of a multiple-region Stage 2 cut being the best observed cut among the 16,500 runs. This non-occurrence is rather surprising, and strongly indicates that at least for our set of test matrices, the full generality of the Leighton-Rao construction is not needed. Therefore, the simplified construction proposed in Section 3.3 will not be worse than the general construction, and can in fact produce cuts with significantly lower quotient costs, as we shall see later.

Other Statistics for the General Construction

We conclude our examination of the general Leighton-Rao construction by giving statistics on other aspects of its empirical behavior.

Variation in quotient cost. For each matrix and each length function used, we compute the minimum and maximum quotient costs that were found over the multiple random runs. It turns out that for some matrix-length function combinations, the maximum produced quotient cost is as large as 5.9 times the minimum cost, while for other combinations, the maximum quotient cost is within 10% of the minimum. The wide differences in quotient cost variation show that many runs of the construction for an input are necessary for producing a good quotient cost.

Stage 2 statistics. Recall that Stage 2 of the construction involves a sequence of calls to a subroutine which either halves the best quotient cost seen or finds a small-radius region containing half the vertices. Stage 2 terminates when the latter event occurs. In a typical run, the number of subroutine calls varied between 2 to

5, but could be as high as 8. The number of calls depends of course on how far the final quotient cost is from the fixed initial cost of 2, and in fact is strongly correlated to the quantity $\log(2/\alpha_{min})$, where α_{min} is the quotient cost found. In addition, the average number of centers examined in a subroutine call across a run is almost always 1 and never exceeds 1.5. This low average shows that multiple-region cuts are seldom examined in Stage 2, and lends further support to the observation that such cuts are unlikely to be the best cut observed in a run.

3.4.3 Results for the Simplified Leighton-Rao Construction

In this section, we present results for the simplified Leighton-Rao construction described in Section 3.3. This construction examines all cuts formed by a single region. We have seen in the previous section that this approach will be superior to the general construction for our set of test matrices. Since the implementation of the simplified construction does not involve randomization, we need only perform one run for each matrix and length function. Therefore, for each matrix there are 10 runs for the 0.1-optimal 3/10-matching classes, 5 runs for the 0.01-optimal 3/10-matching classes, and a single run for the all-pairs length function.

Comparison between the Simplified and General Constructions

Figure 3.4 compares the average quotient costs obtained by the simplified construction to those for the general construction. As before we have grouped the repeated-length matrices to the left of the dotted vertical line in the plot. We see that the simplified construction can be significantly better than the general construction. In fact there are cases where its average quotient cost is less than 0.6 times that of the general construction. Furthermore it improves on the general construction for all classes of length functions, with no class enjoying distinctly better improvements over all the matrices. In addition, the improvements seem to be more pronounced for the unique-length matrices, probably because the higher number of distinct discretized length functions presents the simplified construction with greater opportunities for finding better cuts. These results are not surprising since we know that for our set of test matrices, the general construction behaves as a randomized truncated version of the simplified construction, by examining regions grown from some randomly selected centers.

As for the minimum quotient cost found by the two constructions, it turns out that they differ only for 8 of the 33 matrices. These matrices and their minimum quotient costs and percentage improvements, are given in Table 3.4. We see that all 8 matrices have unique discretized length functions.

Comparison for the Classes of Length Functions

We next compare in Figure 3.5 the average quotient cost for each class of length functions to the minimum quotient cost found. Again we have separated out the

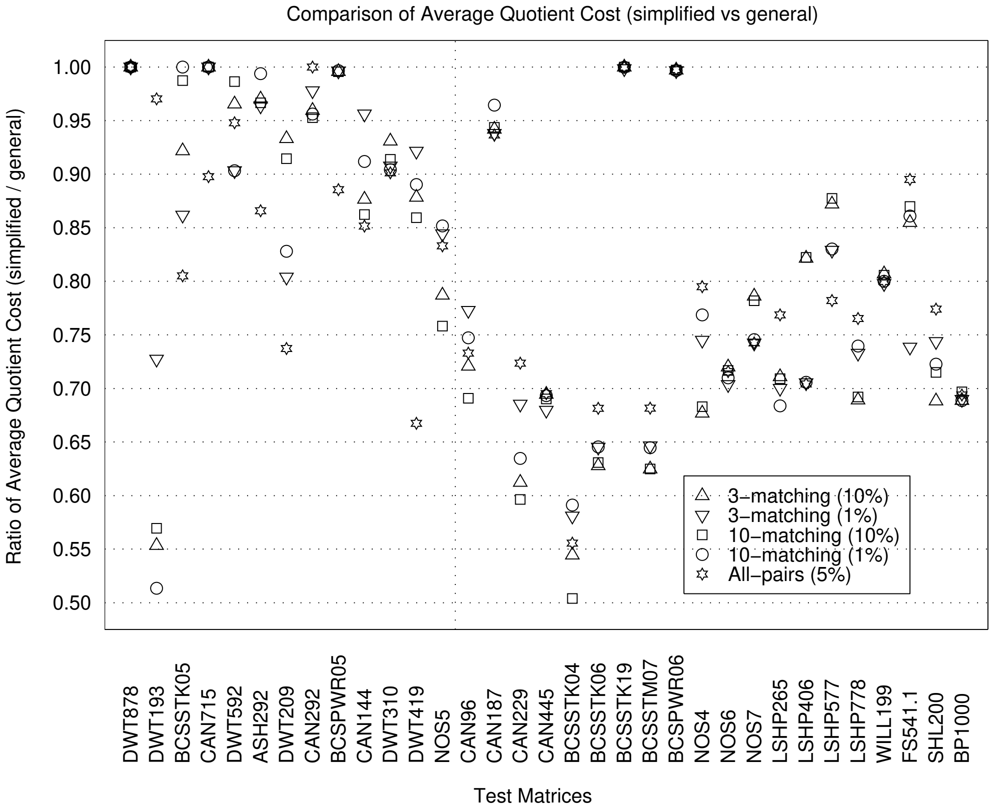


Figure 3.4: Comparison of the average quotient cost for each class of length functions for the simplified construction to those for the general construction.

Table 3.4: Matrices for which the minimum quotient cost found by the simplified and general Leighton-Rao constructions differed.

	Minimum Quotient Cost Found	
Matrix	General	Simplified
CAN445	0.2400	0.2079 (−13.4%)
LSHP265	0.1231	0.1212 (−1.5%)
LSHP406	0.1005	0.0985 (−2.0%)
LSHP577	0.0832	0.0826 (−0.7%)
LSHP778	0.0722	0.0706 (−2.2%)
WILL199	0.8449	0.8081 (−4.4%)
SHL200	0.5143	0.4762 (−7.4%)
BP1000	0.9610	0.8000 (−16.8%)

two types of matrices in the plot. We see that for each matrix, the average quotient

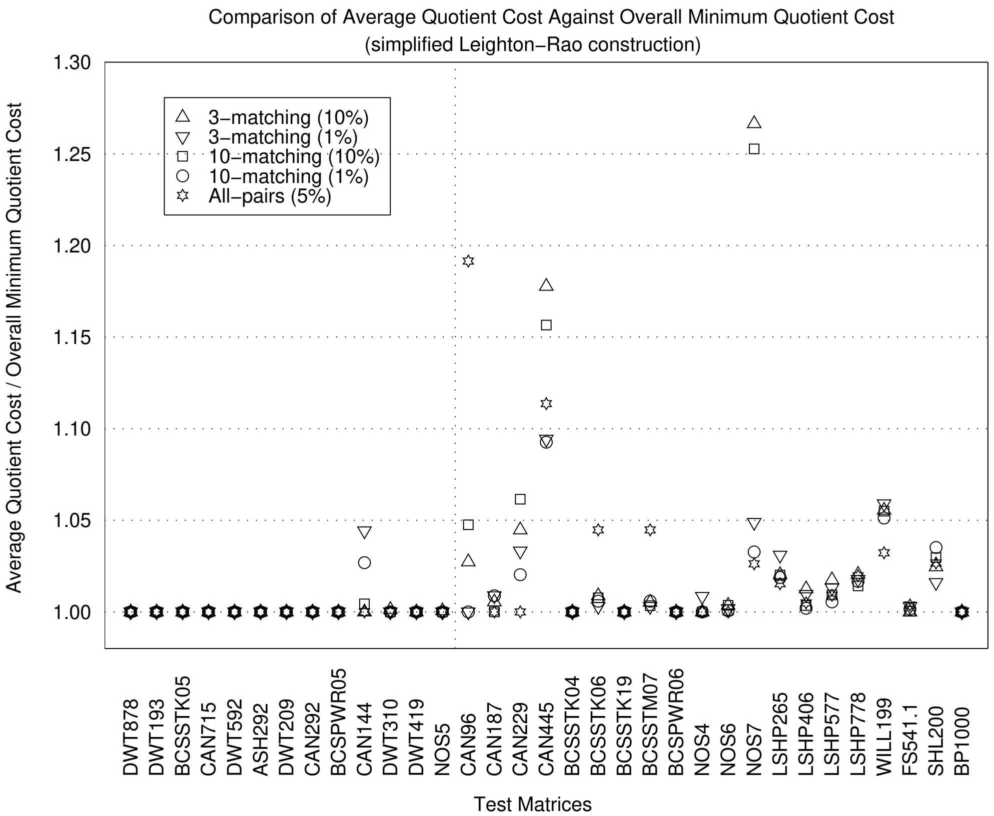


Figure 3.5: Comparison of the average quotient cost for each class of length functions to the minimum quotient cost over all classes, for the simplified Leighton-Rao construction.

cost for any class of length functions is at most 27% above the overall minimum quotient cost for that matrix. This contrasts sharply with the earlier average-to-minimum quotient cost comparison in Figure 3.3 for the general construction, and shows that by examining regions from all possible centers, we have "smoothed" out the performance of the construction. A related smoothing effect occurs on the variation in average quotient costs among the classes of length functions. With the exception of the matrices CAN96 and NOS7, we see that the average quotient costs of the various classes are now very similar to each other. More strikingly, the figure shows that for the repeated-length matrices, the average quotient costs of all their classes are virtually identical to the minimum, except for the matrix CAN144. Only 4 of the unique-length matrices exhibit this behavior. We shall explain this phenomenon more fully in Section 3.4.4, when we look at the cuts that are produced by a better variant of the construction.

Validation of the Simplified Construction

Despite the impressive results for the simplified construction relative to the general construction, we remind the reader that we do not have theoretical guarantees on the quality of the quotient cost of the cut produced, unlike the performance bounds that we were able to prove for the general construction. As we pointed out in Section 3.3, we incorporate an a posteriori expansion rate check into the simplified construction. The success of this check then enables us to conclude that the produced cut enjoys the same performance guarantee as that from the general construction. Specifically, for the unit-capacity case, the proof of the general Leighton-Rao construction relies on the existence of a region $B(z,r)$ containing roughly half the vertices such that the quantity $\log(W(z,r))/(r-1)$ is at least $\log(1+\alpha_{min}/33)$, where α_{min} is the quotient cost of the cut produced. The relation between the two quantities shows that the regions centered at z are rapidly increasing in weight as the radius increases. For the simplified construction, the a posteriori check compares the largest radius-scaled log-weight ratio over all sufficiently large regions to the desired expansion rate $\log(1+\alpha_{min}/33)$. It turns out that for our test cases, the largest observed ratio in a run is between 35 to 538 times that of the desired expansion rate. Thus the a posteriori check is always overwhelmingly successful, and the simplified construction is validated. The large excess of the observed ratio over the desired expansion rate, however, is more properly regarded as an indication that the expansion rate sought in the Leighton-Rao proof is overly pessimistic in practice.

3.4.4 Refining the Simplified Construction

There is a simple but effective procedure for reducing the quotient cost of a cut that we can incorporate into the Leighton-Rao construction. Consider a vertex cut (A,B,C) examined in the construction. Suppose a vertex in C has neighbors in A or B, but not in both.[5] Then v can be moved into A or B accordingly without violating the cut property. It is easily seen that moving v gives a cut with lower quotient cost. We therefore refine the cut by moving out of C all such vertices. The refinement procedure is easily implemented as a single pass over the vertices in C.

Unfortunately, the refinement procedure is not monotone in its reduction of quotient cost. Given two vertex cuts, where the first has lower quotient cost than the second, the quotient cost of the refined first cut can be higher than that of the second. Within the context of the Leighton-Rao construction, the non-monotonicity of the refinement process suggests that it is not effective to only refine the cut produced from the construction. Instead it is necessary to refine *each* cut that is examined, and then output the refined cut with the lowest quotient cost. For the simplified Leighton-Rao construction, refining each cut is a costly operation relative to the time taken to grow a region to the next radius, although the refined

[5] There will not be vertices from C that only have neighbors in C, because in the Leighton-Rao construction, C is the boundary of a region.

construction still takes much less time than that needed to solve the concurrent flow, as we shall see in Section 3.4.7.

Comparison between the Refined Simplified and Simplified Constructions

We compare the quotient costs of the cuts produced by the simplified Leighton-Rao construction with and without refinement in Figure 3.6. In the figure, we have plotted the ratios of the average quotient costs of each class of length functions. As before, we have separated the repeated-length matrices from the unique-length matrices. We see from the figure that refinement can have a significant impact,

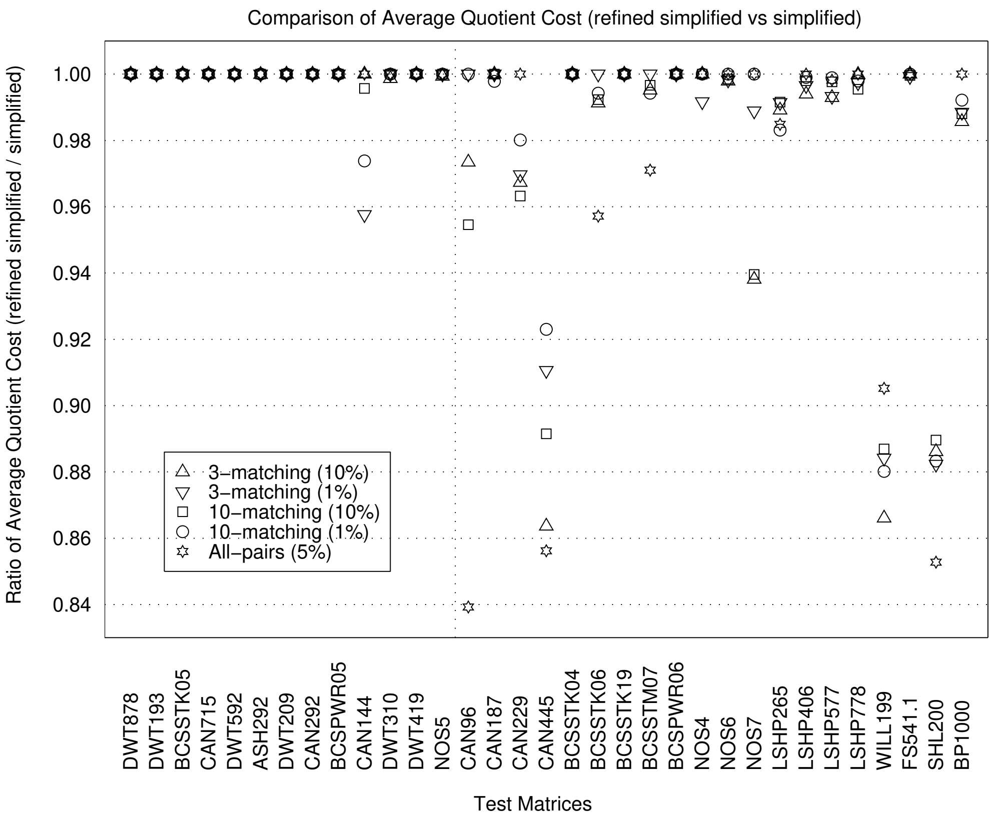


Figure 3.6: Comparison of the average quotient cost for each class of length functions for the refined simplified construction to those for the simplified construction.

at times lowering the average quotient cost by more than 10%. On the other hand, there are a majority of instances where the improvement is less than 2%. And as in the previous plots, the behavior of the refinement procedure on the repeated-length matrices is different from that on the unique-length matrices, with only one repeated-length matrix being improved upon, whereas most of the unique-length matrices displayed improvement.

The comparison between the minimum quotient costs found by the refined simplified construction and that found by the simplified construction is remarkably

Table 3.5: Matrices for which the minimum quotient cost found by the refined simplified and simplified Leighton-Rao constructions differ.

Matrix	Minimum Quotient Cost Found	
	Simplified	Refined Simplified
ASH292	0.06278	0.06272 (−0.1%)
CAN445	0.20785	0.19091 (−8.2%)
LSHP577	0.08318	0.08229 (−1.1%)
LSHP778	0.07059	0.07040 (−0.3%)
WILL199	0.80808	0.71503 (−11.5%)
SHL200	0.47619	0.40752 (−14.4%)
BP1000	0.80000	0.75912 (−5.1%)

similar to the earlier comparison between the simplified and general constructions. For 26 of the 33 matrices, the minimum quotient costs found by the simplified construction with and without refinement are the same. The minimum quotient costs and the percentage improvements for the other 7 matrices are given in Table 3.5. We see that only one of the 7 matrices (ASH292) has repeated discretized length functions, but the improvement in minimum quotient cost is negligible. In addition, 6 of these matrices appear among the earlier corresponding exceptions listed in Table 3.4.

As for the average number of vertices moved in refining a cut, there is very little variation across the various classes of length functions for each matrix, but considerable differences among the various matrices. Overall, the average number of vertices moved ranged between 0.6 to 52 for the set of matrices. In relative terms, between 0.08% to 6.4% of the vertices of the graph were moved. There was however no relation between the absolute/relative average number of vertices moved and the degree of improvement obtained from refinement.

Comparison for the Classes of Length Functions

Figure 3.7 compares the average quotient cost of the refined cuts for each class of length functions to the minimum over all the classes. We see that refining the cuts has further smoothed out the average quotient costs, as compared to Figure 3.5 for the unrefined simplified construction. Indeed, the average quotient cost is at most 19% above the minimum, as compared to 27% for the unrefined construction. As before, with the exception of the matrices CAN445 and NOS7, the average quotient costs for the various classes of length functions are very similar to each other, with no class being distinctly superior to the others. In addition we see that for the repeated-length matrices, the quotient costs for its various classes are virtually identical.

We further evaluate the quotient costs of the various classes of length functions

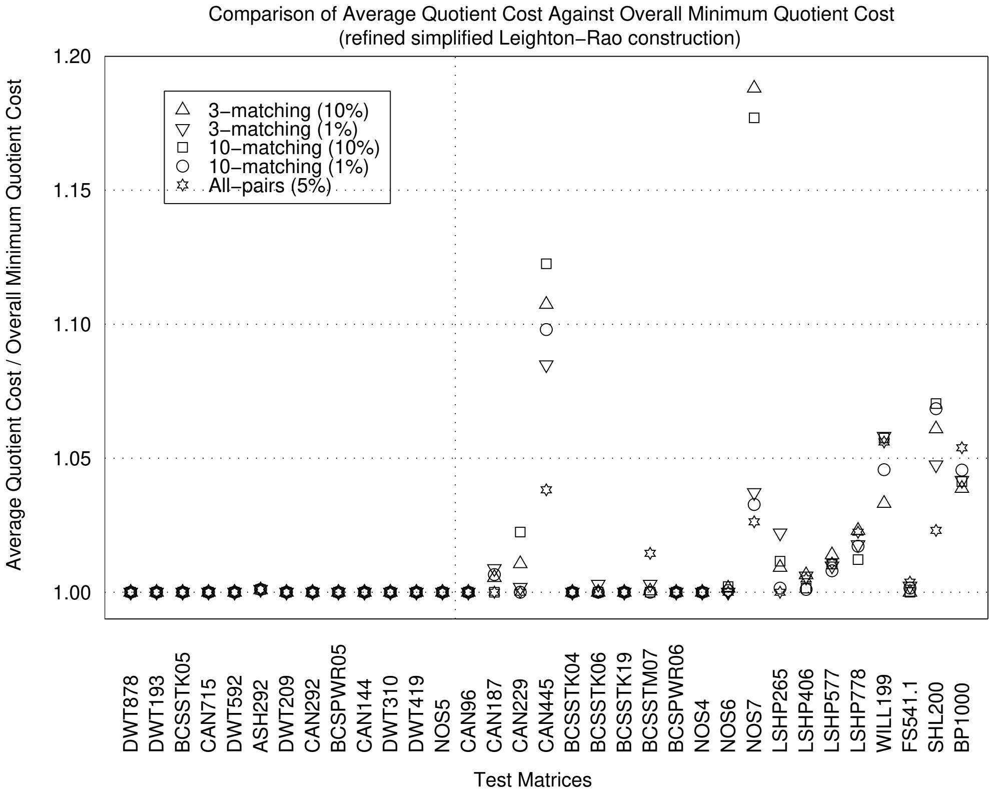


Figure 3.7: Comparison of the average quotient cost for each class of length functions to the minimum quotient cost over all classes, for the refined simplified Leighton-Rao construction.

for the refined simplified construction in Figure 3.8. Instead of looking at the average quotient costs of each class, we now compare its minimum quotient to the overall minimum. The figure looks similiar to the previous one, and we see that for each class of length functions, there are several matrices on which it did not achieve the overall minimum observed quotient cost.

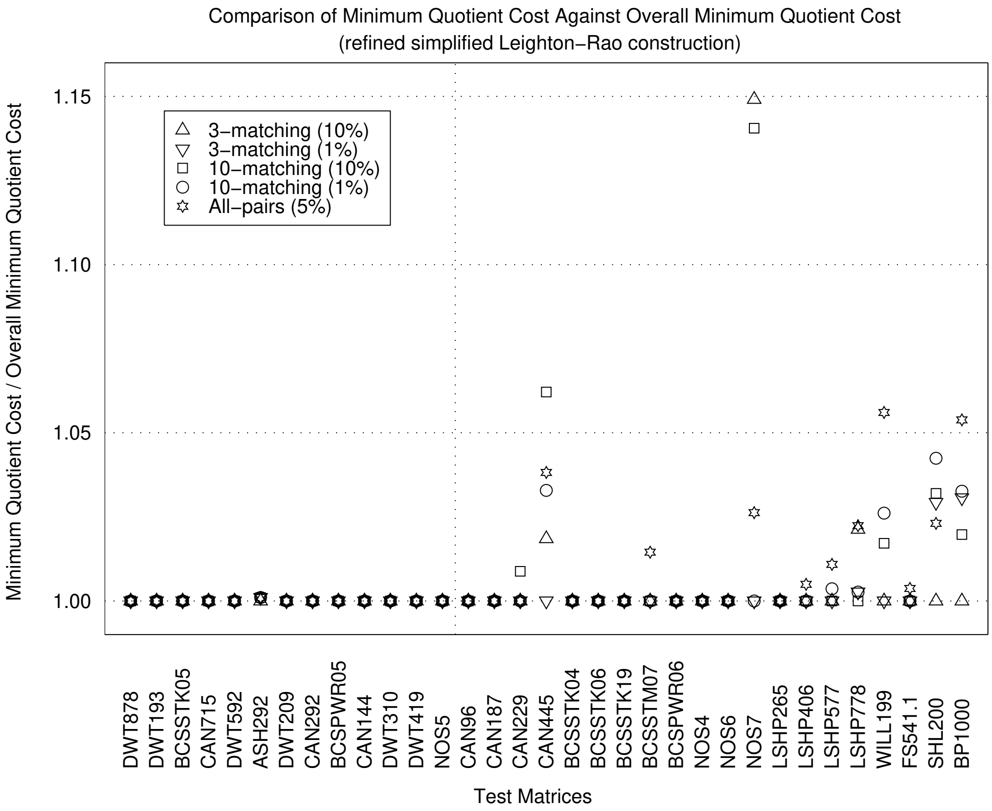


Figure 3.8: Comparison of the minimum quotient cost for each class of length functions to the minimum quotient cost over all classes, for the refined simplified Leighton-Rao construction.

Overall Statistics for the Cuts Found

In our evaluation of the various Leighton-Rao constructions so far, we have presented statistics that were computed on each class of length functions. We now put aside this distinction and consider instead overall statistics on the cuts that were found for the refined simplified construction, with the purpose of further understanding the behavior of the construction.

We first look in Table 3.6 at the minimum and maximum quotient costs that were found for each matrix. The table on the left gives the costs for the repeated-length matrices, while the table on the right is for the unique-length matrices. The maximum quotient costs found are expressed as percentages above the corresponding minimum quotient costs.

Table 3.6: Minimum and maximum quotient costs found for the refined simplified Leighton-Rao construction. The maximum quotient cost is expressed as a percentage above the minimum cost. The table on the left is for the repeated-length matrices, while the table on the right is for unique-length matrices.

Matrix	n	Quotient Cost	
		Min	Max
DWT878	878	0.0260	0%
DWT193	193	0.2198	0%
BCSSTK05	153	0.1600	0%
CAN715	715	0.0811	0%
DWT592	592	0.0531	0%
ASH292	292	0.0627	0.1%
DWT209	209	0.1065	0%
CAN292	292	0.0901	0%
BCSPWR05	443	0.0267	0%
CAN144	144	0.0833	0%
DWT310	310	0.0459	0%
DWT419	419	0.0642	0%
NOS5	468	0.2222	0%

Matrix	n	Quotient Cost	
		Min	Max
CAN96	96	0.2500	0%
CAN187	187	0.1176	2.2%
CAN229	229	0.1659	4.6%
CAN445	445	0.1909	21.7%
BCSSTK04	132	0.3636	0%
BCSSTK06	420	0.1714	1.4%
BCSSTK19	817	0.0073	0%
BCSSTM07	420	0.1714	1.4%
BCSPWR06	1454	0.0055	0%
NOS4	100	0.1414	0%
NOS6	675	0.0444	0.6%
NOS7	729	0.1674	21.8%
LSHP265	265	0.1212	5.6%
LSHP406	406	0.0985	2.5%
LSHP577	577	0.0823	3.1%
LSHP778	778	0.0704	2.5%
WILL199	199	0.7150	11.3%
FS541.1	541	0.0702	0.4%
SHL200	663	0.4075	11.8%
BP1000	822	0.7591	5.4%

We see that the maximum quotient costs for the repeated-length matrices are always identical to their minimum costs, with the exception of the matrix ASH292, for which it is only 0.1% higher. The agreement of the maximum quotient cost with the minimum for a matrix is a consequence of there being only one distinct quotient cost among the 31 cuts that were found for that matrix, as we shall examine in greater detail later. Turning to the unique-length matrices, we see that there are only 5 matrices out of the 20 where the maximum and minimum quotient costs coincide. The highest relative maximum quotient costs, in excess of 20% above the minimum, occur for the matrices CAN445 and NOS7. These matrices have larger variations in the quotient costs of their cuts, and so the average quotient cost for a class of length functions can be significantly higher than the minimum quotient cost, as we previously observed in Figure 3.7.

From the tables, we also observe that there is a wide range of minimum quotient costs found across the set of test matrices, with the matrix BCSPWR06 having the smallest minimum quotient cost of 0.0055 and the matrix BP1000 having the largest at 0.7591. There is no apparent relation between the size of the minimum quotient cost for a matrix and it having repeated or unique discretized length functions, nor is there any relation between the absolute sizes of the minimum quotient costs and the relative sizes of the corresponding maximum costs.

We now look for repetitions within the cuts produced for each matrix, and the data is given in Table 3.7. As before, we have tabulated the repeated-length matrices separately from the unique-length matrices. For each repeated-length matrix, we give the number of distinct discretized length functions that it possesses, and the number of distinct cuts and quotient costs among the 31 cuts produced by the refined simplified construction. We also give the number of cuts which achieved the minimum quotient cost found. Similar data is presented for the unique-length matrices.

We see from the tables that for 9 of the 13 repeated-length matrices, there was only one distinct cut among the 31 produced. The same phenomenon holds for the unique-length matrices BCSSTK19 and BCSPWR06. This is rather surprising since many of those matrices have numerous distinct discretized length functions, and yet these length functions result in the same cut being produced. We speculate that these matrices are "robust" with respect to the Leighton-Rao construction.[6] In addition we see that although there were multiple distinct cuts found for the repeated-length matrices CAN292, CAN144 and DWT419, and for the unique-length matrices CAN96, BCSSTK04 and NOS4, these distinct cuts have the same quotient cost. A similar situation is present for the unique-length matrices CAN229, BCSSTK06, BCSSTM07, LSHP265, LSHP406, LSHP577 and FS541.1, where the number of distinct cuts found is much higher than the number of distinct quotient costs among those cuts. These instances of multiple cuts having the same quotient cost suggest that those matrices have some degree of symmetry in their graphs. Finally we observe that there are only a few matrices (CAN445,

[6]Another possibility is that the distinct discretized length functions are in fact close to each other, which we have not attempted to measure.

Table 3.7: Number of distinct cuts and quotient costs and frequency of minimum quotient cost among the 31 cuts produced by the refined simplified Leighton-Rao construction. The table on the left is for the repeated-length matrices, while the table on the right is for unique-length matrices.

Matrix	# Distinct			Min q
	$l^{\rightarrow}$	Cuts	q	Freq
DWT878	1	1	1	31
DWT193	17	1	1	31
BCSSTK05	19	1	1	31
CAN715	22	1	1	31
DWT592	22	1	1	31
ASH292	23	2	2	1
DWT209	24	1	1	31
CAN292	26	3	1	31
BCSPWR05	27	1	1	31
CAN144	30	11	1	31
DWT310	30	1	1	31
DWT419	30	2	1	31
NOS5	30	1	1	31

Matrix	# Distinct		Min q
	Cuts	q	Freq
CAN96	8	1	31
CAN187	4	3	24
CAN229	22	6	11
CAN445	31	26	1
BCSSTK04	2	1	31
BCSSTK06	12	2	30
BCSSTK19	1	1	31
BCSSTM07	10	3	28
BCSPWR06	1	1	31
NOS4	4	1	31
NOS6	5	3	22
NOS7	29	23	4
LSHP265	21	6	12
LSHP406	29	6	17
LSHP577	28	7	3
LSHP778	31	10	1
WILL199	31	21	3
FS541.1	26	2	25
SHL200	31	31	1
BP1000	26	22	1

NOS7, WILL199, SHL200 and BP1000) with a high number of distinct cuts found. As expected, the two matrices (CAN445 and NOS7) with significant variation in the average quotient costs of each class of length functions (cf. Figure 3.7) belong in that category.

Final Remarks on the Various Constructions and Classes of Length Functions

Throughout our empirical evaluation of the Leighton-Rao construction, we have seen that except for a handful of matrices, the quality of the cuts produced is quite similar across the various classes of length functions used, with no one class clearly dominating the others. In particular, for the refined simplified construction, Figure 3.7 showed that the average quotient costs of each class were virtually identical to the minimum quotient costs for many of the test matrices, and were close to each other for all but 2 of the matrices. For the two exceptional matrices CAN445 and NOS7, the average quotient costs for the 0.1-optimal classes of length functions were the highest, at between 10%-12% and 17%-19% above the minimum respectively.

Given the highly similar behavior of the various classes of length functions on almost all of the test matrices, we conclude that most of the time the class of length function used in the Leighton-Rao construction has little systematic impact on the quality of the cuts produced. In addition we have seen that the refined simplified version is the best among the three constructions, and as we shall see in Section 3.4.7, its running time is still reasonable. Therefore, to produce the sparse cuts that are needed in the elimination ordering algorithm of Chapter 4, we use the refined simplified Leighton-Rao construction on the class of length functions that are computationally easiest to solve, viz. those that are obtained by solving to within 10% a concurrent flow problem where the demand graph is composed of three matchings.

3.4.5 Comparison to Breadth-First Search Cuts

The evaluation of the Leighton-Rao constructions have so far been focused on the differences between the various constructions and the variations in the quality of the cuts produced. We now turn our attention to comparing the Leighton-Rao constructions against the following simplistic cut construction procedure. Given a graph, we compute a cut for it by running the refined simplified construction with a discretized length function assigning unit length to all the vertices. It is easily seen that for this simplistic construction, the region growing process from a given center reduces to breadth-first search (BFS) from that center. We shall refer to the resulting cut as the *refined BFS cut*.

Since the general Leighton-Rao construction has worst-case guarantees on the quotient cost of the cut produced (cf. Theorems 3.7 and 3.8), we compare the worst cuts from each of the Leighton-Rao constructions to the refined BFS cuts in Figure 3.9. For each construction and matrix, we plot the ratio of the maximum quotient cost found to the quotient cost of the refined BFS cut. As in previous plots we have separated the repeated-length matrices from the unique-length matrices.

We see surprisingly that for many of the cases, the maximum quotient cost

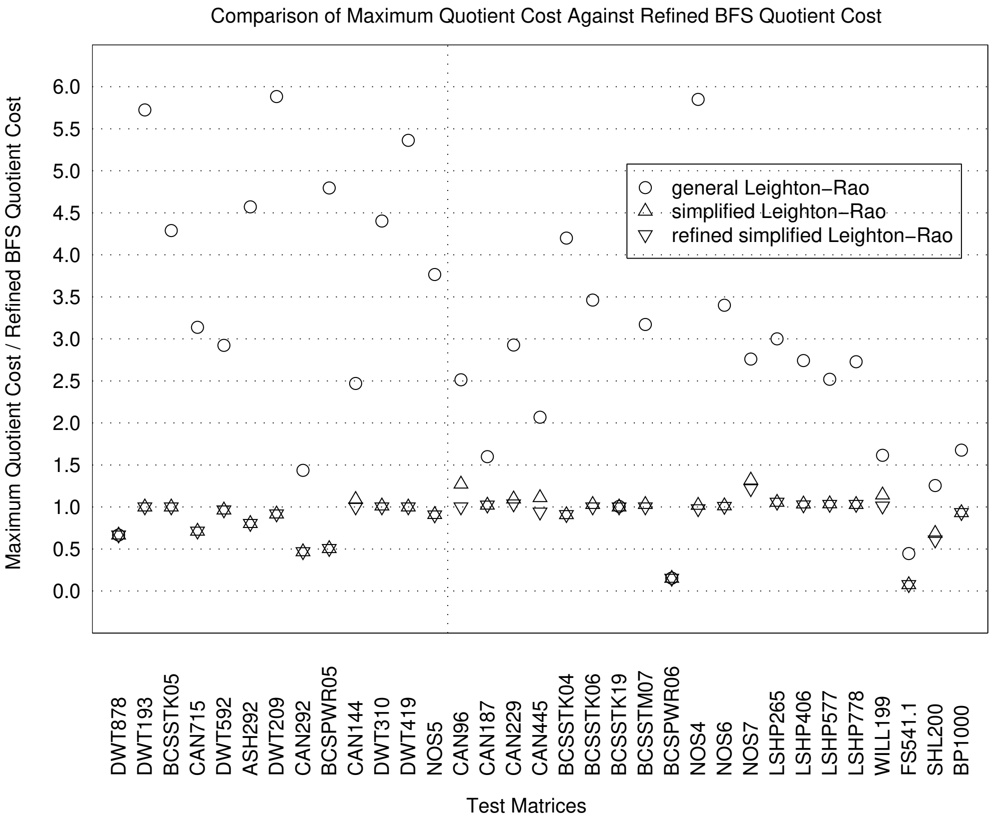


Figure 3.9: Comparison of the maximum quotient cost found by each of the Leighton-Rao constructions to the quotient cost of the refined BFS cut.

found by the Leighton-Rao construction was *worse* than the refined BFS quotient cost. Indeed, for 29 of the 33 matrices, the maximum quotient cost found by the general Leighton-Rao construction was at least 10% higher than the refined BFS quotient cost, with the worst matrix having a maximum quotient cost that was close to 6 times the refined BFS quotient cost. The simplified constructions fare somewhat better in comparision to the refined BFS cuts. For the unrefined simplified construction, there are 16 matrices for which the maximum quotient cost found was higher than the refined BFS quotient cost, of which four matrices had an excess greater than 10%, with the worst at 27%. There are also four matrices where the maximum quotient cost agreed with the refined BFS quotient cost. As for the refined simplified construction, there are 9 matrices for which the maximum quotient cost found was higher than the refined BFS quotient cost, of which 8 of those excesses was under 5.6% and the largest was 22%. In addition, 9 of the matrices had maximum quotient cost equal to the refined BFS quotient cost.

On the other hand, the figure also shows that there are numerous matrices for which the Leighton-Rao constructions were significantly better in the quotient costs of the cuts that it finds, with an order-of-magnitude better quality in some cases. In particular, for the matrices BCSPWR06 and FS541.1, the worst quotient cost found by both the unrefined and refined simplified constructions were respectively only 15% and 7.6% of the refined BFS quotient cost.

We note also from the figure that for a matrix, there is no apparent relation between how well the refined BFS cut compares against the Leighton-Rao cuts and whether that matrix has repeated or unique discretized length functions.

We further compare the Leighton-Rao constructed cuts to the refined BFS cuts in Figure 3.10. For each construction and matrix, we now plot the ratio of the minimum quotient cost found to the quotient cost of the refined BFS cut.

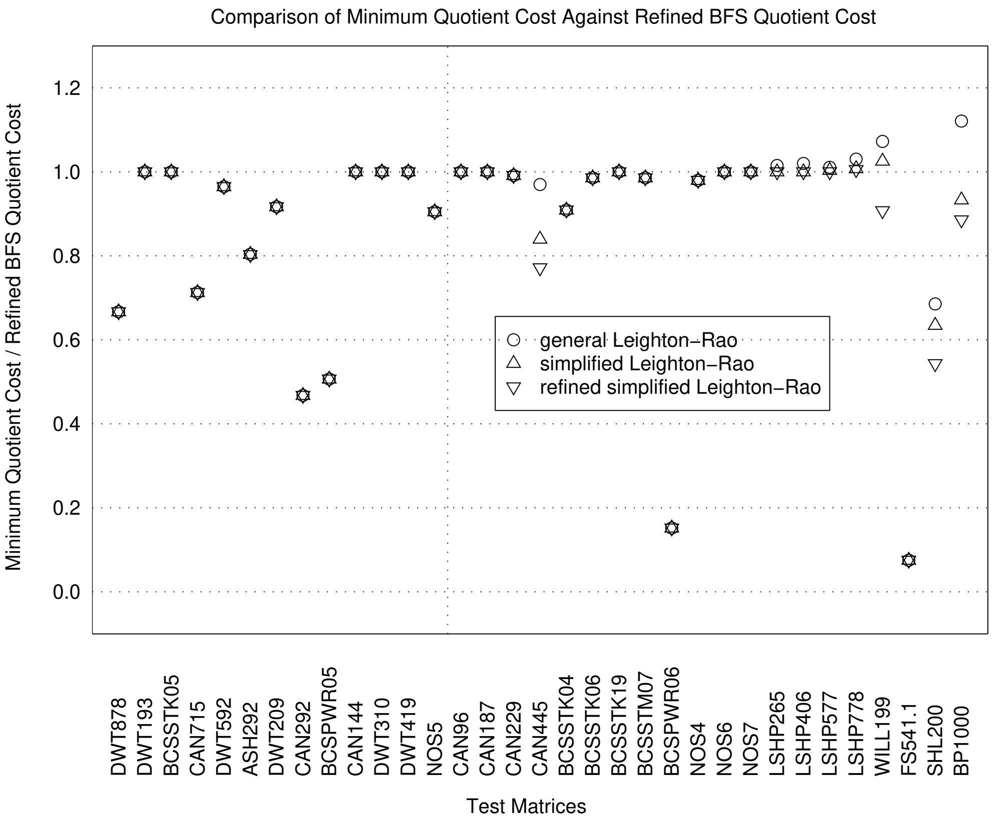


Figure 3.10: Comparison of the minimum quotient cost found by each of the Leighton-Rao constructions to the quotient cost of the refined BFS cut.

Surprisingly, there are still matrices for which the minimum quotient cost found by one or more of the Leighton-Rao constructions exceeded the refined BFS quotient cost. In fact, for the refined simplified construction, which we have seen to be the best among the three variants, each of the 31 cuts found for the matrix LSHP778 has slightly higher quotient cost (under 3% excess) than that of the refined BFS cut. In addition, for the matrix NOS7, four of the 31 cuts from the refined simplified construction have the same quotient cost as the refined BFS cut, six of them have quotient costs that are 2.6% to 6.2% higher, while the remaining 21 cuts have quotient costs that are at least 14% higher.

It is interesting to attempt to explain why the refined BFS construction can be so effective in practice. We shall do so by looking at the unit-length function within the context of an all-pairs concurrent flow problem and its dual. Recall that the ideal length function for the Leighton-Rao construction is given by the optimal solution to such a dual problem. We therefore compute the dual objective value for the unit-length function and compare it against the optimal value. However we do not have the optimal value at hand, and so we instead make the comparison against the (primal) flow congestion obtained for the all-pairs concurrent flow problem solved with an error tolerance of 5%. As we have remarked in Section 3.4.1, these congestion values were within 1.25% of optimal. Note that the deviation of the objective value of the unit-length function from the flow congestion upper bounds its deviation from the optimal value.

Figure 3.11 compares the minimum, average and maximum quotient costs for the cuts found by the refined simplified construction to the refined BFS quotient costs. The matrices in the plot are arranged in ascending order of the ratio of the dual objective value for the unit-length function to the corresponding all-pairs flow congestion. For example, in one extreme case CAN96, the objective value for the unit-length function is within 0.1% of the flow congestion, and consequently within 0.1% of optimal. On the other extreme, the unit-length dual objective value for the matrix BCSPWR06 is only 8.6% of the flow congestion. We see from the figure that there is some relation between the relative dual objective values and the performance of the refined BFS cuts against the Leighton-Rao cuts. On the high end, there are eight matrices (LSHP778 through CAN96) where the unit-length function is 0.1-optimal for the all-pairs problem and so has comparable quality to the classes of length functions that we have used previously. We would therefore expect that the refined BFS cut would be good for those matrices. Indeed for 7 of those 8 matrices, the maximum quotient cost found by the refined simplified Leighton-Rao construction is higher than the refined BFS quotient cost, while the two costs are equal for the remaining matrix CAN96. At the other extreme, for the matrices BCSPWR06 through ASH292, where the dual objective value was at most 44.1% of the flow congestion, we see that the quotient cost of the refined BFS cut is always worse than the quotient costs of all the cuts from the refined simplified construction, with the exception of the matrix BCSSTK19. The relative size of the unit-length objective value therefore partially explains the performance of the refined BFS cuts. Note that the figure also shows that there is no apparent relation between the performance of the refined BFS cuts and the multiplicity of the quotient costs of the cuts produced by the refined simplified construction. For example, there was only one distinct quotient cost found for the matrix FS541.1, and the quotient cost of the refined BFS cut is poor in comparision. However, several of the matrices (eg. CAN144) also have only one distinct quotient cost, which agreed with the refined BFS quotient cost.

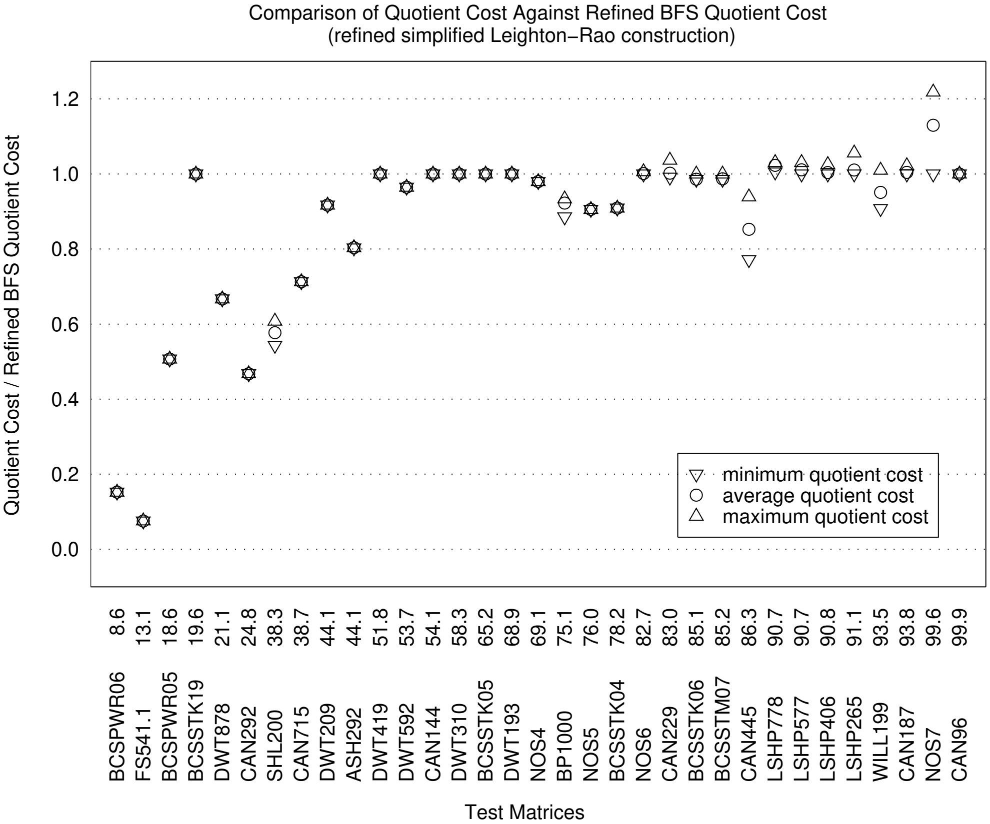


Figure 3.11: Comparison of the minimum, average and maximum quotient cost found by the refined simplified construction to the quotient cost of the refined BFS cut. The number above each matrix is the dual objective value of the unit-length function expressed as a percentage of the corresponding all-pairs flow congestion.

3.4.6 Comparison of Ratio Costs to LP Lower Bounds

In Section 3.1.3, we showed that the reciprocal of the optimal flow congestion of an all-pairs concurrent flow problem is a lower bound on the minimum ratio cost attainable in a graph. As Leighton and Rao have observed (cf. Section 3.1.1), the lower bound may not be achievable in general, and there exists graphs for which there is an $O(\log n)$-factor separation between the lower bound and the minimum ratio cost for that graph. We therefore now evaluate the graphs of our test matrices to see if they possess the $O(\log n)$ separation property by looking at the ratio costs of the cuts found by the various Leighton-Rao constructions. As in the previous section on the refined breadth-first search cuts, the flow congestion values that we used were obtained by solving an all-pairs problem with an error tolerance of 5%. We have seen in Section 3.4.1 that these congestion values were actually within 1.25% of optimal. Note that the reciprocals of these almost-optimal congestion values are slightly lower than the reciprocals of the optimal congestion values, and so they provide valid lower bounds.

Since the Leigthon-Rao construction seeks a cut with low quotient cost, it does not necessarily produce a cut with low ratio cost. However, the minimum quotient cost of a graph is between $n/2$ and n times its minimum ratio cost, as we have observed in Section 3.1.2, and so the two quantities can be estimated within a factor of 2 of each other. Therefore a Leighton-Rao cut that has ratio cost no more than twice the lower bound can be considered to be good with respect to minimizing the ratio cost. In general, the ratio cost of the cut produced by the Leighton-Rao construction also has the same $O(\log n)$ optimality guarantee as does the quotient cost, because of the factor-of-2 relation between these two quantities.

Figure 3.12 compares the ratio cost of the cuts found by the Leighton-Rao construction to the all-pairs lower bound. For each construction and matrix, we plot the maximum ratio cost among the cuts found divided by the corresponding lower bound. We have also plotted the ratio costs of the refined BFS cuts in the same manner. The matrices in the figure are arranged in ascending order of their vertex sizes, with the natural logarithms of the number of vertices shown above the matrices. We see from the figure that the general Leighton-Rao construction has the worst maximum ratio costs among the three variants, and is only better than the refined BFS construction for three of the matrices (FS541.1, DWT878 and BCSPWR06). Indeed the general construction produced a maximum ratio cost that is more than twice the lower bound for 29 of the 33 matrices, with one matrix (BCSPWR05) where the maximum ratio cost was more than 7 times the lower bound. In contrast, the unrefined and refined simplified construction have only 2 and 3 matrices respectively for which they produced maximum ratio costs exceeding twice the lower bound, while the ratio cost of the refined BFS cuts crossed that threshold for 6 of the 33 matrices, with one matrix (FS541.1) where it exceeded 7 times the lower bound. These results are however not surprising given the similar comparison between the maximum quotient costs of the various constructions in Section 3.4.5 and the factor-of-2 relation between the minimum quotient and ratio costs. We see from the figure that there are also many instances

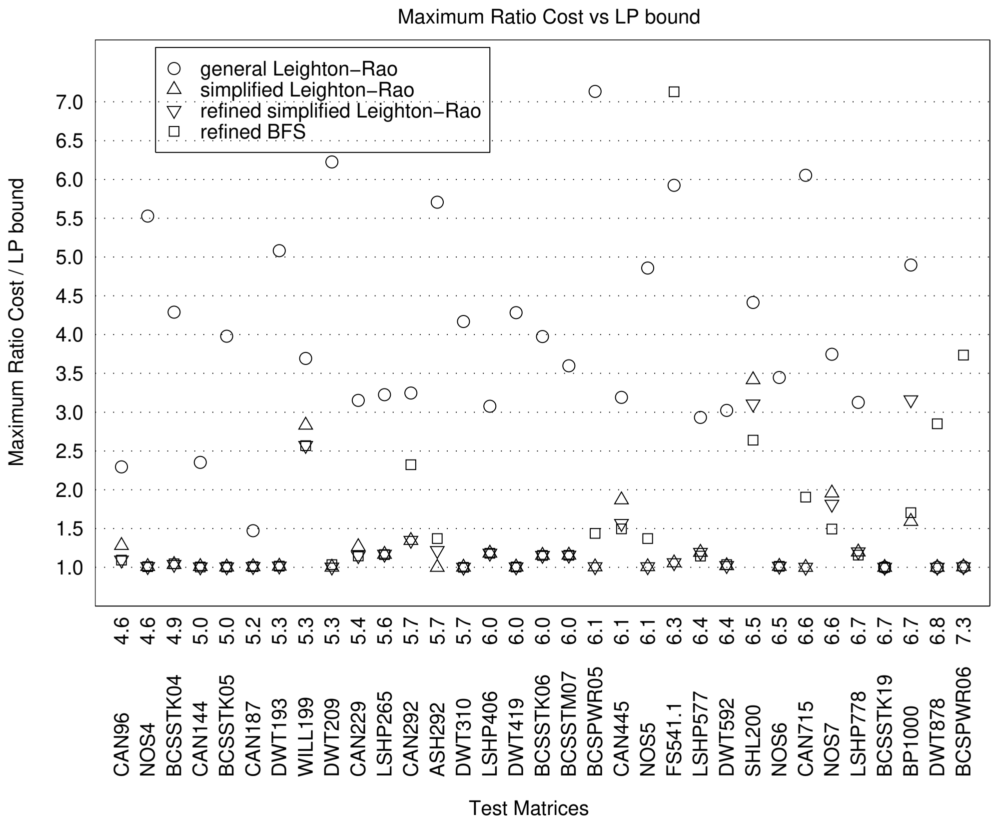


Figure 3.12: Comparison of the maximum ratio cost found by each of the Leighton-Rao constructions and the refined BFS ratio cost to the all-pairs concurrent flow lower bound. The number above each matrix is the natural logarithm of the number of vertices in its graph.

where the ratio costs of the Leighton-Rao and BFS cuts are practically equal to the lower bound, indicating that those particular matrices do not have an $O(\log n)$ ratio cost separation.

Although refining a cut is easily seen to result in another with lower quotient and ratio costs, it is not the case that the refined simplified construction will always find a cut with lower ratio cost than the unrefined simplified construction on the same input. This is because the cut with the lowest quotient cost sought by these constructions can be different from the cut with the lowest ratio cost. The matrix BP1000 in the figure provides one such example. Note however that the maximum ratio cost found by the refined simplified construction is within a factor of 2 of that found by the unrefined construction.

We also note from Figure 3.12 that there is no apparent relation between the quality with respect to the lower bound of the ratio costs of the Leighton-Rao cuts and the size of the graph. There does not seem to be a relation also between the optimality of the refined BFS ratio costs and the size of the graph.

To more conclusively determine if our test matrices possess the $O(\log n)$ separation property, we look at the minimum ratio costs of the cuts produced by the Leighton-Rao constructions in Figure 3.13. The matrices in the plot are arranged in ascending order of the relative dual objective value of the unit-length function, as in Figure 3.10. We have also plotted the ratio costs of the refined BFS cuts with respect to the lower bound. For the matrices BCSPWR06 and FS541.1, the refined BFS ratio costs are off the scale, and their values are indicated in parentheses. The figure shows that for many of the matrices, the minimum ratio cost attained by one or more of the constructions was practically identical to the lower bound. The notable exceptions are the matrices SHL200, BP1000, WILL199 and NOS7. However the minimum ratio costs attained by at least one of the constructions on these four matrices were within a factor of 2 of the lower bound. We therefore conclude that none of our test matrices possesses the $O(\log n)$ separation property. Finally we note that there is some relation between the optimality of the refined BFS ratio costs and the quality of the unit-length function. Among the matrices with the 10 lowest unit-length dual objective values (BCSPWR06 through ASH292), the ratio costs of the refined BFS cuts were at least 30% above the lower bound for 8 of the matrices. Only five of the other 23 matrices crossed the 30% threshold. This relation parallels the one we saw in Section 3.4.5 on the quotient costs of the refined BFS cuts.

3.4.7 Miscellaneous Statistics for the Leighton-Rao Constructions

We conclude our discussion on the empirical behavior of the Leighton-Rao construction by presenting statistics on the running time of the various versions, the balancedness of the cuts produced, and the lengths of the vertices that are placed in those cuts.

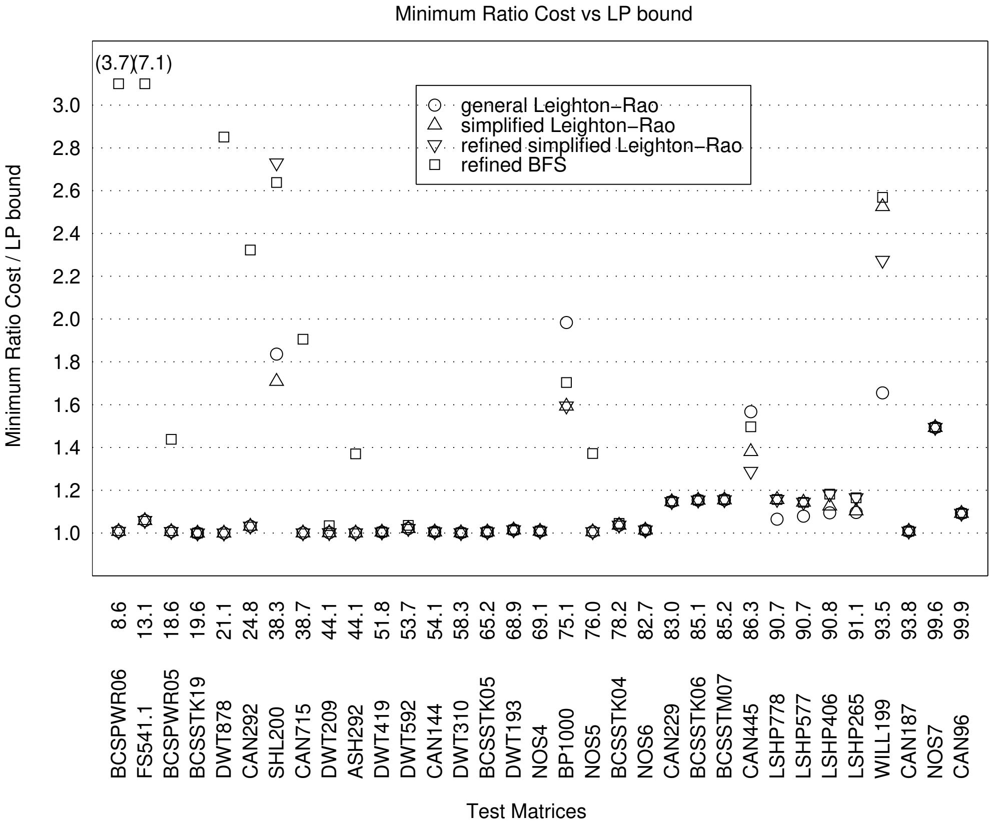


Figure 3.13: Comparison of the minimum ratio cost found by each of the Leighton-Rao constructions and the refined BFS ratio cost to the all-pairs concurrent flow lower bound. The number above each matrix is the dual objective value of the unit-length function expressed as a percentage of the corresponding all-pairs flow congestion.

Running time. We compare the average running times of the region-growing and cut-examination procesess for each of the three constructions to the time taken to find a 0.1-optimal solution to a concurrent flow problem whose demand graph is composed of 3 matchings:

- For the general Leighton-Rao construction, there are 19 matrices where the average construction time was under 2% of the time taken to solve the concurrent flow problem. The rest of the matrices had construction times between 2.6% to 9.5% of the concurrent flow solution time.

- For the simplified construction, there are 21 matrices where the average construction time was under 14% of the concurrent flow solution time and 12 matrices where it exceeded 23% but was at most 45%.

- For the refined simplified construction, there are 7 matrices where the average construction time was under 10% of the concurrent flow solution time, 15 matrices where it was between 10% and 31%, and 8 matrices where it exceeded 38% and reached up to 82%.

We see that although the refined simplified construction is the slowest among the three, its running time is substantially lower than the solution time for the concurrent flow. Indeed, for two-thirds of the test matrices, the refined simplified construction time is at most one-third the concurrent flow solution time. We conclude therefore that in practice, the bottleneck in the Leighton-Rao construction is in the solution of the concurrent flow problem.

3.5 Local Search for Sparse Vertex Cuts

The major drawback of the Leighton-Rao sparse vertex cut construction is that it is computationally expensive. In fact more than 90% of its total running time is spent in solving the concurrent flow and its dual. In practice, one usually turns to cut generation procedures that are based on fast heuristics to get reasonable (but not provably) good cuts. One family of such procedures is based on local search, in which an initial solution is repeatedly improved by making small local pertubrations. One of the most popular local search strategies is simluated annealing, and we describe in this section how we applied it to the problem of finding sparse cuts. Our investigation is a scaled-down version of the empirical study performed by Lang and Rao [55], where they evaluated various heuristics for constructing approximately optimal *edge* cuts. The focus here is on assessing the potential of simulated annealing. Hence we will only perform a limited set of experiments with simulated annealing. In addition, we will not evaluate other heuristics like the Fiduccia-Mattheyses variant [23] of the well-known Kernighan-Lin heuristic [49] (as was done by Lang and Rao [55]) or other optimization strategies like tabu search or genetic algorithms.

3.5.1 The Simulated Annealing Method

Simulated annealing is a general-purpose local search method for the approximate solution of difficult combinatorial optimization problems. It was introduced by Kirkpatrick *et al* [51] [50] and independently by Cerny [12], and has been successfully applied to various NP-hard problems, including VLSI design [88], the traveling salesman problem [48] and graph bisection [46]. The principal difference between simulated annealing and simple local search strategies is that the method sometimes moves from one solution to another that is *worse* in quality, and hence it is capable of backing out from poor local minima. We shall motivate and present a framework for simulated annealing in this section. For further details on simulated annealing and the extensive collection of problems to which it has been applied, we refer the reader to the books by van Laarhoven and Aarts [87] and by Otten and van Ginneken [69], and to the annotated bibliography by Collins *et al* [14].

The underlying ideas in simulated annealing come from an analogy with statistical mechanics, specifically the way that metals cool and anneal or liquids freeze and crystallize. At high temperatures, the molecules of a liquid move about freely. As the liquid is cooled, the molecules lose thermal mobility. By carefully controlling the cooling process, one can ensure that the molecules will arrange themselves into a perfect crystal. This crystal is the state of global minimum energy for the physical system. If, however, the liquid is rapidly cooled or "quenched," it instead freezes into a polycrystalline or amorphous arrangement with widespread irregularities. Such states are local energy minima having somewhat higher energy than a pure crystal. In general, the energy E of a thermodynamical system (eg. the liquid) at thermal equilibrium at a temperature T is probabilistically distributed according to the Boltzman distribution:

$$Prob(E) \propto e^{-E/kT}.$$

The quantity k is Boltzman's constant, and is a constant of nature relating temperature to energy. We see that the system favors lower energy states, but that there is a possibility of it being a high-energy state. Hence there is a corresponding chance of the system leaving a local energy minimum in search of a global minimum. However, the probability of such an "uphill" move, especially a significant one, decreases quickly as the temperature drops. By slowly lowering the temperature through a series of levels, we allow the system to make the necessary uphill moves for reaching the global energy minimum.

The thermodynamical principles of a system at equilibrium were first incorporated into a computational setting by Metropolis *et al* [67], who developed a simulation technique for a system at a fixed temperature T. At each step, the simulated system is presented with a proposed change to the current configuration. If the energy E_2 of the modified configuration is less than or equal to the energy E_1 of the current configuration, the system accepts that change. If $E_2 > E_1$, the proposed change instead occurs with probability $e^{-(E_2-E_1)/kT}$. By combining the Metropolis *et al simulation* algorithm with an *annealing* schedule of decreasing temperatures, Kirkpatrick *et al* devised the simulated annealing method. Simulated annealing is thus an algorithmic analog of physical annealing.

To apply simulated annealing to an optimization problem, the following four elements have to be specified:

1. A description of the possible system configurations. The set of configurations is usually just the set of feasible solutions to the problem instance. In some applications, it may be enlarged to include infeasible solutions, with the idea of providing more "escape" routes from poor local minima. It is then necessary to also provide a procedure for converting an infeasible solution into a feasible one.

2. A generator of random changes to the current configuration, with the property that it should be possible to go from one configuration to any other configuration via a finite sequence of changes. We let M denote the number of changes (or "moves") available from a typical configuration.

3. A cost function (analog of energy) to be minimized by the procedure. The cost function is typically the objective function of the optimization problem at hand. However, in the case where infeasible solutions are included as configurations, the cost function is a combination of the objective function and a penalty function reflecting the extent of infeasibility of a configuration. Another possibility is for the cost function to be some auxiliary function whose minimization has the side effect of minimizing the objective function.

4. A control parameter T (analog of temperature) and an annealing schedule describing the set of decreasing temperatures to use and the amount of time to be spent on the Metropolis algorithm at each T. There exists annealing schedules which are guaranteed to produce optimal configurations (see, for example, Geman and Geman [29] and Sasaki and Hajek [78]), just as in physical annealing. Unfortunately, such schedules involve an exponential number of steps and are therefore not feasible. In practice, finding a good annealing schedule is difficult and usually requires trial-and-error experiments. Such practical schedules have no guarantees on the quality, i.e. how close to optimality, of the solution produced.

The above elements are combined into the generic simulated annealing algorithm below. The algorithm essentially follows that given in Johnson *et al* [47], and allows the system configurations to include infeasible solutions.

<u>SIMULATEDANNEALING</u>

/ initialization */*
generate an initial feasible configuration $\mathcal{C}$
set current best optimal solution $\mathcal{C}^*$ to $\mathcal{C}$
choose an initial temperature $T > 0$

/ annealing */*
$freezecnt \leftarrow 0$ */* # Ts with few configuration changes */*

while $freezecnt <$ FREEZELIM
 /* *Metropolis algorithm* */
 $trials \leftarrow 0$ /* *number of proposed configuration changes* */
 $changes \leftarrow 0$ /* *number of actual configuration changes* */
 while $trials <$ SIZEFACTOR $\cdot M$ **and** $changes <$ CUTOFF $\cdot M$
  $trials \leftarrow trials + 1$
  generate a random change and let $\mathcal{C}'$ be the new configuration
  if $cost(\mathcal{C}') \leq cost(\mathcal{C})$ **then**   /* *downhill move* */
   $changes \leftarrow changes + 1$
   $\mathcal{C} \leftarrow \mathcal{C}'$
   if $\mathcal{C}$ is feasible **and** $cost(\mathcal{C}) < cost(\mathcal{C}^*)$ **then**
    $\mathcal{C}^* \leftarrow \mathcal{C}$
    $freezecnt \leftarrow 0$
   endif
  else   /* *uphill move* */
   generate a random number r in $[0, 1]$
   if $r \leq exp(-(cost(\mathcal{C}') - cost(\mathcal{C})) \ / \ T)$ **then**
    $changes \leftarrow changes + 1$
    $\mathcal{C} \leftarrow \mathcal{C}'$
   endif
  endif
 endwhile
 $T \leftarrow$ TEMPFACTOR $\cdot T$
 if $changes/trials <$ MINFRACT **then** $freezecnt \leftarrow freezecnt + 1$
endwhile
if C is infeasible
 convert C into a feasible solution
 if $cost(\mathcal{C}) < cost(\mathcal{C}^*)$ **then** $\mathcal{C}^* \leftarrow \mathcal{C}$
endif
output $\mathcal{C}^*$

The annealing schedule in the generic algorithm is controlled by several parameters. Recall that M denotes the number of changes available in a typical configuration. At each temperature setting (i.e. the Metropolis algorithm), we control the time spent by limiting the number of proposed configuration changes to SIZEFACTOR $\cdot M$ and the number of configurations reached to CUTOFF $\cdot M$. The latter cutoff restriction is motivated by observations (see [51] [50] [46] [47]) that the time spent at high temperatures seems to contribute little to the quality of the final solution. One way to reduce that time is to start with a lower temperature. However a starting temperature that is too low may cause the solution quality to degrade. Using cutoffs on the number of configurations reached allows us to save time at high temperatures, where configuration changes are more likely, while still leaving a safety margin in the starting temperature.

As for the temperature settings, we use a geometric sequence with the ratio TEMPFACTOR < 1. Johnson *et al* [46] [47] recommend choosing an initial temperature such that the *changes/trials* ratio is initially approximately equal to some parameter INITPROB. The annealing algorithm does not end with a directly specified final temperature. Instead, it terminates when the last FREEZELIM temperature settings did not improve the best feasible configuration seen so far, and the fraction of accepted changes is at most MINFRACT at each of those temperatures.

3.5.2 Simulated Annealing for Sparse Vertex Cuts

We describe in this section three approaches to finding vertex cuts with low quotient cost using simulated annealing. We shall use the generic algorithm described at the end of the previous section. Since we are only interested in assessing the potential of simulated annealing in finding sparse vertex cuts and not in conducting an extensive empirical study, we do not attempt to find good values for the parameters of the generic algorithm nor examine the tradeoffs between them. In addition, we do not try various vertex cuts as initial solutions for annealing, nor do we expend effort in aggressively optimizing the implementation. We adopt the following parameter values, which were determined by extensive testing by Johnson *et al* [46] to be good for the graph bisection problem:

$$\text{SIZEFACTOR} = 16, \quad \text{CUTOFF} = 4,$$
$$\text{INITPROB} = 0.4, \quad \text{TEMPFACTOR} = 0.95, \quad \text{MINFRACT} = 0.02, \quad \text{FREEZELIM} = 5.$$

We feel that these parameter settings are reasonable for the sparse vertex cut problem since it is related to the graph bisection problem. We remind the reader however that there is no theoretical guarantee on the quotient cost of the cut produced by simluated annealing, unlike the $O(\log n)$ performance guarantee from the Leighton-Rao construction in Section 3.2.

Initial Vertex Cut

We construct an initial vertex cut for each of the three approaches by running the refined simplified Leighton-Rao construction of Section 3.4.3 on a discretized length function which assigns each vertex a length of 1. For that uniform length function, the region growing process around a center z reduces to breadth first search from z. The entire Leighton-Rao construction therefore takes time $O(n(m+n))$, which in practice is only a tiny fraction of the time spent by simulated annealing.

It is conceivable that one could apply the simplified Leighton-Rao construction to other predetermined non-optimal vertex length functions to generate alternative initial vertex cuts, or to use completely different methods. As indicated earlier, we do not investigate these possiblities, as they are outside the scope of our study.

Initial Temperature

As stated in the discussion of the generic algorithm, the initial temperature T_0 is chosen such that the *changes/trials* ratio is initially approximately equal to the parameter INITIPROB. For each problem instance and annealing approach, we perform a trial phase where we run the Metropolis algorithm at various temperatures, each time starting from the initial vertex cut just described. We then manually select T_0 to be the temperature where the *changes/trials* ratio in the Metropolis run is closest to INITPROB, and use that starting temperature for multiple simulated annealing runs.

First Annealing Approach

In the first annealing approach, we let the set of configurations be the set of all vertex cuts of the input graph, and define the cost of a configuration to be its quotient cost. Let (A, B, C) denote the current configuration. We generate a random change by uniformly randomly picking a vertex v and moving it from the set it is in to one of the other two sets. However, the move can result in a partition that is *not* a vertex cut. It is therefore necessary to move *additional* vertices to restore the cut property. This is done as follows for the various source and destination set combinations. The key observation is that the cut property places no restriction on when a vertex can be in C, but does so on membership in A or B. In the description below, we let $(A_{new}, B_{new}, C_{new})$ be the configuration after the move. Recall that $\Gamma(v)$ is the set of vertices adjacent to v.

(i) Move v from C to A. Then the neighbors of v which are in B cannot remain there, and we move them into C. More precisely:

$$A_{new} \leftarrow A \cup \{v\}, \quad B_{new} \leftarrow B \setminus \Gamma(v), \quad \text{and} \quad C_{new} \leftarrow (C \setminus \{v\}) \cup (\Gamma(v) \cap B).$$

Instead of moving a neighor $w \in B$ of v into C, we could have moved it into A. However this may result in additional moves required for the neighbors of w to maintain the cut property. The update we have chosen moves the fewest number of vertices, although it has the possible effect of substantially enlarging C and hence significantly increasing the quotient cost. In general, the new quotient cost can be higher or lower than the current quotient cost.

(ii) Move v from C to B. This is symmetric to (i), with the roles of A and B interchanged.

(iii) Move v from A to C. The move does not destroy the cut property. Therefore

$$A_{new} \leftarrow A \setminus \{v\}, \quad B_{new} \leftarrow B, \quad \text{and} \quad C_{new} \leftarrow C \cup \{v\}.$$

It is easy to verify that the move is “uphill” and always increases the quotient cost, unless $B = \emptyset$.

(iv) Move v from B to C. This is symmetric to (iii).

(v) Move v from A to B. Then the neighbors of v which are in A cannot remain there, and we move them into C:

$$A_{new} \leftarrow A \setminus (\{v\} \cup \Gamma(v)), \quad B_{new} \leftarrow B \cup \{v\}, \quad \text{and} \quad C_{new} \leftarrow C \cup (\Gamma(v) \cap A).$$

As in (i), the update operation moves the fewest number of vertices. In addition, the new quotient cost can be higher or lower than the current quotient cost.

(vi) Move v from B to A. This is symmetric to (v).

It is easy to verify that the above list of moves generates a connected configuration space. Indeed, given two vertex cuts (A_1, B_1, C_1) and (A_2, B_2, C_2), we can go from the first cut to the second by the following sequence of intermediate cuts:

1. Go from $(A_1,\ B_1,\ C_1)$ to $(A_1,\ \emptyset,\ B_1 \cup C_1)$ using B-to-C moves.
2. Go from $(A_1,\ \emptyset,\ B_1 \cup C_1)$ to $(A_1 \cup B_1,\ \emptyset,\ C_1)$ using C-to-A moves.
3. Go from $(A_1 \cup B_1,\ \emptyset,\ C_1)$ to $(A_2 \cup B_2,\ \emptyset,\ C_2)$ using C-to-A and A-to-C moves.
4. Go from $(A_2 \cup B_2,\ \emptyset,\ C_2)$ to $(A_2,\ \emptyset,\ B_2 \cup C_2)$ using A-to-C moves.
5. Go from $(A_2,\ \emptyset,\ B_2 \cup C_2)$ to $(A_2,\ B_2,\ C_2)$ using C-to-B moves.

Note that the above transformation between two vertex cuts does not use A-to-B and B-to-A moves. This is not surprising because an A-to-B (B-to-A) move is equivalent to an A-to-C move followed by a C-to-B move (B-to-C move followed by C-to-A move). We include the A-to-B and B-to-A moves to give the annealing system greater "power" in modifying its current configuration.

Second Annealing Approach

We enhance the direct first approach by enlarging the set of configurations to include all 3-way partitions (A, B, C) of the vertex set V. The hope is that the additional configurations provide new "escape" routes out of local minima. The possible moves from a configuration are the same as those from the first approach, except that there is now no need to move the neighbors of the selected vertex v to restore the cut property. It is trivial to see that the set of moves gives a connected configuration space.

Since the configurations now include partitions that are not vertex cuts, the configuration cost function is modified by adding a penalty function that is nonzero for invalid cuts. In addition, a cleanup procedure must be provided to convert an infeasible configuration into a feasible one.

Specifically, let $e(A, B, C)$ be the number of edges in the given graph that go from a vertex in A to a vertex in B. The cost function is then:

$$cost(A, B, C) = q(A, B, C) + \text{PENALTY} \cdot e(A, B, C).$$

The parameter PENALTY is the *infeasiblity factor*. With a sufficiently high PENALTY value (eg. 2 for the unit-capacity case), one can ensure that the cost of an infeasible configuration always exceeds the cost of a feasible configuration, and thus drive the annealing algorithm towards feasible configurations. However, Johnson *et al*

[46] have observed that for the graph bisection problem, having a high imbalance factor (corresponding to our infeasiblity factor) discourages simulated annealing from visiting infeasble configurations and effectively negates the advantage of introducing those configurations. On the other extreme, the algorithm is likely to stop with a highly infeasible configuration when a very small factor is used, which may not be easily convertible to a good vertex cut. We therefore consider more carefully how to choose an appropriate value for PENALTY.

There are two natural values of PENALTY that suggest themselves. One would expect that a reasonable cost function should ensure that a configuration whose cost is a local minimum is actually feasible. For the unit-capacity case, which is our sole interest, the desired property holds when PENALTY $> \frac{1}{3}$, except for degenerate cases with $C = \emptyset$ or graphs with at most 5 vertices. This is easily seen by taking an infeasible configuration (A, B, C) and looking at how its cost changes when a vertex $v \in B$ with a neighbor in A is moved into C. Alternatively, one can relax the property by requiring it to hold only for the global cost minimum. The relaxed property is equivalent to specifying that a configuration with minimum cost is indeed a vertex cut with minimum quotient cost, which ensures that simulated annealing has the correct objective. This property easily holds when PENALTY $> \alpha$, where α is the (unknown) minimum quotient cost of the problem instance. A practical limit on PENALTY is obtained by using the quotient cost α_0 of the initial feasible configuration, or perhaps some fraction of it.

Surprisingly, the above suggested values for PENALTY may still be too high in practice. This conclusion is borne out by the observations of Johnson *et al* [46], briefly described above, on a penalty function approach to the related graph bisection problem. In the graph bisection problem, the objective is to divide the set of vertices into two equal-sized subsets V_1 and V_2 such that the cutsize, which is the number of edges with one endpoint in V_1 and the other in V_2, is minimized. Johnson *et al* evaluated a simulated annealing implementation where V_1 and V_2 need not have the same size, but their imbalance is factored into the cost function

$$cost(V_1, V_2) = cutsize(V_1, V_2) + \text{IMBALANCE} \cdot (|V_1| - |V_2|)^2.$$

Their experiments show that values of IMBALANCE between 0.003 and 0.2, a range with a 60-fold variation, resulted in equally good cutsizes, while higher values were worse. It is easily verified that none of those values guarantees that a 2-way partition with minimum cost is actually balanced. We see therefore that that it is difficult to predetermine an appropriate value for PENALTY. In our runs, we shall try the following values: $\min(\frac{1}{3}, \alpha_0)$, $\frac{1}{10}\min(\frac{1}{3}, \alpha_0)$ and $\frac{1}{100}\min(\frac{1}{3}, \alpha_0)$.

It is possible that the final configuration from simulated annealing is not feasible. We convert an infeasible final configuration into a vertex cut as follows. Let $(A_{bad}, B_{bad}, C_{bad})$ be a partition that is not a vertex cut. In the case where there are only a few edges between A_{bad} and B_{bad}, one can expect that moving some vertices from A_{bad} or B_{bad} into C_{bad} would eliminate those edges. Therefore we look at the two vertex cuts $(A_{bad},\ V \setminus (A_{bad} \cup \Gamma(A_{bad})),\ \Gamma(A_{bad}) \setminus A_{bad})$ and $(V \setminus (B_{bad} \cup \Gamma(B_{bad})),\ B_{bad},\ \Gamma(B_{bad}) \setminus B_{bad})$ and choose the one with the lower quotient cost. Note that we have seen this technique of defining a vertex cut in the Leighton-Rao construction, where we used the boundary of a region to separate it from the rest of the graph.

Third Annealing Approach

In the previous two approaches, there are essentially two "degrees" of freedom in defining a configuration, i.e. any two of the sets in a partition (A, B, C) fully defines the partition. We now consider a third annealing approach where there is only one degree of freedom in defining a configuration. The approach uses the neighborhood technique that we have just seen.

The set of configurations in the third approach consists of *minimum* vertex cuts specified by a single set (we will define the minimum property later). Let A be a non-empty proper subset of V. As we previously observed, $\mathcal{C}_A = (A,\ V \setminus (A \cup \Gamma(A)),\ \Gamma(A) \setminus A)$ is a vertex cut. Observe that *with respect to* A, $\Gamma(A) \setminus A$ is the *minimum* set of vertices that have to be removed to separate A from the rest of the graph. In other words, any vertex cut (A, B, C) will have $C \supseteq \Gamma(A) \setminus A$ and $B \subseteq V \setminus (A \cup \Gamma(A))$. Furthermore, it is easily verified that the quotient cost of the cut (A, B, C) cannot be lower than that of $\mathcal{C}_A$. There is therefore no loss of generality in restricting the set of configurations to be $\{\mathcal{C}_A : A \subset V, A \neq \emptyset, A \neq V\}$. Note that we do not allow A to be empty or equal to V, for otherwise $\Gamma(A) = \emptyset$, and the resulting partition is degenerate and is not a vertex cut.

The cost function for a configuration is just its quotient cost. The possible changes from a configuration are particulary simple, and involve uniformly randomly selecting a vertex $v \in V$, and moving v either into or out of A and redefining the vertex cut. In fact the vertex cut can be updated incrementally instead of being recomputed. Let (A, B, C) be the current cut, with $C = \Gamma(A) \setminus A$, and $(A_{new}, B_{new}, C_{new})$ be the new cut. We show below how we update A_{new} and C_{new}:

(i) Move v into A. Then any neighbor of v that is in B becomes adjacent to A_{new} and hence ends up in C_{new}. Furthermore, C_{new} does not contain v even if $v \in C$ originally. More precisely:

$$A_{new} \leftarrow A \cup \{v\} \quad \text{and} \quad C_{new} \leftarrow (C \setminus \{v\}) \cup (\Gamma(v) \cap B).$$

(ii) Move v out of A. Then any vertex in C where v is its only neighbor in A is omitted from C_{new}. In addition, if v is adjacent to some vertex of A, then it is moved into C_{new}. More precisely:

$$\begin{aligned} A_{new} &\leftarrow A \setminus \{v\} \\ C_{new} &\leftarrow \begin{cases} C \setminus \{w \in \Gamma(v) : \Gamma(w) \cap A = \{v\}\} & \text{if } v \notin \Gamma(A) \\ (C \setminus \{w \in \Gamma(v) : \Gamma(w) \cap A = \{v\}\}) \cup \{v\} & \text{if } v \in \Gamma(A) \end{cases} \end{aligned}$$

It is trivial to see that we can go from any configuration to any other configuration using the two types of moves above.

3.5.3 Experimental Results for Simulated Annealing for Sparse Cuts

We describe in this section some experimental results on simulated annealing applied to finding sparse vertex cuts. We remind the reader that we are only attempting to get an idea of the usefulness of simulated annealing for this problem, and hence our experiments are not as extensive as those that we performed for the Leighton-Rao construction in Section 3.4. As before, we used the set of 33 matrices from the Harwell-Boeing sparse matrix collection [18] [19] for our runs. The sizes of these matrices are listed in Table 3.1 in Section 3.4.1.

We presented in Section 3.5.2 three classes of annealing approaches for this problem: using general cuts, using arbitrary 3-way partitions with a penalty for crossing edges, and using cuts defined by the boundary of a subset of vertices. For the penalty function approach, we tried three different penalty factors (1, 0.1 and 0.01). Altogether, five annealing methods were performed, using the annealing parameters and setup that were described in Section 3.5.2. For each annealing method and each matrix, 30 random simulated annealing runs were performed. As a postprocessing step, we apply the refinement procedure described in Section 3.4.4 to the best cut found by each run. This refinement procedure moves unnecessary vertices out of the cut into the components, thus producing a modified cut with lower quotient cost.

It turned out that for some runs, the termination condition based on the last few temperature settings having a small fraction of actual configuration changes was consistently violated. This happened even though no improvement to the best cut found was made for many temperature steps. We therefore strengthened the termination condition by limiting the number of temperature steps in a run to 500.

We first compare the average quotient costs of the cuts found by simulated annealing in Figure 3.14. Among the three penalty function methods, the one with a penalty factor of 0.1 was usually the best or close to the best. We therefore do not give the results for the other two factors. In the figure, we have plotted the average quotient cost for each approach divided by the quotient cost of the refined BFS cut. Recall that this cut was used as the initial cut for the annealing runs, and so the cut found by simulated annealing cannot have worst cost than it. In addition, we have shown in the figure the average quotient costs of the refined simplified version of the Leighton-Rao construction with respect to the refined BFS quotient costs. We saw in Section 3.4 that this version was the best among the various Leighton-Rao constructions.

We see from the figure that the performance of simulated annealing is rather mixed. There are several matrices (eg. CAN96, CAN144 and CAN187) for which none of the annealing approaches was able to improve upon the initial refined BFS cut. However, the refined simplified Leighton-Rao construction had better average quotient cost than the refined BFS quotient cost for only two of those matrices (DWT592 and DWT878), and in fact it had *worse* average quotient cost for four of those matrices (NOS7, LSHP265, LSHP577, LSHP778). This suggests that the refined BFS cuts for those matrices were already good and therefore could

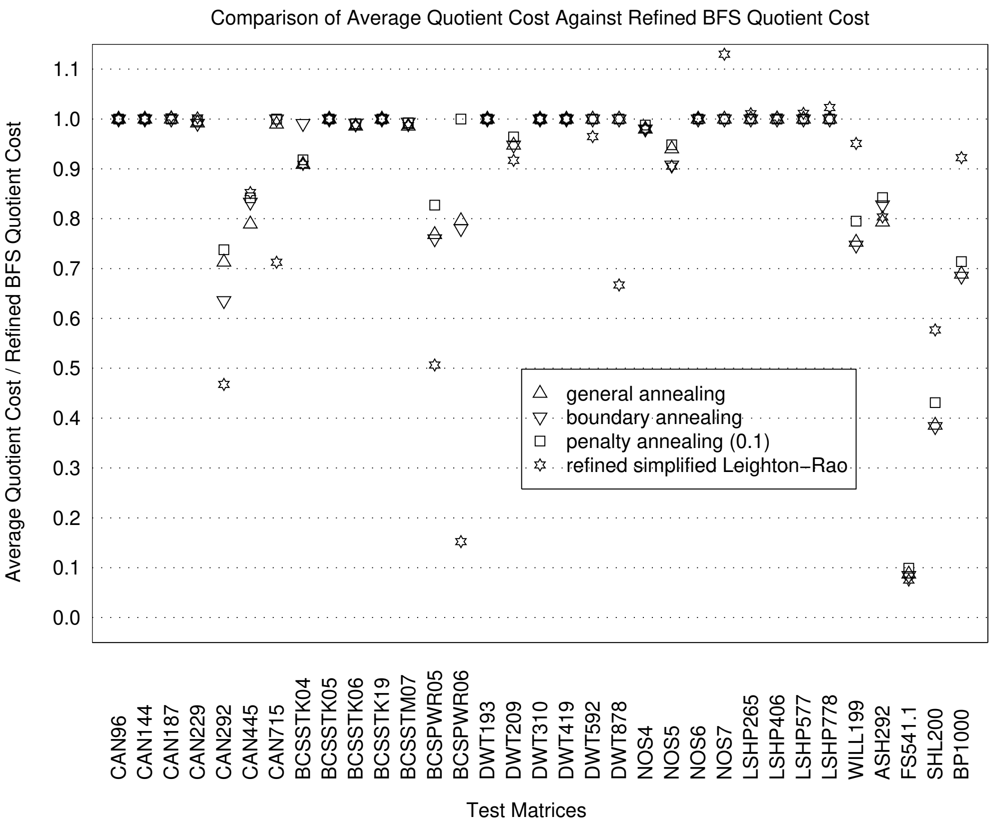


Figure 3.14: Comparison of the average quotient cost for each annealing approach and for the refined simplified Leighton-Rao construction to the refined BFS quotient cost.

not be further improved upon, and corroborates a similar observation made in Section 3.4.5 in comparing the Leighton-Rao construction and the refined BFS cuts.

As for the matrices for which simulated annealing showed an improvement over the refined BFS construction, we see that simulated annealing is clearly outperformed by the Leighton-Rao construction in some instances (eg. CAN292, CAN715, BCSPWR05 and BCSPWR06), but there are also instances where all three annealing approaches were better (eg. WILL199, SHL200, BP1000). Note that there is a wide range of vertex sizes in the graphs of both groups of matrices. We also see that the three annealing approaches have very similar average quotient costs. The only matrices where they differ significantly on are CAN292, BCSSTK04, BCSPWR05 and BCSPWR06, with the penalty function approach being the worst on three of those matrices.

As a further evaluation of simulated annealing, we ran the different approaches on those matrices for which the minimum quotient cost found by the refined simplified Leighton-Rao construction was better than the refined BFS quotient cost, using that best Leighton-Rao cut as the initial cut. There are 19 such matrices. It turned out that simulated annealing was able to further improve the cuts for 6 of those matrices. However the improvements were either small (under 1.5%) or the annealed cuts had very similar quotient costs to those obtained earlier with the refined BFS cuts as the starting cuts.

The mixed results that we have observed for simulated annealing show that it can be better than the theoretically superior Leighton-Rao construction. The results are however far from conclusive, and further experimentation with the annealing approaches and parameters is necessary to see if simulated annealing can be consistently better. We note that Lang and Rao [55] have performed a more detailed empirical study on low quotient cost edge cuts, where they compared a local search heuristic (the Fiduccia-Mattheyses heuristic [23]) and a construction method similar in spirit to the Leighton-Rao construction. In that study, they observed that the various methods produced reasonable edge cuts for 1000-vertex geometric graphs, and that it was only on larger test graphs with 3000 or more vertices that the Leighton-Rao-like construction was clearly superior. It is conceivable that a similar situation exists for simulated annealing on vertex cuts.

We conclude this section with some remarks on the running time and on the cut refinement postprocessing step. The running time for simulated annealing was close to that taken by the Leighton-Rao construction (with the time taken to solve the associated concurrent flow included). For all three approaches, the times taken were usually between 50% to 200% of that for the Leighton-Rao construction, although there were some runs that took 10 times as long. However, our implementation is not optimized, and these running times can be greatly reduced with a more careful implementation.

As stated earlier, we applied the cut refinement procedure as a postprocessing step on the best cut found by simulated annealing. In the discussion of this procedure in Section 3.4.4, it was observed that cut refinement is not "monotone" in behavior, and so it should be applied to each cut that is examined rather than only on the final cut produced. It seems natural that we should also refine each cut that is observed by simulated annealing, and then output the best refined cut. However, the statistics from the postprocessing phase indicated that cut refinement only improved 0.6% of the best cuts found by simulated annealing. This low figure is not surprising given that some of the vertex moves considered by the Metropolis algorithm are *identical* to those that occur during cut refinement. Thus the refinement procedure is already implicitly present in simulated annealing. While it is possible that directly applying cut refinement within the Metropolis algorithm can lead to better performance, we did not explore this possibility.

3.6 Approximately Optimal Balanced Cuts

When cuts are used in a divide-and-conquer algorithm, as in the elimination ordering algorithm of Chapter 4, one would like the components that remain after removing the cut to be roughly equal in size. This ensures that the recursive subproblems have about the same size and so the recursion tree cannot become overly lop-sided. We formalize the desired property by defining the notion of balancedness, and give an algorithm for finding approximately optimal balanced vertex cuts.

Given a vertex cut (A, B, C) and a balance factor $\beta \geq \frac{1}{2}$, we say that the cut is *β-balanced* if the larger component after the removal of the cut has at most βn vertices, i.e. $\max(|A|, |B|) \leq \beta n$, or equivalently, $\min(|A|, |B|) + |C| \geq (1 - \beta)n$. We also call (A, B, C) a $(1 - \beta, \beta)$ separator. The cost of the cut is $c(C)$, where as usual, c is a capacity function on the vertices of the graph. For the unit-capacity case, the cost of the cut is just the size of C, and we also refer to it as the size of the cut. An *optimal* β-balanced cut is a β-balanced cut of lowest cost.

Our definition of β-balancedness differs slightly from that given by Leighton and Rao [57], which follows directly from the usual definition of balancedness for edge cuts when applied to the standard transformation described in Section 3.1.2 for converting a vertex cut problem on an undirected graph into an edge cut problem on a directed graph. Specifically, they define a vertex cut (A, B, C) to be β-balanced if $\max(|A|, |B|) + \frac{1}{2}|C| \leq \beta n$. Clearly the Leighton-Rao definition is stronger than the one we have given here. However, within the context of recursive algorithms where the subproblems are based on the components, our balancedness measure, which does not include $|C|$, is more natural. Note that under both definitions of balancedness, a graph always has a $\frac{1}{2}$-balanced vertex cut.

In two variants of the elimination ordering algorithm, described in Sections 4.4 and 4.6, the recursive subproblems are formed by the components in conjunction with the cut. The natural β-balancedness measure there for a cut (A, B, C) would require that $\max(|A|, |B|) + |C| \leq \beta n$. This measure is however not valid, since not all graphs have $\frac{1}{2}$-balanced cuts under it, or for that matter, β-balanced cuts for any fixed $\frac{1}{2} \leq \beta < 1$.

In the rest of this section, we will examine two algorithms for finding balanced cuts. The first algorithm, due to Leighton and Rao, finds β-balanced cuts that have approximately optimal cost. The second algorithm is a heuristic that is used in the popular sparse matrix package SPARSPAK [35]. Finally, we present experimental results comparing the balanced cuts found by these two constructions.

3.6.1 The Leighton-Rao Balanced Cut Construction

The Leighton-Rao balanced cut algorithm constructs a β-balanced cut whose cost is approximately optimal. The cost of the constructed cut is however not measured relative to the cost of an optimal β-balanced cut, but is instead compared to that of an optimal cut for a lower balance factor.

Theorem 3.9 (Leighton & Rao [57]) *Let $\beta > \beta' \geq \frac{1}{2}$ and $\beta \geq \frac{2}{3}$. There exists a polynomial time algorithm that finds a β-balanced vertex cut whose cost is within an $O(\log n)\frac{1}{\beta-\beta'}$ factor of the cost of an optimal β'-balanced vertex cut.*

It is an open question as to whether there exists a polynomial time algorithm which finds a β-balanced cut whose cost is within an $O(\log n)$ factor of the cost of an optimal β-balanced cut. Note also that Theorem 3.9 requires that the desired balance factor be at least $\frac{2}{3}$. This is because the algorithm might be unable to produce any cut with lower balance factor.

The Leighton-Rao balanced cut algorithm given below constructs an approximate optimal β-balanced cut by repeatedly finding approximately optimal flux cuts and removing them from the underlying graph. These cuts are found by the Leighton-Rao sparse cut construction described in Section 3.2, and their quotient costs ensure that the cost of the cut is low. By aggregating a sufficient number of sparse cuts, the algorithm is able to produce a cut where both of its associated components are not too large. Throughout the algorithm, C_{tot} contains the vertices in the cut while A_{tot} contains the vertices in the larger component. We assume that the input graph is connected.

<u>Balanced Leighton-Rao</u>

$\hat{V} \leftarrow V, \quad A_{tot} \leftarrow \emptyset, \quad C_{tot} \leftarrow \emptyset$

while $|A_{tot}| + |C_{tot}| < (1-\beta)n$
    if the subgraph $G(\hat{V})$ induced by $\hat{V}$ is connected **then**
        find an approximate node flux cut (A, B, C) in $G(\hat{V})$, with $|A| \leq |B|$
    else
        partition $\hat{V}$ into two subsets A and B with $0 < |A| \leq |B|$,
            where A and B are disconnected from each other in $G(\hat{V})$,
            and let $C \leftarrow \emptyset$
    endif

    $A_{tot} \leftarrow A_{tot} \cup A, \quad C_{tot} \leftarrow C_{tot} \cup C, \quad \hat{V} \leftarrow B$
endwhile

output $(A_{tot},\ V \setminus (A_{tot} \cup C_{tot}),\ C_{tot})$

The above algorithm is easily adapted to produce a balanced cut following the Leighton-Rao definition of β-balancedness by modifying the condition in the while loop to $|A_{tot}| + \frac{1}{2}|C_{tot}| < (1-\beta)n$. The modified while loop might then terminate in a fewer number of iterations. Note also that although there is a theoretical possibility of the remaining subgraph induced by $\hat{V}$ being disconnected, it did not occur in the runs that we performed.

Let $\bar{A}_{tot}$ and $\bar{C}_{tot}$ be the values of A_{tot} and C_{tot} at the end of the algorithm, and $\bar{B}_{tot} = V \setminus (\bar{A}_{tot} \cup \bar{C}_{tot})$. Clearly $\bar{C}_{tot} \neq \emptyset$ since the input graph is assumed to be connected, and $\bar{C}_{tot}$ separates $\bar{A}_{tot}$ from $\bar{B}_{tot}$. It is possible that $(\bar{A}_{tot}, \bar{B}_{tot}, \bar{C}_{tot})$ is actually a multi-way cut instead of a 2-way cut, i.e. removing $\bar{C}_{tot}$ from the graph results in more than 2 components. In that case, $\bar{A}_{tot}$ and/or $\bar{B}_{tot}$ are each composed of a union of some of those components, and it is still true that each component has at most βn vertices.

Proof of Correctness

We first show that $(\bar{A}_{tot}, \bar{B}_{tot}, \bar{C}_{tot})$ is β-balanced and then prove that it has approximately optimal cost. These two properties of the constructed cut hold independently of each other, with the first following from the terminating condition of the while loop and the second due to the approximate optimality of the flux cuts used. Together they establish Theorem 3.9. The algorithm can therefore be modified to aggregate other types of cuts to form a balanced cut with different optimality properties.

Lemma 3.10 *If $\beta \geq \frac{2}{3}$, the cut $(\bar{A}_{tot}, \bar{B}_{tot}, \bar{C}_{tot})$ is β-balanced.*

Proof. Since the termination condition of the while loop guarantees that $|\bar{B}_{tot}| \leq \beta n$, we only need to show that $|\bar{A}_{tot}| \leq \beta n$.

Let $\tilde{A}_{tot}$, $\tilde{C}_{tot}$ and $\tilde{V}$ be the values of A_{tot}, C_{tot} and $\hat{V}$ at the start of the last iteration, and $(\tilde{A}, \tilde{B}, \tilde{C})$ be the approximate flux cut found in that iteration. Observe that $\tilde{A}_{tot}$ and $\tilde{C}_{tot}$ together contain the vertices that have previously been removed, i.e. $\tilde{A}_{tot} \cup \tilde{C}_{tot} \cup \tilde{V} = V$. In addition, we know that $|\tilde{A}_{tot}| + |\tilde{C}_{tot}| < (1-\beta)n$, $|\tilde{A}| \leq \frac{1}{2}|\tilde{V}|$ and $\bar{A}_{tot} = \tilde{A}_{tot} \cup \tilde{A}$. Thus

$$\begin{aligned}
|\bar{A}_{tot}| &= |\tilde{A}_{tot}| + |\tilde{A}| \\
&\leq |\tilde{A}_{tot}| + \tfrac{1}{2}|\tilde{V}| \\
&< \tfrac{1}{2}(|\tilde{A}_{tot}| + |\tilde{C}_{tot}|) + \tfrac{1}{2}(|\tilde{A}_{tot}| + |\tilde{C}_{tot}| + |\tilde{V}|) \\
&< \tfrac{1}{2}(1-\beta)n + \tfrac{1}{2}n \\
&\leq \beta n,
\end{aligned}$$

with the last inequality being true precisely when $\beta \geq \frac{2}{3}$. ∎

Lemma 3.11 *The cost of the cut $(\bar{A}_{tot}, \bar{B}_{tot}, \bar{C}_{tot})$ is within an $O(\log n)\frac{1}{\beta - \beta'}$ factor of the cost of an optimal β'-balanced cut, where $\beta > \beta' \geq \frac{1}{2}$.*

Proof. Consider the beginning of one iteration of the algorithm. Let (A', B', C') be an optimal β'-balanced cut, and assume without loss of generality that $|A' \cap \hat{V}| \leq |B' \cap \hat{V}|$. We will bound the capacity $c(C)$ of the cut found in this iteration. For the case when the subgraph $G(\hat{V})$ induced by $\hat{V}$ is not connected, $c(C)$ is just zero. We therefore assume that $G(\hat{V})$ is connected, and so $C' \cap \hat{V} \neq \emptyset$, for otherwise $G(\hat{V})$ would have two components $A' \cap \hat{V}$ and $B' \cap \hat{V}$. Thus $(A' \cap \hat{V}, B' \cap \hat{V}, C' \cap \hat{V})$ is a vertex cut in $G(\hat{V})$, and we know by the approximate optimality of (A, B, C) that

$$\frac{c(C)}{|A| + \frac{1}{2}|C|} \leq O(\log n) \frac{c(C' \cap \hat{V})}{|A' \cap \hat{V}| + \frac{1}{2}|C' \cap \hat{V}|}.$$

Using the relations $|A_{tot}| + |C_{tot}| < (1-\beta)n$, $|A'| + |C'| \geq (1-\beta')n$, and the fact that $V = \hat{V} \cup A_{tot} \cup C_{tot}$, we can lower bound the denominator on the right-hand-side of the above inequality:

$$\begin{aligned} |A' \cap \hat{V}| + \tfrac{1}{2}|C' \cap \hat{V}| &\geq \tfrac{1}{2}(|A' \cap \hat{V}| + |C' \cap \hat{V}|) \\ &= \tfrac{1}{2}|(A' \cup C') \setminus (A_{tot} \cup C_{tot})| \\ &\geq \frac{\beta - \beta'}{2} n. \end{aligned}$$

It follows then that

$$\begin{aligned} c(C) &\leq O(\log n)(|A| + \tfrac{1}{2}|C|) \frac{c(C')}{|A' \cap \hat{V}| + \frac{1}{2}|C' \cap \hat{V}|} \\ &\leq \frac{O(\log n)}{\beta - \beta'} \frac{(|A| + \frac{1}{2}|C|)}{n} c(C'). \end{aligned}$$

Having related the capacity of the cuts at each iteration to the capacity of an optimal β'-balanced cut, we can bound the capacity of the cut that the algorithm constructs. Let $(A_1, B_1, C_1), \ldots, (A_j, B_j, C_j)$ be the approximate flux cuts found in the iterations. Then

$$\begin{aligned} c(\bar{C}_{tot}) &= \sum_{i=1}^{j} c(C_i) \\ &\leq \frac{1}{\beta - \beta'} O(\log n)\, c(C') \sum_{i=1}^{j} \frac{|A_i| + \frac{1}{2}|C_i|}{n} \\ &\leq \frac{1}{\beta - \beta'} O(\log n)\, c(C'), \end{aligned}$$

since $(A_1 \cup C_1), \ldots, (A_j \cup C_j)$ are pairwise disjoint. ■

3.6.2 A Balanced Cut Heuristic

The SPARSPAK sparse matrix package [35] provides an elimination ordering subroutine that applies nested dissection to general undirected graphs. The cuts that are used for this process are found by a fast heuristic introduced by George and Liu [32] that is based on breadth-first search (BFS). There are no guarantees on the balance of the cut found nor on its cost. We briefly describe in this section the ideas behind this heuristic. Since the base cases in SPARSPAK's nested dissection consists of complete graphs, we assume that the input graph G is connected but not complete.

We first make some preliminary definitions. For a vertex v, we define its *eccentricity* $e(v)$ to be the height of the BFS tree rooted at v. Thus $e(v)$ is the maximum hop distance between v and any other vertex. The diameter of the graph is therefore the maximum eccentricity over all vertices, and a vertex whose eccentricity is equal to the diameter is called a *peripheral* vertex. Peripheral vertices are easily found in $O(n(m+n))$ time. A vertex whose eccentricity is close to the diameter is referred to as a *pseudoperipheral* vertex.

The George-Liu cut heuristic first finds a pseudoperipheral vertex using a practically linear time algorithm that we shall describe later, and then performs breadth-first search from that vertex. The desired cut is formed by the non-leaf vertices at the middle level of the BFS tree (or the middle level further from the root if there are an even number of levels). Removing this cut gives components containing the vertices in the subtrees below the separating level and the truncated subtree above the separating level. Since the BFS tree has close to maximum height, it is likely that these subtrees will have about the same number of vertices.

To find a pseudoperipheral vertex, George and Liu [32] used a variant of the algorithm first proposed by Gibbs, Poole and Stockmeyer [36]. The algorithm is iterative and examines a sequence of BFS trees. The first BFS tree T_1 is grown from some arbitrarily chosen vertex v_1. Given one BFS tree T_i, the next tree T_{i+1} is constructed by picking a vertex v_{i+1} of minimum degree from the lowest level of T_i and then performing breadth-first search from v_{i+1}. Note that the height of T_{i+1} is at least the hop distance between v_i and v_{i+1}, which coincides with the height of T_i. The algorithm stops when the heights of the last two BFS trees are identical, and the root of the last BFS tree is output as the desired pseudoperipheral vertex.[7] In practice, the algorithm examines only a few BFS trees, and so runs in linear time.

3.6.3 Experimental Results for Balanced Cut Constructions

We compare in this section the performance of the Leighton-Rao balanced cut construction and of the SPARSPAK cut heuristic in practice. As in previous empirical evaluations (cf. Sections 3.4 and Section 3.5.3), our test cases are the graphs of 33 matrices from the Harwell-Boeing collection of sparse matrices [18] [19]. The sizes

[7] The original Gibbs-Poole-Stockmeyer algorithm [36] examines the BFS trees of each vertex in the lowest level, in order of increasing degree, till one is found that has greater height. If all the BFS trees are of the same height as the current one, the algorithm stops.

of these matrices were given in Table 3.1 of Section 3.4.1. For each input graph, we performed 30 runs of the Leighton-Rao balanced cut construction with a balance factor of $\frac{2}{3}$. Since the SPARSPAK heuristic is two-or-more orders of magnitude faster than the Leigthon-Rao construction, we ran the heuristic 1000 times for each input graph. These 1000 runs presented the deterministic implementation of the SPARSPAK heuristic with the original input graph and 999 vertex-permuted versions of it.

Comparison of Cut and Largest Component Sizes

Figure 3.15 shows the sizes of the cuts found by the Leighton-Rao construction and by the SPARSPAK heuristic. In that figure, we have plotted for each matrix the range of cut sizes found, expressed as a fraction of the number of vertices in the graph. For example, for the fifth-to-last matrix WILL199, the ranges for both constructions overlap, with the Leighton-Rao construction producing cuts having between 33% to 38% of the vertices, while those found by SPARSPAK contained between 26% to 44% of the vertices. The corresponding plot for the sizes of the

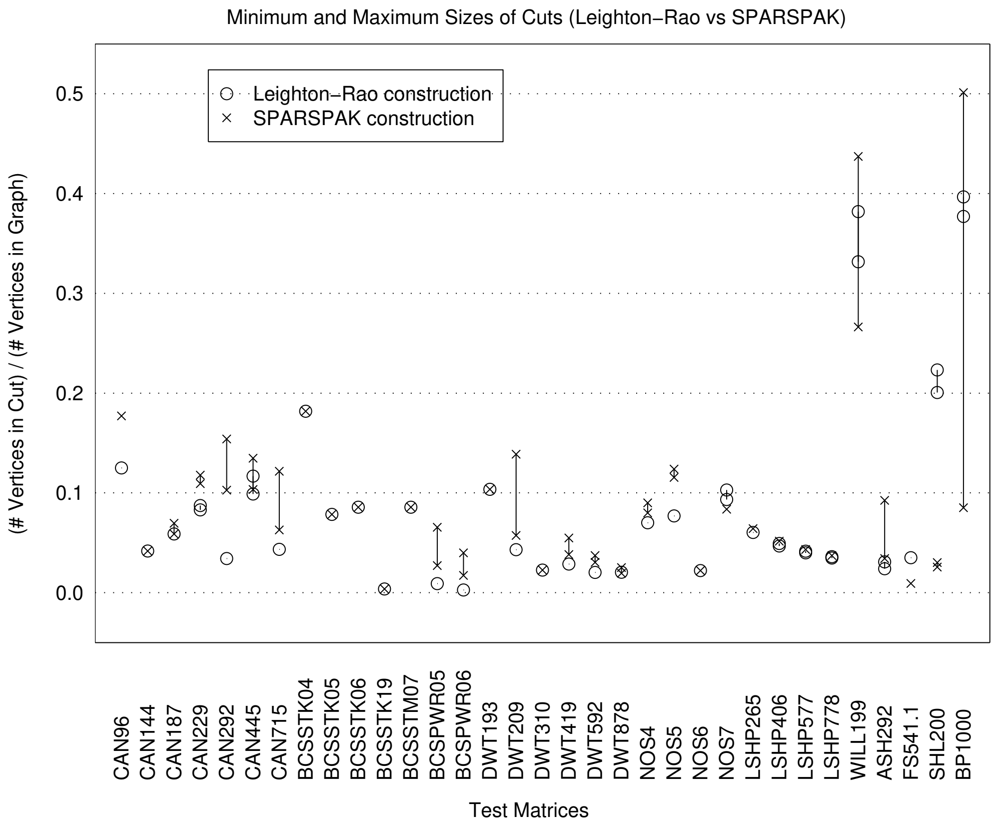


Figure 3.15: Comparison of the range of cut sizes found by the Leighton-Rao construction and by the SPARSPAK heuristic. The cut sizes are expressed as a fraction of the number of vertices in the graph.

largest component is given in Figure 3.16. In that figure, we show the ranges of the largest component sizes associated with the removal of the Leighton-Rao and SPARSPAK cuts, where the component sizes are expressed as a fraction of the number of vertices in the graph.

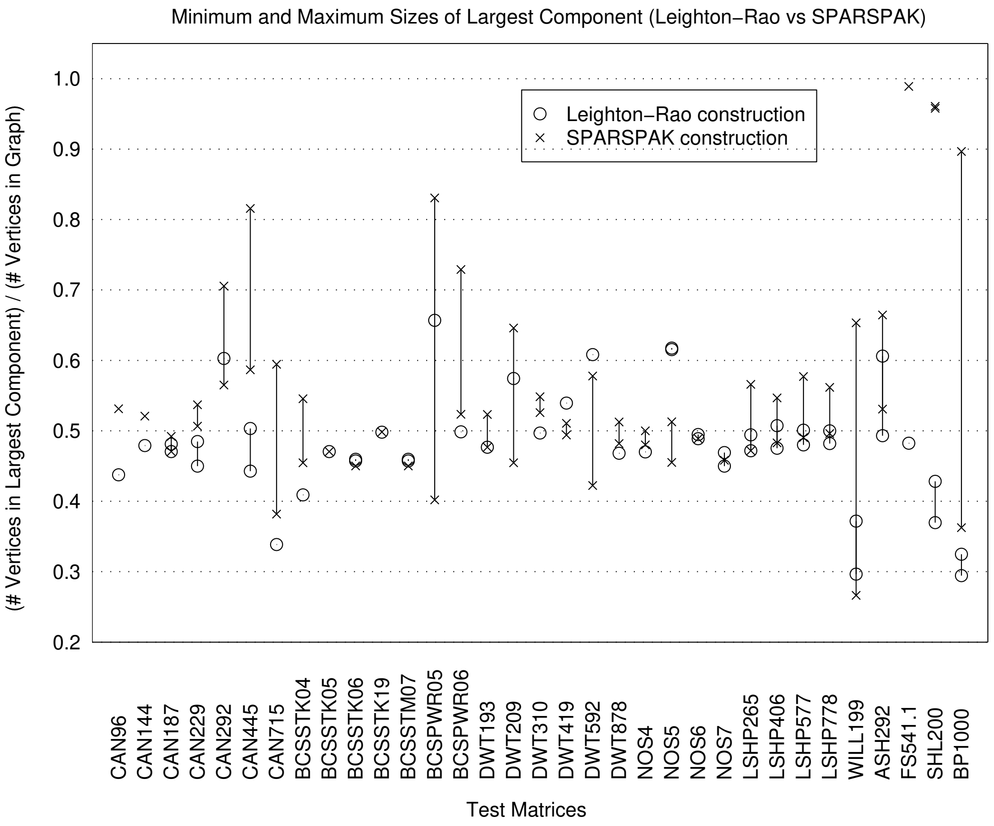


Figure 3.16: Comparison of the range of sizes of the largest component associated with the cuts found by the Leighton-Rao construction and by the SPARSPAK heuristic. The component sizes are expressed as a fraction of the number of vertices in the graph.

We see from the figures that the multiple runs of the Leighton-Rao construction on each input produced cuts and associated largest components of very similar sizes. Indeed, for 27 of the 33 matrices, there was practically only one distinct cut size found. These 27 matrices included the 20 matrices for which there was practically one distinct largest component size. For some of these matrices, the number of distinct cuts found was 3 or fewer, while for others, there were at least 10 distinct cuts among the 30 found. A similar picture is present for the multiple runs of the SPARSPAK heuristic. For about half of the matrices, SPARSPAK produced practically one cut size, with this group of matrices containing those for which there was practically one largest component size.

In general, the range of cut sizes found by the Leighton-Rao construction was small, with only 5 instances (CAN445, NOS7, WILL199, ASH292, SHL200) where the maximum-to-minimum ratio of cut sizes exceeded 1.1, but was less than 1.3. We see also a corresponding narrow range for the largest component sizes. These tight ranges are due to the guaranteed balancedness of the Leighton-Rao cuts. In contrast, the range of cut sizes found by SPARSPAK for a matrix can be quite large: the maximum-to-minimum ratio for the matrices CAN715, BCSPWR05, BCSPWR06, DWT209, WILL199, ASH292 and BP1000 all exceeded 1.5, with the worst matrix BP1000 having a ratio of 5.9. These large ranges reflect the lack of a balancedness guarantee for the SPARSPAK heuristic. Indeed for those matrices and several others, there were large variations in the sizes of the associated largest components.

Figure 3.15 also shows that there was high overlap in the matrices for which both the Leighton-Rao construction and the SPARSPAK heuristic found practically only one distinct cut size. Curiously, for many of those common matrices (eg. CAN144 and BCSSTK04), the two methods produced identical or virtually identical cut sizes, though they did not necessarily find identical cuts. Apart from those special matrices, we see that the *minimum* size of the cuts found by SPARSPAK is usually greater than the *maximum* (or only distinct) size of the Leighton-Rao cuts. The significant exceptions occur on the matrices CAN445, WILL199 and BP1000, where the Leighton-Rao and SPARSPAK ranges overlap, and for the matrices FS541.1 and SHL200, where SPARSPAK was always better than Leighton-Rao. However, for those 5 matrices, the sizes of the associated largest component for SPARSPAK were usually much larger than those for the Leighton-Rao construction.

Overall, the largest component associated with a Leighton-Rao cut typically contained between 40% to 60% of the vertices in the test graph. For the instances where the largest component had less than 40% of the vertices (CAN715, WILL199, SHL200 and BP1000), it turned out that the Leighton-Rao cuts were multi-way instead of 2-way, i.e. the removal of those cuts divided the remaining graph into more than 2 connected components. In sharp contrast, there were several instances where the largest component associated with a SPARSPAK cut contained more than 80% of the vertices, and the largest component size actually reached 99% for the matrix FS541.1. These extreme largest component sizes clearly show the lack of a balancedness guarantee for the SPARSPAK heuristic.

Finally, we observe from Figure 3.15 that with the exception of the matrices BCSSTK04, WILL199, SHL200 and BP1000, the graphs of our test matrices had small $\frac{2}{3}$-balanced cuts typically containing no more than 10% of the vertices of the graph, as found by the Leighton-Rao construction.

Number of Components and Ratio of Component Sizes

We now give statistics on the number of connected components resulting from the removal of the Leighton-Rao and SPARPSAK cuts, and on the ratio of those component sizes. For the Leighton-Rao construction, there were 20 matrices for which the cuts found were always 2-way, 10 matrices where they were always multi-way, and 3 matrices with a mixture of 2-way and multi-way cuts. The multi-way cuts usually disconnected the graph into 5 or fewer components, although those for the matrix CAN292 left 19 components, and those for the matrix SHL200 and BP1000 left between 9 to 29 and between 30 to 65 components respectively. The proportions of matrices with only 2-way, only multi-way and mixed cuts found by SPARSPAK are similar. There were some matrices though for which Leighton-Rao always found 2-way cuts while SPARSPAK consistently gave multi-way cuts with a large number of components, and vice versa.

As for the ratio of component sizes, the size of the smaller component associated with a 2-way Leighton-Rao cut was at least half that of the larger component, and exceeded 80% of the larger component size in most of the cases. When the Leighton-Rao cut was multi-way, the size of the second largest component was as least a quarter of that of the largest component. The corresponding figures for SPARSPAK are lower, again reflecting the lack of a balancedness guarantee. When the SPARSPAK cut was 2-way, the smaller component size was at least 65% that of the larger component size in most of the cases. There were instances though when the ratio was much lower, including the pathological matrix FS541.1 that we observed earlier, where the larger component already contained 99% of the vertices of the graph. For multi-way SPARSPAK cuts, the size of the second largest component was as small as 10% of that of the largest component, even when the largest component contained no more than two-thirds of the vertices of the graph.

Other Statistics for the Leighton-Rao Balanced Cut Construction

We conclude our discussion on the experimental results by giving further statistics for the Leighton-Rao balanced cut construction.

Number of Flux Cuts Aggregated. Recall from Section 3.6.1 that the balanced cut construction aggregates approximately optimal flux cuts to achieve the desired balance factor. Those flux cuts are found using the refined simplified Leighton-Rao construction described in Section 3.4.4. It turned out that only one flux cut was used in almost all the runs, with the remaining runs using only 2 flux cuts. The results are not surprising since we saw in Section 3.4.4 that almost all of the flux cuts found by the refined simplified construction were already $\frac{2}{3}$-balanced.

Balancedness under the Leighton-Rao Definition. As indicated in Section 3.6.1, our definition of balancedness differs from that given by Leighton and Rao [57]. Specifically, we say that a vertex cut (A, B, C) is β-balanced if $\max(|A|, |B|) \leq \beta n$, while Leighton and Rao impose the stricter requirement that $\max(|A|, |B|) + \frac{1}{2}|C| \leq \beta n$. The distinctions between the two definitions turned out to be irrelevant for our experiments, for all the cuts found were also $\frac{2}{3}$-balanced according to the Leighton-Rao definition. Interestingly, the cuts were 0.7-balanced under the definition that includes the entire size of the cut, i.e. the total number of vertices in the cut and its associated larger component never exceeded 70% of the number of vertices in the test graph.

3.7 Balanced Low-Deficiency Cuts

We discussed in the previous section the problem of finding a balanced cut with approximately minimum size, or more generally, with approximately minimum capacity. This objective is desirable in divide-and-conquer algorithms since the size of the cut determines the extent of interaction between it and the remaining components, and this interaction introduces additional cost when solutions to the recursive subproblems are combined together to form the final solution.

However, for some versions of the recursive elimination ordering algorithm that we shall consider in Chapter 4, the size of the cut is irrelevant. Instead the appropriate measure of a cut is its *deficiency*, which is the number of pairs of nonadjacent vertices in the cut. In general, the graph elimination process fills in those missing edges, and turns the subgraph induced by the cut vertices into a clique. We point out that in those versions of the ordering algorithm, the recursion occurs on subproblems defined by the union of the cut vertices with the components that remain after the removal of the cut vertices.

There are no known efficient algorithms for producing balanced cuts with minimum or approximately minimum deficiency. Note that the deficiency measure is not additive: given two disjoint vertex subsets C_1 and C_2, the deficiency of $C_1 \cup C_2$ is in general larger than the sum of the individual subset deficiencies. Therefore, the Leighton-Rao technique of aggregating sparse cuts to form balanced cuts is not easily applicable here. We instead apply simulated annealing directly to the balanced low-deficiency cut problem in Section 3.7.1, and present experimental results for it in Section 3.7.2.

3.7.1 Simulated Annealing for Low-Deficiency Cuts

In Section 3.5.1, we presented a generic framework for simulated annealing and then applied it to the problem of finding sparse vertex cuts in Section 3.5.2. We use the same framework here to find low-deficiency vertex cuts. As in the other application of simulated annealing, we do not attempt to find good values for the various annealing parameters. Instead we use the parameter values listed in Section 3.5.2.

Annealing Approach and Cost Function

As we discussed in Section 3.5.2, there are three approaches for mapping vertex cuts onto configurations: letting the space of configurations be general vertex cuts, expanding the configuration space to include infeasible 3-way partitions but penalizing for crossing edges in the cost function, or restricting the configurations to vertex cuts defined by the boundary of a subset of vertices. The experimental results in Section 3.5.3 showed that these approaches essentially have similar performance. Since the penalty function approach involves a cost function that is parameterized by a penalty factor, we do not consider it as it would complicate the cost function for balanced deficiency cuts. Of the other two approaches, we arbitrarily select the boundary cut approach.

We now consider the possibilites for the cost function. Intuitively, we want to assign a low cost to a cut (A, B, C) that has low deficiency and good balance, i.e. with A and B having roughly equal size. Since the elimination ordering algorithm (cf. Section 4.6) in which we use low-deficiency cuts includes the cut vertices in the subproblems, it seems desirable to also reflect this inclusion in the cost function. A candidate cost function meeting the stated criteria is the following:

$$cost(A, B, C) = def(C) - \text{BALWT} \cdot (|A| + |C|)(|B| + |C|),$$

where $def(C)$ denotes the deficiency of C and BALWT is a positive weight. Unfortunately, the quantity $(|A| + |C|)(|B| + |C|)$ is trivially (almost) maximized by letting $|C|$ be as large as possible, contrary to our intuition of a balanced cut. This side-effect is present for any product $(|A| + \gamma|C|)(|B| + \gamma|C|)$ with $\gamma \geq \frac{1}{2}$. We therefore omit the size of C from the "balancedness" term of the next cost function

$$abscost(A, B, C) = def(C) - \text{BALWT} \cdot (|A||B|).$$

We shall refer to the above function as the *absolute deficiency cost function*. Note that the absolute deficiency cost implicitly favors small cuts since they have lower deficiencies. In addition, a small BALWT value places greater emphasis on the cut deficiency, while a high value emphasizes the balancedness of the cut. We choose a suitable value for BALWT by trial and error, as described in the next section.

Instead of using the deficiency of a cut in the cost function, we can look at its *fractional deficiency*, which is the fraction of edges that are missing. For $|C| > 1$, the fractional deficiency $fractdef(C)$ is the deficiency divided by $|C|(|C| - 1)/2$, while the fractional deficiency of a single-vertex cut, i.e an articulation point, is defined to be zero. The corresponding *fractional deficiency cost function* is:

$$fractcost(A, B, C) = fractdef(C) - \text{BALWT} \cdot (|A||B|/n^2),$$

where we scale the product $|A||B|$ accordingly. Unlike the deficiency measure, the fractional deficiency is unbiased about the number of vertices in the cut, and so the two cost functions are qualitatively different.

The two cost functions that we have proposed do not incorporate any characteristics of the underlying graph. For example, we can introduce an expression dependent on the average vertex degree in the cost function to reflect the sparsity of the graph. This is not useful, however, for our set of test graphs, which are those of sparse matrices. Indeed, for those graphs, the density of edges is at most 10%, and so most cuts will have fractional deficiency exceeding 90%.

Initial Vertex Cuts

The initial vertex cut for simulated annealing is found in a similar manner to that described in Section 3.5.2. Specifically, we perform breadth-first search from each vertex, and potentially look at two groups of cuts. The first group of cuts is defined by the boundaries of the regions grown, while the second group is obtained by applying the cut refinement procedure of Section 3.4.4 to those cuts in the first group. Since refining a cut lowers its absolute deficiency cost, we only need to examine the cuts in the second group for that cost function. For the fractional deficiency cost function, it is necessary to examine both groups of cuts.

Temperature Settings

To determine the initial temperature, we start from a guess and increase or decrease it by a factor of 0.8 until the percentage of moves made in the first invocation of the Metropolis algorithm is between 40% and 70%. By experimentation, we determined that a good guess to use in conjunction with the absolute deficiency cost function is the number of vertices in the graph, while that for the fractional deficiency cost function is the constant 0.3.

We also limited the number of temperature steps tried to 500, following the experimental observations in Section 3.5.3. To further reduce the running time, we checked if the best observed cost has decreased at every 50 temperature steps, and stop if no improvement has been made.

3.7.2 Experimental Results for Low-Deficiency Cuts

We present an empricial evaluation of the two different cost functions with various values of BALWT in this section. Our goal is to identify a combination for use by the elimination ordering algorithm in Section 4.6. The selected combination should not exhibit pathological behavior, and its performance should be comparable to or better than that of other well-behaved combinations. As in the previous experiment on simulated annealing (cf. Section 3.5.3), we used the set of 33 matrices from the Harwell-Boeing sparse matrix collection [18] [19]. The sizes of these matrices were listed in Table 3.1 in Section 3.4.1. For the absolute deficiency cost function, we used the values 3, 1, 0.3 and 0.1 for the parameter BALWT. We also tried the higher values 30 and 10, but they did not improve upon the values of 3 or 1 in terms of deficiency, fractional deficiency or total size of the cut and the largest component. Similarly, for the fractional deficiency cost function, we used the parameter values 30, 10, 3 and 1, and rejected the higher values of 300 and 100. For each cost function, value of BALWT and each matrix, we performed 30 random runs.

It turned out that for the fractional deficiency cost function and a value of 1 for BALWT, the average size of the largest component associated with a cut exceeded 90% of the number of vertices in the input graph on 30 of the 33 matrices. Similarly, the average largest component size exceeded 85% on 18 matrices when BALWT was set at 3. We therefore omit those two values from further consideration. As for the absolute deficiency cost function, the BALWT values of 0.1 and 0.3 also exhibited pathological behavior, but to a lesser degree. Specifically, on three of the test matrices, the average largest component sizes associated with those two BALWT values exceeded 85% of the number of the vertices in the graph. The average absolute and fractional deficiencies of the cuts found with those two lower values were not significantly better than those found with the two higher values, except on the pathological matrices. So we also omit the two lower BALWT values in subsequent discussion.

In the rest of the discussion, we abbreviate the cost function and BALWT value combinations by *abs*@3, *abs*@1, *fract*@30 and *fract*@10. We first look at the average deficiency of the cuts found. There was a wide range of deficiencies among the cuts for the set of test matrices, with average values from 9 to about 20,000 (and some zero-deficiency cuts). As a fraction of the square of the number of vertices in the graph, the average deficiencies ranged from $8 \cdot 10^{-6}$ to 0.04. We therefore do not plot the actual values of the average deficiencies, but instead express them as a multiple of the minimum average deficiency of the four combinations. The plot is given in Figure 3.17. For example, for the last matrix BP1000, the *abs*@1 combination had the lowest average deficiency, while that for *abs*@3 was about 30% higher, and those for the fractional deficiency cost functions were about 50% higher. Note that on the matrix BCSSTK19, all four combinations produced an average deficiency of zero, and the same phenomenon occurred on the matrix SHL200 for *fract*@10. In addition, the average deficiency for *abs*@3 on the matrix FS541.1 is plotted off the scale, with its value indicated in parentheses. We see from the figure that the performance of the four combinations was virtually identical on many of the matrices. In addition, the *abs*@1 combination was usually better than or comparable to the *abs*@3 combination, except on the matrix BCSPWR05. This agrees with the heavier emphasis on deficiency in the cost function for the lower BALWT value. As for the fractional deficiency cost function, there were matrices (eg. BCSPWR05 and BCSPWR06) where the higher BALWT value was much better, and matrices (eg. NOS5 and FS541.1) with the opposite behavior. We crudely rank the four combinations by comparing the sum of their relative average deficiencies across the set of matrices. The totals for the combinations *abs*@3, *abs*@1, *fract*@30 and *fract*@30 were 36.7, 33.9, 34.7 and 33.5 respectively, from which we conclude that the *abs*@3 combination is the worst, while the *fract*@30 combination is slight worse than the *abs*@1 combination, which in turn in is slightly worse than the *fract*@10 combination.

We next look at the average fractional deficiency of the cuts produced by the four combinations. The plot of the actual values are given in Figure 3.18. We see again that there were many matrices on which the four combinations had virtually identical average fractional deficiencies. In fact the totals of the average fractional deficiencies across all matrices were very similar for all four combinations. The four totals were 27.43, 27.38, 27.41 and 27.08 respectively for the *abs*@3, *abs*@1, *fract*@30 and *fract*@10 combinations.

In our last comparison of the four combinations, we look at the average total sizes of the cut and the largest component. As remarked earlier, this statistic is appropriate since the elimination ordering algorithm based on balanced low-deficiency cuts recurses on subproblems defined by a component and the cut vertices. We plot this statistic in Figure 3.19, where it is expressed as a fraction of the number of vertices in the graph. We see from the figure that the combination *fract*@10 exhibited pathological behavior on the matrix SHL200, with almost all the vertices

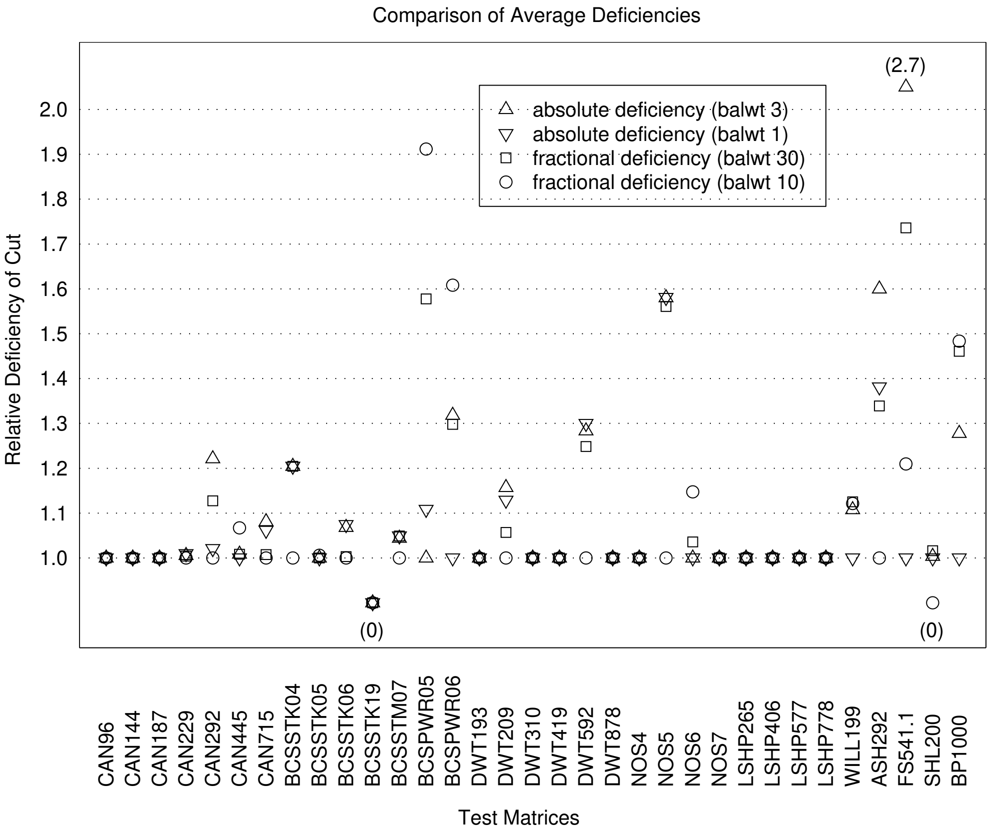


Figure 3.17: Comparison of the average deficiency of the cuts found by simulated annealing using the absolute and the fractional deficiency cost functions. The average deficiency for each cost function and each BALWT is expressed as a multiple of the minimum average deficiency among the four methods.

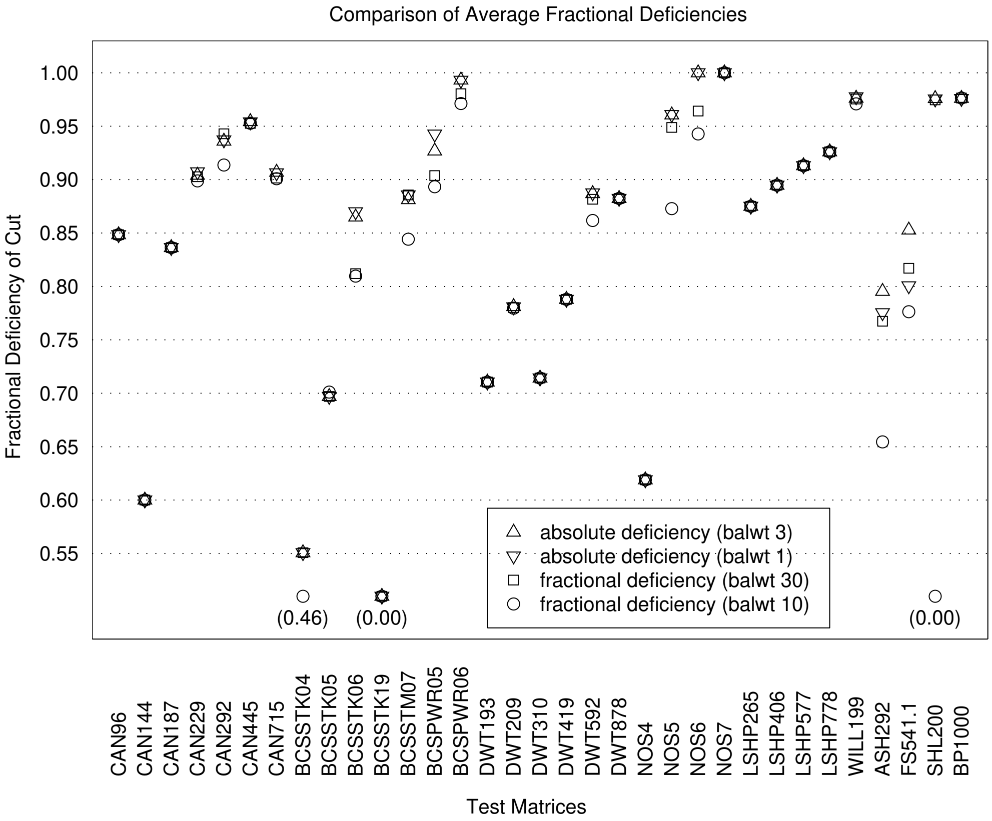


Figure 3.18: Comparison of the average fractional deficiency of the cuts found by simulated annealing using the absolute and the fractional deficiency cost functions.

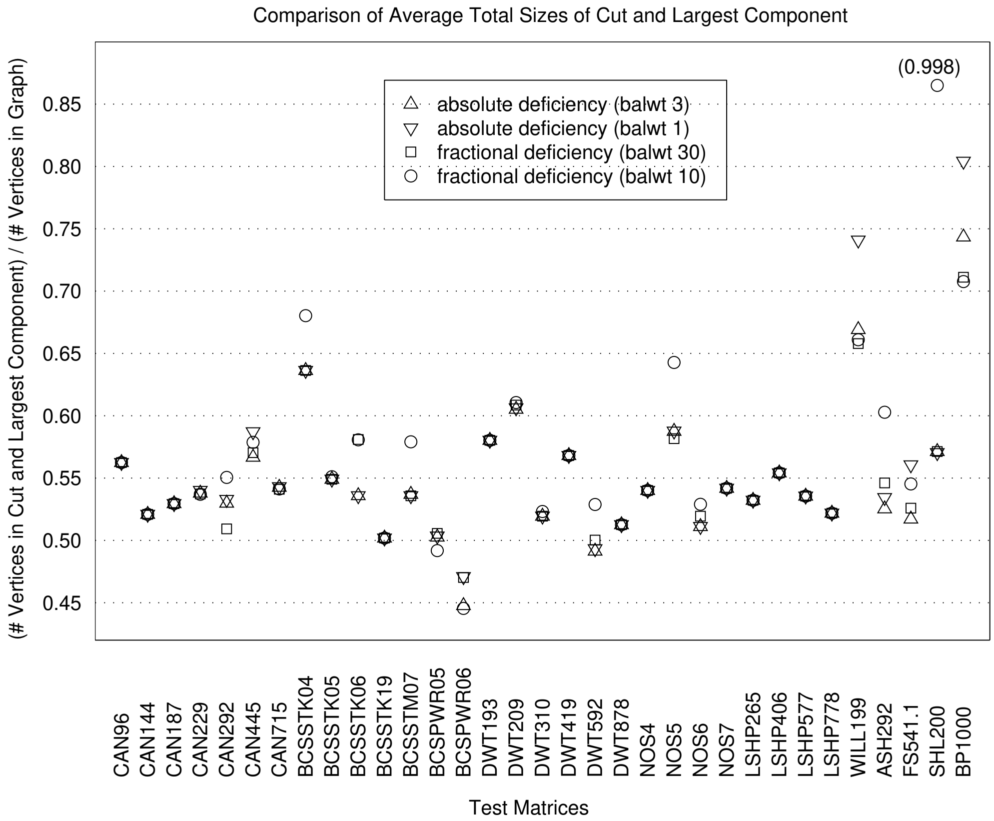


Figure 3.19: Comparison of the average total size of the cut and the largest component found by simulated annealing using the absolute and the fractional deficiency cost functions. The total sizes are expressed as a fraction of the number of vertices in the graph.

in the graph appearing in the largest component or the cut. We thus rule out that combination. Apart from that pathological case, we observe that the average total sizes of the cut and the largest component were almost always under 70%. This compares favorably with the experimental results in Section 3.6.3 for the Leighton-Rao balanced cut construction, in which we saw that the average total sizes of the cut and largest component never exceeded the 70% threshold.

We make a selection among the remaining three combinations by dropping the combinations that performed worst under the deficiency or the average total cut and component size criteria. For the latter statistic, the above figure shows that the abs@1 combination had the largest average total size for those instances where this average crossed the 70% threshold. We also saw earlier that the abs@3 combination was justifiably the worst with respect to average deficiency. Therefore we select the $fract$@30 combination for use in the elimination ordering algorithm.

Chapter 4

Nested Dissection Elimination Orderings

The class of nested dissection methods uses a divide-and-conquer approach to construct an elimination ordering. The vertices of the separator are numbered last, while those in the remaining components are numbered recursively. On graphs where special separators exist, such as planar graphs, those separators are used and the resulting ordering typically has provable bounds on its fill and operation count. On general graphs, the separators are usually found by heuristics in practice, such as the SPARSPAK heuristic described in Section 3.6.2. However, theoretically superior but computationally more intensive constructions, like the Leighton-Rao balanced separator construction of Section 3.6.1, can be used.

We first review the relation between elimination orderings and chordal graphs in Section 4.1, and describe in Section 4.2 an algorithm for finding an ordering with no fill-in when such an ordering exists. In Sections 4.3 and 4.4, we discuss the regular nested dissection method and one generalization of it. We then formulate in Section 4.5 a restricted version of the ordering problem, and describe a reduction of the restricted problem to the standard problem. The reduction forms the framework for a new nested dissection method that we examine in Section 4.6. Finally, we empirically evaluate the various nested dissection methods in Section 4.7.

4.1 Elimination Orderings and Chordal Graphs

We introduced in Section 1.3 a graph theoretic interpretation for elimination. As we saw there, the formulation provides a convenient way for describing elimination ordering algorithms. We now consider in greater depth this formulation, and show an equivalence between the graphs of matrices that possess an ordering with no fill-in and chordal graphs, a class of widely studied graphs.

We first review the graph elimination process. Let $G = (V, E)$ be the graph of a symmetric positive definite matrix A. We assume without loss of generality that the matrix A is *irreducible*, i.e. there is no permutation of the rows and columns of A that makes it into a block diagonal matrix, for otherwise elimination can be performed independently on the blocks. The irreducibility condition is equivalent to G being connected. We therefore assume throughout this chapter that G is connected.

Let $\pi : V \leftrightarrow \{1, \ldots, n\}$ be a vertex ordering. Recall from Section 1.3 that eliminating the vertices in the order given by π produces the following sequence of elimination graphs $G_\pi^1, \ldots, G_\pi^n$:

1. G_π^i is a graph on the vertices numbered i and higher,
2. G_π^1 is just G, and
3. G_π^{i+1} is obtained from G_π^i by adding edges to make the neighbors of $\pi^{-1}(i)$ in G_π^i into a clique, and by the deletion of $\pi^{-1}(i)$ and its incident edges. The edges introduced into G_π^{i+1} correspond to the fill-in from step i of elimination.

The *filled elimination graph* G_π^* for the ordering π is the graph on V whose set of edges is the union of those in the elimination graphs $G_\pi^1, \ldots, G_\pi^n$. Note that G_π^* describes exactly the nonzero entries below the diagonal in the factor L for the factorization $PAP^T = LDL^T$, where P is the permutation of the matrix A corresponding to π.

We can in fact directly relate the edges in G_π^* to those in G. Observe that for two nonadjacent vertices u and v in G, there is an edge uv in G_π^* if and only if u and v are neighbors of a vertex w in some elimination graph, and w is eliminated before both u and v. Applying the observation repeatedly, we obtain the following useful result.

Theorem 4.1 (Rose, Tarjan & Leuker [76]) *Let G be a graph and π be an elimination ordering for G. Let $u, v \in V$. Then uv is an edge in G_π^* if and only if there exists $l \geq 0$ and vertices $w_1, \ldots, w_l$ such that $u - w_1 - \ldots - w_l - v$ is a path in G and $\pi(w_i) < \min(\pi(u), \pi(v))$ $\forall i$.*

The following result follows easily from the definition of a filled graph.

Lemma 4.2 *Let $G = (V, E)$ be a subgraph of $H = (V, F)$, i.e. $E \subseteq F$, and α be some ordering of V. Then G_α^* is a subgraph of H_α^*.*

We now introduce some special classes of elimination orderings. We say that α is a *perfect elimination ordering* for G if $G_\alpha^* = G$, that it is a *minimal elimination ordering* if there does not exist another ordering π where the edges of G_π^* are properly contained in G_α^*, and that it is a *minimum elimination ordering* if the number of edges in G_α^* is no more than the number of edges in G_π^* for any ordering π of G. If a perfect elimination ordering exists for G, we say that G is a *perfect elimination graph.* Note that the filled graph G_π^* of any ordering π for a graph G is trivially a perfect elimination graph since π is a perfect elimination ordering for G_π^*.

The problem of finding a minimum elimination ordering was shown to be NP-complete by Yannakakis [89]. As for finding a perfect elimination ordering (if

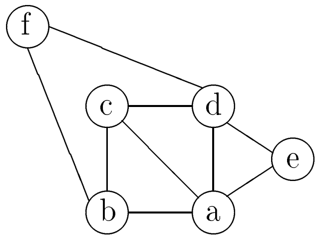


Figure 4.1: An example of a nonchordal graph. Adding the edge bd would make the graph chordal.

one exists) and finding a minimal elimination ordering, Rose, Tarjan and Leuker [76] gave algorithms with $O(n+m)$ and $O(nm)$ running times respectively. Both algorithms use a variant of breadth-first search known as lexicographic search. We shall look at their algorithm for perfect elimination orderings in Section 4.2.

Fulkerson and Gross [27] characterized the class of perfect elimination graphs in terms of chordal graphs.

Definition 4.1 *A graph G is* chordal *(also called* triangulated*) if for every cycle μ of length > 3, there exists an edge between two nonconsecutive vertices of μ. Such edges are called* chords *of the cycle.*

For example, the graph in Figure 4.1 is not chordal, but the addition of the edge bd would make it so.

For a graph $G = (V, E)$ and a subset S of the vertices, we denote by $G(S)$ the subgraph of G induced by the vertices of S. We also denote by $G - S$ the subgraph induced by $V \setminus S$. Suppose G is a chordal graph. Let μ be a cycle in $G(S)$ of length at least 4. Then μ is also a cycle in G, and it has a chord uv. Since u and v are in S, this chord is also in $G(S)$. We have therefore shown the following result.

Lemma 4.3 *Any vertex-induced subgraph of a chordal graph is also chordal.*

For a graph $G = (V, E)$, a *chordalization* (or *triangulation*) T is a set of edges whose addition makes G chordal. We refer to $G = (V, E \cup T)$ as a *chordal extension* of G. T is *minimal* if no proper subset of it is also a chordalization, and is *minimum* if its size $|T|$ is no larger than the size of any chordalization of G.

The filled graph G^*_π is in fact chordal. Indeed, for any cycle μ in G^*_π, and the vertex z of μ with the lowest ordering number, the two vertices u and v in μ adjacent to z have a chord joining them, as seen by applying Theorem 4.1 to the edges uz and vz. We can view the elimination process defined by π on G as a procedure for constructing a chordal extension of G. Clearly if G is a perfect elimination graph, then $G = G^*_\pi$ for some ordering π, and so G is chordal. The converse is also true.

Theorem 4.4 (Fulkerson & Gross [27]) *G is a perfect elimination graph if and only if it is chordal.*

For a graph $G = (V, E)$, let $\mathcal{T}(G)$ be the set of all possible chordal extensions of G, and let $\mathcal{F}(G) = \{G^*_\pi : \pi \text{ is an ordering of } V\}$ be the set of filled graphs arising from all possible elimination orderings of V. Given the equivalence between perfect elimination graphs and chordal graphs, it is natural to consider if $\mathcal{T}(G)$ and $\mathcal{F}(G)$ are identical. As remarked earlier, $\mathcal{F}(G) \subseteq \mathcal{T}(G)$ since a filled graph is necessarily chordal. However, it is *not* true that $\mathcal{T}(G) \subseteq \mathcal{F}(G)$. For example, let G be a cycle on n vertices and K be a clique on those vertices. Clearly, K is a (degenerate) chordal extension of G. Observe that for any vertex ordering π, G^*_π is a proper subgraph of K since no edges are added to the *first* vertex that is eliminated. So K cannot be realized as the filled graph of any elimination ordering.

Note, however, that in the above example, K is not a minimum chordal extension of G, since it is well-known that a minimum chordalization of a cycle has only $n - 3$ edges [41]. Indeed, every minimal chordal extension (which includes all minimum chordal extensions) can be realized as the filled graph of some elimination ordering, as we shall now show. Let H be a minimal chordal extension for a graph G, and let π be a perfect elimination ordering for H, which exists by the equivalence between chordal graphs and perfect elimination graphs. Consider the filled graph G^*_π. Clearly, G^*_π is a subgraph of H since G is a subgraph of H and $H^*_\pi = H$. But G^*_π, which is chordal, cannot be a proper subgraph of H by the minimality of H. Thus $G^*_\pi = H$, and H is realized by the ordering π.

For more on chordal graphs and their relation to elimination, we refer the reader to the book by Golumbic [41] and to the tutorial by Blair and Peyton [8].

4.2 An Algorithm for Perfect Elimination Orderings

As we saw in the previous section, every chordal graph has a perfect elimination ordering. Rose, Tarjan and Leuker [76] were the first to give a linear time algorithm for finding such orderings. We shall use this algorithm and exploit one of its properties in the variant of nested dissection that we propose in Section 4.5. Throughout this section, we shall focus exclusively on chordal graphs.

The Rose-Tarjan-Leuker perfect elimination ordering algorithm [76] uses a variant of breadth-first search known as *lexicographic search*. Suppose we perform breadth-first search from some vertex of a chordal graph $G = (V, E)$. It is well known that breadth-first search partitions the vertices of the graph into levels, based on their hop distances from the root vertex, such that two adjacent vertices in G are either at the same level or are one level apart. Consider now a perfect elimination ordering π for G, and three vertices u, v and w on three increasing adjacent levels. Suppose uv and vw are both edges in G. By the previous remark we know that uw is not an edge of G. It follows then from Theorem 4.1 that $\pi(v) > \min(\pi(u), \pi(w))$. This suggests that there is a perfect elimination ordering for G that is consistent with the partial ordering by levels, as shown below.

To resolve the ties within the levels given by breadth-first search, Rose, Tarjan and Leuker associate a label to each vertex and use a *lexicographic* comparison rule for labels. Specifically, each vertex v has a label consisting of a possibly empty subset of $\{1, \ldots, n\}$, ordered in decreasing order. Given two label $L_1 = [p_1, p_2, \ldots, p_k]$ and $L_2 = [q_1, q_2, \ldots, q_l]$, we define $L_1 < L_2$ if

(i) for some j, $p_i = q_i$ for $i = 1, \ldots, j-1$ and $p_j < q_j$, or

(ii) $k < l$ and $p_i = q_i$ for $i = 1, \ldots, k$.

In addition, $L_1 = L_2$ if $k = l$ and $p_i = q_i$ for all $1 \leq i \leq k$. For example, $[7, 5, 3] < [7, 6]$ and $[7, 5, 3] < [7, 5, 3, 1]$. Note that an empty label is always smaller than any nonempty label.

The Rose-Tarjan-Leuker algorithm constructs an ordering π in *reverse* for a graph by performing breadth-first search from an arbitrarily chosen vertex and maintaining labels for the vertices. These labels are used to break ties between the vertices.

<u>Rose-Tarjan-Leuker</u>

assign the label $\emptyset$ to all vertices
for $i = n, n-1, \ldots, 1$
    select an unnumbered vertex v with the largest label
    $\pi(v) = i$
    for each unnumbered vertex w adjacent to v
        add i to the label for w
    endfor
endfor

Figure 4.2 illustrates some of the iterations of the algorithm on an example graph, and also shows the final ordering and labels. The vertex numbers are given within the circles while the labels are written next to them. The example is taken from [76]. Note that the assignment of numbers to vertices proceeds according to the levels of a breadth-first search from the vertex numbered 10.

It can be shown that the ordering produced by lexicographic breadth-first search on a chordal graph is perfect. In addition, the algorithm can be implemented to run in $O(n + m)$ time. For the proof of the perfectness of the ordering and the details of the implementation, including the analysis of its running time, we refer the reader to [76].

We now prove a result that will be crucial to an ordering scheme that we shall propose in Section 4.5.

Theorem 4.5 *Let $G = (V, E)$ be a chordal graph, and suppose $L \subseteq V$ is a set of vertices that forms a clique in G. There exists a perfect elimination ordering of G that numbers the vertices in L last.*

Proof. We show that the desired ordering can be found by the Rose-Tarjan-Leuker algorithm. Since the algorithm produces an ordering in reverse, it suffices to show

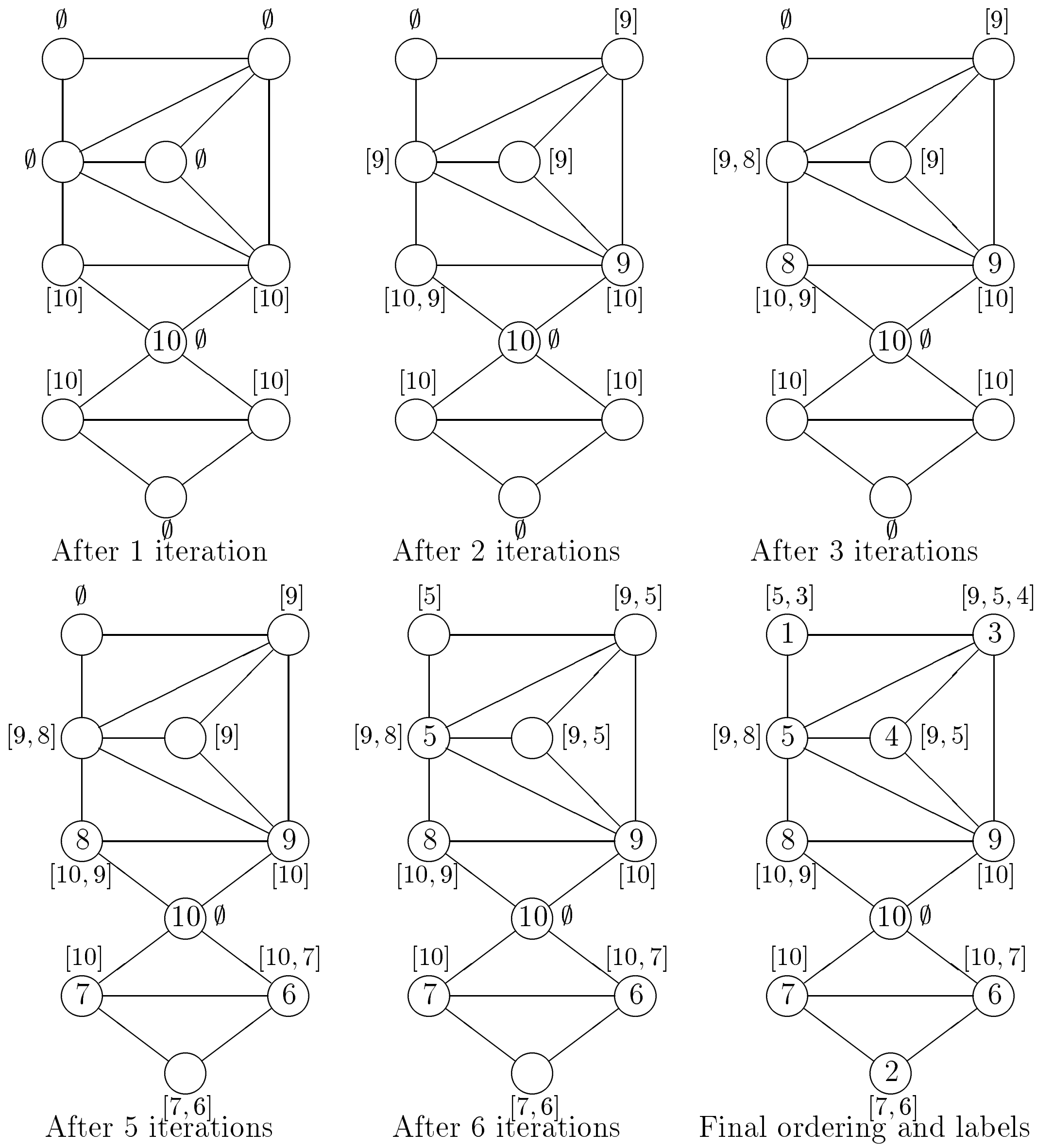


Figure 4.2: An illustration of the Rose-Tarjan-Leuker perfect elimination ordering algorithm.

that for some run of the algorithm, all the vertices in L are assigned numbers prior to a vertex outside of L being numbered.

Suppose the algorithm first picks a vertex $z_1 \in L$, and assigns it the number n. After this first step, all the neighbors of z_1, in particular, those in L, will have the label $[n]$, while the remaining vertices have the empty label. So at the second step, the algorithm can pick another vertex $z_2 \in L$, and number it $n-1$. At that point, all the unnumbered vertices in L will have the label $[n, n-1]$, which is the *largest* label in the graph. Thus another vertex in L can be chosen at the third step. In general, after k of the vertices in L have been numbered in succession, the unnumbered vertices in L will always have the largest label in the graph, because of the mutual adjacency of the vertices in L. It is possible though that other vertices in the graph may also have the largest label. So the Rose-Tarjan-Leuker algorithm can always pick vertices in L during the initial iterations, until all those vertices have been numbered. Consequently, we obtain a perfect ordering with the vertices in L numbered last, as claimed. ∎

Rose, Tarjan and Leuker also showed that the lexicographic breadth-first search scheme can be extended to produce a minimal ordering for a graph by modifying the label updating step. The resulting minimal ordering algorithm runs in time $O(nm)$ and the details of it can be found in [76]. Subsequently, Tarjan [84] [85] gave another linear time algorithm for finding a perfect ordering for a chordal graph. That algorithm, known as the *maximum cardinality search* algorithm, is also based on breadth-first search and also produces an ordering in reverse. At each iteration, it assigns the next ordering number to an unnumbered vertex with the most numbered neighbors, with ties broken arbitrarily. The maximum cardinality search algorithm can also be used to generate a perfect ordering for our fixed-last-vertices chordal graph ordering problem above.

4.3 Nested Dissection

As we saw in Section 1.5, the method of nested dissection is aymptotically optimal for certain classes of graphs, with respect to the fill and operation count. By using the Leighton-Rao approximately optimal balanced separator construction in Section 3.6.1 for the nested disssection process, Agrawal, Klein and Ravi [2] were able to show that the resulting ordering is approximately optimal with respect to the fill, operation count and elimination height. Our goal in this section is to understand the analysis behind this important result of Agrawal, Klein and Ravi. The ideas developed here will also be useful when we examine two variants of nested dissection in Sections 4.4 and 4.5. In particular, we shall look at how fill-in occurs for a nested dissection ordering. We will prove some of the results that we state, and also illustrate them by examples.

4.3.1 Nested Dissection and Separator Trees

We begin by stating the nested dissection method of George [31] in its full generality for arbitrary graphs. Let G be a graph that we wish to order using nested dissection. The ordering is produced by the following recursive routine with an initial call of NESTEDDISSECTION$(G, 1, n)$.

NESTEDDISSECTION(G, a, b)

if $n = 1$ **then**
  number the single vertex a
else
  find a "good" vertex separator X, and let $G - X$ have k
    connected components $A_1, \ldots, A_k$ with $n_1, \ldots, n_k$ vertices
  number the vertices in X arbitrarily from $b - |X| + 1$ to b
  for $i = 1, \ldots, k$
    call ND$(A_i,\ a + \sum_{j=1}^{i-1} n_j,\ a - 1 + \sum_{j=1}^{i} n_j)$
  endfor
endif

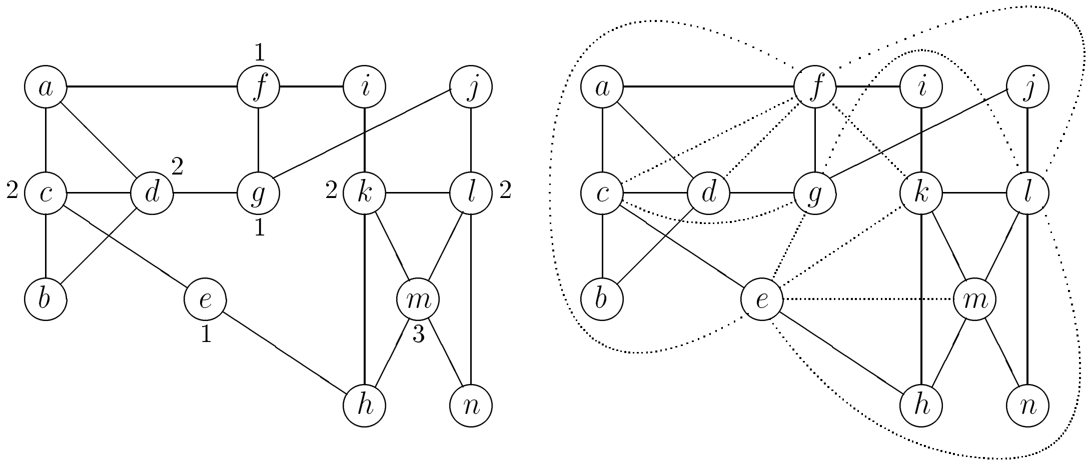
Throughout this section, we let α denote a nested dissection ordering for the graph G.

The quality of the nested dissection ordering α and the analysis that can be made for it depends of course on the properties of the separators used in the recursive process. We defer the discussion of these issues to Sections 4.3.2 and 4.3.3, and focus here on understanding the nested dissection process in general.

We present an example of a nested dissection ordering in Figure 4.3. This example will serve to illustrate some of the main ideas of this and the following two sections. For each vertex v in the example graph Q, the number l next to it, if any, indicates that v is contained in a separator used at level l of the recursion. Thus the top-level separator in the nested dissection process on Q is $\{e, f, g\}$, while the two separators at the next level of recursion are $\{c, d\}$ and $\{k, l\}$. The filled graph Q^*_α for the ordering is also shown, with the fill-in edges indicated by the dotted lines.

We can represent the nested dissection process by a *separator tree*, where each tree node is a separator that was used. For purposes of this representation, we regard the single vertex in a base case as a degenerate separator. The use of a separator tree to model the nested dissection process and to analyze the fill of the ordering produced was introduced by Gilbert and Tarjan [39]. The separator tree for a graph G is defined recursively as follows. The root node is the separator of G that was used; each subtree is the separator tree for one of the components of G remaining after the removal of the separator. To avoid confusion, we will not refer to the nodes of the separator tree as vertices. We show in Figure 4.4 the separator tree for our running example.

We now introduce some notation and definitions for working with the separator tree. For a vertex v, we let X_v denote the separator containing v. For a separator



Graph Q | Filled graph Q^*_α

vertex	b	a	d	c	i	j	n	h	m	k	l	e	f	g
α-number	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure 4.3: An example of nested dissection and the ordering produced.

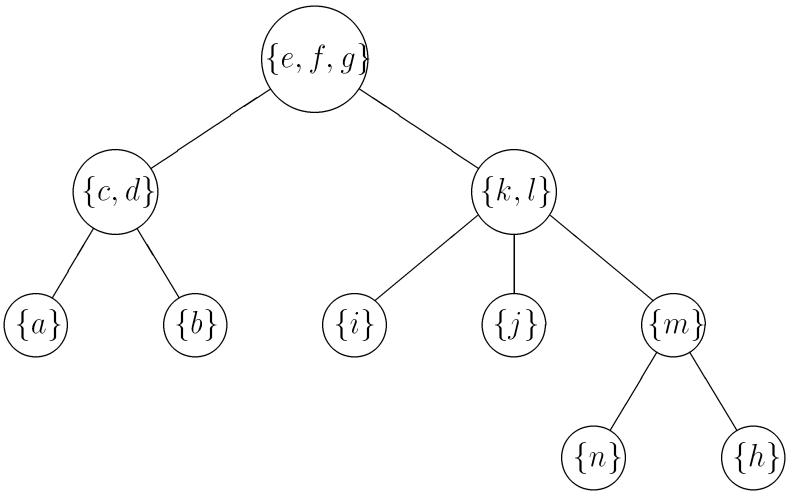


Figure 4.4: The separator tree for the nested dissection process on the graph Q.

X, we let $V(X)$ be the union of the separators contained in the subtree rooted at X. We also define *level*(X) to be the distance of X from the root node, and say that a separator X is an *ancestor* of a separator Y (or Y is a *descendant* of X) if Y is in the subtree rooted at X. Note that we allow $X = Y$. For example, for the illustrative graph Q, $X_c = \{c, d\}$, $V(\{k, l\}) = \{h, i, j, k, l, m, n\}$ and X_k is an ancestor of X_n.

The separator tree is a useful device because it reflects much of the structure of the filled graph G^*_α. More precisely, for any edge uv in G^*_α, the separator X_u must either be an ancestor or descendant of X_v. This fact is proved below and illustrated in Figure 4.5, where we have superimposed the edges of our example graph Q and the fill-in edges upon its associated separator tree.

Lemma 4.6 (Gilbert & Tarjan [39]) *Suppose vw is an edge in G^*_α and $\alpha(v) > \alpha(w)$. Then X_v is an ancestor of X_w.*

Proof. Suppose it is not the case that X_v is an ancestor of X_w or X_w is an ancestor of X_v. Then the lowest common ancestor X_{lca} of X_v and X_w is different from X_v and X_w. Let $X_1, \ldots, X_l$ be the ancestors of X_{lca} in the separator tree. Since the removal of $X_1 \cup \ldots X_l \cup X_{lca}$ from the graph G separates v from w, it follows that any path from v to w must include a vertex in $X_1 \cup \ldots X_l \cup X_{lca}$. By Theorem 4.1, the last remark contradicts the existence of the edge vw in G^*_α, for the vertices in that collection of separators are all numbered higher than both v and w. Therefore X_v is an ancestor of X_w since $\alpha(v) > \alpha(w)$. ∎

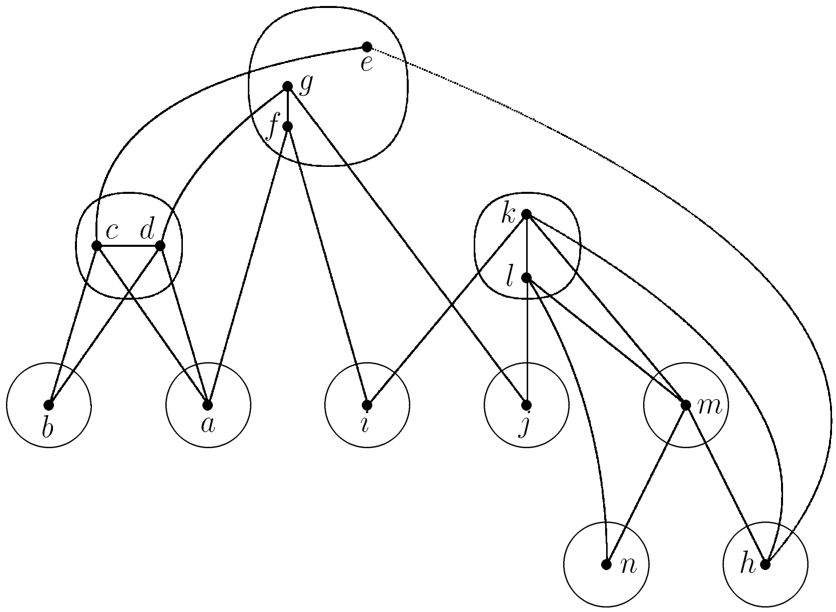
We can in fact more precisely characterize the edges in G^*_π in terms of the edges in G and the separator tree.

Lemma 4.7 (Gilbert & Tarjan [39]) *Suppose vw is an edge in G^*_α and $\alpha(v) > \alpha(w)$. Then there is an edge in G from v to a vertex in a separator that is a descendant of X_w.*

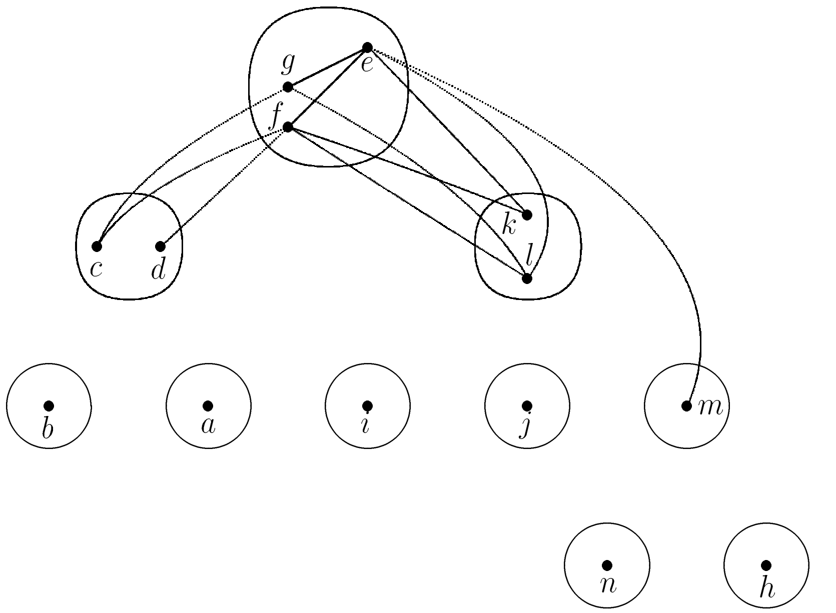
Proof. If vw is an edge in G, then the lemma is trivially true since X_w is a descendant of itself. Assume therefore that vw is not an edge in G. It follows from Theorem 4.1 that there must exists a path $v - z_1 - \ldots - z_p - w$, $p \geq 1$, with $\alpha(z_i) < \alpha(w)$ $\forall i$. We wish to show that X_{z_1} is a descendant of X_w. Suppose it is not the case. By the common ancestors argument used in Lemma 4.6, we see that the path from z_1 to w must contain a vertex in a proper ancestor of X_w, which is a contradiction since those vertices are numbered higher than w. ∎

As an illustration of Lemma 4.7, we see from Figure 4.5 for our example graph that there is a fill-in edge ek. This fill-in edge arises when vertex h is eliminated. Observe that eh is an edge of G, and that the separator X_h is indeed a descendant of X_k, as guaranteed by the lemma.

We next show that the maximum degree of the graph G can be used to bound the number of separators at a given level that contain neighbors of some fixed



(i) Edges of Q



(ii) Fill-in edges

Figure 4.5: The example graph Q and its fill-in edges superimposed upon its separator tree. Note that the edges only go between vertices whose containing separators satisfy an ancestor-descendant relationship.

vertex. To do that, we make another definition. Let $v \in V$. We say that a separator X with $level(X) \geq level(X_v)$ is *α-connected* to v if there exists $w \in X$ such that vw is an edge in G^*_α. By Lemma 4.6, if X is α-connected to v, then X_v is an ancestor of X.

Lemma 4.8 *Let $v \in V$, and $\mathcal{R}$ be the set of separators at level l that are α-connected to v, for some fixed $l \geq level(X_v)$. Then $|\mathcal{R}| \leq d$, where d is the maximum degree of G.*

Proof. By Lemma 4.7, each $X \in \mathcal{R}$ has a descendant which contains a vertex adjacent to v in G. The result follows since there are at most d vertices adjacent to v and the subtrees rooted at each $X \in \mathcal{R}$ are disjoint. ■

We have seen so far how fill-in occurs between the vertices in a component and the vertices in a separator. We now show that the vertices in a separator X suffer complete fill, i.e. the subgraph induced by X in the filled graph G^*_π is a clique, provided some connectivity constraint holds for the vertices in X.

Lemma 4.9 *Let X be a separator in the nested dissection process, and let G_X be the subgraph of G that it decomposes. Suppose that for each pair of vertices u and v in X, there is a common component of $G_X - X$ to which both u and v are adjacent to. Then the vertices in X are mutually adjacent in G^*_π.*

Proof. Let $u, v \in X$, and let A be the component of $G_X - X$ to which both u and v are adjacent. Then there is a path from u to v that only goes through the vertices in A. Since the vertices in A are numbered lower than both u and v, it follows from Theorem 4.1 that uv is an edge in G^*_π, as desired. ■

4.3.2 Nearly Optimal Nested Dissection

The quality of the ordering produced by nested dissection depends on the separators used in the process. For rectangular girds, there are natural cross-shaped separators which divide up the grid into four roughly equal-sized rectangular subgrids. Using these separators, George [31] showed that the resulting nested dissection ordering for an n-node rectangular grid generates $O(n \log n)$ fill and requires $O(n^{3/2})$ arithmetic operations. These bounds are asymptotically optimal, since George [31] and later Hoffman *et al* [44] proved that any elimination ordering for a rectangular grid has $\Omega(n \log n)$ fill and requires $\Omega(n^{3/2})$ arithmetic operations. Correspondingly, by using the planar separator theorem [62], Gilbert and Tarjan [39] proved that a nested dissection ordering for a planar graph has the same fill and operation count bounds as rectangular grids. Their results also apply to two-dimensional finite element graphs and graphs of bounded genus.

Given those positive results, it is natural to ask if similar performance bounds can be proved for nested dissection on general graphs. In this section, we prove an existential approximate optimality result for the fill of a nested dissection ordering. We shall see that for any n-vertex graph with maximum degree d, there is a nested dissection ordering whose fill is within an $O(d \log n)$ factor of the minimum possible fill. This result is due to Gilbert [37], and is a precursor to the worst-case performance bounds that Agrawal, Klein and Ravi [2] subsequently gave for nested dissection in conjunction with approximately optimal balanced separators found by the Leighton-Rao construction. We shall present the Agrawal-Klein-Ravi analysis in Section 4.3.3. The guarantee that we show here is different in nature from those in Gilbert and Tarjan [39], in that it directly compares the fill of the almost optimal nested dissection ordering to the minimum possible fill for the given graph.

The key idea in the proof of approximately optimal fill is the existence of special separators in chordal graphs. We shall use such separators for the vertex-induced subgraphs of a minimum chordal extension of G to construct the desired nested dissection ordering. Gilbert, Rose and Edenbrandt [38] proved several separator theorems for unweighted and weighted chordal graphs. We state below the version that is needed here. We say that a separator is a *clique separator* if it is a clique.

Theorem 4.10 (Gilbert, Rose & Edenbrandt [38]) *Let G be a chordal graph. Then G has a clique separator X such that each connected component of $G - X$ has at most $\frac{1}{2}n$ vertices and at most $\frac{2}{3}m$ edges.*

Note that the separator in Theorem 4.10 ensures that there will not be too many vertices or too many edges in the components remaining after its removal. In addition, the separator itself has only $O(\sqrt{m})$ vertices because it is a clique.

For a graph G, we denote by $|G|$ the number of edges in G. We now state the approximate fill guarantee and prove it. As usual, we assume that the graph G is connected.

Theorem 4.11 (Gilbert [37]) *Let G be a graph with maximum degree d. There exists a nested dissection ordering π for G where the number of edges in the filled graph G^*_π is within a factor of $O(d \log n)$ of the number of edges in a minimum chordal extension G^* of G.*

Proof. Let m^* denote the number of edges in G^*. We construct the desired nested dissection ordering π by specifying the separators used in the nested dissection process. At the same time, we also upper bound the sizes of those separators.

Observe that the graph of a nested dissection subproblem, including the top-level problem, is a subgraph of G induced by some subset S of the vertices. By Theorem 4.10, we know that there is a $\frac{1}{2}$-balanced clique separator X_S for the subgraph $G^*(S)$ of G^* induced by S, such that each component $C_1, \ldots, C_k$ of $G^*(S) - X_S$ has at most $\frac{2}{3}|G^*(S)|$ edges. Note that X_S is also a separator for $G(S)$. We assume that the nested dissection algorithm chooses X_S to separate $G(S)$. Let $A_1, \ldots, A_k$ denote the components of $G(S) - X_S$. Each component A_i must be a subgraph of some component C_j, and so the number of edges in A_i is also at most $\frac{2}{3}|G^*(S)|$. Applying the idea repeatedly, and numbering the recursion levels starting from 0, we see that if $G(S)$ is the graph of a problem at level l of nested dissection, then

$$|G^*(S)| \leq \left(\frac{2}{3}\right)^l m^*.$$

So the clique separator X_S used by nested dissection contains $O\left(\sqrt{(\frac{2}{3})^l m^*}\right)$ vertices.

We now upper bound the number of edges in the filled graph G^*_π for the nested dissection ordering π. We first fix a level l in the separator tree and a separator X at level l, and count the number of fill edges whose higher-numbered endpoint is in X. This is accomplished by defining a correspondence between a subset of the edges in G^*_π and a subset of the edges in G. Fix a vertex $v \in X$, and consider an edge $vw \in G^*_\pi$. By Lemma 4.7, we know that there is a vertex z such that vz is an edge of G and the separator X_z containing z is a descendant of the separator X_w containing w. We say that the edge vw corresponds to the edge vz. The number of edges of G^*_π that can correspond to an edge vz of G is therefore bounded by the total number of vertices in the ancestors of X_z in the separator tree. By the earlier bound on the sizes of the separators, this total is at most

$$O\left(\sqrt{m^*} + \sqrt{\frac{2}{3}m^*} + \sqrt{\frac{4}{9}m^*} + \ldots\right) = O(\sqrt{m^*}).$$

Let $G(S)$ be the subgraph of G that X separates in the nested dissection process. By construction, there are at most $O(\sqrt{|G^*(S)|})$ vertices in X. Since each of the vertices in X has at most d neighbors in G, we see that there are $O(d\sqrt{m^*|G^*(S)|})$ edges of G^*_π with higher-numbered endpoints in X.

Note that this analysis double counts the edges in G^*_π that are between vertices in X, but there are only $(|G^*(S)|) = O(\sqrt{m^*|G^*(S)|})$ such edges.

Since the subgraphs separated by the separators at level l are disjoint, it can be shown that the total number of edges in G^*_π with higher-numbered endpoints in a level-l separator is $O(d\sqrt{m^*|G^*|}) = O(dm^*)$. By the $\frac{1}{2}$-balancedness of the separators used, we know that the separator tree has $O(\log n)$ levels. Summing over all the levels of the separator tree, we see that the number of edges in G^*_π is $O(dm^* \log n)$, which is within an $O(d \log n)$ factor of the number of edges in a minimum chordal extension, as claimed. ■

4.3.3 The Agrawal-Klein-Ravi Algorithm

The result of Gilbert [37] that we examined in the previous section shows that on general graphs, nested dissection can produce an ordering that is approximately optimal with respect to fill. However, the result is of limited practical value because it requires knowledge of separators that are related to a minimum chordal extension of the given graph in order to carry out the approximately optimal nested dissection process. By using separators that are almost as good as those arising from a minimum chordal extension, Agrawal, Klein and Ravi [2] were able to adapt Gilbert's existential approximate optimality analysis to give a *worst-case* guarantee for the fill of their nested dissection ordering. They also proved similar guarantees for the multiplication count and the elimination height of their nested dissection ordering.

The separators that Agrawal, Klein and Ravi use for nested dissection are constructed by the Leighton-Rao balanced separator construction [57], as described in Section 3.6.1. These separators are $\frac{2}{3}$-balanced and have the following property.

Theorem 4.12 (Leighton & Rao [57]) *There exists a polynomial time algorithm that finds a $\frac{2}{3}$-balanced vertex separator whose size is within an $O(\log n)$ factor of the size of an optimal $\frac{1}{2}$-balanced separator.*

We shall refer to the nested dissection algorithm that uses $\frac{2}{3}$-balanced separators from the Leighton-Rao construction as the *Agrawal-Klein-Ravi algorithm.* In this section, we shall prove the fill guarantee of the Agrawal-Klein-Ravi algorithm on graphs with bounded degree and state the corresponding result for unbounded-degree graphs. We shall also discuss the performance guarantee on multiplication count and prove the guarantee on elimination height. We note that the proofs of the guarantees depend only on the approximation guarantees of the separators used, and not on the actual method by which they are found.

Analysis of Fill

We use the separator tree introduced in Section 4.3.1 and its properties to obtain a performance bound on the fill for the ordering found by the Agrawal-Klein-Ravi algorithm when it is given a graph with bounded degree. Observe that the height of the separator tree is $O(\log n)$ since at each step of the nested dissection, the removal of a separator reduces the number of vertices (in a largest component) by a constant factor $\frac{2}{3}$.

Let G^* be a minimum chordalization of G. Note that $|G^*|$ is a lower bound on the fill of any ordering. We first relate the sizes of the separators at a given level of the separator tree to $|G^*|$. To do so, we use the following simplified version of the chordal graph clique separator theorem used in Gilbert's analysis in Section 4.3.2.

Lemma 4.13 ([38]) *A chordal graph has a $\frac{1}{2}$-balanced clique separator.*

Theorem 4.14 *Let $X_1, \ldots, X_p$ be the separators at a given level of the separator tree. Then*

$$\sum_{i=1}^{p} |X_i|^2 = O(\log^2 n)|G^*|.$$

Proof. For each separator X_i, let V_i be the union of the separators in the subtree rooted at X_i. Then $\{V_i\}$ forms a partition of a subset of V. Let G_i and G_i^* be the subgraphs of G and G^* induced by V_i. Since G_i^* is chordal, it has a $\frac{1}{2}$-balanced clique separator C_i^*, whose size $|C_i^*|$ satisfies $\frac{1}{2}|C_i^*|(|C_i^*|-1) \leq |G_i^*|$. Note that C_i^* is also a $\frac{1}{2}$-balanced separator of G_i. So $|X_i| = O(\log|V_i|)|C_i^*|$ since X_i is an approximately optimal separator of G_i. Therefore

$$\sum_{i=1}^{p} |X_i|^2 = \sum O(\log^2|V_i|)|C_i^*|^2 = O(\log^2 n)|G^*|,$$

as desired. ■

The sizes of the separators can be used to upper bound the fill from the nested dissection ordering α. Let v be a vertex in some separator at a given level k. Recall that we defined a separator X at level $l \geq k$ to be α-connected to v if X contains a neighbor of v in G_α^*. Therefore the number of edges in G_α^* between v and vertices in level-l separators is at most the sum of the sizes of the level-l separators that are α-connected to v. This fact is used to prove the fill bound in Theorem 4.15 below.

Theorem 4.15 (Agrawal, Klein & Ravi [2]) *The elimination ordering found by the algorithm has fill that is within an $O(\sqrt{d}\log^4 n)$ factor of optimal, where d is the maximum degree of the graph.*

Proof. Consider fixed levels k and l with $l \geq k$. We bound the number of edges in G_α^* between a vertex in a level-k separator and a vertex in a level-l separator.

Let $X_1, \ldots, X_p$ denote the separators at level k. By Lemma 4.8, each vertex in a level-k separator has at most d separators at level l that are α-connected to it. For each X_i, pick a vertex $v_i \in X_i$ such that the sum of the sizes of its level-l α-connected separators is maximum, and let $Y_{i,1}, \ldots, Y_{i,q_i}$ be those separators. Then the number of edges in G_α^* between vertices in level-k separators and vertices in level-l separators is bounded by

$$\begin{aligned}
\sum_{i=1}^{p}(|X_i|\sum_{j=1}^{q_i}|Y_{i,j}|) &= \sum_{i=1}^{p}\sum_{j=1}^{q_i}|X_i||Y_{i,j}| && (4.1)\\
&\leq \sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i}|X_i|^2}\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i}|Y_{i,j}|^2} && (4.2)\\
&\leq \sqrt{d\sum_{i=1}^{p}|X_i|^2}\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i}|Y_{i,j}|^2}. && (4.3)
\end{aligned}$$

The first inequality follows from the Cauchy-Schwarz inequality, while the second inequality follows from $q_i \leq d$. Now, $\{X_i\}_{1\leq i\leq p}$ are the separators at level k, while $\{Y_{i,j}\}_{1\leq i\leq p, 1\leq j\leq q_i}$ is a subset of the separators at level l. By Theorem 4.14, $\sqrt{\sum_{i=1}^{p}|X_i|^2} = O(\log n)\sqrt{|G^*|}$ and $\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i}|Y_{i,j}|^2} = O(\log n)\sqrt{|G^*|}$. Therefore, the quantity (4.3) is bounded by $\sqrt{d}O(\log^2 n)|G^*|$. Summing over all levels k and l gives the theorem. ■

The fill analysis above is easily modified to give the following weaker form of Gilbert's existential approximate optimality result for nested dissection.

Theorem 4.16 *Let G be a graph with maximum degree d. There exists a nested dissection ordering whose fill is within an $O(\sqrt{d}\log^2 n)$ factor of optimal.*

Agrawal [1] also gave a performance guarantee on graphs with unbounded degree. We state his result below and refer the interested reader to [1] for the proof.

Theorem 4.17 (Agrawal [1]) *The elimination ordering found by the algorithm yields a fill of size $O(F^{*\frac{3}{4}}\sqrt{m}\log^{3.5} n)$ where F^* is the size of the minimum fill and m is the number of edges in the graph.*

Analysis of Multiplication Count

We do not analyze here the multiplication count of the Agrawal-Klein-Ravi ordering, but instead briefly describe how the multiplication count is characterized. The interested reader should refer to [2] for the details.

Recall from Section 1.3 that when a vertex v is eliminated, edges are added to make its higher-numbered neighbors into a clique. The number of edges in this clique is equal to the number of multiplications performed in eliminating v. The total number of edges in the cliques associated with each vertex therefore yields the number of multiplication operations for the elimination ordering. Using this characterization, Agrawal, Klein and Ravi showed the following bound on the multiplication count.

Theorem 4.18 (Agrawal, Klein & Ravi [2]) *The elimination ordering found by the algorithm has a multiplication count that is within an $O(d\log^6 n)$ factor of optimal, where d is the maximum degree of the graph.*

Analysis of Elimination Height

The analysis of the elimination height of the nested dissection ordering α is considerably simpler than that for the fill and for the multiplication count. Recall from Section 1.3 that if two vertices v and w are adjacent in the filled graph G^*_α, then the higher numbered vertex cannot be eliminated before the lower numbered one. Therefore the height of α is the number of vertices in a longest directed path in the graph obtained by orienting the edges in G^*_α from lowered numbered vertices to higher numbered ones. By Lemma 4.6, these directed edges can only go from a vertex in one separator X to a vertex in an ancestor of X. Therefore, the elimination height of α is no more than the maximum total size of the separators that occur on a tree path from a leaf node to the root node of the separator tree. Correspondingly, we can lower bound the minimum elimination height by looking at appropriate clique separators in the filled graph of an ordering with minimum height. The two bounds are used to prove the performance guarantee stated below.

Note that the guarantee does not have a dependence on the maximum degree of the graph.

Theorem 4.19 (Agrawal, Klein & Ravi [2]) *The elimination ordering found by the algorithm has a height that is within an $O(\log^2 n)$ factor of optimal.*

Proof. Let h be the minimum possible elimination height and π be an elimination ordering for G of height h. Denote the filled graph of the ordering π by G^*. Let $X_1, \ldots, X_k$ be the separators that occur on a maximum-total-size path described above. By definition, $\sum_i |X_i|$ is an upper bound on the height for the nested dissection ordering α.

Consider one of those separators X_i, and let $G(S)$ be the vertex-induced subgraph of G that it separates. Let C be a $\frac{1}{2}$-balanced clique separator for $G^*(S)$. Observe that C is also a $\frac{1}{2}$-balanced separator for $G(S)$. Thus the Leighton-Rao construction guarantees that $|X_i| = O(\log n)|C|$. Since C is a clique in the filled graph for the minimum height ordering π, it produces a directed path with $|C|$ vertices when the edges of that filled graph are oriented. So $|C| \leq h$, and consequently, $|X_i| = O(h \log n)$. Since $k = O(\log n)$, summing over the separators $X_1, \ldots, X_k$ gives the stated performance guarantee. ■

4.4 Generalized Nested Dissection

Nested dissection is typical of many divide-and-conquer algorithms in that the vertex separator that is used to decompose a problem is not included as part of the recursive subproblems. When the solutions to the subproblems are later combined to form a solution to the original problem, the interaction between the separator and the subproblems introduces additional cost. In the case of nested dissection, when orderings for the subproblems are concatenated to give a complete ordering, the fill that occurs between the components and the separators, which we call the *cross-fill*, is additional. A good nested dissection process limits the cross-fill by using small separators.

Given the drawback mentioned above, it is natural to extend the nested dissection process by including the separator in the recursive subproblems. In this section, we shall focus on one such extension, known as *generalized nested dissection*, which was introduced by Lipton, Rose and Tarjan [61]. The inclusion of the separator in the subproblems simplifies the analysis of the fill for special classes of graphs. In fact, Lipton, Rose and Tarjan gave fill and multiplication bounds for any class of graphs having a theorem which guarantees small balanced separators. The proofs of these bounds employ the standard technique of bounding the quantity of interest using a recurrence relation that models the recursive process, and does not appeal to the separator tree device that was used in the analysis for regular nested dissection.[1] However, this variant of nested dissection has not been as well-studied since, nor has it been used in practice. Nonetheless, the ideas contained therein are interesting, and serve as a motivation for a theoretically superior variant of nested dissection that we shall propose and examine in Section 4.6.

We first describe the generalized dissection algorithm in Section 4.4.1 and then examine the nature of its recursive problems in Section 4.4.2. We state and briefly discuss in Section 4.4.3 some of the results of Lipton, Rose and Tarjan on the performance of generalized nested dissection on special classes of graphs. We will not prove those results, as our focus is on gaining an understanding of this variant of nested dissection on general graphs.

4.4.1 The Generalized Nested Dissection Algorithm

It is convenient to first introduce additional terminology to help describe the generalized nested dissection process. Let G be a graph and X be a subset of its vertices. We denote the clique on the vertices in X by $K(X)$, and we write $G - K(X)$ to mean the graph G with the removal of all edges between vertices in X.

The generalized nested dissection algorithm of Lipton, Rose and Tarjan is given below. Given a graph $G = (V, E)$, a set of previously numbered vertices $N \subseteq V$, and a range $[a, b]$ with $b = a + (|V| - |N|) - 1$, the routine GENERALIZEDND produces a nested dissection ordering in the range $[a, b]$ for the unnumbered vertices in G. The algorithm first finds a separator X and uses it to divide up the graph into two parts A and B. It numbers the unnumbered vertices in X last, and then calls itself recursively on the subgraphs induced by A (or B) and X, where edges between vertices in X are removed. To order an initially unnumbered graph G, we make the call GENERALIZEDND$(G, \emptyset, 1, n)$.

<u>GENERALIZEDND$(G = (V, E), N, a, b)$</u>

if G has only one unnumbered vertex **then**
    number that vertex a
else
    find a "good" separator X for G, and group the vertices of
        the components of $G - X$ into two parts A and B
    let i, j, k be the number of unnumbered vertices in A, B and X respectively
    number the unnumbered vertices in X from $b - k + 1$ to b
    call GENERALIZEDND$((G - K(X))(A \cup X),\ (N \cap A) \cup X,\ a,\ a + i - 1)$
        to number the subproblem on the vertices in A and X
        with edges between vertices in X removed
    call GENERALIZEDND$((G-K(X))(B \cup X), (N \cap B) \cup X, a+i, a+i+j-1)$
        to number the subproblem on the vertices in B and X
        with edges between vertices in X removed

[1] As a chronological note, we point out that the Lipton, Rose and Tarjan [61] paper predates by 8 years the Gilbert and Tarjan paper [39] that analyzed the performance of regular nested dissection on planar and related graphs.

endif

We have not explicitly specified in the algorithm the criteria for choosing the separators to be used in the dissection process. For graphs with separator theorems, like those considered in Section 4.4.3, the appropriate separators are given by the theorems. In the general case, we use $\frac{2}{3}$-balanced separators found by the Leighton-Rao balanced separator construction described in Section 3.6.1.

Note that the algorithm groups the components $G - X$ into two parts and recurses on the subproblems defined by those parts and the separator, instead of using the usual multi-way decomposition where each component of $G-X$ gives rise to a subproblem. This distinction is usually irrelevant in practice, but is significant in the context of proving performance bounds for generalized nested dissection on special classes of graphs, as we shall see in Section 4.4.3.

In addition, in actual implementation, we do not include all the separator vertices in the subproblem, but only those that are adjacent to the group of components forming the subproblem. This leads to potentially smaller subproblems, but does not improve on the bounds given in Section 4.4.3 on the performance of the algorithm on special graphs. The version we have given here is also notationally simpler to discuss.

Finally, we repeat that the algorithm leaves out edges that are between vertices in the separator when forming a subproblem. This omission also does not confer any theoretical advantage in the analysis of the special classes of graphs considered in Section 4.4.3.

We trace in Figure 4.6 the generalized nested dissection algorithm on our running example graph Q.

4.4.2 Discussion of Generalized Nested Dissection

We now examine several aspects of generalized nested dissection, and contrast its behavior to regular nested dissection. We first formally define the binary *recursion tree* for the generalized nested dissection process recursively as follows. The root node of this tree is the input problem, while each of the two subtrees is the recursion tree for a subproblem associated with the removal of the separator from the input graph. We say that one problem P_1 is an ancestor of another problem P_2 if P_1 is the parent of P_2, or is an ancestor of P_2's parent. In this case, we also say that the graph of P_1 is an ancestor of the graph of P_2. We number the recursion levels starting from 1 for the root node. The recursion tree is simply an analog of the separator tree that we have used in analyzing regular nested dissection.

Throughout this section, $G = (V, E)$ will denote the input graph that gives rise to the recursion tree.

Subproblem Characteristics

We first characterize some of the properties of the problems that occur in the generalized nested dissection process. Each problem (or node) in the recursion

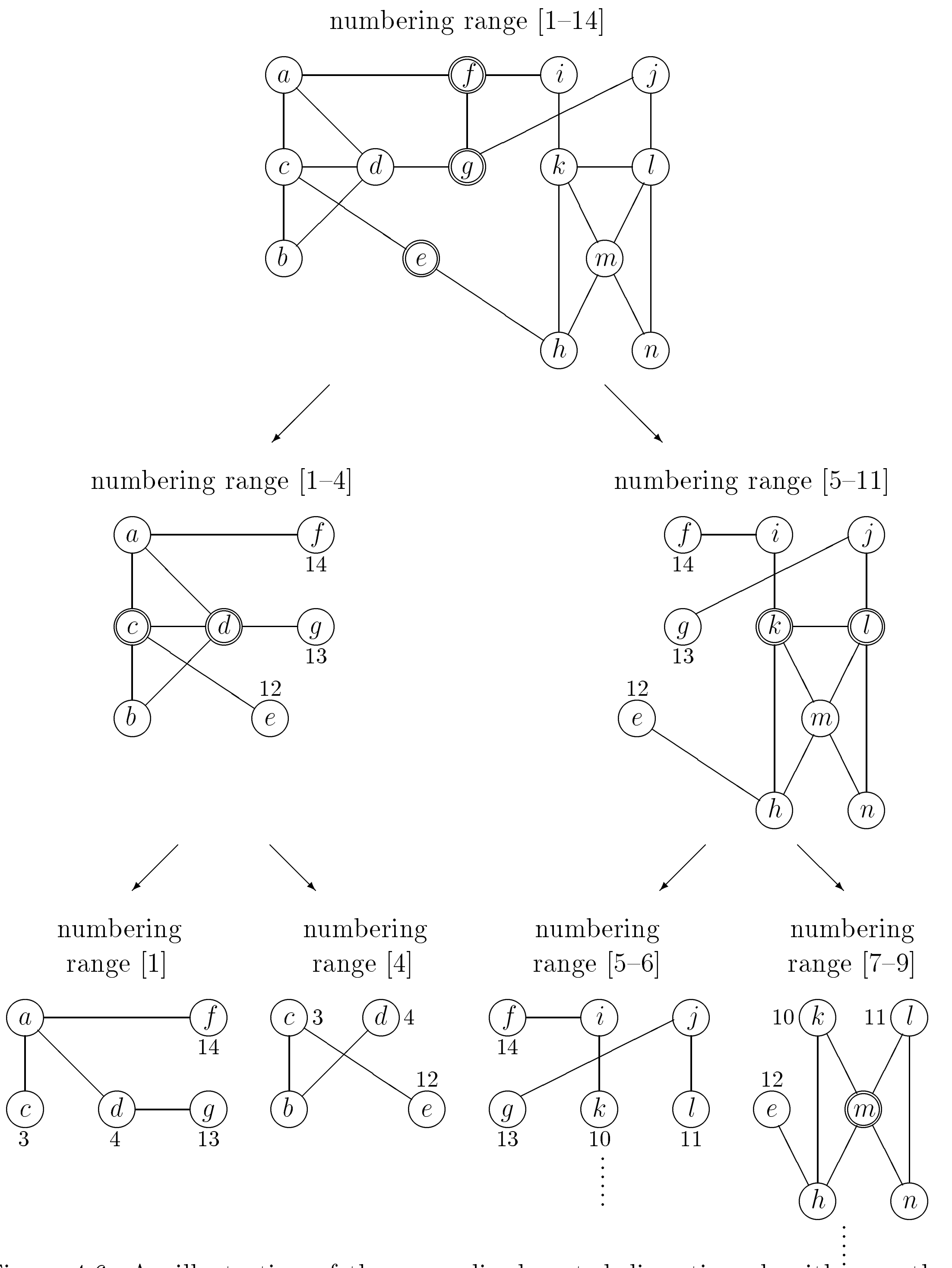


Figure 4.6: An illustration of the generalized nested dissection algorithm on the running example graph Q. The separator vertices for each problem are doubly-circled, while previously numbered vertices are indicated by their numbers.

tree is specified by a graph $G' = (V', E')$ and a set of vertices $N' \subseteq V'$ that have been previously numbered. The goal of the problem is to find a contiguous ordering for the unnumbered vertices. Note that when a problem is decomposed into two subproblems, those subproblems are not vertex disjoint since they both include the separator used to decompose the problem. However, the subproblems are edge-disjoint since the edges between vertices in the separator are omitted from them. In addition, previously numbered vertices in the problem that appear in a subproblem retain that status, and the separator vertices also gain that status. Applying these facts repeatedly, we have the following simple observation on the problems at each level of the recursion tree and a relation between a problem and its ancestor problems.

Lemma 4.20 *The graphs of the problems at a given level l of the recursion tree are pairwise edge-disjoint.*

Lemma 4.21 *Let $G_l = (V_l, E_l)$ be the graph of a problem at level l of the recursion tree and N_l be its set of previously numbered vertices. For each $1 \leq i \leq l-1$, let X_i be the separator used to decompose its level-i ancestor problem. Then G_l is the subgraph of $G - K(X_1) - \ldots - K(X_{l-1})$ induced by V_l. Furthermore, N_l is equal to $(X_1 \cup \ldots \cup X_{l-1}) \cap V_l$.*

In contrast, the problems at each recursion level of regular nested dissection are pairwise vertex-disjoint and consequently also pairwise edge-disjoint.

We note from Lemma 4.21 that the graph $G_l = (V_l, E_l)$ of a problem in the recursion tree is in general not equal to $G(V_l)$, the subgraph of the input graph induced by V_l, but is instead a proper subgraph of $G(V_l)$. For example, we see from Figure 4.6 that the graph for the problem on the vertices $\{a, c, d, f, g\}$ is the vertex-induced subgraph of Q on those vertices without the edges cd and fg. Note also that the lemma does not claim that there are no edges between the already numbered vertices. An edge may be present between two previously numbered vertices provided both vertices are not in one of the separators for the ancestor problems. Referring again to Figure 4.6, we see that in the subproblem on the vertices $\{a, c, d, f, g\}$, there is an edge between the two previously numbered vertices d and g, but they do not appear together in any of the ancestor separators $\{e, f, g\}$ and $\{c, d\}$.

Since the previously numbered vertices for a problem come from the separators for its ancestor problems, it is conceivable that they separate the unnumbered vertices from the rest of the input graph. We prove the result below. Note that the proof does not depend on the fact that the edges between vertices in a separator for a problem are omitted from its subproblems.

Lemma 4.22 *Let $G_l = (V_l, E_l)$ be the graph of a problem at level l of the recursion tree, and $N_l \subseteq V_l$ be its set of previously numbered vertices. There are no edges in the input graph G between the unnumbered vertices in G_l and the vertices that are not in G_l, i.e. N_l separates $V_l \setminus N_l$ from $V \setminus V_l$ in G.*

Proof. We prove the result by induction on l. The lemma is degenerately true at level 1. Assume that the lemma is true at level $l-1$.

Let $G_{l-1} = (V_{l-1}, E_{l-1})$ be the graph of the parent problem of the level-l problem under consideration. By construction, $V_l = A \cup X$, where X is the separator used to decompose G_{l-1} and A is the set of vertices in one of the two groups of the components of $G_{l-1} - X$. Let B denote the vertices in the other group.

Let v be an unnumbered vertex in G_l and $w \in V \setminus V_l$ be a vertex outside G_l. Then v is an unnumbered vertex in G_{l-1} and in fact v must be in A since the vertices in X are already numbered within G_l. If w is outside G_{l-1}, then the inductive hypothesis shows that vw is not an edge G. If w is inside G_{l-1}, then it must be in B since it is not in V_l, from which it follows that vw cannot be an edge in G since X separates v from w in G_{l-1}. ■

We next show that by including the separator within a subproblem, the subproblem captures the fill-in possibilities of its unnumbered vertices.

Lemma 4.23 *Let $G = (V, E)$ be the input graph and α be the ordering produced by generalized nested dissection. Let $G_l = (V_l, E_l)$ be the graph of a problem at level l of the recursion tree, and N_l be its set of previously been numbered vertices. For any unnumbered vertex v of G_l and a vertex $w \in V$, vw is an edge in the filled graph G^*_α only if $w \in V_l$.*

Proof. We again apply induction on the level l. At level 1, the only problem is on the input graph itself, and so the lemma holds degenerately. Assume inductively that the lemma holds at level $l-1$.

Let $G_{l-1} = (V_{l-1}, E_{l-1})$ be the graph of the parent problem of the level-l problem under consideration. By construction, $V_l = A \cup X$, where X is the separator used to decompose G_{l-1} and A is the set of vertices in one of the two groups of the components of $G_{l-1} - X$. Let B denote the vertices in the other group.

Let v be an unnumbered vertex in G_l, and suppose that vw is a fill edge. By construction, v must be in A. In addition, v is also an unnumbered vertex in G_{l-1}, and so w is a vertex of G_{l-1} by the induction hypothesis. Since X separates A from B in G_{l-1}, w cannot be in B. Thus w is in $A \cup X$, which is precisely V_l, and so the fill edge vw is in G_l, as desired. ■

The following observation is an immediate consequence of Lemma 4.21.

Lemma 4.24 *Let $G_l = (V_l, E_l)$ be the graph of a problem at level l and let X_i be the separator used to decompose its level-i ancestor problem, $1 \le i \le l-1$. Then for each $1 \le i \le l-1$, $X_i \cap V_l$ is an independent set of vertices in G_l.*

4.4.3 Generalized Nested Dissection on Special Classes of Graphs

We state but do not prove in this section some of the results given by Lipton, Rose and Tarjan [61] for generalized nested dissection on special classes of graphs. In addition, we shall present an example on which one of the results does not hold if the generalized nested dissection algorithm uses multi-way recursion instead of two-way recursion. We note that the proofs of these results employ the standard technique of bounding the quantity of interest using a recurrence relation that models the recursive process. In contrast, the corresponding analysis given by Gilbert and Tarjan [39] for planar and related graphs uses results on separator trees. The recurrence relation approach is applicable here because Lemma 4.23 shows that each subproblem completely captures the fill possibilities of its unnumbered vertices. Thus the fill for a problem can be expressed as the fill involving the separator, a quantity which can be bounded solely by the parameters of the problem, and the sum of the fills for the subproblems. We note also that the proofs do not use the fact that edges between vertices in the separator are omitted from the subsequent subproblems. The interested reader should refer to [61] for details of the proofs.

The classes of graphs that Lipton, Rose and Tarjan consider satisfy a separator theorem of the following form. Let S be a class of graphs that is closed under the subgraph relation, i.e. if $G \in S$ and G_1 is a subgraph of G, then $G_1 \in S$. The class S satisfies an $f(n)$-separator theorem if there are constants $\frac{1}{2} \leq \rho \leq 1$ and $\gamma > 0$ for which any n-vertex graph $G = (V, E)$ in S has the following property: there is a vertex separator X of size $|X| \leq \gamma n$ such that the components of $G - X$ can be grouped into two parts A and B, each with at most ρn vertices. Note that this says that the separator is small in addition to being ρ-balanced. For example, planar graphs satisfy a $\sqrt{n}$-separator theorem with $\rho = \frac{2}{3}$ and $\gamma = 2\sqrt{2}$ [62].

The following result gives fill and multiplication count bounds for classes of graphs with polynomial-sized balanced separators. It is assumed that these separators are used in the generalized nested dissection process.

Theorem 4.25 (Lipton, Rose & Tarjan [61]) *Let S be a class of graphs closed under the subgraph relation on which an n^σ-separator theorem holds. Then for any n-vertex graph G, any generalized nested dissection ordering has:*

(i) $O(n^{2\sigma})$ *fill and* $O(n^{3\sigma})$ *multiplication count if* $\sigma > \frac{1}{2}$,

(ii) $O(n \log n)$ *fill and* $O(n^{\frac{3}{2}})$ *multiplication count if* $\sigma = \frac{1}{2}$,

(iii) $O(n)$ *fill and* $O(n^{3\sigma})$ *multiplication count if* $\frac{1}{3} < \sigma < \frac{1}{2}$,

(iv) $O(n)$ *fill and* $O(n \log n)$ *multiplication count if* $\sigma = \frac{1}{3}$, *and*

(v) $O(n)$ *fill and* $O(n)$ *multiplication count if* $\sigma < \frac{1}{3}$.

Multi-Way Versus Two-Way Recursion

Gilbert and Tarjan [39] constructed a family of graphs which showed that the fact that there are two recursive subproblems for each problem in generalized nested dissection is essential to the fill bound given by Lipton, Rose and Tarjan. Each constructed graph V_k is nonplanar but sparse, with $n = k^2$ vertices and $O(m)$ edges, and is such that all subgraphs of it satisfy a $\sqrt{n}$-separator theorem. We assume that those $\sqrt{n}$ separators are used in generalized nested dissection. Thus Theorem 4.25 guarantees that any ordering produced by the algorithm has $O(n \log n)$ fill. However, if the dissection process was multi-way and instead recursed on the subproblems formed by each component and the separator, then Gilbert and Tarjan showed that one of the generalized nested dissection ordering has $\Omega(n^{\frac{5}{4}})$ fill.

Although the distinction between two-way and multi-way recursion makes no difference on general graphs, it critically affects the separators that are available when they are chosen according to a separator theorem. Indeed, for the bad dissection process on the graph V_k, using multi-way recursion decomposes the input problem into $k-1$ subproblems on k-vertex graphs. Each of the separators used for the subproblems has $\Theta(\sqrt{k})$ vertices. However, when two-way recursion is used, the $k-1$ subproblems are aggregated into two groups with $\Theta(k^2)$ vertices each. Although the union of the separators for the subproblems constituting an aggregate is also a separator for the aggregate, the union has $\Theta(k\sqrt{k}) = \Theta(\sqrt[3/2]{k})$ vertices, and so does not satisfy the separator theorem for the aggregate. The separators that can be used for the aggregate have size only $\Theta(k)$, which is substantially smaller than the size of the union.

In contrast, if separators were chosen using some criteria which does not include absolute bounds on their sizes (for example, via the Leighton-Rao balanced separator construction), then the union of the separators for a group of subproblems is likely to be an acceptable separator for the aggregate.

Gilbert and Tarjan also showed that some regular nested dissection ordering also has $\Omega(n^{\frac{5}{4}})$ fill on the graph V_k, regardless of whether two-way or multi-way recursion is used in the dissection process.

4.5 The Restricted Ordering Problem

As we saw in Section 4.4, the ordering subproblems that occur in the generalized nested dissection method of Lipton, Rose and Tarjan [61] have the property that some vertices have been previously numbered and therefore should not be renumbered. The previously numbered vertices are from the separators that were used earlier in the nested dissection process, and hence their assigned numbers are higher than those that will be given to the unnumbered vertices in a subproblem. It is natural and interesting to abstract these constrained subproblems into the following *restricted ordering problem.*

> *The Restricted Ordering Problem.* Let $G = (V, E)$ be a graph and $L \subseteq V$ be a subset of the vertices. Find a good elimination ordering of V in which the vertices in L are ordered last.

Observe that the normal ordering problem is the special case of the restricted ordering problem with $L = \emptyset$. It would seem therefore that the restricted ordering problem is more difficult to solve, but it is in fact not so. We shall present in this section a reduction for the restricted ordering problem that uses a surprisingly simple reduction from the restricted ordering problem to the normal problem. The reduction preserves any optimality properties of the corresponding normal ordering. Using our reduction to the restricted ordering problem, we will develop another variant of nested dissection in Section 4.6 and examine some of its properties.

4.5.1 A Reduction for the Restricted Ordering Problem

The restricted ordering problem closely resembles the fixed-last-vertices perfect elimination ordering problem on chordal graphs that we described in Section 4.2. Recall that in the latter problem, we wish to find a perfect ordering of a chordal graph such that a given set of *mutually adjacent* vertices is ordered last. We showed in Section 4.2 that such an ordering can be found by the lexicographic search algorithm of Rose, Tarjan and Leuker [76]. These special perfect elimination orderings are the key ingredient in our reduction for the restricted ordering problem, as we shall see next.

We first introduce some terminology for describing the reduction. Let $G = (V, E)$ be the graph that we wish to order, and $L \subseteq V$ be the set of vertices that are to appear last in the ordering. We say that an ordering π of V is a L-restricted ordering if it numbers the vertices in L last. Recall from Section 4.4.1 that $K(L)$ denotes the clique on the vertices in L. We define the *encoded graph* $G + K(L)$ to be G with the addition of edges between each pair of vertices in L. Thus the vertices of L form a clique in $G + K(L)$. Throughout this section we reserve the symbols π and σ for L-restricted orderings, and the symbols α and β for normal orderings. We assume that the graph G is connected.

We find a good restricted ordering for the graph G and the set of last vertices L as follows:

1. Construct the encoded graph $G + K(L)$.

2. Find a good normal ordering α for $G + K(L)$.

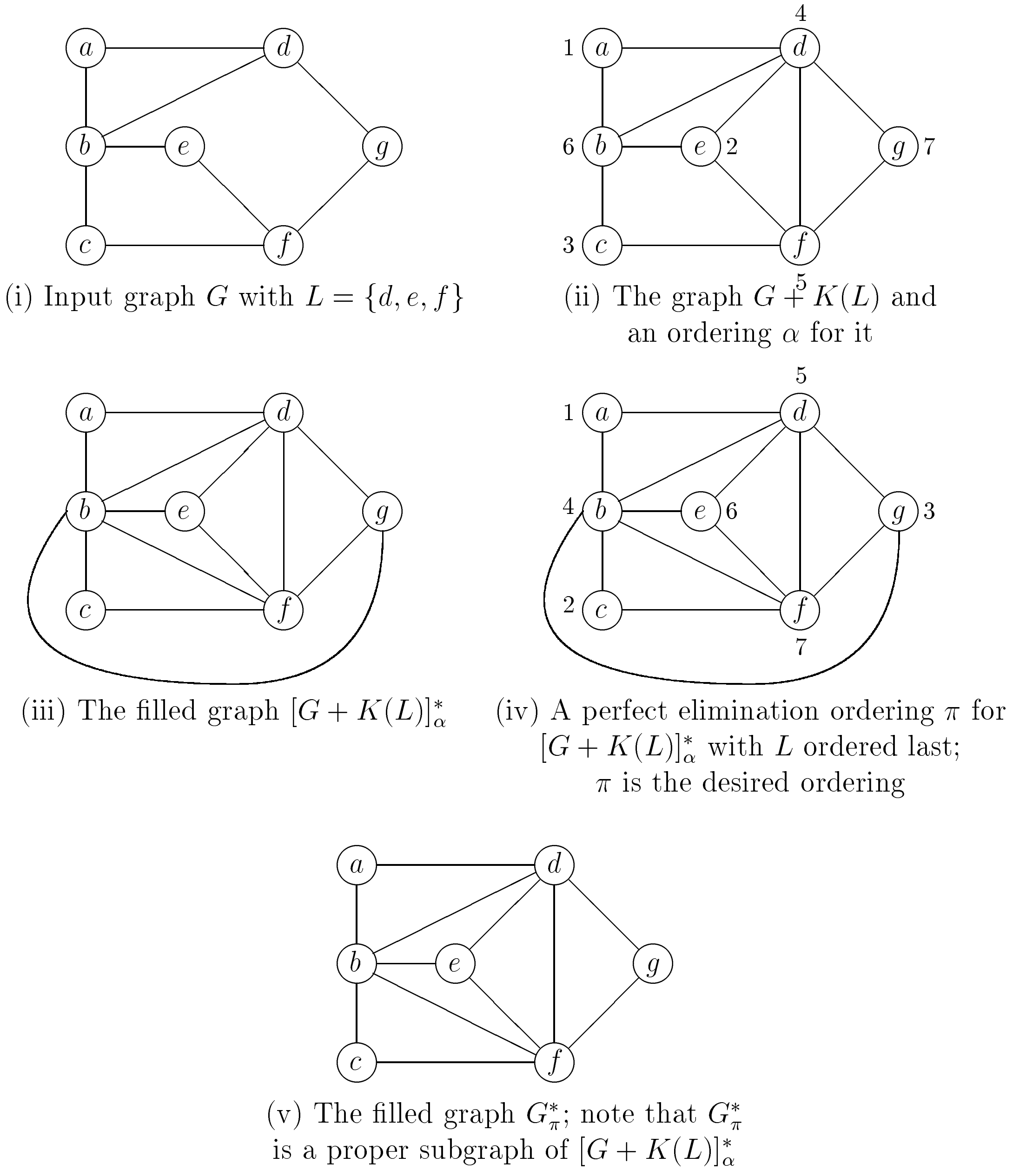
3. Eliminate $G + K(L)$ using the ordering α to obtain the filled graph $(G + K(L))^*_\alpha$. Note that since the vertices in L form a clique in $G + K(L)$, they also form a clique in $(G + K(L))^*_\alpha$.

4. Using lexicographic search, find a perfect ordering π for $(G + K(L))^*_\alpha$ that orders the vertices in L last. π is the desired L-restricted ordering for G.

Figure 4.7 illustrates the above construction on a small example graph.

4.5.2 Discussion of the Restricted Ordering Construction

We now examine the properties of the construction. We shall see that the constructed L-restricted ordering π and the intermediate normal ordering α are closely related, and that π possesses whatever optimality properties that α has. In essence,



(i) Input graph G with $L = \{d, e, f\}$

(ii) The graph $G + K(L)$ and an ordering α for it

(iii) The filled graph $[G + K(L)]^*_\alpha$

(iv) A perfect elimination ordering π for $[G + K(L)]^*_\alpha$ with L ordered last; π is the desired ordering

(v) The filled graph G^*_π; note that G^*_π is a proper subgraph of $[G + K(L)]^*_\alpha$

Figure 4.7: An illustration of the reduction for the restricted ordering problem.

we can regard the ordering π as being "hidden" in α, and that it is "recovered" from α by lexicographic search.

We first consider the definition of the encoded graph $G+K(L)$. From our earlier discussion of the lexicographic search algorithm, we know that for the algorithm to produce a perfect elimination ordering that numbers the vertices in L last, we have to present it with a chordal graph where the vertices in L forms a clique. Since filled elimination graphs are chordal, they are natural candidates for this purpose. However, the vertices in L are unlikely to be mutually adjacent in the filled graph G^*_β obtained from each normal ordering β for G. By our definition of the graph $G + K(L)$, we trivially ensure that *any* of its filled graphs will have a clique on L and can be used to recover a perfect elimination ordering. Thus there are no constraints on the normal orderings that are used in the construction. The graph $G + K(L)$ can be thought of as "encoding" the requirement that vertices in L be numbered last.

There is another justification for using the encoded graph $G + K(L)$ in the construction. Suppose that the graph G and the set L satisfy the following connectivity constraint: the subgraph induced by the vertices in $V \setminus L$ is connected and each vertex in L is adjacent to some vertex not in L. Such a scenario holds in the Lipton-Rose-Tarjan generalized nested dissection algorithm, in which the separator vertices adjacent to a component are included in the subproblem for that component. Let σ be *any* L-restricted ordering for G. By Lemma 4.9, we know that the vertices in L form a clique in the filled graph G^*_σ. In fact, a stronger result holds.

Lemma 4.26 *Assume that the graph G and the set of last vertices L satisfy the connectivity constraint. For any L-restricted ordering σ of the graph G, the filled graphs G^*_σ and $(G + K(L))^*_\sigma$ are identical.*

We see from Lemma 4.26 that it is equivalent to work with the graph $G + K(L)$ instead of G.

We now relate the L-restricted ordering π produced by the construction and the intermediate normal ordering α. Since G is a subgraph of the filled graph $(G + K(L))^*_\alpha$, and π is a perfect elimination ordering of that filled graph, we have the following result.

Lemma 4.27 *The filled graph G^*_π is a subgraph of the filled graph $(G + K(L))^*_\alpha$.*

The subgraph relationship in the above lemma may be proper, as seen from the earlier example in Figure 4.7.

Finally, we show that π enjoys whatever optimality properties that α has.

Lemma 4.28 *Assume that the graph G and the set of last vertices L satisfy the connectivity constraint. If α is a minimum (minimal) ordering for $G+K(L)$, then π is a minimum (minimal) L-restricted ordering for G and $G^*_\pi = [G + K(L)]^*_\alpha$.*

Proof. We prove the lemma by contradiction. Suppose π is not a minimal L-restricted ordering for G. There is a L-restricted ordering σ whose filled graph G^*_σ

is a proper subgraph of the filled graph G^*_π. By Lemma 4.27, G^*_π is a subgraph of $(G + K(L))^*_\alpha$, and so G^*_σ is a proper subgraph of $(G + K(L))^*_\alpha$. However, by Lemma 4.26, $(G + K(L))^*_\sigma = G^*_\sigma$, contradicting the minimality of α. A similar contradiction argument holds when α is a minimum ordering for $G + K(L)$. ■

4.6 Restricted Nested Dissection

In our reduction for the restricted ordering problem in the previous section, we construct the desired restricted ordering by first finding a normal ordering for the encoded graph, and then recovering the restricted ordering via elimination and lexicographic search. We can in fact use a divide-and-conquer approach to find the intermediate normal ordering, and this leads to a variant of nested dissection that we shall call *restricted nested dissection.* In restricted nested dissection, the separator vertices adjacent to a component are included in the subproblem for that component, as in the generalized nested dissection scheme of Lipton, Rose and Tarjan [61]. However, instead of not assigning numbers to those separator vertices present in a subproblem, we ensure by our reduction for the restricted ordering problem that they are numbered last. The numbering of those separator vertices are then discarded upon returning to the higher-level problem.

4.6.1 The Restricted Nested Dissection Algorithm

We give below our variant of nested dissection. Given a graph G and a set L of last vertices, and a range $[a, b]$ with $b = a + n - 1$, the routine RESTRICTEDND produces a nested dissection ordering in the range $[a, b]$, with the vertices in L numbered last. The routine assumes that the vertices in L form a clique in the given graph G (degenerately true when $L = \emptyset$). This assumption is satisfied by the recursive calls that are made. To number an initially unnumbered graph G, we make the call RESTRICTEDND$(G, \emptyset, 1, n)$.

```
RESTRICTEDND(G = (V, E), L, a, b)
if G is a clique then
    arbitrarily number the vertices in V \ L from a to b − |L|
    arbitrarily number the vertices in L from b − |L| + 1 to b
else
    /* find a nested dissection ordering α from 1 to n for G + K(L) = G */
    find a 2/3-balanced separator X for the graph G
    let G − X have k components A_1, ..., A_k with n_1, ..., n_k vertices
    for i = 1, ..., k
        let X_i be the vertices in X that are adjacent to A_i
        call RESTRICTEDND(G_X(V_i ∪ X_i), X_i, 1 + Σ_{j=1}^{i-1} n_j, Σ_{j=1}^{i} n_j + |X_i|),
            where G_X(A_i ∪ X_i) denotes the subgraph of the encoded graph G_X
            induced by the vertices in A_i and X_i
```

discard the numbers assigned to vertices in X_i
endfor
number the vertices in X arbitrarily from $n - |X| + 1$ to n

/ recover desired L-restricted ordering for $G + K(L) = G$ */*
eliminate G using the ordering α to obtain the filled graph G^*_α
find a perfect ordering π in the range $[a, b]$ for G^*_α that orders L last
for each vertex v
assign the number $\pi(v) + a - 1$ to v
endfor
endif

In the general case, the restricted dissection algorithm models the reduction that we gave in Section 4.5 for the restricted ordering problem to find the desired ordering. It produces the intermediate ordering α for $G + K(L)$, which by assumption is the same as G, by decomposing the graph G using an appropriate separator X and then calling itself on the subproblems. Each subproblem is defined by the subgraph of G on the vertices in a component A_i and the separator vertices X_i adjacent to it. The goal of the subproblem is to find an ordering which numbers the vertices in X_i last. To do this, the algorithm also makes the vertices of X_i into a clique within the subproblem graph, before calling itself. The orderings for the subproblems are then composed, with the numbers assigned by the subproblems to the separator vertices discarded. Having found the intermediate ordering α, the algorithm then performs the elimination and lexicographic search steps to recover the desired ordering π from it.

We trace in Figure 4.8 the restricted nested dissection algorithm on the running example from earlier sections.

4.6.2 Comparison of Restricted Nested Dissection to Generalized Nested Dissection

We compare and constrast the subproblems that occur in the restricted nested dissection algorithm with those in the generalized nested dissection method of Lipton, Rose and Tarjan [61] described in Section 4.4. Observe that for our algorithm, the vertices to be numbered last in a subproblem are those of the separator X used in the current decomposition. Since the subsequent elimination and lexicographic search steps recover an L-restricted ordering from the normal ordering composed from the subproblems, the separators from ancestor problems do not have a direct influence on the subproblem. In contrast, we saw earlier in Section 4.4 that the already-numbered status of separator vertices propagate down the subproblems of the recursion tree for generalized nested dissection. Each subproblem there potentially contains vertices from all ancestor separators used at previous recursion levels, and those vertices are not to be assigned numbers.

We also saw that the subproblems at a given recursion level for generalized nested dissection are edge-disjoint. This property is clearly not true for restricted

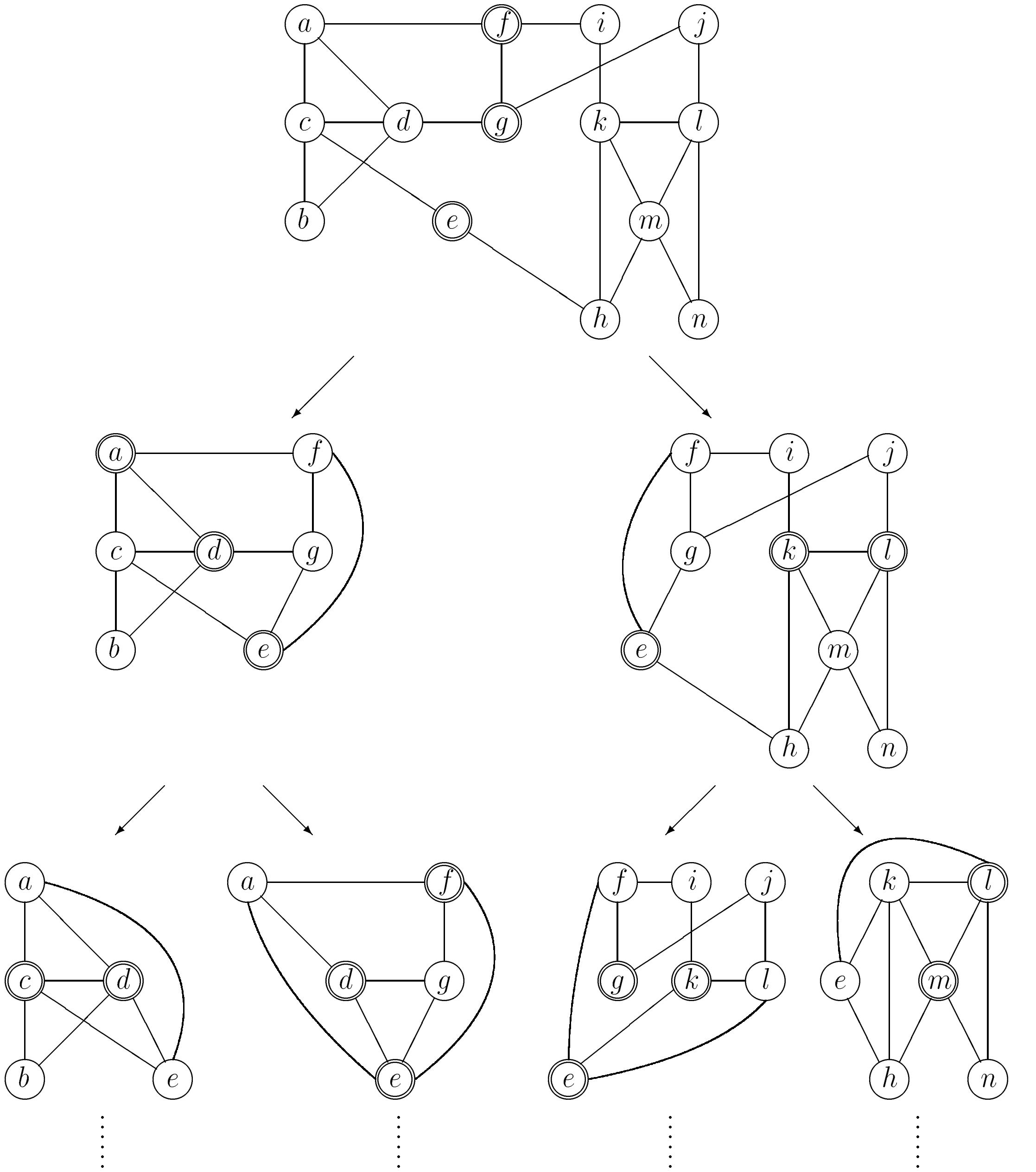


Figure 4.8: An illustration of the restricted nested dissection algorithm on the running example graph Q. The separator vertices for each problem are doubly-circled.

nested dissection, given the introduction of the clique on the separator vertices. In fact, the addition of edges has both theoretical and computational consequences. On the theoretical end, a class of graphs with special separator structures, eg. planar graphs, may not be closed under the addition of edges. The addition of edges also results in subproblems with many more edges, which are computationally more expensive to process.

4.6.3 Optimality of Restricted Nested Dissection

We show in this section that the restricted nested dissection algorithm is existentially optimal in that by choosing the appropriate separators to use, it can produce a minimum elimination ordering for an unnumbered graph. This result contrasts with that given in Section 4.3.2 for regular nested dissection, in which Gilbert [37] showed that there is an ordering that is nearly optimal. Therefore, restricted nested dissection has existential optimal performance on general graphs, whereas regular nested dissection has only suboptimal performance.

We first precisely state the optimality result that we shall prove. Throughout this section, we assume as usual that the graph G is connected.

Theorem 4.29 *Let H be any minimal chordal extension of a graph G. There is a restricted nested dissection ordering π of G for which the filled graph G^*_π is identical to H.*

Since any minimum chordal extension of G is also minimal, Theorem 4.29 guarantees that a minimum elimination ordering for G can be found by restricted nested dissection. It follows that in the best case, restricted nested dissection is optimal with respect to fill.

The proof of Theorem 4.29 uses ideas similar to those used to prove the suboptimality of regular nested dissection on general graphs. In particular, we exploit the fact that every chordal graph has a clique separator. It is not necessary for our proof that the separator be $\frac{1}{2}$-balanced or that its associated components have at most $\frac{2}{3}m$ edges, as was the case in the nested dissection analysis given in Section 4.3.2. We also use the fact that any vertex-induced subgraph of a chordal graph is also chordal.

The following lemma shows that the minimality of a chordal extension is preserved in subgraphs resembling those that occur in restricted nested dissection. Note that a separator for a chordal extension of G is also a separator for G.

Lemma 4.30 *Let H be a minimal chordal extension of a graph G, and X be any clique separator for H. Let A be a component remaining when X is removed from G. The subgraph H_{AX} of H induced by the vertices in A and X is a minimal chordal extension of the corresponding vertex-induced subgraph G_{AX} of $G + K(X)$.*

Proof. Suppose for the sake of contradiction that H_{AX} is not a minimal chordal extension of G_{AX}. Then there exists a chordal extension $\tilde{H}_{AX}$ of G_{AX} which is a proper subgraph of H_{AX}. Let $H_{\bar{A}}$ be the (chordal) subgraph of H induced by the vertices not in A. We claim that the union $\tilde{H}$ of the subgraphs $\tilde{H}_{AX}$ and $H_{\bar{A}}$ is a chordal extension of G. This contradicts the minimality of H since X forms a clique in both $H_{\bar{A}}$ and $\tilde{H}_{AX}$, and $\tilde{H}_{AX}$ is a proper subgraph of H_{AX}.

Let μ be a cycle in $\tilde{H}$ of length at least 4. We need to show that μ has a chord. If μ contains only vertices from A and X, then μ is a cycle in the chordal graph $\tilde{H}_{AX}$, and so it has a chord. Similarly, if μ does not contain any vertices from A, then it has a chord since it is also a cycle in $H_{\bar{A}}$. Suppose now that μ contains a vertex v in A and a vertex w that is not in A and also not in X. Since X is a separator for the graph $\tilde{H}$, it follows that μ must contain two vertices of X, and these two vertices are not adjacent in μ. By the completeness of X, there is an edge joining those two vertices, and so μ has a chord.

Thus the graph $\tilde{H}$ is chordal, yielding the desired contradiction. ■

The next lemma is a slight variant of Lemma 4.30, in which only the separator vertices adjacent to the component A are used in inducing the two subgraphs. The subgraph of $G + K(X)$ in Lemma 4.31 is therefore a subproblem in restricted nested dissection. Lemma 4.31 follows easily from Lemma 4.30 by using the fact that X_A separates A from $X \setminus X_A$ in the subgraph of H induced by $A \cup X$.

Lemma 4.31 *Let H be a minimal chordal extension of a graph G, and let X be any clique separator for H. Let A be a component remaining when X is removed from G, and X_A be those vertices in X that are adjacent to A in G. The subgraph H_{AX} of H induced by the vertices in A and X_A is a minimal chordal extension of the corresponding vertex-induced subgraph G_{AX} of G_X.*

Using Lemma 4.31, we can prove a stronger form of Theorem 4.29. This stronger form admits a proof by induction on the number of vertices in G.

Theorem 4.32 *Let H be any minimal chordal extension of a graph G, and let L be any clique in G. There is a restricted nested dissection ordering π of G that numbers the vertices in L last, and for which the filled graph G^*_π is identical to H.*

Proof. If G is complete, then the theorem is trivially true since the only chordal extension of G is itself. Suppose that G is not complete. Let X be a clique separator for H, and assume that the restricted nested dissection algorithm chooses X as the separator for G. Let A_i be the components remaining when X is removed from G, and let X_i be the vertices in X that are adjacent to A_i.

Consider the subproblems that the algorithm recurses on. Recall that G_X is the graph obtained from G by adding edges to each pair of vertices in X. The graph of G_i of the subproblem i is the subgraph of G_X induced by the vertices in A_i and X_i. By Lemma 4.31, the subgraph H_i of the graph H induced by the vertices in A_i and X_i is a minimal chordal extension of G_i. Since X_i forms a clique in G_i, it follows from the inductive hypothesis that there is a restricted nested dissection ordering π_i for G_i that numbers the vertices in X_i last, and for which the filled graph $(G_i)^*_{\pi_i}$ is identical to H_i.

The restricted nested dissection algorithm composes the orderings π_i into an ordering α such that the vertices in A_i are numbered consecutively in the order given by π_i, and are numbered before the vertices in a component A_j with $j > i$. In addition α numbers the vertices in X last. Consider eliminating the graph $G + K(L) = G$ in the order α to obtain the filled graph $(G + K(L))^*_\alpha$. For each component A_i, it is easy to verify that the subgraph of $(G+K(L))^*_\alpha$ induced by the vertices in A_i and X_i is a subgraph of the corresponding vertex-induced subgraph of H. Thus $(G + K(L))^*_\alpha$ must be a subgraph of H.

Finally, the algorithm recovers via lexicographic search an ordering π for G that numbers the vertices in L last. By Lemma 4.27, it follows that the filled graph G^*_π for the recovered ordering π is also a subgraph of H. But H is a minimal chordal extension of G, and G^*_π is also chordal. Thus G^*_π is identical to H, and π is the desired ordering. ■

4.7 Experimental Results for Nested Dissection Methods

We present in this section an empirical evaluation of the various nested dissection methods discussed in this chapter. To judge the effectiveness of these methods, we compare them against the minimum degree heuristic, which is widely used in sparse matrix packages. This heuristic was described in Section 1.4.1, and uses the simple rule of eliminating at each step a vertex of minimum degree. We remind the reader that there are no known guarantees on the quality of the ordering produced by the heuristic. However, it turns out that the minimum degree heuristic is surprisingly difficult to beat in practice.

In addition, we also compare the nested dissection methods against the variant found in the sparse matrix package SPARSPAK [35]. The SPARSPAK variant does not use approximately optimal balanced separators, but instead employs the breadth-first search based heuristic described in Section 3.6.2 to construct separators. The separators found by this heuristic do not have any balancedness or size guarantees, and lead to decidely inferior nested dissection orderings. The results strongly suggest that the quality of the separators used in nested dissection has a direct impact on the quality of the ordering produced.

We first describe the setup for our experimental runs in Section 4.7.1, followed by some preliminary results on enhancements for each ordering method in Section 4.7.2. The main results on the fill, multiplication count and elimination height of the ordering methods are presented and discussed in Section 4.7.3, and are accompanied in Section 4.7.4 by a more detailed examination of the recursive behavior and the subproblems of the nested dissection methods. A summary of the five ordering methods is also given. Finally, we evaluate in Section 4.7.5 the use of balanced low-deficiency separators in place of approximate optimal balanced separators.

4.7.1 Experimental Setup

Implementation Enhancements

In our implementation, we included two simple enhancements for simplifying a graph before an elimination ordering is computed. The first enhancement is the removal of simplicial vertices. A simplicial vertex is a vertex whose neighbors form a clique in the graph. So eliminating such a vertex first does not create any fill-in. Note that a vertex which is not simplicial may become so after one or more simplicial vertices have been removed. We therefore perform multiple passes over the graph, removing simplicial vertices and numbering them first, until no new simplicial vertices are found.

The second enhancement is the use of articulation points to decompose a graph. An articulation point is a special separator that suffers no fill when it is used to decompose the graph. In fact, a minimum chordal extension of a graph with an articulation point can be found by composing minimum chordal extensions of the subgraphs formed by the components and the articulation point. So if a graph has an articulation point, we number it last and then recursively number the components caused by the removal of that articulation point.

After articulation points (if any) have been removed from an input graph, no new ones appear in the subproblems for restricted nested dissection since the included separator vertices form cliques in those subproblems. However, for regular and generalized nested dissection, new articulation points may be present in the subproblems. For example, if the input graph is a cycle, it has no articulation points and is decomposed by removing two nonadjacent vertices. The resulting subproblems for regular/generalized nested dissection are paths, all of whose internal vertices are articulation points, while the subproblems for restricted nested dissection are again cycles.

Test Cases

For our evaluation of the ordering methods, we use the graphs of 33 matrices from the standard Harwell-Boeing collection of sparse matrices [18]. This set of 33 matrices is identical to that used in Chapter 2 for tuning and evaluating the concurrent flow implementation and in Chapter 3 for evaluating the various cut constructions. In the subsequent presentation, we will not make a distinction between a matrix and its graph.

Table 4.1 gives the characteristics of the 33 matrices used. For each graph, we list the number of vertices and edges, and the minimum, average and maximum degree. In addition, we give the number of simplicial vertices in the graph as found by the multi-pass algorithm described earlier, and the number of articulation points in the graph, before and after the removal of simplicial vertices.

We see that the graphs generally have low average degree, although in some instances (eg. FS541.1, SHL200 and BP1000), the maximum degree is very high. In addition, about half the graphs have simplicial vertices, and they include those graphs that have articulation points. For the graphs BCSPWR05 and BCSPWR06,

Table 4.1: Characteristics of matrices used in evaluating the nested dissection methods.

Matrix	n	m	Degree			# Simp	# Arti Points	
			Min	Avg	Max	Vertices	Before	After
CAN96	96	336	5	7.0	8	0	0	0
CAN144	144	576	5	8.0	14	48	0	0
CAN187	187	652	4	7.0	9	0	0	0
CAN229	229	774	2	6.8	12	18	0	0
CAN292	292	1124	3	7.7	34	6	0	0
CAN445	445	1682	3	7.6	12	0	0	0
CAN715	715	2975	1	8.3	104	19	2	0
BCSSTK04	132	1758	14	26.6	45	0	0	0
BCSSTK05	153	1135	3	14.8	23	0	0	0
BCSSTK06	420	3720	1	17.7	27	60	4	0
BCSSTK19	817	3018	1	7.4	10	48	18	0
BCSSTM07	420	3416	1	16.3	25	56	4	0
BCSPWR05	443	590	1	2.7	9	166	101	3
BCSPWR06	1454	1923	1	2.6	12	477	326	7
DWT193	193	1650	7	17.1	29	15	0	0
DWT209	209	767	3	7.3	16	0	0	0
DWT310	310	1069	3	6.9	10	3	0	0
DWT419	419	1572	5	7.5	12	0	0	0
DWT592	592	2256	3	7.6	14	4	0	0
DWT878	878	3285	3	7.5	9	9	1	1
NOS4	100	247	1	4.9	6	7	7	0
NOS5	468	2352	6	10.1	22	0	0	0
NOS6	675	1290	2	3.8	4	0	0	0
NOS7	729	1944	3	5.3	6	0	0	0
LSHP265	265	744	3	5.6	6	0	0	0
LSHP406	406	1155	3	5.7	6	0	0	0
LSHP577	577	1656	3	5.7	6	0	0	0
LSHP778	778	2247	3	5.8	6	0	0	0
WILL199	199	660	2	6.6	13	0	0	0
ASH292	292	958	3	6.6	13	19	0	0
FS541.1	541	2466	5	9.1	540	0	0	0
SHL200	663	1720	2	5.2	439	1	0	0
BP1000	822	4635	2	11.3	307	0	0	0

more than 30% of their vertices are simplicial, and these graphs also have a large number of articulation points. The number of articulation points for these graphs drops drastically after simplicial vertices have been removed, as is the case with all but one (NOS7) of the other graphs that have articulation points. An articulation point disappears because all but one of the components that it decomposes the graph into contains only simplicial vertices. Therefore for purposes of our implementation, which applies simplicial vertex removal before articulation point decomposition, there are only 3 graphs (BCSPWR05, BCSPWR06 and DWT878) with articulation points.

For each of the nested dissection methods discussed in this chapter, we ran it with and without articulation point and simplical vertex processing, and also using multi-way or two-way recursion where appropriate (see Section 4.7.2). For each setting of a nested dissection method and for each graph, ten runs were performed by varying the seeding of a pseudorandom number generator, which controls the random choices in the Leighton-Rao balanced separator construction. For each graph, there is therefore a direct correspondence between the 10 runs of each nested dissection method, in that the set of 10 top-level separators used to decompose the graph is the same for each method.

As for the minimum degree algorithm and SPARSPAK's nested dissection, we performed 1000 runs on each graph. These 1000 runs presented the deterministic implementations of both of those algorithms with the original input graph and 999 vertex-permuted versions of it. Note that permuting the vertices of a graph can have a great impact on these algorithms, since it changes the order in which vertices are processed. We also ran both algorithms another 1000 times with articulation point and simplicial vertex processing in place for the matrices with such features.[2] We performed such a large number of runs for these two methods as they are overwhelmingly faster than our nested dissection methods, which use the computationally intensive Leighton-Rao balanced separator construction, often by three or more orders of magnitude.

In the subsequent comparison of the nested dissection methods, we shall abbreviate the Agrawal-Klein-Ravi algorithm of Section 4.3 as the *AKR algorithm*, the Lipton-Rose-Tarjan generalized nested dissection algorithm of Section 4.4 as the *LRT algorithm*, and the restricted nested dissection algorithm of Section 4.6 as the *RND algorithm*.

4.7.2 Preliminary Comparisons

Before comparing the various ordering methods to each other, we first examine the effects of articulation point and simplicial vertex processing, and of using multi-way or two-way recursion. We shall see that these variations have minor effect on

[2] If an articulation point is found, it is used in conjunction with restricted nested dissection to decompose the graph. Simplicial vertex processing and articulation point decomposition are applied until no further simplification is possible, and then the minimum degree algorithm or SPARSPAK's algorithm is used to order the remaining components.

the performance of each method, and consequently it is not necessary to do a two-dimensional comparison of the various ordering methods under different settings.

Runs With and Without Articulation Point and Simplicial Vertex Processing

Ideally, the use of articulation point and simplicial vertex processing will improve the performance of an ordering method, since the processing simplifies the graph without introducing any fill-in, as was discussed in Section 4.7.1. However, in practice, the processing may be detrimental since the "good" balanced separators in the graph may become unbalanced in the simplified graph. We summarize below our comparison of the average fill, multiplication count and elimination height of runs with the processing at all recursion levels to the corresponding averages of the runs without any such processing.

- For the AKR algorithm, there were only 4 matrices out of 33 where the processing reduced one or more of the averages, and the achieved reduction was at most 3.5%. In contrast, the with-processing averages for most of the matrices were worse, and were typically up to 8% higher. There was one matrix where the multiplication count increased by 18% and 4 matrices (including the 3 graphs BCSPWR05, BCSPWR06 and DWT878 with articulation points) where the height increased by 15% or more.

- For the LRT algorithm, the results were mixed. The with-processing averages were usually within ±5% of their counterparts, although there were 4 matrices (including the graph BCSPWR06 and DWT878) where the height went up by 10% or more.

- For the RND algorithm, the results were also mixed and the averages usually fell within the ±5% range, but there were 2 graphs (including the graph BCSPWR05) where the processing reduced the height by more than 15%.

- For the minimum degree algorithm, the processing is only applicable to input graphs with articulation points and/or simplicial vertices. For such matrices, the with- and without-processing averages were usually within 5% of each other, and were always within 15% of each other.

- For SPARSPAK's variant of nested dissection, the processing is also restricted to input graphs with articulation points and/or simplicial vertices. For such matrices, the ±5% observation again applied. However, for the graphs BCSPWR05 and BCSPWR06, which have articulation points, the processing improved the averages by more than 15%, and even as much as 40%. These significant improvements hint at the impact of good separators on the quality of the ordering found by divide-and-conquer approaches.

Given these mixed results, it is clear that articulation point and simplicial vertex processing cannot be uniformly applied across all the test graphs. As our goal is to compare the various nested dissection methods and not to evaluate simple performance-improving enhancements, we did try not other combinations involving the use of articulation point decomposition and/or simplicial vertex removal at none, one or more of the recursion levels. To account for the occasional improvement from such processing, we shall use the better of the with- and without-processing statistics for the minimum degree algorithm and the SPARSPAK algorithm when comparing the various ordering methods later on. As for the AKR, LRT and RND algorithms, we use the results from runs where no such processing was performed. The reason for this is that the processing complicates the statistics for the recursion processes of these methods, and we would like to compare their recursive behavior. Our comparison is therefore slightly biased against the AKR, LRT and RND algorithms, although we note that our overall conclusions regarding the ordering methods do not change if we also use the better of the two sets of statistics for those algorithms.

Multi-Way Versus Two-Way Recursion

As we pointed out in Section 4.4, the generalized nested dissection algorithm of Lipton, Rose and Tarjan [61] does not form a subproblem from each of the components remaining in the graph after the removal of a separator, but instead collects the components into two groups to form two subproblems. The two-way recursion is essential for proving fill and multiplication bounds on classes of graphs with separator theorems. It is therefore interesting to see if this theoretical necessity has any practical impact in general.

Figure 4.9 compares the LRT algorithm (without articulation point and simplicial vertex processing) with multi-way recursion and with two-way recursion. In the figure, we have plotted the ratio of the average fill/multiplication count/height obtained by the multi-way version to the corresponding averages for the two-way version. We see that the behavior of the LRT algorithm was virtually identical for both forms of recursion, without one form clearly being better than the other. This is probably due to the fact that we are using separators from the Leighton-Rao balanced separator construction. In contrast, the proofs of performance bounds that Lipton, Rose and Tarjan gave for graphs with separator theorems assume that those special separators were used. It is quite unlikely that our test graphs possess such separator theorems, although we did not test for them (eg. by testing for planarity).

We also compared the averages for the RND algorithm with multi-way and with two-way recursion. The results were again mixed, with the multi-way and two-way averages almost always within 2% of each other. We note that similar comparisons were seen for the LRT and the RND algorithms with articulation point and simplicial vertex processing in place.

As for regular nested dissection, the recursion distinction is irrelevant since using two-way recursion will create subproblems on disconnected graphs, which are then partitioned using null separators into their constituent components across one or more recursion levels

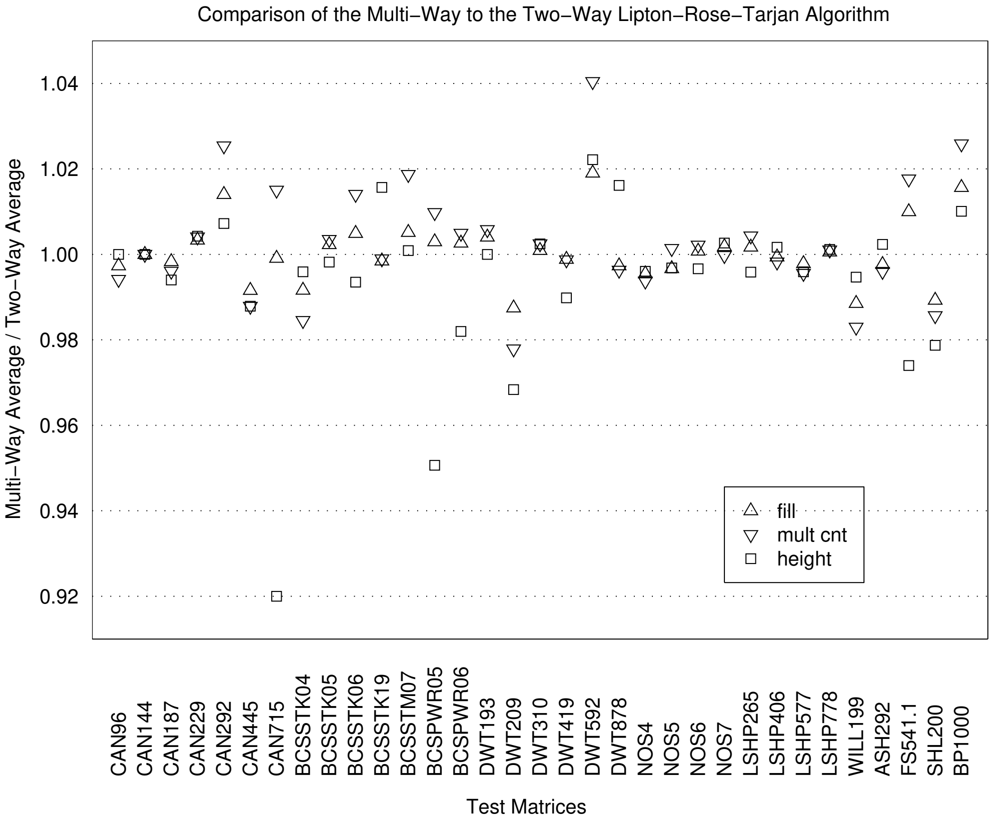


Figure 4.9: Comparison of the average fill, multiplication count and elimination height for the multi-way to the two-way Lipton-Rose-Tarjan algorithm.

Given the virtually identical behavior of multi-way and two-way recursion, we arbitrarily chose multi-way recursion for all subsequent runs and comparison of the nested dissection methods.

Variations Across Runs

We now examine the variation of the three statistics (fill, multiplication count and height) across the multiple runs performed on each graph. Specifically, we computed for each graph the ratio of the highest fill to the lowest fill of an ordering produced, and also the corresponding ratios for the multiplication count and height. For the minimum degree algorithm, the highest-to-lowest fill ratio for a graph varied between 1.0 to 1.3, while the multiplication count ratio varied between 1.0 to 1.7, and the height ratio varied between 1.04 and 2.4. These results corroborate the observations of George and Liu [34] that the minimum degree algorithm is very sensitive to how ties are broken among vertices with the minimum degree, thus leading to high variation in the quality of the ordering.

As for the SPARSPAK algorithm, the fill ratio varied between 1.02 and 2.1, the multiplicaton count ratio varied between 1.03 and 4.9, and the height ratio varied between 1.0 and 2.1. The wide variation is due to SPARSPAK's use of a heuristic for finding separators.

In contrast, the variations for the Leighton-Rao based nested dissection methods were lower for fill and multiplication count, and much lower for the height:

- For the AKR algorithm, the highest fill was at most 16% above the lowest, the highest multiplication count was at most 30% above the lowest, and the highest height was at most 12% above the lowest.

- For the LRT algorithm, the highest-to-lowest fill ratio was at most 14%, the multiplication count ratio was at most 25%, and the height ratio was at most 19%.

- For the RND algorithm, the fill ratio did not exceed 19%, the multiplication count ratio did not exceed 24%, and the height ratio did not exceed 50%.

We note that the average-to-lowest ratio is typically about half of the highest-to-lowest ratio.

Since the minimum degree and SPARSPAK algorithms are overwhelmingly faster than the AKR, LRT and RND methods, we use the best orderings produced (with respect to fill, multiplication count or height) by these algorithms in subsequent comparisons. As for the AKR, LRT and RND algorithms, we use the averages of these statistics over the runs of those algorithms.

4.7.3 Results for Nested Dissection with Leighton-Rao Separators

We now compare the performance of the various nested dissection methods and of the minimum degree algorithm, using the choice of setting as described in the previous section. We shall examine the standard performance criteria of fill, multiplication count and elimination height. The statistics for the recursion processes generated by the AKR, LRT and RND algorithms will be presented in Section 4.7.4. We remind the reader that for each statistic, we use in the comparisons the lowest values for the minimum degree algorithm and the SPARSPAK algorithm, and the average values for the AKR, LRT and RND algorithms.

Comparison of Fill

We first compare the lowest fill for the minimum degree and the SPARSPAK orderings and the average fill for the AKR, RND and LRT orderings, and this is shown in Figure 4.10. For each graph and each ordering method, we have

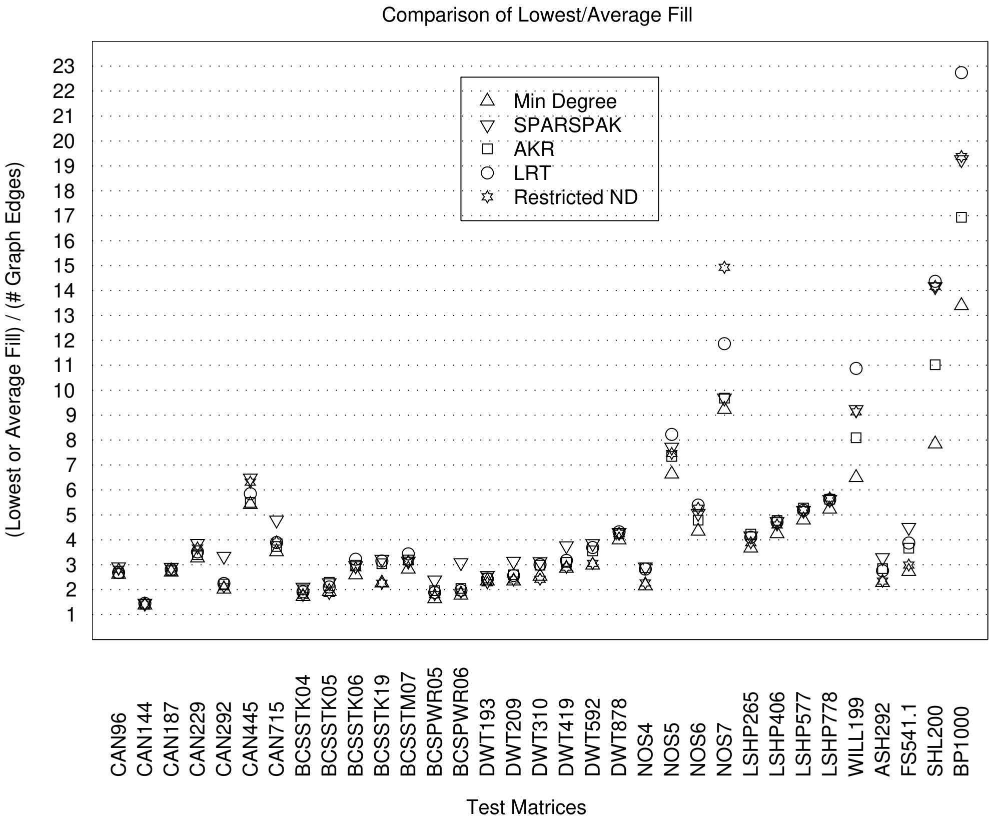


Figure 4.10: Comparison of the lowest fill for the minimum degree and the SPARSPAK algorithms and the average fill for the AKR, LRT and RND algorithms.

plotted the ratio of the lowest or average number of edges in the filled graph for the orderings produced to the number of edges in the input graph. We see that some of the graphs had low fill, while others (eg. NOS7 and BP1000) suffered high fill. In addition, the figure also shows that the minimum degree algorithm was always the best among the 5 methods on the graphs with higher fill. As we shall see next, it was almost always the best on the remaining graphs. These initial results indicate that the minimum degree algorithm is difficult to beat in practice, and lowers our expectation of how well the nested dissection methods will fare in comparison. We note that the minimum degree algorithm has been continuously enhanced since its introduction 30 years ago, and so its superiority on actual test matrices should not be all that surprising.

To better compare the nested dissection methods against the minimum degree algorithm, we plot the results of the previous figure differently in Figure 4.11. For

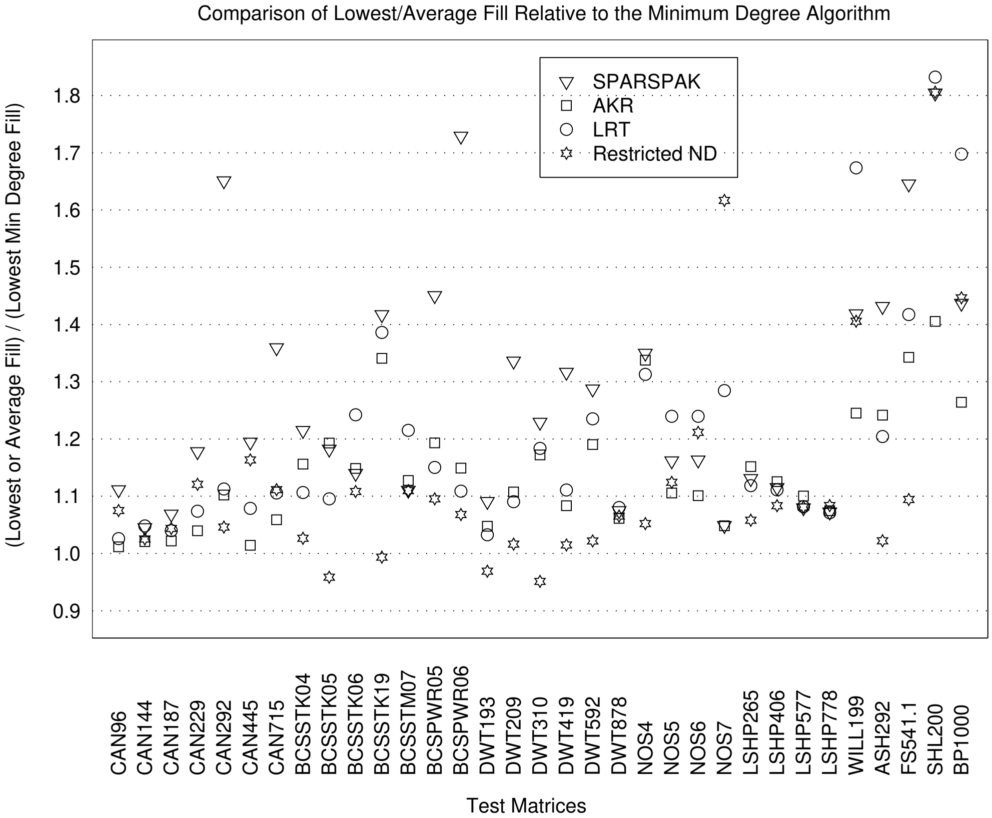


Figure 4.11: Comparison of the lowest fill for the SPARSPAK algorithm and the average fill for the AKR, LRT and RND algorithms, relative to the lowest fill for the minimum degree algorithm.

each of the nested dissection methods, we now show the ratio of its fill to that for the minimum degree algorithm. We observe that there were no graphs on which the AKR or LRT algorithms were better than the minimum degree algorithm, and that there were 4 graphs (BCSSTK05, BCSSTK19, DWT193 and DWT310) where the RND algorithm was slightly better. In those cases, the improvement never exceeded 5%.

In contrast, we see that each of the four nested dissection methods had several instances where its fill was more than 20% higher than that of the minimum degree algorithm. This attests again to the practical superiority of the minimum degree algorithm. We also observe that on the graphs WILL199, SHL200 and BP1000, each of the AKR, LRT and RND methods had fill that was more than 20% higher than that of the minimum degree algorithm, but that on the remaining graphs, at least one of those three methods had excess fill less than 10%. In addition, we also note that those three graphs had high fill relative to the number of edges in the graphs. We shall see in Section 4.7.4 that the top-level separators used by the AKR, LRT and RND algorithms on these three graphs were relatively large, thus contributing significant fill.

As for SPARSAK's variant of nested dissection, we see that it was the worst of the four nested dissection methods on a majority of the graphs, sometimes by a wide margin. Curiously, there were some graphs (eg. BCSSTM07, NOS7 and BP1000) on which SPARSPAK had significantly lower fill than one or more of the AKR, LRT and RND algorithms. We can in fact explain this phenomena.

Recall that the SPARSPAK and the AKR algorithms are both based on regular nested dissection. We therefore directly compare these two algorithms. A closer look shows that the fill for SPARSPAK was usually worse than that for the AKR algorithm, and was slightly better on the graphs BCSSTK05, BCSSTK06, BCSSTM07, LSHP265, LSHP406 and LSHP577. From the experimental results in Section 3.6.3, we know that on those graphs, the multiple runs of the SPARSPAK heuristic and of the Leigthon-Rao constructions produced separators that had virtually the same size. Thus the SPARSPAK algorithm is competitive with the AKR algorithm on those graphs because they both used almost the same separators.[3] The superiority of SPARSPAK over the LRT and RND algorithms on some of those graphs is therefore simply a consequence of the AKR algorithm being better than those two algorithms.

We note also from Section 3.6.3 that in the cases where the sizes of the separators found by SPARSPAK were very different from those of the Leighton-Rao construction, the SPARSPAK fill is much higher than that of the AKR algorithm. These results clearly indicate that the quality of the separators used in nested dissection has a direct bearing on the the fill of the ordering produced.

We now compare the fill of the AKR, LRT and RND orderings. From the same figure, we observe that there were 6 and 14 graphs on which the AKR and RND algorithms respectively were distinctly the best (by a margin of at least 5%) among these 3 algorithms. There were no instances in which the LRT algorithm outperformed the other two by at least 5%. In addition, the relative performance of these algorithms on the 11 graphs (BCSSTK05, BCSSTK19, DWT310, DWT592, NOS4, NOS7, and WILL199 through BP1000) where the worst of the 3 was at least 20% higher than the best is as follows:

- The AKR algorithm was the best on 4 of those graphs, and was the worst or close to the worst on 4 of those graphs.

- The LRT algorithm was the worst or close to the worst on 8 of those graphs,

[3] This remark is true at the top-level of recursion, and is also probably true at lower levels due to the high coincidence of the top-level separators.

and was never the best.

- The RND algorithm was the best on 7 of those graphs, and was the worst on 2 of those graphs.

The preceding comparisons show that the RND algorithm was justifiably the best, while the LRT algorithm was the worst, among the three nested dissection methods discussed in this chapter. Note however, that the fill of the RND orderings was much higher than that of the AKR orderings on the graphs NOS7 and SHL200. We will see in Section 4.7.4 that this is due to the top-level separator of each graph having high deficiency, i.e. the number of missing edges between separator vertices is large relative to the number of edges in the graph.

Comparison of Multiplication Count

We next look at the lowest or average multiplication count for the various orderings. These are given in Figure 4.12, where the ratio of the multiplication count for each of the nested dissection method to that for the minimum degree algorithm is plotted. Note that the multiplication count for SPARSPAK is plotted off the scale for 3 of the graphs. The multiplication count plot in Figure 4.12 looks similar to that for fill in Figure 4.11, albeit on an almost doubly-expanded scale. This is to be expected, since fill and operation counts are closely related quantities of an ordering (cf. Section 1.1.2). We see that there were 6 graphs on which the multiplication count of the RND ordering were better than those of the minimum degree algorithm, but that those improvements were at most 12%. On the other hand, the AKR, LRT and RND algorithms each had several graphs on which the multiplication count was more than 40% higher than that of the minimum degree algorithm.

The relative comparison of the multiplication count for the AKR, LRT and RND orderings is also similar to that for fill, but even more favourable to the RND orderings. There were 4 and 21 graphs on which the AKR and RND algorithms respectively were the best by a margin of at least 10%. The set of 4 graphs for the AKR algorithm is contained in its corresponding set for fill, while the set of 21 graphs for the RND algorithm contains all but one graph in its corresponding set for fill. The set of graphs on which the multiplication count of the worst of the three methods was at least 40% higher than the best also coincides with the corresponding set for fill. Overall, the relative behavior of the three methods on each graph with respect to multiplication count was almost identical to that with respect to fill, except for the graph NOS6, on which the RND method had close to the worst fill but had the best multiplication count.

As for SPARSPAK, its multiplication count was usually the worst of the 5 ordering methods, although it was better than the minimum degree algorithm on the graph NOS7. There were also several instances where SPARSPAK was better than one or more of the AKR, LRT and RND algorithms, and they have a high degree of overlap with the corresponding instances for fill (eg. BCSSTK06,

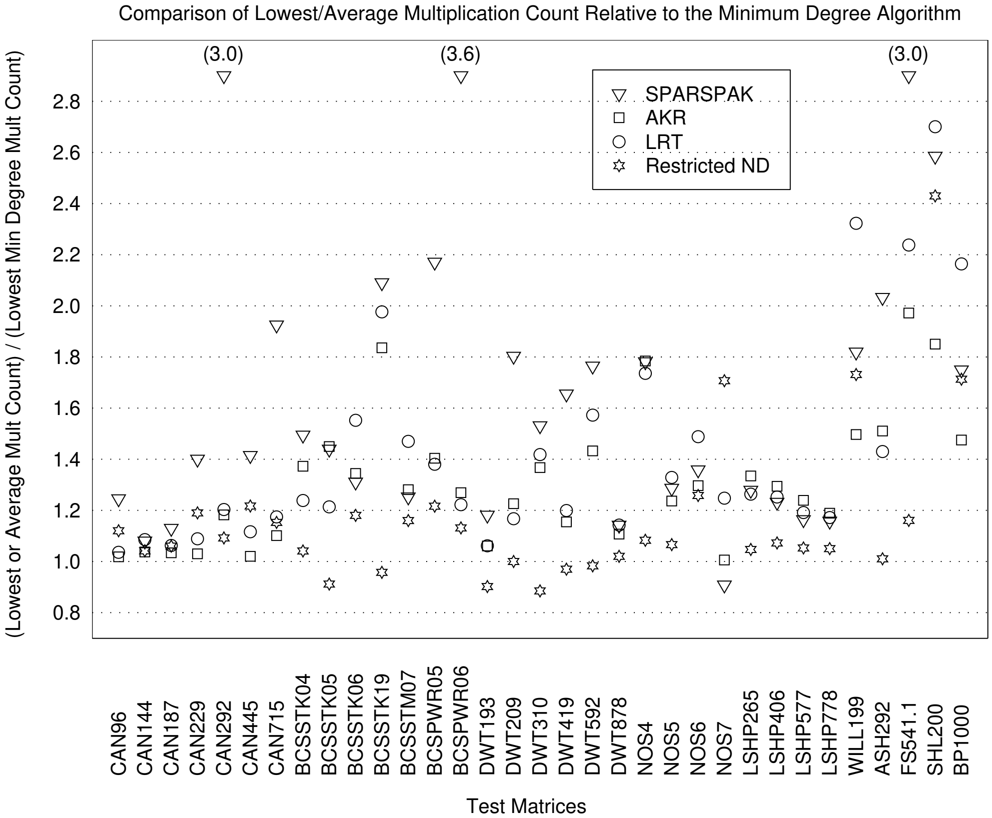


Figure 4.12: Comparison of the lowest multiplication count for the SPARSPAK algorithm and the average multiplication count for the AKR, LRT and RND algorithms, relative to the lowest multiplication count for the minimum degree algorithm.

BCSSTM07, NOS6, NOS7, WILL199 and BP1000). The almost complete overlap again reflects the close relation between fill and multiplication count.

Comparison of Elimination Height

We now turn to the elimination height statistics for the orderings. They are shown in Figure 4.13, where the ratio of the height for each nested dissection method to that for the minimum degree algorithm is plotted. The most striking feature of

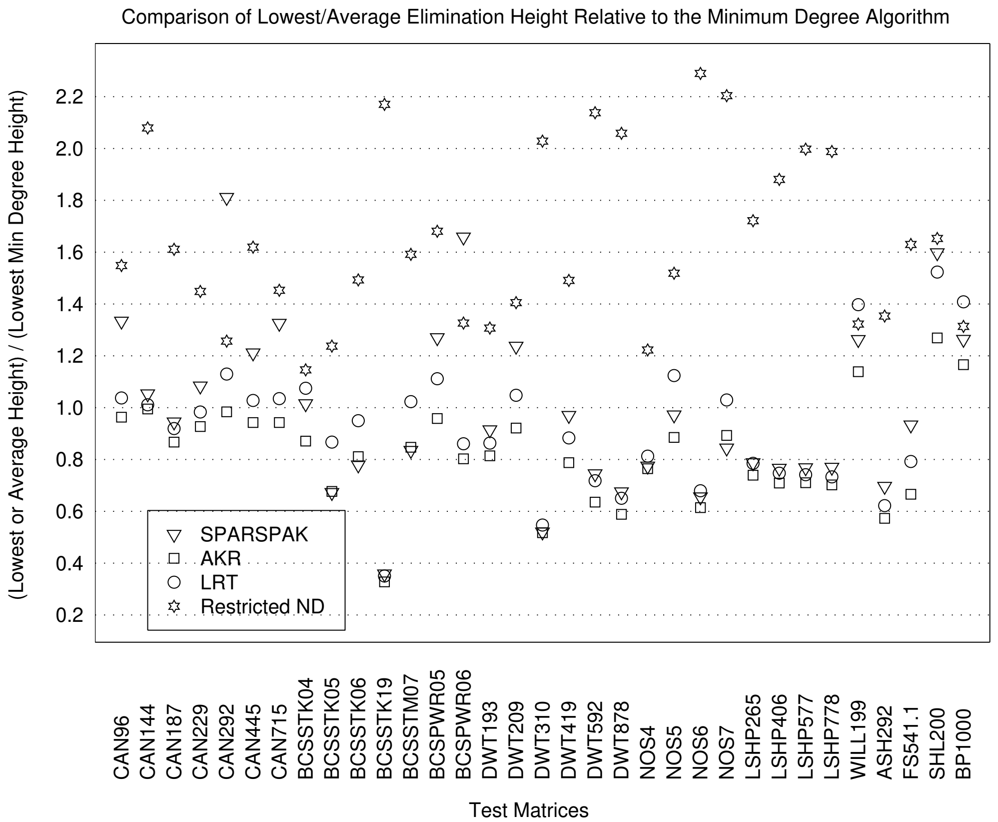


Figure 4.13: Comparison of the lowest elimination height for the SPARSPAK algorithm and the average height for the AKR, LRT and RND algorithms, relative to the lowest elimination height for the minimum degree algorithm.

Figure 4.13 is that the minimum degree orderings do not have good height. For example, the SPARSPAK, AKR and LRT orderings have much lower height on the graph BCSSTK19. By its greedy approach, the minimum degree algorithm directly attempts to minimize fill. Since fill and elimination height are not directly related quantities, the greedy approach can be detrimental to the height on some graphs. For example, as Gilbert [37] pointed out, the minimum degree algorithm generates an ordering that has no fill-in on a graph that is a path, but that ordering has height at least $\frac{1}{2}n$. In contrast a nested dissection ordering has height $O(\log n)$ since each of the separators used consist of only a single vertex.

We look more closely at how each of the nested dissection methods of this chapter fared against the minimum degree algorithm and against each other on the lowest height. We see that the heights of the AKR orderings were always the best of these three methods, with the exception of the graphs BCSSKT06, BCSSTM07 and NOS7 where it was slightly worse than the SPARSPAK heights. In addition, there were only 3 graphs (WILL199, SHL200 and BP1000) on which it was worse than the minimum degree algorithm. These results agree with those given by Agrawal, Klein and Ravi [2] on a set of smaller test matrices, also from the Harwell-Boeing collection [18].

We note that the 3 graphs on which the AKR orderings had height greater than that of the minimum degree algorithm are those where the fill of all the nested dissection methods was much worse than the fill of the minimum degree algorithm. This is not surprising, given that the AKR algorithm approximately minimizes fill, multiplication count and height at the same time.

In sharp contrast, we see that the RND orderings were almost always the worse among the three orderings, and were always worse than the minimum degree orderings. The poor performance of the RND orderings is due to the fact that the separators it uses are larger than those of the LRT and AKR algorithms, as we shall see in Section 4.7.4. Each separator induces a clique in the filled graph, as discussed in Section 4.6.2, and so those used by the RND algorithm introduce long directed paths when the filled graphs are oriented according to the definition of elimination height.

As for the LRT orderings, their heights were within 10% of those of the AKR orderings on 18 of the matrices, and were within 30% on the remaining cases. There were 10 graphs on which the LRT algorithm was worse than the minimum degree algorithm. The comparable behavior of the LRT algorithm to the AKR algorithm is due to its separators being of similar size to those used by the AKR algorithm.

Finally, we see that SPARSPAK's orderings were competitive with the other nested disssection orderings in terms of height. There were only 2 graphs (CAN292 and BCSPWR06) on which the heights of the SPARSPAK orderings were the worst among the nested dissection methods, and 14 graphs on which the heights were no more than 10% higher than those of the AKR orderings. In addition, the heights were lower than those of the AKR orderings on the graphs BCSSTK06, BCSSTM07 and NOS7, and lower than that of the minimum degree algorithm on 20 of the graphs. On those graphs where SPARSPAK is comparable to the AKR algorithm, the separators used by SPARSPAK are again close in size to those used by the AKR algorithm. These results further confirm that "good" balanced separators are have direct impact on the quality of a nested dissection ordering.

4.7.4 Analysis of the Behavior of the Nested Dissection Algorithms

In this section, we shall look at several aspects of the behavior of the AKR, LRT and RND algorithms. The focus will be on trying to understand how these aspects affect the fill and elimination height of the ordering produced. We do not consider multiplication count due to its direct relation to fill.

Intracomponent and Component-Separator Fill

The overall fill that occurs from a nested dissection ordering can be categorized into three classes based on the *top-level* decomposition of the input graph. Specifically, the fill edges are either within the separator, within one of the components, or they are between the separator and a component. We refer to the last class of edges as *cross edges*. The drawback of regular nested dissection, in particular, the AKR algorithm, is that the subproblems do not capture the possibility of cross edges (Section 4.4). Instead, regular nested dissection relies on the small size of the separators used to minimize cross edges. In contrast, the LRT and RND algorithms include the separator vertices as part of the subproblems, thus directly accounting for the cross edges that can occur. The LRT algorithm requires that the included separator vertices be ordered last, while the RND algorithm does not explicitly impose such a restriction. The latter is possible due to the recovery phase which converts the recursively computed ordering into one where the included vertices come last.

It is interesting to see if the intuitive justification for including the separator vertices holds up in practice, and we therefore present statistics on the division of fill between the separators and components. Note that by our experimental setup, as described in Section 4.7.1, the set of top-level separators used by the AKR algorithm to decompose a given input graph was the *same* as those for the LRT and RND algorithms. Since these separators suffer complete fill, the average fill in the top-level separator for all three algorithms were identical.

We first look at the average fill in the top-level separator and also the average total fill in the top-level components for the three algorithms, and this is shown in Figure 4.14. The corresponding statistics for the top-level cross fill will be given later. The average separator/component fill are expressed as a fraction of the lowest fill of the corresponding minimum degree orderings. We choose this scaling as the average overall fill of the AKR, LRT and RND orderings relative to the number of edges in the input graph varied widely across the set of test graphs, as we saw in Figure 4.10, but were always within a factor of 2 of that of the minimum degree orderings.

We see a dichotomy in Figure 4.14 for the average separator fill. In all but 3 of the graphs, the separator fill was low (under 20% of the fill for the minimum degree algorithm), while the (total) component fill was high and typically exceeded 60%. On the other hand, for the graphs WILL199, SHL200 and BP1000, the separator fill exceeded 60% but the component fill was under 30%. This strongly suggests

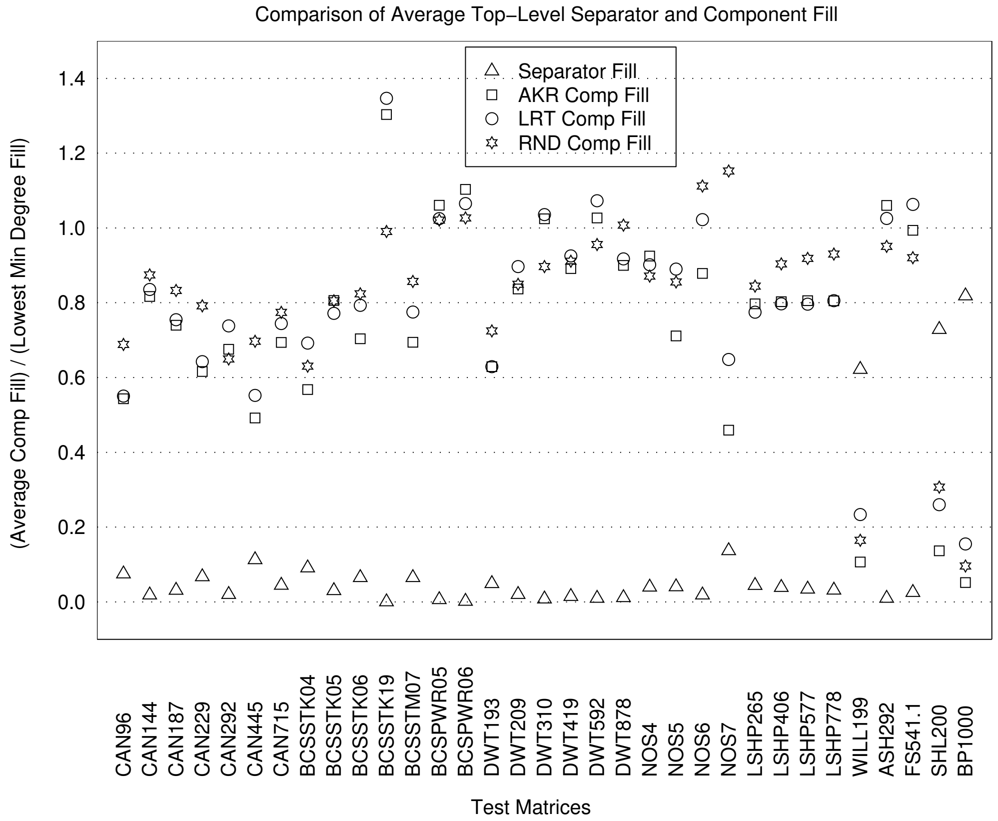


Figure 4.14: Comparison of the average fill in the top-level separator and components for the nested dissection algorithms, relative to the lowest fill for the minimum degree algorithm.

that those three graphs are special. In fact, referring back to Figure 4.10, we see that the highest fill (relative to the number of edges in the input graph) for all *five* ordering methods occur on those 3 graphs and also on the graph NOS7. Note that the separator fill for the graph NOS7 is also somewhat high as compared to the other separators. In addition, Figure 4.11 showed that on those graphs, the fill of the AKR, LRT and RND algorithms were at least 20% higher than that of the minimum degree algorithm. In contrast, the fill of one or more of these algorithms is within 10% of that of the minimum degree algorithm on each of the remaining graphs.

Recall that the separators used in nested dissection suffer complete fill (see Section 4.3) and that by construction, they are $\frac{2}{3}$-balanced and have approximately minimum size. The results therefore indicate that if a sparse graph *does not* have small balanced separators, then any elimination ordering will have high fill. However, the minimum degree algorithm might not create cliques on the large separators of the graph, and so nested dissection orderings are likely to fare poorly against minimum degree orderings.

We now compare the component fill of the AKR, LRT and RND algorithms. The AKR algorithm had the lowest (or close to lowest) component fill on 20 of the graphs and the highest (or close to highest) on 5 of the graphs. The corresponding numbers for the LRT algorithm are 7 and 11, while those for the RND algorithm are 9 and 21. We see therefore that the AKR algorithm was overall the best in minimizing the component fill, while the RND algorithm was the worst. This agrees with expectations, as the AKR subproblems are exclusively on the components, while the RND subproblems include the separator vertices.

We next look at the average top-level cross fill for the three algorithms in Figure 4.15, where the fill is again expressed as a fraction of the lowest fill for the corresponding minimum degree orderings. We have also plotted the average top-level separator fill in the graph. We see that the dichotomy on the separator fill is reflected in the cross fill. On the graphs WILL199, SHL200 and BP1000, where the separator fill was relatively high, the cross fill was also high and typically exceeded 50%. In contrast, on the remaining graphs, the cross fill was almost always under 40%.

The figure also shows that the cross fill for the RND algorithm was almost always the lowest, except on the pathological graphs WILL199, SHL200 and BP1000. In addition, the cross fill for the AKR and LRT algorithms were virtually identical on most of the graphs. The notable exceptions were the graphs BCSSTK04 and DWT209, on which the LRT algorithm was much better, and the pathological graphs, on which the AKR algorithm was clearly superior to the LRT algorithm, which in turn was much better than the RND algorithm. These observations (disregarding the pathological cases) support the claim that including separator vertices in the subproblems, as is done in the RND and LRT algorithms, can lead to orderings with lower cross fill. The clear success of the RND algorithm in achieving this goal is due to its conversion of each subproblem into an equivalent one without any restrictions on the ordering of the included separator vertices. In contrast, the LRT algorithm produces ordering constrained subproblems, and so it is not as

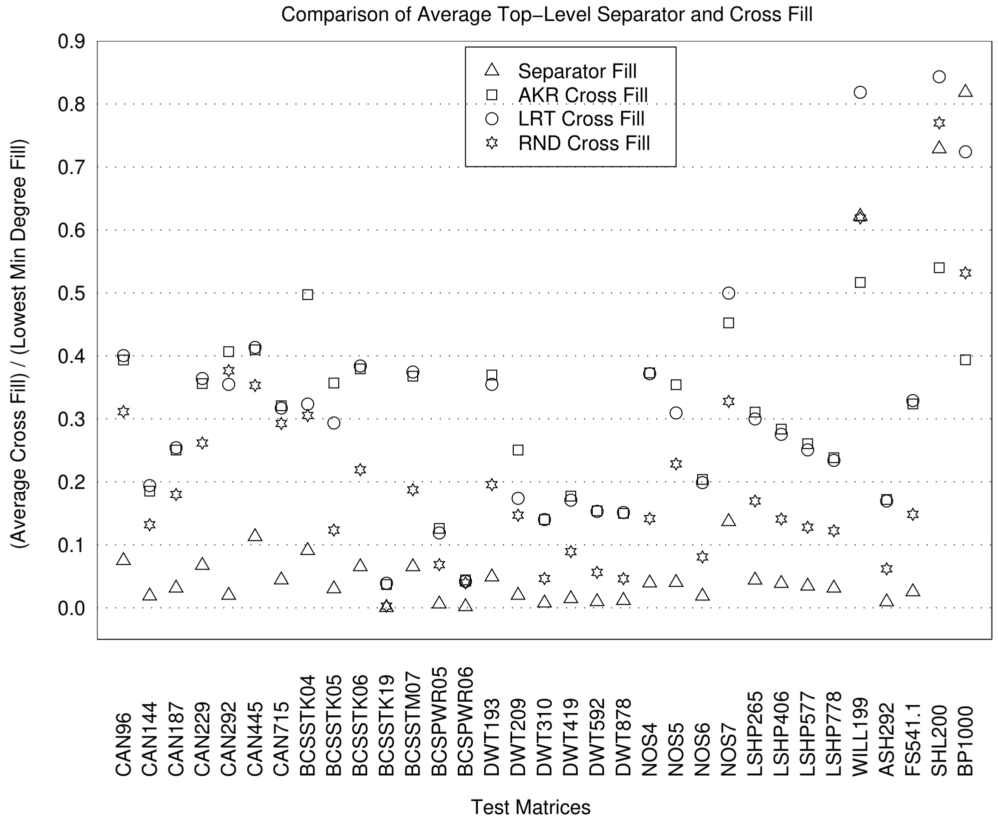


Figure 4.15: Comparison of the average fill between the top-level separator and components for the nested dissection algorithms, relative to the lowest fill for the minimum degree algorithm.

successful as the RND algorithm in reducing cross fill.

Separator Statistics

We next present some statistics on the sizes of the separators used in nested dissection. For each ordering method and each input graph, we compute the average sizes of the separators used at the top and 2nd level of recursion. Note that the average size of the top-level separators used for a given input graph is the *same* for the AKR, LRT and RND algorithms, by the setup for our experiments. Figure 4.16 shows these top and 2nd-level average sizes, where the size of a separator is expressed as a fraction of the *top-level* input graph, for ease of comparison.

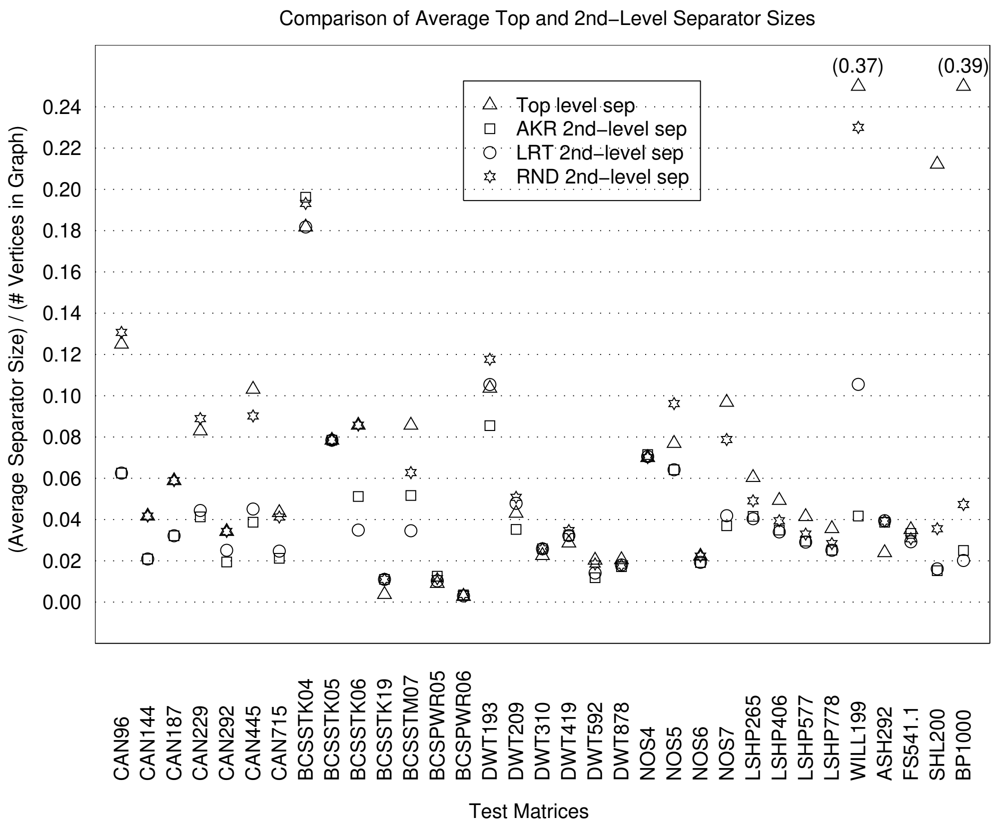


Figure 4.16: Comparison of the average sizes of the separators at the top and 2nd level of recursion for the nested dissection methods. The size of a separator is expressed as a fraction of the vertex size of the top-level input graph.

We see that the average top-level separator usually contained less than 10% of the vertices in the input graph. However, on the pathological graphs WILL199, SHL200 and BP1000, the top-level separators were considerably larger, explaining the high fill that occurred on these separators. In fact, the absolute deficiency, expressed as a fraction of the number of edges in a graph, is 1.2 for NOS7, 3.9

for WILL199, 5.6 for SHL200, and 10.7 for BP1000, and is at most 0.58 for the remaining graphs.

In addition, the ratios of the average size of the separators to the maximum size of a component arising from those separators were 1.08, 0.50 and 1.23 for WILL199, SHL200 and BP1000 respectively, while the corresponding ratios for the other test graphs ranged from 0.01 to 0.44. The large ratios are bad for the RND algorithm since the 2nd-level subproblems will have relatively large cliques, and explain why the RND algorithm also fares poorly against the AKR algorithm on those three graphs.

Among the 2nd-level separators, those used by the RND algorithm were almost always the largest, whereas those used by the AKR and the LRT algorithms were usually of very similar sizes. This comes as no surprise, since each 2nd level subproblem of the RND algorithm is larger than the corresponding one in the AKR algorithm because of the inclusion of the top-level separator. Furthermore, there is a clique on those included vertices, thereby requiring a separator with a greater number of vertices to decompose that subproblem. In contrast, although each LRT subproblem has the same size as its RND counterpart, there are no edges between the included vertices, and so these additional vertices impose a less severe constraint on the separator used for that subproblem. To support this conclusion, we computed the following statistic. For each 2nd level separator S_2 of the LRT or RND algorithm, we count the number of vertices in it that are also in the top-level separator S_1, and then divide that count by the size of S_2, i.e. we compute the fraction $\#(S_2 \cup S_1)/|S_2|$. We then computed the average of these fractions for each of the LRT and RND algorithms, and for each input graph. It turned out that the averages for the LRT algorithm ranged between 0 and 0.17 across the test graphs, while the averages for the RND algorithm ranged between 0 to 0.79. Thus a much smaller proportion of the 2nd-level separators in the LRT algorithm come from the top-level separator, corroborating the conclusion that the included vertices have less impact on the LRT algorithm as compared to the RND algorithm.

Other Recursion Statistics

Figure 4.17 plots the average number of recursion levels for each method against the logarithmic size of the input graphs. Note that the logarithms are taken to the base 1.5, since we used separators that are $\frac{2}{3}$-balanced. From the figure, we see that the average number of recursion levels for both the AKR and LRT algorithms were approximately linear in the logarithm of the number of vertices in the input graph. A linear regression fit gives the coefficients of 0.66 and 0.78 respectively for the AKR and LRT algorithms. This shows that for the AKR algorithm, the average size of a component when a subproblem is decomposed is much less than $\frac{2}{3}$ of the size of the subproblem, as guaranteed by the definition of a $\frac{2}{3}$-balanced separator. For the LRT algorithm, the subproblems do not shrink as quickly in size because of the inclusion of separator vertices from the previous level.

We see from the figure that the average number of recursion levels for the RND algorithm was also approximately linear in the logarithmic vertex size of

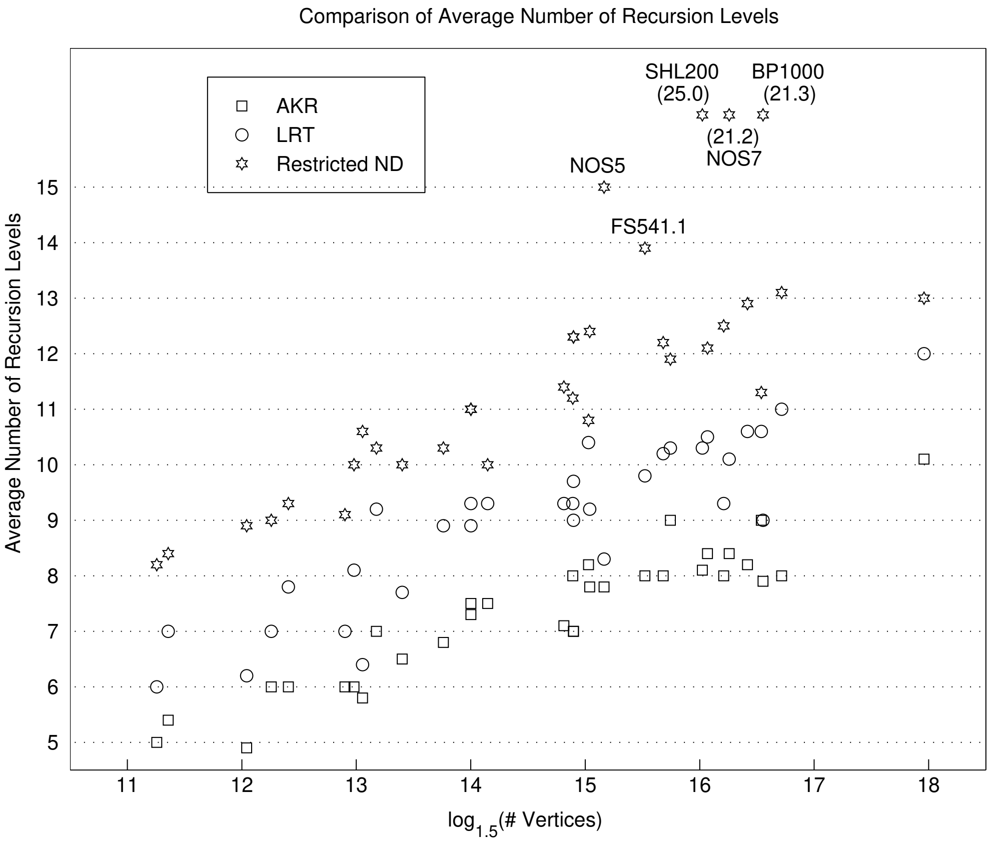


Figure 4.17: Comparison of the average number of recursion levels for the nested dissection methods.

the graphs. However, there were three graphs (NOS7, SHL200 and BP1000) on which the number of levels was disproportionately high, and these graphs have been plotted off the scale, with the number of levels given in parentheses. Two other instances (NOS5 and FS541.1) with slightly higher than expected number of recursion levels are also marked in the plot. Referring back to Figure 4.11, we see that the fill for the RND algorithm on graphs NOS7, SHL200 and BP1000 was much worse than that of the AKR algorithm. However, the other pathological graph WILL199 from earlier discussions did not have an exceedingly high number of recursion levels. This shows that the occurrence of too many recursion levels lead to poor performance for the RND algorithm, but that it was not a necessary condition for such performance.

In addition, the linear regression coefficient for the RND algorithm, excluding the graphs NOS7, SHL200 and BP1000 was 1.03. The subproblems in the RND algorithm did not shrink as quickly as those in the LRT algorithm because the separators used by the RND algorithm are larger than those used by the LRT algorithm, from the 2nd level onwards.

Summary of Results

We have seen that the minimum degree algorithm is surprisingly difficult to beat in practice with respect to fill and multiplication count. There were some instances though on which the RND algorithm had lower fill and multiplication count than the minimum degree algorithm. Overall, the RND algorithm was the best among the three nested dissection methods discussed in this chapter, while the LRT algorithm was the worst. The instances on which the RND algorithm performed poorly relative to the AKR algorithm were due to the use of large separators with high deficiency.

The importance of the quality of the separator used was further reflected in the performance of SPARSPAK's variant of regular nested dissection. When the separator used by SPARSPAK was almost identical to those found by the Leighton-Rao balanced separator construction, the quality of the SPARSPAK orderings were very similar to those of the AKR algorithm. However, when SPARSPAK used separators that were not as good, the quality of its orderings dropped significantly.

Finally, the nested dissection methods, with the exception of the RND algorithm, were much better than the minimum degree algorithm in terms of the elimination height of the ordering produced. This shows that minimum degree orderings have poor height. The inability of the RND algorithm to outperform the minimum degree algorithm is due to the larger sizes of the separators that it used.

4.7.5 Results for Nested Dissection with Low-Deficiency Separators

We saw in the previous section that the nested dissection methods performed poorly when the separators used were large and had high deficiencies. We therefore investigated the use of low-deficiency balanced separators in place of the Leighton-Rao balanced separators. These low-deficiency balanced separators were found by simulated annealing as described in Section 3.7. We did not modify SPARSPAK's version of nested dissection to use low-deficiency separators, since the earlier results indicated that SPARSPAK's version was the worst among the four nested dissection methods, and resembled the Agrawal-Klein-Ravi algorithm on those instances where it performed well. In addition, we only present results for fill, given the close relation between multiplication count and fill, and the fact that the AKR and LRT algorithms were superior to the minimum degree algorithm with respect to elimination height.

Figure 4.18 gives the average fill of the AKR, LRT and RND algorithms using low-deficiency balanced separators. As before, we performed 10 runs for each method. The plots are again with respect to the lowest fill of the minimum degree orderings.

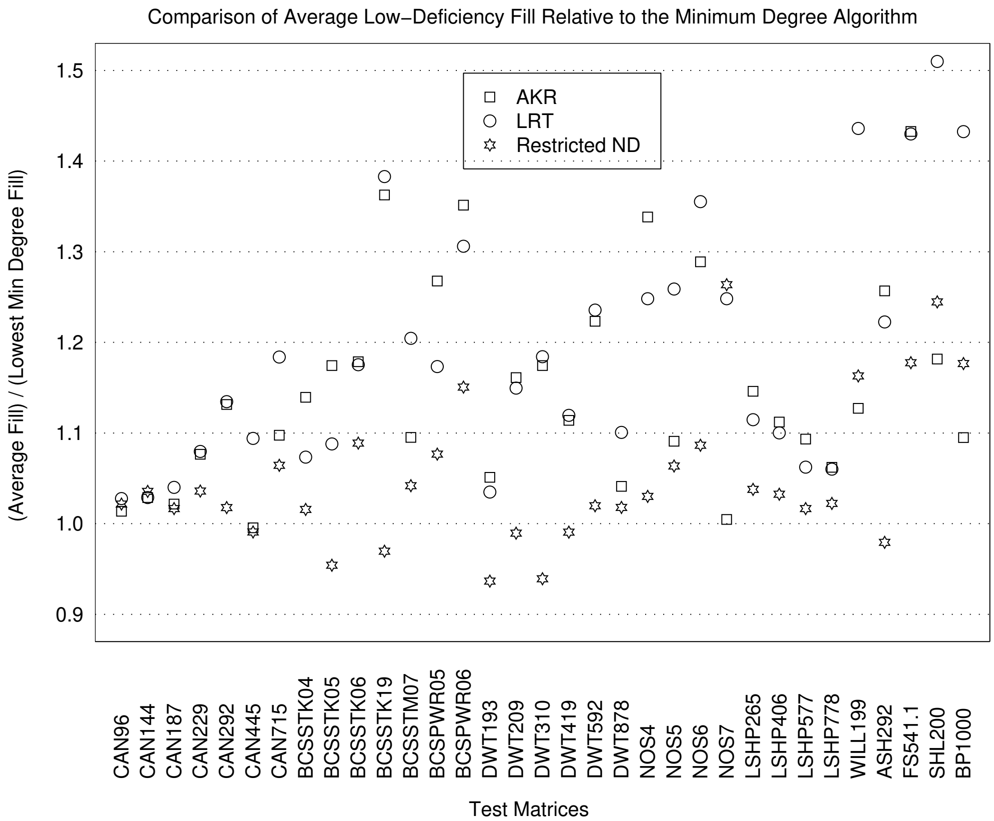


Figure 4.18: Comparison of the average fill for the AKR, LRT and RND algorithms using low-deficiency separators, relative to the lowest fill for the minimum degree algorithm.

In comparing Figure 4.18 against the corresponding Figure 4.11, we see that the poor behavior of one or more of the nested dissection methods on the graphs NOS7, WILL199, SHL200 and BP1000 have been moderated. The average sizes of the low-deficiency separators for these graphs were $0.084n$, $0.29n$, $0.14n$ and $0.27n$ respectively. In contrast, the average sizes of the Leighton-Rao separators were significantly higher, at $0.097n$, $0.37n$, $0.21n$ and $0.39n$. More strikingly, the average deficiencies of the low-deficiency separators were $0.94m$, $2.4m$, $2.5m$ and $5.1m$ respectively, while those for the Leighton-Rao separators were drastically higher, with values of $1.3m$, $3.9m$, $5.6m$ and $10.7m$. These statistics show that the low-deficiency separators are a better choice on those graphs, although the (still) high sizes and deficiencies of these separators indicate that nested dissection is perhaps not the appropriate ordering method for those graphs.

It is interesting to look at the efficacy of low-deficiency separators for the AKR, LRT and RND algorithms.

- For the AKR algorithm, the fill of the low-deficiency-based ordering was higher than that of the Leighton-Rao-based ordering on 18 of the 33 graphs. There were 3 graphs where the low-deficiency fill exceeded 1.05 times the Leighton-Rao fill, and 3 graphs (WILL199, SHL200 and BP1000) where it was under 0.95 times the Leighton-Rao fill.

- For the LRT algorithm, the low-deficiency fill was higher than the Leighton-Rao fill on 17 graphs. There were 5 graphs where the low-deficiency fill exceeded 1.05 times the Leighton-Rao fill, and 3 graphs (WILL199, SHL200 and BP1000) where it was under 0.95 times the Leighton-Rao fill.

- For the RND algorithm, the low deficiency fill was higher than the Leighton-Rao fill on only 3 graphs. There were 2 graphs where the low-deficiency fill exceeded 1.05 times the Leighton-Rao fill, but 11 graphs (including WILL199, SHL200 and BP1000) where it was under 0.95 times the Leighton-Rao fill.

The above results clearly show that low deficiency separators are well-suited to restricted nested dissection, but are somewhat indifferent to the AKR and LRT algorithms. This conclusion is also seen in Figure 4.18, in which there were 8 graphs where the RND algorithm with low-deficiency separators had lower fill than the minimum degree algorithm, whereas there were only 4 such graphs when Leighton-Rao balanced separators were used.

Finally, we also observed that the plot of the number of recursion levels for the RND algorithm with low-deficiency separators against the logarithmic size of the graph did not exhibit the anomalies that were present in Figure 4.17. This further corroborates the usefulness of such separators for restricted nested dissection.

Chapter 5

Conclusion

We conclude this thesis by summarizing the major results and identifying open problems raised by this research.

5.1 Summary of Results

Elimination Orderings

We compared in Section 4.7 the three nested dissection methods discussed in Chapter 4: the Agrawal-Klein-Ravi algorithm, the Lipton-Rose-Tarjan generalized nested dissection algorithm and restricted nested dissection. Overall, the restricted nested dissection algorithm was the best among the three methods, while the Lipton-Rose-Tarjan algorithm was the worst. The instances on which the restricted nested dissection algorithm performed poorly relative to the Agrawal-Klein-Ravi algorithm were due to the use of large separators with high deficiency. Those instances improved greatly when balanced low-deficiency separators were used in place of approximately optimal balanced separators.

The importance of the quality of the separator used was further reflected in the performance of SPARSPAK's heuristic variant of regular nested dissection. When the separators used by SPARSPAK were almost identical to those found by the Leighton-Rao balanced separator construction, the quality of the SPARSPAK orderings were very similar to those of the Agrawal-Klein-Ravi algorithm. However, when SPARSPAK used separators that were not as good, the quality of its orderings dropped significantly.

Finally, the results also highlight the unreasonable effectiveness of the minimum degree algorithm in achieving low fill and multiplication count in practice. The algorithm has yet to be theoretically justified since its introduction many years ago, and any guarantees on the quality of minimum degree orderings on general or even special classes of graphs would be valuable.

Cut Constructions

Our experimental evaluation in Section 3.4 of the Leighton-Rao sparse cut construction showed that its full generality was not needed. All the cuts found by the general construction of Section 3.2 turned out to be composed only of a single region. Thus, in practice, the simplified construction described in Section 3.3 enjoys the $O(\log n)$ performance guarantee that the general construction possesses.

In addition, we saw that the quality of the dual length functions used in the construction did not significantly influence the quality of the cuts found. The graphs in our test suite also did not exhibit the $O(\log n)$ max-flow min-cut gap. Finally we also observed that the sparse cuts that were found were almost always $\frac{2}{3}$-balanced.

Solution of Concurrent Flows

Our results in Section 2.5.1 showed that solving a concurrent flow problem using the concurrent flow approximation algorithm was orders-of-magnitude faster than using linear programming methods.

With respect to the implementation of the concurrent flow approximation algorithm, the detailed tuning experiments in Section 2.4 demonstrated that the running time of the implementation can be substantially reduced. In particular, checking for the optimality of the computed solution has significant performance implications that have to be considered when designing a practical optimality checking scheme.

5.2 Open Problems

While the experimental results in Section 4.7 indicated that the restricted nested dissection algorithm was usually superior to the Agrawal-Klein-Ravi algorithm, we were unable to provide corresponding performance guarantees on the quality of restricted nested dissection orderings. Proving such guarantees appears to be a difficult problem as it involves relating the minimum chordal extension of a graph G to that of G with a clique introduced. The latter problem has not been previously studied.

The experimental results also demonstrated the usefulness of balanced low-deficiency separators. There are no known constructions for such separators that are provably approximately optimal. A theoretical result in this regard can perhaps be applied to obtain better performance guarantees for the Agrawal-Klein-Ravi algorithm or for restricted nested dissection. It is also interesting to consider the use of other heuristics for finding balanced low-deficiency separators.

Bibliography

[1] A. Agrawal. *Network Design and Network Cut Dualities: Approximation Algorithms and Applications*. PhD thesis, Brown University, 1991. Technical Report CS-91-60.

[2] A. Agrawal, P. Klein, and R. Ravi. Cutting down on fill using nested dissection: Provably good elimination orderings. In J. A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, volume 56 of *The IMA Volumes in Mathematics and Its Applications*, pages 31–55. Springer-Verlag, New York, 1993.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[4] M. Ajtai. Recursive construction for 3-regular expanders. *Combinatorica*, 14(4):379–416, 1994.

[5] N. Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.

[6] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. In *25th Annual ACM Symposium on the Theory of Computing*, 1993.

[7] F. Bien. Constructions of telephone networks by group representations. *Notices of the American Mathematical Society*, 36:5–22, 1989.

[8] J. R. S. Blair and B. Peyton. An introduction to chordal graphs and clique trees. In A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 1–29. Springer-Verlag, 1993.

[9] M. Blum, R. M. Karp, O. Vornberger, C. H. Papadimitriou, and M. Yannakakis. The complexity of testing whether a graph is a superconcentrator. *Information Processing Letters*, 13(4–5):164–167, 1981.

[10] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18:238–255, 1995.

[11] J. M. Borger, T. S. Kang, and P. N. Klein. Approximating concurrent flow with unit demands and capacities: An implementation. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matchings: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 371–386, Providence, 1993. DIMACS, American Mathematical Society.

[12] V. Cerny. A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[13] F. R. K. Chung. Constructing random-like graphs. In B. Bollobás, editor, *Probabilistic Combinatorics and Its Applications*, volume 44 of *Proceedings of Symposia in Applied Mathematics*, pages 21–55. American Mathematical Society, 1991.

[14] N. E. Collins, R. W. Eglese, and B. L. Golden. Simulated annealing: An annotated bibliography. *American Journal of the Mathematics of Management Science*, 8:205–307, 1988.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Massachusetts, 1994.

[16] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report CSD-95-883, University of California, Berkeley, September 1995.

[17] R. B. Dial. Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM*, 12(11):632–633, 1969.

[18] I. Duff, R. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15:1–14, 1989.

[19] I. Duff, R. Grimes, and J. G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection (release I). Technical Report TR/PA/92/86, Research and Technology Division, Boeing Computer Services, Seattle, Washington, October 1992. The user's guide and the matrix collection are available via `http://math.nist.gov/MatrixMarket/collections/hb.html`.

[20] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Monographs on Numerical Analysis. Clarendon Press, Oxford, 1986.

[21] I. S. Duff and J. K. Reid. MA27: A set of FORTRAN subroutines for solving sparse symmetric sets of linear equations. Technical Report AERE R 10533, Harwell, U. K., 1982.

[22] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. The Yale sparse matrix package I: The symmetric codes. *International Journal of Numerical Methods in Engineering*, 18:1145–1151, 1982.

[23] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[24] A. P. A. A. for the Minimum Fill-in Problem. Assaf natanzon and ron shamir and roded sharan. To appear in STOCS '98.

[25] L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[26] M. L. Fredman and R. E. Tarjan. Fibonnaci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[27] D. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15:835–855, 1965.

[28] O. Gabber and Z. Galil. Explicit constructions of linear-sized superconcentrators. *Journal of Computer and System Sciences*, 22:407–420, 1981.

[29] S. Geman and D. Geman. Stochastic relaxation, gibbs distribution, and the Bayesian restoration of images. In *IEEE Proceedings on Pattern Analysis and Machine Intelligence*, volume PAMI-6, pages 721–741, 1984.

[30] J. A. George. Block elimination on finite element systems of equations. In D. J. Rose and R. A. Willoughby, editors, *Sparse Matrices and Their Applications*, pages 101–114. IBM, Plenum Press, 1972.

[31] J. A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, April 1973.

[32] J. A. George and J. W. H. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal on Numerical Analysis*, 15(5):1053–1069, October 1978.

[33] J. A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Series in Computational Mathematics. Prentice-Hall, Englewood Cliffs, 1981.

[34] J. A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, March 1989.

[35] J. A. George, J. W. H. Liu, and E. G. Ng. User's guide for SPARSPAK: Waterloo sparse linear equations package. Technical Report CS-78-30 (revised), Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1980.

[36] N. E. Gibbs, W. G. Poole Jr., and P. K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, April 1976.

[37] J. R. Gilbert. Some nested dissection is nearly optimal. *Information Processing Letters*, 26:325–328, 1988.

[38] J. R. Gilbert, D. J. Rose, and A. Edenbrandt. A separator theorem for chordal graphs. *SIAM Journal on Algebraic and Discrete Methods*, 5:306–313, 1984.

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

[40] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

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

[42] M. D. Hansen. Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems. In *30th Annual Symposium on Foundations of Computer Science*, pages 604–610, 1989.

[43] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–460, September 1991.

[44] A. J. Hoffman, M. S. Martin, and D. J. Rose. Complexity bounds for regular finite difference and finite element grids. *SIAM Journal on Numerical Analysis*, 10(2):364–369, April 1973.

[45] T. C. Hu. Multi-commmodity network flows. *Operations Research*, 11(3):344–360, 1963.

[46] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, 1989.

[47] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[48] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, Ltd., 1997.

[49] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[50] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5/6):975–986, 1984.

[51] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[52] P. Klein, S. Plotkin, C. Stein, and Éva Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparsest cuts. *SIAM Journal on Computing*, 23(3):466–487, June 1994.

[53] P. Klein, S. Rao, A. Agrawal, and R. Ravi. An approximate max-flow min-cut relation for undirected multicommodity flow, with applications. *Combinatorica*, 15(2):187–202, 1995.

[54] D. E. Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 1973.

[55] K. Lang and S. Rao. Finding near-optimal cuts: An empirical evaluation. In *4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 212–221, 1993.

[56] T. Leighton, F. Makedon, S. Plotkin, C. Stein, Éva Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences*, 50(2):228–243, April 1995.

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

[58] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. Unpublished manuscript, September 1993.

[59] T. Leong, P. Shor, and C. Stein. Implementation of a combinatorial multicommodity flow algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matchings: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 387–405, Providence, 1993. DIMACS, American Mathematical Society.

[60] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.

[61] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, April 1979.

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

[63] J. W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, June 1985.

[64] J. W. H. Liu. The minimum degree ordering with constraints. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1136–1145, November 1989.

[65] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, January 1990.

[66] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, March 1992.

[67] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[68] H. Okamura and P. D. Seymour. Multicommodity flows in planar graphs. *Journal of Combinatorial Theory, Series B*, 31:75–81, 1981.

[69] R. H. J. M. Otten and L. P. P. P. van Ginneken. *The Annealing Algorithm.* Kluwer Academic Publishers, Boston, 1989.

[70] S. Parter. The use of linear graphs in Gaussian elimination. *SIAM Review*, 3:119–130, 1961.

[71] N. Pippenger. Superconcentrators. *SIAM Journal on Computing*, 6(2):298–304, June 1977.

[72] T. Radzik. Fast deterministic approximation for the multicommodity flow problem. In *6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 486–492, 1995.

[73] R. Ravi, A. Agrawal, and P. Klein. Ordering problems approximated: Single-processor scheduling and interval graph completion. In *18th International Conference on Automata, Languages, and Programming*, volume 510 of *Lecture Notes in Computer Science*, New York, 1991. Springer-Verlag.

[74] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

[75] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, 34:176–197, 1978.

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

[77] D. J. Rose and R. A. Willoughby, editors. *Sparse Matrices and Their Applications.* The IBM Research Symposia Series. Plenum Press, 1972.

[78] G. H. Sasaki and B. Hajek. The time complexity of maximum matching by simulated annealing. *Journal of the ACM*, 35:387–403, 1988.

[79] F. Shahrokhi. *Design and Analysis of Efficient Algorithms to Solve the Maximum Concurrent Flow Problem.* PhD thesis, Western Michigan University, 1986.

[80] F. Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37(2):318–334, April 1990.

[81] D. B. Shmoys. Cut problems and their application to divide-and-conquer. In D. S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, Boston, MA, 1997.

[82] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[83] R. M. Tanner. Explicit concentrators from generalized n-gons. *SIAM Journal on Algebraic and Discrete Methods*, 5(3):287–293, September 1984.

[84] R. E. Tarjan. Maximum cardinality search and chordal graphs. Unpublished Lecture Notes CS259, 1976.

[85] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13:566–579, 1984.

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

[87] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications.* Kluwer Academic Publishers, Boston, 1987.

[88] D. F. Wong, H. W. Leong, and C. L. Liu. *Simulated Annealing for VLSI Design.* Kluwer Academic Publishers, Boston, 1988.

[89] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, March 1981.

[90] Zienkiewicz and Taylor. *The Finite Element Method.* McGraw-Hill, Boston, 1989.