Network Design and Network Cut Dualities: Approximation Algorithms and Applications

Ajit Kumar Agrawal
Ph.D. Dissertation

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-60
September 1991

Network Design and Network Cut Dualities: Approximation Algorithms and Applications

by

Ajit Kumar Agrawal

B. Tech., Electronics and Electrical Communications Engg.
Indian Institute of Technology, Kharagpur, India, May 1984

M. S., Electrical Engineering
Yale University, May 1985

M. S., Computer Science
Brown University, May 1990

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May 1992

Vita

I joined the Ph.D. program in Computer Science at Brown University in 1989. I received an M.S. degree in Electrical Engineering in 1985, and an M. Phil. degree in Computer Science in 1986, both from Yale University. I worked at Thinking Machines Corporation for two years, and spent another year at Yale University before coming to Brown University.

I was born in Aurangabad, and attended St. Xavier's High School in Patna, India. I received my B. Tech. degree in Electronics and Electrical Engineering from Indian Institute of Technology, Kharagpur, India.

Abstract

We give approximation algorithms for two network design problems. These problems arise in designing communication networks where a prespecified set of sites need to be connected through a network. Different applications judge the merit of the networks using different cost criteria. We consider two such cost criteria. Under the first criterion, a network of minimum total edge-cost is to be designed. This problem is known as the *Steiner tree problem*. Under the second criterion, the network's maximum degree is to be minimized, and is referred to as the *min-degree problem*. Both these problems are known to be NP-complete.

Our algorithms output a network in polynomial time whose cost is within a small multiplicative factor of the minimum network cost. Our algorithm for the Steiner tree problem extends to a more general setting of the connectivity requirements of the network, known as the *generalized Steiner Problem*. No approximation algorithms were previously known for either the generalized Steiner problem or the min-degree problem.

Underlying the proofs of performance for the above algorithms are approximate min-max equality relations, which we establish, between the network design problems and some combinatorial problems in finding network cuts. In turn, we also provide approximation algorithms for these combinatorial problems related to network cuts, and describe their applications.

We also apply a previously known approximation algorithm for yet another combinatorial quantity related to network cuts to a problem that arises in the solution of sparse linear systems. We present the first polynomial-time algorithm that finds an elimination ordering that guarantees approximately optimal fill in solving certain symmetric sparse linear systems. Our algorithm is a variant of the well-known nested dissection algorithm and uses a result of Leighton and Rao for finding a balanced node separator in a graph. We also give some experimental results showing the quality of the elimination orderings produced by our algorithm.

Acknowledgements

My advisor, Philip Klein. Who has been generous with his time, and ideas, and unbounded in his enthusiasm, and encouragement. Who gave me the privilege of being his first advisee. Lennart Johnsson. For introducing me to Computer Science. For giving me the opportunity to work at Thinking Machines Corporation where I learned to appreciate the broader aspects of Computer Science and parallel machines in particular. R. Ravi. For staring at the whiteboard with me for hours. And for his friendship. Without his help this thesis would not have been possible. Paris Kanellakis, and Jeff Vitter. For agreeing to be on my thesis committee and taking the time to read this thesis and offer suggestions. John Gilbert. For his encouragement and help. Tom Leighton and Satish Rao. For their work on graph separators, which served as a guiding light for all of the results in this thesis.

Mary Andrade. Whose cheerful assistance in contacting Brown faculty I cannot forget. John Savage, and Jeff Vitter. For encouraging me to come to Brown. Sandeep Bhatt, Lennart Johnsson, and Guy Steele, Jr. For recommending me to Brown. My officemate Yi-Jing Lin. For tolerating the frequent spate of loud brain-storming sessions in the office.

Prof. Swapan Majumdar. For being the source of inspiration for me and for many other IITians from Kharagpur. For the indelible impression on my life.

My family. Papaji. Who took the greatest pride in the achievements of his children and settled for nothing less than the best education for them. Amma. For her love. For taking the heat for all my mischiefs. For the pickles that can still brighten up my day. Rajesh, Sangeeta, Anil, and Sunil. For their academic achievements which set the ground for my desire to achieve the same. My brother Sunil. For tolerating me in the face of our daily childhood fights, and selflessly sharing with me all he learned. My wife Uma. For being there for me, always. For her gaiety and love for partying, that easily diluted the stress from the demands of a Ph.D. degree.

I dedicate this thesis to my parents, Mrs. Premlata Agrawal and Dr. Raghava P. Agrawal, to whom I can give no greater a gift.

Credits

The results reported in Chapters 3 and 4 regarding the network design problems were obtained jointly with my advisor Philip Klein, and my colleague R. Ravi [1, 2]. The theoretical results reported in Chapter 5 regarding elimination orderings were obtained jointly with Philip Klein, and R. Ravi [90]. The experiments reported in Chapter 5 were conducted jointly with John Gilbert, Philip Klein, and Sarah Kang.

I would like to thank John Gilbert, Sarah Kang, Philip Klein, and R. Ravi for allowing me to include our joint work in this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 NP-completeness

The role of an algorithm designer is to find efficient ways of solving problems. One of the strongest performance measures of an algorithm is the number of steps it takes to produce the solution as a function of the input size. For example, if we assume that each arithmetic operation is a single-step operation, then the number of steps required to add n numbers is $n - 1$. As another example, the simple method we learnt in high school for multiplying two $n \times n$ matrices requires $2n^3 - n^2$ arithmetic steps.[1] The number of steps is a machine-independent measure of performance. It can be translated in a straightforward manner to the time it takes to execute the algorithm on a given machine, and hence is also a measure of the time complexity of the algorithm.

The time complexity indicates how the number of steps required to solve the problem scales as the problem size grows. Obviously, the lower the time complexity of the algorithm, the larger a problem we can hope to solve in a reasonable amount of time. For both the problems mentioned above, the time complexity can be expressed as a low order polynomial of the input size, which is fortunate, since there is a large body of important problems for which *no* polynomial-time algorithms are known. By a polynomial-time algorithm, we mean an algorithm whose time complexity can be expressed as a fixed degree (independent of the problem size) polynomial in terms of the input size.[2] What is worse, is that it is unlikely that polynomial-time algorithms exist for these problems.[3]

In fact there is a well-studied class hierarchy of problems divided by their time (or space) complexity. The reader is referred to the books by Lewis and Papadimitriou [100], and Hopcroft and Ullman [81] for study of this complexity theory. The classes relevant for this thesis are the classes known as P, and NP-complete. Problems in the class P are known to have polynomial-time solutions, while the problems in the class NP-complete are not

[1] Algorithms requiring fewer steps are also known for this problem [149, 22].

[2] For notational convenience, it is traditional to express the complexity in the Big-oh notation. An algorithm has time complexity $O(n^\alpha)$ if there exists a constant c such that for an input of size n, the algorithm terminates in less than cn^α steps.

[3] It is tantamount to solving the famous "P = NP?" question.

known to have such solutions. Moreover, a polynomial-time solution for any problem in the class NP-complete will imply one for every other problem in this class. In this thesis, we shall study a few NP-complete problems.

The class P has come to refer to the class of tractable problems, since most problems in this class are known to have efficient algorithms. The best time complexities of the algorithms for the problems in this class are typically a low degree (less than 10) polynomial in the input size. While this in itself is no guarantee that these algorithms are in fact practical, there is no inherent reason to believe that the solutions to these problems cannot be made practical.

On the contrary, the NP-complete problems tend to share an inherent *combinatorial explosion* bottleneck; the space of possible solutions to be searched is very large and no efficient (polynomial-time) methods for searching this space are known.[4] Readers might be familiar with the *Traveling Salesman Problem* (TSP); it is one of the most well-known problems in this class. The problem (roughly) is for a traveling salesman to optimize his/her tour length of a given set of n cities. There are $n!$ possible orderings of the cities in the tour, and the search space is prohibitively large even for, say, 500 cities ($n = 500$).

1.2 Approximation Algorithms

How shall we then go about attacking such problems? Designing faster computers can only be a weak and partial solution to the problem, since our appetite to solve more complex problems grows more quickly than the advances in computer technology can provide for. It is then not surprising that the onus for a practical solution to such a problem falls on algorithm designers. In an effort to counter the problem of combinatorial explosion, researchers design heuristics that try to search through the space of possibilities efficiently. Many heuristics for the TSP are known [50, 139, 104, 77, 78, 38, 14]. A number of general techniques are also known that can serve as a heuristic for a large class of problems. Some of the well-known techniques are the branch-and-bound methods [25], cutting-plane methods (see [121]), simulated annealing [156], genetic algorithms [68], etc. All the heuristics, however, share one drawback – while they might terminate quickly on many instances of the problem, there is no guarantee that they will do so on every instance. Frankly, we cannot expect that of *any* algorithm, simply because the problem itself seems intrinsically hard to solve.

What then is the obvious alternative if it is too hard to find the optimal solutions to these NP-complete optimization problems? In order to achieve good running times for these problems, we will have to sacrifice the quality of the solution somewhat. Such tradeoffs are not uncommon even in other branches of science. This then brings us to the soul of this thesis, namely, *approximation algorithms.*

An approximation algorithm does not aim to provide the *best* solution, it aims to provide a *good* solution *quickly.* It is characterized by two features. First, it *runs fast*, which in our case will refer to polynomial-time algorithms. Second, it comes with a *performance guarantee* that the solution produced by the algorithm is no

[4] Though NP-complete problems are generally framed as decision problems, formulating them as optimization problems is easy. Here when we refer to such problems, we mean to refer to their optimization versions.

worse than some non-trivial quantifiable description. For example, for undirected graphs whose edge-distances obey triangle inequality, there is an $O(n^3)$ time approximation algorithm [14] that finds a tour whose tour-length is never more than a factor of $3/2$ times the optimal (smallest) tour-length. As part of this work, we provide the first approximation algorithms for some very important problems in communication network design, and solving sparse linear systems of equations. These problems are known to be NP-complete.

As a warning to the reader, it should be noted that not all problems must even have approximation algorithms. In fact, there is also a large body of problems that are probably non-approximable; it can be shown that finding a good approximation for these problems is as hard as finding the optimal solution.

There are two aspects to the study of approximation algorithms, practical and theoretical, and they are not entirely divorced from each other. The prohibitive nature of the search for the best solution warrants the practicality of the approximation algorithms. Moreover, a guarantee on the solution can help evaluate the tradeoffs between the quality of the solution and the price paid to achieve it. It is also the case that in practice the solution obtained by an approximation algorithm is often better than what its worst-case theoretical analysis predicts. Even for a perfectionist desiring the best solution, starting the crank of a heuristic algorithm from a "good" solution may speed up its termination. Thus, an approximation algorithm can also be used as a preprocessing phase for many of the heuristic search algorithms. On the theoretical side, as we shall see, in establishing the performance guarantee for such problems, one often exposes the finer structure of the problems being explored. In this work we derive approximation algorithms for some problems in communication network designs, and in the process, also establish their relationships to other seemingly unrelated hard combinatorial quantities. In doing so, we have also provided practical approximation algorithms for these other NP-complete problems. Moreover, the relationships we establish provide a provably good criteria (lower bound) for judging the quality of the solution, and can be used in branch-and-bound techniques to prune the search for the optimal solution.

1.3 Approximate Min-Max Equalities

Establishing the performance guarantee is the hardest part of any approximation algorithm. We shall demonstrate the key concepts in such a process with the help of a well-known example, namely, finding a minimum edge-coloring of a graph. The problem is to assign colors to the edges of a graph such that no two edges incident to a node have the same color, and the number of colors is minimized. The problem is known to be NP-complete [80, 49].

The question is: What characteristics of an algorithm will help us claim something about its performance guarantee? Any approximation algorithm must establish the quality of its solution with respect to the best solution. However, the best solution itself is unavailable (since that is what we are seeking). We must hence establish another common ground for comparing the two solutions. The trick then is to establish a lower bound (in the case of minimization problems, an upper bound otherwise) on the best solution in terms of a quantity

to which we can relate the quality of the solution produced by the algorithm. For example, in the edge-coloring problem, by the very description of the problem, a lower bound on the number of colors required is the maximum degree of the graph.

In proving the performance guarantee of an algorithm, it is now sufficient to relate the number of colors it requires to the maximum degree of the graph. In fact, Vizing [157] showed that that the number of colors required for edge-coloring a graph is at most one plus the maximum degree of the graph. It then follows in a straightforward manner that

$$\text{max degree} \leq \text{min edge-colors} \leq 1 + \text{max degree}$$

Vizing's algorithm hence requires at most one plus the optimal number of colors. In fact we have proved more. We proved that the minimum number of edge-colors approximately equals the maximum degree of the graph. This is an example of an approximate min-max relationship. Such min-max relations provide insight into combinatorial quantities, and form the very heart of the study of combinatorics.

There has been much work in the study of exact min-max relations [71]. In fact, the origin of the study of the class of polynomial-time algorithms is linked to the study of the equality between the maximum flow and the minimum cut in a bipartite graph for single commodity flows. The study of approximate min-max equalities is more recent.

1.4 Killing Two Birds with One Stone: Approximating Both the MIN and the MAX Problems

Two facts about the approximation algorithm for the edge-coloring problem should be noted. First, the performance guarantee is an *additive constant* of 1, which is the best approximation one can achieve for a problem with an integral solution. For the problems discussed in this thesis, the guarantees on the solutions are *multiplicative* factors of the best solution, and these factors might even be functions (typically a slow growing function like logarithm) of the input size. Second, the lower bound for the number of colors was stated in terms of an easily computable quantity, namely, the maximum degree of the graph. Quite often, as we shall see in this thesis, the lower bound might itself be a combinatorially hard problem to solve. While the challenge for solving the problem escalates in such cases, so do the rewards.

Consider a minimization problem with minimum value X^*. Let us assume that we are able to express its lower bound as another maximization problem with maximum value Y^*. We shall refer to this maximization problem as the *dual* of the minimization problem. This lower bound gives us the following relationship:

$$Y^* \leq X^* \tag{1.1}$$

Now, consider an instance of the maximization problem with value Y, and an instance of the minimization

problem with value X. Then we trivially also have

$$Y \leq Y^* \tag{1.2}$$

$$X^* \leq X \tag{1.3}$$

since the value of any feasible solution for the maximization problem can be at most the maximum value, and for a minimization problem must be at least the minimum value.

If we could construct instances Y, and X such that their values are related, for example, if

$$X \leq f \cdot Y \tag{1.4}$$

for some small quantity f, then we can observe from (1.1), (1.2), (1.3) and (1.4) that

$$X \leq f \cdot X^* \tag{1.5}$$

$$Y \geq \frac{1}{f} Y^* \tag{1.6}$$

This implies that the instances we have constructed are both within a factor of f of their optimal values, and we have approximation algorithms for both the minimization and the maximization problems *simultaneously.* This is the approach we shall be taking in this work. For the network design problems that we consider here, not only do we provide approximation algorithms for the original minimization problems but also provide approximation algorithms for their dual maximization problems. As we shall see, approximating these dual problems also have interesting applications.

1.5 Our Technique

We noticed in the example of the edge-coloring problem that we could easily establish a useful lower bound on the minimum number of edge-colors, and this was crucial to proving the performance guarantee from Vizing's result. The question then arises as to how shall we establish good lower (upper) bounds for a combinatorial minimization (maximization) problem? Are there any general techniques for doing so?

Unfortunately, no such general techniques are known. However, in this work, we have identified a technique that draws from the linear programming domain, and is applicable to a wide range of problems. While the technique itself is not the focus of our work, we feel that further investigation of its issues might produce a broad classification of hard optimization problems that can be well approximated through this technique.

The technique is the following. Consider the optimization problem we want to approximate. For the discussion here, let us consider a minimization problem X. We give a four-step recipe for establishing a lower bound for X in terms of another combinatorial quantity.

1) Formulate the optimization problem X as a (mixed) 0-1 integer program. A mixed 0-1 integer program is represented by an optimization function and a set of constraints, each of which is linear in the variables for the problem. In addition, some or all of the variables are constrained to take on only 0-1 values.

2) Relax the integrality constraint in (1) to form a linear program.

3) Form the linear program that is dual to the linear program in (2). This new linear program is a maximization problem.

4) Constrain the dual linear program in (3) by requiring that certain variables take on *uniform* values, i.e. in any solution, all the variables with non-zero values take on the same value. For example, certain variables might be constrained to take on 0-1 values, in which case the constrained program is a (mixed) integer program. However, in general, a uniform solution is not an integral solution.

   Now, interpret this constrained program as the solution of some combinatorial maximization problem Y.

Let us consider the equalities and inequalities that follow:

$$\begin{aligned} \min X &= \min \{\text{solution of the mixed integer program in (1)}\} && (1.7)\\ &\geq \min \{\text{solution of the linear program in (2)}\} && (1.8)\\ &= \max \{\text{solution of the dual linear program in (3)}\} && (1.9)\\ &\geq \max \{\text{solution of the constrained program in (4)}\} && (1.10)\\ &= \max Y && (1.11) \end{aligned}$$

Equalities (1.7) and (1.11) follow from the construction. Inequalities (1.8) and (1.10) follow from the fact that any solution to a constrained program is also a solution to the corresponding linear program. Equality (1.9) follows from the standard theory on duality of linear programs.

Thus we have achieved a lower bound on the minimum value of X in terms of the maximum value of another combinatorial quantity Y. The challenge in successfully applying the above 4-step recipe lies in finding a good formulation of the given problem (step 1) and in interpreting the dual program (step 4). These two steps can be quite challenging. Developing the right combinatorial quantities to work with is half the battle in establishing such approximate min-max equalities. In this thesis we have successfully applied the above technique to solve two problems in network design. Though we do not explicitly give any integer programs in this thesis, underlying each min-max relation that we establish is the 4-step recipe given above.

Min-max relations between combinatorial quantities by means of linear programming duality is a well studied branch of combinatorics. Thus, in some sense, our approach is not new. However, such studies in the past have been restricted to showing either exact min-max equalities or an approximate equality between the value of an integer program and its linear relaxation. Our study identifies it to be a contender technique even for proving approximate min-max relations. Researchers in the past have been able to characterize classes of optimization problems for which exact min-max relations hold. These characterizations are based on the structure of the constraint matrix used in formulating the linear program, and many general theorems exist in this field. The reader is referred to the book by Grötschel, Lovász, and Schrijver [71] for an excellent description

of the whole area. We have faith that similar characterizations for showing *approximate* min-max equalities will be successfully explored in the future.

1.6 Linear and Integer Programs

Solving integer programs is a hard problem. No polynomial-time algorithm for the general class of integer programs is known [49]. However, polynomial-time algorithms for solving linear programs are well-known (see [87, 71]).[5] For many integer programs, it can be shown that an optimal solution to the linear programs without the integrality constraints, can be easily rounded to produce a good approximation to the optimal integral solution. Even in the absence of a theoretical analysis of the quality of the rounded solution, such a method is often used as a heuristic for approximating the solution to the integer program.

1.7 Our Contributions

The thrust of this thesis is towards the design of approximation algorithms for network design problems. A network design problem is to provide communication channels between some prespecified locations at minimum cost. The cost criterion is typically different for different applications. We study two different popular cost criteria. The problems we study are known to be NP-complete, and we provide the first approximation algorithms for them. Our results have applications in optimally laying down telephone, electrical or computer cables, choosing routes for airline industry, designing robust communication networks, etc.

Our work is based on a common theme of establishing and exploiting the combinatorial relationships between "designing" networks and their combinatorial dual problems, namely, "blocking the design of" (or "splitting") such networks. In the course of establishing these relationships, we also provide the first approximation algorithms for some other NP-complete problems relating to splitting a network. These approximations, in turn, can be applied to estimating the reliability of a network under random failures, identifying weak spots in a network against inadvertent or malicious failure of its components, etc.

We also provide the first approximation algorithm for minimizing the space and time requirement for solving certain sparse linear systems using Gaussian elimination. Solving sparse linear systems are at the very heart of numerical analysis, and form the core of a vast majority of numerical problems in science and engineering. Our algorithm is based on a previously known result by Leighton and Rao [96] relating to splitting a network.

1.8 A Road-map of This Thesis

Chapter 2 presents a comprehensive description of the problems we consider in this thesis. In this chapter, we provide the motivation for considering these problems, and establish lower bounds for some of the problems

[5] The most popular algorithm for solving linear programs is however, the Simplex Algorithm [26], which requires exponential time in the worst case. However, in practice it performs very well.

using the technique we described earlier in Section 1.5. In this process, we introduce the other combinatorial quantities relevant to this work. We also present an overview of the main results presented in this thesis. Chapters 3, 4 & 5 deal with these results in depth. Further details about these chapters are presented at the end of Chapter 2. Chapter 6 details some of the open issues relating to this work.

Chapter 2

Problems and Results

2.1 Introduction: Network Design

A network design problem aims at designing a network of communication links capable of handling prescribed communication requirements at minimum cost. Such problems can be applicable in diverse settings. The communication links could take the form of highways, air routes, postal mail links, telephone links, or power lines, among others. The communication requirements could correspondingly denote transportation requirements of physical commodities, or physical connections between points, as in telephone or power industry. The cost criteria however, can differ tremendously with applications.

Being of fundamental importance to many industries, network design problems have received much attention (see [118, 45]). Garey and Johnson [49] give a comprehensive list of many NP-complete problems in network design.

This thesis is divided into three parts. The first two parts establish new approximation algorithms for two network design problems and relate them to problems in network cuts. The third part applies a previous result relating to network cuts to a problem in solving sparse linear systems. This chapter gives an overview of our main results. We introduce the problem of network design formally in Section 2.2. Our results for the two network design problems are presented in Sections 2.3 and 2.4. Section 2.5 presents our result concerning solving sparse linear systems. For the three problems considered here, we derive good lower bounds in this chapter. We thus introduce other combinatorial quantities relevant to this thesis, and present results regarding these combinatorial quantities. We close the chapter by describing the organization of the rest of this thesis in Section 2.6.

2.2 Problem Definitions

A *requirement-multijoin problem* consists of a graph $G = (V, E)$ together with a collection $R = \{R_1, \ldots, R_k\}$ of *connectivity requirements*; each connectivity requirement R_i consists of a *site pair* $\{s_i, t_i\}$, $s_i, t_i \in V$, and

Problem	*Specification of network connectivity*
R-multijoin	Requirement site pairs with integral requirement values
R-join	Requirement site pairs with unit requirement values
Uniform R-join (Steiner tree)	Any subset of the nodes of the graph
Spanning R-join (Spanning tree)	All the nodes of the graph

Table 2.1: The different specifications of a network design problem.

a positive *requirement value* r_i. A feasible solution, which we call a *requirement multijoin* is a multigraph N consisting of zero or more copies of the edges of G, such that for every requirement pair $R_i = (\{s_i, t_i\}, r_i)$, there are at least r_i edge-disjoint paths between s_i and t_i. Note that the paths for different requirement pairs are allowed to overlap. For ease of notation, we shall refer to a connectivity requirement simply as a *requirement*, and a requirement multijoin by an *R-multijoin*. The endpoints of the requirements will be called *sites*. The sites of a requirement are called *mates*.

When all the requirements have value 1, then we shall refer to the R-multijoin problem as a *requirement join* (henceforth *R-join*) problem and the solution as an *R-join*. The R-join then simply refers to a network in which there is a path between the nodes of every requirement site pair. Moreover, since we require only a single path between the sites, any minimal R-join will never contain multiple copies of an edge. Note that the solution network need not be connected, as a set of disconnected paths might be sufficient to establish the required connectivities.

An important further restriction of the R-join problem is when all we require is that a given set of sites, $S \subseteq V$, be connected through the final network. Such a problem can easily be represented as an R-join problem in which we specify a unit requirement between every pair of sites in S. We shall refer to the input requirements in this case as *uniform* requirements, and call this problem a *uniform R-join* problem. A uniform R-join is known as a *Steiner tree* in the literature. An important subclass of the uniform R-join problem is the *spanning R-join* problem, where it is required that we connect all the nodes of an input graph, and that the solution is a spanning tree. Table 2.1 summarizes the above definitions.

In representing the uniform R-join problem in a naive fashion, $|S|^2/2$ requirements must be specified, one between each distinct pair of nodes in S. However, observe that the connectivity relation is transitive; paths between site pairs $\{a, b\}$ and $\{b, c\}$ in a network imply a path between the site pair $\{a, c\}$. With this observation, we see that $|S| - 1$ requirements are, in fact, sufficient to represent the uniform R-join problem. We choose some site $v \in S$, and specify a unit requirement between v and every other node u in S ($u \in S, u \neq v$). In the final network, there is a path from every node in S to u, thus giving us the desired paths between every pair of nodes. A compact representation is useful for implementation purposes.

2.3 Minimum Edge-Cost R-join

We start out by motivating the first of the two network design problems considered in this thesis. We then present the results relating to this problem.

2.3.1 The First Approximation Algorithm

Consider the following scenario: Client industries of a telephone company have requested commercial telephone connections between pairs of their offices in different cities. The telephone company must then install a network of fiber-optic telephone links that accommodates all the clients' requirements; there is a path using these links between every pair of cities specified by the clients. Given the cost of installing links between different cities, the company must now decide which links to install so as to minimize its cost.

We refer to the above problem as a *minimum edge-cost R-join* problem. Formally, the problem is as follows: Given a graph $G(V, E)$ with nonnegative edge-costs, and a set of unit requirements, the objective is to find an R-join of minimum total edge-cost.

In the above problem, it is assumed that a link can be used simultaneously by many clients. This model is justified by the use of fiber-optic links which have sufficiently high bandwidth that they are unlikely to be saturated even with multiple calls routed through them simultaneously.

This problem can also be applicable to industries other than the telephone industry. For example, in transportation industry, an airline (rail company, or a postal company) may aim to service pairs of cities, and wants to minimize its cost of buying the air routes (installing the rail lines, or having courier services). The design problem can also be applied to a network of power lines, to electrical wiring in a building, or to computer networks.

The version of this problem with uniform requirements is known in the literature as the *Steiner problem in networks*, and this problem is known to be NP-complete. In fact, it belonged to the list of first seven problems shown NP-complete by Karp [88]. Given the range of its applications, it is not surprising that this problem has been well-studied. Many enumeration algorithms [75, 4, 28, 94, 146, 8, 5], heuristics [135, 163, 83], and approximation algorithms [152, 93, 41, 125, 150, 116, 166] are known for the problem. polynomial-time solutions for restricted classes of graphs are also known [160].

However, none of these algorithms address the more general problem of handling requirements between pairs of sites. We address this problem here, and provide the first known approximation algorithm for the problem.

Theorem 2.3.1 *There is an $O(mn)$ time algorithm for finding an R-join of edge-cost at most $2-2/k$ times optimal, where k is the number of sites specified by clients, and m and n are respectively the number of edges and nodes in the input graph.*

If all the edges have have the same cost then minimizing the cost of an R-join is equivalent to minimizing the number of edges in the R-join. For this case, we have an algorithm with a better running time than the

case with arbitrary edge-costs. This case also forms the basis for other results in this section.

Theorem 2.3.2 *There is an $O(m\alpha(m,n) + n\log n)$ time algorithm for finding an R-join of size at most $2 - 2/k$ times optimal, where α is the inverse Ackerman's function.*

We also consider the problem with arbitrary integral requirement values. This problem is known in the literature as the *generalized Steiner problem in networks* (see [160]). The motivation for studying this case is to achieve higher network reliability. Consider the situation in which a client requires a reliable communication between its specified site pair. In the presence of communication-link failures, higher reliability can be achieved by requiring multiple paths between the site pairs, and this justifies the need for arbitrary requirement values. Note that in such a network; if k paths for a requirement must use the same edge, we must install k distinct physical links corresponding to that edge. The resulting network is hence a multigraph.

We provide the first approximation algorithm for the problem with arbitrary requirement values. We decompose the problem into multiple problems with unit requirement values, each of which we approximate using our algorithm of Theorem 2.3.1. The performance guarantee for the case of arbitrary requirement values is hence worse than that for the case of unit requirement values.

Theorem 2.3.3 *There is an $O(mn\log|R|)$ algorithm for finding an R-multijoin of cost at most $(2 - 2/k)\lceil\log_2(|R|+1)\rceil$ times optimal, subject to arbitrary integral edge-connectivity requirement values r_{ij}, where $|R|$ is the largest requirement value.*

An approximation algorithm for a restricted version of this problem, namely the *survivable network design* problem, was previously presented by Goemans and Bertsimas [72].

2.3.2 A Combinatorial Basis: Relation to Packing of R-cuts

Our combinatorial proof of the approximation algorithm is different from any of the previous approximations known even for the restricted version of the problem. Our proof establishes an approximate min-max equality between the minimum-cost R-join and a combinatorial quantity relating to *requirement cuts.* A *requirement cut* consists of a set of edges whose removal cuts all paths between the sites of at least one requirement pair; on removing the edges in the cut from the graph, the endpoints of at least one requirement pair belong to different connected components. We shall refer to a requirement cut by an *R-cut.*[1] Note that on removing the edges of an R-cut, no R-join can now be established since there is at least one requirement pair whose sites cannot be joined by any path. Thus any R-join must contain an edge from every R-cut. This gives us the following simple lemma that will be useful in establishing a lower bound for the problem of minimum edge-cost R-join.

Lemma 2.3.1 *Every requirement-join and every requirement-cut have at least one edge in common.*

[1] Our terminology of R-joins and R-cuts is analogous to the T-joins and T-cuts that are well-known in the literature (see [71, 114])

We shall now establish a lower bound for the minimum edge-cost R-join problem. We shall first consider the case of all the edges having the same cost. Later we shall relax this restriction.

Unweighted case

We shall first consider the unweighted case where each edge has unit cost. The cost of an R-join hence equals the number of edges it has, which we shall refer to as its *size*.

Before we establish a lower bound for this case, we need some definitions. We shall refer to a collection $\mathcal{C}$ of R-cuts as a $\frac{1}{k}$-*packing of R-cuts* if no edge of the graph is contained in more than k cuts in $\mathcal{C}$. For example, a 1-*packing* of R-cuts is a collection of edge-disjoint R-cuts. For $k = 2$, the cuts in $\mathcal{C}$ are nearly edge-disjoint; each edge is in at most 2 cuts. With these definitions, we are now ready to establish the following lower bound on the minimum size of an R-join.

Lemma 2.3.2 *The minimum size of an R-join is at least one-half the maximum size of a $\frac{1}{2}$-packing of requirement cuts.*

Proof: Let N be a network design and let $\mathcal{C} = \{A_1, \ldots, A_p\}$ be a $\frac{1}{2}$-packing consisting of p requirement cuts. By Lemma 2.3.1, N must contain an edge from each of the cuts in $\mathcal{C}$. Since an edge can be in at most two cuts, it follows that N must have at least $\frac{p}{2}$ distinct edges. □

One of the main results of our work is to show that the above lower bound is extremely good. We prove that the two combinatorial quantities in Lemma 2.3.2 are very close in value.

Theorem 2.3.4 *The minimum size of a requirement join is at least one-half, and at most $1 - 1/k$ times, the maximum size of a $\frac{1}{2}$-packing of requirement cuts, where k is the number of sites.*

One may wonder why we have a $\frac{1}{2}$-packing and not a 1-packing in the above results? For one thing, the values of the maximum 1-packing and the minimum-size R-join are not always close to each other. For example, in the spanning tree problem on a complete graph, the minimum size of a spanning tree is $n - 1$, but the maximum size of a 1-packing is 1. This result is not surprising in the light of a result by Colbourn [21]; Colbourn gave an approximation-preserving reduction from the problem of finding a maximum-size 1-packing to the maximum independent set problem.[2] Strong evidence exists that the latter problem is hard to approximate [42]. In the light of this discussion, our choice of $\frac{1}{2}$-packing in the above approximate equality is not coincidental. It turns out that the above relation is existentially tight; a minimum-size spanning tree for a ring graph with n nodes has $n - 1$ edges while the size of any maximum $\frac{1}{2}$-packing of cuts is n.

The story of this approximate min-max relation does not end here. The potential of our technique used in proving Theorem 2.3.4 has been further exploited in the recent work of Goemans and Williamson [73]. They

[2]Colbourn showed that finding the maximum 1-packing is NP-complete. It is easy to see that his transformation is also approximation-preserving.

extended our result to give an approximation algorithm for finding a minimum set of edges that intersect a prespecified set of edge-cuts. In fact, their algorithm works even when the set of edge-cuts can be compactly replaced with an oracle which answers if a given set of edges forms a cut of interest. Using this formulation, they obtain new approximation algorithms for a variety of problems like prize-collecting TSP, prize-collecting Steiner tree, location design, minimum-weight perfect matching with triangle inequality, etc.

Weighted case

For the case when the edges of the graph have arbitrary positive cost, we obtain an approximate min-max relation similar to the unweighted case.

We need a few definitions before we can introduce this new relation. Let $G(V, E)$ be a graph with real non-negative edge-costs c. A *weighted collection* of R-cuts $\mathcal{C}$ consists of a set of R-cuts along with a positive weight λ_i for each cut $C_i \in \mathcal{C}$. A weighted collection is a *c-packing* if it has the following property: for each edge $(u, v) \in E$, the sum of the weights of all the cuts in $\mathcal{C}$ containing the edge is at most its cost $c(u, v)$. For example, in the unweighted case, a $\frac{1}{k}$-packing of cuts can be considered as a packing where each cut has a weight of $\frac{1}{k}$. Since an edge can be in at most k cuts, it follows that the sum of the weights of the cuts containing an edge is at most one. We define the *value* of a packing as the sum of the weights of all the cuts in the packing. For example, a $\frac{1}{k}$-packing of cuts in the unweighted case has value $\frac{p}{k}$, where p is the number of cuts in the packing. By a *maximum c-packing* of cuts, we refer to a c-packing of maximum value. Note that the weights of the cuts in a c-packing can be arbitrary fractional values.

We prove the following approximate min-max relation between the minimum cost R-join and a maximum packing of R-cuts.

Theorem 2.3.5 *The minimum-cost of a requirement-join is at least as much as the value of a maximum c-packing of requirement cuts, and at most* $2(1 - 1/k)$ *times this value, where* $c(u, v)$ *denotes the cost of an edge* $(u, v) \in E$.

The above theorem has been crucial in establishing a new min-max relation for yet another network design problem presented in Section 2.4. This is an example of how one min-max relation helps establish other min-max relations which in turn provide further approximation algorithms.

2.3.3 Packing of Cuts: Approximations and Applications

$\frac{1}{2}$-Packing of cuts and network reliability

In establishing the min-max relation of Theorem 2.3.4, we obtain an approximation algorithm for finding a maximum-sized $\frac{1}{2}$-packing of R-cuts.

Theorem 2.3.6 *There is an $O(m\alpha(m,n)+n\log n)$ time algorithm for finding a $\frac{1}{2}$-packing of requirement cuts whose size is at least $\frac{1}{2-2/k}$ times optimal, where α is the inverse Ackerman's function, k is the number of distinct sites, and m and n are respectively the number of edges and nodes in the graph.*

This algorithm can be applied to estimate the reliability of a network under random failure of its edges. The network reliability problem is the following. Given the failure probabilities for the edges of a network, find the probability that all the given communication requirements continue to be satisfiable under a random failure of edges.

The reliability problem is notoriously hard. It is known to be $\#P$-complete even in the restricted cases of a single requirement pair [155, 129], and a spanning R-join formulation [129]. There is a large body of literature addressing the reliability problem for the two cases mentioned above. For these two cases, many exponential-time exact algorithms are known [145, 158, 120, 27, 119, 46, 140, 6, 130]. Polynomial-time algorithms for the following restricted classes of graphs are also known: trees [153], series-parallel graphs [37], subclasses of planar graphs [9, 141, 126], and complete graphs when all edges have the same failure probability [44]. For arbitrary graphs, ways of deriving lower bounds [127, 18, 12, 113], and upper bounds [112, 21, 20] have also been studied. However, no previous work addresses the reliability problem for the case of arbitrary requirement pairs. Our work addresses this general case, and provides a new heuristic for the reliability problem.

One of the best known heuristic for estimating the reliability of a network in the case of the spanning R-join formulation is the following. Find a large collection $\mathcal{C}$ of edge-disjoint R-cuts. For a surviving network to satisfy all the requirements, at least one edge must survive in each of the cuts $C \in \mathcal{C}$. For a cut C, the probability p_C that all the edges have failed in the cut, is the product of the failure probabilities of each of the edges in C. The probability that at least one edge survives in C is $1 - p_C$. The probability that for each cut $C \in \mathcal{C}$, an edge of C survives is then given by $\prod_{C\in\mathcal{C}}(1 - p_C)$. This value is an upper bound on the reliability of the network and provides one of the best known estimates for the reliability problem.

To obtain a good estimate using the above method, we must find a large collection of edge-disjoint cuts with as few edges as possible. However, finding an edge-disjoint collection of R-cuts of maximum size is hard (see discussion in Section 2.3.2), and no approximation algorithms for the problem are known. Hence this heuristic itself is hard to implement efficiently. Lomonosov and Polesski [112] gave an alternate upper bound for the reliability problem using a set of "uncrossed" cuts.

We suggest a new heuristic for reliability estimation that is adapted from the above heuristic. This heuristic can be efficiently implemented, and also addresses the reliability problem in the general context of arbitrary requirement pairs. The heuristic is the following. We transform the original graph into an auxiliary bipartite graph by replacing an edge (u, v) having failure probability $1 - p$ with two edges (u, x) and (x, v) each having a failure probability of $1 - \sqrt{p}$, where x is a new node. The reliability of the network does not change with this transformation since the probability that the edge (u, v) survives in the original graph is the same as the probability that both the edges (u, x) and (x, v) survive in the auxiliary graph. By employing our algorithm

of Theorem 2.3.6, we can efficiently find an approximately maximum sized collection of edge-disjoint R-cuts in this auxiliary bipartite graph. We can then estimate the reliability of the network in a manner described earlier.

c-packing of R-cuts

Through our algorithmic proof for Theorem 2.3.5, we obtain a polynomial-time approximation algorithm for finding a maximum fractional c-packing of R-cuts. Its value is within a factor of 2 of the optimal. The packing of cuts obtained by our algorithm also has a certain nice property, namely, it is *cross-free*. The cross-free property is of importance in approximating the *node-toughness* of a graph. The discussion of the cross-free property and node-toughness is more relevant to another part of this thesis, and we postpone their discussion until then.

2.4 Minimum-degree Uniform R-join

Now we come to the second network design problem, and we present our results relating to this problem. The problem is to design a network whose maximum degree is minimum. We shall refer to this problem as the *minimum-degree R-join problem*, and to the degree of the optimal network as the *minimum-degree*.

The minimum-degree R-join problem arises in the design of robust decentralized communication networks. The advantage of a network with low degree is that the failure of a node does not adversely affect the rest of the network too much. This problem is also important in the design of switching networks where the same switch is to be installed at each of the nodes. The switch must then be designed to handle as many connections as the maximum degree of any node in the network in which it is to be used.

2.4.1 The First Approximation Algorithm

We study the minimum-degree *uniform* R-join problem. Recall that in a uniform R-join problem, a prespecified set of sites must be connected through the final network. Finding a minimum-degree network is NP-hard, even in the special case when all the nodes must be connected by a spanning tree of smallest maximum degree; a spanning tree of degree two implies the existence of a Hamiltonian path in the graph. A polynomial-time algorithm for the problem is hence unlikely. In this work, we present the first known approximation algorithm for the problem.

Theorem 2.4.1 *Given a set of uniform requirements, there is a polynomial-time algorithm for finding a requirement-join whose degree is $O(\Delta^* \log k + \log n)$, where Δ^* is the optimal degree, k is the number of sites, and n is the number of nodes in the graph.*

Fürer and Raghavachari [47] gave an approximation algorithm for the minimum-degree spanning tree problem. The performance guarantee of their algorithm is $O(\log n)$. Our performance guarantee thus matches theirs for this special case. Gavish [51] formulated the minimum-degree problem as a mixed integer program

and provided an exact solution using the method of Lagrangian multipliers. Lawler [94] showed that matroid methods suffice to solve the following variant of minimum-degree spanning tree problem in polynomial time: given a graph G and an independent set I of nodes of G, find a spanning tree that minimizes the maximum degree of any node in I.

Our result extends to the weighted version of the spanning R-join problem. In the weighted version, each node has an associated non-negative cost denoting the cost of having an edge incident to that node in the network. The node-costs capture the physical cost of connecting an edge to a node. The relative preferences of different nodes can also be captured using node costs. The weighted degree of a node in a network is the product of the cost of the node and its degree in the network. The weighted minimum-degree problem is to design a network whose maximum weighted degree is minimum.

2.4.2 A Combinatorial Basis: Relation to Node-toughness

At the heart of our analysis is a combinatorial min-max relation relating the minimum-degree to *node-toughness* in a graph $G(V, E)$. *Node-toughness* is a measure of how easy it is to disconnect a graph into many components by removing few nodes. The node-toughness of a graph is the minimum ratio of nodes removed to the number of resulting components, namely

$$\min_X \frac{|X|}{\text{number of components of } G - X} \tag{2.1}$$

where X ranges over all node-subsets of G. We shall henceforth refer to node-toughness simply as *toughness*.

Toughness was first defined by Chvátal [17] in a slightly different form. Chvátal defined toughness to be the minimum value of the ratio (2.1) where X ranges only over those node-subsets of V whose removal yields at least two components, i.e. the number of components in $G - X$ is at least 2. By our definition of toughness, the toughness of a graph cannot exceed 1 since the removal of a single node trivially yields at least one component. This upper bound on toughness is not guaranteed by Chvátal's definition. Computing the toughness of a graph (as defined by Chvátal) was recently shown NP-complete by Bauer et al. [7].

To motivate the relation between minimum-degree networks and toughness, let us consider the simple case of a spanning R-join problem in which every node of the graph is a site. Let X be any node-subset. In a spanning tree, every component of $G - X$ has at least one edge to X. Hence the number of spanning tree edges incident to X is at least $\gamma(G - X)$, the number of components of $G - X$. It follows by averaging that some node in X has at least $\gamma(G - X)/|X|$ incident spanning tree edges. Thus we obtain the following simple lemma.

Lemma 2.4.1 *The minimum degree of a spanning tree is at least the reciprocal of node-toughness.*

This observation motivates the definition of *Steiner node-toughness*. Suppose we are given a set of sites. For a node-subset X, a component of $G - X$ is said to be *significant* if there is at least one site in this

component and at least one site not in this component. The Steiner toughness is the minimum ratio

$$\frac{|X|}{\text{number of } \textit{significant} \text{ components of } G - X} \tag{2.2}$$

where X ranges over all node-subsets of G. Note that in a spanning R-join formulation every component is significant.

The following lemma is easily proved in the same way as Lemma 2.4.1.

Lemma 2.4.2 *The minimum-degree of a uniform requirement-join is at least the reciprocal of Steiner toughness.*

Thus the reciprocal of Steiner toughness provides a lower bound on the minimum degree. In fact, we show that it provides a very good lower bound by proving the following approximate min-max equality.

Theorem 2.4.2 *The minimum degree of a uniform requirement-join is at least the reciprocal of Steiner toughness, and is $O(\log n)$ times this quanitity, where n is the number of nodes.*

Interestingly, the proof of this approximate min-max equality relies on the min-max equality for the uniform edge-cost R-join problem given in Theorem 2.3.4, thus reinforcing the power of such combinatorial relations.

Theorem 2.4.2 also represents progress on an unresolved conjecture of Chvátal's. Chvátal observed that every graph with a Hamiltonian cycle has node-toughness at least one—deletion of x nodes results in at most x components. He conjectured that there is a constant t_0 such that every graph with toughness at least t_0 is Hamiltonian. Our min-max equality of Theorem 2.4.2 demonstrates a concrete relationship between toughness and minimum degree. As a consequence, for example, it follows that every graph with toughness at least one has a spanning tree of degree $O(\log n)$.

2.4.3 Node-toughness: Approximations and Applications

Towards proving the relation between the node-toughness and the minimum-degree for a uniform R-join problem, we obtain a constant factor approximation algorithm for Steiner toughness. In fact our algorithm can approximate toughness in the more general setting with requirement site pairs. A significant component in this setting is defined as a component which contains exactly one site of some requirement pair. We shall refer to this generalization of toughness as the *generalized Steiner toughness.*

Theorem 2.4.3 *There is a poly-time algorithm to determine a node-subset $|X|$ for which the generalized Steiner toughness is within a factor of 8 of the minimum.*

In the algorithm, we require a large packing of uncrossed cuts, and we use our algorithm of Theorem 2.3.6 to generate such a packing (refer to the discussion at the end of Section 2.3.3).

The above algorithm is the first approximation algorithm known for the node-toughness problems. A node subset with minimum toughness ratio is the weakest spot in a network; by disabling this set of nodes,

an adversary can inflict the maximum loss in communication per node disabled. Toughness is thus useful in analyzing the resiliency of communication networks to worst-case failures.

Approximating node-toughness in turn leads to other useful algorithms. By iteratively applying the algorithm for toughness, one can break a graph into many (significant) components by removing the set of edges incident to an approximately minimum number of nodes. Namely, repeatedly find a node-subset X that approximately minimizes the toughness ratio, and remove the edges incident to the nodes of X. If the number of components achieved at some stage is k, the number of nodes whose incident edges have been removed is approximately minimum to achieve k components. The approximation factor is $O(\log n)$, where n is the number of nodes in the graph.

Using the same approach, one can approximately compute another graph quantity—the *dissociation number*. Papadimitriou and Yannakakis [122] defined the dissociation number of a graph to be the minimum number of nodes whose removal leaves only isolated nodes and edges. They show that this quantity arises in determining the complexity of certain constrained spanning tree problems. They state, however, that computing the dissociation number of an arbitrary graph is an NP-hard problem. A greedy application of our toughness algorithm yields an approximation algorithm for the dissociation number.

Theorem 2.4.4 *There is a polynomial-time algorithm for finding an approximately minimum set of nodes in a graph whose removal leaves only isolated nodes and edges. The performance guarantee is* $O(\log^2 n)$ *where* n *is the number of nodes in the graph.*

In fact, our algorithms for approximating the node-toughness problems can easily be adapted to edge-toughness problems. Edge-toughness is analogous to node-toughness, and was also formulated by Chvátal [17]. However, the edge-toughness is computable in polynomial time, as was shown by Cunningham [23] and Gusfield [74]. The notion of edge-toughness generalizes to a notion of Steiner edge-toughness. It can be easily shown from our techniques that even this more general quantity can be computed in polynomial time.

2.5 Finding an Elimination Ordering

Now we come to the results in the third part of the thesis. We apply some previously known results relating to another combinatorial quantity in requirement cuts, namely, balanced node-separators in a graph, to a problem in Gaussian elimination.

Solution of linear systems of the form $Ax = b$ is a basic tool in numerical analysis and is commonly used in almost all branches of science and engineering. One of the most popular direct methods of solving a system of linear equations is the Gaussian elimination method, in which the matrix A is first transformed into an upper triangular matrix by forward elimination, and then the solution is found by backward substitution. In the ith step of the forward elimination, the ith equation is used to eliminate the ith variable from all equations numbered higher than i. For numerical stability reasons, it is often necessary to make some row and column

interchanges before an elimination step. A straightforward implementation of this algorithm requires $O(n^3)$ time in the worst-case, where n is the number of equations. However, the coefficient matrix A is often sparse, i.e. has very few non-zero elements. Since the computations on the zero elements in A can be performed trivially without requiring any floating point operations, it is important to take advantage of the sparsity to keep the computational complexity low.

In Gaussian elimination, as the elimination phase progresses, new non-zero elements are introduced into the coefficient matrix, and the matrix tends to become less sparse. The loss in sparsity translates into increased storage requirement, and also increased computation time since the non-zero elements enter into subsequent calculations.

The new non-zero elements introduced during the elimination process are called *fill-in*. Different orders of eliminating the variables may yield very different fill-in. It is thus of prime importance to be able to choose an elimination ordering of the variables that results in small fill-in. The choice of an elimination ordering that results in small fill-in often conflicts with the requirement of an ordering that ensures numerical stability of the solution process. Fortunately, there is a large and powerful class of problems, namely *positive-definite* systems of equations, in which numerical stability is not a problem [57]. In solving such linear equations, we are free to choose an ordering of variables entirely based on our desire to preserve the sparsity of the matrix during the elimination process.

A matrix A is called *symmetric* if A_{ij} equals A_{ji} for every i, j. Positive-definite matrices that are also symmetric frequently arise in structural analysis, signal processing, economics, VLSI simulation, solution of linear programs, and solution of partial differential equations, to name a few. In this work, we study the problem of finding a good elimination ordering for such matrices.

Henceforth, when we refer to a linear system of equations $Ax = b$, we assume that A is a symmetric positive-definite matrix.

2.5.1 Minimizing Fill

We shall define the *fill* for an ordering as the sum of the number of non-zero elements in the matrix, and the fill-in introduced by the ordering. The fill for an ordering measures the amount of storage required, and also has bearing on the total time required for the elimination process. It is thus of interest to find an elimination ordering that minimizes fill.

Finding such an ordering is NP-complete [165]. Nevertheless, this problem is of fundamental importance and a large set of ordering heuristics have been developed [24, 137, 54, 60, 111, 56, 55, 105, 58] (also see [57, 36, 48]). However, none of them are known to give any performance guarantee on the size of the fill. In this work, we present the first polynomial-time algorithm that guarantees approximately optimal fill. Our algorithm performs particularly well when the number of elements in each row (and hence each column) of the matrix is small. Fortunately, many problems in practice, especially those arising from finite-element methods, have such a property due to the physical constraints of the problems being modelled. As stated earlier, all our results are

for the class of symmetric positive-definite systems of equations.

Theorem 2.5.1 *There is a polynomial-time algorithm that finds an elimination ordering yielding approximately optimal fill. The fill for the ordering is within a factor of* $O(\sqrt{d}\log^4 n)$ *of the optimum, where* n *is the number of variables and* d *is the maximum number of non-zero entries in any row or column of the coefficient matrix.*

Our algorithm is a variant of the well-known nested dissection algorithm [54]. It treats the input matrix as the incidence matrix of a graph. The algorithm is based on finding a recursive decomposition of the graph associated with the coefficient matrix. The use of graphs in the study of elimination ordering is not new [123, 136]. There is an obvious way of associating a graph with a symmetric matrix; the variables of the matrix associate with the nodes of the graph, and there is an edge between nodes i and j iff the element (i, j) of the matrix is non-zero. The values of the non-zero elements in the coefficient matrix are not relevant in the case of the symmetric positive-definite systems.

The nested dissection algorithm was first proposed by George [54]. The algorithm was shown to be optimal (within constant factors) in terms of operation count for the case of a regular finite-element mesh [79]. There is much literature available on the subject [53, 54, 31, 102] (also see [57, 36]). Node separators are fundamental to this algorithm. A node-separator consists of a set of nodes whose removal breaks up the graph into pieces. For our purposes, every subset of the nodes is a separator. A separator is *f-balanced*, for some $f < 1$, if no piece on its removal has more than fn nodes, where n is the number of nodes in the graph.

Lipton, Rose, and Tarjan [102], showed that their nested dissection algorithm does well on any class of graphs that are known to have small *balanced* separators. The fill for their algorithm depends upon the size of the separators for the class of graphs being dealt with. Their algorithm, for example, yields $O(n \log n)$ fill for any planar graph based on the facti that planar graphs have $\frac{2}{3}$-balanced separators of size $O(\sqrt{n})$. There are two main differences between the work of Lipton, Rose, and Tarjan and our work. One, we do not assume the existence of any special separator structure in the graphs. Two, our analysis is more strict; we are able to analyze the quality of our result with respect to the minimum fill achievable over all orderings.

Gilbert [63] showed that for a matrix with at most d non-zero elements in each row (and column), there exists a nested dissection algorithm whose fill is $O(d \log n)$. His nested dissection algorithm, however, is inherently non-constructive; the choice of separators in his algorithm depends crucially on the optimally filled matrix.

Our nested dissection algorithm also uses balanced node-separators. No polynomial-time algorithms are known for finding a minimum-size balanced node-separator in a graph. However, we show that choosing near-optimal balanced node-separators is sufficient to achieve near-optimal fill. We employ the approximation algorithm of Leighton and Rao ([96, 97]) to find an approximately balanced separator in a graph.

Though the performance guarantee of our algorithm deteriorates in going from a graph of small degree to one without this restriction, it is the first such algorithm that gives any non-trivial performance bound on the quality of the fill.

Theorem 2.5.2 *There is a polynomial-time algorithm that produces an elimination ordering yielding a fill of $O(F^* \frac{3}{4}\sqrt{m}\log^{3.5} n)$, where F^* is the size of the minimum fill, m is the number of non-zero elements in the given $n \times n$ matrix.*

Note that the performance guarantee for fill from the above theorem is never worse than $O(m^{\frac{1}{4}} \log^{3.5} n)$ since the size of the minimum fill F^* must be at least as much as the number of edges m in the graph.

2.5.2 Minimizing the Operation Count

In fact, our algorithm goes further. We show that the ordering generated by our algorithm approximately minimizes not only fill, but also the total number of arithmetic operations required to solve the system of equations using Gaussian elimination. Much of the previous work in elimination ordering has been concerned with minimizing fill, and surprisingly little attention has been given to minimizing the number of operations. The only exceptions we know of, are the works of Hoffman et al. [79] and Lipton et al. [102], who analyzed the operation counts for their nested dissection algorithms. However, their results only apply to specific classes of problems as explained in the discussions above.

Theorem 2.5.3 *The elimination ordering produced by our algorithm approximately minimizes the total number of arithmetic operations. The performance guarantee is $O(d\log^6 n)$ for an $n \times n$ matrix with a maximum of d non-zero elements in each row and column.*

2.5.3 Solving Sparse Systems in Parallel

With the advent of parallel machines, much attention has naturally been focussed on solving sparse systems in parallel. In a typical parallel implementation of the elimination process, multiple variables are eliminated simultaneously in a step. Such variables must then have no dependencies between them, i.e. there must be no edge in the graph between the nodes corresponding to the variables. For an elimination ordering, the minimum number of parallel steps required to eliminate all the variables is called its *height* (see [109]). It is hence of interest to find an elimination ordering of minimum height. This problem is also known to be NP-complete [128]. Many researchers have given heuristics without any guarantees for this problem in the past [84, 98, 106, 110, 101] (see also [48]). We prove that our nested dissection algorithm can guarantee approximately minimum height.

Theorem 2.5.4 *The elimination ordering produced by our algorithm has height within a factor of $O(\log^2 n)$ of optimal.*

Note that our algorithm itself is not parallel, but is to be used to generate an ordering with small height. Such an approach is suitable for problems where the sparsity structure of the matrix is fixed, and the linear system has to be solved for different coefficient values and/or right hand sides. A good elimination ordering can hence be found in serial as a preprocessing step. Our work hence is different from other work addressing the issue of generating the ordering itself in parallel [105, 124, 99, 67, 66, 32].

Characteristic	*Performance Guarantee*
Fill	$O\left(\min(\sqrt{d}\log^4 n, m^{\frac{1}{4}}\log^{3.5} n)\right)$
Operation Count	$O(d\log^6 n)$
Elimination Height	$O(\log^2 n)$

Table 2.2: Performance guarantees for the elimination ordering produced by our algorithm. In the above table, n, m and d respectively denote the number of nodes, edges, and the maximum degree of the graph associated with the coefficient matrix.

Along with minimizing the height, it is desirable to keep both the fill and the operation count small, since they determine the total space and the total work done by the algorithm. Our algorithm is the first one known that approximately minimizes all three quantities *simultaneously*: fill, operation count, and height. By putting Theorems 2.5.1, 2.5.3 and 2.5.4 together, we get the following result.

Theorem 2.5.5 *The elimination ordering produced by our algorithm simultaneously minimizes height to within a $O(\log^2 n)$ factor, fill to within $O(\sqrt{d}\log^4 n)$ factor, and the operation count to within a $O(d\log^6 n)$ factor of the respective optimum quantities. In the guarantee, d denotes the maximum number of non-zero elements in any row or column of the $n \times n$ coefficient matrix.*

Gilbert [63] has conjectured that there is an ordering that simultaneously minimizes height and approximately minimizes fill (to within a constant factor). Our analysis here represents progress towards proving this conjecture. Bodlaender et al.[10] have independently presented essentially the same algorithm as ours for finding an elimination ordering of approximately minimum height. However, they do not analyze the fill and operation count for their ordering.

The performance guarantees for the elimination ordering obtained by our algorithm is given in Table 2.2.

2.5.4 Graph Chordalization

The graph-theoretic interpretation of the elimination process was first given by Rose [136]. He showed that an elimination step can be interpreted as updating the associated graph by adding new edges; the fill-in introduced by eliminating a variable i turns the higher numbered neighbors of node i into a clique. At the termination of the elimination process, the edges of the associated graph thus obtained correspond directly to the fill yielded by the ordering. Rose showed that this updated graph is in fact *chordal*, and that minimizing fill corresponds to finding a minimum-size chordal graph containing the input graph associated with the matrix. A *chordal* graph is a graph in which every cycle of length at least four has a chord, i.e. there is an edge between two non-consecutive nodes in the cycle. Chordal graphs are also sometimes referred to as *triangulated* graphs.

We exploit the characterization of the elimination process given by Rose. We prove that our nested dissection ordering yields a chordal graph of small size with respect to the optimal.

In this section, we will establish a lower bound on the size of the optimum chordal extension of a graph in terms of the separator sizes found by the nested dissection algorithm. In our nested dissection ordering, we employ an approximation algorithm for finding balanced node separators. However, for ease of exposition below, we shall assume that we have a separator algorithm that finds the best balanced node-separator in a graph. We shall later forego this assumption.

Consider the following tree, called the *separator tree*, representing the nested dissection process on a graph; the separator vertices form the root of the tree, and the trees of each of the pieces are built recursively. To distinguish from the nodes of the tree, we shall refer to the nodes of the graph as *vertices*. A tree node hence stands for a separator that may consist of several vertices.

Let $G(V, E)$ be an input graph and G^* be a minimum sized chordal extension of G. And let T be the separator tree given by our nested dissection ordering (employing an optimal balanced separator algorithm). With each node x in the separator tree T, let us associate three quantities S_x, V_x, and G_x. $S_x \subseteq V$ is the set of vertices forming the node x, $V_x \subseteq V$ is the set of vertices belonging to the nodes of the subtree rooted at x, and $G_x \subseteq G$ is the subgraph induced by the vertex set V_x in the original graph G.

We shall derive a lower bound on the size of the optimal chordal extension G^* in terms of the sizes of the separators at any level of the separator tree. By a level of the tree we mean all the nodes in the tree that are at the same distance from the root. Let $x_1, \ldots, x_p$ be the tree nodes at some level of the separator tree. Since $V_{x_1}, \ldots, V_{x_p}$ are disjoint, it follows that the graphs $G^*_{x_1}, \ldots, G^*_{x_p}$ induced by them in G^* are also disjoint. Thus we have

$$\sum_{i=1}^{p} \left|G^*_{x_i}\right| \leq |G^*| \tag{2.3}$$

where $|G|$ refers to the number of edges (size) of G.

We shall now use the following two previously known results.

Fact 2.5.1 *Every node induced subgraph of a chordal graph is chordal.*

Gilbert, Rose, and Edenbrandt [65] showed that every chordal graph has a balanced clique separator, i.e. a set of nodes that along with being a balanced node separator, induces a clique in the chordal graph. Since the number of edges in this clique can be at most the number of edges in the chordal graph, the following theorem follows.

Theorem 2.5.6 ([65]) *Every chordal graph has a $\frac{1}{2}$-balanced clique separator, and hence has a $\frac{1}{2}$-balanced node separator of size at most $\sqrt{2E}$, where E is the number of edges in the chordal graph.*

By Fact 2.5.1, each of the graphs $G^*_{x_i}$ is chordal. Thus by Theorem 2.5.6, we can hence write

$$|V_{x_i}|^2 \leq 2\left|G^*_{x_i}\right| \tag{2.4}$$

On rewriting (2.3) using this observation, we have the following lemma.

Lemma 2.5.1 *The size of the optimal chordal extension is at least one-half the largest sum of the squares of the sizes of the separators at any level of the nested dissection separator tree.*

One of the main results of the thesis is to show that the nested dissection algorithm in fact yields a chordal graph whose size is close to the lower bound given above.

Theorem 2.5.7 *The size of the optimal chordal graph is at least one-half the sum of the squares of the sizes of the separators at any level of the nested dissection separator tree, and $O(\sqrt{d}\log^4 n)$ times this value, where d is the maximum degree of the graph, and n is the number of nodes.*

In employing an approximation algorithm for finding balanced node separators that has a factor f performance guarantee, we prove that the size of the chordal graph does not increase by more than a $O(f^2)$ factor compared to that obtained by using the optimal balanced node separators. We employ a separator algorithm with an $O(\log n)$-factor performance guarantee, and obtain the following result.

Theorem 2.5.8 *There is a polynomial-time algorithm that generates a nearly optimal chordal extension of an input graph. The size of the chordal graph is $O(\min(|G^*|\sqrt{d}\log^4 n, |G^*|^{\frac{3}{4}}\sqrt{m}\log^{3.5} n))$, where $|G^*|$ is the size of the optimal chordal extension of an input graph of n nodes, m edges, and maximum degree d.*

2.5.5 Experimental Results

Though our nested dissection algorithm predicts good fill, we wanted to check if it is really so in practice. In this thesis, we give the results of our experiments. We implemented the algorithm for finding approximate balanced node-separators as described by Leighton and Rao (see [96]). Their algorithm consists of repeatedly applying the algorithm for finding an approximately sparsest node separator in a graph. The algorithm for finding the node separator consists of two phases. In the first phase, a uniform concurrent flow problem is solved, and in the second phase, the solution of the concurrent flow problem is rounded to produce a node separator in the graph. The first phase requires the solution of a linear program, the time complexity of which, though polynomial, was unacceptable for our purposes. We hence turned to an approximation algorithm for solving the uniform concurrent flow problem [91], which was implemented by Sarah Kang and Philip Klein.

Our experiments were done on a series of matrices obtained from the Harwell-Boeing test suite [34]. This test suite is a compilation of matrices that arose in different branches of engineering. It has been used in benchmark studies by many researchers [35, 107, 101, 108, 109], and is considered as a standard test suite.

We compared the quality of our results to two other publicly available codes. These two codes use two different well-known heuristics. The first is the minimum-degree[3] heuristic code by Joseph Liu [105]. His code is acknowledged as one of the best codes for finding an elimination ordering. The results reported use the latest

[3]No relation to the minimum-degree R-join problem.

version of the code that we obtained from him in July '91. The second code is the nested dissection heuristic that is implemented in SPARSPAK [59], also a widely accepted code for the purpose.

I am happy to report that the quality of the fill obtained by our nested dissection algorithm compared favorably with that obtained by Joseph Liu's minimum-degree code. For some problems, our ordering yielded slightly lower fill, and for others, slightly higher. For the most part, there was no marked difference between the quality of the two heuristics with regard to fill. The operation count for the minimum degree ordering was somewhat better than our nested dissection ordering. Our ordering, however, yielded fill and operation count that was consistently better than the nested dissection algorithm implemented in SPARSPAK. The height of our ordering was generally better than that obtained by either of the other two codes. The experimental results are presented in greater detail in Chapter 5.

The greatest drawback of our algorithm is that it runs much slower than other heuristics like the minimum-degree heuristic. However, our experiments show that the nested dissection method produces orderings for serial machines of quality competitive with, and for parallel machines of quality superior than the minimum degree ordering. For the nested dissection algorithm to become popular in practice, good separator algorithms must be devised. In its present implementation, the algorithm is useful in situations where multiple systems of linear equations have to be solved with the same sparsity structure, and hence the cost of finding a good ordering is amortized over several problems. Such is the case when solving linear programs using interior-point methods. The principal cost of an iteration then is due to solving a symmetric positive-definite system of linear equations. The particular system changes at each iteration, but the graph associated with the system remains the same.

2.6 Rest of the thesis

Chapter 3 deals with the issues relating to the problem of minimum edge-cost R-join. It proves the results claimed in Section 2.3. Chapter 4 is devoted to the minimum-degree R-join problem and its related combinatorial quantities. The results of Section 2.4 are proved in this chapter. Chapter 5 considers the elimination ordering problems in detail and establishes the claims made for these problems. We end the thesis with open issues in Chapter 6.

Chapter 3

Minimum Edge-cost R-join

In Section 2.2, we proposed the requirement-join formalism for a network design problem. We showed that finding a requirement-join of minimum total edge cost has applications in diverse industries, and is one of the important problems in the field of network design. We claimed in Section 2.3 that we have the first approximation algorithm for the problem and that our algorithm is based on a new approximate min-max relation between the minimum cost requirement-join and the maximum c-packing of cuts. We justify these claims in this chapter.

The basis of our work is the new approximate min-max equality between the minimum size of an R-join (unweighted case) and the maximum-size $\frac{1}{2}$-packing of R-cuts. We prove this relationship and in the process give approximation algorithms for the two combinatorial problems. We then build on this work to prove a similar relationship for the weighted case of the problem. We further extend the algorithms to apply to the weighted cases of the R-join and R-multijoin problems. We also discuss the uncrossed property of the requirement cuts obtained by our algorithm, the relevance of which will be apparent in in the next chapter in the discussion for the minimum-degree R-join problem.

3.1 Our Algorithm: Unweighted Case

In the unweighted case of the R-join problem, we assume that each of the edges of the original graph has unit cost. Hence the cost of an R-join equals the number of edges in the R-join. The number of edges in an R-join will also be referred to as its *size*. The algorithm for the unweighted case is the basis for our algorithms for the weighted cases of the R-join and R-multijoin problems.

We describe our algorithm for finding an R-join of approximately minimum-size. Our algorithm exploits the relationship between the minimum-size R-join and the maximum-size $\frac{1}{2}$-packing of R-cuts. The algorithm simultaneously finds a $\frac{1}{2}$-packing of R-cuts and an R-join such that the size of the R-join is small compared to the size of the $\frac{1}{2}$-packing.

3.1.1 Preprocessing

The first step of the algorithm is to transform the original graph G into a bipartite graph G_0 by replacing each edge (u, v) of G with two edges (u, x) and (x, v) in series, where x is a new node. The resulting graph G has the following properties:

- Any minimal network design in G corresponds to a network design in G_0 of half the size.
- Any packing of edge-disjoint requirement cuts in G_0 corresponds to a $\frac{1}{2}$-packing of requirement cuts in G of the same size.

Consequently, in order to prove Theorem 2.3.4 for G, it is sufficient to show the following for G_0.

Theorem 3.1.1 *We can find an R-join N and a 1-packing of requirement cuts $A_1, \ldots, A_\beta$ such that $N \leq 2(1 - 1/k)\beta$ in G_0, where k is the total number of sites.*

Our proof of Theorem 3.1.1 is algorithmic. We shall construct a set of edge-disjoint requirement cuts (1-packing) and a requirement join such that the size of the requirement join is approximately equal to twice the number of edge-disjoint cuts. From Lemma 2.3.2 and the discussion above, it follows that a maximum 1-packing of R-cuts in G_0 is a lower bound on the minimum size of an R-join. By constructing an instance of a 1-packing of R-cuts and an instance of an R-join, such that their values are approximately equal, we show that the instances are in fact approximately optimal and that the approximate min-max relation in Theorem 3.1.1 holds.

Though our algorithm *simultaneously* constructs a 1-packing and an R-join, for the sake of exposition, we shall first give the algorithm just for finding the 1-packing. We shall then augment this algorithm with the steps for constructing the network.

3.1.2 Constructing a 1-packing

We have a greedy approach to finding a large collection of edge-disjoint requirement cuts in the bipartite graph G_0. Our algorithm progresses over a series of *timesteps*. At each timestep, we find new edge-disjoint requirement cuts, starting from those found in the previous timestep, and we add these R-cuts to our collection. By construction we shall ensure that these new R-cuts are edge-disjoint from all the ones already in our collection.

At any timestep, we have a set of node-disjoint *active* trees. These active trees are such that the set of boundary edges of each active tree forms a requirement cut. Moreover, these boundary edges are disjoint. The algorithm starts at timestep zero, and has as many active trees as the number of sites; each active tree contains a distinct site. Since the sites are at even distances in the bipartite graph, it follows that the boundary edges of each of these active trees are disjoint. Moreover, the boundary edges of each such active tree separates the requirement site pair whose one endpoint forms the tree.

The algorithm starts out at timestep zero with the original graph G_0 but modifies the graph as it progresses. We shall denote the graph at the end of timestep t by G_t. In general, the graph G_t is a multigraph obtained from the graph G_{t-1} by contracting certain sets of connected nodes of G_{t-1} into single nodes in G_t. We shall refer to such nodes as *supernodes*. Thus each supernode can be viewed as containing a certain node-induced subgraph of the original graph, and has been formed through a series of contractions over the timesteps. By construction, we shall ensure that a supernode never separates any requirement pair, i.e. if it contains one site of a requirement pair, then it must also contain the other.

We now describe the algorithm at each timestep in detail. At a timestep, we add the boundary edges of each active tree in G_t to our collection of edge-disjoint cuts. We shall show later that the distance between the boundary nodes of the trees is always even and hence the cuts are disjoint. Each active tree is then grown by one breadth-first-search level to include its boundary edges. If after growing, the trees remain node-disjoint, then we consider these trees active even for the next timestep, and proceed to the next timestep. However, in general multiple trees might have common boundary nodes, and thus are no longer node-disjoint. In such a case, we merge these trees, and thus maintain node-disjointness of the active trees for the next timestep. The process is described in detail below.

We shall say that a tree T_1 is *colliding* with another tree T_2 if either the two trees have a node in common, or else there is another tree T_3 such that T_1 and T_2 are each colliding with T_3. For each maximal set of colliding trees, we union their nodes to form a single tree. These sets of colliding trees can be easily found, and we will present the details later.

For each tree formed by merging a set of colliding active trees, we check if there exists a requirement pair with one site within the tree and one outside. If such a requirement pair exists, then we consider this merged tree as active for the next timestep. If no such requirement pair exists, then the tree is considered inactive, and the subgraph induced in G_t by the nodes of the inactive tree, is contracted into a supernode. All trees that do not collide with any other trees also remain active for the next timestep. We then proceed to the next timestep if there is at least one active tree remaining.

That completes the description of the 1-packing algorithm except for the details of merging the colliding trees which we give below.

Merging colliding trees

We now describe the process of finding the maximal sets of merging trees at a given timestep. Let $\mathcal{T}$ be the set of trees active at the beginning of a timestep. If $\mathcal{T}$ is non-empty, we consider some $T \in \mathcal{T}$. We remove T from $\mathcal{T}$, and merge the maximal set of colliding trees containing T. Each of the merging trees is also removed from the set of active trees in the process. We repeat the above process until there are no active trees remaining.

The process of merging the maximal set of colliding trees is simple. We repeatedly look for a tree S that has a node in common with T. We merge S into T, and delete S from the set of active trees. If we cannot find any such tree S to merge into T, then T is a maximal merged tree.

Proof of correctness

We shall now show that the requirement cuts output by the algorithm are in fact edge-disjoint. We shall also show later that the size of this 1-packing is approximately optimal.

We call a node an *original* node if it appears in the original graph. By the *sites of a tree* T in a contracted graph we shall refer only to those nodes of T that are sites and hence by definition also original nodes. We hence exclude from this definition all the sites that are strictly enclosed within a supernode in T. Since each of the active trees in the above algorithm grows by one level at each timestep, we have the following proposition.

Propostion 3.1.1 *After t timesteps, each node in the boundary of an active tree is at distance t in G_t from some site internal to the tree.*

In the initial bipartite graph, all the sites are on the same side of the bipartition. We show that this property continues to hold throughout the algorithm.

Lemma 3.1.1 *After t timesteps, the graph G_t is still bipartite, with all sites on the same side of the bipartition.*

Proof: We prove it by induction on t. The basis $t = 0$ is trivial. We must show that the bipartition property described in the lemma is preserved by contractions. Suppose that G_{t-1} obeys the property, and that after t timesteps, some tree T has just become inactive and is about to be contracted. By Proposition 3.1.1, all the nodes in the boundary of T have distance t from some site. Hence they all belong in the same side of G_{t-1}'s bipartition. It follows that after the nodes of T are contracted to a single node, the bipartition property still holds. □

By Lemma 3.1.1, each of the cuts added to the collection in a timestep are edge-disjoint. By construction, each of the edges that occurs in a cut at a timestep becomes an internal edge to some tree by the end of the timestep and hence cannot appear in further cuts. Thus we obtain the following corollary.

Corollary 3.1.1 *The cuts in the packing found by the algorithm are edge-disjoint.*

3.1.3 Constructing an R-join

We will now describe the algorithm for constructing the R-join for a given set of requirement pairs. The algorithm works hand in hand with that for finding a packing of R-cuts described above.

Some definitions

Before we proceed, we introduce some terminology to help us relate various contracted graphs to each other and to the original graph. We say that a tree T and a supernode v *correspond* to each other if T was contracted to

form the supernode v. We say a node v is a *parent* of v' if v' is a node of the tree corresponding to v. Note that each node has at most one parent since each node belongs to at most one tree. Recall that if a node belongs to two colliding trees, then the trees are merged into a single tree, which becomes the parent of the node.

We say v *encloses* v' if either $v = v'$ or else v is a parent of some node v'' that encloses v'. That is, v encloses v' if by some series of contractions, v' was identified with other nodes to form v. We define the *nesting level* of an original node as zero, and that of a supernode (tree) as one more than the maximum nesting level of any of nodes belonging to the supernode (tree). In contracting a graph G' to G'', each edge (u', v') in G' has a natural *image* in G''; the image of (u', v') in G' is the edge (u'', v'') in G'' where u'' encloses u' and v'' encloses v'. Note that multiple edges of G' may have the same image in G''. In a similar manner an edge (u'', v'') in G'' has (u', v') as its image in G'. This image is not necessarily unique as discussed above.

We denote the (inactive) tree corresponding to a supernode v by T^v, and the supernode corresponding to an inactive tree T by s^T. By the *age* of an inactive tree, we denote the timestep at which the tree was rendered inactive. The age of an active tree is the current timestep. The *age* of a supernode simply refers to the age of the tree corresponding to the supernode.

An overview of the algorithm

For a tree T, we shall refer to all the sites whose mates are not in the tree as *active* sites. Recall that each active tree must contain at least one active site. Our algorithm maintains the invariant that for every active tree at a timestep, all its active sites are connected by a network of edges in the original graph. We call this invariant the *network invariant.*

We shall maintain the network invariant inductively over the timesteps. We shall do so by adding edges to the network so that the invariant holds for each of the merged trees (active or inactive) during a timestep. This then also implies that the invariant holds for the tree corresponding to a supernode.

Before we continue with the algorithm, we would like to convince the reader that if the algorithm maintains the network invariant, then the edges of this network form a valid requirement join when the algorithm terminates. The argument is simple: Consider a requirement q with endpoints u and v. Let the parents of u and v be s_u and s_v respectively. By construction, s_u must be the same supernode as s_v, or else the supernode s_u separates u and v, and hence the requirement q, which is impossible. Moreover, by the network invariant for the tree associated with s_u, its sites are connected by the network. Hence the desired path between the two sites u and v exists in the final network.

Thus we only need to specify how we maintain the network invariant inductively over the timesteps. The invariant clearly holds for the 0th timestep, since each tree contains only a single site which is trivially connected to itself without requiring any edges of the network. We shall assume that the network invariant holds at the beginning of a timestep. While merging two trees, we shall specify which edges to add to the network so as to maintain the network invariant for the resulting merged tree. Since the invariant is made to hold for each pairwise merge, it follows that the invariant holds also for each of the merged trees formed at the timestep.

Before we describe the algorithm, we need a few relevant definitions for relating a network design to different contracted graphs. We shall denote by N^T (respectively N^v) the networks constructed so far for the tree T (respectively the tree T^v corresponding to the supernode v). With any subgraph N of the original graph G_0, we can associate in a natural way a subgraph N_t in G_t, namely that obtained by applying the same node contractions on N as required to contract G_0 to G_t. We shall call N_t the *image* of N in G_t. Thus every network has a corresponding contracted network for every timestep.

R-join algorithm

We are now ready to describe our algorithm for maintaining the network invariant for the merged tree assuming that it holds for each of the two merging trees. Let the two trees being merged at a timestep t be S and T. By construction, there is a node v such that v is common to both S and T. If v is an original node, then we establish two paths, one between v and the network of S, and the other between v and the network of T. In doing so, we have ensured that the active sites belonging to $S \cup T$ are connected, and the network invariant is maintained. If v is a supernode, then instead of establishing a new path between the networks of S and T, we shall connect both of them to the previously established network of v. This will ensure connection between the networks of S and T.

We now describe how to establish a connection between the networks of v and S. Establishing the connection between the networks of v and T is analogous. Recall that by Proposition 3.1.1 there is a path in G_t of length at most t from v to a site in S. Let this site be w and the path be $P_r = \{v = v_0, v_1, \ldots, v_t = w\}$. We shall call this path the *radial path*, and refer to the edges on this path as *radial edges*. Let $P_s = \{v = v_0, v_1, \ldots, v_p\}$ be the smallest subpath of P_r such that the network of v_p is connected to that of S. We shall refer to this path as a *skeletal path*. This path may contain supernodes. Our aim is to expand the skeletal path P_s into a set of edges in the original graph G_0 such that the networks of v and S are connected.

Let (v_i^r, v_{i+1}^r) be an image in G_0 of the radial edge (v_i, v_{i+1}). Note that v_i^r and v_{i+1}^r are enclosed in v_i and v_{i+1} respectively. We add the edge (u_i^r, v_{i+1}^r) to the network. We then establish connections between v_i^r and N^{v_i}, and between v_{i+1}^r and $N^{v_{i+1}}$ as described later. This ensures that each radial edge (v_i, v_{i+1}) in the skeletal path is converted into a network of edges connecting N^{v_i} and $N^{v_{i+1}}$ for every $0 \leq i \leq p-1$. This establishes the desired connection between the networks of the endpoints of the skeletal path.

The procedure of how we establish a connection between the network of a node u and that of a node w enclosing it still remains to be specified. This procedure will be used to establish connection between the original node v_i^r (respectively v_{i+1}^r) and the network of node v_i (respectively v_{i+1}). The procedure is recursive. We establish a connection between the network of u and that of its parent v using the method of skeletal path described above. We then recursively establish a connection between the networks of v and w.

Below we give the pseudocode for connecting two merging trees S and T. The procedure is called CONNECTTREES.

Pseudocode for connecting the networks of S and T

CONNECTTREES(S, T)

Assumption: S and T are two colliding trees.

Goal: To connect the networks of S and T.

T1 Let v be a node common to S and T

T2 call CONNECTTONETWORK(v, S)

T3 call CONNECTTONETWORK(v, T)

end

CONNECTTONETWORK(v, T)

Assumption: v is a node of the tree T.

Goal: To connect the network of v to the network of T.

C1 Find a radial path $P_r = \{v = v_0, \ldots, v_l = w\}$
where w is an site active in T at the time v was added to T.

C2 Find the skeletal path $P_s \subseteq P_r$. Let $P_s = \{v = v_0, \ldots, v_p\}$.

C3 For every edge $e = (v_i, v_{i+1}) \in P$

C4 Let (v_i^r, v_{i+1}^r) be an image of (v_i, v_{i+1}) in G_0

C5 Add the edge (v_i^r, v_{i+1}^r) to the network

C6 if $v_i^r \neq v_i$ then call EXPANDPATH(v_i^r, v_i)

C7 if $v_{i+1}^r \neq v_{i+1}$ then call EXPANDPATH(v_{i+1}^r, v_{i+1})

end

EXPANDPATH(u, w)

Assumption: w is an ancestor of u.

Goal: Establish a path connecting the network of u to that of its ancestor w.

E1 If w is the parent of u

E2 CONNECTTONETWORK(u, T_w)

E3 else w encloses u

E4 Let v be the parent of u

E5 call CONNECTTONETWORK(u, T_v)

E6 call EXPANDPATH(v, w)

end

The correctness of the algorithm follows simply from the construction and the discussion above.

3.1.4 Performance Guarantees

We will prove in this section that the size of the network design given by the algorithm is at most twice the size of the 1-packing found. We shall device a scheme for charging the edges of the network to trees formed during the course of the algorithm.

Each tree active at a timestep earns two dollars, for contributing one edge-disjoint requirement cut to the 1-packing. The total number of dollars earned by the trees over all the timesteps is then exactly twice the size of the 1-packing. When two trees merged, their leftover earnings are also merged and assigned to the merged tree. We assume that each edge costs one dollar, and that the edges of the network are paid for by the trees (active or inactive). We shall show that no tree pays more dollars than it earns. It follows then that the size of the network is at most twice the size of the packing.

We shall device a simple scheme for charging the cost of the network edges to different trees. Every time we merge two trees, we shall specify how to charge for the edges added to the network. With every tree T, we shall assign a *funding tree* $f(T)$, the tree that will be charged for all the edges added to the network on calling ConnectToNetwork(v, T), not counting the its recursive calls. That is, if ConnectToNetwork recursively calls itself with arguments (v', T'), then the edges added by this recursive call are charged to the funding tree of T' and not to that of T. We shall define the funding tree $f(T)$ of T to be the highest ancestor of T with the property that the networks of T and $f(T)$ are connected.

It will be useful to view the funding tree information at any timestep t as a *connectivity forest* constructed as follows. Each of the original nodes of the graph and the supernodes formed during the course of the algorithm is a node in the forest. Let v be the parent of a node u. There is an edge from u to v in the connectivity forest if u's network is connected to that of v.

The above construction defines a forest since every node can have at most one outgoing edge. For any node u belonging to a tree T, the root r of T is the highest ancestor of u such that the networks of u and r are connected. Hence the tree associated with r is the funding tree for the tree associated with u.

By our algorithm, the only way the network of a node u can be connected to that of a node not enclosed in u is if u is on a skeletal path. If such is the case, then the skeletal path must end in the network of u's parent. Once the skeletal path has been expanded into a network of edges, u's network is connected to that of its parent. This implies that if u has ever been on a skeletal path, then the funding tree of u is the same as that of its parent. Alternatively, if u has never been on a skeletal path, then its network is not connected to that of any of its ancestors and hence the funding tree of u is itself. This characterization leads to the following lemma.

Lemma 3.1.2 *For a tree S enclosing T, if S is not the funding tree for T, then S is also not the funding tree for any tree enclosed by T.*

If a node v is in a directed path from some node u to w in the connectivity forest, then the network of v must also be connected to that of w. Moreover, by construction of the connectivity tree, v must be an ancestor of u. We then get the following simple lemma.

Lemma 3.1.3 *If N^u is connected to N^w for some w enclosing u, then N^v is connected to N^w for every v such that w encloses v and v encloses u.*

We can now derive the following lemma.

Lemma 3.1.4 *If N^u is connected to N^w, for some w enclosing u, then the number of edges added to the network by* EXPANDPATH(u, w) *is* 0.

Proof: Unfolding the recursion for EXPANDPATH, we observe that the number of edges added by the call equals

$$\sum_v \text{the number of edges added by } \textsc{ConnectToNetwork}(v, T_{\text{parent}(v)})$$

where the sum is over all nodes v such that w encloses v and v encloses u. By Lemma 3.1.3, it follows that the network of each such node v is connected to that of its parent. This implies that the skeletal path is null for each of the calls to CONNECTTONETWORK and hence the total number of edges added by these calls is zero. Thus the lemma follows. □

We require one more lemma before we can proceed further.

Lemma 3.1.5 *Let $P_r = \{v_0, \ldots, v_l\}$ be a radial path of which $P_s = \{v_0, \ldots, v_p\}$ is a subpath. Then the age of v_p is at most $l - p$.*

Proof: The distance from v_l to v_p is $l - p$. Since P is a radial path, v_l must be an active site at timestep l. Since v_l must be active for all timesteps until l, it follows that v_p must belong to the active tree containing v_l after $l - p$ timesteps. This implies that v_p must have been formed at a timestep earlier than $l - p$. Thus the age of v_p must be less than $l - p$. □

With the above observations about the funding tree structure, we now proceed to show that through this funding scheme, no tree gets charged more than the amount it earns.

Lemma 3.1.6 *In a call to* CONNECTTONETWORK(a, T), *the cost charged to $f(T)$ is at most age(T).*

Proof: We shall prove the above statement by induction on the nesting level of T. If the nesting level is 0, then the statement holds since a is the only node in T. Assuming that the statement holds for a nesting level of $k - 1$, we shall prove that it also holds for k.

Let $P_r = \{v = a_0, a_1, \ldots, a_l = s\}$ be a radial path from v to an active site s in T, of which let $P_s = \{v = a_0, a_1, \ldots, a_p\}, p \leq l$, be the skeletal path. For each edge in the skeletal path, there is an edge added to the network in step C5 of the algorithm. Thus step C5 adds p edges to the network. Let us now account for the edges added by the steps C6 and C7 of the algorithm.

By construction of the skeletal path, the networks of each of the nodes except a_p is not connected to N^T. Hence by Lemma 3.1.2 their funding trees and those of trees enclosed by them are different from T. It then

follows that the cost charged to T by steps C6 and C7 for all edges in P except for the edge (a_{p-1}, a_p) is 0. By the same argument, the cost of step C6 for (a_{p-1}, a_p) charged to $f(T)$ is zero. We hence only need to account for the edges charged in step C7 for the edge (a_{p-1}, a_p). We will prove that this cost is at most the age of a_p. Before we do so, we show that this is sufficient to prove the lemma.

The age of a_p is at most $l - p$ by Lemma 3.1.5. The cost of the call charged to T is hence is at most p from step C5, zero from step C6, and at most $l - p$ from step C7. Thus the lemma follows.

Let (a^r_{p-1}, a^r_p) be the image of (a_{p-1}, a_p) in G_0. It remains to be shown that the cost charged in the call EXPANDPATH(a^r_p, a_p) is at most $age(a_p)$. In order to use the same variables as that used in the pseudocode for EXPANDPATH, let us use u to refer to a^r_p, and w to refer to a_p. There are two possibilities.

1) w is the parent of u. Then the cost of CONNECTTONETWORK equals that for statement E2, which by the induction hypothesis is at most $age(T_w)$ which is $age(v_p)$.

2) u is enclosed in w. Let v be the parent of u. There are then two possibilities.

   a) N^v is connected to N^w. Since v is enclosed in w, its level is less than that of w. Hence by the induction hypothesis, the cost of the statement E5 charged to T_w is at most $age(v)$, which is less than $age(w)$. Moreover, by Lemma 3.1.4, the cost of statement E6 is then zero since N^v is connected to N^w, and the lemma follows.

   b) N^v is not connected to N^w. This implies that the funding tree of v is different to that for w. Hence the cost of statement E5 charged to T_w is zero. The cost is hence that of statement E6, which is a recursive call to EXPANDPATH. This recursive call must finally must bottom out either in case 1 or case 2a and thus prove the lemma.

□

Let us now define the *savings* of a tree T to be the difference between the earnings for the tree and the cost of the network charged to T. We claim the following lemma.

Lemma 3.1.7 *Each active tree at timestep t has savings of at least $2t$ dollars.*

Proof: We shall prove the lemma inductively on the timesteps. The lemma is trivially true at timestep 0, when the savings of each tree is zero. Let us assume that the lemma holds for the active trees at the end of timestep $t - 1$. Since each of the trees earns an additional two dollars at timestep t, it follows that the lemma holds also at the beginning of timestep t. We shall now show that it holds at the end of timestep t.

If a tree does not encounter a merge during the timestep, then the lemma trivially holds for the tree. We shall show that it holds even for the tree formed by merging two trees. That will show that it holds also for a tree formed by a series of merges.

Let the merging trees be S and T. By the induction hypothesis, we assume that each of S and T has a savings of $2t$ dollars each. In merging the two trees, CONNECTTONETWORK is called once for each of the trees, and hence the trees are charged at most t dollars each by Lemma 3.1.6. Thus each of the trees still has a savings of t dollars after merging the trees. The lemma then holds for the merged tree $S \cup T$ since it gets the savings of both S and T. □

The above lemma shows that each of the active trees has some savings. Since the lemma holds until a tree becomes inactive, it follows that at the time a supernode is formed, it has a savings at least twice its age. The question then arises as to how many edges of the network are charged to a supernode v after it is formed. Edges can be charged to a supernode v in only two ways.

1) v is the common boundary node used in merging two trees $T1$ and $T2$. In such a case, there are two calls made to CONNECTTONETWORK, once for each merging tree to connect v's network to that of the tree. Moreover, immediately after the merge, the funding tree of v is now $T_1 \cup T_2$, since the network of v is connected to that of $T_1 \cup T_2$. Hence no further edges can be charged to v.

2) v is an internal node on a skeletal path P ending in the network of a tree T. In this case, the calls for connecting to v's network are made once each by its two incident radial edges in the skeletal path P. However, once the skeletal path has been expanded into a network of edges, the network of v is connected to that of its parent T. Once again no further edges can then be charged to v.

Thus in either of the above two cases, a supernode is charged at most twice with calls to CONNECTTONETWORK. Thus by Lemma 3.1.6 we have the following claim.

Lemma 3.1.8 *The total cost of the network charged to the tree associated with a supernode after the supernode is formed is at most twice the age of the supernode.*

However, by the discussion following Lemma 3.1.7, each supernode has savings of at least twice its age at the time it is formed. It then follows from the above lemma that no supernode is charged more than its savings after it is formed. Thus no tree (active or inactive) is charged more than its earnings, and it follows that the total cost of the network edges is at most the total earnings over all the trees, which equals twice the number of disjoint cuts found by the algorithm.

In fact we will prove a slightly stronger result. We will show that the size of the network is at most $2(1 - \frac{1}{k})$ times the size of the 1-packing found, where k is the number of distinct sites. Consider the connectivity forest at the termination of the algorithm. Since there are no active trees, the roots of the trees in this forest must be supernodes. By definition, the networks of these root supernodes are not connected to those of their parents. It then follows that each such supernode has a savings of at least twice its age. We shall show that at least one such supernode has a large savings to justify the improvement in the approximation factor.

Let us associate with each root r the cuts assigned to any of the nodes belonging to the tree of which r is the root. Each cut in the packing is thus associated with exactly one root. For a root v, let $Sites(v)$ denote the set of sites that are connected by the network N^v. Note that this set is disjoint from that of any other root. We can then derive the following lemma.

Lemma 3.1.9 *For a supernode v, $age(v)$ is at least $|Cuts(v)| / |Sites(v)|$.*

Proof: At a given timestep, each of the active trees adds one disjoint cut to the packing. However, each active tree must contain at least one active site. Hence the number of cuts added at a timestep is at most the number of active sites. Each of the sites belonging to $Sites(v)$ is inactive by the timestep $age(v)$. Hence the number of cuts that could have been added to $Cuts(v)$ is at most $age(v)$ times $|Sites(v)|$. Thus we get

$$age(v) \geq \frac{|Cuts(v)|}{|Sites(v)|}$$

□

From the discussion earlier, each root supernode has a savings at least twice its age. Thus we have a better bound on the size of the R-join:

$$|\text{R-join}| \leq 2\left(|\text{1-packing}| - \sum_{\text{root } v} age(v)\right) \tag{3.1}$$

By Lemma 3.1.9, we have

$$\begin{aligned} age(v) &\geq \frac{|Cuts(v)|}{|Sites(v)|} \\ &\geq \frac{|Cuts(v)|}{k} \end{aligned} \tag{3.2}$$

where k is the number of distinct sites in the graph. Since $\sum_{\text{root } v} |Cuts(v)|$ equals the size of the packing found by the algorithm, we can rewrite (3.1) as

$$|\text{R-join}| \leq 2\left(1 - \frac{1}{k}\right)|\text{1-packing}| \tag{3.3}$$

We have thus related the value of an instance of 1-packing to an instance of R-join in the bipartite graph G_0 and proved Theorem 3.1.1. This implies Theorem 2.3.4 by our discussion in Section 3.1.1. Following our discussion in Section 1.4, the 1-packing and the size of the network formed by our algorithm are approximately optimal. Barring the analysis of the running time of our algorithm, this also proves Theorems 2.3.2 and 2.3.6. We shall discuss the implementation of our algorithm in the next section.

3.1.5 Implementation and Time Complexity

Constructing a $\frac{1}{2}$-packing of R-cuts

The implementation of our algorithm for finding an approximately optimal $\frac{1}{2}$-packing of requirement cuts is straightforward. As input we are given a doubly linked list of edges of each node in the input graph G. We construct the auxiliary bipartite graph G_0 as described earlier, and implement the algorithm for finding a set of edge-disjoint requirement cuts in G_0.

We do not explicitly contract the nodes of G_0 to obtain the graph G_t at a timestep t. The active trees and supernodes in G_t are simply represented by the set of original nodes enclosed by them. For a tree T (active or inactive), let $V(T)$ represent its enclosed set of original nodes. $V(T)$ also uniquely identifies the set of boundary edges $B(T)$ for T; an edge of G_0 belongs to $B(T)$ if and only if it has exactly one endpoint in $V(T)$. The set of active sites $A(T)$ for a tree T can also be identified from $V(T)$; a site $s \in V(T)$ belongs to $A(T)$ if and only if its mate does not belong to $V(T)$. At any timestep, for each active tree T, the set of boundary edges $B(T)$ is added to the 1-packing. The algorithm starts out with each original node in a separate tree. The tree is active if the original node is a site, else we view the tree as inactive and the original node as a supernode.

Recall that at any timestep, we need to achieve the following steps.

1) Grow each active tree by one breadth-first level and find colliding trees.

2) Merge the colliding trees into a single tree.

3) Check if the resulting tree is inactive.

We shall implement the set $V(T)$ for a tree (or a supernode) T as a union-find data structure [154]. Note that at any timestep t, each original node of G_0 belongs either to a supernode in G_t, or belongs to an active tree in G_t. Thus by performing a find operation on an original node u, we can obtain the tree T containing u in $V(T)$. We shall use this basic operation in steps 1 and 2. Two trees T_1 and T_2 are merged by performing a union operation on the sets $V(T_1)$ and $V(T_2)$.

The boundary edges $B(T)$ for a tree T are maintained as a doubly linked list. A tree T is grown and merged with its colliding trees in the following manner. For each edge $(u, w) \in B(T)$, where $u \in V(T)$, and $w \notin V(T)$, we find the tree T' containing w. If $T = T'$, we delete the edge (u, w) from $B(T)$. If $T \neq T'$, we merge the trees T and T' by performing the union operation on the sets $V(T)$ and $V(T')$. Note that T' can either be an active tree, or an inactive tree associated with a supernode. We update the set of boundary edges for the merged tree by deleting the edge (u, w) from $B(T)$ and $B(T')$ and then merging $B(T)$ and $B(T')$.

The set of active sites $A(T)$ for a tree T is maintained as a doubly linked list. On merging two trees T_1 and T_2, we update the active site information for the tree $T_1 \cup T_2$ to be the symmetric difference of the sets $A(T_1)$ and $A(T_2)$. $A(T_1 \cup T_2)$ is constructed in the following manner. Without loss of generality, let us assume that the size of $A(T_1)$ is smaller than the size of $A(T_2)$. For each site $s \in A(T_1)$, we check if the mate of s belongs

to $A(T_2)$. If it does, then we delete s from $A(T_1)$ and s's mate from $A(T_2)$. If s's mate does not belong to $A(T_2)$, we delete s from $A(T_1)$ and add it to $A(T_2)$. $A(T_2)$ is now the set of active sites for $T_1 \cup T_2$.

We now analyze the running time of the algorithm. Let us first analyze the complexity of maintaining the set of original nodes $V(T)$ and the set of boundary edges $B(T)$ for each tree T. Over the course of the algorithm, we need to do a constant number of operations and a maximum of one find and one union operation for each edge. This requires $O(m\alpha(m,n))$, where m is the number of edges in the graph, and α is the inverse of Ackerman's function. Similarly, to maintain the set of all sites $S(T)$ enclosed in a tree T, we do a maximum of m set unions over the course of the algorithm, and the running time is $O(m\alpha(m,n))$.

We claim that over the course of the algorithm, the number of steps required to maintain the list of active sites is at most $O(n \log n)$. Note that the time complexity of merging two trees is proportional to the size of the smaller of the active sets for the merging trees. By charging the cost of the merge to the sites of the tree that has the smaller number of enclosed sites (active or inactive), it can be shown that no site is charged more than $\log_2 n$ times. Hence the running time for this part of the algorithm is $O(n \log n)$.

Thus the total running time for the algorithm is $O(m\alpha(m,n) + n \log n)$. This proves the running time in Theorem 2.3.6.

Constructing the network

In a contrast from the $\frac{1}{2}$-packing algorithm, we explicitly maintain the contracted graph G_t at a timestep t. Recall that we need to support the following operations.

1) Grow each active tree by a breadth-first level in G_t.

2) If two trees collide, then establish connections between their networks.

Each active tree T maintains a list of its nodes belonging to G_t. We shall denote this list by $N(T)$. Each node v in G_t also carries its parent information. If a node $v \in G_t$ belongs to a tree T, then $parent(v) = T$. If v does not belong to any active tree, then we set $parent(v)$ to be v itself. The set of boundary edges $B(T)$ of a tree T is represented as a set of original edges of G_0 as described earlier for the $\frac{1}{2}$-packing algorithm. As in the $\frac{1}{2}$-packing algorithm, associated with each node of G_t is the set of original nodes $V(T)$ enclosed by its associated tree T. Thus as before, given an original node v^r of G_0, we can find the node $v \in G_t$ enclosing v^r by a find operation.

To grow a tree T by a breadth-first level, we consider each edge $(u^r, v^r) \in B(T)$, where $u^r \in V(T)$ and $v^r \notin V(T)$. We find the nodes u and v in G_t enclosing u^r and v^r respectively using a find operation as described above. Note that v is a node of G_t encountered by the edge (u,v) on growing T by a level. There are two possibilities for v.

a) v does not belong to an active tree. In which case we add v to $N(T)$, and make T the parent of v. We set (u,v) to be the *tree edge* for v and mark the edge (u^r, v^r) as an image of this tree edge in G_0.

b) v belongs to the same active tree T. In which case do nothing.

c) v belongs to an active tree $T_2 \neq T$. Then the trees T and T_2 have collided. We must then do the following

1) Find skeletal paths from v to T_2.

2) Set (u, v) to be the tree edge for v. Find the skeletal path from v to T.

3) Expand both the skeletal paths into a network of edges.

4) Merge the two trees T and T_2.

Let us first see how we find the skeletal path from v to a tree T_2. Recall that when v was added to the tree T_2 (step (a)), then a tree edge was identified for v. Let this tree edge be (v, u). If v is already connected to the network of T, then v is the last node on the skeletal path. Else, (v, u) is the next radial edge, and u is the next node, on the skeletal path. We can now establish the rest of the skeletal path in a similar manner by starting at u.

As described in the proof of the performance guarantee, we shall maintain the connectivity information which will assist in determining if a node's network is connected to that of its parent. For each node v, we shall set $conn(v)$ to be 1 if v's network is connected to that of v's parent. Else we set $conn(v)$ to be 0. Thus for a node v on a skeletal path, if $conn(v) = 1$, then the skeletal path must end at v.

Recall that for each tree edge in the contracted graph, we also maintain its image in the original graph. Hence to expand a radial edge (u, v) belonging to a skeletal path, we add its image (u^r, v^r) to the network. We then connect u^r (respectively v^r) to the network of u (respectively the network of v). Since this construction is recursive, it is sufficient to describe how we connect the network of a node u' to that of its parent v'. Recall that the parent of u' is given by $parent(u')$. If $conn(u') = 1$, then the network of u' is already connected to that of its parent and we are done. Else we follow the method of skeletal path as described earlier. After the connection is made, we set $conn(u')$ to be 1.

Having connected the colliding trees, we must merge the colliding trees T and T_2. In doing so, we must update the parent information and also update the set of nodes of the merged tree to be $N(T_1) \cup N(T_2)$. Instead of changing the parent information for each node in $N(T)$ and every node in $N(T_2)$, we consider the smaller tree. Let this tree be T. For each node in $N(T)$, we change its parent pointer to T_2. We also merge the list $N(T_1)$ into $N(T_2)$. T_2 now represents the merged tree $T_1 \cup T_2$. Once all the merges have been performed for an active tree T, we also update the $V(T)$ and $B(T)$ as described in the $\frac{1}{2}$-packing algorithm. That completes the description of the implementation.

We now analyze the time complexity of our algorithm. Since each original edge is in at most one requirement cut, it follows that the total number of tree edges added through the course of the algorithm is at most m. Since establishing a tree edge requires performing a find operation on its endpoints, the total time required for maintaining the tree edges is $O(m\alpha(m, n))$. The time required to find a skeletal path is proportional to the

length of the path. In a manner analogous to that for proving the size of the network, it can be shown that the time required to construct the network is proportional to twice the size of the packing, and hence $O(n)$. Note that the total number of supernodes formed during the course of the algorithm is only $O(n)$. Since in merging two trees, we only update the parent information for the smaller sized tree, it follows that the parent information for no node is changed more than $\log_2 n$ times. Hence the total time required to update this information is $O(n \log n)$. The $\frac{1}{2}$-packing algorithm has a running time of $O(m\alpha(m,n)+n\log n)$. On adding the running times for all the steps described above, we still achieve a running time of $O(m\alpha(m,n)+n\log n)$ for our algorithm.

3.2 Our Algorithm: Weighted Case

3.2.1 The Algorithm

If the edges of the original graph have positive integral costs, then we can easily convert the problem to an unweighted problem. We simply replace each edge of cost c with c edges in series, each with cost 1. Then we just apply our algorithm given in the previous section. While the algorithm is correct, it has two drawbacks. One, the running time of the algorithm depends upon the costs, which is unacceptable. Two, such a strategy does not extend to non-integral costs. We, hence choose not to take the above strategy but rather modify our previous algorithm.

We construct a packing of cuts, where each cut in the packing has a weight, and we relate the cost of the network design to the value of this packing. To find the packing of cuts, we use an algorithm similar to the one for the unweighted case. At any timestep, we have a set of active trees. We grow the trees. In the unweighted case, we knew that we could safely grow each tree by one breadth-first-level, since Lemma 3.1.1 guaranteed that trees will not collide in the middle of an edge. Given that the edges have different costs, such a condition is no longer valid. We must hence replace the notion of breadth-first growth with the notion of shortest-path growth. We compute the smallest distance d by which to grow each tree from its boundary such that either of the following two events happen: at least two trees collide, or else a tree hits a new node. We then grow each of the trees by this amount. We add the corresponding cuts to the packing along with d as the weight for each of the cuts. When trees collide, we merge them and construct a network in exactly the same manner as before. The cost of the network is now the sum of the edge costs of the edges added to the network. We follow exactly the same proof as before to show that the size of the network found by the algorithm is at most twice the value of the packing found. We thus obtain the approximation guarantees in Theorems 2.3.1, and 2.3.5.

3.2.2 The Implementation

Our implementation for the weighted version follows closely along the lines of the unweighted version. However, there is one major difference. We must compute by how much should one grow all the active trees at a timestep.

Note that an edge can simultaneously be in the boundary of two distinct active trees, each containing an endpoint of the edge. In such a case, on growing the two trees by only half the cost of the edge, the trees will collide. On the contrary, if an edge is in the boundary of only one active tree, the tree can grow by as much as the cost of the edge without colliding with another tree.

Thus to compute the amount by which to grow the trees, we maintain the following for each active tree or supernode. We maintain a list of the nodes for the tree, a list of its boundary edges that are in two active trees at the step, and a list of its boundary edges that are not in any other active tree. We shall refer to the former edges as the *popular* edges and the latter as *isolated* edges. With each edge, we keep the amount an edge has been grown into so far. We shall denote this value for an edge e by $dist(e)$. The value of $dist(e)$ starts out as the cost of e.

At a timestep, for each tree, we compute the minimum value of *dist* over all the isolated edges, and the minimum value of the *dist* over all the popular edges. Let these values be x and y respectively. We choose $d = \min(x, \frac{y}{2})$ as the amount by which each of the trees is to be grown at the timestep. We subtract d from *dist* for each of the isolated edges, and $2d$ from *dist* of each of the popular edges. If this value now becomes zero for any of the edges, then we do the following. If the edge was in two active trees, we merge the trees. If the edge was in a single active tree, then that tree has now hit a new node. We update the boundary edges of this active tree by incorporating this new node.

In the process of merging two trees, edges could change identity from isolated edges to popular edges, and vice versa. The former happens, when the endpoint of a isolated edge not in any active tree becomes internal to some active tree. The latter happens when one of the endpoints of a popular edge become internal to a tree that becomes inactive at the timestep. We leave the details of updating the data structures to the reader. The implementation is straightforward and does not use anything but doubly linked lists. It suffices to mention that each timestep can be executed in $O(m)$ time, where m is the number of edges in the graph. The implementation for constructing the part of the network at a timestep can also be easily done in the same time bound by simply using doubly linked lists.

By construction, at each timestep, at least one of the trees has one more node than the previous step. It then follows that the number of timesteps for the algorithm is at most n. Hence the total running time for the algorithm is $O(mn)$, and we obtain the running time for Theorem 2.3.1.

3.3 Our Algorithm: R-multijoin

So far we have dealt with the case in which each site pair need only be connected in the final network. As discussed in Section 2.3, a client may also require that there be at least r_{ij} edge-disjoint paths between her pair of sites.[1] Thus the case dealt with up to now requires each r_{ij} to be either 0 or 1.

[1] In this case, the network is allowed to use multiple copies of edges of the input graph; each copy of a given edge costs the same.

In order to obtain an approximation algorithm for this generalized problem from our algorithm for the R-join case (0–1 requirements), we make use of a heuristic technique due to Goemans and Bertsimas [72]. They propose a technique they call the *tree heuristic*, which consists essentially of decomposing a problem with many different requirement values into a series of simpler problems in which only two requirement values appear. As we mentioned in Section 2.3, they use the technique for solving only a special case of the R-multijoin problem. In conjunction with our new algorithm for the R-joinproblem, however, the technique can be easily adapted to apply to the general case.

Let the different values of r_{ij} be $0 = p_0 < p_1 < p_2 < \cdots < p_s$. For each $0 < d \leq s$, consider the transformed problem

$$r_{ij}^d = \begin{cases} p_d - p_{d-1} & \text{if } r_{ij} \geq p_d \\ 0 & \text{otherwise} \end{cases}$$

which is essentially $p_d - p_{d-1}$ copies of a 0–1 problem problem. Use a standard heuristic to find an approximately optimal solution, and combine the solutions to the s transformed problems to get a solution to the original problem. The resulting performance guarantee is $\Theta(s)$.[2] By using a similar approach, if each r_i is an integer b bits long, then the original problem can be decomposed into b problems, and the resulting performance bound is $2(1 - 1/k)b$. This is how we get the performance bound stated in Theorem 2.3.3.

3.4 Constructing an Optimal c-packing of R-cuts

In Section 3.2, we presented an algorithm for constructing a $\frac{1}{2}$-packing of cuts that is nearly maximum. Each cut in a $\frac{1}{2}$-packing has a weight of either $\frac{1}{2}$ or 1. By allowing the cuts to have arbitrary fractional weights, we can in fact show that we can find a fractional c-packing of R-cuts of maximum value in polynomial time. Our algorithm uses linear programming. We claim that a solution to the following linear program represents a c-packing of cuts, and that a maximum c-packing can be obtained the optimal solution.

Let $G = (V, E)$ be a graph with edge costs C and let Q be a set of specified requirement pairs. Each requirement $q \in Q$ is represented by a pair (s_q, t_q), where s_q and t_q are the two endpoints of the requirement. We first solve the following linear program.

$$\text{maximize} \sum_{q \in Q} d_q(t_q) - d_q(s_q) \tag{3.4}$$

$$\text{subject to} \sum_{q \in Q} \ell_q(uv) \leq C(uv), \text{ for each edge } (u,v) \in E \tag{3.5}$$

$$d_q(v) \leq d_q(u) + \ell_q(uv), \forall (u,v) \in E, \forall q \in Q \tag{3.6}$$

$$\ell_q(uv) \geq 0, \ \forall (u,v) \in E, \forall q \in Q \tag{3.7}$$

$$d_q(u) \geq 0, \ \forall q \in Q, \forall u \in V \tag{3.8}$$

[2] More specifically, Goemans and Bertsimas show the performance bound is $2(1 - 1/k)(\sum_{d=1}^{s}(p_d - p_{d-1})/p_d)$.

$d_q(t_q) - d_q(s_q)$ denotes the length of the shortest path from s_q to t_q under the length assignment ℓ_q.

We now show how to interpret a solution to this linear program (LP) as a packing of requirement cuts. For each requirement pair (s_q, t_q), there is an assignment of nonnegative real lengths to edges ℓ_q. The contribution of that requirement pair to the objective function (3.4) is the shortest path distance from s_q to t_q, under the length assignment ℓ_q. Let $d_q(v)$ be the shortest path distance from s_q to v under the same length assignment. Let X be the set of nodes whose distance from s_q is at most $d_q(t_q)$. We partition the set X into sets $X_0, X_1, \ldots, X_k$ consisting of nodes with progressively increasing distance from s_q. Formally,

$$\begin{aligned} X_0 &= \{v \mid d_q(v) = 0\} \\ X_{i+1} &= \{v \mid d_q(v) > d_q(u), \forall u \in X_i\}, \text{ for } 0 < i < k-1 \\ X_k &= \{v \mid d_q(v) = d_q(t_q)\} \end{aligned}$$

By construction, each node in X_i must have the same distance. Let this distance be denoted by d_q^i. Then we include the following s_q-t_q cuts in our packing:

$$\Gamma(\{X_0 \cup \ldots\ldots X_i\}) \text{ with weight } d_q^{i+1} - d_q^i$$

for $0 \leq i < k$.

Then the sum of the weights of s_q-t_q cuts is indeed the distance from s_q to t_q. Hence the value of the objective function is the sum $\sum_{q \in Q} d_q(t_q)$. Moreover, each edge $uv \in E$ obeys the following *packing constraint*:

the sum of the weights of all cuts containing uv is at most $\sum_{q \in Q} \ell_q(uv)$.

This follows from the fact that $d_q(v) \leq d_q(u) + \ell_q(uv)$ for every requirement pair q and every edge uv. Consequently, the total weight of requirement cuts containing an edge uv, summed over all requirement pairs (s_q, t_q), is at most $C(uv)$ by (3.5) of the above LP.

We thus get the following result.

Lemma 3.4.1 *Given a graph G with edge-costs c and a set of requirement pairs, there is a polynomial-time algorithm for finding an optimal c-packing of requirement cuts.*

3.5 Obtaining a Cross-free c-packing of R-cuts

In this section we define a family of cross-free cuts, and show that the c-packing of cuts constructed by our algorithm is cross-free. This is the only polynomial-time algorithm we know for generating an approximately maximum c-packing of cuts for arbitrary requirement pairs that is also cross-free. The cross-free property will be important in obtaining an approximation algorithm for generalized Steiner toughness later in Chapter 4.

We will denote a cut by $\Gamma(U)$, where U is a node-subset not containing the node r. A pair of distinct cuts $\Gamma(U)$ and $\Gamma(W)$ is said to *cross* if U and W are not disjoint and neither is a subset of the other. (Intuitively,

this means that in a Venn diagram the circles signifying U and W cross.) A family of cuts is *cross-free* (also called *uncrossed*) if no two cuts cross. Note, for example, that for each q, the family of s_q-t_q cuts we obtained in Subsection 3.4 is a cross-free family. However, the family of all the cuts obtained is not generally cross-free.

Let us first show that the c-packing of cuts given by our algorithm of Section 3.2 is cross-free. Recall that if $\Gamma(U)$ is in the packing, then U formed an active tree at the timestep the cut was added to the packing. Consider any two cuts $\Gamma(U)$ and $\Gamma(V)$ added to the packing at timesteps t_1 and t_2 respectively. If $t_1 = t_2$, then U and V are disjoint, since each of the active trees at a timestep are node-disjoint. If $t_1 < t_2$, then U is either contained in some active tree T in G_{t_2}, or it is not. If it is, then either the set of nodes of T equals W, in which case $U \subseteq W$, or else the set of nodes of T, and hence U, is disjoint from W. If U is not in an active tree, thn it is in a supernode, and clearly the nodes of the supernode are disjoint from those of W. A similar argument can be made if $t_1 > t_2$. Thus we have shown the cross-free nature of the packing obtained by our algorithm.

Lemma 3.5.1 *The cuts in the approximately optimal c-packing found by our algorithm are cross-free.*

3.6 Conclusions

In this chapter we established the results relating to the problem of finding a network whose total edge-cost is minimum. We started out by giving an approximation algorithm for the special case when all the edge-costs and requirement values are one. We then built on this result to give an approximation algorithm for the case with arbitrary edge-costs, which was then further used to approximate the case with arbitrary requirement values and arbitrary edge-costs. In each of the cases, we established approximate min-max equality results between the network design problems and packing of requirement cuts. Of special interest is the fact that the packing of cuts obtained by our algorithm is cross-free.

As a substep to the problem of approximating the minimum size of an R-join, we found approximately the maximum 1-packing of requirement cuts in a bipartite graph in which all the sites are on one side of the bipartition. As described in Section 2.3.3, this algorithm can be used in estimating the reliability of a network.

Chapter 4

Minimum-degree R-join

Consider the following application. In computer networks, broadcasting a message from one node to all the others is done routinely through a spanning tree. The amount of work done by any node in such a broadcast is then its degree in this spanning tree. To balance the work load evenly, it is desirable to choose a spanning tree with small maximum degree. As explained in Section 2.4, constructing a network of small degree has several applications. It is important in designing large networks for reliability reasons, for optimal switch design, or for decentralized communications processing. However, finding a network of smallest maximum degree is also NP-complete, and we shall consider approximation algorithms for this problem here.

Earlier we claimed our results relating to this problem in Chapter 2. We shall substantiate those claims in this chapter.

We have the first approximation algorithm for the minimum-degree *uniform* requirement-join problem. We discovered a relationship between the minimum-degree and the node-toughness in a graph, which we formalize here in terms of an approximate min-max equality. In showing this relationship, we also establish the performance guarantee of our algorithm for the minimum-degree problem. As a substep, we also provide a constant factor approximation algorithm for finding the Steiner node-toughness for any uniform requirement-join formulation in a graph. Our algorithm for Steiner toughness also extends to the case of generalized Steiner toughness. Finding the toughness of a graph was finally proved NP-complete only recently [7], and our approximation algorithms are the first ones known for these problems.

Our algorithm for solving the minimum-degree problem is to reduce it to an integral version of the multicommodity problem. This latter problem is well-studied and we shall use some previous results relating to this problem. For establishing the relationship between the minimum-degree to the node-toughness, we shall require some of our results from the previous chapter on the minimum edge-cost R-join problem. We shall also establish results relating the node-toughness to the optimal value of a certain linear program.

This chapter is organized as follows. We start out by giving an approximation algorithm for the generalized

toughness problem. We then give the algorithm for the minimum-degree problem, prove its performance guarantee, and establish the approximate min-max relation between minimum-degree and the inverse of toughness for the case of uniform requirements. When we refer to an R-join problem in this chapter, we refer only to the case of uniform requirements unless otherwise noted.

4.1 Node-toughness

In this section, we shall give a constant factor approximation for the toughness problems. Recall from Section 2.4.2 that given a set of requirements for a graph $G = (V, E)$, the *generalized Steiner toughness* $\tau(X)$ for a node-set X is the ratio of the size of X and the number of significant components in $G - X$. A component is considered significant if it separates at least one requirement pair. The generalized Steiner toughness of a graph is the minimum value of $\tau(X)$ over all sets $X \subset V$. We shall denote this toughness ratio by τ^*. Thus

$$\tau^* = \min_{X \subset V} \frac{|X|}{\text{number of significant components in } G - X}$$

. The *Steiner toughness* is a special case of the generalized Steiner toughness where the requirements are specified entirely by a set of sites $S \subseteq V$; there is a requirement with endpoints u and v if and only if $u, v \in S$. A component is then significant if it contains at least one site from S. In the special case when $S = V$, Steiner toughness will simply be referred to as *toughness.*

For notational convenience, when we refer to toughness in this section, we shall refer to the generalized Steiner toughness, unless otherwise noted.

4.1.1 Our Algorithm: An Overview

To find a node set that achieves nearly optimal toughness ratio, we do the following. Given the set of requirements Q for a graph G, we shall construct a linear program which we shall describe later. The optimal solution of this linear program will be a packing of requirement cuts under certain constraints. While obeying these constraints, we shall find an approximately optimal packing of uncrossed requirement cuts using our techniques described in Section 3.5. Using a simple rule we shall associated a set of nodes with each requirement cut. We shall show that one of the node sets associated with the cuts in the packing yields approximately the best toughness ratio.

We shall now fill in the details of each of the steps.

4.1.2 The Linear Program for Toughness

The linear program is formulated in the following manner. We first modify the original graph G by splitting each edge (u, v) into two edges (u, x) and (x, v), where x is a new node, which we call a *pseudo-node* to distinguish it from the *original* nodes of G. Let $G' = (V', E')$ be the new graph. We define our linear program in terms of G'.

Let Q be the specified set of requirements. We shall denote the two endpoints of a requirement pair $q \in Q$ by s_q and t_q. We are to assign costs R to the original nodes and lengths l_q to the edges of the graph G', so as to

$$\text{maximize} \sum_{q \in Q} d_q(t_q) - d_q(s_q) \tag{4.1}$$

$$\text{subject to} \sum_{q \in Q} \ell_q(ux) \leq R(u), \forall (u,x) \in E' \text{ where } x \text{ is a pseudo-node} \tag{4.2}$$

$$\sum_{v} R(v) \leq 1 \text{ where the sum is over all original nodes.} \tag{4.3}$$

$$d_q(v) \leq d_q(u) + \ell_q(u,v), \forall (u,v) \in E', \forall q \in Q \tag{4.4}$$

$$\ell_q(u,v), d_q(u) \geq 0, \ \forall (u,v) \in E', \forall u \in V', \forall q \in Q \tag{4.5}$$

$d_q(t_q) - d_q(s_q)$ is the shortest path from s_q to t_q under the length assignment ℓ_q.

4.1.3 Obtaining a Cross-free Packing of R-cuts

Once we solve the above linear program, we obtain the costs on the nodes and the edge-lengths. We can then interpret the shortest paths under this length assignment as a packing of requirement cuts using our technique of Section 3.4. This packing satisfies the constraint that for each edge (u,x) the weight of the cuts in the packing containing (u,x) is at most $R(u)$. The optimum value of the objective function for the linear program is the value of this packing.

For the purposes of our algorithm, we shall require a packing of uncrossed requirement cuts which we construct as follows. We solve the above linear program and obtain the node costs for each original node. To each edge (u,x), we assign the cost $R(u)$, and then apply our algorithm of Section 3.2 to find an approximately optimal packing of requirement cuts under this assignment of edge-costs. This packing is cross-free by Lemma 3.5.1 and has value at least one-half the optimal by Theorem 2.3.5.

4.1.4 Obtaining a Toughness Cut

Having found an approximately optimal packing of requirement cuts as described in the previous section, we shall group the cuts in the packing into sets which we call *strata*. Each strata will have the following three properties.

Property 1: All the requirement cuts in a stratum S have the same weight, and we shall refer to it as the *weight* of S. To obtain this property, a cut λ in the packing with weight w might have to be represented multiple times in the packing, and the weight w is then divided among these cuts.

Property 2: For any two requirement cuts $\Gamma(U)$ and $\Gamma(W)$ in a single stratum, node sets U and W are disjoint.

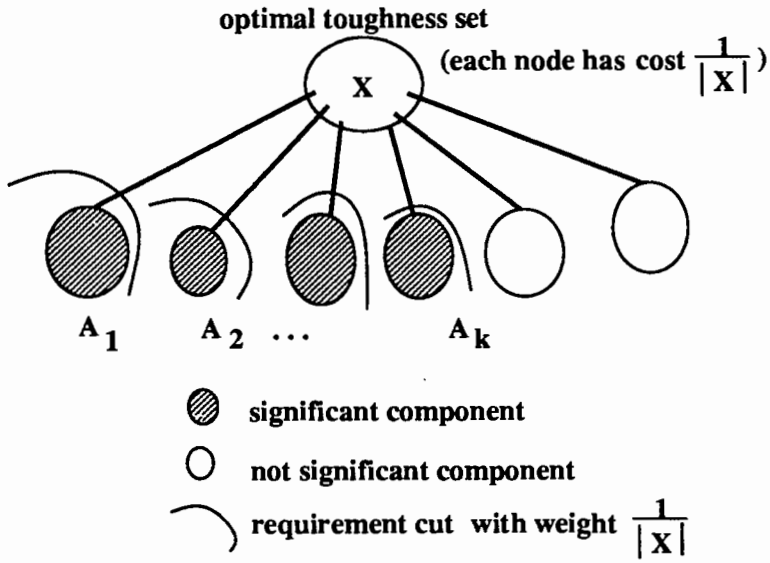

Figure 4.1: A feasible solution for the linear program for node toughness is $\frac{1}{\tau^*}$, which equals $\tau(X)$ for the optimal node-toughness cut X. $\tau(X)$ is given by $\frac{k}{|X|}$.

Property 3: For any original node u of the transformed graph, the total weight of the stratum containing edges incident to u is at most $2R(u)$.

We shall describe a polynomial-time algorithm for constructing the strata in Section 4.1.6. For the present, let us assume that we have such an algorithm. From each stratum S, we shall construct a set of boundary nodes denoted by $B(S)$ in the following manner. We consider the union of the edges in the requirement cuts forming a stratum. For such each such edge (u, x), where x is a pseudo-node and u is an original node, we add u to the node set $B(S)$. We shall show that for one of the strata, the set of boundary nodes thus constructed achieves approximately the minimum toughness.

4.1.5 Performance Guarantee

First, we show that a packing of cuts with value $\frac{1}{\tau^*}$ is achievable by the linear program in (4.1).

Lemma 4.1.1 *The optimal value of the linear program given in (4.1) is at least* $\frac{1}{\tau^*}$.

Proof:

Let X be the optimal toughness cut in the graph. Any pseudo-node with both its neighbors in X is added to X. We assign a uniform cost of $\frac{1}{|X|}$ to each of the original nodes in the cut (see Figure 4.1). The rest of the nodes are assigned a cost of 0. Let the significant components induced in the graph on removal of X be $A = \{A_1, \ldots, A_k\}$. Let A_i contain a separated requirement pair q_i. Then the cocycle $\Gamma(A_i)$ separates q_i and hence is a requirement cut. We assign this cut a weight of $\frac{1}{|X|}$. Equivalently, for each edge u with endpoints in

X and A_i, we assign

$$\ell_q(u) = \begin{cases} \frac{1}{|X|} & \text{if } q = q_i \\ 0 & \text{otherwise} \end{cases}$$

It is easy to check that the length assignments obey the constraints in (4.2). The total weight of the set of cuts found is $\frac{k}{|X|}$, which equals $\frac{1}{\tau^*}$. □

We shall now show an upper bound on the value of the linear program in terms of the toughness ratio obtained by our algorithm. Let $S = (S_1, \ldots, S_k)$, be the strata formed by our algorithm, and let the corresponding set of boundary nodes be $B_1, \ldots, B_k$ respectively. If we represent the toughness achieved by the algorithm by τ_{alg} then by definition

$$\tau_{alg} = \min_{1 \leq i \leq k} \{\tau(B_i)\}$$

We shall show that the optimal value of the linear program is not too much more than $\frac{1}{\tau_{alg}}$.

Lemma 4.1.2 *The optimal value of the toughness linear program given in (4.1) is at most* $2f(\frac{1}{\tau_{alg}}+1)$, *where* f *is the factor of approximation used in finding an uncrossed packing of requirement cuts.*

Proof: Let P_{opt} denote the maximum value of the linear program given in (4.1), and let P_{alg} be the value of the packing obtained by our algorithm. Since the value of the packing does not change in forming the strata, it follows that P_{alg} is the sum over the strata of the weight of the cuts in a stratum. Let us denote the weight of a cut C by $w(C)$ and the weight of a strata S_i by w_i. Then we have

$$P_{alg} = \sum_{S_i \in S} \sum_{C \in S_i} \text{weight of the cut } C$$

Since the weight of each of the cuts in a stratum S_i is the same and equals w_i, it follows that

$$P_{alg} = \sum_{S_i \in S} w_i \sum_{C \in S_i} 1 \tag{4.6}$$

Note that $\sum_{C \in S_i} 1$ equals the number of cuts comprising stratum S_i.

Let γ_i be the number of significant components in $G - B_i$. We claim that the number of cuts in the stratum S_i is no more than the size of B_i plus γ_i. In other words,

$$\sum_{C \in S_i} 1 \leq \gamma_i + |B_i| \tag{4.7}$$

where $|B_i|$ is the number of original nodes in B_i. To understand this, we must understand how the edges in a stratum are converted to a boundary node set. Each cut in a stratum separates at least one requirement pair. Hence by Property 2, the number of components formed on removal of the edges in a stratum is at least as much as the number of cuts forming the stratum. We need to relate this quantity to the number of significant components formed on removing the boundary node set associated with the stratum. Let $\Gamma(U)$ be a requirement

cut in the stratum such that it separates a requirement q. Then q must contain exactly one endpoint in U. Let this endpoint be s_q. There are two possibilities for s_q – it either belongs to the boundary node set associated with this stratum, or it does not. If it belongs to the boundary node set, then it contributes one to the size of the boundary node set. If it does not belong to the boundary node set then q is still separated by the boundary node set. Thus we get (4.7).

Next, by definition of τ_{alg}, we have

$$\frac{|B_i|}{\gamma_i} \geq \tau_{alg}$$

for each i. On substituting the above in equation (4.7), we get

$$\sum_{C \in S_i} 1 \leq |B_i| \left(\frac{1}{\tau_{alg}} + 1 \right) \tag{4.8}$$

On substituting (4.8) into (4.6) we get

$$P_{alg} \leq \sum_{S_i \in S} w_i |B_i| \left(\frac{1}{\tau_{alg}} + 1 \right) \tag{4.9}$$

Since $|B_i|$ is the number of original nodes in B_i, we can write

$$P_{alg} = \left(\frac{1}{\tau_{alg}} + 1 \right) \sum_{S_i \in S} w_i \sum_{\text{original node } v \in B_i} 1 \tag{4.10}$$

On changing the order of summation, we obtain

$$P_{alg} = \left(\frac{1}{\tau_{alg}} + 1 \right) \sum_{\text{original node } v \in V} \sum_{S_i \in S, B_i \ni v} w_i \tag{4.11}$$

Since the total weight of the strata that a node v occurs in is at most $2R(v)$ by Property 3, we get

$$P_{alg} \leq \left(\frac{1}{\tau_{alg}} + 1 \right) \sum_{\text{original node } v \in V} 2R(v) \tag{4.12}$$

$$\leq 2 \left(\frac{1}{\tau_{alg}} + 1 \right) \tag{4.13}$$

where the last inequality follows from the constraint (4.3) on the node costs.

Since the factor of approximation for P_{opt} is f, on substituting $P_{opt} \leq f P_{alg}$ into (4.13), we get the result. □

From Lemmas 4.1.1 and 4.1.2, it follows that the optimal value of the linear program is at least $\frac{1}{\tau^*}$ and at most $2f\left(\frac{1}{\tau_{alg}} + 1\right)$. Hence

$$\frac{1}{\tau^*} \leq P_{opt} \leq 2f\left(\frac{1}{\tau_{alg}} + 1\right) \tag{4.14}$$

implies that

$$\tau_{alg} \leq \frac{2f\tau^*}{1-2f\tau^*} \tag{4.15}$$

As discussed earlier, we can find a fractional packing of uncrossed cuts that is approximately optimal. The approximation factor is a factor of 2. Hence f equals 2, and we get

$$\tau_{alg} \leq \frac{4\tau^*}{1-4\tau^*} \tag{4.16}$$

For $\tau^* \leq \frac{1}{8}$, we directly get $\tau_{alg} \leq 8\tau^*$. However, note that τ^* can be at most 1, for removal of a site yields at least one component. Thus, for the values of τ^* between 1 and $\frac{1}{8}$, removing any site gives a cut with node-toughness within a factor of 8 of the optimal. Thus we can find a node set that achieves a generalized Steiner toughness which is within a factor of 8 of the optimal. This proves the approximation factor stated in Theorem 2.4.3.

4.1.6 Forming the Strata

It only remains to show that given a packing of uncrossed cuts, we can form the strata in polynomial time that satisfy the three properties given in Section 4.1.4.

The Hasse Diagram, and Contiguity of Edges

In this section, we review some known properties of cross-free families of cuts. Recall that we use the notation $\Gamma(W)$ to denote the set of edges that have exactly one onepoint in W.

Consider the Hasse diagram of a cross-free family of cuts $\Gamma(W)$, ordered by the inclusion relation on the node-sets W.

Lemma 4.1.3 *The Hasse diagram is a forest.*

Proof: Suppose $\Gamma(W_1) \preceq \Gamma(W_2)$ and $\Gamma(W_1) \preceq \Gamma(W_3)$. In this case, W_1 is a subset of both W_2 and W_3, so W_2 and W_3 are not disjoint. By the cross-free property, either W_2 is contained in W_3 or vice versa. □

The proof of the following lemma is similar.

Lemma 4.1.4 *If cuts $\Gamma(U)$ and $\Gamma(W)$ are incomparable in the partial order, then the node-sets U and W are disjoint.*

Finally, let (u, v) be an edge, and consider the set $F(u, v)$ of cuts containing the edge (u, v). Each such cut has the form $\Gamma(W)$, where either W contains u and does not contain v, or vice versa. The cuts $\Gamma(W)$ where W contains u must form a simple path going up the Hasse diagram, and similarly for those cuts $\Gamma(W)$ where W contains v. For every cut $\Gamma(W)$ on both paths, W contains both u and v, and so the edge (u, v) does not belong to the cut. Thus the set $F(u, v)$ of cuts containing the edge (u, v) form two upward-going paths in the Hasse diagram as shown in Figure 4.2.

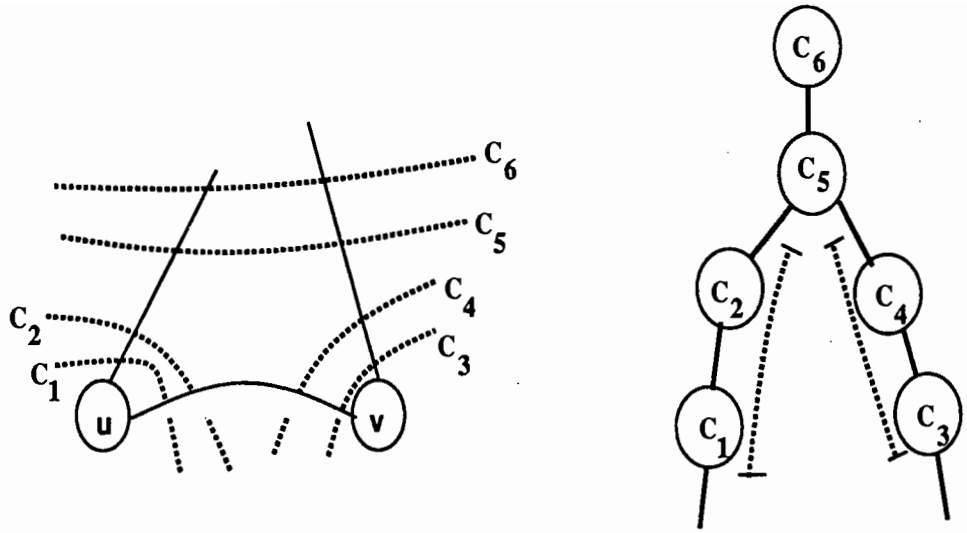

Figure 4.2: A portion of the graph with an edge (u, v) and the corresponding portion of the Hasse diagram of cocycles under set inclusion. The set of cuts containing any edge form two paths in the Hasse digram. The cuts C_1, C_2, C_3 and C_4 containing the edge (u, v) form the two dotted paths in the Hasse diagram.

Stratification and Depth in the Hasse Diagram

We use the properties outlined in Subsection 4.1.6 to define a kind of stratification of the Hasse diagram with nice properties.

We give a recursive construction of the strata. Let F be the set of cuts that are maximal in the partial order, and let λ be the minimum weight of any cut in F. Then the top stratum consists of the cuts F and has weight λ. The remaining strata are those recursively constructed for the partial order obtained by reducing the weight of all cuts in F by λ, and then deleting cuts that have zero weight. Thus intuitively the strata are defined by a successive stripping-off process (see Figure 4.3). Note that for any cut C, the total weight of all strata in which C appears is just the weight of C. We say a stratum contains an edge if the edge belongs to one of the cuts comprising the stratum.

We shall introduce some notations that we will use later. For a cut C, let the *starting depth* $d(C)$ be the total weight of all proper ancestors of C in the Hasse diagram, and let the *ending depth* $d'(C)$ be $d(C)$ plus the weight of C itself. Let $w_1, w_2, \ldots$ be the weights of the strata. Each stratum then corresponds to a range of depth. The first stratum consists of cuts C with $d(C) = 0$ and $d'(C) = w_1$, the second stratum consists of cuts C with $d(C) = w_1$ and $d'(C) = w_1 + w_2$, the third stratum consists of cuts C with $d(C) = w_1 + w_2$ and $d'(C) = w_1 + w_2 + w_3$, and so on. In general, the i^{th} stratum consists of cuts C with $d(C) = w_1 + \cdots + w_{i-1}$ and $d'(C) = w_1 + \cdots + w_i$.

Since there are only a polynomial number of cuts, building the Hasse diagram takes only polynomial time. Moreover, stratification can be done in $O(n)$ time once the Hasse diagram is built. Hence the running time for

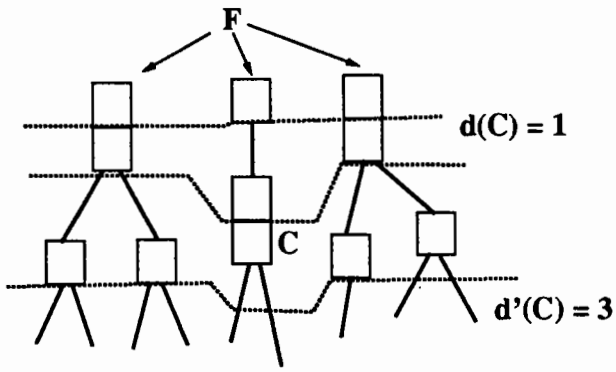

Figure 4.3: Stratifying the Hasse diagram. For simplicity, all the cuts are assumed to have integral weights. A cut is represented by as many squares as its weight. We strip off one square from each of the maximal elements in the Hasse diagram. This forms a stratum. We now repeat the process with the rest of the diagram. The starting depth of a cut C is the number of strata removed before C became a maximal element. The ending depth is the count of the last stratum containing C.

forming the strata is polynomial in n.

Establishing the Properties of the Strata

By construction, each of the cuts in the stratum have the same weight, and hence Property 1 is trivially satisfied. Property 2 follows from Lemma 4.1.4. We shall now that Property 3 also holds for our construction of the strata.

It is immediate from the packing constraints that for an edge (u, x), the total weight of cuts containing (u, x) is at most $R(u)$. We use this fact to bound the total weight of all strata containing any edge with endpoint u.

Lemma 4.1.5 *Let u be an original node of G. The total weight of all strata containing edges incident to u is at most $2R(u)$.*

Proof: We analyze the Hasse diagram. By Lemma 4.1.3, the Hasse diagram is a forest. For convenience in the following proof, we make it a tree by adding a top element of weight zero, whose node-set is the set V of all the nodes of the graph. Formally, the node-set V does not correspond to a cut, i.e. $\Gamma(V)$ is not a cut, since it is not separating any requirement pair. Note that introducing the top element does not change the starting or ending depths $d(\cdot)$ and $d'(\cdot)$ of cuts in the Hasse diagram.

Now we commence the proof. Of the cuts containing edges incident to u, let $C_{\text{lo}} = \Gamma(W_{\text{lo}})$ be the one with the greatest ending depth $d'(C)$. Let $C_{\text{hi}} = \Gamma(W_{\text{hi}})$ be the one with the least starting depth $d(C_{\text{hi}})$. We show that $d'(C_{\text{lo}}) - d(C_{\text{hi}}) \leq 2R(u)$.

Let $C_{\text{md}} = \Gamma(W_{\text{md}})$ be the lowest ancestor of C_{lo} such that $u \in W_{\text{md}}$. Note that C_{md} may be C_{lo} or C_{hi} or any cut in between. Suppose that C_{md} is not C_{lo}. Then since C_{lo} contains some edge incident to u, the node-set W_{lo} must contain some neighbor x of u. Then by definition of the partial order, x is contained in

every node-set $\widehat{W}$ for which $\Gamma(\widehat{W})$ is an ancestor of $C_{\rm lo}$. Consequently, the edge xu is contained in every cut that is an ancestor of $C_{\rm lo}$ and a proper descendent of $C_{\rm md}$. By the capacity constraint of (4.1), the total weight of all such cuts is at most $R(u)$. Thus we have the inequality

$$d'(C_{\rm lo}) - d'(C_{\rm md}) \leq R(u)$$

Note that this inequality holds trivially if $C_{\rm md}$ is $C_{\rm lo}$.

Let the parent of $C_{\rm hi}$ be $C_{\rm hi+1}$. Note that $d'(C_{\rm hi+1}) = d(C_{\rm hi})$. By an argument similar to the above, we can show that $d'(C_{\rm md}) - d'(C_{\rm hi+1}) \leq R(u)$. This implies that

$$d'(C_{\rm md}) - d(C_{\rm hi}) \leq R(u)$$

Summing the two inequalities yields $d'(C_{\rm lo}) - d(C_{\rm hi}) \leq 2R(u)$. □

4.2 Minimum-degree Problem

In this section, we give a background on multicommodity flow. We describe our algorithm for the minimum-degree problem by reducing it to a variant of the multicommodity flow problem, and present a simple proof of the performance guarantee. Using a more involved proof, along with proving the performance guarantee for our algorithm, we also relate the minimum-degree to the inverse of node-toughness.

4.2.1 Concurrent Flow

An essential ingredient to our algorithm is *concurrent flow*. Concurrent flow is an optimization version of multicommodity flow defined by Sharokhi and Matula [144]. Suppose we are given an instance of multicommodity flow: a network with edge- and/or node-capacities, and a set of *commodities*, each specifying a pair s_i, t_i of nodes and a *demand*, a positive value d_i specifying a required amount of flow.

An assignment of flow is *feasible* if for every edge, the flow through that edge is at most the edge's capacity, and for every node, the flow through that node is at most the node's capacity. The instance is *feasible* if there exists a feasible flow. Even for a non-feasible assignment of flow, it is possible to measure how far it is from being feasible. For any assignment of flow satisfying the demands, the utilization u is the maximum ratio of flow on an edge (node) to the capacity of that edge (node). One way to interpret u is that it is the minimum multiplier such that the assignment of flow would be feasible if all capacities were multiplied by u. The concurrent flow problem is to find the minimum utilization u, i.e. the minimum number such that the instance becomes feasible when all capacities are multiplied by u.[1] We shall call u the congestion.

It is easy to find the minimum congestion for a given instance of concurrent flow using any algorithm for linear programming [71]. However, the solution found will, in general, be *fractional*; That is, to achieve this

[1] Sharokhi and Matula originally defined the problem without node capacities, but the extension of the definition is immediate.

congestion flows are split among various paths carrying *fractional* flow values. In this work we shall require that flow paths carry *integral* flow values. Minimizing congestion under this condition is NP-complete [49]. However, Raghavan and Thompson [132] have given a polynomial-time algorithm for approximately minimizing congestion under the integral flow restriction. Raghavan and Thompson show that given any fractional flow solution with congestion u_F, one can use their *randomized rounding* technique to obtain an integral flow solution. The technique is to choose a flow path for each commodity randomly from among the fractional flow paths. The probability of choosing a flow path is proportional to its flow value. They show that the congestion for the integral flow solution produced by this rounding technique is close to u_F.

Theorem 4.2.1 (Raghavan & Thompson) *Let the value of the minimum congestion for a fractional flow solution in a multicommodity flow problem be denoted by u_F. Then there is an integral flow solution with congestion u_I, where $u_I = u_F + O(\sqrt{u_F \log n})$ if $u_F = \Omega(\log n)$, and $u_I = O(\log n)$ otherwise. Here, n is the number of nodes in the graph.*

Notice that the above theorem implies that we can find an integral solution to the node congestion problem with integral congestion $u_F + O(\log n)$. Raghavan [131] has also shown that such an integral solution can be obtained in deterministic polynomial time. Alternatively, one can use the fast approximation algorithm of Klein, Stein, and Tardos [91] to directly obtain an integral solution.

4.2.2 Our Algorithm for Minimum-degree

We now describe the algorithm for finding a network of approximately the minimum degree connecting a given set of sites.

Our algorithm is very simple. We reduce the problem to the following integral concurrent flow problem to minimize node congestion. We take an arbitrary ordering of the sites. Let u be some site and S_u be the set of all sites numbered higher than u in the ordering. For each $v \in S_u$, we shall define a commodity q_{uv} to flow between u and v, and require that the sum over $v \in S_u$ of the flows q_{uv} is 1, i.e.

$$\sum_{v \in S_u} q_{uv} = 1, \forall \text{ site } u$$

We now aim to establish integral flow paths for the prescribed flow requirements as to minimize the maximum node-congestion over the original nodes. Since finding the best solution is NP-complete, we use an approximation algorithm as described in Section 4.2.1. The union of the integral flow paths forms a subgraph connecting the sites of approximately minimum degree.

We now prove the correctness of the algorithm. Since the flow paths are integral, and at least one unit of flow must be pushed between a site u and the sites in S_u, it follows that in the solution there is an integral path from u to some site numbered higher than u. Thus through a series of such integral flow paths, each site is connected to the highest numbered site. The union of these flow paths then form a subgraph connecting all the sites. Moreover, the maximum degree of a node is at most the congestion for the node.

4.2.3 A Simple Proof of the Performance Guarantee

The minimum-degree is at most the integral node congestion for the above flow problem. We shall demonstrate that there exist integral flow paths for the above concurrent flow problem that have node congestion $O(\Delta^* \log k)$, where Δ^* is the minimum degree of any R-join, and k is the number of sites. By Theorem 4.2.1, the node-congestion and hence the value of the minimum-degree found by our algorithm is at most $O(\Delta^* \log k + \log n)$. That establishes the performance guarantee of our algorithm in Theorem 2.4.1.

This proof was proposed by R. Ravi [133], and is similar to one of the steps in our proof for the approximate min-max relation involving the minimum-degree problem that we shall encounter later.

The integral flows demonstrating the claimed congestion are chosen in the following manner. Consider the R-join with minimum-degree Δ^*. Let the Euler tour of this tree be denoted by P. In the Euler tour, some sites may occur multiple times and so we choose exactly one of the copies to be *active*. The commodity associated with a site v is routed along P to an active site w such that w is numbered higher than v and the number of active sites in the path from v to w is minimum. Ties are broken arbitrarily. We shall show that under this scheme, the congestion of each edge in P is at most $4 \log k$. This implies that the value of the node congestion for any node is at most $4 \log k$ times its degree. But since the maximum degree of any node in this tree is Δ^*, the node congestion of any node is at most $4\Delta^* \log k$.

Now, we show that the value of the edge congestion of any edge in the tree is at most $4 \log k$. Since there are exactly two copies of an edge in the Euler tour, we show that the congestion of each copy is at most $2 \log k$. For this, we show that the value of the congestion contributed from flows routed along one direction in the edge is at most $\log k$. The key observation is that if there are x flows being routed through an edge e in some direction, then there must be at least $2^x - 1$ active sites on the side of the path P from which these flows are being routed. Let the sites that originate these flows be numbered $v_1, \ldots, v_i$ in the order of increasing distance (in terms of the number of active sites) from the edge e along P. Note that v_{i+1} must be numbered higher than v_i, for otherwise, it would simply route to v_i using the shorter path, rather than all the way across the edge e. If v_i were closer to v_{i+1} than to the edge e, then the shortest routing path for v_i is to v_{i+1} and not across the edge e as portrayed. Thus the site v_i must be at least as far away from v_{i+1} as from the edge e (See Figure 4.4). It is easy to infer that the number of active sites between the edge e and v_i is at least $2^i - 1$. Since there are k active sites in P, the congestion of the edge in one direction is at most $\log k$ and the result follows.

4.3 Relation to Node-toughness

In this part, we give a different proof of the performance guarantee of our algorithm that also relates the minimum-degree to the inverse of the node-toughness. Once again, our discussion holds only for the case of uniform requirements.

We will show that we can find an integral solution to the concurrent flow problem whose node-congestion is at most $O(\frac{\log k}{\tau^*})$. We can thus construct a network whose maximum degree is also within the same bound.

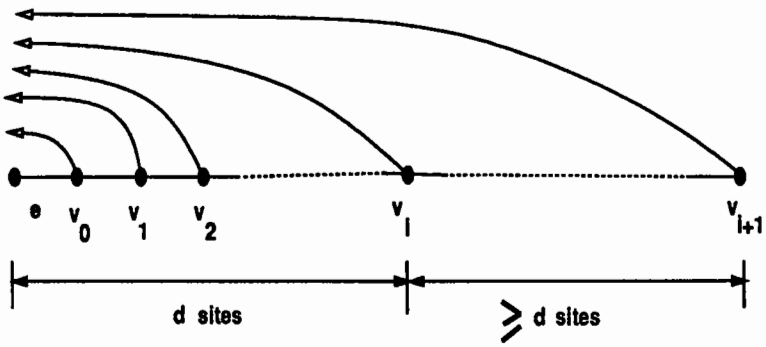

Figure 4.4: P is the Euler tour of the minimum-degree Steiner tree. The active sites that contribute to the congestion of the given edge e in one direction are numbered in the order of occurrence in P as shown. Since the flow is routed along a path in P with the least number of active sites, the number of sites between e and v_i must be at least as many as between v_i and v_{i+1}.

Since $\frac{1}{\tau^*}$ is a lower bound on the value of the minimum-degree problem, it follows from the performance bound in Theorem 4.2.1 that the network we have constructed has a maximum degree that is within $O(\log n)$ of the optimal.

Proving the upper bound on the minimum value of the maximum node-congestion (and hence the minimum-degree) in terms of the inverse of the node-toughness is the essence of our work. We formulate the node-congestion problem as an integer program as described in Section 4.2.2. Let u_I be the optimal solution for this integer program, and let u_F be the optimal solution for the linear relaxation of this program. We shall show that the value of u_F is $O(\frac{\log k}{\tau^*})$. By the results of Raghavan and Thompson given in Theorem 4.2.1, it will then follow that the congestion for our algorithm is $O(\frac{\log k}{\tau^*} + \log n)$.

Let us refer to the linear relaxation of the integer program for the node-congestion problem as LP, and its dual linear program as DP. The optimal value of a linear program equals that of its dual linear program. Hence, it is sufficient to show that the optimal value of DP is within a small factor of $\frac{1}{\tau^*}$. We use the structure of DP to prove this result.

4.3.1 Node-congestion and its Dual Linear Program

We describe the dual linear program DP here. Let $G = (V, E)$ be the initial graph, and let Q be the set of sites. The variables of the program are cost assignments R on the nodes and an induced length assignment ℓ for the edges of the graph. The length ℓ of an edge is at most the sum of the R-values of both its endpoints. The sum of the node-costs over all the nodes is constrained to be at most 1.

$$\text{maximize} \sum_{u \in Q} \mathcal{L}(u) \tag{4.17}$$

$$\text{subject to } \ell(u,v) \leq R(u) + R(v), \text{ for each edge } (u,v) \in E \tag{4.18}$$

$$\sum_{u \in V} R(u) \leq 1 \tag{4.19}$$

$$\mathcal{L}(u) \leq d(v) - d(u),\ \forall v \in S_u, \forall u \in Q \tag{4.20}$$

$$d(v) \leq d(u) + \ell(u,v),\ \forall (u,v) \in E \tag{4.21}$$

$$d(v), R(v) \geq 0,\ \forall v \in V \tag{4.22}$$

$d(v) - d(u)$ denotes the shortest path from u to v under the given length assignment ℓ. $\mathcal{L}(u)$ is the shortest path from u to any of the sites numbered higher than u under the given length assignments. Note that the distance $\ell(u,v)$ for an edge (u,v) is independent of the requirements.

4.3.2 Bounding the Value of the Dual Linear Program: Overview

Suppose we are given an optimal solution to the above linear program. The variables $\ell(u,v)$ assign lengths to the edges. We need to show that, subject to these edge-lengths, the sum of the shortest path distances is small. Our strategy is first to use the length assignments on the edges as costs for the edges. We then consider the tree T with minimum edge-cost connecting up the given set of sites. We show that the sum of the edge-lengths of T is no more than a constant times the inverse of toughness. Then we show that sufficiently short routing paths can be found within this tree; the sum of these path lengths is no more than $O(\log k)$ times the sum of the edge-lengths of T. The second part of the proof resembles that given in Section 4.2.3. We elaborate on these below.

4.3.3 Bounding the Value of the Dual Linear Program: Details

We now prove that the optimal value of the above dual linear program is close to the inverse of the node-toughness. We consider the optimal assignment of edge-lengths obtained by solving the above linear program. We are to show that $\sum_{u \in Q} \mathcal{L}(u)$ small under this length assignment.

For each site $u \in Q$, we shall exhibit a path from u to some site v numbered higher than u such that the sum of these path lengths is $O(\frac{\log k}{\tau^*})$, where k is the number of sites. We shall refer to v as the *parent* of u. The shortest path between any two points cannot be any longer than the path between the points found by our algorithm. It then follows that the optimal value of the dual linear program, and hence u_F, is also $O(\frac{\log k}{\tau^*})$.

4.3.4 Minimum Edge-cost R-join and Node-toughness

We sketch the algorithm below for choosing parents and the associated paths. Consider the original graph under the length assignments given to its edges by the optimal solution of the linear program. We interpret the length of an edge as its cost. We now consider the uniform R-join with the minimum edge-cost spanning all the sites. We first show that this R-join has small total edge-cost with respect to the inverse of toughness. We then show how to choose parents for the sites such that the sum of the lengths from the sites to their parents *using paths in the tree* is a small factor of the cost of the tree.

Lemma 4.3.1 *Consider a feasible edge-length assignment satisfying the constraints of the linear program given in (4.17). There is a network connecting all the sites whose sum of lengths is at most* $\frac{8}{\tau^*} + 8$.

Proof: In Section 4.1.2, we set up a linear program 4.1 whose optimum value is close to the reciprocal of toughness. The key observation is that a feasible length assignment for the linear program in (4.17) also serves as a feasible cost assignment for the linear program in (4.1). By 4.1.2, the maximum value of a packing of cuts under this cost assignment is at most $\frac{4}{\tau^*} + 4$. By Theorem 2.3.4, a network with minimum edge-cost has cost at most twice the value of a maximum packing of cuts. Hence the lemma follows. □

4.3.5 Choosing Parents

Let T be a minimum-cost Steiner tree with total edge-length L. Consider an Euler tour P of T. The length of P is at most twice the total length of the tree, and hence at most $2L$. We think of the Euler tour as a path. A node of the tree can occur multiple times in this path. Of the multiple copies of a site, we mark exactly one copy as *active*. For each active site, we shall choose a higher numbered active parent. The path in P from a site to its parent is a unique subpath of P.

The *parent-choosing* algorithm: The algorithm for choosing parents is simple. For a site v, we choose a parent w such that w is numbered higher than v and the number of active sites in the path P from v to w is minimum. Ties are broken arbitrarily.

Notice that the algorithm to find a parent for each site is identical to that in Section 4.2.3. From the proof in Section 4.2.3, it follows that each edge in the tree is used $O(\log k)$ times in paths from active sites to their parents. Thus, the sum of the lengths of these paths from the active sites to their parents is at most $O(\log k)$ times the length of P, and hence at most $O(L \log k)$.

4.3.6 Performance Guarantee: Putting it all together

To recapitulate, the proof for the performance guarantee is thus broken up into the following steps.

1) The value of minimum-degree is at most the minimum (of the maximum) integral node-congestion.

2) The value of the minimum integral node-congestion is close to its fractional optimal solution (by results of Raghavan and Thompson [132] - see Theorem 4.2.1).

3) The value of the fractional optimal solution of the linear program equals that of its dual by linear programming duality. This value is at most the length of the paths from sites to their chosen parents.

4) We treat the length assignments of the dual linear program as edge costs. We consider the minimum edge-cost R-join. Let its cost be L. The parents are chosen for each of the sites in the original graph such

that the sum of the path lengths between the sites and their parents is at most $O(\log k)$ times L, where k is the number of sites.

5) The cost of L of the minimum edge-cost R-join is within a factor of 8 of the inverse of the node-toughness (Lemma 4.3.1). This proof requires Lemma 4.1.2 and Theorem 2.3.4.

We showed in the introduction that the value of the minimum-degree is at least $\frac{1}{\tau^*}$, while Steps 1 through 5 in the above analysis show that the minimum-degree is at most $O(\frac{\log k}{\tau^*})$. This implies that the degree of the network constructed through the algorithm is within a factor $O(\log k)$ of the optimal. This gives a proof of our claim in Theorem 2.4.1 different from the simple proof given in Section 4.2.3. Moreover, it also shows that the values of $\frac{1}{\tau^*}$ and minimum-degree are within a small multiplicative factor of each other. This implies Theorem 2.4.2.

4.4 Application of Toughness: Dissociation Number

In this section, we give an application of our toughness algorithm for approximating the dissociation number of a graph. Recall that the dissociation number is the minimum number of nodes to remove from a graph $G = (V, E)$ so as to leave only isolated nodes and edges. We shall term the set of nodes to be removed as the minimum *dissociation set.* The main result of this section is to show that we can approximate this dissociation set to within a $O(\log^2 n)$ factor, where n is the number of nodes in the graph. We shall thus prove this claim stated in Theorem 2.4.4.

4.4.1 The Algorithm

The algorithm proceeds in a series of steps. At any step of the algorithm, we have a set of graph components that were formed by removing some of the nodes of the original graph and their associated edges. If there is no component in the set containing more than two nodes, then we output the set of nodes that have been removed so far, and our algorithm terminates. Else, we consider each of the components containing at least three nodes. We estimate its toughness using our algorithm of Theorem 2.4.3. We choose the component with minimum toughness, and remove the nodes forming its approximately minimum toughness cut. We now proceed to the next step. The algorithm starts out with just the original graph in the first step, and ends when all the components remaining are either single nodes or edges.

4.4.2 Notion of Star-cuts

We shall now prove the performance guarantee of our algorithm for the dissociation number. For proving the performance, it will be useful to useful to view the above process in the following manner. Instead of removing nodes, we think of removing only its incident edges. Thus each of the nodes that would have been removed is now a (trivial) component on the removal of its incident edges.

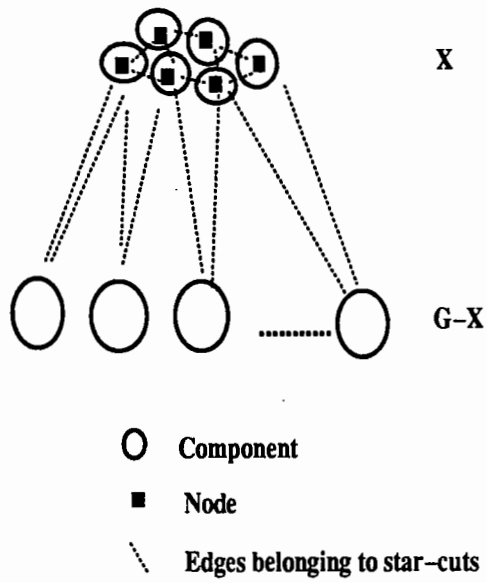

Figure 4.5: Removing the star-cut around a node turns the node into a trivial component. On removing the star-cuts around each of the nodes in a node-set X, we obtain $|X| + \gamma(G - X)$ components.

We shall call the set of edges incident to a node as the star-cut around the node. If the removal of a node set X from G yields $\gamma(G - X)$ components, then the removal of the star-cuts around the nodes in X yields $\gamma(G - X) + |X|$ components (see Figure 4.5). Recall that our algorithm of Section 4.1.1 finds a set of nodes $X \subseteq V$ that has approximately the maximum ratio of $\frac{\gamma(G-X)}{|X|}$. Hence, this node set X also approximately maximizes the ratio $\frac{\gamma(G-X)+|X|}{|X|}$ with a further loss of a factor of 2 in the performance guarantee, since $\frac{\gamma(G-X)}{|X|}$ must be at least 1. The latter ratio is nothing but the ratio of the number of components yielded by the star-cuts around X and the size of X. From Theorem 2.4.3 and the discussion above, the star cuts around X trivially achieve a factor of 16 approximation guarantee for this ratio. However, a more careful analysis yields an approximation guarantee of only 5, the details of which are not important to our discussion here.

Lemma 4.4.1 *Our algorithm of Section 4.1.1 finds a node-set $X \subset V$ such that $\frac{\gamma(G-X)+|X|}{|X|}$ is approximately optimal. The performance guarantee is a factor of* 5.

4.4.3 Performance Guarantee: Analysis

Let G_i be the graph at the beginning of the ith step. We shall call a component *big* if it has at least 3 nodes, and *small* otherwise. Let n_i^{Δ} be the number of nodes belonging to the big components in G_i.

Let our algorithm consist of t steps. We shall partition this sequence of steps into phases in the following manner. If a phase starts at step i, then it ends at a step j when $n_j^{\Delta} \leq \frac{11}{12} n_i^{\Delta}$ for the smallest index $j > i$. The next phase starts at step j. Phase 1 starts at step 1 of the algorithm. Thus each phase consists of the smallest sequence of steps which decreases the count of the nodes belonging to big components by at least a constant

factor of $\frac{11}{12}$. Note that n_1^Δ is at most n, the total number of nodes in the original graph G. Hence the total number of stages is at most $\log_{\frac{12}{11}} n$.

Our proof consists in showing that the total number of star-sets removed in a phase is no more than $O(|D^*| \log n)$, where D^* is the set of star-cuts around the nodes in the optimal dissociation set for the graph. Since there are only $O(\log n)$ phases, this proves the performance guarantee in Theorem 2.4.4.

Let us refer to the number of star-sets removed at a step and a phase of the algorithm by the *cost for the step* and the *cost for the phase* respectively. We then must show the following.

Lemma 4.4.2 *The cost of a phase is at most* $O(\log n)\,|D^*|$.

Before we prove the above lemma, we shall prove some other results on which the proof of this lemma is build. For clear exposition, it will be convenient to number the steps of a phase starting from 1. So from now on whenever we refer to a step i, we shall refer to the ith step in a phase under consideration. For the purpose of the proof we shall think of a phase as starting with *only* the big components of the graph, and breaking it into further components as we proceed with the steps of the phase.

We shall define a potential for each step of a phase, and use this in argument for proving Lemma 4.4.2. However, we need some notation.

Let G_i be the graph at the beginning of the ith step of a phase. G_i, in general, contains big components as well as small components. G_1, however, by our choice contains only big components. We shall refer to the number of components in a graph G by $\gamma(G)$. We shall use C_i to denote $\gamma(G_i)$, and n_i^Δ to denote the number of nodes in G_i belonging to big components. Note that by construction n_1^Δ is the number of nodes in G_1, and that C_1 is at most $\frac{n_1^\Delta}{3}$ since each component of G_1 has at least 3 nodes.

We shall define C^* to be the number of components obtained on removing the star-cuts in D^* from G_1, i.e. $C^* = \gamma(G_1 - D^*)$. We define the potential P_i at the beginning of step i to be

$$P_i = C^* - C_i \tag{4.23}$$

Note that $C^* \leq n_1^\Delta$ trivially, because that is the number of nodes G_1 has. Moreover, $C^* \geq \frac{n_1^\Delta}{2}$, since on removing all the star-cuts in D^*, we are left only with isolated nodes and edges. Since the number of components is strictly increasing with each step, it follows that

$$C_1 < C_2 < C_3 < \ldots$$

$$P_1 > P_2 > P_3 > \ldots$$

Let X_i be the set of star-cuts removed from one of the big components by our algorithm at step i. The cost of the step is then the size of X_i, denoted by $|X_i|$. We shall show that the algorithm achieves a decrease in potential at every step that can be related to the cost for the step.

Lemma 4.4.3 $|X_i| \leq 20\,|D^*|\,\frac{P_i - P_{i+1}}{P_i}$

Proof: Let us denote by G_i^{Δ} and G_i' respectively the graphs consisting of only the big components of G_i and the components consisting of exactly two nodes. The proof proceeds by showing that there exists a set of star-cuts X^* around the nodes in G_i^{Δ}, such that

$$\frac{\gamma(G_i^{\Delta} - X^*) + |X^*|}{|X^*|} \geq \frac{P_i}{2\,|D^*|} \tag{4.24}$$

It then follows by a simple averaging argument that there is a big component such that the subset of X^* belonging only to this component also achieves the above ratio. By Lemma 4.4.1, the star-cuts around X_i removed at a step achieve a ratio within a factor of 5 of the above ratio. We thus have

$$\frac{\gamma(G_i^{\Delta} - X_i) + |X_i|}{|X_i|} \geq \frac{P_i}{10\,|D^*|} \tag{4.25}$$

By construction, the star-cuts removed at the step breaks one of the big components into $C_{i+1} - C_i$ components. It then follows that $\gamma(G_i^{\Delta} - X_i) + |X_i|$ equals $C_{i+1} - C_i + 1$. Substituting this into (4.25) we get

$$\frac{C_{i+1} - C_i + 1}{|X_i|} \geq \frac{P_i}{10\,|D^*|} \tag{4.26}$$

However, note that

$$\frac{C_{i+1} - C_i + 1}{|X_i|} = \frac{P_i - P_{i+1} + 1}{|X_i|}, \quad \text{by (4.23)} \tag{4.27}$$

$$\leq \frac{2(P_i - P_{i+1})}{|X_i|} \tag{4.28}$$

The last inequality follows from the fact that $P_i - P_{i+1}$ is at least one, and hence $(P_i - P_{i+1} + 1)$ is at most $2(P_i - P_{i+1})$. On substituting (4.28) into (4.26), we get the lemma.

It hence remains to show (4.24). Recall that on removing the star-cuts around nodes in D^* from the graph G_1, the total number of components is at least C^*. Hence the same must also hold for G_i. Since G_i has C_i components, the number of additional components generated on removal of the star-cuts around nodes in D^* is at least $C^* - C_i$. Moreover, the additional components can only be generated in components with at least two nodes, i.e. components belonging either to G_i^{Δ} or G_i'. Let the *additional* components generated in G_i^{Δ} and G_i' respectively be A_i^{Δ} and A_i'. Then we have

$$A_i^{\Delta} + A_i' \geq C^* - C_i = P_i$$

Let the set of nodes of D^* belonging to G_i^{Δ} and G_i' be $D^{*\Delta}$ and $D^{*\prime}$ respectively. We observe that each star-cut in $D^{*\prime}$ can only add one extra component in a graph with exactly two nodes, hence $A_i' \leq |D^{*\prime}|$, and we get

$$A_i^{\Delta} + |D^{*\prime}| \geq P_i$$

Since $|D^{*\prime}| \leq |D^{*\prime}| + \left|D^{*\Delta}\right|$, on dividing the above equation by $|D^{*\prime}| + \left|D^{*\Delta}\right|$, we get

$$\frac{A_i^\Delta}{|D^{*\prime}| + \left|D^{*\Delta}\right|} + 1 \geq \frac{P_i}{|D^{*\prime}| + \left|D^{*\Delta}\right|} \tag{4.29}$$

$$\geq \frac{P_i}{|D^*|}, \quad \text{since } \left|D^{*\prime}\right| + \left|D^{*\Delta}\right| \leq |D^*| \tag{4.30}$$

Since $\left|D^{*\prime}\right| \geq 0$, it follows that

$$\frac{A_i^\Delta}{\left|D^{*\Delta}\right|} + 1 \geq \frac{P_i}{|D^*|} \tag{4.31}$$

Since each star-cut around a node turns the node into a trivial component, we get $A_i^\Delta \geq \left|D^{*\Delta}\right|$. We then have $\frac{A_i^\Delta}{\left|D^{*\Delta}\right|} + 1 \leq 2\frac{A_i^\Delta}{\left|D^{*\Delta}\right|}$, and we get

$$\frac{A_i^\Delta}{\left|D^{*\Delta}\right|} \geq \frac{P_i}{2\,|D^*|} \tag{4.32}$$

Recall that A_i^Δ is the number of additional components, and hence is at most the total number of components, generated from G_i^Δ on removal of the star-cuts around $D^{*\Delta}$. Thus on setting X^* to $D^{*\Delta}$, (4.24) is proved. □

Using Lemma 4.4.3, we shall show that the cost of the steps within a phase is small. However, we require one more lemma.

Lemma 4.4.4 *For a step i in a phase, if $P_i \leq \frac{P_1}{12}$, then $n_i^\Delta \leq \frac{11}{12} n_1^\Delta$.*

Proof: By definition of C_i, we get $P_i \leq C^*$. Since C^* is at most n_1^Δ, we get $P_i \leq n_1^\Delta$. Moreover, since $P_i \leq \frac{P_1}{12}$, it follows that

$$P_i \leq \frac{n_1^\Delta}{12} \tag{4.33}$$

Substituting (4.33) and $C^* \geq \frac{n^\Delta}{2}$ into (4.23), we get

$$C_i \geq \frac{n_1^\Delta}{2} - \frac{n_1^\Delta}{12} = \frac{5}{12} n_1^\Delta \tag{4.34}$$

The total number of big components in G_i can be at most $\frac{n_1^\Delta}{3}$, since each such component must have at least 3 nodes. However by (4.34), the total number of components in G_i is at least $\frac{5}{12} n_1^\Delta$, and hence the rest of the components must be small components. Since there are $\frac{5}{12} n_1^\Delta - \frac{n_1^\Delta}{3} = \frac{1}{12} n_1^\Delta$ of them, and each of them contains at least one node, it follows that the maximum number of nodes belonging to big components can be at most $n_1^\Delta - \frac{1}{12} n_1^\Delta$. That proves the lemma. □

Now we are ready to prove the main lemma of this section.

Proof of Lemma 4.4.2: We can rewrite Lemma 4.4.3 as

$$P_{i+1} \leq P_i(1 - \frac{|X_i|}{20\,|D^*|}) \tag{4.35}$$

Let the algorithm terminate after k steps. On unfolding the above recurrence, we get

$$\frac{P_{k+1}}{P_1} \leq \prod_{i=1}^{k}(1 - \frac{|X_i|}{20\,|D^*|}) \tag{4.36}$$

$$\leq \prod_{i=1}^{k} e^{-\frac{|X_i|}{20|D^*|}} \tag{4.37}$$

$$= e^{-\frac{1}{20|D^*|}\sum_{i=1}^{k}|X_i|} \tag{4.38}$$

Taking log of both sides and rearranging, we finally get

$$\sum_{i=1}^{k}|X_i| \leq 20\,|D^*| \ln \frac{P_1}{P_{k+1}} \tag{4.39}$$

There are two cases for P_{k+1}.

1) $P_{k+1} > 0$. Since the potential is always integral, and $P_1 \leq n_1^{\Delta} \leq n$, it follows that

$$\sum_{i=1}^{k}|X_i| \leq 20\,|D^*| \ln P_1 \leq 20\,|D^*| \ln n$$

and the lemma is proved.

2) $P_{k+1} \leq 0$. Then by Lemma 4.4.4, $P_k > \frac{P_1}{12}$, since the algorithm had not achieved the desired reduction in size of the big components by the $(k-1)$st step. We then write

$$\sum_{i=1}^{k}|X_i| = \sum_{i=1}^{k-1}|X_i| + |X_k|$$

The first term is at most $\frac{|D^*|}{20}O(\log n)$ by case 1. We will show that the second term is $O(|D^*|)$, and hence the lemma follows even in this case.

We estimate the value of $|X_k|$ using Lemma 4.4.3.

$$|X_k| \leq 20\,|D^*|\,\frac{P_k - P_{k+1}}{P_k} \tag{4.40}$$

Note that $C^* \geq \frac{n_1^{\Delta}}{2}$, and $C_1 \leq \frac{n_1^{\Delta}}{3}$. Hence,

$$P_1 = C^* - C_1 \geq \frac{n_1^{\Delta}}{2} - \frac{n_1^{\Delta}}{3} \geq \frac{n_1^{\Delta}}{6} \tag{4.41}$$

Since $P_k > \frac{P_1}{12}$, it follows from (4.41) that

$$P_k > \frac{n_1^{\Delta}}{72}$$

Moreover, since the maximum number of components possible is n_1^{Δ}, this is also the maximum potential possible. Thus

$$P_k - P_{k+1} \leq n_1^{\Delta} \tag{4.42}$$

Substituting (4.41) and (4.42) into (4.40), we get$X_k = O(|D^*|)$.

□

4.5 Application of Toughness: Generating Components

In the previous section we described the algorithm for finding an approximate dissociation set. We can apply the same algorithm for breaking up the graph into multiple components by removal of star-cuts. We can show that if at the end of a step, our algorithm has generated k components, then the total number of star-cuts removed so far is within $O(\log n)$ factor of the optimal number of star-cuts that must be removed from the original graph to generate k components.

4.6 Conclusions

We gave an approximation algorithm for the problem of connecting a given set of sites with a network whose maximum degree is the smallest. We related its value to the inverse of toughness in the graph. As a substep, we also gave an approximation algorithm for finding an optimal toughness cut in a graph. We showed how to apply the toughness algorithm repeatedly to break up the graph into multiple components by removing approximately a minimum number of nodes.

Chapter 5

Elimination Ordering

Finding good elimination orderings are at the very heart of solving sparse linear systems. An ordering dictates the amount of storage and the amount of time required to solve the problem using Gaussian elimination. Over the years, many heuristics for finding an elimination ordering with small storage requirement have been developed. Notable among them are the minimum-degree algorithms, nested dissection algorithms, and profile limiting algorithms. An introduction to these and many other heuristics can be found in the books by George and Liu [57], and Duff and Reid [36]. Other heuristics from the area of evidence propagation in belief networks are also known in the literature [89]. However, in spite of the numerous heuristics developed for the problem, none of them provides a non-trivial performance bound for every graph. We give for the first time an algorithm that finds an ordering for symmetric positive definite systems of equations which *provably* requires close to minimum storage. The storage requirement for an ordering is measured by its fill, the total number of elements in the matrix that were either non-zero, or turned non-zero during the course of the algorithm. Our ordering is a nested dissection based ordering.

In fact, we prove more. We show that our algorithm is provably good not only in terms of its fill but also in the total multiplication count. The multiplication count dictates the total time required to solve the problem and hence is of much independent interest. Small fill in general leads to small operation count. However, there is no direct correspondence between an ordering that has the least fill, and an ordering that has the least operation count. In the light of this remark, an ordering that approximately minimizes both these quantities *simultaneously* deserves merit.

Yet another criterion of judging an elimination ordering is its *height*. The height of an elimination ordering gives the maximum serial dependency in eliminating the variables, and thus is a measure of how quickly the problem can be solved in parallel. Orderings that are known to have low fill do not necessarily have small height. for example, a minimum degree ordering gives zero fill for a chain graph of length n. However, it has height n, which is significantly more than the $O(\log n)$ achievable by a nested dissection ordering. There has been much work done recently in trying to get good orderings for parallel elimination of variables (see [48]).

However, none of the orderings previously known guaranteed small height. Our ordering does. It does so along with provably small fill, and provably small operation count, thus approximately minimizing three very useful quantities *simultaneously.*

In solving a system of linear equations $Ax = b$ with A being a symmetric positive definite matrix, an elimination ordering can be determined solely from the structure of the non-zero elements in A. The structure of A is then interpreted as an incidence matrix of a graph, and the fill-in introduced during the elimination process corresponds to adding edges to this graph. As was shown by Rose [136], the augmented graph at the termination of the elimination process is chordal. The number of edges in the chordal graph corresponds to the total fill for the elimination ordering; the fill is the number of original non-zero elements plus the number of non-zero elements introduced during the elimination process. Hence, finding an elimination ordering that minimizes fill is equivalent to finding a minimum chordal extension of the graph denoted by A. This equivalence is at the heart of our analysis. We give an algorithm for finding an approximately minimum chordal extension of a graph and thus give an algorithm for approximately minimizing fill. All the guarantees in this chapter hence apply only to symmetric positive-definite matrices.

Finding a minimum chordal extension of a graph is NP-complete [165]. We start out by proposing a nested dissection algorithm for the problem, and prove its performance guarantee. We then address the performance guarantees of operation count and height of our elimination ordering.

5.1 Notations

A graph G and a matrix A are said to be *corresponding* if $(u, v) \in G$ if and only if $A_{uv} \neq 0$. Very often we shall refer to a matrix as a graph, meaning to refer to the graph corresponding to the matrix. The position of a node v in an elimination ordering α will be denoted by $\alpha(v)$. Throughout this chapter, we shall use G^*_α to refer to the chordal extension of a graph G for a given ordering α. Moreover, we shall refer to the optimal chordal extension of a graph G by G^*; the optimality criteria for the extension will be clear from the context. By $|G|$ for a graph G, we shall refer to its number of edges.

5.2 Graph Chordalization

Chordal graphs are a subclass of perfect graphs and have a perfect elimination ordering. The idea of a perfect elimination ordering itself is derived from the gaussian elimination process. An elimination ordering of G is perfect, if the fill-in induced by the ordering on the matrix corresponding to the graph is zero. One of the simplest characterizations of a chordal graph is that every cycle of length at least four in the graph has an edge between two non-consecutive nodes of the cycle. A good discussion can be found in Golumbic's book [69]. Apart from Gaussian elimination, study of chordal graphs has applications in pedigree analysis, and evidence propagation in belief networks (see [89]).

Graph chordalization is the problem of extending an input graph G to a chordal graph of minimum size. Every chordal extension of a graph can be completely specified by an ordering of the nodes of the graph; the ordering is a perfect elimination ordering for the chordal extension of the graph. For the matrix corresponding to the graph, this ordering of nodes also corresponds to an elimination ordering of the corresponding variables.

To construct the minimal chordal extension of a graph G from a given ordering α of its nodes, we mimic the elimination process in terms of the following operations on G [136]. Let G_i be the graph at the end of step i, and let v be the ith node in the ordering. At step $i+1$, we augment G_i to G_{i+1} by turning all the neighbors of v numbered higher than i in the ordering, into a clique. G_0 is set to be the original graph G. G_n is the desired unique chordal graph corresponding to the ordering α, where n is the number of nodes in the graph.

An ordering of the nodes of a graph is hence sufficient to specify a chordal extension of a graph. We employ this approach in presenting our algorithm for approximately minimum chordalization.

5.3 Our Algorithm

5.3.1 Graph Separators

Before we give the ordering algorithm, we need some related background on an essential ingredient to our algorithm, namely balanced node separators. Recall that a set of nodes X in a graph $G = (V, E)$ is called an *f-balanced node-separator* for some fraction $f < 1$, if no connected component of $G - X$ is of size more than the fraction f of $|V|$. No polynomial-time algorithms are known for finding an f-balanced node separator of minimum size for any constant f. However, using the technique of Leighton and Rao [96], one can find an approximately minimum-sized balanced node separator. This was also shown by Makedon and Tragoudas [115].

Lemma 5.3.1 ([96, 115]) *For a graph (directed or undirected), there exists a polynomial-time algorithm to find a $\frac{2}{3}$-balanced node-separator of size within $O(\log n)$ factor of the optimal $\frac{1}{2}$-balanced node-separator.*

5.3.2 Elimination Ordering Algorithm

Our ordering algorithm is a nested dissection algorithm that is based on a recursive decomposition of the graph.

Given a graph $G = (V, E)$ with n nodes to be numbered in the range $[a, b]$, where $b = a + n - 1$, we proceed as follows. If $n = 1$, we number the node a. Else, we find an approximate $\frac{2}{3}$-balanced node separator X for G using the algorithm in Lemma 5.3.1. We number the vertices in the separator from $b - |X| + 1$ to b in any order. Let there be k connected subgraphs $A_1, \ldots, A_k$ of sizes $n_1, \ldots, n_k$ left on removing X from G. 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 $i \in [1, k]$.

We shall refer to this ordering as α for the rest of this chapter.

5.4 Separator Tree

In establishing the performance guarantees, it will be convenient to relate our ordering to a *separator tree* representation of the recursive decomposition process. Our nested dissection algorithm for a graph G naturally defines a separator tree. The vertices in the balanced node-separator X belong to the root of the tree. The children of the root are subtrees formed recursively for each of the connected components in $G - X$.

The nodes of the original graph will be referred to as *vertices* to distinguish them from the nodes of the separator tree. Our algorithm thus defines an elimination ordering of the vertices of the original graph that is consistent with a postorder traversal of the nodes of the separator tree. However, the algorithm orders the vertices within a tree node arbitrarily.

When we refer to a separator, we refer to the set of vertices belonging to a node in the tree. Let us denote the separator containing a vertex v by X_v and the set of vertices belonging to any of the nodes in the subtree rooted at X_v by T_v. The separator tree naturally defines an ancestor relation on the nodes of the tree. We shall also consider a node to be a trivial ancestor of itself. We shall say that a node u is a *strict ancestor* of a node v if u is an ancestor of v but $u \neq v$.

Let us define the *level of a node* v in the tree as the distance of v from the root, and denote it by $level(v)$. By a level l in the tree, we refer to all the nodes at level l. By the level of a vertex we shall refer to the level in the separator tree of the node it belongs to. The *depth of a tree* refers to the maximum level of any node in the tree. We claim that the depth of the tree is small.

Lemma 5.4.1 *The depth of the separator tree is at most* $O(\log n)$.

Proof: On removing a balanced separator from a graph with n vertices, each of the pieces has at most $\frac{2}{3}n$ vertices. Hence the graph size decreases exponentially with the increase in recursion depth of the nested dissection algorithm. The depth of the separator tree is then at most $\log_{\frac{3}{2}} n$. □

We shall use the following simple observation many times, and hence we state it here.

Propostion 5.4.1 *Let* $x_1, \ldots, x_p$ *be the separators at some level of the separator tree. The vertex sets* $T_{x_1}, \ldots, T_{x_p}$ *are disjoint.*

5.5 Performance Guarantee: Number of Edges

In this section, we establish the performance guarantees for the number of edges and hence the fill for our elimination ordering.

5.5.1 A Lower Bound

We shall first establish a lower bound on the number of edges in the optimally filled graph G^*. In Lemma 2.5.1, we showed a lower bound for $|G^*|$ in terms of the sizes of the separators at any level of the separator

tree. However, we had assumed that we had an optimal separator algorithm. We now relax that restriction, and derive a similar result using the nested dissection tree built with the $O(\log n)$ separator approximation algorithm of Leighton and Rao (see Lemma 5.3.1).

Lemma 5.5.1 *Let $X_1, \ldots, X_p$ be the separators at any level of the nested dissection separator tree. The size of the optimal chordal extension is $\Omega\left(\frac{\sum_{i=1}^{p} |X_i|^2}{\log^2 n}\right)$.*

Proof: Let G_i^* be the subgraph induced in G^* by the vertices belonging to the subtree rooted at X_i. By Theorem 2.5.6, G_i^* has a $\frac{1}{2}$-balanced separator of size at most $\sqrt{2\,|G_i^*|}$. Let this separator be X_i^*. Then we have

$$\begin{aligned} \sum_{i=1}^{p} |X_i^*|^2 &\leq 2 \sum_{i=1}^{p} |G_i^*| \\ &\leq |G^*| \end{aligned} \tag{5.1}$$

The second inequality follows from the disjointness of the subgraphs G_i^* (see Proposition 5.4.1). Let the graph induced in G by the vertices of G_i^* be G_i. Since the edges of G_i is a subset of the edges of G_i^*, it follows that X_i^* is also a $\frac{1}{2}$-balanced separator in G_i. By construction, the vertex set X_i is a $\frac{2}{3}$-balanced node separator in G_i, and on applying Theorem 5.3.1, we have

$$|V_i| = O(\log n)|X_i^*|$$

which implies that

$$\sum_{i=1}^{p} |X_i|^2 \leq O(\log^2 n) \sum_{i=1}^{p} |X_i^*|^2 \tag{5.2}$$

On substituting (5.1) into (5.2), we get

$$\sum_{i=1}^{p} |X_i|^2 \leq O(\log^2 n)|G^*| \tag{5.3}$$

Hence the lemma follows on rewriting the last equation. □

5.5.2 A Characterization of Chordal Graphs

Our aim is to estimate the number of edges in the chordal graph corresponding to the ordering given by our algorithm. To do so, we need a good characterization of these edges. Earlier we discussed one such characterization by specifying how to extend a graph to be chordal given the elimination ordering of its nodes. However, there is in fact a more direct characterization of these edges. We shall employ this characterization in estimating the total number of edges in the chordal extension resulting from our elimination ordering. This characterization is the following.

Lemma 5.5.2 ([138]) *For a given elimination ordering α, an edge (u,v) is in G^*_α if and only if there is a path $P = \{z_0 = u, z_1, \ldots, z_p = v\}$ in G such that $\alpha(z_i) \leq \alpha(u)$ and $\alpha(z_i) \leq \alpha(v)$.*

Using Lemma 5.5.2 and the structure of the separator tree, we claim the following characterization of the edges in a chordal extension given by our nested dissection ordering.

Lemma 5.5.3 *Let α be a nested dissection ordering, and $w, v \in G$ such that $\alpha(w) \leq \alpha(v)$. An edge (w,v) is in G^*_α only if there exists an edge $(u,v) \in G$ such that X_v is an ancestor of X_w, and X_w is an ancestor of X_u.*

Proof: By Lemma 5.5.2, we know that if the edge (w,v) exists in G^*_α, then there is a path $P = \{w = z_0, z_1, \ldots, z_p = v\}$ from w to v such that all the vertices in the path are ordered before w and v. We claim this implies that all the vertices in P belong both to T_w and T_v. We prove it by contradiction. For contradiction, let us assume that such is not the case, and that there is a vertex $z_i \in P$ such that $z_i \notin T_w$. Since X_w is a node-separator, any path from a vertex in T_w to a vertex not in T_w must contain a vertex belonging to a strict ancestor of X_w. But then such a vertex will be numbered higher than w, since the numbering is consistent with a post-ordering of the three nodes. By our assumption of the path P, this cannot be true. Thus each of the vertices on the path P belongs to T_w. A similar argument shows that each of these vertices also belongs to T_v.

Thus X_w and X_v are ancestors of X_{z_i} for every $0 \leq i \leq p$. In particular, X_w and X_v are ancestors of $X_{z_{p-1}}$, and the edge (z_{p-1}, v) exists in G. The only way both X_w and X_v can be ancestors of another separator, is if one of them is an ancestor of the other. Since $\alpha(v) > \alpha(w)$, it follows that X_w must be an ancestor of X_w. Hence the lemma holds. □

5.5.3 An Upper Bound: Small Degree Graphs

Now we shall establish an upper bound on the number of edges in the the chordal graph for the ordering given by our nested dissection algorithm. We shall count the edges to a vertex v from any of the vertices numbered lower than v. For that, we define the notion of an associated tree for each vertex.

The *associated tree* for a vertex v belonging to a separator X is constructed as follows. Let $v_1, \ldots, v_k$ be the neighbors of v such that $level(v_i) \geq level(v)$, for $1 \leq i \leq k$. Let X_i be the separator containing v_i. The associated tree for v is the smallest subtree rooted at X containing each of the separators $X_1, \ldots, X_k$.

Lemma 5.5.3 implies that for every edge $(w,v) \in G^*_\alpha$ where $\alpha(v) > \alpha(w)$, w must belong to the associated tree of v. Thus the total number of edges to v from vertices numbered lower than v in the ordering is at most the number of vertices belonging to all the separators in the associated tree of v. We shall refer to this number for v as the *cost* of v. Thus the total number of edges in our chordal extension is at most the sum of the costs of the vertices.

Theorem 5.5.1 *The total number of edges in the chordal extension yielded by our nested dissection ordering is at most is $O(\sqrt{k}\log^4 n)$ times optimal, where k is the maximum degree of the graph.*

Proof: Let us estimate the sum of the costs of all vertices at a given level l_1 in the tree (see Figure 5.1). Suppose that level consists of separators $X_1, \ldots, X_p$. For $i = 1, \ldots, p$, consider the highest-cost vertex of X_i, and let A_i be the associated subtree for this vertex. For each level l, let $W_l(A_i)$ be the number of vertices in A_i at level l. Then the sum of the costs of vertices at level l_1 is no more than the sum, over all levels l greater than l_1, of the value

$$\sum_{i=1}^{p} |X_i| \cdot W_l(A_i). \tag{5.4}$$

Let A_i have q_i separators $X_{i,1}, \ldots, X_{i,q_i}$ at level l. Since each vertex has a maximum degree of k, it follows that the associated tree of a vertex has at most k leaves. This implies that each level of the tree has at most k nodes, and hence q_i is at most k. Substituting into (5.4), we get

$$\begin{aligned} \sum_{i=1}^{p}\sum_{j=1}^{q_i} |X_i||X_{i,j}| &\leq \sqrt{\sum_{i=1}^{p} q_i|X_i|^2}\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i}|X_{i,j}|^2} \\ &\leq \sqrt{k\sum_{i=1}^{p}|X_i|^2}\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i}|X_{i,j}|^2} \end{aligned} \tag{5.5}$$

where the first inequality follows from the Cauchy-Schwartz inequality, and the second from the fact that $q_i \leq k$. By Lemma 5.5.1 it follows that

$$\sqrt{k\sum_i |X_i|^2} = O(\sqrt{k\,|G^*|}\log n)$$

and similarly

$$\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i}|X_{i,j}|^2} = O(\sqrt{|G^*|}\log n)$$

Thus the right-hand side of (5.5) is $O\left(\sqrt{k}|G^*|\log^2 n\right)$. Summing over all levels l_1 and l, we conclude that there are $O(\sqrt{k}|G^*|\log^4 n)$ edges. □

Our elimination ordering hence yields a chordal graph which has only a polylog factor more edges than the optimal if the maximum degree of the graph is at most a polylog in terms of the number of nodes. This also proves that the fill for such graphs is also provably small. Moreover, many problems in practice, for example finite element problems and linear programming problems, have small degrees and thus our nested dissection ordering is guaranteed to produce small fill.

5.5.4 An Upper Bound: Large Degree Graphs

While the performance bound is polylog for small degree graphs, we cannot claim the same for the unbounded degree graphs. We can, however, claim a non-trivial performance bound which is no worse than a factor of $m^{\frac{1}{4}}\log^4 n$ times the optimal, where m is the number of edges in the graph.

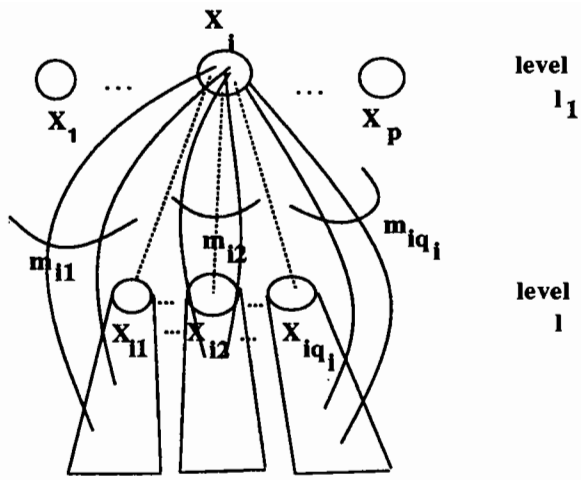

Figure 5.1: Counting the edges in our chordal extension of a graph with large degree.

Theorem 5.5.2 *For an unbounded degree graph G with n vertices and m edges, the total number of edges in G^*_α is $O\left(|G^*|^{\frac{3}{4}}\sqrt{m}\log^{3.5}n\right)$.*

Proof:

Let us first estimate the cost of all the vertices at a given level l_1. Let this level have p separators $X_1, \ldots, X_p$, and let T_i be the tree rooted at separator X_i. Consider another level l greater than l_1 (see Figure 5.1). We will estimate the number of edges in G^*_α between the vertices in X_i and the vertices in T_i. Consider a separator $X_{i,j}$ belonging to T_i at level l, and let $T_{i,j}$ be the tree rooted at this separator. By Lemma 5.5.3, the vertices in $X_{i,j}$ have edges in G^*_α only to those vertices in X_i that have an edge in the original graph to a vertex in $T_{i,j}$. The number of such vertices in X_i is the minimum of $|X_i|$ and $m_{i,j}$, where $m_{i,j}$ is the total number of edges between vertices of X_i and vertices of $T_{i,j}$ in the original graph. Thus the total number of edges between X_i and $X_{i,j}$ in the chordal graph is at most $\min(|X_i|, m_{i,j}) \cdot |X_{i,j}|$, and hence at most $\sqrt{|X_i|\, m_{i,j}} \cdot |X_{i,j}|$. Summing over all the q_i separators belonging to T_i at level l, and again summing over all the p separators at level l_1, we get the total number of edges between vertices at level l_1 and vertices at level l. This value is given by

$$\sum_{i=1}^{p}\sum_{j=1}^{q_i}\sqrt{|X_i|\, m_{i,j}} \cdot |X_{i,j}| \tag{5.6}$$

Once again using Cauchy-Schwartz inequality, (5.6) is at most

$$\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i}|X_i|\, m_{i,j}} \cdot \sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i}|X_{i,j}|^2} \tag{5.7}$$

Let m_i denote the sum $\sum_{j=1}^{q_i} m_{i,j}$. Then $\sum_{i=1}^{p}\sum_{j=1}^{q_i} |X_i|\, m_{i,j}$ equals $\sum_{i=1}^{p} |X_i|\, m_i$. Using Cauchy-Schwartz inequality

$$\begin{aligned}\sum_{i=1}^{p} |X_i|\, m_i &\leq \sqrt{\sum_{i=1}^{p} |X_i|^2} \cdot \sqrt{\sum_{i=1}^{p} {m_i}^2} \\ &= O(\sqrt{|G^*|}\log n \cdot m) \end{aligned} \tag{5.8}$$

Substituting (5.8) and $O\left(\sqrt{|G^*|}\log n\right)$ for $\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{q_i} |X_{i,j}|^2}$ into (5.7), we derive that the total number of edges between a given pair of levels is $O\left(|G^*|^{\frac{3}{4}}\sqrt{m}\log^{1.5} n\right)$. Summing over $O\left(\log^2 n\right)$ choices of the pair of levels l_1 and l, we obtain the result. □

5.6 Performance Guarantee: Number of Multiplications

In this section, we shall establish the performance guarantee for the number of multiplications required by our nested dissection ordering. Since the cost of solving a system of linear equations is dominated by the number of multiplications required for the process, this guarantee reflects the guarantee for the total sequential time required to solve the problem using Gaussian elimination.

5.6.1 A Characterization of Number of Multiplications Required

We shall use the following characterization of the total number of multiplications required by an elimination ordering in terms of the cliques of the filled-in chordal graph. Every vertex v forms a clique in G_α^* with all its neighbors ordered after v. We shall refer to this clique as the *associated clique* for the vertex, and denote it by C_v. The number of multiplications required to eliminate a variable v is the total number of edges in the clique C_v. Thus the total number of multiplications required to eliminate all the variables in a chordal graph equals the sum of the number of edges in the associated cliques of each node.

5.6.2 A Lower Bound

Consider the case when a chordal graph has a clique of size p. Then for any ordering of variables in the clique, the node numbered i has an associated clique of size i, for every i from 1 to p. Thus the total number of multiplications required to eliminate all the variables in this clique is $\sum_{i=1}^{x} i^2$, which is $\Omega\left(p^3\right)$. By Lemma 2.5.1, since every chordal graph has a $\frac{1}{2}$-balanced clique separator, the following lemma easily follows.

Lemma 5.6.1 *For any chordal graph G^*, if p is the size of its clique separator, then $\Omega\left(p^3\right)$ is a lower bound on the number of multiplications required for any elimination ordering.*

Let M^* be the least multiplication count for any elimination ordering of G. We shall extend Lemma 5.6.1 a step further to relate M^* to the sizes of the separators at any level of the separator tree.

Lemma 5.6.2 *Let a given level in the separator tree have p separators $X_1, \ldots, X_p$. Then $\Omega\left(\frac{\sum_{i=1}^{p} |X_i|^3}{\log^3 n}\right)$ is a lower bound on M^*.*

Proof: Let T_i be the subtree rooted at X_i and G_i^* be the subgraph induced by the vertices of T_i in G^*. Since G_i^* is chordal by Fact 2.5.1, it has a clique separator. Since our separator approximation has guarantee of $O(\log n)$, the optimal clique separator must have size $\Omega\left(\frac{|X_i|}{\log n}\right)$. By Lemma 5.6.1, G_i^* must then require $\Omega\left(\frac{|X_i|^3}{\log^3 n}\right)$ multiplications. Since the subgraphs $G_1^*, \ldots, G_p^*$ are disjoint, it follows that any ordering in G^* must require $\Omega\left(\frac{\sum_{i=1}^{p} |X_i|^3}{\log^3 n}\right)$ multiplications. □

5.6.3 An Upper Bound

We shall now derive an upper bound on the number of multiplications required. Let M be the number of multiplications required for the elimination ordering defined by the algorithm. M is given by the sum over all nodes v of the number of edges in v's associated clique. Thus we can write M as $\sum_v \sum_{e \in C_v} 1$, which is the same as $\sum_e \sum_{v: C_v \ni e} 1$. The contribution of an edge to this sum is the number of vertices containing the edge in their associated cliques. We shall refer to this quantity as the *contribution* of the edge. M is hence the sum of the contributions of the edges in G_α^*. We shall use this characterization along with Lemma 5.6.2 to relate M to M^*.

Theorem 5.6.1 *The number of multiplications required by our nested dissection elimination ordering is $O\left(d\log^6 n\right)$ times optimal, where d is the maximum degree of the graph.*

Proof: The contribution of an edge (u, v) is 1 for each vertex z such that C_z contains the edge (u, v). Without loss of generality, let us assume that $\alpha(v) > \alpha(u)$. Since $(u, v) \in G_\alpha^*$, by Lemma 5.5.3, u must belong to the associated tree of v. Since u, v and z belong to the clique C_z, C_z must contain the edges (z, v) and (z, u). Since $\alpha(z) < \alpha(v)$, the presence of the edge (z, v) implies that z must also belong to the associated tree of v. Similarly, the fact that (u, v) is in C_z implies that z must belong to the subtree rooted at u. Thus the only vertices that can contribute to the edge (u, v) are those which belong to the associated tree of v and also belong to the subtree rooted at u. Note that the latter implies that the level of such a vertex is at least as much as that of u.

Our approach in counting the total number of multiplications is the following. We consider all the edges in G_α^* that go between two given levels. For each edge we count the number of vertices in a given third level which contain the edge in its associated clique. We show that this count over all the edges between two levels is at most $O(d\log^3 nM^*)$. Since there are $O(\log^3 n)$ choices of the three levels under consideration, the total number of multiplication is $O(d\log^6 nM^*)$, and we get the lemma.

So let us consider three levels in the separator tree l_1, l_2, and l_3 such that $l_3 \geq l_2 \geq l_1$. Our aim is to count for each edge (u, v) between a vertex v in level l_1 and a vertex u in level l_2, the total number of vertices in the

level l_3 that contain (u,v) in their associated clique. Let this quantity be called M'. M' can be written as

$$M' = \sum_{v \in \text{level } l_1} \sum_{u \in \text{level } l_2, (u,v) \in G_\alpha^*} \sum_{z \in \text{level } l_3, C_z \ni (u,v)} 1 \tag{5.9}$$

We want to estimate M'. Let us denote by M_v the sum $\sum_{u \in \text{level } l_2,(u,v) \in G_\alpha^*} \sum_{z \in \text{level } l_3, C_z \ni (u,v)} 1$ for a vertex v. Let $X_1, \ldots, X_q$ be the separators at level l_1, and let v_i denote the vertex v in X_i for which M_v is maximum. Then we can rewrite (5.9) as

$$M' = \sum_{i=1}^{q} \sum_{v \in X_i} M_v \tag{5.10}$$

$$\leq \sum_{i=1}^{q} \sum_{v \in X_i} M_{v_i} \tag{5.11}$$

$$= \sum_{i=1}^{q} |X_i| M_{v_i} \tag{5.12}$$

Let us now estimate the value of M_{v_i}. Let A_{v_i} denote the associated tree of v_i. Let the separators in A_{v_i} at level l_2 be $X_{i1}, \ldots, X_{iq_i}$. Each of the edges of v_i to level l_2 must have a vertex in A_{v_i} as its endpoint. Consider all the edges between v_i and the vertices of the separator X_{ij}. There are a maximum of $|X_{ij}|$ such edges. By the above discussion, any vertex that has any of these edges in its associated clique must belong to the subtree of A_{v_i} rooted at X_i. All such vertices at level l_3 must then belong to one of the separators in the subtree of A_{v_i} rooted at X_{ij}. Let the separators in A_{v_i} at level l_3 be $X_{ij1}, \ldots, X_{ijq_{ij}}$. Then the maximum number of vertices whose associated cliques can contain an edge between v_i and a vertex in X_{ij} is given by $\sum_{k=1}^{q_{ij}} |X_{ijk}|$, and there can be at most $|X_{ij}|$ such edges. Summing over all the separators in A_{v_i} at level l_2, we get

$$M_{v_i} \leq \sum_{j=1}^{q_i} |X_{ij}| \sum_{k=1}^{q_{ij}} |X_{ijk}| \tag{5.13}$$

We can rewrite (5.12) after substituting (5.13) as

$$M' \leq \sum_{i=1}^{q} |X_i| \sum_{j=1}^{q_i} |X_{ij}| \sum_{k=1}^{q_{ij}} |X_{ijk}| \tag{5.14}$$

By using the inequality $\sum_i x_i y_i z_i \leq \sum_i (x_i^3 + y_i^3 + z_i^3)$, we can rewrite (5.14) as

$$\begin{aligned} M' &\leq \sum_{i=1}^{q} \sum_{j=1}^{q_i} \sum_{k=1}^{q_{ij}} |X_i|^3 + \sum_{i=1}^{q} \sum_{j=1}^{q_i} \sum_{k=1}^{q_{ij}} |X_{ij}|^3 + \sum_{i=1}^{q} \sum_{j=1}^{q_i} \sum_{k=1}^{q_{ij}} |X_{ijk}|^3 \\ &= \sum_{i=1}^{q} |X_i|^3 \sum_{j=1}^{q_i} \sum_{k=1}^{q_{ij}} 1 + \sum_{i=1}^{q} \sum_{j=1}^{q_i} |X_{ij}|^3 \sum_{k=1}^{q_{ij}} 1 + \sum_{i=1}^{q} \sum_{j=1}^{q_i} \sum_{k=1}^{q_{ij}} |X_{ijk}|^3 \end{aligned} \tag{5.15}$$

Since each vertex has degree at most d, it follows that the associated tree of each of the vertices has at most d separators at any level. Hence we have, $\sum_{j=1}^{q_i} \sum_{k=1}^{q_{ij}} 1 \leq d, \forall i$, and $\sum_{k=1}^{q_{ij}} 1 \leq d, \forall i, j$. We can then rewrite

(5.15) as

$$M' \leq d \cdot \sum_{i=1}^{q} |X_i|^3 + d \cdot \sum_{i=1}^{q} \sum_{j=1}^{q_i} |X_{ij}|^3 + \sum_{i=1}^{q} \sum_{j=1}^{q_i} \sum_{k=1}^{q_{ij}} |X_{ijk}|^3 \tag{5.16}$$

Note that each of the terms on the right hand side of (5.16) refers to the separators at a given level, and hence we can apply Lemma 5.6.2, and we get

$$M' \leq dO(M \log^3 n) + dO(M \log^3 n) + O(M \log^3 n) \tag{5.17}$$

$$= O(dM \log^3 n) \tag{5.18}$$

As mentioned before, the total number of multiplications is the sum of M' over all the possible choices of l_1, l_2, and l_3. There being $O(\log^3 n)$ such possible choices, the lemma follows. □

The above theorem shows that the performance guarantee of the our nested dissection algorithm is a polylog factor if the degree of the graph is small. As mentioned earlier, low degree graphs form the bulk of the matrices arising in practice.

5.7 Performance Guarantee: Elimination Height

Since problems in numerical analysis are a favourite for parallel machines, it is natural to consider how well one can perform Gaussian elimination in parallel. The amount of parallel time required by an elimination ordering can be characterized by its height. Multiple variables can be eliminated simultaneously in parallel only if the variables do not have any dependencies between them. In the graph representation, in eliminating a vertex v, we update all the neighbors of v that are numbered higher than v. Hence two vertices cannot be eliminated simultaneously if they have an edge between them. If we think of the edges being directed from the vertex with a lower number to the other, then the height of an ordering α, is the longest directed path in the chordal graph G^*_α. Alternate characterizations of the height in terms of the elimination tree is given by Liu [109].

It is not obvious that an elimination ordering that minimizes fill will also minimize other important quantities like fill, or the multiplication count for the ordering. We already discussed the example of a simple line graph, where the minimum degree heuristic is optimal in terms of fill, but has much worse height than a nested dissection ordering. Gilbert [62] has conjectured that there is an ordering that minimizes height and simultaneously approximately minimizes fill to within a constant factor.

Finding an ordering that minimizes height itself is NP-hard [128]. Hence we have to be content with finding an ordering that approximately minimizes height. It turns out that our nested dissection elimination ordering also approximately minimizes height, and thus we obtain an algorithm that simultaneously minimizes fill, number of multiplications, and height. Contrary to our performance bounds for the fill, and multiplication count, the guarantee for the height is independent of the degree of the input graph, and is always a $O(\log^2 n)$ factor of the optimal. We prove this result in this section.

Bodlaendar et al.[10] have simultaneously given essentially the same ordering as ours for achieving approximately minimum height. The problem of finding an ordering with small height has been studied by many researchers in the past and an excellent survey can be found in the article by Heath, Ng, and Peyton (in [48]).

Since the height of an ordering is of concern when solving a system of linear equations in parallel, it will be desirable to obtain the ordering itself in parallel. However, we do not address that issue here. Our implementation of the algorithm at present is sequential,. We use the technique of Leighton and Rao [96] for finding small balanced separators in a graph, and no efficient parallel implementations are known for it. We suspect that their algorithm cannot be parallelized efficiently. However, we hope that other techniques for finding small graph separators will be developed, which will be more amenable to parallel implementations. The issue of generating the elimination ordering itself in parallel has been studied by other researchers (see [48]). However, none of the previously proposed algorithms have yielded any performance guarantees.

5.7.1 A Lower Bound

From the discussion on the height of an elimination ordering, it follows that the height of any elimination ordering for a clique of size m is m. That gives us the following simple lemma.

Lemma 5.7.1 *For any chordal graph G^*, if m is the size of its clique separator, then the height of any elimination ordering must be $\Omega(m)$.*

We can build on the above lemma to get the following result.

Lemma 5.7.2 *If X a separator of maximum size in the separator tree for a graph G, then any elimination ordering for G must have height $\Omega\left(\frac{|X|}{\log n}\right)$.*

Proof: Let V_X be the set of vertices in the subtree of the separator X. If G^* corresponds to the optimal chordal graph with minimum height over all elimination orders, then let the graph induced by V_X in G^* be G_X. G_X has a clique separator of size $\Omega\left(\frac{|X|}{\log n}\right)$ by Theorem 2.5.6, and the performance guarantee of our separator algorithm (see Theorem 5.3.1). This clique size is a lower bound on the height of any elimination ordering by Lemma 5.7.1, and hence the lemma follows. □

5.7.2 An Upper Bound

We shall now show that the height generated by our nested dissection ordering is not too much more than the optimal height.

Consider the separator tree. Let X be the largest separator in the tree. Consider all the separators at each level. One variable from each of the separators can be eliminated simultaneously as there are no direct edges between the variables. Hence the number of parallel elimination steps for eliminating all the variables at a level is no more than the maximum size of the separators at the level. This size is no more than $|X|$ by assumption.

Since the number of levels is $O(\log n)$, the height of the ordering is at most $O(|X| \log n)$. By Lemma 5.7.2, the value of $|X|$ is at most $O(\log n)$ times the minimum height of any elimination ordering. It then follows that the height of our ordering is at most $O(\log^2 n)$ times the minimum height over all orderings. We thus prove our claim of this performance guarantee in Theorem 2.5.4.

5.8 Experimental Results

We now present the results of the experiments with our nested dissection algorithm. Our intention is to back up the provably theoretical performance of our ordering with some experimental data.

We implemented the algorithm for finding our nested dissection ordering. We compare the quality of our results to those obtained from the minimum-degree heuristic implemented by Joseph Liu, and the nested dissection algorithm in SPARSPAK, a publicly available sparse linear package.

The minimum-degree heuristic is by far the most commonly used and acknowledged as the most effective heuristic known for finding good elimination orderings. It has a rich history. It originated from the work of Markowitz in 1957, has undergone many enhancements over the last fifteen years, and has been incorporated in many publicly available codes like MA28, YALESMP, and SPARSPAK. Much statistics regarding the performance of this heuristic and all the enhancements are also available in literature. George and Liu [58] present an excellent survey of the developments and enhancements in the minimum-degree heuristic. They suggest that a minimum-degree heuristic with certain enhancements [105] outperforms other variations of this heuristic. We obtained the latest version of the code implementing this heuristic from Joseph Liu in July '91, and that is what we shall refer to as the minimum-degree code for the purposes of the comparison. We also wanted to compare our nested dissection ordering against an already existing one. The SPARSPAK nested dissection was an ideal choice because of its popularity.

We compared the fill, the total number of multiplications, and the height of our ordering with those obtained by the other two codes for a variety of matrices. These matrices were obtained from the Harwell-Boeing test set of sparse matrices [34, 33]. They are symmetric positive-definite matrices that are derived from real applications in the industry. They have also been extensively used as a test suite by many researchers (see discussion in Section 2.5.5). Many of the matrices that we used came from structural engineering and finite-element analysis problems.

We report our results below. We give the names of the matrices from the Harwell-Boeing collection, and the actual values of the three quantities of interest for the orderings. For the other two codes, we also compute the percentage difference in the values of the three quantities as compared to the values for our ordering. We give the number of non-zero elements in the original matrix is given in the table for reference.

Our fill is roughly the same as (usually within $\pm$ 11%) of the minimum-degree ordering. The height of our ordering is generally better than that of the minimum-degree ordering. The latter however, has better performance in terms of the number of multiplications. Compared to the SPARSPAK nested dissection ordering,

matrix	*# edges*	*Our ordering*	*Minimum Degree*		*SPARSPAK*	
		#	#	% Change	#	% Change
CAN24	160	213	214	+0%	228	+7%
CAN61	557	752	669	-11%	781	+4%
CAN96	768	1895	1856	-2%	2166	+14%
CAN144	1296	1683	1746	+4%	1812	+8%
CAN187	1491	3776	3735	-1%	4067	+8%
CAN229	1777	5502	5883	+7%	7439	+35%
BCSSTK01	400	901	906	+0%	1072	+19%
BCSSTK04	3648	7121	6544	-8%	9414	+32%
BCSSTK05	2423	5022	4524	-10%	5415	+8%
BCSSTK06	7860	22249	20782	-7%	24116	+8%
BCSPWR02	167	212	215	+1%	265	+25%
BCSPWR05	1623	2720	2425	-11%	4557	+67%
DWT193	3439	8556	8155	-5%	9489	+11%
DWT209	1743	4118	3812	-7%	6263	+52%
NOS4	594	1515	1206	-20%	1754	+16%

Table 5.1: Comparison of fill: fill is the total number of elements in the matrix that were either non-zero or became non-zero during the course of elimination

our ordering seems to fare well in all the three criteria.

matrix	*# edges*	*Our ordering*	*Minimum Degree*		*SPARSPAK*	
		#	#	% Change	#	% Change
CAN24	160	1076	895	-17%	1162	+8%
CAN61	557	5794	3757	-35%	6201	+7%
CAN96	768	22983	22360	-3%	30349	+32%
CAN144	1296	13165	14435	+10%	14172	+8%
CAN187	1491	49313	48754	-1%	58165	+18%
CAN229	1777	104839	119890	+14%	184882	+76%
BCSSTK01	400	8688	8893	+2%	11706	+35%
BCSSTK04	3648	201202	144314	-28%	316461	+57%
BCSSTK05	2423	95389	51549	-46%	110254	+16%
BCSSTK06	7860	815252	639199	-21%	993936	+22%
BCSPWR02	167	772	658	-15%	1147	+48%
BCSPWR05	1623	17299	11568	-33%	50220	+190%
DWT193	3439	229852	185812	-19%	291769	+27%
DWT209	1743	60381	45457	-25%	126012	+109%
NOS4	594	15967	8585	-46%	19331	+21%

Table 5.2: Comparison of multiplication count

matrix	*# edges*	*Our ordering*	*Minimum Degree*		*SPARSPAK*	
		#	#	% Change	#	% Change
CAN24	160	9	11	+22%	10	+11%
CAN61	557	14	24	+71%	18	+29%
CAN96	768	28	26	-7%	36	+29%
CAN144	1296	16	18	+12%	20	+25%
CAN187	1491	32	42	+31%	34	+6%
CAN229	1777	48	52	+8%	71	+48%
BCSSTK01	400	25	24	-4%	30	+20%
BCSSTK04	3648	61	86	+41%	72	+18%
BCSSTK05	2423	41	84	+105%	43	+5%
BCSSTK06	7860	97	138	+42%	92	-5%
BCSPWR02	167	5	13	+160%	9	+80%
BCSPWR05	1623	23	31	+35%	56	+143%
DWT193	3439	58	92	+59%	73	+26%
DWT209	1743	32	54	+69%	54	+69%
NOS4	594	24	30	+25%	27	+12%

Table 5.3: Comparison of height

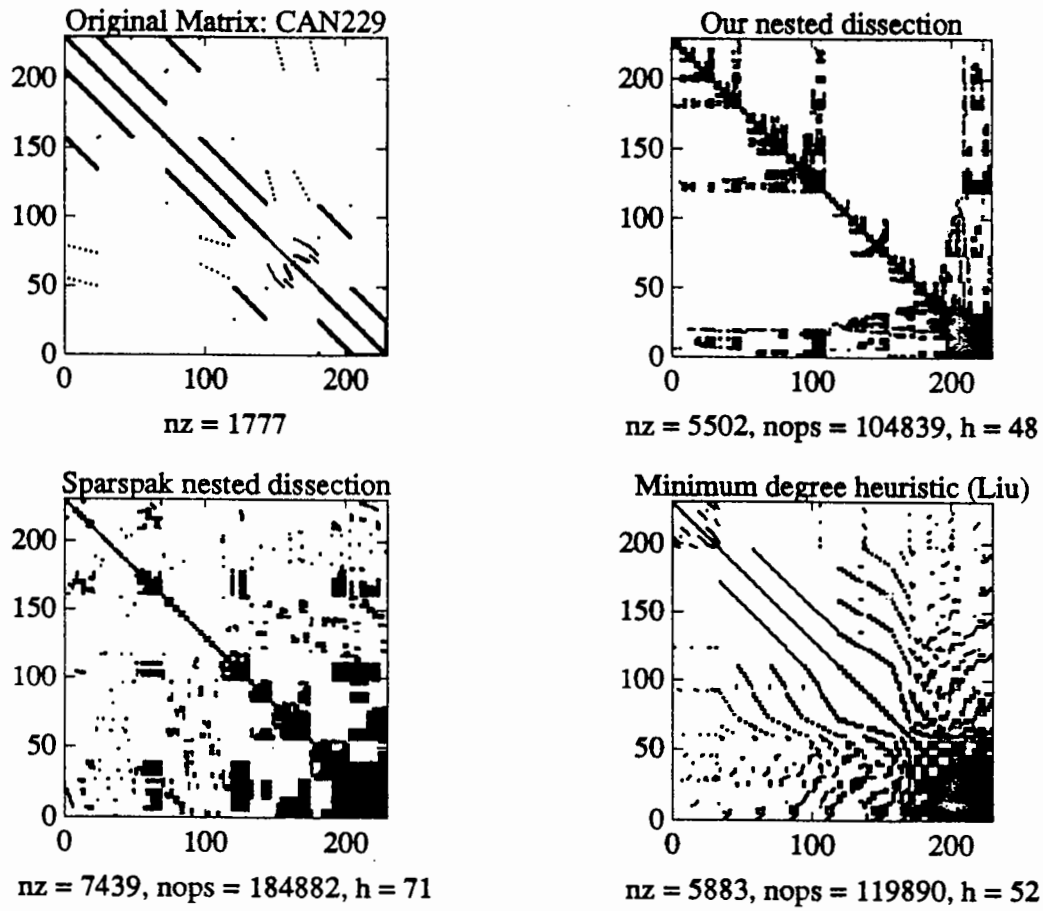

Figure 5.2: The quality of the elimination orderings produced by the three codes are compared. The original matrix from the Harwell-Boeing test-suite and is called CAN229. The fill, the number of multiplications, and the height for the orderings are specified.

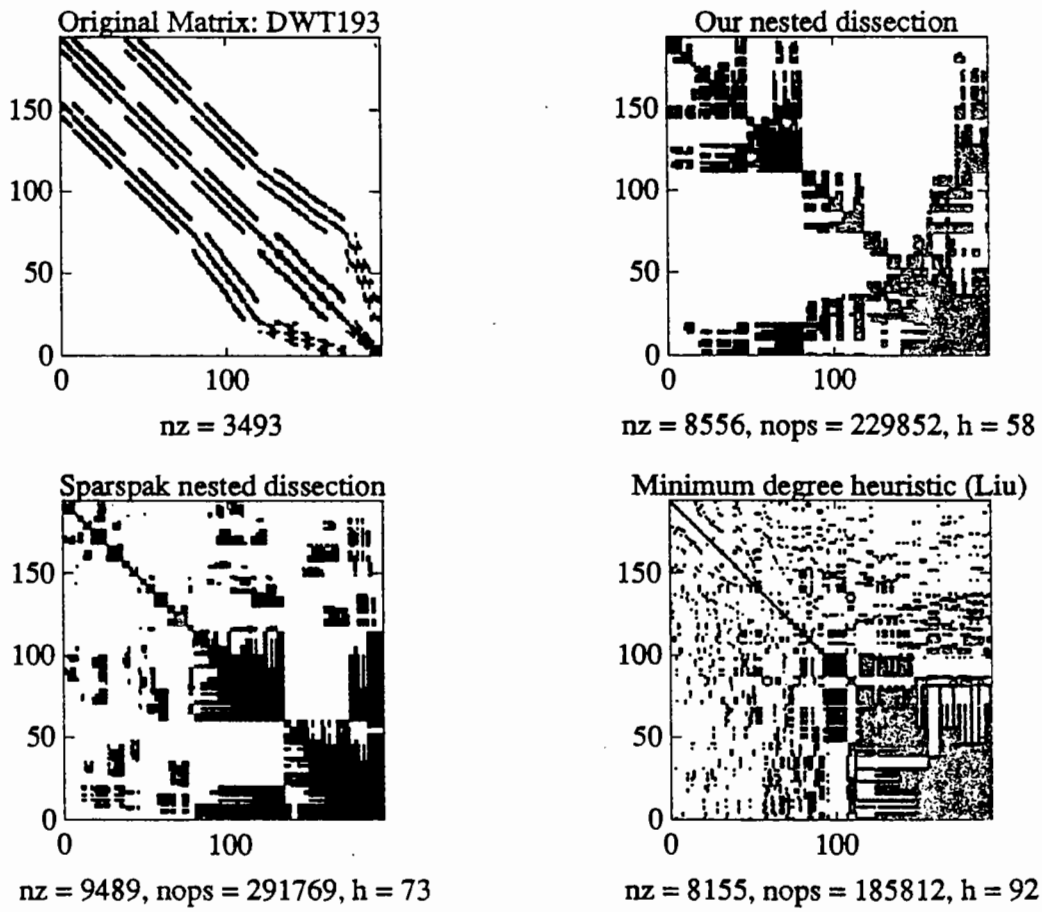

Figure 5.3: The quality of the elimination orderings produced by the three codes are compared. The original matrix from the Harwell-Boeing test-suite and is called DWT209. The fill, the number of multiplications, and the height for the orderings are specified.

Chapter 6

Conclusions and Open Issues

The approximate min-max relations presented in this thesis stand as a testimony to our technique for deriving such relationships using linear programming dualities. Central to this thesis have been the combinatorial problems in requirement-joins and requirement-cuts. We have proved new min-max relations by building on related results that were either previously known or were proved by us. In the process, we provided many new approximation algorithms for these combinatorial quantities. Further, we have been able to apply these approximation algorithms to diverse applications like reliability, and solution of sparse linear systems.

Our study suggests some new directions for further research and many open issues. We list them here.

- Formalizing the technique in Section 1.5: Approximating optimal integer programs by using their linear relaxations have received much attention in the past. Finding characterizations of the kind of problems that are guaranteed to yield good approximations by this method is one of the most challenging open problems in the study of such approximation algorithms.

- Characterizing R-join problems: We presented approximation algorithms for two network design problems, and the algorithms have been quite different for both of them. Is there a characterization of the class of network design problems that can use the same or similar algorithms? Part of this question has already been answered by Goemans and Williamson [73], who generalized our algorithm for the minimum edge-cost R-join to apply to a very general class of network design problems.

- Good Integer program representations of R-joins: There can be many integer program formulations for a combinatorial problem. However, the linear relaxations of only few of them might guarantee an approximation. The problems dealt with in this thesis were formulated as integral flow problems. Is there some common structure in the flow-related integer programs that can be used to characterize a class of integer programs that can be well approximated?

- Fully polynomial approximation algorithms: Our algorithms have a fixed guarantee. Ideally, one should be able to design an algorithm that trades off performance guarantee with the running time.

- Directed R-join problems: All our algorithms are for the undirected versions of the network design problems. Similar problems for the corresponding directed versions remain unanswered.

- Minimum node-cost R-join: Finding an R-join which has minimum total node-cost is analogous to the problem of minimum edge-cost R-join. This problem is a generalization of the set-cover problem. We believed [1] that our result for the minimum edge-cost problem directly translates to the node-cost problem. However, we were mistaken,[1] and the issue remains open.

- Running time for the minimum edge-cost R-join: While our algorithm for the unweighted minimum edge-cost R-join problem has good running time, we think that the running time for the weighted case can be further improved.

- Performance bound for R-multijoin: Our approximation algorithm for the R-multijoin problem has a logarithmic performance ratio. Our analysis follows that of Goemans and Bertsimas [72], who can show that the analysis is tight. Hence, to improve the performance bound, a different approach is needed; one that possibly deals simultaneously with widely varying requirement values.

- Performance bound for the minimum edge-cost R-join: Zelikovsky recently gave a $\frac{11}{6}$-factor approximation algorithm for the *uniform* R-join problem. It is worth investigating if his approach can lead to better approximations for the R-join problem as well.

- Performance bound for the minimum-degree R-join: We can show that the analysis of our approach is tight. However, we think there is room for improving the approximation guarantee for the minimum-degree problem to a constant. That will require a different algorithm and possibly a different approach.

- Weighted minimum-degree R-join: Our algorithm for the minimum-degree R-join extends to the weighted case of the spanning R-join setting. However, the issue of weighted minimum-degree problem even under the uniform R-join setting remains unresolved.

- Weighted node-toughness: Our algorithm for the node-toughness problem does not extend to the case when the nodes are allowed to have arbitrary weights. It will be interesting to have an approximation algorithm for the weighted node-toughness problem.

- Complexity of the maximum $\frac{1}{2}$-packing problem: We do not yet know whether finding the maximum size of a $\frac{1}{2}$-packing of requirement cuts is NP-complete. We guess it is, however, we have not been able to prove the same.

- Running time for finding good balanced separators: The running time of our nested dissection algorithm for finding an elimination ordering directly depends upon the running time of the balanced separator

[1] Communicated to us by Piotr Berman and B. Raghavachari, Pennsylvania State University.

algorithm. For the algorithm to gain acceptance, we must have a faster approximate separator algorithm. Separators have numerous other applications also and hence having fast separator algorithms are of much independent interest.

- Finding in parallel an elimination ordering of small height: Our nested dissection ordering is a good ordering for solving sparse linear systems in parallel. However, our algorithm for finding the elimination ordering itself is inherently sequential at present. That is because no parallel approximation algorithms are yet known for finding balanced separators in a graph. It is of interest to find a parallel algorithm that produces an ordering that has provably small height.

  Finding a parallel algorithm for approximating minimum balanced node separators in a graph is independently of much interest.

- Improving the performance bounds for the ordering problems: The performance guarantees for the fill and the operation counts for our nested dissection ordering depend upon the maximum degree of the graph associated with the coefficient matrix. It is a challenging problem to find a polynomial-time ordering algorithm whose performance guarantees are independent of the degree of the input graph. Even obtaining an ordering algorithm whose performance gurantees are proportional to the *average* degree of the input graph will be a significant theoretical advance.

- Experiments with variants of our nested dissection algorithm: While our nested dissection algorithm seems to perform well in practice, we have not yet experimented with variants of our algorithm. We think that further experience with this algorithm might suggest practical enhancements to the elimination orderings produced by the algorithm.

Bibliography

[1] A. Agrawal, P. Klein and R. Ravi, "When trees collide: an approximation algorithm for the generalized Steiner tree problem on networks," *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*(1991), pp. 134-144.

[2] A. Agrawal, P. Klein and R. Ravi, "How tough is the minimum-degree Steiner tree? A new approximate min-max equality," Technical Report CS-91-33, Brown University (1991).

[3] S. B. Akers, "Routing," In M. A. Breuer, editor, *Design Automation of Digital Systems, Vol. 1: Theory and Techniques*, pp. 283-333. Prentice-Hall Inc. (1972).

[4] Y. P. Aneja, "An integer linear programming approach to the Steiner problem in graphs," *Networks, vol. 10* (1980) 167-178.

[5] A. Balakrishnan, and N. R. Patel, "Problem reduction methods and a tree generation algorithm for the Steiner network problem," Unpublished paper (1985).

[6] M. O. Ball, and G. L. Nemhauser, "Matroids and a reliability analysis problem," *Mathematics of Operations Research, vol. 4* (1979), pp. 132-143.

[7] D. Bauer, S.L. Hakimi, and E. Schmeichel, "Recognizing tough graphs is NP-hard," *Discrete Applied Mathematics, vol. 28* (1990), pp. 191-195.

[8] J. E. Beasley, "An SST-based algorithm for the Steiner problem in graphs," Techn. Rep., Dept. of Man. Science, Imperial College, London (1985).

[9] D. Beinstock, "An algorithm for reliability analysis of planar graphs," WP-20-85-86, GSIA, Carnegie-Mellon University (1985).

[10] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson and T. Kloks, "Approximating treewidth, pathwidth, and minimum elimination tree height," Technical Report CSL-90-01, Xerox Corporation, Palo Alto Research Center (1990).

[11] W. M. Boyce, "An improved program for the full Steiner tree problem," *ACM Transactions on Mathematical Software, vol. 3* (1977), pp. 359-385.

[12] T. B. Brecht, and C. J. Colbourn, "Lower bounds for two-terminal network reliability," CCNG Report E-127, University of Waterloo (1985).

[13] S. K. Chang, "The generation of minimal trees with a Steiner topology," *Journal of the ACM, vol. 19* (1972), pp. 669-711.

[14] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA. (1976).

[15] F. R. K. Chung, and R. L. Graham, "A new bound for Euclidean Steiner minimum trees," *Ann. N.Y. Acad. Sci., vol. 440* (1985), pp. 328-346.

[16] F. R. K. Chung, and F. K. Hwang, "A lower bound for the Steiner tree problem," *SIAM J. Appl. Math., vol. 34* (1978), pp. 27-36.

[17] Chvatal V., "Tough graphs and Hamiltonian circuits," *Discrete Math. 5* (1973), pp. 215-228.

[18] J. Clausen, and L. A. Hansen, "Finding k edge-disjoint spanning trees of minimum total weight in a network: an application of matroid theory," *Mathematical Programming Study, vol. 13* (1980), pp. 88-101.

[19] E. J. Cockayne, and D. G. Schiller, "Computation of Steiner minimal trees," in *Combinatorics*, D. A. Welsh, and D. R. Wodall (eds.) (1972).

[20] C. J. Colbourn, *The combinatorics of network reliability*, Oxford University Press (1987).

[21] C. J. Colbourn, "Edge-packings of graphs and network reliability," *J. Discrete Math., vol. 72* (1988), pp. 49-61.

[22] D. Coppersmith, and S. Winograd, "Matrix multiplication via arithmetic progression," *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (1987), pp. 1-6.

[23] W. H. Cunningham, "Optimal attack and reinforcement of a network," *Journal of the Association for Computing Machinery 32* (1985), pp. 549-561.

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

[25] R. J. Dakin, "A tree-search algorithm for mixed integer programming problems," *Comp. J., vol. 8* (1965), pp. 250-255.

[26] G. B. Dantzig, "Linear programming and extensions," *Princeton University Press, Princeton, NJ* (1963).

[27] J. deMercado, N. Spyratos, and B. A. Bowen, "A method for calculation of network reliability," *IEEE Trans. Reliability, vol. 25* (1976), pp. 71-76.

[28] S. E. Dreyfus, and R. A. Wagner, "The Steiner problem in graphs," *Networks, vol. 1* (1971), pp. 195-207.

[29] D. Z. Du, and F. K. Hwang, "A new lower bound for the Steiner ratio," *Trans. Amer. Math. Soc., vol. 278* (1983), pp. 137-148.

[30] D. Z. Du, and F. K. Hwang, "An approach for proving lower bounds: solution of Gilbert-Pollak's conjecture on Steiner ratio," *Proceedings of the 31st Annual IEEE Conference on Foundations of Computer Science* (1990), pp. 76-85.

[31] I. S. Duff, A. M. Erisman, and J. K. Reid, "On George's nested dissection method," *SIAM Journal on Numerical Analysis, vol. 13* (1976), pp. 686-695.

[32] I. Duff, N. Gould, M. Lescrenier, and J. K. Reid, "The multifrontal method in a parallel environment," in *Advances in Numerical Computation*, M. Cox and S. Hammarling, eds., Oxford University Press (1990).

[33] I. Duff, G. Grimes, and J. G. Lewis, "Users' guide for the Harwell-Boeing sparse matrix collection," Manuscript (1988).

[34] I. Duff, G. Grimes, and J. G. Lewis, "Sparse matrix test problems," *ACM Transactions on Mathematical Software, vol. 15* (1989), pp. 1-14.

[35] I. Duff, and J. K. Reid, "The multifrontal solution of indefinite sparse symmetric linear equations," *ACM Transactions on Mathematical Software, vol. 9* (1983), pp. 302-325.

[36] I. Duff, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press (1986).

[37] R. J. Duffin, "Topology of series-parallel networks," *J. Math. Anal. Appl., vol. 10* (1965), pp. 303-318.

[38] W. L. Eastman, "Linear programming with pattern constraints," Ph.D. Thesis, Report No. BL. 20, The Computational Laboratory, Harvard University (1958).

[39] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal on Mathematics, vol. 17* (1965), pp. 449-467.

[40] J. Edmonds and E. L. Johnson, "Matching, Euler tours and the Chinese postman," *Math. Programming, vol. 5* (1973), pp. 88-124.

[41] C. El-Arbi, "One heuristique pour le problem de l'arbre de Steiner," *R.A.I.R.O. Operations Research, vol. 12* (1978), pp. 207-212.

[42] U. Feige, S. Goldwasser, L. Lovász, and S. Safra, "Approximating clique is almost NP-complete," To appear in the *Proceedings of the 32nd Annual IEEE Conference on Foundations of Computer Science* (1991).

[43] L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, New Jersey (1962).

[44] O. Frank, and W. Gaul, "On reliability in stochastic graphs," *Networks, vol. 12* (1982), pp. 119-126.

[45] H. Frank, and I. T. Frisch, "Communication, Transmission, and Transportation Networks," Addison-Wesley (1971).

[46] L. Fratta, and U. G. Montanari, "A Boolean algebra method for computing the network reliability in a communication network," *IEEE Trans. Circuit Theory, vol. 20* (1973), pp. 203-211.

[47] Martin Fürer and Balaji Raghavachari, "An $\mathcal{NC}$ approximation algorithm for the minimum degree spanning tree problem," Proceedings of the 28th Annual Allerton Conference on Communication, Control and Computing (1990), pp. 274-281.

[48] K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*, SIAM (1990).

[49] M. R. Garey and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*, W. H. Freeman, San Francisco (1979).

[50] J. Gavett, "Three heuristic rules for sequencing jobs to a single production facility," *Management Science, vol. 11* (1965), pp. 166-176.

[51] B. Gavish, "Topological design of centralized computer networks - formulations and algorithms," *Networks 12* (1982), pp. 355-377.

[52] George, J. A., "Computer implementation of a finite element method," Tech. Report STAN-CS-208, Stanford University (1971).

[53] George, J. A., "Block elimination of finite element system of equations," in *Sparse Matrices and Their Applications*, D. J. Rose and R. A. Willoughby, eds., Plenum Press (1972).

[54] George, J. A., "Nested Dissection of a regular finite element mesh," *SIAM Journal on Numerical Analysis 10* (1973), pp. 345-367.

[55] George, J. A., "An automatic one-way dissection algorithm for irregular finite-element problems," *SIAM Journal on Numerical Analysis, vol. 17* (1980), pp. 740-751.

[56] George, J. A., and J. W. Liu, "An automatic nested dissection algorithm for irregular finite-element problems," *SIAM Journal on Numerical Analysis, vol. 15* (1978), pp. 1053-1069.

[57] George, J. A., and J. W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall Inc. (1981).

[58] George, J. A., and J. W. Liu, "The evolution of the minimum degree ordering algorithm," *SIAM Review, vol. 31* (1989), pp. 1-19.

[59] George, J. A., J. W. Liu, and E. G. Ng, "User's guide for SPARSPAK: Waterloo sparse linear equations package," Tech. Rep. CS78-30 (revised), Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada (1980).

[60] 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, vol. 13* (1976), pp. 236-250.

[61] E. N. Gilbert, and H. O. Pollak, "Steiner minimal trees," *SIAM Journal on Applied Mathematics, vol. 16* (1968), pp. 1-29.

[62] J. R. Gilbert, "Some Nested Dissection Order is Nearly Optimal," *Information Processing Letters* 26 (1987/88), pp. 325-328.

[63] J. R. Gilbert, personal communication (1989).

[64] J. R. Gilbert and H. Hafsteinsson, "Approximating treewidth, minimum front size, and minimum elimination tree height," manuscript, 1989.

[65] J. R. Gilbert, D. J. Rose and A. Edenbrandt, "A separator theorem for chordal graphs," *SIAM J. Alg. Disc. Meth. 5* (1984), pp. 306-313.

[66] J. R. Gilbert, and R. Schreiber, "Hightly parallel sparse Cholesky factorization," Tech. Report CSL-90-7, Xerox Palo Alto Research Center, 1990.

[67] J. R. Gilbert, and E. Zmijewski, "A parallel graph partitioning algorithm for a message-passing multiprocessor," *International Journal of Parallel Programming, vol. 16* (1987), pp. 427-449.

[68] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley (1989).

[69] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York (1980).

[70] R. L. Graham, and F. K. Hwang, "Remarks on Steiner minimal trees," *Bull. Inst. Math. Acad. Sinica, vol. 4* (1976), pp. 177-182.

[71] M. Grötschel, L. Lovász and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag (1988).

[72] M. X. Goemans and D. J. Bertsimas, "Survivable networks, linear programming relaxations and the parsimonious property," *OR 216-90, Center for Operations Research, MIT* (1990).

[73] M. X. Goemans and D. P. Williamson, "A general approximation technique for constrained forest problems," manuscript (1991).

[74] D. Gusfield, "Connectivity and edge-disjoint spanning trees," *Information Processing Letters 16* (1983), pp. 87-89.

[75] S. L. Hakimi, "Steiner's problem in graphs and its implications," *Networks, vol. 1* (1971), pp. 113-133.

[76] Mark D. Hansen, "Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems," *Proceedings of the 30th Annual IEEE Conference on Foundations of Computer Science* (1989), pp. 604-609.

[77] M. Held, and R. M. Karp, "The traveling salesman problem and minimum spanning trees," *Operations Research, vol. 18* (1970), pp. 1138-1162.

[78] M. Held, and R. M. Karp, "The traveling salesman problem and minimum spanning trees: part II," *Math. Prog., vol. 1* (1971), pp. 6-25.

[79] 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, vol. 10* (1973), pp. 364-369.

[80] I. Holyer, "The NP-completeness of edge-colouring," *SIAM Journal on Computing, vol. 10* (1981), pp. 718-721.

[81] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley (1979).

[82] F. K. Hwang, "On Steiner minimal trees with rectilinear distance," *SIAM Journal on Applied Mathematics, vol. 30(1)* (1976), pp. 104-114.

[83] A. Jain, "Probabilistic analysis of an LP relaxation bound for the Steiner problem in networks," *Networks, vol. 19* (1989), pp. 793-801.

[84] J. Jess, and H. Kees, "A data structure for parallel L/U decomposition," *IEEE Transactions on Computers, vol. 31* (1982), pp. 231-239.

[85] D. S. Johnson, "The NP-completeness column, an ongoing guide," *Journal of Algorithms, vol. 5* (1984), pp. 147-160.

[86] D. S. Johnson, "The NP-completeness column, an ongoing guide," *Journal of Algorithms, vol. 6* (1985), pp. 434-451.

[87] N. Karmakar, "A new polynomial-time algorithm for linear programming," *Combinatorica, vol. 4*, pp. 373-395 (1984).

[88] R. M. Karp, "Reducibility among combinatorial problems," in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations.* Plenum Press, New York (1972), pp. 85-103.

[89] U. Kjærulff, "Triangulation of graphs - Algorithms giving small total state space," *R 90-09, Institute for Electronic Systems, Department of Mathematics and Computer Science, University of Aalborg* (1990).

[90] P. N. Klein, A. Agrawal, R. Ravi and S. Rao, "Approximation through multicommodity flow," *31st Annual IEEE Conference on Foundations of Computer Science,* (1990), pp. 726-737.

[91] P. Klein, C. Stein and É. Tardos, "Leighton-Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities," *Proceedings of the 22nd ACM Symposium on Theory of Computing* (1990), pp. 310-321.

[92] P. Korhonen, "An algorithm for transforming a spanning tree into a Steiner tree," *Proc. 9th Int. Math. Progr. Symp.* (Budapest 1976), pp. 349-357.

[93] L. Kou, G. Markowsky and L. Berman, "A fast algorithm for Steiner trees," *Acta Informatica, vol. 15* (1981), pp. 141-145.

[94] E. L. Lawler, "Combinatorial Optimization: Networks and Matroids," Holt, Rinehart and Winston, New York (1976).

[95] F. T. Leighton, Fillia Makedon, Serge Plotkin, Clifford Stein, Eva Tardos, Spyros Tragoudas, "Fast approximation algorithms for multicommodity flow problems," *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing* (1991), pp. 101-111.

[96] F. T. Leighton and S. Rao, "An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms," *Proceedings of the 29th Annual IEEE Conference on Foundations of Computer Science* (1988), pp. 422-431.

[97] F. T. Leighton, F. Makedon and S. Tragoudas, personal communication, 1990

[98] C. Leiserson, and J. Lewis, "Orderings for parallel sparse symmetric factorization," in *Parallel Processing for Scientific Computing*, G. Rodrigue, ed., Philadelphia, PA, 1987, SIAM, pp. 27-32.

[99] M. Leuze, "Independent set orderings for parallel matrix factorization by Gaussian elimination," *Parallel Computing, vol. 10* (1989), pp. 177-191.

[100] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall Inc. (1981).

[101] J. Lewis, B. Peyton, and A. Pothen, "A fast algorithm for reordering sparse matrices for parallel factorization," *SIAM Journal on Scientific and Statistical Computing, vol. 10* (1989), pp. 1156-1173.

[102] R. J. Lipton, D. J. Rose and R. E. Tarjan, "Generalized nested dissection," *SIAM Journal on Numerical Analysis 16* (1979), pp. 346-358.

[103] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem, *SIAM Journal on Computing 9* (1980), pp. 615-627.

[104] J. D. Little, K. G. Murty, D. W. Sweeney, and C. Karel, "An algorithm for the traveling salesman problem," *Operations Research, vol. 11* (1963), pp. 972-989.

[105] J. W. Liu, "Modification of the minimum degree algorithm by multiple elimination," *ACM Transactions on Mathematical Software, vol. 12* (1985), pp. 141-153.

[106] J. W. Liu, "Reordering sparse matrices for parallel elimination," *Parallel Computing, vol. 11* (1989), pp. 73-91.

[107] J. W. Liu, "The minimum degree ordering with constraints," *SIAM Journal on Scientific and Statistical Computing, vol. 10* (1989), pp. 1136-1145.

[108] J. W. Liu, "A graph partitioning algorithm by node separators," *ACM Transactions on Mathematical Software, vol. 15* (1989), pp. 198-219.

[109] J. W. Liu, "The role of elimination trees in sparse factorization," *SIAM Journal on Matrix Analysis and Applications, vol. 11* (1990), pp. 134-172.

[110] J. W. Liu, and A. Mirzaian, "A linear reordering algorithm for parallel pivoting of chordal graphs," *SIAM Journal on Discrete Mathematics, vol. 2* (1989), pp. 100-107.

[111] J. W. Liu, and A. H. Sherman, "Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices," *SIAM Journal on Numerical Analysis, vol. 13* (1976), pp. 198-213.

[112] M. V. Lomonosov, and V. P. Polesski, "An upper bound of network reliability," *Problems of Information Transmission, vol. 7* (1971), pp. 337-339.

[113] M. V. Lomonosov, and V. P. Polesski, "Lower bound of network reliability," *Problems of Information Transmission, vol. 8* (1972), pp. 118-123.

[114] L. Lovász, and M. D. Plummer, "Matching Theorey," Akademiai Kiado; Budapest (1986).

[115] F. Makedon, and S. Tragoudas, "Approximating the minimum net expansion: near optimal solutions to circuit partitioning problems," Manuscript (1991).

[116] K. Mehlhorn, "A faster approximation algorithm for the Steiner problem in graphs," *Information Processing Letters, vol. 27(3)* (1988), pp. 125-128.

[117] Z. A. Melzak, "On the problem of Steiner," *Canad. Math. Bull., vol. 4* (1961), pp. 143-148.

[118] M. Minoux, "Network synthesis and optimum network design problems: models, solution methods and applications," *Networks, vol. 19* (1989), pp. 313-360.

[119] K. B. Misra, "An algorithm for the reliability of redundant networks," *IEEE Trans. Reliability, vol. 19* (1970), pp. 146-151.

[120] E. F. Moore and C. E. Shannon, "Reliable circuits using less reliable relays," *J. Franklin Inst., vol. 262* (1956), pp. 281-297.

[121] C. H. Papadimitriou, and K. Steiglitz, "Combinatorial Optimization," Prentice-Hall Inc. (1982).

[122] C. H. Papadimitriou, and M. Yannakakis, "The complexity of restricted minimum spanning tree problems," *Lecture Notes in Computer Science 71* (1979), pp. 460-470.

[123] S. Parter, "The use of linear graphs in Gaussian elimination," *SIAM Review, vol. 3* (1961), pp. 364-369.

[124] F. Peters, "Parallel pivoting algorithms for sparse symmetric matrices," *Parallel Computing, vol. 1* (1984), pp. 99-110.

[125] J. Plesnik, "A bound for the Steiner tree problem in graphs," *Math. Slovaca, vol. 31* (1981), pp. 155-163.

[126] T. Politof, and A. Satyanarayana, "A linear time algorithm to compute reliability of planar cube-free networks," *TIMS/ORSA Conference* (1984).

[127] V. P. Polesski, "A lower bound for the reliability of information networks," *Problems of Information Transmission, vol. 7* (1971), pp. 165-171.

[128] A. Pothen, "The complexity of optimal elimination trees," Tech. Report CS-88-16, Department of Computer Science, The Pennsylvania State University, University Park, PA, 1988.

[129] J. S. Provan, and M. O. Ball, "The complexity of counting cuts and of computing the probability that a graph is connected," *SIAM Journal on Computing, vol. 12* (1983), pp. 777-788.

[130] J. S. Provan, and M. O. Ball, "Computing network reliability in time polynomial in the number of cuts," *Operations Research, vol. 32* (1984), pp. 516-524.

[131] P. Raghavan, "Probabilistic construction of deterministic algorithms: approximating packing integer programs," *Proceedings of the 27th Annual IEEE Conference on Foundations of Computer Science* (1986), pp. 10-18.

[132] P. Raghavan and C.D. Thompson, "Provably good routing in graphs: regular arrays," *Proceedings of the 17th Annual Symposium on Theory of Computing* (1985), pp. 79-87.

[133] R. Ravi, personal communication (1991).

[134] R. Ravi, A. Agrawal and P. Klein, "Ordering problems approximated: single processor scheduling and interval graph completion," *Proceedings of the 18th International Colloquium on Automata, Languages and Programming* (1991).

[135] V. J. Rayward-Smith, "The computation of nearly minimal Steiner trees in graphs," *Int. J. Math. Educ. Sci. Tech., vol. 14* (1983), pp. 15-23.

[136] D. J. Rose, "Triangulated graphs and the elimination process," *Journal of Math. Anal. Appl. 32* (1970), p. 597-609.

[137] D. J. Rose, "A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations," in *Graph Theory and Computing*, R. C. Read, ed., Academic Press (1972), pp. 183-217.

[138] D. J. Rose, R. E. Tarjan and G. S. Lueker, "Algorithmic aspects of vertex elimination on graphs," *SIAM J. Comp. 5* (1976), pp. 266-283.

[139] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An analysis of several heuristics for the traveling salesman problem," *SIAM Journal on Computing, vol. 6* (1977), pp. 563-581.

[140] A. Satyanarayana, and A. Prabhakar, "New topological formula and rapid algorithm for reliability analysis of complex networks," *IEEE Trans. Reliability, vol. 27* (1978), pp. 82-100.

[141] A. Satyanarayana, and R. K. Wood, "Computing the k-terminal reliability of normal form graphs in polynomial time," *TIMS/ORSA Conference* (1983).

[142] A. Segev, "The node-weighted Steiner tree problem," *Networks, vol. 17* (1987), pp. 1-17.

[143] R. Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Trans. on Mathematical Software 8:3* (1982), pp. 256-276.

[144] F. Shahrokhi and D. W. Matula. "The maximum concurrent flow problem," Technical Report CSR-283, New Mexico Tech., 1988.

[145] M. L. Shooman, *Probabilistic Reliability: An Engineering Approach*, McGraw-Hill (1968).

[146] M. L. Shore, L. R. Foulds, and P. B. Gibbons, "An algorithm for the Steiner problem in graphs," *Networks, vol. 12* (1982), pp. 323-333.

[147] J. M. Smith, "An $O(n \log n)$ heuristic for Steiner minimal tree problems on the Euclidean metric," *Networks, vol. 11* (1981), pp. 23-39.

[148] J. M. Smith, D. T. Lee, and J. S. Liebman, "An $O(n \log n)$ heuristic algorithm for the rectilinear Steiner minimal tree problem," *Eng. Optimization, vol. 4* (1980), pp. 179-192.

[149] V. Strassen, "The asymptotic spectrum of tensors and the exponent of matrix multiplication," *Proceedings of the 27th Annual IEEE Conference of Foundations of Computer Science* (1986), pp. 49-54.

[150] G. F. Sullivan, "Approximation algorithms for Steiner tree problems," Tech. Rep. 249, Dept. of Comp. Sci., Yale Univ. (1982).

[151] H. Suzuki, T. Akama and T. Nishizeki, "Finding Steiner forests in planar graphs," *1st Ann. ACM-SIAM Symp. on Disc. Alg.* (1990), pp. 444-453.

[152] H. Takahashi and A. Matsuyama, " An approximate solution for the Steiner problem in graphs," *Math. Japonica, vol. 24* (1980), pp. 573-577.

[153] A. Tanenbaum, *Computer Networks*, Prentice-Hall Inc. (1981).

[154] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. Assoc. Comput. Mach., vol. 22* (1975), pp. 215-225.

[155] L. G. Valiant, "The complexity of enumeration and reliability problems," *Siam J. Comput., vol. 8* (1979), pp. 410-421.

[156] P. J. M. van Laarhoven, and E. H. L. Aarts, "Simulated Annealing: Theory and Practice," Kluwer Academic Publishers, Dordrecht, Holland (1987).

[157] V. G. Vizing, "On an estimate of the chromatic class of a p-graph" (in Russian), *Diskretnyĭ Analiz, vol. 3* (1964), pp. 11-12.

[158] O. Wing, and P. Demetriou, "Analysis of probabilistic networks," *IEEE Transactions on Communication Technology, vol. 12* (1964), pp. 38-40.

[159] Pawel Winter, "Steiner problem in networks : a survey," *Networks* (1987), pp. 129-167.

[160] Pawel Winter, "Generalized Steiner problem in outerplanar graphs,", *BIT 25* (1985), pp. 485-496.

[161] Pawel Winter, "An algorithm for the Steiner problem in the Euclidean plane," *Networks, vol. 15* (1985), pp. 323-345.

[162] R. T. Wong, "Worst-case analysis of network design problem heuristics," *SIAM J. Alg. Disc. Math., vol. 1* (1980), pp. 51-63.

[163] R. T. Wong, "A dual ascent approach for Steiner tree problems on a directed graph," *Math. Program., vol. 28*, (1984), pp. 271-287.

[164] Y. F. Wu, P. Windmayer and C. K. Wong, "A faster approximation algorithm for the Steiner problem in graphs," *Acta Informatica, vol. 23* (1986), pp. 321-331.

[165] M. Yannakakis, "Computing the minimum fill-in is NP-complete," *SIAM J. Algebraic and Discrete Methods 2* (1981), pp. 77-79.

[166] A. Z. Zelikovsky, "The 11/6-approximation algorithm for the Steiner problem on networks," *Institute of Mathematics, Kishinev, USSR.*