On the Design of Approximation Algorithms for a Class of Graph Problems

by

David Paul Williamson

S.B., Mathematics with Computer Science
Massachusetts Institute of Technology
(1989)
S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1990)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1993

Signature of Author_____
Department of Electrical Engineering and Computer Science
August 27, 1993

Certified by_____
Michel X. Goemans
Assistant Professor of Applied Mathematics
Thesis Supervisor

Accepted by_____
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

On the Design of Approximation Algorithms for a Class of Graph Problems

by

David Paul Williamson

Submitted to the Department of Electrical Engineering and Computer Science

on August 27, 1993,

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Abstract

Many problems in combinatorial optimization are known to be NP-hard, and thus it is unlikely that there exist any polynomial-time algorithms to solve them. Because a number of these problems are of practical interest, researchers have turned to devising polynomial-time *approximation algorithms* to solve them. An α-approximation algorithm runs in polynomial time and is guaranteed to produce a solution of cost within α times the optimal cost.

One difficulty in the field of approximation algorithm design is that until quite recently most algorithms were designed in an ad hoc manner, using the structure of the particular problem under consideration. In this thesis, we present a single technique for approximating a large class of graph problems. The technique is based on the standard primal-dual method, and shows the importance of this method in designing approximation algorithms.

The class of problems we consider involves selecting a minimum-cost set of edges such that at least $f(S)$ edges have exactly one endpoint in each subset of vertices S, for certain easily characterized functions f. We show that our technique leads to $2\mathcal{H}(f_{\max})$-approximation algorithms for problems in this class, where $f_{\max} = \max_S f(S)$ and $\mathcal{H}(k)$ is the harmonic function $1 + \frac{1}{2} + \cdots + \frac{1}{k}$. The class captures a wide variety of classical and complex problems in combinatorial optimization, including the minimum-cost spanning tree, Steiner tree, generalized Steiner tree, survivable network design, T-join, minimum-weight perfect matching (under triangle inequality), two-matching (under triangle inequality), and location-design problems. Our algorithm runs in $O(n^2 m' f_{\max} + n^2 \omega_f f_{\max})$ time on a graph of n vertices and m edges, where $m' = \min(n f_{\max}, m)$ and ω_f is the time to compute the function f. We obtain faster time bounds for specific problems of interest.

The algorithms produced by the technique are interesting for several reasons. First, they generalize classical algorithms for minimum-cost spanning trees and shortest paths. Second, they provide approximation algorithms for problems in P (such as the minimum-weight perfect matching problem under triangle inequality) that run asymptotically faster than the best-known algorithms that solve these problems exactly. Third, they provide approximation algorithms for problems which had no previously known approximation algorithm. For instance, the technique gives the first approximation algorithm for the survivable network

design problem, which is the problem of finding a minimum-cost set of edges such that there are r_{ij} edge-disjoint paths between each pair of vertices i and j. Fourth, the algorithms given by the technique improve on the running time and/or the approximation factor of several known approximation algorithms.

Extensions of the main technique of this thesis lead to approximation algorithms for problems which do not fall in the class of problems mentioned above. For example, we give a 2-approximation algorithm for the prize-collecting traveling salesman problem and a $2\mathcal{H}(k)$-approximation algorithm for the minimum-cost k-vertex-connected subgraph problem.

Finally, we conduct an experimental study of our 2-approximation algorithm for minimum-weight perfect matching on Euclidean instances. We present computational results for both random and real-world instances having between 1,000 and 131,072 vertices. The results indicate that our algorithm generates a matching within 2% of optimality in most cases. In over 1,400 experiments, the algorithm was never more than 4% from optimal.

Keywords: Combinatorial Optimization, Approximation Algorithms, Primal-Dual Algorithms, Graph Algorithms, Integer Programming, Steiner Trees, Matching, T-joins, Traveling Salesman Problem, Connectivity

Thesis Supervisor: Michel X. Goemans

Title: Assistant Professor of Applied Mathematics

Acknowledgments

This thesis would not have been possible without the help of many others, whom I would like to acknowledge here.

Michel Goemans has been a great advisor, collaborator, and friend. I am very grateful to him for his advice, accessability, and tremendous insight, without which I would probably still be struggling to find a thesis topic. It will be hard to find someone as easy to work with (and for), and who has ideas as good. I am proud to have been his first student in what I am sure will be a long and distinguished career.

The list of my other collaborators in the research contained in this thesis reads like an abridged version of *Who's Who in Combinatorial Optimization*: Hal Gabow, Andrew Goldberg, Milena Mihail, Serge Plotkin, R. Ravi, David Shmoys, Éva Tardos, and Vijay Vazirani. I feel fortunate to have met and worked with all of them. I am especially grateful to Vijay for repeatedly emphasizing to me the importance of the techniques used in the design of the algorithms. Although they were not coauthors, both David Johnson and David Applegate made many helpful suggestions for the algorithm implementation and experiments discussed in Chapter 8. Additionally, the experiments in that chapter would never have been completed without them: David Johnson offered to let me use AT&T Bell Labs' computers when nothing around MIT seemed appropriate, and David Applegate kept my experiments running. Finally, I am very grateful to David Johnson for bringing me to AT&T Bell Labs for the summer of '92. During that time I met Hal and Vijay, and a good

portion of the research in this thesis was accomplished.

I am grateful to those who helped me in preparing this document. Phil Klein and David Shmoys read through various drafts and provided many useful comments. David Wald provided PostScript routines for generating the figures in Chapter 8. Rob Schapire donated the LaTeX macros for the chapter headings. David Jones answered several LaTeX questions.

I am grateful for a National Science Foundation Fellowship, which funded my studies for most of my graduate career. I am also thankful for the generosity of Tom Leighton, who supplemented the fellowship at various points even though I was never officially his student. These supplements came from Air Force contract F49620-92-J-0125, and DARPA contracts N00014-89-J-1988 and N00014-92-J-1799.

Many people made my graduate years at MIT pleasant and interesting, and they deserve credit as well. The students of MIT's Theory Group kept things lively with discussions ranging from complexity theory to the stock market; I especially thank my officemates Rafail Ostrovsky, Alex Russell, Ravi Sundaram, David Wald, and Joel Wein. The other "Shmoys/Tardos Advisees in Exile" (Carolyn Norton, Cliff Stein, and Joel Wein) also deserve special mention for their friendship, help, and collaboration on various projects. Thanks to Cliff and Joel, I now know what a point guard is. I also thank Cliff for advice about the exam/job/thesis process, Joel for interesting discussions on theology, and Carolyn for commiserating about thesis writing. Many thanks goes to the group's secretary, Be Hubbard, for her cheerfulness and concern. Outside the Theory world, a large group of people kept me sane and human these past few years. They were always ready with interesting conversation, listening ears, advice, concern, a homemade meal or two, and their rich friendship. These include Mark Baker, Patty Bergstrom, Brent Chambers, Deb Chen, Jeff Collier, Barbara Collins, Sherry Curp, Jenny Graham, Mark Hickman, Fred Hoth, Chenny Lin, Greg Richardson, and the ST:TNG gang (with Eleanor, Beth, and Brian). For wise advice and encouragement, I am grateful to Tim Kochems and John Cuyler, as well as the rest of the ministerial staff at Park Street Church. Finally, I am grateful to my parents, who have always been there for me.

I strive after simplicity in my mathematics, partly because it is about all I am really capable of, but also partly because it is where I see the real beauty of mathematics. It has

never ceased to be a source of wonder to me that this world of ideas provokes and demands an aesthetic response. Even complete scientific materialists are tempted to invoke God's name when confronted these beauties; as a theist, I find it an appropriate reaction. I am grateful to God for re-creating a few of his thoughts these past few years. But I believe more than mere theism. That God should become human, and reveal his main characteristic as love, rather than truth or beauty, I find more awe-inspiring still.

Credits

Most of the work in this thesis originally appeared in a sequence of six papers, written with a number of different coauthors: Goemans and Williamson [53] (GW); Williamson, Goemans, Mihail, and Vazirani [127] (WGMV); Gabow, Goemans, and Williamson [46] (GGW); Goemans, Goldberg, Plotkin, Shmoys, Tardos, and Williamson [50] (GGPSTW); Ravi and Williamson [107] (RW); and Williamson and Goemans [126] (WG).

The first paper to appear, GW, contained the algorithm Approx-Proper-0-1 and the $O(n^2 \log n)$ time bound given in Section 5.2.1. The reduction of the performance guarantee to the total-degree inequality appeared there, as well as the proof of Theorem 4.1.1. In addition, the paper defined proper functions and listed all the various applications that could be solved by a proper function with range $\{0, 1\}$. GW also defined the edge duplication techniques à la Agrawal, Klein, and Ravi, and Goemans and Bertsimas, and noted that the central algorithm could be extended to the lower capacitated and location-routing problems, although the extension was cumbersome and has not appeared in print. The algorithms for the prize-collecting problems also appeared in GW.

The next paper, WGMV, introduced uncrossable functions, Approx-Uncrossable with Regular-Edge-Delete, and the proof of performance guarantee in Theorem 4.1.6. It also introduced a form of Approx-Proper in which *every* deficient set was augmented in each phase. This turned out to give a performance guarantee of $2f_{\max}$ in the worst case, although WGMV also showed that the bound could be improved in some cases. The

discussion of the primal-dual method in Section 1.3 is based on a section from this paper.

The paper GGW showed how to implement the algorithms of WGMV efficiently. In particular, it introduced EFFICIENT-EDGE-DELETE. The proof of performance guarantee in Theorem 4.1.7 appeared there in a slightly different form. All of the discussion of Chapter 5 not found in GW is from GGW, although it is applied to the APPROX-PROPER algorithm from WGMV.

Somewhat later, GGSPTW showed that by augmenting the maximally deficient sets, the $2\mathcal{H}(f_{\max})$ performance guarantee could be achieved. This paper introduced the weakly supermodular functions, and generalized APPROX-PROPER to APPROX-WEAKLY-SUPERMODULAR. It also pointed out that the efficient algorithms from GGW could be adapted to APPROX-WEAKLY-SUPERMODULAR. The $2\mathcal{H}(m)$ bound in the edge duplication section is from this paper, as is the algorithm for the fixed-charge network design problem.

The minimum-cost k-vertex-connected subgraph algorithm is from RW.

The computational study in Chapter 8 is from WG.

Table of Contents

CHAPTER 1

Introduction

The methods used for designing such [approximation] algorithms tend to be rather problem specific, although a few guiding principles have been identified and can provide a useful starting point.

– M.R. Garey and D.S. Johnson (1979), *Computers and Intractability*, p. 122

One of our hopes for this thesis has failed to be realized. This was that the proofs of our results, and the ideas involved in them, might be of use to researchers investigating other problems. Unfortunately, the best proofs we could find have turned out to be quite domain dependent, and even worse, the major proofs are exceedingly long.

– D.S. Johnson (1973), *Near-Optimal Bin Packing Algorithms*, Ph.D. Thesis, p. 13

1.1 General Introduction

The fields of computer science and combinatorial optimization have been closely related since their modern-day beginnings in the middle of this century. The desire to solve optimization problems was in part responsible for the construction of some early computers [62]; likewise, the existence of computational power encouraged the development of algorithms for optimization problems that had previously been too large to solve by hand. In

the 1960's, researchers began to notice that there were several optimization problems that seemed to require searching through all of the possible solutions, then picking the solution of maximum (or minimum) value. Since the number of possibilities was usually exponential or superexponential in the size of the description of a problem instance, these optimization problems seemed effectively resistant to solution by computer except for small instances. The existence of these problems was partially responsible for the interest within computer science in studying the kinds of problems computers could solve using bounded amounts of resources (such as time). In the mid 60s, this study led to the characterization of "good" algorithms as polynomial-time algorithms, by Edmonds [32] and a few others. The formalization of "intractable" problems as NP-complete or NP-hard came a few years later in the independent works of Cook [20] and Levin [83]. These two papers both effectively showed that finding a complete subgraph (or clique) of maximum size in a graph is NP-hard; this insight led to a groundbreaking paper of Karp [68], which showed that many of the optimization problems that had puzzled researchers in the 60s are NP-complete.[1] Whether there exist any polynomial-time algorithms for the NP-complete or NP-hard problems continues to be a major open question, although it is generally believed that no such algorithms exist.

Whether or not such algorithms exist, there are still many NP-hard optimization problems that people would like to solve without waiting for the P vs. NP problem to be settled. Many approaches to these problems have been developed within the past twenty to thirty years. The approaches can be divided into roughly two classes. One class attempts to find the optimal solution to the problem, without guaranteeing that the algorithm runs in polynomial time. One approach of this type formulates the optimization problem as an integer program, solves the associated linear programming relaxation of the integer program, then attempts to find violated constraints (called *cutting planes*) to add to the linear program so as to force the solution to be integral. These cutting plane algorithms are sometimes embedded within enumerative algorithms that cut off some of the branches of the search based on the values of solutions found in other branches; this technique is called *branch*

[1] See Sipser [112] for a nice overview of the history of the P vs. NP problem, including a version posed in a letter from Gödel to Von Neumann in 1956. Both Cook and Levin showed that the subgraph isomorphism problem is NP-complete. Additionally, Levin showed that a decision version of the set cover problem is NP-complete.

and bound. The other class of approach attempts to find a solution in a reasonable amount of time without guaranteeing that the solution found is optimal. Approaches within this class include local search techniques that start with some feasible solution and make local improvements until a local optimal solution is reached. Some approaches (such as *simulated annealing*) allow non-improvement steps to be made with a certain probability in order to avoid being trapped in local optima. Both classes of approaches have been applied successfully in practice to certain optimization problems, including, for instance, the well-known Traveling Salesman Problem (TSP). Applegate, Bixby, Chvátal, and Cook have used a variation on branch and bound to solve a 4461-city TSP instance to optimality [4], while Johnson [40] has used a local search algorithm due to Lin and Kernighan [85] to solve million city instances to within two percent of the cost of the optimal solution.

This thesis is concerned with an approach within the second class, known as *approximation algorithms*. Approximation algorithms attempt to have the best of both classes by running in polynomial time and guaranteeing that the solution produced has a value that is within a factor of α of the value of an optimal solution. The factor α is sometimes called the *performance guarantee* or the *approximation bound* of the algorithm; an approximation algorithm with a performance guarantee of α is often called an α-approximation algorithm. Under our definition, any algorithm called an approximation algorithm or α-approximation algorithm must run in polynomial time, while algorithms with specified performance guarantees may use either polynomial or superpolynomial time. We contrast approximation algorithms with *exact algorithms* for optimization problems. An exact algorithm finds an optimal solution to an optimization problem in polynomial time. Obviously, no exact algorithms are known for any NP-hard optimization problem.

The first known approximation algorithm solved the problem of minimizing the makespan of schedules for identical parallel machines, and was due to Graham [55]. Graham's paper appeared in 1966, five years in advance of Cook's paper on NP-completeness; notice that nothing in the definition of an approximation algorithm depends on the intractability of the problem being approximated. Nevertheless, the paper that established approximation algorithms in the field of algorithm design was Johnson's 1974 paper [66], which contained approximation algorithms for a half-dozen problems. These problems, which included max-

imum satisfiability, set covering, and maximum clique, are still considered central to the field today. Since 1974, hundreds of papers have appeared establishing new approximation algorithms for various problems, or improving the running times or performance guarantees of known approximation algorithms. In the past few years, interest in approximation algorithms has expanded from combinatorial optimization and algorithm design into complexity theory, as complexity theorists have realized that the ability to approximate a solution is fundamentally related to certain types of computation [37, 7, 6].

As the quotes at the beginning of this chapter attest, one difficulty in the field of approximation algorithm design is that, until quite recently, the design of approximation algorithms had tended to be ad hoc; no unifying principles or techniques were known or enunciated. Instead, designers confined themselves to using the specific structure of the problem at hand. In the past two or three years, however, researchers have shown that one central technique can often be used to design approximation algorithms for many related problems (see, for example, Leighton and Rao [80], Klein, Rao, Agrawal, and Ravi [76, 73], and Plotkin, Shmoys, and Tardos [102]). The purpose of this thesis is to propose a new technique and to show its usefulness in designing approximation algorithms for a large number of problems, from classical problems in combinatorial optimization to complex problems that had no previously known approximation algorithms. It is also a goal of this thesis to do what could not be done twenty years ago; that is, to provide a tool for algorithm designers to create approximation algorithms for new problems.

1.2 The Problems and Results

The class of optimization problems considered in this thesis primarily take the following form. Given an undirected graph $G = (V, E)$, a function $f : 2^V \rightarrow \mathbb{Z}$, and a non-negative cost function on the edges, $c : E \rightarrow \mathbb{Q}_+$, the problems can be modelled by the following integer program:

$$\text{Min} \quad \sum_{e \in E} c_e x_e$$

subject to:

$$(IP) \qquad x(\delta(S)) \geq f(S) \qquad S \subset V$$
$$x_e \in \{0,1\} \qquad e \in E,$$

where c_e is the cost of edge e, $x(F) = \sum_{e \in F} x_e$ and $\delta(S)$ denotes the *coboundary* of S; that is, the set of edges of E having exactly one endpoint in S. The integer program (IP) can be interpreted as a very special type of covering problem in which we need to find a minimum-cost set of edges that cover each coboundary $\delta(S)$ with at least $f(S)$ edges. For this reason, we will refer to problems defined by the integer program (IP) as *edge-covering problems*. This definition generalizes the classical edge-covering problem in which one must find a minimum-cost set of edges such that each vertex is incident to at least one edge. The classical problem can be modelled by (IP) with the function $f(S) = 1$ if $|S| = 1$ and $f(S) = 0$ otherwise.

Notice that since the edge costs are non-negative, any feasible solution to an edge-covering problem has an edge-minimal solution of no greater cost. Thus we will generally restrict our attention to edge-minimal solutions to (IP).

For the most part, we will consider edge-covering problems corresponding to functions f which we call *proper* functions.

Definition 1.2.1 A function $f : 2^V \rightarrow \mathbb{N}$ is *proper* if it obeys the following properties:

- [Symmetry] $f(S) = f(V - S)$ for all $S \subset V$; and
- [Maximality] If A and B are disjoint, then $f(A \cup B) \leq \max\{f(A), f(B)\}$.

We will also require that $f(V) = 0$ for every proper function f. Any edge-covering problem defined by a proper function f will be referred to as a *proper* edge-covering problem.

Many interesting problems can be modelled by proper edge-covering problems. In Table 1.1, we have indicated some examples of proper functions along with corresponding set of edge-minimal subgraphs induced by the integer program (IP). For example, the Steiner tree problem is a proper edge-covering problem. The Steiner tree problem is the problem of finding a minimum-cost set of edges spanning a set $T \subseteq V$ of terminals. As indicated in

Input	$f(S)$	Edge-minimal subgraph
	$f(S) = 1 \quad \forall S \neq V$	Spanning trees
$s, t \in V$	$f(S) = \begin{cases} 1 & \lvert S \cap \{s,t\}\rvert = 1 \\ 0 & \text{otherwise} \end{cases}$	s-t paths
$T \subseteq V$	$f(S) = \begin{cases} 1 & \emptyset \neq S \cap T \neq T \\ 0 & \text{otherwise} \end{cases}$	Steiner trees with terminals T
$T \subseteq V$	$f(S) = \begin{cases} 1 & \lvert S \cap T\rvert \text{ odd} \\ 0 & \text{otherwise} \end{cases}$	T-joins
	$f(S) = k \quad \forall S \neq V$	k-edge-connected subgraphs

Table 1.1: Various classical proper edge-covering problems.

the table, the problem is a proper edge-covering problem corresponding to

$$f(S) = \begin{cases} 1 & \text{if } \emptyset \neq S \cap T \neq T \\ 0 & \text{otherwise.} \end{cases}$$

Any set of edges feasible for (IP) with the given function f must connect all terminals, and any Steiner tree is feasible for (IP) with function f. Moreover, we claim that f is proper. It is easy to see that f is symmetric. To see that it obeys the maximality property, suppose that it does not, and there exist disjoint sets A and B such that $f(A) = f(B) = 0$ while $f(A \cup B) = 1$. If $f(A \cup B) = 1$ then by the definition of f, the set $A \cup B$ must separate two terminals, say s and t; that is, $s, t \in T$, $s \in A \cup B$, and $t \notin A \cup B$. But then either A or B must separate s and t, contradicting $f(A) = f(B) = 0$. Therefore, f is proper.

Many more complex combinatorial optimization problems, such as the non-fixed point-to-point connection problem and the survivable network design problem can also be modelled by proper edge-covering problems. In Chapter 6, we discuss all the various proper edge-covering problems of interest that we know.

Finding optimal solutions to proper edge-covering problems is well-known to be NP-hard, even under many restrictions. For example, the Steiner tree problem is NP-hard even when the cost function satisfies the Euclidean metric [48], and the minimum-cost 2-edge

connected subgraph problem is NP-hard even if all edge weights are 1 [36]. Approximation algorithms for proper edge-covering problems were previously known only for special cases.

This thesis will present the first approximation algorithm for proper edge-covering problems. This result is summed up in the following theorem. First we define some notation, which we will continue to use throughout the thesis. Given an undirected graph $G = (V, E)$, we define $n = |V|$ and $m = |E|$. Given a function $f : 2^V \rightarrow \mathbb{Z}$, define $f_{\max} = \max_S f(S)$. We assume that the function f is given to the approximation algorithm as an oracle which on input $S \subseteq V$ returns $f(S)$. Let ω_f be the maximum amount of time taken by an oracle to compute the function f on a set S. Define $m' = \min(nf_{\max}, m)$. Finally, define the harmonic function $\mathcal{H}$ as $\mathcal{H}(k) = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$.

Theorem 1.2.2 For any proper edge-covering problem, there exists a $2\mathcal{H}(f_{\max})$-approximation algorithm, APPROX-PROPER, that runs in $O(n^2 m' f_{\max} + n^2 \omega_f f_{\max})$ time on an undirected graph $G = (V, E)$ and a proper function f.

The theorem immediately extends to augmentation versions of proper edge-covering problems, in which one must augment a given graph so as to satisfy a proper function f at minimum cost. The augmentation algorithm also runs in polynomial time and has a $2\mathcal{H}(f_{\max})$ performance guarantee.

For most practical problems, we expect that $\omega_f = O(n)$ and $f_{\max} = O(1)$, so that the running time is $O(n^3)$. For particular problems of interest, the running time can be significantly improved. For example, when $f_{\max} = 1$, the running time can be improved to $O(n(n + \sqrt{m \log\log n}))$. As can be seen in Table 1.1 above, even such a restricted function f captures many interesting problems.

We list several problems to which our techniques can be applied in Tables 1.2 and 1.3. In Table 1.2 we list problems for which previously known approximation or exact algorithms exist. In most cases we improve either the performance guarantee α or the running time of the algorithm; in some cases we improve both. In Table 1.3 we list problems which had no previously known approximation algorithm.

In practice, we expect that the quality of the solutions produced will be better than the theoretical bounds established in the theorem. In Chapter 8, we discuss an implemen-

Problem	Known α	Time	Due to	Our α	Our time
Steiner tree	2	$O(n^2)$	Mehlhorn [91]	2	$O(n^2 + n\sqrt{m \log\log n})$
	11/6	$O(mn + n^2 \log n)$	Zelikovsky [130, 131]		
	16/9	$O(n^{7/2})$	Berman and Ramaiyer [14]		
Generalized Steiner tree	2	$O(n^2 \log n)$	Agrawal, Klein, Ravi [2]	2	$O(n^2 + n\sqrt{m \log\log n})$
	2	$O(n\sqrt{m} \log n)$	Klein [75]		
k-edge connected subgraph	2	$O(kn^3 \log n)$	Khuller, Vishkin [71]	$2\mathcal{H}(k)$	$O(kn(kn + \sqrt{m \log\log n}))$
Generalized Steiner 2-edge connected subgraph	3	$O(n^2 \log n)$	Klein, Ravi [74]	3	$O(n^2 + n\sqrt{m \log\log n}))$
Survivable network design (with edge duplication)	$2L$	$O(Ln^2 \log n)$	Agrawal, Klein, Ravi [2]	$2L$	$O(L(n^2 + n\sqrt{m \log\log n}))$
Prize-collecting traveling salesman (with triangle inequality)	2.5	$poly(n)$	Bienstock, Goemans, Simchi-Levi, Williamson [16]	2	$O(n^2 + n\sqrt{m \log\log n})$
Prize-collecting Steiner tree	3	$poly(n)$	Bienstock, Goemans, Simchi-Levi, Williamson [16]	2	$O(n^2 + n\sqrt{m \log\log n})$
Min-weight perfect matching (with triangle inequality)	1	$O(n(m + n \log n))$	Gabow [44]	2	$O(n^2 + n\sqrt{m \log\log n})$
	1	$O(m\sqrt{n\alpha(m,n) \log n} \log nC)$	Gabow and Tarjan [47]		
	$3 + 2\epsilon$	$O(n^2 \log^{2.5} n \log(1/\epsilon))$	Vaidya [121]		
2-matching (with triangle inequality)	1	$O(n(m + n \log n))$	Gabow [44]	2	$O(n^2 + n\sqrt{m \log\log n})$
	1	$O(m\sqrt{n\alpha(m,n) \log n} \log nC)$	Gabow and Tarjan [47]		
T-join	1	$O(n(m + n \log n))$	Gabow [44]	2	$O(n^2 + n\sqrt{m \log\log n})$
	1	$O(m\sqrt{n\alpha(m,n) \log n} \log nC)$	Gabow and Tarjan [47]		

Table 1.2: Improved approximation algorithms for previously approximated problems. In the survivable network design problem, $L = \log f_{max}$.

Problem	Our α	Our time
Survivable network design [58] (with no edge replication)	$2\mathcal{H}(f_{\max})$	$O(nf_{\max}(nf_{\max} + \sqrt{m \log\log n}))$
Triangle-free 2-matching (with triangle inequality)	2	$O(m + n \log n)$
Non-fixed point-to-point connection [84]	2	$O(n^2 + n\sqrt{m \log\log n})$
Exact k-tree partition (with triangle inequality)	$4(1 - \frac{1}{k})$	$O(n^2 + n\sqrt{m \log\log n})$
Exact k-path partition (with triangle inequality)	$4(1 - \frac{1}{k})$	$O(n^2 + n\sqrt{m \log\log n})$
Exact k-cycle partition (with triangle inequality)	$4(1 - \frac{1}{k})$	$O(n^2 + n\sqrt{m \log\log n})$
At least k-tree partition	2	$O(m + n \log n)$
At least k-path partition (with triangle inequality)	4	$O(m + n \log n)$
At least k-cycle partition (with triangle inequality)	2	$O(m + n \log n)$
Location-design problems	2	$O(m + n \log n)$
k-vertex-connected subgraph	$2\mathcal{H}(k)$	$O(k^4 n^3)$
Fixed-charge network design	$2f_{\max}$	$O(f_{\max}(nm' f_{\max} + n^2 m' + n^2 \omega_f))$

Table 1.3: New approximation algorithms for previously unapproximated problems.

tation of the algorithm for the problem of minimum-weight perfect matching on Euclidean instances, which can be solved approximately using the proper edge-covering algorithm as a subroutine. Our theoretical results imply a 2-approximation algorithm for matching, but in over 1,400 experiments on instances ranging in size from 1,000 to 131,072 vertices, the algorithm was never more than 4% away from optimal and was usually within 2%.

Almost all edge-covering problems of interest are proper edge-covering problems. However, our approximation algorithm for solving proper edge-covering problems is based on algorithms for approximating (IP) with two other types of functions, which we call *weakly supermodular* functions and *uncrossable* functions. We will use a convention in which all weakly supermodular and proper functions are denoted by f, and uncrossable functions are denoted by h.

Definition 1.2.3 A function $f : 2^V \to \mathbb{Z}$ is *weakly supermodular* if $f(V) = 0$ and for any two sets $A, B \subset V$,

$$f(A) + f(B) \leq \max\{f(A \cup B) + f(A \cap B), f(A - B) + f(B - A)\}.$$

Definition 1.2.4 A function $h : 2^V \to \{0, 1\}$ is *uncrossable* if $h(V) = 0$ and for any two sets $A, B \subset V$ such that $h(A) = h(B) = 1$, then either $h(A \cap B) = h(A \cup B) = 1$ or $h(A - B) = h(B - A) = 1$.

Notice that the uncrossable functions contain the weakly supermodular functions with range {0,1}. The names for these two classes of functions derive from terminology in optimization theory. A function f on 2^V is called *supermodular* if

$$f(A) + f(B) \leq f(A \cup B) + f(A \cap B);$$

if f is symmetric, then supermodularity implies $f(A) + f(B) \leq f(A - B) + f(B - A)$ as well. Two sets A and B are said to *cross* (or are *crossing*) if $A \cap B \neq \emptyset$, $A \not\subseteq B$, and $B \not\subseteq A$. We will often need to replace or "uncross" two crossing sets A and B with either $A - B$ and $B - A$ or $A \cup B$ and $A \cap B$; hence the function name. These concepts, as well as the concept of a *submodular* function $(f(A) + f(B) \geq f(A \cup B) + f(A \cap B))$ have deep implications in

network flow theory and other areas (c.f. Lovász and Plummer [88], Lovász [87]). We will explore some of the properties of these functions in Chapter 2.

We call problems modelled by the integer program (IP) with weakly supermodular functions and with uncrossable functions *weakly supermodular edge-covering* problems and *uncrossable edge-covering* problems, respectively. Examples of uncrossable edge-covering problem include the shortest s-t path problem, which can be modelled by the function $h(S) = 1$ iff $s \in S$ and $t \notin S$, and the classical edge-covering problem, which corresponds to the function $h(S) = 1$ iff $|S| = 1$.

Unfortunately, it is not possible to provide approximation algorithms for the classes of weakly supermodular and uncrossable edge-covering problems as a whole given the oracle model we have used so far. To see this, notice that a function in which $f(S) = 1$ for some arbitrary set $S \subset V$ and $f(S') = 0$ for $S' \neq S$ is both weakly supermodular and uncrossable. For such a function it would take an exponential number of calls to the oracle merely to find a violated constraint of the integer program (IP). However, we can implement approximation algorithms if we have a *strong* oracle, which when given a function f and a non-feasible solution x to the integer program (IP) finds all maximally violated constraints that are minimal with respect to inclusion; that is, the oracle finds a collection of all minimal sets S such that $f(S) - x(\delta(S)) = \max_T(f(T) - x(\delta(T))) > 0$. We will denote this oracle by MAX-VIOLATED(f, x). Given an edge set F, we will also use the notation MAX-VIOLATED(f, F) as shorthand for MAX-VIOLATED(f, x) with x the incidence vector of F. An *incidence vector* of an edge set F is a solution $x_e = 1$ if $e \in F$ and $x_e = 0$ otherwise. Define $\delta_F(S)$ to be the set of edges $F \cap \delta(S)$. Then MAX-VIOLATED(f, F) finds a collection of minimal sets S such that $f(S) - |\delta_F(S)| = \max_T(f(T) - |\delta_F(T)|) > 0$. Chapter 2 will show that such a collection consists of disjoint sets.

Let σ_f (σ_h) be the maximum amount of time taken by the strong oracle for a weakly supermodular function f (an uncrossable function h). We can then prove the following two results.

Theorem 1.2.5 For any weakly supermodular edge-covering problem, there is a $2\mathcal{H}(f_{\max})$-approximation algorithm, APPROX-WEAKLY-SUPERMODULAR, which runs in $O(f_{\max}(n^2 + n\sqrt{m \log\log n} + n\omega_f + n\sigma_f))$ time on an undirected graph $G = (V, E)$ and a weakly supermod-

ular function f.

Theorem 1.2.6 For any uncrossable edge-covering problem, there is a 2-approximation algorithm, APPROX-UNCROSSABLE, which runs in $O(n^2 + n\sqrt{m \log\log n} + n\omega_h + n\sigma_h)$ time on an undirected graph $G = (V, E)$ and an uncrossable function h.

We will show in Chapter 3 that the algorithm APPROX-WEAKLY-SUPERMODULAR can be implemented simply as $f_{\max}$ calls to the algorithm APPROX-UNCROSSABLE. To obtain our approximation algorithm for proper edge-covering problems, we show in Chapter 2 that all proper functions are weakly supermodular functions, and we show in Chapter 5 that for proper functions f the strong oracle can be implemented in terms of the weak oracle in $\sigma_f = O(nm' + n^2\omega_f)$ time for all calls to the strong oracle generated by the algorithm APPROX-WEAKLY-SUPERMODULAR. Thus the time bound of APPROX-WEAKLY-SUPERMODULAR implies the time bound of APPROX-PROPER in Theorem 1.2.2; an implementation trick will lead to the reduced number of calls to the weak oracle given in the theorem.

1.3 The Primal-Dual Method

The basis of the technique used to derive these algorithms is none other than the primal-dual method, applied to the domain of approximation algorithms. The primal-dual approach has been used extensively in exact algorithms for optimization problems. The best-known algorithms for several of the most basic problems in combinatorial optimization, including matching, network flows, and shortest paths, are based on this approach [98]. Some of these algorithms (Dijkstra's shortest path algorithm [29], for example) can be described without referring to the primal-dual framework. In other cases, such as Edmonds' algorithm for the minimum-weight non-bipartite matching problem [31], the primal-dual approach appears to be crucial.

The standard primal-dual method relies on elementary linear programming theory; we refer the reader unfamiliar with the basic theorems and terminology to introductions in Chvátal [19] or Strang [118, Ch. 8]. The standard method requires a primal linear program that models the optimization problem to be solved, and works as follows. It begins with a

feasible solution to the dual linear program and an infeasible solution to the primal such that the complementary slackness conditions hold. The method then alternately improves the feasibility of the primal solution and the value of the dual solution, while maintaining the complementary slackness conditions. The algorithm terminates when the primal solution becomes feasible: a well-known fact of linear programming is that primal and dual feasible solutions are optimal if they obey the complementary slackness conditions. The combinatorial structure of the problem at hand is used for designing the improvement steps. Thus, for optimization problems solvable in polynomial-time, the primal-dual method simply gives an efficient combinatorial algorithm for solving the associated linear program.

It is part of the purpose of this thesis to state a corresponding primal-dual method for approximation algorithms, using our algorithm for approximating uncrossable edge-covering problems as an example. Many approximation algorithms have used duality or linear programming-based duality arguments (see, for example, Wolsey [129], Leighton and Rao [80], Lenstra, Shmoys, and Tardos [81]). A few of these fit within the framework we are about to describe, notably the algorithms of Chvátal [18] and Bar-Yehuda and Even [10] for the set-covering problem. Also, several heuristics for various optimization problems use primal-dual type algorithms, although they do not fit in the framework given here, and have no performance guarantees (see Nemhauser and Wolsey [94], §II.5.3).

Suppose we wish to solve an uncrossable edge-covering problem for an uncrossable function h. Because $h_{\max} = 1$, the problem can be modelled by the following integer program (IP_h) in which the constraints $x_e \in \{0,1\}$ have been relaxed:

$$
\begin{aligned}
&\text{Min} \quad \sum_{e \in E} c_e x_e \\
&\text{subject to:} \\
(IP_h) \qquad & x(\delta(S)) \geq h(S) && S \subset V \\
& x_e \geq 0 && e \in E, \\
& x_e \text{ integer.}
\end{aligned}
$$

Consider the linear programming relaxation of (IP_h)

$$\begin{aligned} &\text{Min} && \sum_{e \in E} c_e x_e \\ &\text{subject to:} \\ (LP_h) \qquad & && x(\delta(S)) \geq h(S) && S \subset V \\ & && x_e \geq 0 && e \in E, \end{aligned}$$

and its dual,

$$\begin{aligned} &\text{Max} && \sum_{S} h(S) y_S \\ &\text{subject to:} \\ (D_h) \qquad & && \sum_{S : e \in \delta(S)} y_S \leq c_e && e \in E \\ & && y_S \geq 0 && S \subset V. \end{aligned}$$

The dual linear program is a packing problem in which we pack an amount y_S on to each cut S so that no edge is "overpacked" and so as to maximize the amount packed on to cuts for which $h(S) = 1$.

Since (LP_h) is not guaranteed to have integral optimal solutions, we will not always be able to find an integral solution to (LP_h) and a solution to (D_h) that obey all the complementary slackness conditions. For these linear programs, the complementary slackness conditions are of two types:

(a) Primal complementary slackness conditions, which correspond to the primal variables; i.e.,

$$x_e > 0 \Rightarrow \sum_{S : e \in \delta(S)} y_S = c_e.$$

(b) Dual complementary slackness conditions, which correspond to the dual variables; i.e.,

$$y_S > 0 \Rightarrow \sum_{e \in \delta(S)} x_e = h(S).$$

The primal-dual method for approximation algorithms maintains one set of comple-

mentary slackness conditions while relaxing the other set. The algorithms of Chvátal and Bar-Yehuda and Even applied to this special class of set-covering problems effectively ensure the primal conditions and relax the dual conditions to

(b')

$$y_S > 0 \Rightarrow h(S) \leq \sum_{e \in \delta(S)} x_e \leq \alpha h(S).$$

For this relaxation of the complementary slackness conditions, a performance guarantee of α for the algorithm follows easily, as we will now show. Let Z^*_{LP-h} be the value of the optimal solution to (LP_h) (and thus (D_h)), and let Z^*_{IP-h} be the value of the optimal solution to (IP_h). Then given a primal integral solution x and a dual feasible solution y generated by the primal-dual algorithm, we have

$$\begin{aligned}
\sum_{e \in E} c_e x_e &= \sum_{e \in E} x_e \sum_{S : e \in \delta(S)} y_S \\
&= \sum_S y_S \sum_{e \in \delta(S)} x_e \\
&\leq \sum_S y_S (\alpha h(S)) \\
&\leq \alpha Z^*_{LP-h} \\
&\leq \alpha Z^*_{IP-h}.
\end{aligned}$$

Our algorithm for approximating uncrossable edge-covering problems will relax the dual constraints (b) in a somewhat different manner. Instead of enforcing (b') for the final primal and dual solutions, we will only ensure that (b') holds in an *average* sense during any given dual improvement step. It turns out that this is sufficient to prove that $\sum_e c_e x_e \leq 2 \sum_S h(S) y_S$, which (as above) implies performance guarantee of 2. We describe the relaxation of the dual complementary slackness conditions at more length in Section 4.1.

Notice that a proof that the algorithm constructs solutions x and y such that $\sum_e c_e x_e \leq 2 \sum_S h(S) y_S$ also implies a bound on the maximum ratio of the value of an optimal solution to (IP_h) to the value of an optimal solution to (D_h). This ratio is called the *relative duality gap*. The proof of the performance guarantee shows that the relative duality gap is at most

2 for all non-negative cost functions c and all uncrossable functions h. In Chapter 4, our theorem about the performance guarantee of Approx-Weakly-Supermodular will have the following consequence.

Theorem 1.3.1 For all non-negative cost functions c and all weakly supermodular f, the relative duality gap of (IP) is at most $2\mathcal{H}(f_{\max})$.

The use of the primal-dual method also gives an instance-by-instance estimate on the nearness to optimality of the algorithm. If the algorithm constructs solutions x and y, then the value of the solution x can be no more than a factor $(\sum_e c_e x_e)/(\sum_S h(S)y_S)$ away from the value of the optimal solution.

Our use of the primal-dual method for approximating various classes of edge-covering problems is derived from a paper of Agrawal, Klein, and Ravi [2] on a 2-approximation algorithm for the generalized Steiner tree problem. Given sets of terminals $T_i \subseteq V$ for $i = 1, \ldots, p$, the generalized Steiner tree problem is to find a minimum-cost set of edges such that all the vertices are connected in each set of terminals T_i. As we will discuss in Chapter 6, this problem is a proper edge-covering problem. The algorithm of Agrawal et al. implicitly uses the entire framework we have described above, including the average enforcement of the relaxed constraint (b'). Our central algorithm, Approx-Uncrossable, generalizes their use of the primal-dual method to uncrossable edge-covering problems, and makes their use of linear programming duality explicit.

The primal-dual method may be more important for designing approximation algorithms than for designing exact algorithms. As we observed above, optimization problems solvable in polynomial time via the primal-dual method can also be solved using linear programming. As far as we know, the same cannot be said of the NP-hard problems to which we apply the primal-dual method for approximation algorithms.

1.4 Previous Results

The proper, weakly supermodular, and uncrossable edge-covering problems are defined for the first time in this thesis. Hence there has been no previous work done on these

problems as classes of problems. Nevertheless, a good deal of work has been done on specific problems within the classes, and on generalizations of the class of edge-covering problems. We discuss here the previous results on generalizations, and defer discussion of results of specific problems until Chapters 6 and 7, in which we describe the particular problems that can be solved using our algorithms.

As was mentioned in a previous section, the integer program (IP) can be viewed as a special type of covering problem. Many people have investigated approximation algorithms for general covering problems. These covering problems can be expressed as the integer program (COV),

$$(COV) \qquad \begin{aligned} \text{Min} \quad & c^T \cdot x \\ & A \cdot x \geq b \\ & x \geq 0 \\ & x \text{ integer}, \end{aligned}$$

where c and x are $p \times 1$ vectors, b is a $q \times 1$ vector, and A is a $p \times q$ 0-1 matrix. Thus the uncrossable edge-covering problems are contained within the class of covering problems in which $c \geq 0$, b is 0-1, $p = m$, and $q = 2^n - 2$, while proper and weakly supermodular edge-covering problems are contained in the class in which $b \in \mathbb{N}^q$.

Essentially two approaches to this problem have been developed, both of which run in time polynomial in the size of A. Johnson [66] and Lovász [86] started the first approach by giving greedy algorithms for approximating the case in which $b = c = \mathbf{1}$, where $\mathbf{1}$ is the vector of all 1s (note that inequalities corresponding to entries for which $b_i = 0$ can be deleted). If a^j is the sum of the entries of the jth column of A, the performance guarantee of their algorithms is $\mathcal{H}(\max_j a^j)$. By using a linear programming duality argument, Chvátal [18] extended their algorithms to handle non-negative cost vectors. His algorithm has the same performance guarantee, and can be interpreted as a primal-dual algorithm. Dobson [30] generalized Chvátal's algorithm to handle general non-negative integral vectors b, also with the same performance guarantee.

The second approach to approximating (COV) began with Hochbaum [61]. Hochbaum

showed how to use the optimal dual solution to the linear programming relaxation of (COV) to obtain an approximate integer solution in polynomial time, given that $b = \mathbf{1}$. If a_i is the sum of the entries of the ith row of A, the performance guarantee of her algorithm is $\max_i a_i$. Bar-Yehuda and Even [10] showed how Hochbaum's algorithm could be carried out without solving the linear program by giving a primal-dual $(\max_i a_i)$-approximation algorithm. Hall and Hochbaum [60] then extended their algorithm to handle general non-negative integral vectors b.

Although the number of constraints of the integer program (IP) is exponential in n, the algorithm of Hall and Hochbaum can be applied to approximate (IP) in time polynomial in n and m if there exists a polynomial-time subroutine to find a violated constraint. The strong oracle MAX-VIOLATED can find such violated constraints, and we show how to implement the strong oracle in polynomial time in Section 5.1, proving that their algorithm is an m-approximation algorithm for any proper edge-covering problem. It is not clear how to apply the Chvátal/Dobson approach to approximate edge-covering problems in polynomial time, however. A straightforward implementation of their algorithms involves counting the number of constraints $\sum_{e \in \delta(S)} x_e \geq f(S)$ that would be satisfied by adding a given edge, and it is unclear how this can be done efficiently.

As mentioned in the previous section, the work in this thesis began not as a specialization of these covering algorithms but as a generalization and simplification of an algorithm due to Agrawal, Klein, and Ravi [2] for the generalized Steiner tree problem. In response to an early part of this thesis describing how to approximate (IP) for all proper functions with $f_{\max} = 1$ [53], Klein and Ravi [74] showed how to approximate (IP) for all proper functions with range $\{0, 2\}$. This work was subsequently generalized to handle all proper functions [127], after which both the efficiency [46] and the performance guarantee [50] of the algorithm was improved.

1.5 Overview of Contents

Given this broad overview of the problems we will consider and previous approaches to them, we now turn to our technique for approximating edge-covering problems. We be-

gin in Chapter 2 with a few preliminary lemmas about proper, uncrossable, and weakly supermodular functions. In Chapter 3, we introduce our basic algorithmic technique with an algorithm, Approx-Proper-0-1, for approximating proper edge-covering problems for proper functions with range $\{0,1\}$. We then show how this algorithm generalizes to an algorithm for uncrossable edge-covering problems (Approx-Uncrossable). The chapter concludes with an algorithm that reduces weakly supermodular edge-covering problems to a sequence of uncrossable edge-covering problems. Chapter 4 proves the performance guarantees of the algorithms. In Chapter 5, we fill in the details of the algorithms, and show how each step can be implemented in polynomial time. In Chapter 6, we show how the algorithms can be applied to many graph problems in combinatorial optimization, and Chapter 7 describes how the technique can be extended to several interesting problems that are not modelled by the integer program (IP).

Chapter 8 turns to one particular problem that can be approximated using the algorithm: the Euclidean perfect matching problem. We describe an actual implementation of our algorithm and compare it to exact matching algorithms and various matching heuristics. We investigate the quality and structure of solutions produced on both random and structured instances.

Finally, Chapter 9 contains some concluding thoughts about the technique given in this thesis, and lists some problems that remain unresolved.

Although this thesis has been written so as to cover all of the results we have derived in this area, we have also attempted to make the basic ideas accessible to the casual reader. We suggest that those only interested in a central, self-contained idea from the thesis read about the algorithm for proper functions with range $\{0,1\}$ (Section 3.1), its proof of performance guarantee (Chapter 4 up to Section 4.1.1), and its applications (most of Chapter 6). These results are also found in an early paper from the thesis [53]. With this background, the reader will also be able to understand the experimental study in Chapter 8 and several of the extensions given in Chapter 7. It is also possible to survey all of the main algorithmic and analytic ideas without grappling with some of the more complicated ideas needed to make the algorithms efficient. To do this, one may skip or delay reading the efficient version of Approx-Uncrossable given in Section 3.2.2, its proof of performance guarantee in

Section 4.1.3, and the implementation details of Chapter 5. The remainder of the thesis can be understood without reading these sections.

In addition to some basic knowledge of linear programming, we will assume some background in network flows and data structures, particularly in Chapter 5. We refer the reader unfamiliar with these topics to a basic text in algorithms, such as Cormen, Leiserson, and Rivest [21].

CHAPTER 2

Preliminaries

In this chapter, we briefly establish a few facts about weakly supermodular, proper, and uncrossable functions that will be important in the following pages. We restate the definitions here for convenience.

- A function $f : 2^V \to \mathbb{Z}$ is weakly supermodular if $f(V) = 0$ and for any two sets $A, B \subset V$,

$$f(A) + f(B) \leq \max\{f(A - B) + f(B - A), f(A \cap B) + f(A \cup B)\}.$$

- A function $f : 2^V \to \mathbb{N}$ is proper if $f(V) = 0$ and it obeys the following two properties:
  - [Symmetry] $f(S) = f(V - S)$ for all $S \subset V$; and
  - [Maximality] If A and B are disjoint, then $f(A \cup B) \leq \max\{f(A), f(B)\}$.
- A function $h : 2^V \to \{0, 1\}$ is uncrossable if $h(V) = 0$ and for any two sets $A, B \subset V$ such that $h(A) = h(B) = 1$, then either $h(A \cap B) = h(A \cup B) = 1$ or $h(A - B) = h(B - A) = 1$.

We have already noted that all weakly supermodular functions with range $\{0, 1\}$ are uncrossable functions. This is a special case of the following observation.

Observation 2.0.1 Let f be a weakly supermodular function. Then

$$h(S) = \begin{cases} 1 & \text{if } f(S) = f_{\max} \\ 0 & \text{otherwise} \end{cases}$$

is an uncrossable function.

We now relate proper functions and weakly supermodular functions.

Theorem 2.0.2 If f is a proper function, then f is weakly supermodular.

Proof: By the properties of proper functions, we have the following four inequalities:

- $\max\{f(A-B), f(A \cap B)\} \geq f(A)$.
- $\max\{f(B-A), f(A \cup B)\} \geq f(A)$.
- $\max\{f(B-A), f(A \cap B)\} \geq f(B)$.
- $\max\{f(A-B), f(A \cup B)\} \geq f(B)$.

Summing the two inequalities involving the minimum of $f(A-B)$, $f(B-A)$, $f(A \cup B)$, $f(A \cap B)$ shows that $f(A) + f(B) \leq \max\{f(A-B) + f(B-A), f(A \cap B) + f(A \cup B)\}$, as desired. ■

We will also need the following observation about proper functions.

Observation 2.0.3 If f is proper, and A,B, and C form a partition of V, then the maximum of $f(A)$,$f(B)$, and $f(C)$ is not uniquely attained.

Proof: Let C attain the maximum. The observation follows from the symmetry of f applied to $V - C$, and the maximality property applied to A,B, and $A \cup B = V - C$. ■

Corollary 2.0.4 If f is proper, then for disjoint sets A and B, the maximum of $f(A)$, $f(B)$, and $f(A \cup B)$ is not uniquely attained.

It turns out a slight variation on a weakly supermodular function is also weakly supermodular.

Lemma 2.0.5 If f is weakly supermodular and $x \in \mathbb{N}^{|E|}$, then $f(S) - x(\delta(S))$ is also weakly supermodular.

Proof: First, observe that $f(V) - x(\delta(V)) = 0$ since $\delta(V) = \emptyset$. It is well-known that $x(\delta(S))$ is both symmetric ($x(\delta(S)) = x(\delta(V - S))$) and submodular: that is, given $x \in \mathbb{N}^{|E|}$, for any two sets of vertices A and B,

$$x(\delta(A)) + x(\delta(B)) \geq x(\delta(A \cup B)) + x(\delta(A \cap B)).$$

Given symmetry, this also implies

$$x(\delta(A)) + x(\delta(B)) \geq x(\delta(A - B)) + x(\delta(B - A)).$$

The second property also implies the first. The lemma statement follows straightforwardly from these two inequalities. ■

A similar lemma holds for uncrossable functions.

Lemma 2.0.6 If h is uncrossable and $x \in \mathbb{N}^{|E|}$, then $h'(S) = \max\{h(S) - x(\delta(S)), 0\}$ is also uncrossable.

Proof: Pick any A and B such that $h'(A) = h'(B) = 1$. Then $h(A) = h(B) = 1$ and $x(\delta(A)) = x(\delta(B)) = 0$. By the submodularity of $x(\delta(S))$, $x(\delta(A - B)) = x(\delta(B - A)) = x(\delta(A \cap B)) = x(\delta(A \cup B)) = 0$. Since h is uncrossable, either $h(A - B) = h(B - A) = 1$ or $h(A \cup B) = h(A \cap B) = 1$, and thus the same statement holds true of h'. ■

Lemmas 2.0.5 and 2.0.6 lead to the following theorems, which are central to the design and analysis of the approximation algorithms. Recall that the strong oracle MAX-VIOLATED(f, x) returns a collection of maximally violated sets that are minimal with respect to inclusion; that is, it returns minimal sets S such that $f(S) - x(\delta(S)) = \max_T(f(T) - x(\delta(T))) > 0$.

Theorem 2.0.7 Let h be an uncrossable function, and $x \in \mathbb{N}^{|E|}$. Then the collection of sets returned by MAX-VIOLATED(h, x) are disjoint. Furthermore, no maximally violated set crosses any set in MAX-VIOLATED(h, x).

Proof: By Lemma 2.0.6, the function $h'(S) = \max\{h(S) - x(\delta(S)), 0\}$ is uncrossable. Since $h_{\max} = 1$, if there exist any violated sets then $h'(S) = 1$ exactly when S is a maximally violated set. Suppose MAX-VIOLATED returns two maximally violated sets A and B that are not disjoint. By the properties of h', either $A - B$ and $B - A$, or $A \cup B$ and $A \cap B$ must also be maximally violated, which contradicts the minimality of A and B. Similarly, if MAX-VIOLATED returns A, then no maximally violated set B crosses A; otherwise, the minimality of A is contradicted. ■

Corollary 2.0.8 Let f be a weakly supermodular function, and $x \in \mathbb{N}^{|E|}$. Then the collection of sets returned by MAX-VIOLATED(f, x) are disjoint. Furthermore, no maximally violated set crosses any set in MAX-VIOLATED(f, x).

Proof: By Lemma 2.0.5, $f'(S) = f(S) - x(\delta(S))$ is weakly supermodular, and by Observation 2.0.1, the function

$$h'(S) = \begin{cases} 1 & \text{if } f'(S) = f'_{\max} = \max\{f(S) - x(\delta(S))\} \\ 0 & \text{otherwise} \end{cases}$$

is uncrossable. The rest of the proof follows as above. ■

We conclude with a few further properties of uncrossable and proper functions.

Lemma 2.0.9 If h is uncrossable, then $h'(S) = \max\{h(S), h(V - S)\}$ is also uncrossable.

Proof: Suppose $h'(A) = 1$ is implied by $h(A) = 1$ and $h'(B) = 1$ is implied by $h(V - B) = 1$. Other cases will be similar. Since h is uncrossable then either $h((V - B) - A) = h(A - (V - B)) = 1$ or $h(A \cup (V - B)) = h(A \cap (V - B)) = 1$. The first case implies $h(V - (A \cup B)) = h(A \cap B) = 1$ and hence $h'(A \cup B) = h'(A \cap B) = 1$. The second case implies $h(V - (B - A)) = h(A - B) = 1$ and hence $h'(B - A) = h'(A - B) = 1$. ■

In the lemma above, h' represents a symmetric version of h. Notice that the optimal solutions for the integer program (IP) are the same for both h and h'. Forcing h to be symmetric, however, may destroy other properties of h. For example, if h obeys the maximality property, then it may not be the case that h' does also.

Observation 2.0.10 Let f be a proper function, and let c be a positive integer. Then the following functions are also proper:

- $\lceil f(S)/c \rceil$,
- $\lfloor f(S)/c \rfloor$,
- $\min\{f(S), c\}$,
- $\max\{f(S) - c, 0\}$.

Lemma 2.0.11 Let f be a proper function. Then $f_{\max} = \max\{f(v) | v \in V\}$.

Proof: Let S be a set such that $f(S) = f_{\max}$. The lemma statement follows from the maximality property applied to the vertices $v \in S$. ■

CHAPTER 3

The High-Level Algorithms

In this chapter, we give high-level descriptions of the algorithms to approximate weakly supermodular edge-covering problems (APPROX-WEAKLY-SUPERMODULAR) and uncrossable edge-covering problems (APPROX-UNCROSSABLE). We begin in Section 3.1 with a simplified version of APPROX-UNCROSSABLE for uncrossable functions h that are also proper. We then show how this algorithm generalizes to APPROX-UNCROSSABLE in Section 3.2. Finally, Section 3.3 shows how to reduce the problem of approximating weakly supermodular edge-covering problems to that of uncrossable edge-covering problems.

3.1 The Algorithm for Proper 0-1 Functions

We begin the chapter by giving an algorithm for a subclass of proper edge-covering problems, in which the proper function h has range $\{0, 1\}$. Notice that Theorem 2.0.2 and Observation 2.0.1 imply that any proper function with range $\{0, 1\}$ must also be uncrossable. Recall from Table 1.1 in the introduction that many interesting problems fall into this class, including Steiner trees and T-joins.

The algorithm consists of two stages. In the *edge addition* stage, the algorithm starts with an empty forest F and iteratively adds edges until the resulting forest F is a feasible solution for the integer program (IP) with the function h. Feasibility implies that there are

no violated sets with respect to h and F, where a *violated set* is a set S with $h(S) = 1$ and $\delta_F(S) = \emptyset$. In the *edge deletion* stage, the algorithm deletes redundant edges.

The algorithm is given as APPROX-PROPER-0-1 in Figure 3-1. It takes as input a vertex set V, an edge set E, non-negative costs c_e on each edge $e \in E$, and a proper function $h : 2^V \to \{0, 1\}$.

The algorithm uses the primal-dual method for approximation algorithms as it was outlined in the introduction. The primal integer program is (IP_h),

$$
\begin{aligned}
&\text{Min} && \sum_{e \in E} c_e x_e && \\
&\text{subject to:} && && \\
(IP_h) \qquad & && x(\delta(S)) \geq h(S) && S \subset V \\
& && x_e \geq 0 && e \in E, \\
& && x_e \text{ integer.} &&
\end{aligned}
$$

and the associated dual of the linear programming relaxation is (D_h),

$$
\begin{aligned}
&\text{Max} && \sum_{S} h(S) y_S && \\
&\text{subject to:} && && \\
(D_h) \qquad & && \sum_{S : e \in \delta(S)} y_S \leq c_e && e \in E \\
& && y_S \geq 0 && S \subset V.
\end{aligned}
$$

In the edge addition stage, the algorithm begins with the primal infeasible solution $F = \emptyset$ and the dual feasible solution $y_S = 0$ for all S. As long as there exists a connected component C such that $h(C) = 1$, there exists a violated set C with respect to h (since $\delta_F(C) = \emptyset$) and the primal solution is infeasible. In this case, the algorithm iteratively performs a primal-dual improvement step. Let $\mathcal{C}$ denote the collection of connected components C for which $h(C) = 1$. We will call these sets C *active* sets. In each *iteration*, the algorithm increases the value of the dual solution by uniformly raising the variables y_C corresponding to the

active sets $C \in \mathcal{C}$ until the dual constraint for some edge $e \in E$ becomes tight, i.e.,

$$c_e = \sum_{S:e \in \delta(S)} y_S.$$

Notice that a constraint must become tight for some edge e joining two different components, one of which is an active set. Such an edge e is then added to F, improving primal feasibility. If $h(C) = 0$ for all connected components C of F, then the algorithm goes on to the edge deletion stage, otherwise it iterates the primal-dual improvement step. We will show below that such a set of edges F must be a primal feasible solution. Thus at the end of the first stage, the algorithm has both a primal and a dual feasible solution for (IP_h) such that the primal complementary slackness conditions hold (that is, $c_e = \sum_{S:e \in \delta(S)} y_S$ for all $e \in F$).

The edge deletion stage of the algorithm removes redundant edges from F. This stage is necessary to obtain a good performance guarantee for the algorithm. When edge e is considered, it is removed from the current set F if $h(C) = 0$ for all connected components of $F - e$. At the end of the edge deletion stage, the remaining set of edges, F', is still primal feasible and the primal complementary slackness conditions still hold. In addition, the dual complementary slackness conditions will now hold in an average sense. Proving this fact will result in the proof of the performance guarantee of the algorithm (i.e., Theorem 1.2.6), as will be shown in Chapter 4.

The algorithm is formally described in Figure 3-1. We now need to show that the algorithm given in the figure behaves as we have described. We keep track of the connected components of the edge set F in the sets $\mathcal{C}$ and $\mathcal{I}$. A component N is in $\mathcal{C}$ if $h(N) = 1$, and is in $\mathcal{I}$ otherwise. To see that the dual solution generated in steps 2 and 11 is feasible for (D_h), note first that initially $\sum_{e \in \delta(S)} y_S = 0 \leq c_e$ for all $e \in E$. We show by induction that these constraints continue to hold. Notice that it can be shown by induction that $d(v) = \sum_{S:v \in S} y_S$ for each vertex v; thus as long as vertices u and v are in different components, $\sum_{e \in \delta(S)} y_S = d(u) + d(v)$ for any edge $e = (u, v)$. Define $a(u) = 1$ if u is in an active set in the current iteration. It follows that in this iteration y_C can be increased by ϵ for each

APPROX-PROPER-0-1 (V, E, c, h)

1 $F \leftarrow \emptyset$

2 *Comment: Implicitly set* $y_S \leftarrow 0$ *for all* $S \subset V$

3 $\mathcal{C} \leftarrow \{\{v\} : v \in V, h(\{v\}) = 1\}$

4 $\mathcal{I} \leftarrow \{\{v\} : v \in V, h(\{v\}) = 0\}$

5 $a(v) \leftarrow h(\{v\})$ for all $v \in V$

6 For each $v \in V$ do $d(v) \leftarrow 0$

7 **while** $|\mathcal{C}| > 0$

8 Find edge $e = (u, v)$ with $u \in C_p \in \mathcal{C}$, $v \in C_q \in \mathcal{C} \cup \mathcal{I}$, $C_p \neq C_q$ that minimizes $\epsilon = \frac{c_e - d(u) - d(v)}{a(u) + a(v)}$

9 $F \leftarrow F \cup \{e\}$

10 For all $v \in C_r \in \mathcal{C}$ do $d(v) \leftarrow d(v) + \epsilon$

11 *Comment: Implicitly set* $y_C \leftarrow y_C + \epsilon$ *for all* $C \in \mathcal{C}$

12 Delete C_p and C_q from $\mathcal{C}$ and $\mathcal{I}$

13 **if** $h(C_p \cup C_q) = 1$

14 $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_p \cup C_q\}$

15 $a(v) \leftarrow 1$ for all $v \in C_p \cup C_q$

16 **else**

17 $\mathcal{I} \leftarrow \mathcal{I} \cup \{C_p \cup C_q\}$

18 $a(v) \leftarrow 0$ for all $v \in C_p \cup C_q$

19 *Comment: Edge deletion stage:* PROPER-0-1-EDGE-DELETE

20 $F' \leftarrow \{e \in F :$ For some connected component N of $(V, F - \{e\}), h(N) = 1\}$

21 **return** F'

Figure 3-1: The algorithm for proper functions h with range {0,1}.

active set C without violating the dual constraints as long as

$$d(u) + d(v) + \epsilon \cdot a(u) + \epsilon \cdot a(v) \leq c_e,$$

for all $e = (u, v)$ such that u and v are in different components. Thus the largest feasible increase in ϵ for a particular iteration is given by the formula in step 8. Once the endpoints u and v of an edge $e = (u, v)$ are both contained in the same active set C, then $\sum_{S:e \in \delta(S)} y_S$ does not increase. Hence when the algorithm terminates, the dual solution y constructed by the algorithm will be feasible for (D_h).

We have not discussed how various parts of this algorithm can be implemented efficiently. In particular, we have not said how to find the edge minimizing ϵ in step 8, nor how the edge deletion step can be carried out. We return to these questions in Chapter 5.

We now show that a set of edges A is a primal feasible solution if $h(N) = 0$ for all connected components N of A. This shows that the edge set F obtained at the end of the edge addition stage is feasible. We then show that $h(N) = 0$ for all connected components N of F', proving that the final set of edges produced by the algorithm is also primal feasible.

Observation 3.1.1 If h is a proper function, and $h(S) = 0$ and $h(B) = 0$ for some $B \subseteq S$, then $h(S - B) = 0$.

Proof: By Corollary 2.0.4. If $h(S - B) = 1$, then the maximum of $S - B$, B, and S is uniquely attained. ■

Theorem 3.1.2 Let A be a set of edges such that $h(N) = 0$ for all connected components N of A. Then the edge set A is a feasible solution to (IP_h).

Proof: Suppose that S is violated given the edge set A; that is, $\delta_A(S) = \emptyset$ and $h(S) = 1$. Let $N_1, \ldots, N_p$ be the components of A. In order for $\delta_A(S) = \emptyset$, it must be the case that for all i, either $S \cap N_i = \emptyset$ or $S \cap N_i = N_i$. Thus $S = N_{i_1} \cup \ldots N_{i_k}$ for some $i_1, \ldots, i_k$. By assumption, $h(N_i) = 0$ for all i, so $h(S) = 0$ by the maximality of h. This contradicts our assumption that $h(S) = 1$. ■

Theorem 3.1.3 For each connected component N of F', $h(N) = 0$.

Proof: By the construction of F', $N \subseteq C$ for some component C of F. Note that at the end of the edge addition stage, $h(C) = 0$ for any such C. Now, let $e_1, \ldots, e_k$ be edges of F such that $e_i \in \delta(N)$ (possibly $k = 0$). Let N_i and $C - N_i$ be the two components created by removing e_i from the edges of component C, with $N \subseteq C - N_i$ (see Figure 3-2). Note that since $e_i \notin F'$, it must be the case that $h(N_i) = 0$. Note also that the sets $N, N_1, N_2, \ldots, N_k$ form a partition of C. So then $h(C - N) = h(\cup_{i=1}^{k} N_i) = 0$ by maximality. Since $h(C) = 0$, Observation 3.1.1 implies that $h(N) = 0$. ■

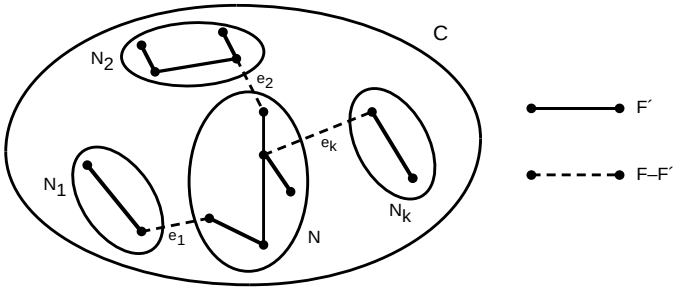

Figure 3-2: Illustration of Theorem 3.1.3.

We show an example of the algorithm at work in Figures 3-3 through 3-11. The instance is given by points in the Euclidean plane: each point represents a vertex, and the cost of edge (i, j) is given by the Euclidean distance between points i and j. The proper function being used is $h(S) = |S| (\text{mod } 2)$. This function is symmetric as long as $|V|$ is even. The function also obeys the maximality property since whenever $|A|$ and $|B|$ are both even for disjoint sets A and B, then $|A \cup B|$ is also even. In this geometric instance, the variable $d(v)$ for vertex v can be represented by a circle of radius $d(v)$ around the point v. Connected components in $\mathcal{C}$ are surrounded by thick-lined circles, while components in $\mathcal{I}$ are given by thin-lined circles. Increasing the dual variables for each active component uniformly reduces to increasing the circles around the vertices in each active component uniformly. When two circles around vertices u and v from different components touch each other, then the packing constraint for edge (u, v) is tight, and we add edge (u, v) to F. The region of the plane defined by these circles correspond to the "moats" of Jünger and Pulleyblank [67].

We can now show that Approx-Proper-0-1 is a generalization of some classical graph algorithms. The shortest s-t path problem corresponds to the proper function $h(S) = 1$ if and only if $|S \cap \{s, t\}| = 1$. The algorithm adds minimum-cost edges extending paths from both s and t in a manner reminiscent of Nicholson's bidirectional shortest path algorithm [95]. The main loop terminates when s and t are in the same component, and the edge deletion stage removes all edges not on the path from s to t. Thus for this problem, whenever $y_S > 0$, $|F' \cap \delta(S)| = 1$, and whenever $e \in F'$, $\sum_{S: e \in \delta(S)} y_S = c_e$. In other words, the primal and dual feasible solutions F' and y obey the complementary slackness conditions; hence the

Figure 3-3: Initial instance of example of Approx-Proper-0-1.

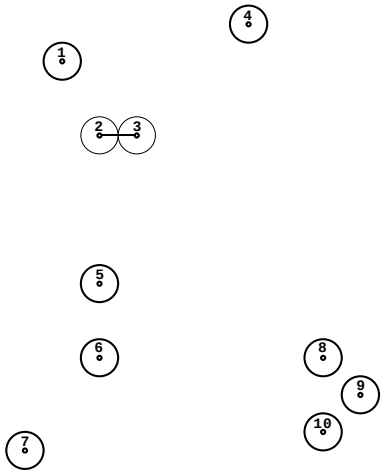

Figure 3-4: Iteration 1 in example of Approx-Proper-0-1.

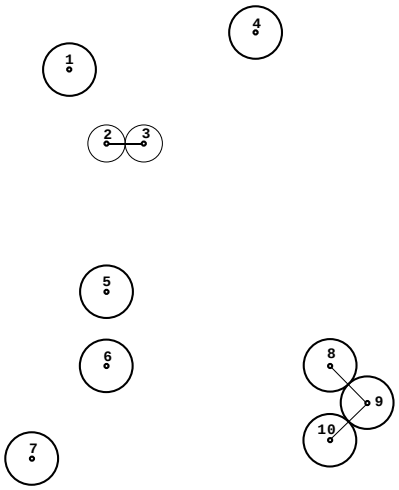

Figure 3-5: Iterations 2 and 3 in example of APPROX-PROPER-0-1.

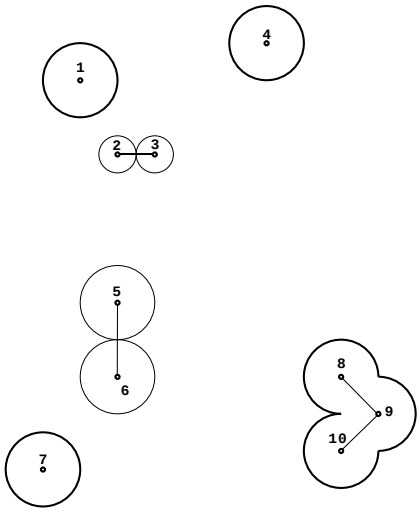

Figure 3-6: Iteration 4 in example of APPROX-PROPER-0-1.

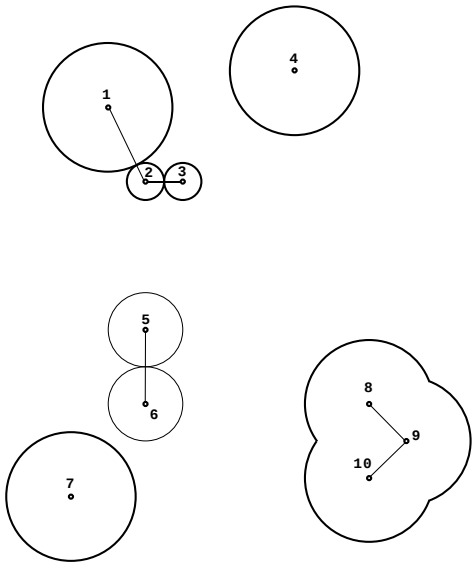

Figure 3-7: Iteration 5 in example of Approx-Proper-0-1.

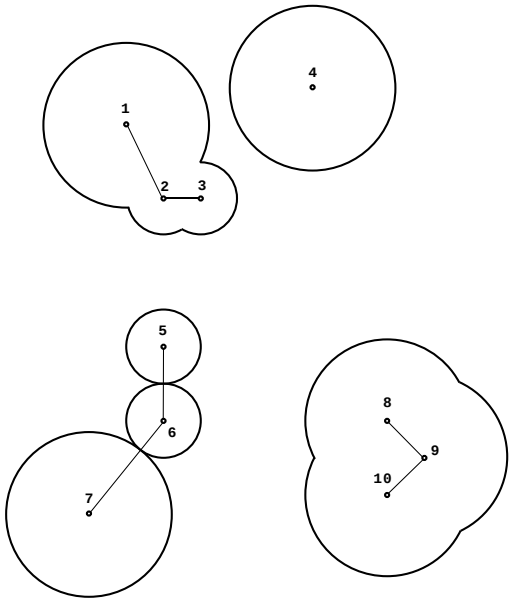

Figure 3-8: Iteration 6 in example of Approx-Proper-0-1.

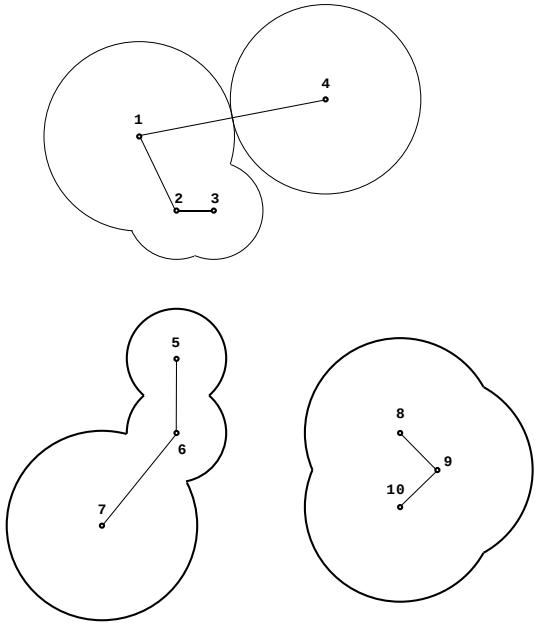

Figure 3-9: Iteration 7 in example of APPROX-PROPER-0-1.

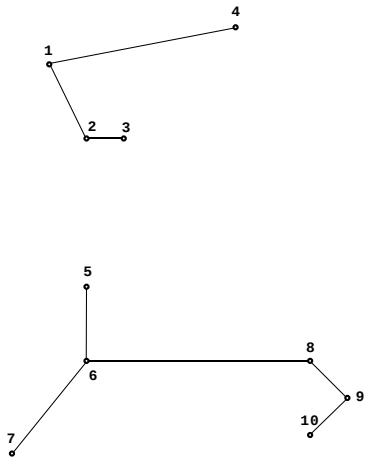

Figure 3-10: The set of edges F in example of APPROX-PROPER-0-1.

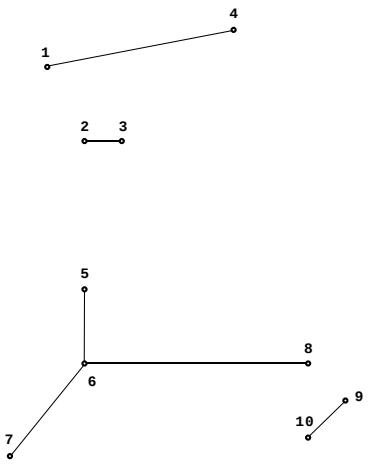

Figure 3-11: The final set of edges F' in example of APPROX-PROPER-0-1.

solutions are optimal. The minimum-cost spanning tree problem corresponds to a proper function $h(S) = 1$ for $\emptyset \subset S \subset V$. For this function h, our algorithm reduces to Kruskal's algorithm [77]: all connected components will always be active, and thus in each iteration the minimum-cost edge joining two components will be selected. Since Kruskal's algorithm produces the optimal minimum-cost spanning tree, our algorithm will also. The solutions produced do not obey the complementary slackness conditions, but induce optimal solutions for a stronger linear programming formulation of the spanning tree problem introduced by Jünger and Pulleyblank [67].

We conclude the section by observing that in any iteration of the algorithm, the set of connected components in $\mathcal{C}$ are exactly those sets that would be returned by the strong oracle MAX-VIOLATED(h, F). Recall that the strong oracle returns the minimal sets S such that $h(S) - |\delta_F(S)| = \max_T(h(T) - |\delta_F(T)|) > 0$. By a "minimal" violated set, we mean that none of its proper subsets are violated. For an uncrossable function h, these sets are the minimal sets S such that $h(S) = 1$ and $\delta_F(S) = \emptyset$; that is, the sets returned by the strong oracle are the minimal violated sets.

Theorem 3.1.4 Given a proper function h with range $\{0,1\}$, the sets $C \in \mathcal{C}$ correspond exactly to the minimally violated sets in any iteration of the algorithm.

Proof: First, it is not too hard to see that any set $C \in \mathcal{C}$ must be active: since each set C represents a connected component of F in this iteration, no subset of C is violated. Thus each $C \in \mathcal{C}$ is a minimally violated set since $h(C) = 1$. Now we must show that no other minimally violated sets exist. Any such set cannot contain any $C \in \mathcal{C}$ (otherwise it would not be minimal), nor can it intersect any connected component of F (otherwise it would not be violated). Hence any such set S must be the union of some subset of the sets in $\mathcal{I}$. Each set $I \in \mathcal{I}$ has $h(I) = 0$, and so by the maximality property of proper functions $h(S) = 0$. Thus S is not a violated set. ■

3.2 The Algorithm for Uncrossable Edge-Covering Problems

3.2.1 The Main Algorithm

In this section, we generalize the algorithm of the previous section to handle all uncrossable edge-covering problems. The algorithm, Approx-Uncrossable, is given in Figure 3-12. The basic structure of the algorithm is exactly the same as before. There is an edge addition stage, which adds edges to F until there are no violated sets left. Thus F is a primal feasible solution for the integer program (IP_h). The edges in each iteration of the algorithm are chosen using the primal-dual method. The edge addition stage is followed by an edge deletion stage in which redundant edges in F are removed, in order to ensure a good performance guarantee.

There are two central ways in which Approx-Uncrossable differs from Approx-Proper-0-1, the first in the edge addition stage and the second in the edge deletion stage. In the edge addition stage of the Approx-Proper-0-1 algorithm, we focused our attention on the connected components of the graph. In particular, in each iteration of the algorithm, we increased the dual variables y_C of each connected component C such that $h(C) = 1$. The sets C were called active sets. In each iteration of Approx-Uncrossable, the active sets will be the minimal violated sets, as given by the strong oracle Max-Violated(h, F). Theorem 3.1.4 shows that this notion of active sets is a generalization of the active sets

of the previous algorithm. Let $\mathcal{C}$ denote the collection of active sets in an iteration of the algorithm. Then the primal-dual improvement step for APPROX-UNCROSSABLE increases the dual variables y_C uniformly for all $C \in \mathcal{C}$ until the dual constraint for some edge $e \in E$ becomes tight, i.e.,

$$c_e = \sum_{S:e \in \delta(S)} y_S.$$

A constraint must become tight for some edge $e \in \delta(C)$ of an active set C. As before, the edge e is then added to F, improving primal feasibility. The edge addition stage terminates when there are no minimally violated sets, and hence no violated sets, implying that the edge set F is feasible for (IP_h). As before, we will also have a dual feasible solution y such that the primal complementary slackness conditions hold; that is, $c_e = \sum_{S:e \in \delta(S)} y_S$ for all $e \in F$.

The edge deletion stage of APPROX-UNCROSSABLE differs from that of APPROX-PROPER-0-1 in that the edges are considered for deletion in the reverse of the order in which they were added to F. When edge e is considered, it is removed from the current set F if $F - e$ is still a primal feasible solution. At the end of the edge deletion stage, the remaining set of edges, F', is still primal feasible, the primal complementary slackness conditions still hold, and the dual complementary slackness conditions will hold in an average sense, implying a good performance guarantee. It turns out that other edge deletion routines are possible which still have a good performance guarantee; we will provide one in the next section which leads to a more efficient algorithm.

The algorithm APPROX-UNCROSSABLE is formally described in Figure 3-12. It takes as input a vertex set V, an edge set E, non-negative costs c_e on each edge $e \in E$, and an uncrossable function h. It returns a feasible set of edges F' for the integer program (IP_h). Before we can prove that the algorithm works as we have described above, we need to establish a few lemmas about the behavior of active sets. A central fact that is used both by the algorithm and in its analysis is that the minimal violated sets with respect to h are disjoint. This fact is a consequence of Theorem 2.0.7. We restate a part of the theorem here that will be useful.

Lemma 3.2.1 Let h be an uncrossable function and let $F \subseteq E$. If A and B are violated sets

APPROX-UNCROSSABLE (V, E_h, c, h)

1 $F \leftarrow \emptyset$

2 *Comment: Implicitly set* $y_S \leftarrow 0$ *for all* $S \subset V$

3 $i \leftarrow 0$

4 $d(v) \leftarrow 0$ for all $v \in V$

5 $\mathcal{C} \leftarrow$ MAX-VIOLATED$(h, \emptyset)$

6 $a(v) \leftarrow \begin{cases} 1 & \text{if } v \in C \in \mathcal{C} \\ 0 & \text{otherwise} \end{cases}$ for all $v \in V$.

7 **while** $|\mathcal{C}| > 0$

8 $\quad i \leftarrow i + 1$

9 $\quad$ *Comment: Begin iteration* i.

10 $\quad$ Find edge $e_i = (u, v)$ with $e_i \in \delta(C)$ for some $C \in \mathcal{C}$ that minimizes $\epsilon = \frac{c_e - d(u) - d(v)}{a(u) + a(v)}$

11 $\quad$ For all $v \in V$ do $d(v) \leftarrow d(v) + \epsilon \cdot a(v)$

12 $\quad$ *Comment: Implicitly set* $y_C \leftarrow y_C + \epsilon$ *for all* $C \in \mathcal{C}$

13 $\quad F \leftarrow F \cup \{e_i\}$

14 $\quad \mathcal{C} \leftarrow$ MAX-VIOLATED(h, F)

15 $\quad$ Update $a(v)$

16 $\quad$ *Comment: End iteration* i.

17 *Comment: Edge deletion stage:* REGULAR-EDGE-DELETE

18 $F' \leftarrow F$

19 **for** $j \leftarrow i$ **downto** 1

20 $\quad$ If $F' - \{e_j\}$ is a feasible solution

21 $\quad\quad F' \leftarrow F' - \{e_j\}$

22 **return** F'

Figure 3-12: The algorithm for uncrossable functions h.

with respect to h and F, then either $A - B$ and $B - A$ are violated, or $A \cap B$ and $A \cup B$ are violated.

Proof: Define $h'(S) = \max\{h(S) - |\delta_F(S)|, 0\}$. Then $h'(S) = 1$ if and only if S is violated. The lemma statement follows from Lemma 2.0.6, which shows that h' is uncrossable. ■

Let $\cup\mathcal{C}$ denote the family of sets formed by taking all active sets over all iterations of the algorithm. We say a family of sets is *laminar* if no two sets in the family are crossing. Recall that two sets A and B cross if $A \cap B \neq \emptyset$ and neither $A \subseteq B$ nor $B \subseteq A$.

Lemma 3.2.2 The family of sets $\cup\mathcal{C}$ is laminar.

Proof: Suppose there exist two sets $C, C' \in \cup\mathcal{C}$ that cross, such that C was an active set in iteration i and C' in iteration j, with $i \leq j$. But then C' must have been a violated set in iteration i, and by Lemma 3.2.1 this would contradict the minimality of C. ■

Thus if in some iteration we select edge e in the coboundary of a currently active set C, then the new active set that is formed in the next iteration (if any) must contain C. At most one other active set C' in the current iteration might contain e in its coboundary, and by the lemma any active set that contains C in some future iteration must contain C' as well.

We now show that the algorithm given in the figure behaves as we have described. As with Approx-Proper-0-1, we prove by induction on the iterations of the algorithm that the dual solution generated in steps 2 and 12 is feasible for (D_h). The base case of the induction is trivial. As in Approx-Proper-0-1, define $a(u) = 1$ if u is in an active set in the current iteration. Then one can show again by induction that $d(v) = \sum_{S:v\in S} y_S$ for each vertex v, since we increase $d(v)$ by $a(v) \cdot \epsilon$ in each iteration, and the active sets are always disjoint. Lemma 3.2.2 implies that for any edge $e = (u, v) \in \delta(C)$ for some active set C, the vertices u and v have not been contained in the same active set in any previous iteration. Thus $\sum_{e\in\delta(S)} y_S = d(u) + d(v)$ for any such edge $e = (u, v)$. It follows that in this iteration y_C can be increased by ϵ for each active set C without violating the dual constraints as long as

$$d(u) + d(v) + \epsilon \cdot a(u) + \epsilon \cdot a(v) \leq c_e,$$

for all $e = (u, v) \in \delta(C')$ of some active set C'. Thus the largest feasible increase in ϵ for a particular iteration is given by the formula in step 10. Once the endpoints u and v of an edge $e = (u, v)$ are both contained in the same active set C, then $\sum_{S:e\in\delta(S)} y_S$ does not increase. To see this, note that in any future iteration any set with e in its coboundary will not be violated. Hence when the algorithm terminates, the dual solution y constructed by the algorithm will be feasible for (D_h).

We have still not explained how to select the edge minimizing ϵ in step 10. We also have not explained how MAX-VIOLATED can be implemented. We again defer these issues to Chapter 5. We use the MAX-VIOLATED oracle to implement the edge deletion stage in step 20. Suppose we wish to check whether $F' - \{e_j\}$ is a feasible solution given that F' is a feasible solution. We can simply call MAX-VIOLATED$(h, F' - e_j)$ and see whether it returns a violated set.

To conclude the section, we would like to point out that, for a given edge-covering problem, the behavior of APPROX-UNCROSSABLE depends on the choice of the uncrossable function h used to model the problem. For example, we have seen in Chapter 2 that if one "symmetrizes" an uncrossable function h, then the resulting function h' (defined by $h'(S) = \max\{h(S), h(V - S)\}$) is uncrossable, symmetric, and defines the same edge-covering problem. However, APPROX-UNCROSSABLE behaves differently on h and h' since the minimal violated sets may differ. The minimal violated sets are closely related, however, as is shown in the following lemma.

Lemma 3.2.3 Let h be an uncrossable function, and $h' = \max\{h(S), h(V - S)\}$. Either MAX-VIOLATED$(h', F)$ = MAX-VIOLATED(h, F) or there exists a set S such that MAX-VIOLATED(h', F) = MAX-VIOLATED$(h, F) \cup \{S\}$.

Proof: We first show that MAX-VIOLATED$(h', F) \supseteq$ MAX-VIOLATED(h, F). If $A \in$ MAX-VIOLATED(h, F) but $A \notin$ MAX-VIOLATED(h', F), then there must exist a set $B \subset A$ in MAX-VIOLATED(h', F). Furthermore, it must be the case that B is violated with respect to h', $h(B) = 0$ and $h(V - B) = 1$. But then A crosses $V - B$, which contradicts the minimality of A with respect to h by Lemma 3.2.1.

Assume that there exists two sets $A, B \in$ MAX-VIOLATED$(h', F) -$ MAX-VIOLATED(h, F).

This implies that $h(A) = h(B) = 0$, but $h(V - A) = h(V - B) = 1$. Since A and B must be disjoint, then $h((V - A) \cup (V - B)) = h(V) = 0$. Thus the fact that h is uncrossable implies that $h((V - A) - (V - B)) = h(B) = 1$ and $h((V - B) - (V - A)) = h(A) = 1$, a contradiction. ■

In order to apply APPROX-UNCROSSABLE to a symmetrized function h', we must have access to an oracle MAX-VIOLATED for h'. In general, one can simulate a call to MAX-VIOLATED(h', F) by $O(n)$ calls to the oracle MAX-VIOLATED for h. For some problems of interest, however, we can implement an oracle MAX-VIOLATED for h' directly; we will see an example of this in Section 7.1.

As an example of the difference between a symmetric and a non-symmetric function h, we again consider the shortest s-t path problem. In the previous section, it was observed that this problem could be modelled by the proper function $h'(S) = 1$ if $|S \cap \{s, t\}| = 1$ and $h'(S) = 0$ otherwise. We can also model the problem by the uncrossable function $h(S) = 1$ iff $s \in S, t \notin S$. Note that h' is a symmetrization of h. By the same complementary slackness argument as given in the previous section, APPROX-UNCROSSABLE will produce an optimal solution to this problem. The two functions behave differently in that APPROX-PROPER-0-1 behaves like Nicholson's algorithm for the shortest s-t path problem given the function h'; with the function h, APPROX-UNCROSSABLE will emulate Dijkstra's algorithm [29]. To see this, one can show by induction that when APPROX-UNCROSSABLE is run on function h, the edge set F at any iteration will form a single connected component C containing the vertex s, and C will be the sole active set. The edge addition stage terminates when C contains t. For any vertex $u \in C$, the value $d(s) - d(u)$ will give the length of the shortest path from s to u. In each iteration, the algorithm selects the edge $e = (u, v)$, $u \in C$, $v \notin C$, that minimizes $c_e - d(u)$ (as $d(v) = 0$ for $v \notin C$). Since this is equivalent to minimizing $c_e + d(s) - d(u)$, the algorithm effectively selects the vertex $v \notin C$ adjacent to a vertex $u \in C$ that minimizes the shortest path from s to v; that is, the algorithm exactly mimics the behavior of Dijkstra's algorithm. Then $d(s)$ is increased by $c_e - d(u)$, so that $d(s) - d(v)$ is the length of the shortest path from s to v. The variables $d(i)$ for $i \in C$ are also increased by this amount, maintaining the values $d(s) - d(i)$. Finally, (u, v) is added to F, effectively adding v to C.

In the next section, we turn to a more efficient variation of Approx-Uncrossable. The ideas required to make Approx-Uncrossable efficient are somewhat involved, and so the first-time reader may wish to advance to Section 3.3, which gives the algorithm for weakly supermodular edge-covering problems, and return to the efficient version of Approx-Uncrossable at a later point.

3.2.2 A More Efficient Variation

As was stated in the introduction, the edge-covering problems of interest are primarily proper edge-covering problems. Although the edge addition and the edge deletion stages given above both use $O(n)$ calls to Max-Violated, we will show in Chapter 5 that the sequence of calls of the edge addition stage can be implemented rather efficiently when Approx-Uncrossable is called as a subroutine of our algorithm for proper edge-covering problems. The bottleneck of the approximation algorithm for proper edge-covering problems becomes implementing the $O(n)$ calls to Max-Violated by the edge deletion stage, Regular-Edge-Delete.

We can, however, give another edge deletion stage, Efficient-Edge-Delete, which runs in $O(n)$ time, makes no calls to Max-Violated, and still yields a feasible solution with the same performance guarantee. In a sense, this edge deletion stage removes as few edges as possible while still giving a good performance guarantee. This contrasts with Regular-Edge-Delete, which removes as many edges as possible while still giving a feasible solution. We will return to this intuitive characterization of Efficient-Edge-Delete when we prove the performance guarantee in Chapter 4, and give only its formal description here.

One of the consequences of this kind of edge deletion stage is that it does not necessarily return edge-minimal solutions. As far as we know, this fact has no negative implications other than that on average we would expect that an algorithm using Efficient-Edge-Delete would give solutions with somewhat greater cost than an algorithm using Regular-Edge-Delete.

Before we define the new edge deletion stage, we define some notation and we try to provide an intuitive feel for the concepts involved. Suppose for a moment that the

uncrossable function h under consideration is such that no new active sets are ever created: we have some initial collection $\mathcal{C}$ of active sets, and in each iteration we select some edge in the coboundary of at least one, and at most two, active sets from $\mathcal{C}$. If the edge is in the coboundary of exactly one active set, we call it a *1-edge*, otherwise it is a *2-edge*. Once an edge in the coboundary of an active C is selected, then C is no longer violated, and hence no longer active. Thus in this case, there will be at most $|\mathcal{C}|$ iterations of the edge addition step. Let F be the set of edges added during these iterations.

Still assuming that no new active sets are created, we can consider F as defining a forest on the initial collection of active sets (see Figure 3-13). Consider also how each tree of this forest "grows" as edges of F are added. Each component adds a "node" by adding 1-edges which have one endpoint in a currently active set (the new "node") and the other in a set that was active in some previous iteration (a "node" in the growing component); see Figure 3-14. Notice that we never add edges whose endpoints are both in previously active sets, and thus we never link two components. From this observation, it is not hard to see that 2-edges must always start growing a new component.

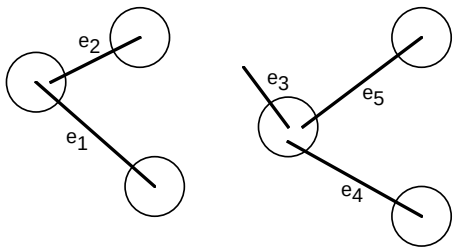

Figure 3-13: A "forest" on active sets. Each circle represents an active set.

The edges that particularly interest us for the new edge deletion step are 1-edges e such that the edges added after e form a tree on the sets active before the addition of e. We call these edges *special edges*, and we define them more formally below. In Figure 3-13, e_3 and e_5 are special edges. The other edges are not special edges: for example, e_4 is not a special edge because e_5 contains an endpoint not in a set that is active just before e_4 is added. We will show that special edges have a nice combinatorial structure such that we can remove some of them and simultaneously ensure a good performance guarantee and

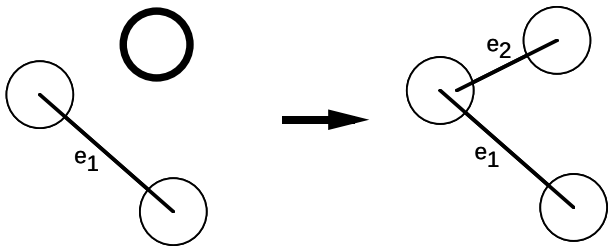

Figure 3-14: Growing a component of the forest. The thin circles represent previously active sets; the thick circle represents an active set.

maintain feasibility.

To generalize this concept to the case where new active sets are formed, we partition the edges and the active sets. Let $\cup\mathcal{C}$ be the collection of all active sets formed over all iterations of the algorithm, and augment $\cup\mathcal{C}$ by adding the set V to the collection. Recall from Lemma 3.2.2 that $\cup\mathcal{C}$ is laminar. We define a tree $\mathcal{T}$ based on $\cup\mathcal{C}$, with one vertex v_C for each $C \in \cup\mathcal{C}$. Thus we make v_C a parent of v_D in the tree $\mathcal{T}$ if C is the smallest set in $\cup\mathcal{C}$ that properly contains D. Let $\mathcal{D}(C)$ denote the collection of sets corresponding to the children of v_C in $\mathcal{T}$. The collection $\mathcal{D}(C)$ can be thought of as an equivalence class of the active sets. For edges, let $C(e)$ denote the smallest set $C \in \cup\mathcal{C}$ that contains both endpoints of e. The set of all edges of F for which $C(e) = C$ is denoted F_C. The edges in F_C can be thought of as an equivalence class of the edges of F. The behavior of the edges F_C on the active sets in $\mathcal{D}(C)$ will now be as in the case above in which no new active sets are formed.

We can now formally define the special edges. Let $\mathcal{A}(e)$ denote the sets in $\mathcal{D}(C(e))$ that are active just before edge e is selected. Say that an edge set H forms a spanning tree on a family of disjoint vertex sets $\{C_i\}_{i=1}^{k}$ if H forms a spanning tree on the graph induced by considering each set C_i as a vertex. Then e is special if it is a 1-edge, and the edges added to $F_{C(e)}$ after e form a spanning tree on the sets in $\mathcal{A}(e)$.

We illustrate the concept in its full generality in Figure 3-15. Frames 1–9 correspond to the edge addition stage, frames 13–16 to EFFICIENT-EDGE-DELETE, while the final solution is depicted in frame 11. In frames 1–9, the edges correspond to the edges of F while the

rounded boxes represent the active sets. In the figure, only edges e_4 and e_8 are 2-edges, the others being 1-edges. The special edges in the figure are e_1, e_5, e_6 and e_7. The edge e_2 is not special because e_4 is a 2-edge (and hence forms a new "component" on the sets of $\mathcal{A}(e_2)$). The edge e_3 is not special because e_5 has an endpoint not in one of the sets of $\mathcal{A}(e_3)$.

The new edge deletion stage is given in Figure 3-16. It scans the edges of F in the reverse order of their selection in the edge addition stage. Let $A(e)$ denote the union of the sets in $\mathcal{A}(e)$; that is, $A(e) = \bigcup_{C \in \mathcal{A}(e)} C$. The edge deletion stage removes edge e from F if e is special, no other edge of $F_{C(e)}$ has already been removed, and all remaining edges of F in $\delta(C(e))$ are also in $\delta(A(e))$. It does not remove e if $C(e) = V$. We illustrate EFFICIENT-EDGE-DELETE in frames 13–16 of Figure 3-15. Recall that the special edges are e_1, e_5, e_6 and e_7. Frames 13, 14, 15 and 16 correspond to the situation just before the possible removal of edge e_7, e_6, e_5 and e_1 respectively. In the frame corresponding to e_i, the vertices in $A(e_i)$ are represented in white. Edge e_7 is removed since $e_8 \in \delta(A(e_7))$. Edge e_6 is not removed since $e_8 \notin \delta(A(e_6))$. Although $e_7 \notin \delta(A(e_5))$, edge e_5 is removed since e_7 was previously removed. Edge e_1 is not removed since $e_5 \in C(e_1)$ has already been removed. The resulting forest is represented in frame 11.

We conclude the section by proving that EFFICIENT-EDGE-DELETE delivers a feasible solution for the integer program (IP_h), and save the proofs of performance guarantee and running time for Chapters 4 and 5 respectively.

Theorem 3.2.4 The edge set F' remaining after the edge deletion stage is a feasible solution for (IP_h).

Proof: The edge addition stage terminates with no active sets, and thus no violated sets. So we need only prove that EFFICIENT-EDGE-DELETE maintains feasibility. Assume it does not. Suppose F' is feasible for (IP_h) but $F' - e$ is not, and the edge deletion step removes e from F'. The situation just before the removal of e is illustrated in Figure 3-17. Let $C = C(e)$. By the definition of EFFICIENT-EDGE-DELETE, $C \neq V$. Let S be a set violated by $F' - e$. We call the iteration of the edge addition step in which e was added the "current iteration". The set S was violated in the current iteration because of the ordering of the edge deletion step.

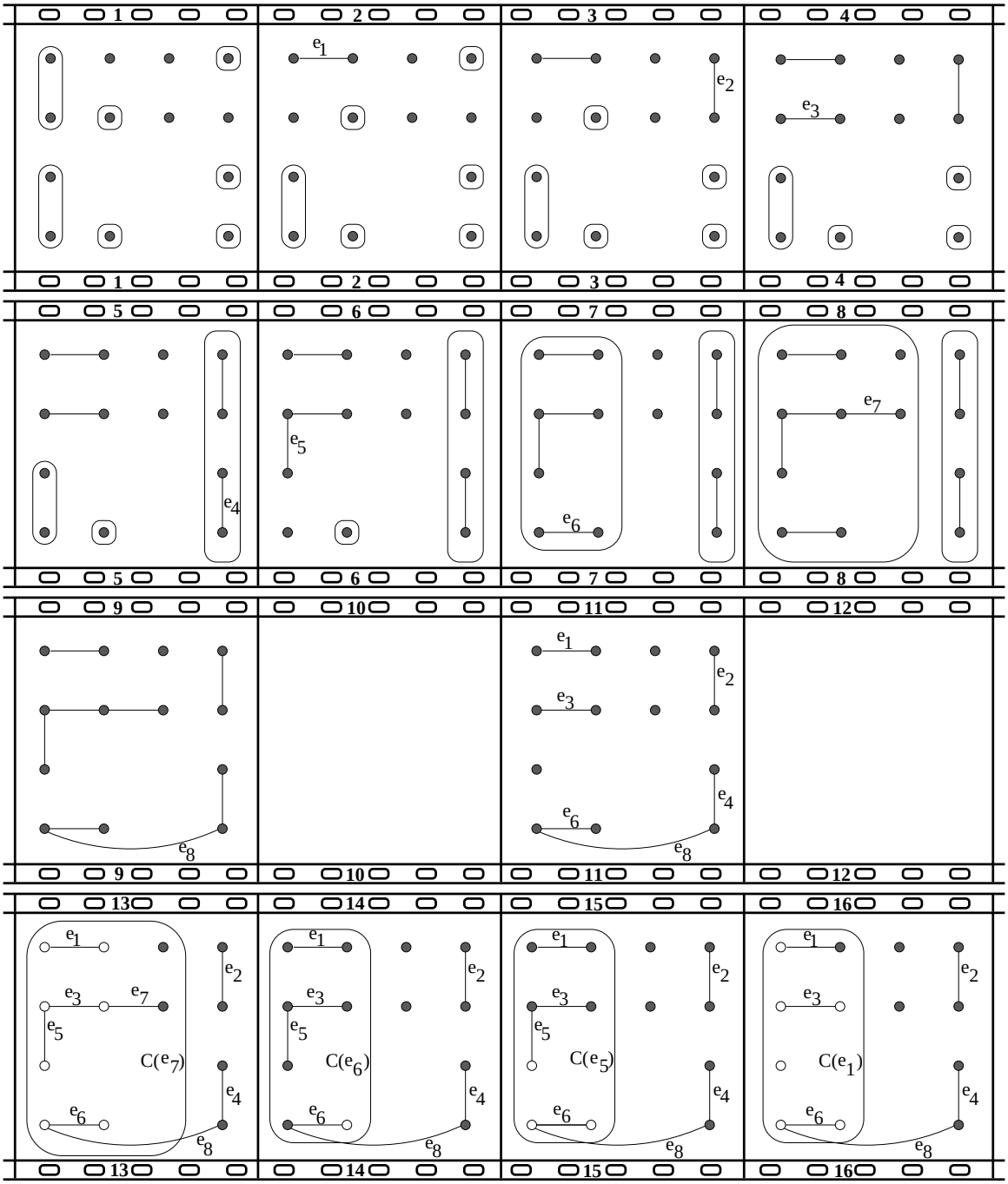

Figure 3-15: Simulation run of the algorithm.

EFFICIENT-EDGE-DELETE

1 $F' \leftarrow F$
2 Let all sets that were active be unmarked. Mark the set V.
3 **for** $j \leftarrow i$ **downto** 1
4 If e_j is special and $C(e_j)$ is unmarked and $\delta_{F'}(A(e_j)) \supseteq \delta_{F'}(C(e_j))$ then
5 $F' \leftarrow F' - \{e_j\}$
6 Mark $C(e_j)$
7 **return** F'

Figure 3-16: A more efficient edge deletion stage.

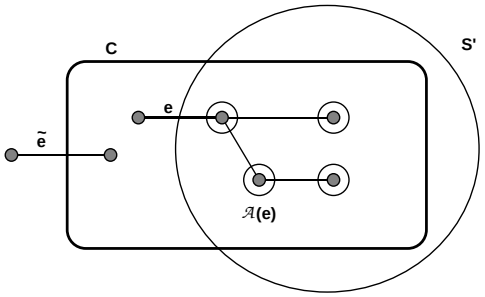

Figure 3-17: Notation used in the proof of Theorem 3.2.4.

Notice that all sets in $\mathcal{A}(e)$ must also be violated in the current iteration, by the definition of $\mathcal{A}(e)$. By Lemma 3.2.1, there exists a violated set S' that does not cross any set in $\mathcal{A}(e)$. In fact we can show that S' does not cross $A(e)$: because no edge of F_C has been removed so far in the edge deletion process, the edges of F_C added after e form a spanning tree on the family $\mathcal{A}(e)$. Thus if S' crosses $A(e)$ but not any set in $\mathcal{A}(e)$, it would intersect an edge of F_C added after e, contradicting the fact that S' is violated.

We now assume that $A(e) \subseteq S'$; the case that $A(e) \subseteq V - S'$ is similar. Since $C \neq V$, the set C, as well as S', is violated just before e was added. By Lemma 3.2.1, either $C - S'$ and $S' - C$ are violated or $C \cap S'$ and $C \cup S'$ are violated just before e was added. However $C - S'$ cannot contain an active set, so it is not violated. Thus $C \cup S'$ is violated. Since it was not violated before e was removed, F' contains an edge $\tilde{e}$ with exactly one endpoint in $V - (C \cup S')$. The edge $\tilde{e}$ is not in the coboundary of S' since $\delta(S') \cap F = \{e\}$ and $e \neq \tilde{e}$. Thus the other endpoint of $\tilde{e}$ is in $C - S'$. On the one hand this implies $\tilde{e} \in \delta(C(e))$; on the

other it implies $\tilde{e} \notin \delta(A(e))$. But this contradicts the removal of e in the clean-up step. ■

3.3 The Algorithm for Weakly Supermodular Edge-Covering Problems

Recall that a weakly supermodular edge-covering problem is defined by the integer program

$$
\begin{aligned}
&\text{Min} \quad \sum_{e \in E} c_e x_e \\
&\text{subject to:} \\
(IP) \qquad & x(\delta(S)) \geq f(S) && S \subset V \\
& x_e \in \{0,1\} && e \in E,
\end{aligned}
$$

for some weakly supermodular function f. In this section we show that approximating a weakly supermodular edge-covering problem can be decomposed into $f_{\max}$ calls to our algorithm for approximating an uncrossable edge-covering problem. Our algorithm to approximate (IP), Approx-Weakly-Supermodular, is summarized in Figure 3-18. It takes as input a vertex set V, an edge set E, non-negative costs c_e for each edge $e \in E$, and a weakly supermodular function f. Its output is a set of edges that is feasible for (IP).

Since we proved in Chapter 2 that all proper functions are also weakly supermodular, our algorithm for approximating proper edge-covering problems, Approx-Proper, is the same as Approx-Weakly-Supermodular. Notice that the first step in the algorithm, step 2, is to compute $f_{\max}$ by using the strong oracle Max-Violated. The oracle call Max-Violated$(f, \emptyset)$ must return a set S for which $f(S) = \max_T f(T)$. By Lemma 2.0.11, for proper functions we can compute $f_{\max}$ by using n calls to the weak oracle that computes f.

The algorithm works by successively augmenting a set of edges in $f_{\max}$ *phases*. Let F_p denote the set of edges selected by the end of phase p. We start from the set $F_0 = \emptyset$. In each phase p we specify certain sets S such that we need to augment the current solution F_{p-1} by adding at least one edge to the coboundary of each S. In particular, we specify the sets that are maximally violated; that is, the sets S for which the *deficiency* $\Delta_p(S) = f(S) - |\delta_{F_{p-1}}(S)|$

is maximized. By setting $h_p(S) = 1$ for these sets S and $h_p(S) = 0$ otherwise, we can apply Approx-Uncrossable to perform the augmentation; the function h_p is uncrossable by Lemma 2.0.5 and Observation 2.0.1. Thus the maximum deficiency will be decreased in each phase of the algorithm. Since the maximum deficiency is $f_{\max}$ at the start of the algorithm, after $f_{\max}$ phases, all deficiencies will be non-positive, so that $|\delta_{F_{f_{\max}}}(S)| \geq f(S)$ for all S. In other words, $F_{f_{\max}}$ will be a feasible solution to the integer program (IP).

The idea of augmenting a graph in phases has been used previously in many graph algorithms, including the two-phase approximation algorithms of Frederickson and Ja'Ja' [39] and Klein and Ravi [74], and exact algorithms due to Naor, Gusfield, and Martel [93] for solving unweighted graph augmentation problems.

More formally, in phase p we augment the sets S for which $\Delta_p(S) \equiv f(S) - |\delta_{F_{p-1}}(S)| = f_{\max} - p + 1$. We will show by induction that $\Delta_p(S) \leq f_{\max} - p + 1$ for all $S \subset V$, and there exists some S for which the inequality is tight. Thus we can apply Approx-Uncrossable to the function

$$h_p(S) = \begin{cases} 1 & \text{if } \Delta_p(S) = f_{\max} - p + 1 \\ 0 & \text{otherwise,} \end{cases}$$

on the graph $G = (V, E_p)$, where $E_p = E - F_{p-1}$, since the function h_p is uncrossable by Lemma 2.0.5 and Observation 2.0.1. Now for the inductive proof. Certainly $\Delta_1(S) \leq f_{\max}$ for all $S \subset V$, and $\Delta_1(S) = f_{\max}$ for some S. Suppose by induction that $\Delta_p(S) \leq f_{\max} - p + 1$. Let F' be the set of edges returned by Approx-Uncrossable, and let $F_p = F_{p-1} \cup F'$. Since F' is a feasible solution to (IP) for the function h_p, $|\delta_{F'}(S)| \geq 1$ for each set S such that $\Delta_p(S) = f_{\max} - p + 1$. One can also check that the last edge selected by Approx-Uncrossable cannot be removed without losing feasibility, implying the existence of a set S such that $\Delta_p(S) = f_{\max} - p + 1$ and $|\delta_{F'}(S)| = 1$. Because $F' \subset E - F_{p-1}$, it follows that $\Delta_{p+1}(S) \equiv f(S) - |\delta_{F_p}(S)| < f_{\max} - p + 1$ for all sets S for which $\Delta_p(S) = f_{\max} - p + 1$, and thus for all sets $S \subset V$. It also follows that there exists a set S for which $\Delta_{p+1}(S) = (f_{\max} - p + 1) - 1$.

In the case of uncrossable functions generated by the Approx-Weakly-Supermodular algorithm, observe that the strong oracle for h_p can be implemented directly in terms of the strong oracle for f. That is, Max-Violated(h_p, F) = Max-Violated$(f, F \cup F_{p-1})$.

APPROX-WEAKLY-SUPERMODULAR (V, E, c, f)

1 $F_0 \leftarrow \emptyset$

2 $f_{\max} \leftarrow f(S)$ for $S \in$ MAX-VIOLATED$(f, \emptyset)$

3 **for** $p \leftarrow 1$ **to** $f_{\max}$

4 *Comment: Phase p.*

5 $\Delta_p(S) \leftarrow f(S) - |\delta_{F_{p-1}}(S)|$ for all $S \subset V$

6 $h_p(S) \leftarrow \begin{cases} 1 & \text{if } \Delta_p(S) = f_{\max} - p + 1 \\ 0 & \text{otherwise} \end{cases}$

7 $E_p \leftarrow E - F_{p-1}$

8 $F' \leftarrow$ APPROX-UNCROSSABLE(V, E_p, c, h_p)

9 $F_p \leftarrow F_{p-1} \cup F'$

10 **return** $F_{f_{\max}}$

Figure 3-18: The approximation algorithm for weakly supermodular edge-covering problems.

This follows from the definition of h_p and MAX-VIOLATED.

CHAPTER 4

Performance Guarantees

In the preceding chapter we have given high-level algorithms that find feasible solutions for edge-covering problems defined by proper functions with range $\{0,1\}$ (APPROX-PROPER-0-1), for uncrossable edge-covering problems (APPROX-UNCROSSABLE), and for weakly supermodular edge-covering problems (APPROX-WEAKLY-SUPERMODULAR). This chapter shows that these feasible solutions have a value that is close to the value of the optimal solution. To prove this, we use the dual solutions implicitly generated by the algorithms. In particular, we will show the following three theorems. For a given uncrossable function h, we let ℓ_h designate the maximum number of disjoint sets S of vertices such that $h(S) = 1$.

Theorem 4.0.1 Let $Z^*_{IP\_h}$ be the value of an optimal solution to the integer program (IP_h) given by a proper function $h : 2^V \rightarrow \{0,1\}$. Then APPROX-PROPER-0-1 produces a set of edges F' and a feasible solution y to (D_h) such that

$$\sum_{e \in F'} c_e \leq \left(2 - \frac{2}{\ell_h}\right) \sum_S h(S) y_S \leq \left(2 - \frac{2}{\ell_h}\right) Z^*_{IP}.$$

Theorem 4.0.2 Let $Z^*_{IP\_h}$ be the value of an optimal solution to the integer program (IP_h) given by a symmetric uncrossable function h. Then APPROX-UNCROSSABLE, using either REGULAR-EDGE-DELETE or EFFICIENT-EDGE-DELETE, produces a set of edges F' and a

feasible solution y to (D_h) such that

$$\sum_{e \in F'} c_e \leq \left(2 - \frac{2}{\ell_h}\right) \sum_S h(S) y_S \leq \left(2 - \frac{2}{\ell_h}\right) Z^*_{IP}.$$

Theorem 4.0.3 Let Z^*_{IP} be the value of an optimal solution to a weakly supermodular edge-covering problem given by a weakly supermodular function f, and Z^*_D be the value of an optimal solution to the dual of its linear programming relaxation. Then APPROX-WEAKLY-SUPERMODULAR produces a set of edges $F_{f_{\max}}$ such that

$$\sum_{e \in F_{f_{\max}}} c_e \leq 2\mathcal{H}(f_{\max}) Z^*_D \leq 2\mathcal{H}(f_{\max}) Z^*_{IP}.$$

If the uncrossable function h is not symmetric, then we show that the factor $2 - \frac{2}{\ell_h}$ in Theorem 4.0.2 becomes $2 - \frac{1}{\ell_h}$.

To better define the performance guarantees, we prove the following bound on ℓ_h.

Lemma 4.0.4 For any uncrossable function h, $\ell_h = |\text{MAX-VIOLATED}(h, \emptyset)|$. If h is also proper, then $\ell_h = |v \in V : h(\{v\}) = 1|$.

Proof: Let $S_1, \ldots, S_\ell$ be disjoint sets such that $h(S_i) = 1$ and ℓ is maximum. Choose the S_i's so that $\cup_i S_i$ is as small as possible. By definition, the S_i's are violated sets for the empty set of edges and are minimal since $\cup S_i$ is as small as possible. Hence the S_i's are the sets returned by MAX-VIOLATED$(h, \emptyset)$.

If h is also proper, the maximality property implies that the S_i are singletons, proving the desired result. ■

Because the performance guarantees of the algorithms are proved by comparing the value of a feasible integral solution to the value of a feasible solution to the dual of the linear programming relaxation, the theorems also imply results about the relative duality gap of the integer program (IP) for weakly supermodular and uncrossable functions. As was stated in the introduction, the relative duality gap of an integer program is the ratio of the value of an optimal solution to the integer program to the value of an optimal solution to its linear programming relaxation. Theorems 4.0.2 and 4.0.3 imply that the relative duality gap for (IP) is bounded above by 2 and $2\mathcal{H}(f_{\max})$ for all uncrossable and weakly supermodular

functions, respectively, and all non-negative edge costs. This statement holds even for functions for which we cannot implement the strong oracle Max-Violated in polynomial time. We can further observe that if $\ell_h = 2$, then Theorems 4.0.1 and 4.0.2 imply that the performance guarantee of Approx-Proper-0-1 and Approx-Uncrossable is 1. As a result, the algorithm must construct both a primal optimal solution and a dual optimal solution, thereby showing that (IP) is equivalent to its linear programming relaxation in this case.

The chapter is structured as follows. In the next section, we prove Theorems 4.0.1 and 4.0.2. We first reduce these two proofs to a combinatorial inequality about graph augmentations, and then show that the inequality holds for the Approx-Proper-0-1 algorithm and the Approx-Uncrossable algorithm with both the Regular-Edge-Delete and the Efficient-Edge-Delete deletion routines. The combinatorial inequality may be interesting in its own right; the essence of the inequality is that the sum of the degrees of vertices in any edge-minimal augmented graph is no more than twice the number of vertices that need augmenting. We will make this statement more precise in the next section. Once we have proven Theorem 4.0.2, we use it in Section 4.2 to prove the performance guarantee for Approx-Weakly-Supermodular. We conclude the chapter in Section 4.3 with an example that shows that the performance guarantee of Approx-Weakly-Supermodular is essentially tight. The example also indicates that a substantially different approach will be needed to achieve an algorithm with a better performance guarantee.

4.1 Proofs of Performance Guarantee for Approx-Uncrossable

In both algorithms Approx-Proper-0-1 and Approx-Uncrossable, the primal complementary slackness conditions are maintained; thus we know for any edge $e \in F'$, no matter how edges are deleted, that $c_e = \sum_{S:e\in\delta(S)} y_S$. Thus the cost of the solution F' is

$$\sum_{e\in F'} c_e = \sum_{e\in F'} \sum_{S:e\in\delta(S)} y_S = \sum_{S\subset V} y_S |\delta_{F'}(S)|.$$

Thus in order to prove Theorems 4.0.2 and 4.0.1, all we need to do is show that

$$\sum_{S \subset V} y_S |\delta_{F'}(S)| \leq \left(2 - \frac{2}{\ell_h}\right) \sum_{S \subset V} h(S) y_S.$$

Because $\sum_S h(S) y_S$ is the dual objective function, and y is a feasible dual solution, this inequality can be proved by induction on the while loop of both algorithms. Certainly the inequality holds before the first iteration of the loop, since initially all $y_S = 0$. Consider the collection $\mathcal{C}$ of active sets at the beginning of some iteration of the loop. The left-hand side of the inequality will increase by $\sum_{C \in \mathcal{C}} \epsilon \cdot |\delta_{F'}(C)|$ in this iteration while the increase of the right-hand side will be $(2 - \frac{2}{\ell_h})\epsilon \cdot |\mathcal{C}|$.

The inductive proof will follow from a proof that at any iteration,

$$\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq 2|\mathcal{C}| - 2. \tag{4.1}$$

Because $|\mathcal{C}| \leq \ell_h$ for all possible collections $\mathcal{C}$ of active sets, it follows that the inequality implies that $\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq (2 - \frac{2}{\ell_h})|\mathcal{C}|$. Another way of looking at inequality (4.1) is that we show that the sum of the degrees of the active sets with respect to F' is no more than $2|\mathcal{C}| - 2$, as if F' were a forest on the active sets. For this reason we call inequality (4.1) the *total-degree inequality*. The proof of the inequality can be viewed as a charging scheme in which we show that there are many active sets of degree one that compensate for high degree active sets. It can also be viewed as a statement about the edges needed to augment a set of edges to a feasible solution for an uncrossable function h: if there are $|\mathcal{C}|$ minimally violated sets, then the total number of edges that will be added to the coboundary of these sets is at most $2|\mathcal{C}| - 2$, no matter how future active sets are created.

A third way to view the total-degree inequality (4.1) is as a relaxation of the dual complementary slackness conditions. Recall from the introduction that typical primal-dual approximation algorithms maintain primal complementary slackness conditions, as we have done, but relax the dual complementary slackness conditions to

$$y_S > 0 \Rightarrow h(S) \leq \sum_{e \in \delta(S)} x_e \leq \alpha h(S).$$

Inequality (4.1) shows that for our algorithm the standard relaxed dual complementary slackness conditions hold when averaged over the active sets of an iteration. That is, when we increase the variables y_C for $C \in \mathcal{C}$ in some iteration, inequality (4.1) implies that

$$\sum_{C \in \mathcal{C}} \left(h(C) \leq \sum_{e \in \delta(C)} x_e \leq 2h(C) - \frac{2}{|\mathcal{C}|} \right),$$

or

$$|\mathcal{C}| \leq \sum_{e \in \delta(C): C \in \mathcal{C}} x_e \leq 2|\mathcal{C}| - 2.$$

We now need to show that the total-degree inequality (4.1) holds for the algorithm APPROX-PROPER-0-1 and the algorithm APPROX-UNCROSSABLE with the edge deletion stages REGULAR-EDGE-DELETE and EFFICIENT-EDGE-DELETE. For the remainder of this section, we will concentrate on the active sets $C \in \mathcal{C}$ of some particular iteration of the algorithm, which we call the current iteration.

4.1.1 The Total-Degree Inequality for APPROX-PROPER-0-1

Theorem 4.1.1 In any iteration of APPROX-PROPER-0-1, inequality (4.1) holds.

Proof: Let $\mathcal{I}'$ be the collection of connected components N in $\mathcal{I}$ such that $\delta_{F'}(N) \neq \emptyset$. To begin, construct a graph H by considering the sets in $\mathcal{C}$ and $\mathcal{I}'$ of the current iteration as vertices of H, and the edges $e \in \delta_{F'}(S)$ for all $S \in \mathcal{C} \cup \mathcal{I}'$ as the edges of H. Notice that H is a forest.

We claim that no leaf in H corresponds to a set from $\mathcal{I}'$. To see this, suppose otherwise, and let v be a leaf of H, I_v its associated set from $\mathcal{I}'$, e the edge incident to v, and N the component of F which contains I_v. Let P and $N - P$ be the two components formed by removing edge e from the edges of component N. Without loss of generality, say that $I_v \subseteq P$. The set $P - I_v$ is partitioned by some of the components of the current iteration; call these $P_1, \ldots, P_k$ (see Figure 4-1). Since I_v is a leaf, no edge in F' connects I_v to any P_i. Thus by the construction of F', $h(\cup P_i) = 0$. Since $h(I_v) = 0$ also, it follows that $h(P) = 0$. We know $h(N) = 0$, so by Observation 3.1.1 $h(N - P) = 0$ as well, and thus by the construction of F', $e \notin F'$, which is a contradiction.

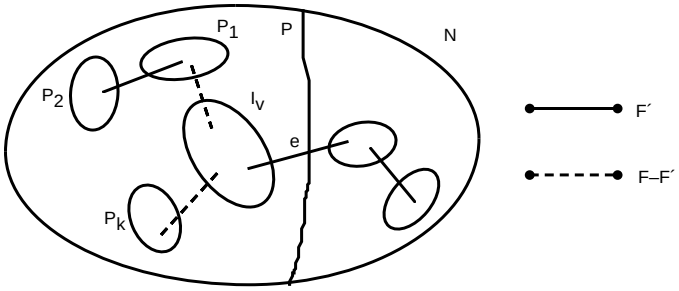

Figure 4-1: Illustration of claim that all leaves of H are active.

Let $d_N = |\delta_{F'}(N)|$ for a component $N \in \mathcal{C} \cup \mathcal{I}'$. Thus d_N gives the degree of the vertex v in the graph H that corresponds to the component N. Then

$$\begin{aligned} \sum_{N \in \mathcal{C}} d_N &= \sum_{N \in \mathcal{C} \cup \mathcal{I}'} d_N - \sum_{N \in \mathcal{I}'} d_N \\ &\leq 2(|\mathcal{C}| + |\mathcal{I}'| - 1) - 2|\mathcal{I}'| \\ &= 2|\mathcal{C}| - 2. \end{aligned}$$

This inequality holds since H is a forest with at most $|\mathcal{C}| + |\mathcal{I}'| - 1$ edges, and since each vertex corresponding to an inactive component in $\mathcal{I}'$ has degree at least 2. ■

4.1.2 The Total-Degree Inequality for Approx-Uncrossable with Regular-Edge-Delete

The proof of the total-degree inequality for the Approx-Uncrossable algorithm becomes somewhat more complicated than the proof for Approx-Proper-0-1. In the previous proof, we could take the structure of the current forest F in terms of the connected components in $\mathcal{C}$ and $\mathcal{I}$, and abstract a forest from this structure. We then showed that all of the leaves of the forest corresponded to active sets, thus proving the total-degree inequality. The proof here follows the same general strategy, but abstracting the forest becomes more difficult. Here we show that Regular-Edge-Delete induces a tree-like structure on specific violated sets of the current iteration.

We will assume that the uncrossable function h is also symmetric; that is, $h(S) = h(V - S)$ for all $S \subset V$. This is true of all uncrossable functions generated by Approx-

Proper. At the end of the proof we will note that a non-symmetric h implies a performance guarantee of $2 - \frac{1}{\ell_h}$ instead of $2 - \frac{2}{\ell_h}$.

Define $Y = \bigcup_{C \in \mathcal{C}} \delta_{F'}(C)$; that is, Y is the set of edges in F' that are in the coboundary of the currently active sets. Notice that these edges must have been added during or after the current iteration.

Lemma 4.1.2 For each edge $e \in Y$ there exists a witness set $S_e \subset V$ such that

1. $h(S_e) = 1$,
2. $\delta_{F'}(S_e) = \{e\}$,
3. For each $C \in \mathcal{C}$ either $C \subseteq S_e$ or $C \cap S_e = \emptyset$.

Proof: Any edge $e \in Y$ is also in F', and thus during Regular-Edge-Delete the removal of e causes h to be violated for some S. In other words, there can exist no other $e' \in F'$ that is also in $\delta(S)$. This set S will be the witness set S_e for e, and clearly satisfies (1) and (2). Now let F_b be all the edges added before the current iteration. To show (3), notice that when considering edge e in Regular-Edge-Delete, no edge in F_b had yet been removed. Hence S_e is violated even if all the edges of F_b are included; that is, S_e is violated in the current iteration. Thus (3) follows by Lemma 3.2.1 and the minimality of the active sets $C \in \mathcal{C}$. ■

Consider a collection of sets S_e satisfying the conditions of the preceding lemma, taken over all the edges e in Y. Call such a collection a *witness family*. Recall that any collection of sets is called *laminar* if there is no crossing pair of sets A, B in the collection.

Lemma 4.1.3 There exists a laminar witness family.

Proof: By the previous lemma, there exists a witness family. From this collection of sets we can form a laminar collection of sets as follows. We maintain that $h(S) = 1$ for all sets S in the collection. If the collection is not laminar, there exists a crossing pair of sets A, B. Because $h(A) = h(B) = 1$, either $h(A - B) = h(B - A) = 1$ or $h(A \cup B) = h(A \cap B) = 1$. If the latter is true, we uncross A and B by replacing them in the collection with $A \cup B$ and

$A \cap B$ (the other case is analogous). This procedure terminates with a laminar collection since whenever two sets are uncrossed, the total number of pairs of sets that cross is reduced. To see this, note that if a set X in the collection crosses both A and B, then replacing A and B with $A - B$ and $B - A$, or $A \cap B$ and $A \cup B$ cannot increase the total number of sets that X crosses. If X crosses only A and is not contained in B, then it cannot cross $B - A$ or $A \cap B$. If X crosses only A and is contained in B, then it cannot cross $A \cup B$ or $A - B$, and so again uncrossing A and B cannot increase the total number of sets that X crosses. Thus uncrossing A and B does not increase the total number of pairs of sets that cross, and in fact decreases the total by at least one, since A no longer crosses B.

We claim that the resulting laminar collection forms a witness family. This claim can be proven by induction on the uncrossing process. Property (3) obviously continues to hold when any two sets are uncrossed. Suppose we have two witness sets S_1 and S_2 corresponding to edges e_1 and e_2 such that S_1 and S_2 cross. Since h is uncrossable, either $h(S_1 \cup S_2) = h(S_1 \cap S_2) = 1$ or $h(S_1 - S_2) = h(S_2 - S_1) = 1$. Without loss of generality, suppose $h(S_1 \cup S_2) = h(S_1 \cap S_2) = 1$. By submodularity, $2 = |\delta_{F'}(S_1)| + |\delta_{F'}(S_2)| \geq |\delta_{F'}(S_1 \cap S_2)| + |\delta_{F'}(S_1 \cup S_2)|$. Because $h(S_1 \cup S_2) = 1$, it is not the case that $S_1 \cup S_2 = V$. Thus by the feasibility of F', $|\delta_{F'}(S_1 \cap S_2)| \geq 1$ and $|\delta_{F'}(S_1 \cup S_2)| \geq 1$. Hence it must be the case that if $h(S_1 \cup S_2) = h(S_1 \cap S_2) = 1$, then $|\delta_{F'}(S_1 \cap S_2)| = |\delta_{F'}(S_1 \cup S_2)| = 1$. Therefore either $S_1 \cap S_2$ is a witness set for e_1 and $S_1 \cup S_2$ is a witness set for e_2, or vice versa.■

Let $\mathcal{S}$ be a laminar witness family. Add the set V to the family. The family can be viewed as defining a tree H with a vertex v_S for each $S \in \mathcal{S}$ and edge (v_S, v_T) if T is the smallest element of $\mathcal{S}$ properly containing S. Each active set $C \in \mathcal{C}$ is associated with the smallest set $S \in \mathcal{S}$ that contains it. We will call a vertex v_S active if S is associated with some active set C. Let $\mathcal{L}(v_S)$ be the collection of sets $C \in \mathcal{C}$ associated with an active vertex v_S.

Lemma 4.1.4 The tree H has no inactive leaf.

Proof: Only V and the minimal (under inclusion) witness sets can correspond to leaves. Any minimal witness set is a violated set in the current iteration, and thus must contain an active set which corresponds to it. Let S be any maximal witness set. Given that h is

symmetric, both S and $V - S$ are violated sets in the current iteration, and thus contain active sets C. Therefore, v_V cannot be simultaneously a leaf and inactive. ■

Lemma 4.1.5 For any active vertex v_S in H, the degree of v_S is at least $\sum_{C \in \mathcal{L}(v_S)} |\delta_{F'}(C)|$.

Proof: Note that the one-to-one mapping between the edges of Y and the witness sets implies a one-to-one mapping between the edges of Y and the edges of H: each witness set S defines a unique edge (v_S, v_T) of H, where T contains S. Consider any edge $e \in \delta_{F'}(C)$ for some $C \in \mathcal{C}$. Let (v_{S_e}, v_T) be the edge defined by the witness set S_e. The active set C must be associated with either v_{S_e} or v_T. By summing over all edges $e \in \delta_{F'}(C)$ for all active sets C corresponding to an active vertex v_S of H (that is, all $C \in \mathcal{L}(v_S)$), we obtain the lemma. ■

Theorem 4.1.6 In any iteration of APPROX-UNCROSSABLE with REGULAR-EDGE-DELETE, inequality (4.1) holds.

Proof: Let H_a denote the set of active vertices in H and let d_v denote the degree of a vertex v. Then, as is shown in the proof of Theorem 4.1.1,

$$\sum_{v \in H_a} d_v = \sum_{v \in H} d_v - \sum_{v \in H - H_a} d_v \leq 2(|H| - 1) - 2(|H| - |H_a|) = 2|H_a| - 2.$$

This inequality holds since H is a tree with $|H| - 1$ edges, and since each vertex in $H - H_a$ has degree at least 2. The lemma above implies that $\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq \sum_{v \in H_a} d_v$, while clearly $|H_a| \leq |\mathcal{C}|$. Thus

$$\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq 2|\mathcal{C}| - 2.$$

■

If h is not a symmetric function, then we can obtain a slightly weaker performance guarantee. In this case, the tree H can have at most one inactive leaf, v_V. Following the same logic as above, we obtain that $\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq 2|\mathcal{C}| - 1$, which implies a performance guarantee of $2 - \frac{1}{\ell_h}$ for APPROX-UNCROSSABLE. Thus if $\ell_h = 1$ for a non-symmetric function h, the performance guarantee for APPROX-UNCROSSABLE is 1. The algorithm will

construct primal and dual optimal solutions, implying that (IP) is equivalent to its linear programming relaxation in this case.

As far as performance guarantees are concerned, however, we do not need to worry about non-symmetric functions. Notice that by applying Approx-Uncrossable to the symmetrization h' of h as described at the end of Section 3.2.1, one obtains a performance guarantee of $2 - \frac{2}{\ell_{h'}}$. Since, by Lemmas 3.2.3 and 4.0.4, $\ell_{h'} \leq \ell_h + 1$, we see that $2 - \frac{2}{\ell_{h'}} \leq 2 - \frac{2}{\ell_h + 1} \leq 2 - \frac{1}{\ell_h}$. As a result, using the symmetrized function h' never hurts in terms of the performance guarantee.

4.1.3 The Total-Degree Inequality for Approx-Uncrossable with Efficient-Edge-Delete

Our strategy for proving the total-degree inequality so far has been to construct a forest on the active sets and the edges in $\bigcup_{C \in \mathcal{C}} \delta_{F'}(C)$ and then show that the leaves of the forest correspond to active sets. With Efficient-Edge-Delete, the proof strategy becomes quite different. Here we show that the special edges are exactly those edges whose removal can ensure the performance guarantee. Suppose for a moment, as we did in Section 3.2.2, that no new active sets are created in subsequent iterations. Let e be the edge chosen in the current iteration, and suppose that e is a special edge. Recall that we informally defined a 1-edge e as special if the edges added to F after e form a tree on the sets active just before e is added. If e is added in the current iteration and is special, then $\sum_{C \in \mathcal{C}} |\delta_F(C)| = 2|\mathcal{C}| - 1$, but removing e causes the total-degree inequality to be satisfied (see Figure 4-2). This is the central intuition of the proof; we employ it recursively in order to handle the more general case.

We preface our proof of the total-degree inequality for Approx-Uncrossable with Efficient-Edge-Delete by reviewing some definitions. Recall that in Section 3.2.2 we define $\cup\mathcal{C}$ to be the collection of active sets over all iterations, plus the set V. We define a tree $\mathcal{T}$ with one vertex v_C for each $C \in \cup\mathcal{C}$. The node v_C is a parent of v_D in the tree if C is the smallest set in $\cup\mathcal{C}$ that properly contains D. The collection of sets corresponding to the children of v_C is denoted $\mathcal{D}(C)$. The set $C(e)$ is the smallest set $C \in \cup\mathcal{C}$ that contains both endpoints of e; F_C is the subset of edges e of F such that $C(e) = C$. We think of $\mathcal{D}(C)$

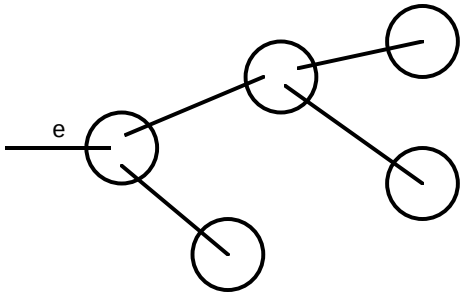

Figure 4-2: A bad case for the performance guarantee. Circles represent sets active just before edge e is added. The edge e is special.

and F_C as associated equivalence classes on the active sets and the edges of F. Finally, the notation $\mathcal{A}(e)$ denotes all sets in $\mathcal{D}(C(e))$ that are active just before e is selected.

Theorem 4.1.7 In any iteration of APPROX-UNCROSSABLE with EFFICIENT-EDGE-DELETE, inequality (4.1) holds.

Proof: Define the subtree $\mathcal{T}'$ of $\mathcal{T}$ to contain all the vertices corresponding to sets that will be active in or after the current iteration. Thus the leaves of the tree correspond exactly to the sets in $\mathcal{C}$ in the current iteration. Define $\mathcal{E}(C)$ to be the sets corresponding to the children of an internal node v_C of $\mathcal{T}'$; that is, $\mathcal{E}(C)$ contains the sets of $\mathcal{D}(C)$ that are active in or after the current iteration. Let Y be the set of edges selected in or after the current iteration. Let $Y' = Y \cap F'$.

We will prove inequality (4.1) by showing for each internal node v_C of the tree $\mathcal{T}'$ that

$$\sum_{S \in \mathcal{E}(C)} |\delta_{Y'}(S)| - |\delta_{Y'}(C)| \leq 2|\mathcal{E}(C)| - 2. \tag{4.2}$$

In effect, we prove a version of the total-degree inequality for each "equivalence class" C, subtracting off the contribution made to the total degree made by edges with only one endpoint in C (see Figure 4-3). Given that $|\delta_{Y'}(V)| = 0$, by summing this inequality over all internal nodes v_C of the tree, we will obtain

$$\sum_{C \in \mathcal{C}} |\delta_{Y'}(C)| \leq 2|\mathcal{C}| - 2. \tag{4.3}$$

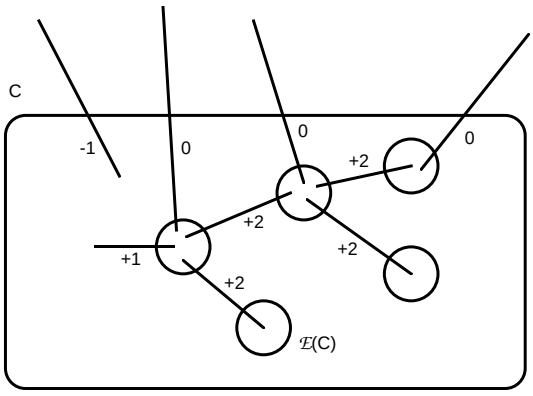

Figure 4-3: An illustration of inequality (4.2). Circles represent sets in $\mathcal{E}(C)$. Numbers are the coefficient of the edge in the left-hand side of inequality (4.2).

To see this, observe that on the left-hand side, the negative term $-|\delta_{Y'}(C)|$ for an internal vertex v_C is cancelled by the positive term in the inequality on the parent of v_C, leaving only the positive terms corresponding to the leaves. Similarly, on the right-hand side, the contribution of -2 for each internal vertex v_C is cancelled by a contribution of 2 by the parent of v_C, leaving a positive contribution of 2 for each leaf and a contribution of -2 by the vertex v_V. Inequality (4.3) implies the total-degree inequality since $\delta_{Y'}(C) = \delta_{F'}(C)$ for any active set $C \in \mathcal{C}$; that is, no edge of F' in the coboundary of an active C could have been added before the current iteration.

Now we must prove inequality (4.2) on each internal node v_C of the tree. Let $k = |\mathcal{E}(C)|$. Let $I = F_C \cap Y$, let $\Phi = \sum_{S \in \mathcal{E}(C)} |\delta_I(S)|$, and let J be the subset of edges in Y' with one endpoint in $V - C$ and one endpoint in $C - \bigcup_{S \in \mathcal{E}(C)} S$. Then the inequality on the internal node v_C is implied by

$$\Phi - |J| \leq 2k - 2.$$

The idea behind proving this inequality is that we will always be able to show that $\Phi \leq 2k - 1$, and we will be able to show that $\Phi \leq 2k - 2$ when I contains a 2-edge or has more than one "connected component" on the sets $\mathcal{E}(C)$. Thus the bad case is exactly when there is a special edge e, no other edges in I have been removed, and $J = \emptyset$, which is precisely when Efficient-Edge-Delete will remove edge e.

In any iteration in which an edge of I is selected, we must make an active set $S \in \mathcal{E}(C)$ inactive. Thus $|I| \leq k$. Each edge in I contributes at most 2 to Φ, so that we have $\Phi \leq 2k$. If an edge in I is a 2-edge, then it must make 2 active sets in $\mathcal{E}(C)$ inactive while contributing at most 2 to Φ, proving that $\Phi \leq 2k-2$, which implies the inequality. Note that if $C = V$ and the function h is symmetric, then the final edge in I must be a 2-edge, between the final two active sets.

So assume I consists of 1-edges. Let e be the first edge of I that was selected; i.e., other edges in I were selected in iterations after e was selected. Notice that e can only contribute 1 to Φ, since it is a 1-edge. Thus $\Phi \leq 2k-1$. Since e is the first edge of I selected, it must be the case that $\mathcal{A}(e) = \mathcal{E}(C(e))$. If e is not special, then I contains an edge e' with an endpoint not in any $S \in \mathcal{E}(C)$. The edge e' contributes 1 to Φ, giving $\Phi \leq 2k-2$.

Now suppose e is special. If some edge of I is deleted in the final edge set F', then $\Phi \leq 2k-2$, since the edge must have contributed 1 to Φ. If e is special, was not deleted, and $C \neq V$, then by the properties of EFFICIENT-EDGE-DELETE it must be the case that $J \neq \emptyset$; hence the inequality must hold. ■

If h is not symmetric, then for $C = V$ we can only prove that $\Phi \leq 2k-1$. Summing over all internal nodes in the tree $\mathcal{T}$ leads to the inequality $\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq 2|\mathcal{C}| - 1$.

4.2 Performance Guarantee for APPROX-WEAKLY-SUPERMODULAR

Given the proofs of the performance guarantee for APPROX-UNCROSSABLE, we can now turn to providing a proof for the APPROX-WEAKLY-SUPERMODULAR algorithm. We first show that the dual solution y constructed in phase p by APPROX-UNCROSSABLE can be mapped to a feasible solution to the dual of the linear programming relaxation of (IP). This dual is:

$$\text{Max} \quad \sum_{S \subset V} f(S) y_S - \sum_{e \in E} z_e$$

subject to:

$$(D) \qquad \sum_{S : e \in \delta(S)} y_S \leq c_e + z_e \qquad e \in E, \tag{4.1}$$

$$y_S \geq 0 \qquad S \subset V,$$

$$z_e \geq 0 \qquad\qquad e \in E.$$

Given the dual variables y constructed by the algorithm in phase p, define $z_e = \sum_{S:e\in\delta(S)} y_S$ for all $e \in F_{p-1}$, and $z_e = 0$ otherwise. Notice that by this definition,

$$\sum_{e\in E} z_e = \sum_{e\in F_{p-1}} \sum_{S:e\in\delta(S)} y_S = \sum_S |\delta_{F_{p-1}}(S)| y_S.$$

Lemma 4.2.1 The vector (y, z) is a feasible solution for (D).

Proof: By the construction of y by Approx-Uncrossable, for $e \in E_p \equiv E - F_{p-1}$, we know that $\sum_{S:e\in\delta(S)} y_S \leq c_e$. Thus the constraints (4.1) hold for $e \notin F_{p-1}$. For $e \in F_{p-1}$, the definition of z_e ensures that the constraint (4.1) holds. ■

We now provide a proof of Theorem 4.0.3.

Proof: From Lemma 4.2.1, we know that in phase p

$$\begin{aligned} Z_D^* &\geq \sum_S f(S) y_S - \sum_{e\in E} z_e \\ &= \sum_S (f(S) - |\delta_{F_{p-1}}(S)|) y_S \\ &= (f_{\max} - p + 1) \sum_S y_S, \end{aligned}$$

where we have used the fact that in phase p the dual variable $y_S > 0$ only if the deficiency of S $(f(S) - |\delta_{F_{p-1}}(S)|)$ is $f_{\max} - p + 1$. Using the proof of the performance guarantee for Approx-Uncrossable and summing over all phases, we obtain that

$$\sum_{e\in F_{f_{\max}}} c_e \leq 2 \sum_{p=1}^{f_{\max}} \frac{1}{f_{\max} - p + 1} Z_D^* = 2\mathcal{H}(f_{\max})\, Z_D^*,$$

proving the desired result. ■

The performance guarantee of Approx-Weakly-Supermodular is actually somewhat tighter than is given in the theorem. If the weakly supermodular function f is symmetric, then the uncrossable functions h_p defined in each phase will also be symmetric, and the performance guarantee improves to $\sum_{p=1}^{f_{\max}} (2 - \frac{2}{\ell_{h_p}}) \frac{1}{f_{\max}-p+1}$. If f is not symmetric, then the performance guarantee is $\sum_{p=1}^{f_{\max}} (2 - \frac{1}{\ell_{h_p}}) \frac{1}{f_{\max}-p+1}$.

Theorem 4.0.3 also applies to the *augmentation* version of these problems: Given an initial set of edges F_0, find a minimum-cost set of edges to add to F_0 such that the resulting graph satisfies the weakly supermodular function f. We merely apply APPROX-WEAKLY-SUPERMODULAR to the function $f'(S) = f(S) - |\delta_{F_0}(S)|$, which is weakly supermodular by Lemma 2.0.5. This results in a performance guarantee of $2\mathcal{H}(f'_{\max})$. We will show in Chapter 5 that we can implement the strong oracle efficiently for such functions f' when f is proper.

4.3 A Tight Example

The performance guarantee of $O(\log f_{\max})$ is essentially tight. To show this, we construct a family of instances for which the cost of the solution returned by APPROX-WEAKLY-SUPERMODULAR differs from the optimal cost by a factor exceeding $\frac{1}{2}\log_2(f_{\max})$. The edge-covering problem in the example will be a minimum-cost flow problem, i.e., the problem of finding k edge-disjoint paths between two nodes s and t of minimum cost. The problem can be modelled by the integer program (IP) with the weakly supermodular function

$$f(S) = \begin{cases} k & \text{if } s \in S, t \notin S \\ 0 & \text{otherwise.} \end{cases}$$

We now show that f is weakly supermodular: pick any two sets A and B. Obviously weak supermodularity holds if $f(A) = f(B) = 0$. If $f(A) = k$ and $f(B) = 0$, then either $A - B$ or $A \cap B$ must contain s but not t, and weak supermodularity holds. If $f(A) = f(B) = k$, then $f(A \cup B) = f(A \cap B) = k$, and the property holds. We could also model this problem with the proper function

$$f'(S) = \begin{cases} k & \text{if } |S \cap \{s,t\}| = 1 \\ 0 & \text{otherwise,} \end{cases}$$

but it will be easier for us to work with the previous function.

We claim that on this function f, the algorithm APPROX-WEAKLY-SUPERMODULAR

can be simulated by k shortest path computations. First notice that maximum deficiency in phase p is $f(S) - |\delta_{F_{p-1}}(S)| = k - (p-1)$; this implies that $|\delta_{F_{p-1}}(S)| \geq p-1$ for all sets S with $s \in S, t \notin S$. Therefore there are at least $p-1$ edge-disjoint paths between s and t. In phase p, we consider the function $h_p(S) = 1$ iff $s \in S, t \notin S$, and $|\delta_{F_{p-1}}(S)| = p-1$. Notice that $\ell_{h_p} = 1$, so by the proof of performance guarantee for Approx-Uncrossable, the algorithm will find an optimal solution for this uncrossable edge-covering problem. Thus the overall performance guarantee of Approx-Weakly-Supermodular for this problem will be $\mathcal{H}(k)$.

We now show that the optimal solution for each uncrossable edge-covering problem can be found by solving a shortest s-t path problem. Choose any $p-1$ edge-disjoint paths from s to t in F_{p-1}, and direct all the edges in the paths from t to s. Bidirect all other edges of E. Assign all directed edges in F_{p-1} a cost of 0, and all other edges their regular cost. A path from s to t corresponds to a solution of the uncrossable edge-covering problem of the same cost: notice that for any set S such that $h_p(S) = 1$, all zero-cost directed edges from F_{p-1} must be directed from vertices outside S to vertices in S. Thus each set S such that $h_p(S) = 1$ is covered by an edge from $E - F_{p-1}$. Likewise, any solution to the uncrossable edge-covering problem can be made to correspond to an s-t path of no greater cost: any set $S, s \in S, t \notin S$, for which $h_p(S) = 0$ must have some zero-cost edge of F_{p-1} directed out of S. This fact implies that t is reachable from s by directed edges corresponding to the solution of the uncrossable edge-covering problem plus zero-cost edges. Hence the shortest s-t path corresponds to the minimum-cost solution of the uncrossable edge-covering problem.

We can now give our tight example for Approx-Weakly-Supermodular on this function f. The tight example is built recursively (see Figure 4-4). G_0 is a graph on two nodes s and t with a single edge of cost c between them. To construct graph G_i, take two copies of G_{i-1}. Identify the nodes s and t in the two graphs. In the first copy, subdivide the unique edge of cost c entering t, and assign cost 0 to the part entering t, and c to the other part. In the second copy, subdivide the unique edge of cost c leaving s, and assign cost 0 to the part leaving s and cost c to the remaining part. Then connect the two new nodes by an edge of cost $2^{i-1}(c-\epsilon)$. The problem on graph G_i is to find $2^i - 1$ disjoint paths between s and t; we set $k = 2^i - 1$ and use the function f as above.

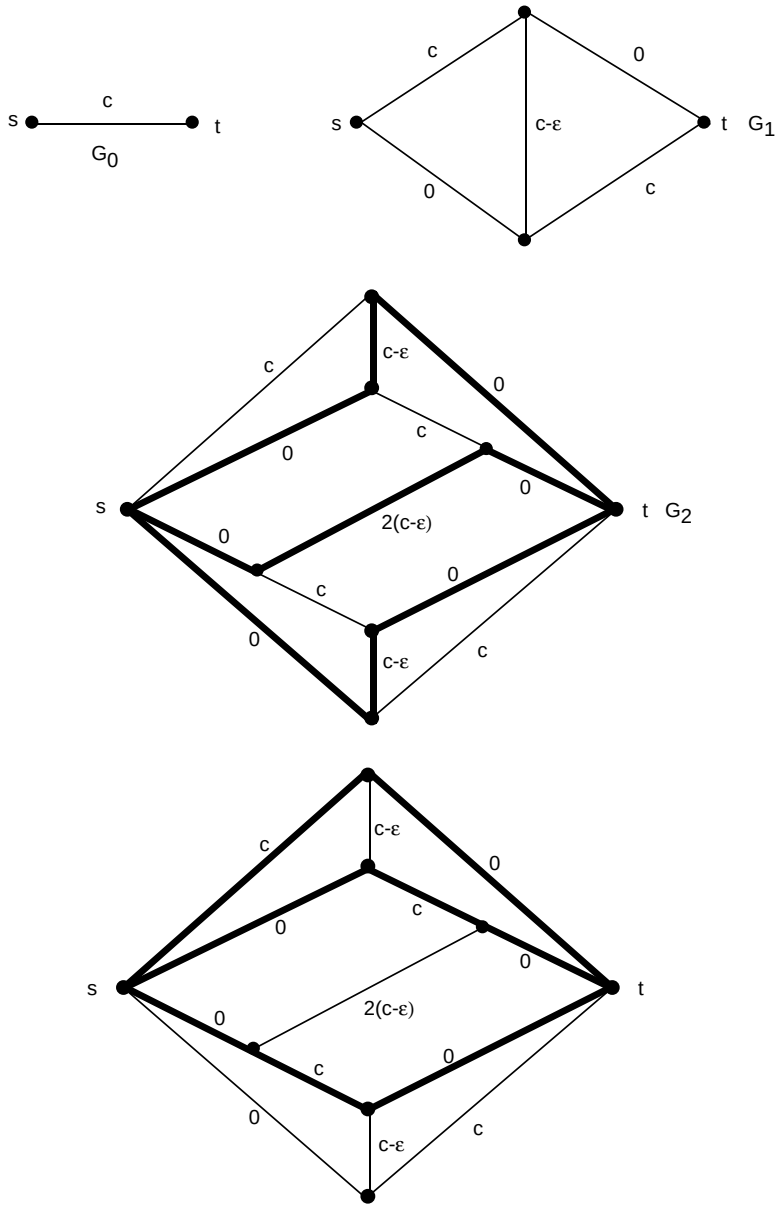

Figure 4-4: The tight example for APPROX-WEAKLY-SUPERMODULAR. Bold edges in the first instance of G_2 are the edges selected by the algorithm. Bold edges in the second instance denote the optimal solution.

Given the discussion above, the algorithm APPROX-WEAKLY-SUPERMODULAR will select 2^{i-1} node-disjoint paths of cost $(c-\epsilon)$, 2^{i-2} node-disjoint paths of cost $2(c-\epsilon)$, etc., for a total cost of $i2^{i-1}(c-\epsilon)$. Notice that there exists a solution of cost $(2^i-1)c$: take 2^i-1 copies of the graph G_0, where each has been subdivided into a path of length c. Hence, the cost of the solution returned by APPROX-WEAKLY-SUPERMODULAR differs from the optimal cost by a factor of $\frac{i2^{i-1}}{2^i-1}\frac{c-\epsilon}{c} \geq \frac{1}{2}\log_2(f_{\max})$.

The example suggests that any algorithm that greedily augments its solutions in phases will also have a worst-case performance guarantee of $O(\log f_{\max})$. A better algorithm for weakly supermodular edge-covering problems will probably require a more global approach than the multi-phase augmentation that we use here.

CHAPTER 5

Implementing the Algorithms

In this chapter we turn from the high-level outlines of the algorithms to the details of how these algorithms can be implemented efficiently. In particular, Chapter 3 did not discuss how to do the following:

1. Implement the strong oracle MAX-VIOLATED for proper functions f.
2. Select the edge e minimizing ϵ in steps 10 and 8 of APPROX-UNCROSSABLE and APPROX-PROPER-0-1 respectively.
3. Implement the edge deletion stages EFFICIENT-EDGE-DELETE and PROPER-0-1-EDGE-DELETE.

Once we have completed the discussion of these steps, we will be able to show the following theorems.

Theorem 5.0.1 The algorithm APPROX-PROPER, using EFFICIENT-EDGE-DELETE, can be implemented in time $O(n^2 m' f_{\max} + n^2 \omega_f f_{\max})$, where $m' = \min(n f_{\max}, m)$ and ω_f is the time taken by the weak oracle to compute the proper function f.

Theorem 5.0.2 The algorithm APPROX-PROPER-0-1 can be implemented in time $O(n(n + \sqrt{m \log\log n} + \omega_h))$, where ω_h is the time taken by a weak oracle to compute the proper function h.

In addition, we will be able to show that the strong oracle can be made to run faster for a particular problem called the survivable network design problem.

Theorem 5.0.3 The algorithm APPROX-PROPER, using EFFICIENT-EDGE-DELETE, can be implemented for the survivable network design problem in $O(nf_{\max}(nf_{\max}+\sqrt{m\log\log n}))$ time.

The chapter is structured as follows. In Section 5.1, we will show how to implement the strong oracle MAX-VIOLATED for proper functions f. We will then give a faster implementation for the sequence of oracle calls generated by APPROX-PROPER with EFFICIENT-EDGE-DELETE and a still faster implementation for APPROX-PROPER for the survivable network design problem. Section 5.2 will show how to select the edge minimizing ϵ. The algorithms for EFFICIENT-EDGE-DELETE and PROPER-0-1-EDGE-DELETE are given in Sections 5.3 and 5.4, and the chapter concludes in Section 5.5 with a derivation of the theorems above.

5.1 Implementing the Strong Oracle

5.1.1 A General Implementation

We turn now to the problem of implementing the strong oracle MAX-VIOLATED for proper functions f. Recall that for a function f and edge set F, MAX-VIOLATED(f,F) returns a collection of the minimal (with respect to inclusion) sets S such that $f(S)-|\delta_F(S)| = \max_T(f(T)-|\delta_F(T)|) > 0$, if any such sets exists. We have called these sets the maximally violated sets. Notice that if we can implement the strong oracle for a function f, then we can also implement it for a function $f'(S) = f(S) - |\delta_Y(S)|$ for any edge set Y that is disjoint from F: a call to MAX-VIOLATED(f',F) will be equivalent to a call to MAX-VIOLATED$(f, F\cup Y)$. Thus we will be able to implement the strong oracle for the weakly supermodular function f' if f is proper.

We will also be able to implement a variation of the strong oracle MAX-VIOLATED(f,x) for $x\in\mathbb{Q}_+^{|E|}$. This variation is a version of a problem called the *separation problem*, which merely requires finding any violated set T such that $f(T)-x(\delta(T))>0$. The ability to solve the separation problem in polynomial time is known to imply that the ellipsoid algorithm for linear programming can be used to optimize over the LP relaxation of (IP) in polynomial

time [57]. Solving the separation problem also allows us to implement the m-approximation algorithm of Hall and Hochbaum [60] for the integer program (IP), as was noted in the Introduction.

To implement MAX-VIOLATED(f, x), we use the Gomory-Hu cut tree [54] on a graph with edge capacities x_e. The Gomory-Hu cut tree is a very useful structure from network flow theory that gives information about minimum s-t cuts in a capacitated graph. Given the graph $G = (V, E)$ with edge capacities x_e, the Gomory-Hu procedure returns a tree H with values w_e on its edges such that the value of the minimum cut between any two vertices s and t is given by the smallest value w on the unique path in H between s and t. Let S_e and $V - S_e$ be the partition of the vertex set induced when e is removed. The tree H also has the property that $w_e = x(\delta(S_e))$.

The procedure for constructing the tree works as follows. It starts from one supervertex containing all vertices of the graph. At any stage of the construction, there is a partial tree whose (super)vertices form a partition of the vertex set. The procedure selects two vertices u and v within a supervertex A, shrinks the vertices in each connected component resulting from the removal of A from the cut tree, and computes the maximum flow and minimum cut between u and v in the resulting shrunk graph. The supervertex A is split into two supervertices linked by an edge, in such a way that the removal of this edge induces the computed mincut. The new edge of the cut tree gets labeled with the value of the maximum flow between u and v. The procedure terminates when no supervertices remain. The overall procedure requires $n - 1$ maximum flow computations.

We now show that the Gomory-Hu tree has structural properties that will allow us to implement MAX-VIOLATED(f, x). To simplify the presentation, we assume that for any edge e, some specified vertex, say vertex 1, does not belong to the set S_e. Also, given a subset S of vertices, let $\gamma(S)$ denote the edges of the cut-tree H with exactly one endpoint in S.

Lemma 5.1.1 Let $S \subset V$. Then

$$x(\delta(S)) \geq \max_{e \in \gamma(S)} w_e.$$

Proof: We show that for any $e = (i, j) \in \gamma(S)$ we have $x(\delta(S)) \geq w_e = x(\delta(S_e))$. This is immediate since, by definition of the cut tree, S_e is a minimum cut (or simply *mincut*) separating i and j and therefore has value no greater than the value of any other cut separating i and j. But $\delta(S)$ is precisely such a cut. ■

Lemma 5.1.2 Let f be a proper function. Then, for any $S \subset V$, we have

$$f(S) \leq \max_{e \in \gamma(S)} f(S_e).$$

Proof: Let $(V_1, \ldots, V_k)$ be the vertex sets of the components of the cut tree after removing the vertices in S. By definition, since $V - S = V_1 \cup \ldots \cup V_k$ and the V_i are disjoint, we have

$$f(S) = f(V - S) \leq \max(f(V_1), f(V_2), \ldots, f(V_k)). \tag{5.1}$$

Consider any V_i. Assume that $1 \in V_i$ (otherwise consider $V - V_i$). Notice that

$$V - V_i = \cup_{e \in \gamma(V_i)} S_e,$$

and the sets S_e appearing in the union are disjoint. Hence,

$$f(V - V_i) = f(V_i) \leq \max_{e \in \gamma(V_i)} f(S_e).$$

But, by definition of V_i, we must have $\gamma(V_i) \subseteq \gamma(S)$. Therefore,

$$f(V_i) \leq \max_{e \in \gamma(S)} f(S_e). \tag{5.2}$$

Combining (5.1) and (5.2), we obtain the desired result. ■

These lemmas have a number of interesting consequences, which are easy to derive.

Theorem 5.1.3 $\max_S \{f(S) - x(\delta(S))\} = \max_{e \in H} \{f(S_e) - x(\delta(S_e))\}$.

Proof: For any given set S, Lemmas 5.1.1 and 5.1.2 imply that

$$f(S) - x(\delta(S)) \leq \max_{e \in \gamma(S)} f(S_e) - \max_{e \in \gamma(S)} x(\delta(S_e)) \leq \max_{e \in \gamma(S)} (f(S_e) - x(\delta(S_e))),$$

so that

$$\max_S \{f(S) - x(\delta(S))\} \leq \max_{e \in H} \{f(S_e) - x(\delta(S_e))\}.$$

Obviously the inequality must in fact be an equality. ■

The theorem allows us to solve the problem of finding a set S such that $f(S) - x(\delta(S)) = \max_T \{f(T) - x(\delta(T))\}$ (and hence allows us to solve the separation problem) by solving $n-1$ maximum flow problems and restricting attention to the $n-1$ cuts defined by the cut tree. In addition, this theorem generalizes a result of Padberg and Rao [96] for T-cuts (cuts S for which $|S \cap T|$ is odd) or odd cuts (for which $|S|$ is odd). Their result states that the minimum T-cut or odd cut is among the cuts of the Gomory-Hu tree. To derive this result from Theorem 5.1.3, we set

$$f(S) = \begin{cases} M & \text{if } |S \cap T| \text{ odd} \\ 0 & \text{otherwise,} \end{cases}$$

where $M > x(E)$. Using similar logic, our theorem shows that the minimum cut for any proper function is among the cuts of the Gomory-Hu tree. Ravi and Klein [104] independently showed that Padberg and Rao's result could be generalized to proper functions f with $f_{\max} = 1$.

The cut tree does not immediately give the *minimal* maximally violated sets. Assume that $x \in \mathbb{N}^{|E|}$, and let $\Delta_{\max}$ be the maximum deficiency; that is, $\Delta_{\max} = \max_T \{f(T) - x(\delta(T))\}$. Let $g(S) = \max\{f(S) - \Delta_{\max}, 0\}$. By Observation 2.0.10, g is a proper function. For all sets S, $x(\delta(S)) \geq g(S)$, and a set S is maximally violated iff $x(\delta(S)) = g(S)$. Let H denote the edges in the cut tree. The following lemma will help us identify the minimal maximally violated sets.

Lemma 5.1.4 Any maximally violated set S is a minimum cut between s and t for some edge $e = (s,t) \in H$ satisfying $w_e = g(S_e) = x(\delta(S)) = g(S)$.

Proof: Let S be any maximally violated set, so that $x(\delta(S)) = g(S)$. From Lemma 5.1.2, there must exist an edge $e = (s,t)$ in $\gamma(S)$ with $g(S_e) \geq g(S)$. Lemma 5.1.1 implies that $w_e \leq x(\delta(S))$. But since $w_e = x(\delta(S_e))$ and $x(\delta(S_e)) \geq g(S_e)$ it must be the case that

$w_e = x(\delta(S)) = g(S) = g(S_e)$. Since S_e is a minimum cut between s and t of value w_e it follows that S is also a minimum cut between s and t. ■

For each edge $e = (s,t) \in H$ with $w_e = g(S_e)$, we shall keep track of *all* (s,t) mincuts (that is, minimum cuts between s and t). To do this, we use the compact representation of all (s,t) mincuts due to Picard and Queyranne [100]. The combination of the Gomory-Hu tree and the Picard-Queyranne representation has also previously been used by Gusfield and Naor [59], though for different reasons.

In network flow theory, a *residual graph* of an s-t flow ξ in a graph with capacities x_e consists of the directed edges for which $x_e - \xi_e$ is non-zero. If ξ is a maximum flow, then there is no directed path in the residual graph from s to t, since then there would be an augmenting path. The Picard-Queyranne representation is a directed graph G_e formed from the residual graph of a maximum flow from s to t for the edge $e = (s,t) \in H$. The connected components of the residual graph not containing s and t are disregarded. The graph G_e is constructed by contracting each strongly connected component, as well as the set of all vertices reachable from s, and the set of all vertices that can reach t. Because all strongly connected components are contracted and there is no path from s to t, G_e is acyclic. Each vertex of G_e is a supervertex representing a set of vertices of the original graph; furthermore, the supervertices of G_e form a partition on V. For notational simplicity, let S and T denote the supervertices of G_e containing s and t respectively. Picard and Queyranne observe that there is a 1-1 correspondence between the (s,t) mincuts and the (T,S) *dicuts* of G_e, where a (T,S) dicut is a cut with all arcs directed from the side of the cut containing T to the side containing S. Given a maximum flow, the digraph G_e can be computed in $O(m)$ time since the residual graph contains $O(m)$ edges.

Consider a topological ordering of G_e. Because we contracted into the supervertex T all vertices in the residual graph that can reach t, the first supervertex in the ordering must be T. Also, there must be a directed path in G_e from every supervertex to S, so S must be the last supervertex in the ordering. By definition, all the supervertices smaller (or bigger) than some supervertex A in the ordering must induce a (T,S) dicut and hence an (s,t) mincut but, clearly, not all (s,t) mincuts arise in this fashion. Nevertheless, we will show that we can limit our attention to particular (s,t) mincuts arising in this way.

Lemma 5.1.5 Let $e = (s,t)$ be an edge in the Gomory-Hu tree H such that $w_e = g(S_e)$. There exists a maximally violated set separating s from t if and only if there exists a supervertex A of G_e with $g(A) \geq g(S_e)$.

Proof: Only if part. By Lemma 5.1.4 and the properties of G_e, any maximally violated set C separating must be the union of supervertices A_i and must have $g(C) = g(S_e)$. By the maximality property of proper functions, at least one of these supervertices A_i must satisfy $g(A) \geq g(S_e)$.

If part. Choose the first (last) supervertex A in the ordering such that $g(A) \geq g(S_e)$. Consider the union C of all the predecessors (successors) of A in G_e. This set C induces an (s,t) mincut. Moreover, $C - A$ consists of the union of supervertices A_i with $g(A_i) < g(S_e)$, so that by maximality, $g(C - A) < g(S_e)$. By Corollary 2.0.4, the maximum of $g(C - A)$, $g(A)$, and $g(C)$ cannot be uniquely attained, so that $g(C) \geq g(S_e)$. The Gomory-Hu tree implies that $x(\delta(C)) = w_e = g(S_e)$, so that $x(\delta(C)) \leq g(C)$. But $g(C) \leq x(\delta(C))$, so that it must follow that $x(\delta(C)) = g(C)$ and C is a maximally violated set. ■

Theorem 5.1.6 Let e be an edge in the Gomory-Hu tree H such that $w_e = g(S_e)$. Let A be the first vertex in the topological ordering of G_e such that $g(A) \geq g(S_e)$, and let C be A together with its predecessors in G_e. If there exists a minimal maximally violated set separating t from s, it must be C.

Proof: Suppose there is another minimal maximally violated set C' containing t but not s. By Corollary 2.0.8, a minimal maximally violated set cannot cross a maximally violated set, since this would contradict minimality. Hence, $C' \subseteq C$. Since C' is a maximally violated set, it must be the union of supervertices of G_e. Moreover, it must contain A since $g(C') \geq g(S_e)$ and A is the only supervertex within C which has a $g(A) \geq g(S_e)$. Furthermore, since C' corresponds to an (T,S) dicut of G_e, it contains all predecessors of A in G_e. Thus $C' = C$.

■

We can find such another such set separating s from t in a similar manner, by going backwards in the topological ordering. Each set can be found in $O(m + n\omega_g)$ time: $O(m)$ time to construct G_e and perform a topological sort, and $O(n\omega_g)$ time to find the supervertex A with $g(A) \geq g(S_e)$. Note that the time ω_g for the weak oracle to compute g must be

the same as ω_f. By doing this for all edges $e \in H$ with $w_e = g(S_e)$, one thus constructs in $O(nm + n^2\omega_f)$ time a family of $O(n)$ maximally violated sets guaranteed to contain all minimal maximally violated sets; this follows from Lemma 5.1.4. The minimal maximally violated sets can be obtained from this family in $O(n^2)$ time by finding the minimal sets in the family. This can be done by keeping track, for each vertex, of the set (if unique) of smallest cardinality containing it.

Let $MF(n,m)$ denote the time taken to compute a maximum flow on n vertices and m edges. The Gomory-Hu cut tree can be computed in $(n-1)\cdot MF(n,m)$ time. The preceding discussion yields the following theorem.

Theorem 5.1.7 The strong oracle MAX-VIOLATED(f,x) can be implemented in $O(nMF(n,m)+nm+n^2\omega_f)$ time for any proper function f and $x \in \mathbb{N}^{|E|}$.

The currently best known algorithm for maximum flow is due to Phillips and Westbrook [99], and runs in $O(mn\log_{m/n} n + n\log^{2+\epsilon} n)$ time. An algorithm due to King, Tarjan, and Rao [72] runs in $O(mn + n^{2+\epsilon})$ time, and so is slightly faster on sparse graphs.

5.1.2 An Implementation for APPROX-PROPER

A somewhat faster implementation of the strong oracle can be given for the sequence of calls generated by the algorithm APPROX-PROPER.

As we observed in Section 3.3, all calls to the strong oracle in the algorithm APPROX-PROPER for a proper function f are equivalent to calls to MAX-VIOLATED$(f, F \cup F_{p-1})$, for an edge set F in APPROX-UNCROSSABLE and an edge set F_{p-1} from phase $p-1$ of APPROX-PROPER. Since $|F \cup F_{p-1}| \leq \min(nf_{\max}, m) = m'$, we only need to worry about these m' edges in implementing the strong oracle. Moreover, in the construction of the Gomory-Hu cut tree, we need not solve the maximum flow problems to optimality. We can stop as soon as the flow has value $f_{\max}$, as is justified below. Such a flow can be obtained in $O(m'f_{\max})$ time by locating up to $f_{\max}$ augmenting paths. For small values of $f_{\max}$, we can construct the Gomory-Hu cut tree without using a maximum flow subroutine. For example, the case $f_{\max} = 1$ reduces to finding connected components while the case $f_{\max} = 2$ reduces to computing the 2-edge-connected components of a graph (which can be done in linear

time [3, pgs. 179–187]).

To avoid computing maximum flows to optimality, we modify the procedure for constructing the cut tree, since whenever the maximum flow has value greater or equal to $f_{\max}$ we do not need to use information from the associated mincut. To see this, note that we only consider edges of the cut tree for which $w_e = g_p(S_e)$, where $g_p(S) = \max\{f(S) - \Delta_{\max}, 0\}$; recall that in phase p, the maximum deficiency $\Delta_{\max} = f_{\max} - p + 1 \geq 0$. Our modified procedure maintains a forest on each supervertex of the cut tree. In the classical algorithm, these forests are empty but, in our case, the edges of these forests correspond to maximum flow problems whose value was at least $f_{\max}$. If all forests are trees, we replace every supervertex by its associated tree and we output the resulting tree as the modified Gomory-Hu cut tree. Otherwise, we select two vertices, say u and v, of two different components of a forest of the same supervertex, say A. We compute the maximum flow (up to the value $f_{\max}$) between u and v in the shrunk graph, as in the classical procedure. If the maximum flow value is at least $f_{\max}$, we add the edge (u, v) to the forest of the supervertex A; otherwise, we split A (and its forest) as in the classical algorithm. The correctness of this procedure follows from the correctness of the Gomory-Hu procedure: the reason for maintaining forests on the supervertices is so that no "underestimated" edge of value $f_{\max}$ appears in the shrunk graph.

We should also point out that Lemma 5.1.1 is still valid for this modified cut tree. Thus, for the proper function f and the incidence vector of a graph with at most m' edges, the modified cut tree can be constructed in $O(nm'f_{\max})$ time and the separation problem can be solved in $O(nm'f_{\max} + n\omega_f)$ time. Using the argument of the previous section, we can then implement the strong oracle in $O(nm'f_{\max} + n^2\omega_f)$ time.

We can do still better, however, by noticing that the subsequent calls to the strong oracle within a phase can be performed easily by updating the information in the Picard-Queyranne representations G_e. The first call to the strong oracle from Approx-Uncrossable in phase p will be of the form Max-Violated(f, F_{p-1}) (since $F = \emptyset$). In each iteration of Approx-Uncrossable the algorithm adds an edge $\tilde{e}$ to F and calls Max-Violated$(f, F_{p-1} \cup F)$. Notice that the minimal maximally violated sets corresponding to $F_{p-1} \cup F$ must be maximally violated sets for F_{p-1}. Thus we can update the $O(n)$

Picard-Queyranne representations G_e by adding the (bidirected) edge $\tilde{e}$ to the residual graphs and recomputing their strongly connected components in $O(m')$ time per residual graph. Adding the edge $\tilde{e}$ to the Picard-Queyranne representations eliminates all mincuts containing $\tilde{e}$ from consideration. Then, as in the discussion of the prior section, we create a family of $O(n)$ candidates for the minimal maximally violated sets and extract from this family the minimal sets. In this case though, we can just make one call to the weak oracle (instead of $O(n)$) per G_e since, by adding an edge to a residual graph, only one new strongly connected component can be created. Thus there is at most one new supervertex formed in each graph G_e. Recomputing the minimal maximally violated sets for each edge added therefore takes $O(nm' + n\omega_f)$ time. Given that it takes $O(nm' f_{\max} + n^2\omega_f)$ time to compute the initial sets of the phase, that at most n additional calls to MAX-VIOLATED are made during a phase, and that $f_{\max} \leq n$ for any edge-covering problem, we obtain the following running time.

Theorem 5.1.8 The calls to the strong oracle MAX-VIOLATED during a phase of APPROX-PROPER can be implemented in $O(n^2 m' + n^2\omega_f)$ time.

In Chapter 7, we will sometimes want to use the strong oracle as implemented here on variations of proper edge-covering problems for which $f_{\max} > n$. Given the preceding discussion, we can implement such an oracle in $O(nm' f_{\max} + n^2 m' + n^2\omega_f)$ time.

5.1.3 An Implementation for a Special Case

The calls to the strong oracle can be made still faster for a particular problem of interest. The problem is called the *survivable network design problem* or the *generalized Steiner network problem*; we will discuss it in more detail in Chapter 6. In this problem, a value r_{ij} is given for each pair of vertices i and j, and the object is to find the minimum-cost set of edges such that there are at least r_{ij} edge-disjoint paths between each i and j. The problem can be modelled by the integer program (IP) with the proper function $f(S) = \max_{i \in S, j \notin S} r_{ij}$. It is easy to see that the function is symmetric. For any two disjoint sets A and B, let $r_{cd} = \max_{i \in A \cup B, j \notin A \cup B} r_{ij}$. If $c \in A$, then $f(A) \geq r_{cd}$, else $f(B) \geq r_{cd}$. Thus the function also obeys the maximality property.

For the survivable network design problem, we don't need to construct the modified Gomory-Hu cut tree. Instead, we can use a maximum-cost spanning tree T of the graph having cost r_{ij} on edge (i,j). Any set of edges satisfying the connectivity requirements of the edges of T satisfies all given requirements r_{ij} (Gomory and Hu [54]). As in the general case, maximally violated sets must correspond to mincuts associated with an edge $e=(s,t)$ of T such that the cut value is $g_p(S_e) = r_{st} - \Delta_{\max} = r_{st} - f_{\max} + p - 1$. However, in this case, we have a 1-to-1 mapping: any such mincut must correspond to a maximally violated set. As a result, any minimal maximally violated set must be a *minimal* (s,t) mincut (separating s from t) or a *minimal* (t,s) mincut (separating t from s) for some edge $e=(s,t)$ in the tree T.

We maintain these minimal mincuts for the sequence of calls to the strong oracle over the entire algorithm. At the beginning of phase p we update a maximum s-t flow for each edge (s,t) in the tree T. We start from the flows that were computed in phase $p-1$ and find augmenting paths for each edge e in T up to the value $g_p(S_e)$, if possible. By doing this, in phase p we can detect any edge of T for which the cut value is $g_p(S_e)$. Thus over the course of the algorithm we must find at most $f_{\max}$ augmenting paths for the flow for each edge of T, leading to a total time bound of $O(nm'f_{\max})$ for maintaining these flows.

Suppose for an edge $e=(s,t)$ in T, the cut value is $g_p(S_e) = r_{st} - f_{\max} + p - 1$ in phase p. Initially, the minimal (s,t) mincut consists of all vertices reachable from s in the residual graph of a maximum flow from s to t. The minimal (t,s) mincut is similar. As before, we can extract the minimal maximally violated sets from these mincuts in $O(n^2)$ time. Whenever an iteration adds an edge $e=(u,v)$ to the edge set F, each minimal mincut is updated. For example for the minimal (s,t) mincut, if u is reachable from s then so is v, as is any vertex reachable from v by residual edges. If t becomes reachable then we disregard edge (s,t) in the tree T for the rest of the phase. The total time in phase p to update the minimal (s,t) mincut amounts to a search of the residual graph, and thus uses time $O(m')$. Thus the total time in a phase for updating all edges (s,t) of T is $O(nm')$. To find the new minimal maximally violated set (if any) resulting from the addition of edge (u,v), we search through the $O(n)$ candidate sets for the smallest maximally violated set containing u. This set will be a new minimal maximally violated set if it contains no other currently

minimal maximally violated sets. The search for the set takes $O(n)$ time. Therefore, the time for the calls to the strong oracle in a phase p is $O(nm')$ for the survivable network design problem, leading to an total time bound of $O(nm'f_{\max})$ over the course of the whole algorithm.

Theorem 5.1.9 The calls to the strong oracle MAX-VIOLATED for the survivable network design problem can be implemented in $O(nm'f_{\max})$ time.

5.2 Selecting Edges

5.2.1 A Simple $O(n^2 \log n)$ Implementation

The goal of this section is to show how to find the edge $e = (u, v)$ that minimizes $\epsilon(e) = \frac{c_e - d(u) - d(v)}{a(u)+a(v)}$ quickly in each iteration of APPROX-UNCROSSABLE or APPROX-PROPER-0-1. We will sometimes call $\epsilon(e)$ the *reduced cost* of edge e. We begin by giving a simple method that takes $O(n^2 \log n + \omega_h)$ time for all the edge selections, and then build on these ideas to obtain an implementation that takes $O(n^2 + n\sqrt{m \log \log n} + n\omega_h)$ time.

In order to implement this step, we will maintain a union-find structure on the set of vertices. We will use the union-find data structure due to Tarjan [120]. The sets of the union-find structure will be called *a-sets*. The a-sets in any iteration will correspond to the currently active sets, the sets active in previous iterations not contained in any currently active set, and vertices that have not yet been in any active set. Whenever an edge spanning two a-sets is selected but no new active set contains them, we merge together the two a-sets. Because the collection of active sets over all iterations form a laminar family (Lemma 3.2.2), the a-sets in an iteration can always be derived by merging together a-sets from the previous iteration. A-sets have the property that all vertices v in the same a-set have a value $a(v)$ which will be identical throughout the rest of the algorithm. Along with each a-set we will keep a bit that indicates whether the a-set is currently active or not. The time to update the a-sets through the course of the algorithm is $O(n\alpha(n, n) + n\omega_h)$ for merging the various components and updating the bits, where α is the inverse Ackermann function.

In APPROX-PROPER-0-1, the a-sets take on a particularly simple form. By Theorem

3.1.4, the active sets are the connected components C for which $h(C) = 1$. Since we merge a-sets together whenever a selected edge spans them, the a-sets that do not correspond to active sets will correspond to components C for which $h(C) = 0$. Thus the a-sets in a given iteration of Approx-Proper-0-1 are simply the connected components of F.

As a naive approach to finding the minimum reduced-cost edge, we can simply use $O(m\alpha(m, n))$ time to compute the reduced cost for each edge $e = (u, v)$ and to check whether or not the edge spans two different a-sets. By being somewhat more careful, we can reduce the time taken to find the minimum edge in dense graphs to $O(n \log n)$. We need three ideas for this reduced time bound. The first idea is to introduce a notion of time into the algorithm. We let the time T be 0 at the beginning of the algorithm, and increment it by the value of ϵ each time through the main loop. The second idea is that instead of computing the reduced cost for an edge every time through the loop, we can maintain a priority queue of edges, where the key of an edge is the time T at which its reduced cost is expected to be zero. We call this quantity the *addition time* of the edge. If we know whether the endpoints of an edge are in active sets or not, and assume that the activity (or inactivity) will continue indefinitely, it is easy to compute the addition time: it is simply the current time plus the current reduced cost of the edge. Of course the activity of a set can change, but this occurs only when it is merged with other sets, and only edges in the coboundary of the sets are affected. In this case, we can recompute the addition time for each affected edge, delete the element with the old addition time, and reinsert it with the new addition time. The last idea we need for the lower time bound is that we only need to maintain a single edge between any two a-sets. If there are parallel edges between any two a-sets, one of the edges will always have an addition time no greater than that of the others; hence the others may be removed from consideration altogether.

Combining these ideas, we get the following algorithm for edge selection: first, we calculate the initial key value (addition time) for each edge and insert each edge into the queue (in time $O(m \log n)$). Each time though the loop, we find the minimum reduced-cost edge $e = (u, v)$ by extracting the minimum element from the queue. Selecting edge e will cause some number of a-sets to be merged. Whenever we merge two a-sets A and B, we delete all edges incident to A and B from the queue. For each a-set D different from A and

B we update the keys of the two edges from A to D and B to D, select the one edge that has the minimum key value, then reinsert it into the queue. Each merge requires $O(n)$ queue insertions and deletions, and since there can be at most n merges, the total time spent in maintaining the queue will be $O(n^2 \log n)$. In practice, we would be somewhat more careful when several a-sets are merged together at once, to avoid inserting and deleting the same edges several times. However, the asymptotic running time remains the same. This time dominates the time spent in maintaining the a-sets.

5.2.2 A Better Time Bound By Using Packets

A time bound of $O(n(n + \sqrt{m \log\log n} + \omega_h))$ for selecting edges can be achieved by proving a lemma that allows irrelevant edges to be ignored, and by using the data structure idea of packets due to Gabow, Galil, and Spencer [45]. Before explaining this improvement, we must define some additional notation. Let $A(u)$ denote the a-set containing the vertex u. At any time, let a denote the number of a-sets. Assume that in choosing the next edge to add, the edge addition step breaks ties for smallest addition time according to some fixed numbering of the edges. Thus any set of edges is totally ordered by their addition times; in particular, the kth smallest edge is unique. Through the remainder of this subsection, we compare edges using addition time, not cost; e.g., "kth smallest edge" refers to addition time.

Lemma 5.2.1 Fix an iteration in the main loop of APPROX-UNCROSSABLE (or APPROX-PROPER-0-1). Consider an edge $(u, v) \in \delta(A(u))$ that is not among the $2k$ smallest edges of $\delta(A(u))$. Then (u, v) is not added to F until $A(u)$ has changed or $A(v)$ has changed or a has decreased by k.

Proof: Suppose (u, v) is added to F in an iteration when neither $A(u)$ nor $A(v)$ has changed. Consider an edge (u', v'), one of the $2k$ smallest edges of $\delta(A(u))$. When (u, v) is added to F, (u', v') has not been added (since $A(u)$ has not changed). Thus the reduced cost of (u', v') has increased, implying that $A(v')$ has changed. Thus the $2k$ distinct sets $A(v')$ have changed. These $2k$ changes must be the result of merging various a-sets which include the $2k$ sets $A(v')$. Thus a must have decreased by at least k. ■

The lemma above will allow us to ignore particular sets of edges during some portions of the main loop of APPROX-UNCROSSABLE or APPROX-PROPER-0-1. In order to take advantage of the lemma, we partition the main loop into *subphases*. Let r be a parameter to be chosen later. Each time a has decreased by r or more since the start of the last subphase, a new subphase will begin. We will designate certain edges to be *awake* in such a way that Lemma 5.2.1 will imply that an edge (u, v) that is not awake in this subphase need not be considered unless $A(u)$ or $A(v)$ changes. At the beginning of a subphase, we choose the $2r$ smallest edges in the coboundary of every a-set. Any edge chosen by the a-sets of both its endpoints will be initially designated an awake edge.

In order to keep track of the awake edges, we use a number of priority queues. The key for each entry in a queue will be the edge's current addition time. The awake edges incident to each a-set are partitioned into priority queues called *packets* of no more than $\log n$ edges each. One packet for each a-set will be called the *growing packet*; any edges added to the a-set will be inserted in this packet. All other packets of the a-set are *ordinary packets*. In addition to the packets, we also maintain a priority queue D. Each awake edge is incident to the two a-sets of its endpoints. Thus an awake edge is in two packets corresponding to these two a-sets. Any awake edge that is the minimum of both its packets is added to D; such an edge is called a *double minimum*. Notice that the edge with the smallest addition time in any iteration will be the minimum edge in D.

Given these data structures, the edge addition step will work as follows. To start a subphase, each a-set chooses the $2r$ smallest edges in its coboundary. The awake edges are organized into packets. Any edge that is a double minimum is placed in D. To select the next edge e for F, choose the smallest edge in D. Let A be the new a-set created by adding e to F. Delete all edges incident to vertices of A from their packets and from D. If this causes a new packet minimum to be a double minimum, add it to D. Note that now all packets corresponding to A are empty, so initialize a new growing packet. We now need to choose the awake edges incident to A. To do this, examine all edges incident to A, awake or not. Discard any parallel edges between A and other a-sets, always keeping the smallest. The $2r$ smallest undiscarded edges incident to A will be designated awake edges. Add each such edge (u, v) to the two growing packets of $A(u)$ and $A(v)$ (one of these is A).

Whenever a growing packet gets $\log n$ edges, make it an ordinary packet and start a new growing packet; also possibly add an entry to D for a new double minimum. The algorithm is correct because it maintains the defining properties of the packets and D.

Before we estimate the running time, we observe that in any subphase, for any a-set A, at most $3r$ edges of $\delta(A)$ become awake. To see this, notice at most $2r$ such edges are awake when any a-set A is initialized. After initialization, each iteration can make at most one more edge of $\delta(A)$ awake. Thus at most r more edges are made awake before the subphase ends. As a result, any a-set A has at most $3r/\log n$ packets at any point in a subphase.

First we bound the time for deletions and insertions from the priority queues. By the observation above, an iteration that decreases the number a of a-sets by j deletes at most $O(jr)$ edges from packets and at most $O(jr/\log n)$ edges from D. Thus all addition steps delete a total of $O(nr)$ edges from packets and $O(nr/\log n)$ edges from D, for a total time of $O(nr\log\log n + nr)$ for all deletions since packets have at most $\log n$ edges and D has at most n edges. To bound the time on the insertions, note that there can be at most $O(nr)$ edges in packets and at most $O(nr/\log n)$ edges in D. Since these bounds are no larger than the total number of edge deletions, the total time for edge insertions must also be $O(nr\log\log n)$.

Putting everything together, note that there are at most n/r subphases. Using linear-time selection, we can construct packets and D in $O(m)$ time at the start of each subphase. We need $O(n)$ time whenever an edge is selected for discarding parallel edges and examining the edges incident to A. Thus the total time is $O(n^2 + nm/r + nr\log\log n)$. Choosing $r = \sqrt{m/\log\log n}$ gives total time $O(n(n + \sqrt{m\log\log n}))$ for the edge addition step.

5.3 Implementing Efficient-Edge-Delete

This section shows how to implement Efficient-Edge-Delete in $O(n)$ time. The implementation is based on a tree $\mathcal{T}'$ and auxiliary arborescences τ_C which we now define.

The tree $\mathcal{T}'$ is a modification of the tree $\mathcal{T}$ used to represent the active sets over the course of the algorithm, as defined in Section 3.2.2. Recall that $\mathcal{T}$ is constructed by creating a vertex v_C for each $C \in \cup\mathcal{C}$, where $\cup\mathcal{C}$ is the collection of all active sets over all iterations,

plus the set V. The vertex v_D is a parent of v_C in $\mathcal{T}$ if D is the smallest set in $\cup\mathcal{C}$ that properly contains C. To make $\mathcal{T}'$, we create one additional child vertex for each internal node v_C of the tree to represent the vertices in C that are not in the other children of v_C.

For each vertex v_C in $\mathcal{T}'$, we construct an arborescence τ_C. Recall that a branching is a directed forest in which every node has in-degree at most 1, whereas an arborescence is a connected branching. Using the edges of F_C, we form a branching β_C on the nodes corresponding to the children of C in $\mathcal{T}'$. An edge (v_A, v_B) is created for each edge $e \in F_C$ when edge e is in the coboundary of the sets A and B. Each 2-edge is directed arbitrarily, while a 1-edge is directed towards the active set that defines it. The edges of F_C form a branching since two edges of F cannot be directed towards the same active set. It can be converted into an arborescence τ_C by adding a node connected to all the roots of the branching. Both the tree $\mathcal{T}'$ and the arborescences τ_C can be easily constructed during the edge addition stage of APPROX-UNCROSSABLE and the running time for their constructions can be charged to this stage.

Before we explain the clean-up procedure, we note that a 1-edge e is special if and only if all the edges of F_C added to τ_C ($C = C(e)$) after e form a "subarborescence" with the head of e as its root.

We find all special edges in $O(n)$ time as follows. We consider each active set $C \in \cup\mathcal{C}$ in turn. Suppose F_C consists of $e_1, \ldots, e_t$, where e_i was added before e_j for $i < j$. Let l_i denote the least common ancestor of the heads of $e_i, \ldots, e_t$ in τ_C. By the reasoning above, a 1-edge e_i is special if and only if l_i is the head of edge e_i in τ_C. The l_i's can be found in linear time by processing the edges in reverse order, by marking the nodes along the path to the ancestor and by stopping at the first previously marked node.

We would like to implement EFFICIENT-EDGE-DELETE by performing a top-down traversal of $\mathcal{T}'$. At each vertex v_C of $\mathcal{T}'$, we would process the edges of F_C in the reverse order in which they appear in F, detecting and possibly removing the special edges of F_C. To be able to do this, we need to argue that processing the edges of F in this order results in the same set of edges F' as if we removed edges by processing the edges of F in reverse order. This follows from observing that the removal of an edge e depends only on the edges in F_C and $\delta_F(C)$, and affects only the removal of edges in F_C and in F_D, where $e \in \delta(D)$. Such

a set D must be a child of vertex v_C, and any vertex v_A such that edges of $\delta_F(C)$ are in F_A must be an ancestor of C. Thus if we remove edges in a top-down traversal, the set of edges F' will be the same as before.

When visiting a vertex v_C in $\mathcal{T}'$, we need to decide which special edge of F_C to remove, if any. Call a child v_B of v_C *hit* if there exists an edge already processed but not removed that is simultaneously in the coboundaries of B and C. Assume first that we know the set of hit children of v_C. We need to remove the deepest special edge e in τ_C (if any) whose head is an ancestor in τ_C of all the hit children of v_C: then all remaining edges in $\delta(C)$ will be guaranteed to be in $\delta(A(e))$. Given that we know the hit nodes, we can detect the appropriate special edge to remove while determining the special edges of F_C. To find the hit nodes, whenever we keep an edge $e = (u, v) \in F'$, we mark the nodes of the paths in $\mathcal{T}'$ from the leaves u and v up to (but excluding) their common ancestor $v_{C(e)}$ as hit nodes. In order for this procedure to run in linear time, we stop before reaching $C(e)$ if we encounter an already hit node. The validity of this argument follows from the fact that we perform a top-down traversal of $\mathcal{T}'$. Therefore, the overall running time of the clean-up step is $O(n)$ time.

5.4 Implementing Proper-0-1-Edge-Delete

Compared to the other edge deletion steps, the edge deletion step for Approx-Proper-0-1 is relatively simple. To compute F' from F, we iterate through the components C of F. Given a component C, we root the tree at some vertex, put each leaf of the tree in a separate list, and compute h of each leaf. An edge joining a vertex to its parent is discarded if the h value for the set of vertices in its subtree is 0. Whenever we have computed the h value for all the children of some vertex v, we concatenate the lists of all the children of v, add v to the list, and compute h of the vertices in the list. We continue this process until we have examined every edge in the tree. Since there are $O(n)$ edges, the process takes $O(n + n\omega_h)$ time.

5.5 Putting Everything Together

We can now finally give time bounds on the algorithms Approx-Proper and Approx-Proper-0-1.

In Approx-Proper, there are $f_{\max}$ phases for a proper function f. In each phase, we call Approx-Uncrossable, which must select edges, call the strong oracle, then delete edges. By the reasoning in the previous sections, this takes $O(n(n+\sqrt{m \log\log n}+\omega_f)+(n^2m'+n^2\omega_f)+n) = O(n^2m'+n^2\omega_f)$ time per phase, using Efficient-Edge-Delete. This leads to an overall time bound of $O(f_{\max}(n^2m'+n^2\omega_f))$, proving Theorem 5.0.1. Notice that if we used Regular-Edge-Delete, we would have to call the strong oracle $O(n)$ times for the edge deletion stage and the time bound would be $O(n^2m'f_{\max}^2 + n^3\omega_f f_{\max})$. For the survivable network design problem mentioned in Section 5.1.3, the overall time bound is $O(nf_{\max}(nf_{\max}+\sqrt{m \log\log n}+\omega_f))$, using Efficient-Edge-Delete. In this case the time bound for the algorithm with Regular-Edge-Delete is still $O(n^2m'f_{\max}^2 + n^3\omega_f f_{\max})$.

In Approx-Proper-0-1, we use the edge addition step and Proper-0-1-Edge-Delete. The algorithm does not call the strong oracle. Therefore the running time is $O(n(n+\sqrt{m \log\log n}+\omega_h))$, proving Theorem 5.0.2.

CHAPTER 6

Applications

In the past few chapters, we have developed a $2\mathcal{H}(f_{\max})$-approximation algorithm for all weakly supermodular edge-covering problems, have proven its correctness, and have shown how to implement it efficiently for proper functions. In the next two chapters, we turn to the applications of this algorithm and of the techniques underlying the algorithm. In this chapter, we enumerate many interesting graph problems which are proper edge-covering problems, and we discuss how our work fits in with previous work on these problems.

6.1 The Survivable Network Design Problem

In the survivable network design problem, we are given a non-negative connectivity requirement r_{ij} for every unordered pair of vertices i, j. The goal is to find a minimum-cost subgraph in which each pair of vertices i, j is connected by at least r_{ij} edge-disjoint paths. This problem is also sometimes called the *generalized Steiner network problem.* It arises in the design of fiber-optic telephone networks [58]. The survivable network design problem is a proper edge-covering problem given the function $f(S) = \max_{i \in S, j \notin S} r_{ij}$. Setting $R = \max_{i,j} r_{ij}$, APPROX-PROPER gives a $2\mathcal{H}(R)$-approximation algorithm for the generalized Steiner network problem. By the discussion of Section 5.5, the algorithm runs in $O(Rn(Rn + \sqrt{m \log \log n}))$ time for this problem.

Before this thesis, no approximation algorithm was known even for the case $r_{ij} = \min(r_i, r_j)$, $r_i \in \{0, 1, 2\}$. However, several algorithms were known for special cases of the survivable network design problem. When $r_{ij} = k$ for all pairs of vertices i, j, the problem becomes the *minimum-cost k-edge-connected subgraph problem*; that is, the problem of finding the minimum-cost subgraph such that there are at least k edge-disjoint paths between every pair of vertices. In 1981, Frederickson and Ja'Ja' [39] developed a 3-approximation algorithm for the 2-edge-connected subgraph problem. Recently, Khuller and Vishkin [71] developed a 2-approximation algorithm for the k-edge-connected subgraph problem for any k, which runs in $O(kn^3 \log n)$ time. Their algorithm does not seem to extend to the "Steiner" variant of this problem in which $r_{ij} \in \{0, k\}$ for all i, j. There is a 3-approximation algorithm due to Klein and Ravi [74] that can solve the Steiner variant when $r_{ij} \in \{0, 2\}$.

When $r_{ij} \in \{0, 1\}$, the problem becomes the *generalized Steiner tree problem*. Given sets $T_i \subseteq V$, $i = 1, \ldots, p$, the generalized Steiner tree problem is that of finding a minimum-cost forest that connects all vertices in each T_i. In this case, $\max_{ij} r_{ij} = 1$, and thus we can use Approx-Proper-0-1 to obtain a $(2 - \frac{2}{\ell})$-approximation algorithm that runs in $O(n^2 + n\sqrt{m \log\log n})$ time, where $\ell = |\{v \in V : f(\{v\}) = 1\}| = \left|\bigcup_{i=1,\ldots,p} T_i\right|$. Agrawal, Klein, and Ravi [2] gave the first approximation algorithm for this problem. It also has a performance guarantee of $2 - \frac{2}{\ell}$. As we stated in the introduction, their algorithm led to the research in this thesis. Although their use of the primal-dual method is similar to ours, their algorithm is somewhat different: it shrinks and expands parts of the graph, uses recursive calls to itself to construct its solution, and has no equivalent of our edge deletion stage.

When $p = 1$, the generalized Steiner tree problem reduces to the classical Steiner tree problem. For a long time, the best approximation algorithm for this problem had a performance guarantee of $2 - \frac{2}{\ell}$ (for a survey, see Winter [128]) but recently Zelikovsky [130, 131] obtained an $\frac{11}{6}$-approximation algorithm. An improved $\frac{16}{9}$-approximation algorithm based upon Zelikovsky's ideas was later proposed by Berman and Ramaiyer [14]. Zelikovsky's algorithm runs in $O(\ell(m + n\ell + n \log n))$ time, and the Berman and Ramaiyer algorithm runs in $O(n^{7/2})$ time.

The performance guarantee of our algorithm can be shown to be tight for the Steiner tree problem. When $p = 1$, our algorithm reduces to the standard minimum-cost spanning tree heuristic as given in Goemans and Bertsimas [52]. The heuristic can produce solutions which have cost $2 - \frac{2}{\ell}$ times the optimal cost, as is shown by Goemans and Bertsimas [52].

6.2 The T-join Problem

Given an even subset T of vertices, the T-join problem consists of finding a minimum-cost set of edges that has odd degree at vertices in T and even degree at vertices not in T. Edmonds and Johnson [34] have shown that the T-join problem can be solved in polynomial time and can be formulated by the linear programming relaxation of (IP) with the proper function $h(S) = 1$ if $|S \cap T|$ is odd and 0 otherwise. Using Approx-Proper-0-1, we obtain a $(2 - \frac{2}{|T|})$-approximation algorithm for the T-join problem. The edge-removing step of Approx-Proper-0-1 guarantees that the solution produced is a T-join (see the next section).

The performance guarantee of our algorithm is tight for the T-join problem. Figure 6-1 (a)-(c) shows an example on 8 vertices in which the minimum-cost T-join with $T = V$ has cost $4 + 3\epsilon$, while the solution produced by the algorithm has cost 7, yielding a worst-case ratio of approximately $\frac{7}{4} = 2 - \frac{2}{8}$. Clearly the example can be extended to larger numbers of vertices and to an arbitrary set T.

When $T = \{s, t\}$, the T-join problem reduces to the shortest s-t path problem. Our algorithm finds the optimal solution in this case, since $2 - \frac{2}{|T|} = 1$.

6.3 The Minimum-Weight Perfect Matching Problem

The minimum-weight perfect matching problem is the problem of finding a minimum-cost set of non-adjacent edges that cover all vertices. This problem can be solved in polynomial time by a primal-dual algorithm discovered by Edmonds [31]. The fastest strongly polynomial-time implementation of Edmonds' algorithm is due to Gabow [44]. Its running time is $O(n(m + n \log n))$. For integral costs bounded by C, the best weakly polynomial

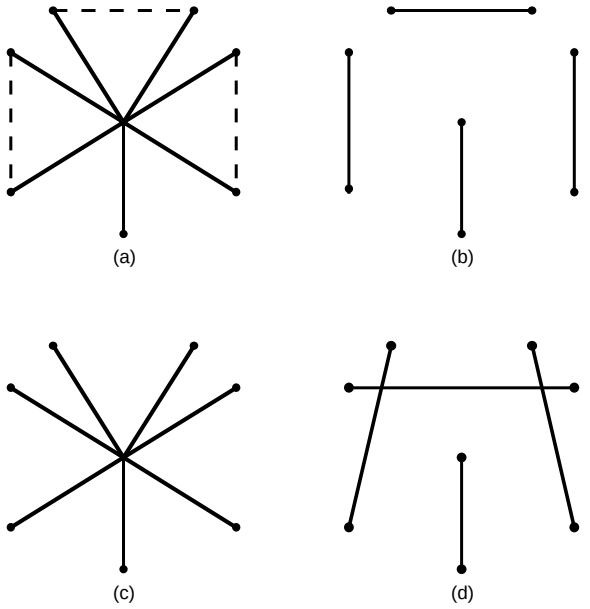

Figure 6-1: Worst-case example for V-join and matching. Graph (a) gives the instance: plain edges have cost 1, dotted edges have cost $1+\epsilon$, and all other edges have cost 2. Graph (b) is the minimum-cost solution. Graph (c) is the set of edges found by APPROX-PROPER-0-1, and graph (d) shows a bad (but possible) shortcutting of the edges to a matching.

algorithm runs in $O(m\sqrt{n\alpha(m,n)\log n}\ \log nC)$ time and is due to Gabow and Tarjan [47]. For instances drawn from the Euclidean plane, Vaidya [122] gives an $O(n^{2.5}\log^4 n)$-time implementation of Edmonds' algorithm.

These algorithms are fairly complicated and have high worst-case running times. This motivated the search for faster approximation algorithms. Reingold and Tarjan [110] have shown that the greedy procedure has a tight performance guarantee of $\frac{4}{3}n^{0.585}$ for general non-negative cost functions. Supowit, Plaisted and Reingold [119] and Plaisted [101] have proposed an $O(\min(n^2\log n, m\log^2 n))$-time approximation algorithm for instances that obey the triangle inequality. Their algorithm has a tight performance guarantee of $2\log_3(1.5n)$. As shown by Gabow [43], a scaling algorithm for the maximum-weight matching problem can be used to obtain an $(1+1/n^a)$-approximation algorithm ($a \geq 0$) for the minimum-weight perfect matching problem. Moreover, if the original exact algorithm runs in $O(f(m,n)\log C)$ time, the resulting approximation algorithm runs in $O(m\sqrt{n\log n} + (1+a)f(m,n)\log n)$ time. The fastest known scaling algorithm for

maximum-weight matching is the algorithm of Gabow and Tarjan [47] mentioned in the previous paragraph. Vaidya [121] obtains a $(3+2\epsilon)$-approximation algorithm for minimum-weight perfect matching instances satisfying the triangle inequality. His algorithm runs in $O(n^2 \log^{2.5} n \log(1/\epsilon))$ time.

The algorithm APPROX-PROPER-0-1 can be used to approximate the minimum-weight perfect matching problem when the edge costs obey the triangle inequality. We use the algorithm with the proper function $h(S)$ being the parity of $|S|$, i.e. $h(S) = 1$ if $|S|$ is odd and 0 if $|S|$ is even. This function is the same as the one used for the T-join problem when $T = V$. The algorithm returns a forest whose components have even size. More precisely, the forest is a V-join, and each vertex has odd degree. To see this, suppose a vertex v has even degree, and suppose that removing any edge incident to v results in two odd-sized components. Then the component containing v can be partitioned into v plus an even number of odd-sized components, contradicting the fact that the component must have even size.

The forest can be transformed into a perfect matching with no increase of cost in several ways. One way is to repeatedly take two edges (u, v) and (v, w) from a vertex v of degree three or more and to replace these edges with the edge (u, w). This procedure maintains the property that the vertices have odd degree. After $O(n)$ iterations, each vertex has degree one. Another method is to double each edge, transforming each component into an Eulerian graph of even size. We can then get a tour of each graph by shortcutting the traversal of the graph. Each tour defines two matchings, and we take the cheapest of the two. For either method, the overall procedure gives an approximation algorithm for weighted perfect matching which runs in $O(n^2 + n\sqrt{m \log \log n})$ time and has a performance guarantee of $2 - \frac{2}{n}$.

The performance guarantee of the algorithm is tight for this problem also, as is shown in Figure 6-1 (d).

We will discuss the application of our algorithm to this problem in much more detail in the computational study presented in Chapter 8.

6.4 Point-to-Point Connection Problems

In the fixed point-to-point connection problem, we are given a set $C = \{c_1, \ldots, c_p\}$ of sources and a set $D = \{d_1, \ldots, d_p\}$ of destinations in a graph $G = (V, E)$ and we need to find a minimum-cost set F of edges such that c_i is connected to d_i in F [84]. This problem is a special case of the generalized Steiner tree problem where $T_i = \{c_i, d_i\}$. In a variation of the problem called the non-fixed point-to-point connection problem, each component of the forest F is only required to contain the same number of sources and destinations. Both problems are NP-complete [84]. They arise in the context of circuit switching and VLSI design.

The non-fixed case is a proper edge-covering problem with $h(S) = 1$ if $|S \cap C| \neq |S \cap D|$ and 0 otherwise. For this problem, we obtain a $(2 - \frac{1}{p})$-approximation algorithm by using Approx-Proper-0-1. No previous approximation algorithm was known.

6.5 Exact Partitioning Problems

In the exact tree (cycle, path) partitioning problem, for a given k we must find a minimum-cost collection of vertex-disjoint trees (cycles, paths) of size k that cover all vertices. These problems and related NP-complete problems arise in the design of communication networks, vehicle routing and cluster analysis. These problems generalize the minimum-weight perfect matching problem (in which each component must have size exactly 2), the traveling salesman problem, the Hamiltonian path problem and the minimum-cost spanning tree problem. No approximation algorithms for the general problems were known prior to this thesis.

We can approximate the exact tree, cycle and path partitioning problems for instances that satisfy the triangle inequality. To do this, we consider the proper edge-covering problem with the function $h(S) = 1$ if $S \not\equiv 0 \pmod{k}$ and $h(S) = 0$ otherwise. The algorithm Approx-Proper-0-1 finds a forest in which each component has a number of vertices which is a multiple of k, and such that the cost of the forest is within $2 - \frac{2}{n}$ of the optimal such forest. Obviously the cost of the optimal such forest is a lower bound on the optimal exact tree and path partitions. Given the forest, we duplicate each edge and find a tour of each component by shortcutting the resulting Eulerian graph on each component. If

we remove every kth edge of the tour, starting at some edge, the tour is partitioned into paths of k nodes each. Some choice of edges to be removed (i.e., some choice of starting edge) accounts for at least $\frac{1}{k}$ of the cost of the tour, and so we remove these edges. Thus this algorithm is a $(4(1-\frac{1}{k})(1-\frac{1}{n}))$-approximation algorithm for the exact tree and path partitioning problems.

To produce a solution for the exact cycle partitioning problem, we add the edge joining the endpoints of each path; given the triangle inequality, this at most doubles the cost of the solution produced. We claim, however, that the algorithm is still a $(4(1-\frac{1}{k})(1-\frac{1}{n}))$-approximation algorithm for the cycle problem. To see that this claim is true, note that the following linear program is a linear programming relaxation of the exact cycle partitioning program, given the function h above:

$$\begin{aligned} \text{Min} \quad & \sum_{e \in E} c_e x_e \\ \text{subject to:} & \\ & x(\delta(S)) \geq 2h(S) && S \subset V \\ & x_e \geq 0 && e \in E. \end{aligned}$$

Its dual is

$$\begin{aligned} \text{Max} \quad & 2 \sum_{S \subset V} h(S) \cdot y_S \\ \text{subject to:} & \\ & \sum_{S : e \in \delta(S)} y_S \leq c_e && e \in E, \\ & y_S \geq 0 && \emptyset \neq S \subset V. \end{aligned}$$

We know the algorithm produces a solution y that is feasible for this dual such that $\sum_{e \in F'} c_e \leq (2 - \frac{2}{n}) \sum y_S$. The argument above shows how to take the set of edges F' and produce a set of edges T such that T is a solution to the exact cycle partitioning

problem, and $\sum_{e \in T} c_e \leq 4(1 - \frac{1}{k}) \sum_{e \in F'} c_e$. Thus

$$\sum_{e \in T} c_e \leq 8 \left(1 - \frac{1}{k}\right) \left(1 - \frac{1}{n}\right) \sum y_S.$$

Since $2 \sum y_S$ is the dual objective function, $2 \sum y_S$ is a lower bound on the cost of the optimal exact cycle partition, Z_C^*. Thus

$$\sum_{e \in T} c_e \leq 4 \left(1 - \frac{1}{k}\right) \left(1 - \frac{1}{n}\right) Z_C^*.$$

CHAPTER 7

Extensions

The techniques used in developing the main algorithms are sufficiently general that they can be applied to other graph problems which do not fall in the class of proper edge-covering problems, and even to some problems which cannot be modelled by the integer program (IP). In this chapter we consider how the main algorithms can be modified to solve some of these problems. We begin with fairly simple modifications in the following sections, and move towards more complicated modifications later in the chapter which will allow us to solve the prize-collecting traveling salesman problem and the k-vertex-connected subgraph problem. These modifications result in approximation algorithms with performance guarantees better than the best previously known guarantees (as for the prize-collecting problems) or algorithms for problems that had no previously known approximation algorithms (as for the k-vertex-connected subgraph problem). Perhaps more importantly, this chapter shows that the primal-dual technique developed in this thesis is surprisingly versatile and robust.

7.1 Solving Some Non-Proper Edge-Covering Problems

We observed in a previous chapter that the reason we cannot in general implement APPROX-UNCROSSABLE in polynomial time is that we do not know how to implement the strong oracle for a general uncrossable function in polynomial time. It turns out that there are other subclasses of uncrossable functions for which it is easy to find the minimal maximally

SYMM-MAX-ORACLE (h, F)

```
1   Let C_1, ..., C_k, I_1, ..., I_l be the connected components of (V, F), where h(C_i) = 1 and
    h(I_i) = 0
2   If h(⋃_i C_i) = 0
3     return {C_1, ..., C_k}
4   else
5     S ← ⋃_i C_i
6     For j ← 1 to l
7       If h(I_j ∪ ⋃_i C_i) = 1
8         S ← S ∪ I_j
9     return {C_1, ..., C_k, V − S}
```

Figure 7-1: The strong oracle for the symmetrized version of an uncrossable function h that obeys the maximality property.

violated sets. For any of these problems we can use the algorithm APPROX-UNCROSSABLE to get a 2-approximation algorithm to the problem.

One such class of functions is the uncrossable functions $h : 2^V \rightarrow \{0,1\}$ which obey the maximality property $h(A \cup B) \leq \max(h(A), h(B))$ for disjoint A and B. If h is also symmetric, then h is proper, so the interesting case is when h is not symmetric. Given the maximality property, we can maintain the active sets in the same way as we did in the algorithm APPROX-PROPER-0-1; that is, in an iteration of the algorithm, we divide the connected components C of the graph into two sets: $C \in \mathcal{C}$ if $h(C) = 1$ and $C \in \mathcal{I}$ if $h(C) = 0$. Then $\mathcal{C}$ corresponds exactly to the active sets, as is shown in Theorem 3.1.4. Given that the uncrossable functions may not be symmetric, we then have a $(2 - \frac{1}{\ell_h})$-approximation algorithm for this class of functions, where $\ell_h = \{v \in V : h(\{v\}) = 1\}$.

As we have mentioned after Theorem 4.1.6, if h is not symmetric, we can consider a symmetric version h' of h and obtain a performance guarantee of $2 - \frac{2}{\ell_h + 1}$. When h obeys the maximality property, we can also easily provide an implementation of the strong oracle for h', although h' itself will no longer obey the maximality property. The strong oracle for h', SYMM-MAX-ORACLE, is given in Figure 7-1. We now prove its correctness.

Theorem 7.1.1 If h is an uncrossable function that obeys the maximality property, and $h'(S) = \max(h(S), h(V - S))$ for all S, then SYMM-MAX-ORACLE(h', F) returns the minimal violated sets for h' with respect to the edge set F.

Proof: By the arguments above about the active sets for h and by Lemma 3.2.3, the oracle correctly returns all connected components C such that $h(C) = 1$. The lemma also states that there exists at most one more minimal violated set A. Any candidate violated set A must be the union of connected components of F such that $h(V - A) = 1$; furthermore, since A must be disjoint from the components C_i such that $h(C_i) = 1$, the set $V - A$ must contain $\bigcup_i C_i$. If $h(\bigcup_i C_i) = 0$, then by the maximality property, no additional violated set can exist. Suppose $h(\bigcup_i C_i) = 1$, and let A be the additional set returned by the algorithm. To see that $h(V - A) = 1$, let $B = \bigcup_i C_i \cup I_j$, for some $I_j \subset V - A$ (if any exists). Pick any $I_k \in (V - A) - B$. By construction, we know that $h(B) = 1$ and $h(\bigcup_i C_i \cup I_j) = 1$, but that $h((bigcup_i C_i \cup I_j) - B) = h(I_j) = 0$. Therefore $h(B \cup I_k) = 1$. Set B to $B \cup I_k$ and repeat the argument until $B = V - A$.

Now suppose there is a smaller violated set A'. Then there must exist I_j such that $I_j \subset V - A'$, but $I_j \not\subset V - A$. But by construction of the algorithm, $h(I_j \cup \bigcup_i C_i) = 0$, which by maximality implies that $h(V - A') = 0$, a contradiction. ■

The uncrossable functions that obey the maximality property turn out to include some interesting problems, which we discuss in the next two subsections. In addition, some recent work on one of these problems leads us to propose a simpler 2-approximation algorithm for a subclass of this class of functions.

7.1.1 Lower-Capacitated Partitioning Problems

The *lower-capacitated partitioning problems* are like the exact partitioning problems except that each component is required to have at least k vertices rather than exactly k vertices. The lower capacitated *cycle* partitioning problem is a variant of the 2-matching problem. More precisely, the cases $k = 2$, 3 and 4 correspond to integer, binary and triangle-free binary 2-matchings respectively. The lower-capacitated cycle partitioning problem is NP-complete for $k \geq 5$ (Papadimitriou in Cornuéjols and Pulleyblank [22] for $k \geq 6$ and Vornberger [124] for $k = 5$), polynomially solvable for $k = 2$ or 3 (Edmonds and Johnson [33]), while its complexity for $k = 4$ is open. Imielinska, Kalantari, and Khachiyan [63] have shown that the lower-capacitated tree partitioning problem is NP-complete for $k \geq 4$, even if the edge costs obey the triangle inequality.

The lower-capacitated tree partitioning problem is an uncrossable edge-covering problem with the uncrossable function $h(S) = 1$ if $0 < |S| < k$ and 0 otherwise. Notice that this function also obeys the maximality property. Hence we have a $(2 - \frac{2}{n})$-approximation algorithm for this problem for any k (by using the symmetrized version of h). Furthermore, assuming the triangle inequality, the algorithm can be turned into a $(2 - \frac{2}{n})$-approximation algorithm for the lower-capacitated cycle partitioning problem and a $(4 - \frac{4}{n})$-approximation algorithm for the lower-capacitated path partitioning problem by using algorithms and analysis similar to that given for the exact path and cycle partitioning problems.

As far as we know, there was no approximation algorithm known for this problem for general k prior to the work in this thesis. However, after a preliminary part of this thesis appeared [53], Imielinska et al. [63] showed that the algorithm Approx-Lower-Cap-Tree given in Figure 7-2 is a 2-approximation algorithm for the lower-capacitated tree partitioning problem. By duplicating edges and shortcutting to tours, the algorithm also implies a 4-approximation algorithm for the lower-capacitated cycle partitioning problem when edge costs obey the triangle inequality. The algorithm takes as input a graph $G = (V, E)$, edge costs c, and a bound k. It computes a minimum-cost spanning tree, and sorts the edges by cost. The algorithm then considers each edge of the spanning tree in order, starting with the minimum-cost edge. Each edge will span two connected components: if at least one of these is currently smaller than k, the edge gets added to the final edge set. Because the algorithm merely requires finding and sorting the edges of a minimum-cost spanning tree, it can be made to run in $O(m + n \log n)$ time for general graphs and $O(n \log n)$ time for Euclidean graphs, which is faster than our algorithms.

We can, however, recast their algorithm into our framework and in the process improve their approximation algorithm for the cycle problem from a 4-approximation algorithm to a 2-approximation algorithm. In addition, we can generalize the algorithm to handle all uncrossable functions that obey the maximality property and the additional property that whenever $h(S) = 0$, then $h(T) = 0$ for all $T \supseteq S$. We give our amended algorithm in Figure 7-3, and structure it to resemble Approx-Proper-0-1, although this will turn out to be unnecessary. There are two main differences between Approx-Proper-0-1 and this algorithm. First, we delete edges in the reverse of the order in which they were added,

APPROX-LOWER-CAP-TREE (V, E, c, k)

```
1   F ← ∅
2   W ← MINIMUM-COST-SPANNING-TREE (V, E, c)
3   Sort edges of W = {e_1, ..., e_{n-1}} so that c_{e_1} ≤ ··· ≤ c_{e_{n-1}}
4   C ← {{v} : v ∈ V}
5   I ← ∅
6   i ← 1
7   While |C| > 0
8     If e_i = (u, v), u ∈ C_p ∈ C ∪ I, v ∈ C_q ∈ C ∪ I, and either |C_p| < k or |C_q| < k
9       F ← F ∪ {e_i}
10      Delete C_p and C_q from C and I
11      If |C_p ∪ C_q| < k
12        C ← C ∪ {C_p ∪ C_q}
13      else
14        I ← I ∪ {C_p ∪ C_q}
15    i ← i + 1
16  return F
```

Figure 7-2: Imielinska et al.'s algorithm for the lower-capacitated tree partitioning problem.

instead of in an arbitrary manner. Second, in each iteration we increase the dual variables for *all* connected components, instead of the connected components C for which $h(C) = 1$. Notice that this implies that the variables $d(v)$ will be identical for all v, and thus the edge selected in each iteration will simply be the minimum-cost edge joining two distinct connected components; that is, the edges selected are exactly the edges of the minimum-cost spanning tree. Hence we can reduce the algorithm to the algorithm in Figure 7-4, which is similar to the algorithm of Imielinska et al.: we add the edges of the spanning tree until $h(C) = 0$ for each connected component C. Then we delete unnecessary edges in the reverse of the order they were added. For purposes of analysis, we will concentrate on the first version of the algorithm.

By construction $h(N) = 0$ for all connected components of F'. Thus the feasibility of the solution produced by the algorithm follows by Theorem 3.1.2, given that h obeys the maximality property. We also show the following theorem.

Theorem 7.1.2 Let Z^*_{IP-h} be the value of an optimal solution to the integer program (IP_h) given by a uncrossable function $h : 2^V \rightarrow \{0,1\}$ that obeys the maximality property and the property that if $h(S) = 0$ then $h(T) = 0$ for all $T \supseteq S$. Then APPROX-RESTRICTED-UNCROSSABLE produces a set of edges F' and a feasible solution y to (D_h) such that

$$\sum_{e \in F'} c_e \leq 2 \sum_S h(S) y_S \leq 2 Z^*_{IP}.$$

Proof: Using the argument of Section 4.1, we can reduce the theorem to showing that

$$\sum_{S \in \mathcal{C} \cup \mathcal{I}} y_S |\delta_{F'}(S)| \leq 2 \sum_S h(S) y_S,$$

which follows from

$$\sum_{S \in \mathcal{C} \cup \mathcal{I}} |\delta_{F'}(S)| \leq 2|\mathcal{C}|,$$

if this statement is true at every iteration. This statement differs from the standard total-degree inequality in that the sum on the left-hand side is taken over both the active and inactive components, rather than just the active sets. This difference is caused by increasing the dual variables for all connected components in each iteration.

APPROX-RESTRICTED-UNCROSSABLE (V, E, c, h)

1 $F \leftarrow \emptyset$
2 *Comment: Implicitly set* $y_S \leftarrow 0$ *for all* $S \subset V$
3 $\mathcal{C} \leftarrow \{\{v\} : v \in V, h(\{v\}) = 1\}$
4 $\mathcal{I} \leftarrow \{\{v\} : v \in V, h(\{v\}) = 0\}$
5 For each $v \in V$ do $d(v) \leftarrow 0$
6 $i \leftarrow 0$
7 **While** $|\mathcal{C}| > 0$
8 $i \leftarrow i + 1$
9 Find edge $e_i = (u, v)$ with $u \in C_p \in \mathcal{C} \cup \mathcal{I}$, $v \in C_q \in \mathcal{C} \cup \mathcal{I}$, $C_p \neq C_q$ that minimizes $\epsilon = \frac{c_e - d(u) - d(v)}{2}$
10 $F \leftarrow F \cup \{e_i\}$
11 For all v do $d(v) \leftarrow d(v) + \epsilon$
12 *Comment: Implicitly set* $y_C \leftarrow y_C + \epsilon$ *for all* $C \in \mathcal{C} \cup \mathcal{I}$
13 Delete C_p and C_q from $\mathcal{C}$ and $\mathcal{I}$
14 **If** $h(C_p \cup C_q) = 1$
15 $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_p \cup C_q\}$
16 **else**
17 $\mathcal{I} \leftarrow \mathcal{I} \cup \{C_p \cup C_q\}$
18 $F' \leftarrow F$
19 **for** $j \leftarrow i$ **downto** 1
20 If $h(N) = 0$ for all connected components N of $(V, F' - \{e_j\})$
21 $F' \leftarrow F' - \{e_j\}$
22 **return** F'

Figure 7-3: A generalization of Imielinska et al.'s algorithm.

APPROX-RESTRICTED-UNCROSSABLE (V, E, c, h)

```
1   F ← ∅
2   W ← MINIMUM-COST-SPANNING-TREE(V, E, c)
3   Sort edges of W = {e_1, ..., e_{n-1}} so that c_{e_1} ≤ ··· ≤ c_{e_{n-1}}
4   While h(N) = 1 for some connected component N of F
5     i ← i + 1
6     F ← F ∪ {e_i}
7   F' ← F
8   for j ← i downto 1
9     If h(N) = 0 for all connected components N of (V, F' − {e_j})
10      F' ← F' − {e_j}
11  return F'
```

Figure 7-4: A simple version of APPROX-RESTRICTED-UNCROSSABLE.

Pick an arbitrary iteration. As in the proof of Theorem 4.1.1 for APPROX-PROPER-0-1, we construct a graph H by considering the sets in $\mathcal{C}$ and $\mathcal{I}$ of the current iteration as vertices of H, and the edges $e \in \delta_{F'}(S)$ for all $S \in \mathcal{C} \cup \mathcal{I}$ as the edges of H. We claim that for every connected component in H, there can be at most one vertex corresponding to a set in $\mathcal{I}$. Suppose there exists a connected component of H with two vertices corresponding to sets $I_1, I_2 \in \mathcal{I}$. Pick any edge e in H on the path in H between the vertices corresponding to I_1 and I_2. Notice that any edge e in H must have been added after all the edges in the connected components $S \in \mathcal{C} \cup \mathcal{I}$. Also, notice that all edges in H are in the final set of edges F'. Thus when we tested whether e should be deleted, removing e formed two connected components N_1 and N_2 with $I_1 \subseteq N_1$ and $I_2 \subseteq N_2$. But since $h(I_1) = h(I_2) = 0$ by hypothesis, it must also be the case that $h(N_1) = h(N_2) = 0$ by the properties of h, implying that e must have been deleted from F', a contradiction.

Let $d_N = |\delta_{F'}(N)|$ for a component $N \in \mathcal{C} \cup \mathcal{I}$. Thus d_N gives the degree of the vertex v in the graph H that corresponds to the component N. Let c be the number of components of H. By the claim above, $|\mathcal{I}| \leq c$. Then

$$\sum_{N \in \mathcal{C} \cup \mathcal{I}} d_N = 2(|\mathcal{C}| + |\mathcal{I}| - c) \leq 2|\mathcal{C}|,$$

since the forest has at most $|\mathcal{C}| + |\mathcal{I}| - c$ edges. ■

As with Imielinska et al.'s algorithm, we can implement this algorithm in $O(m+n\log n+n\omega_h)$ time for general graphs or $O(n\log n + n\omega_h)$ time for Euclidean graphs. Also, because we generate a dual lower bound, we can use the same techniques as in the exact partitioning problem to prove that duplicating and shortcutting edges from the lower-capacitated tree 2-approximation algorithm yields a 2-approximation algorithm for the lower-capacitated cycle problem.

7.1.2 The Classical Edge-Covering Problem

The classical edge-covering problem is that of selecting a minimum-cost set of edges such that each vertex is adjacent to at least one edge. The problem can be solved in polynomial time via a reduction to the minimum-weight perfect matching problem [57, p. 259]. The problem is also an uncrossable edge-covering problem with $h(S) = 1$ iff $|S| = 1$. It is easy to see that h is uncrossable. Also, h obeys the maximality property, and the property that $h(S) = 0$ implies $h(T) = 0$ for $T \supseteq S$. Thus Approx-Restricted-Uncrossable yields a 2-approximation algorithm for this problem. In this case, the algorithm becomes particularly easy to describe: we repeatedly choose the cheapest edge adjacent to any uncovered vertex until all vertices are covered. We then go through the edges backwards, and remove unnecessary edges. In fact, it is simple to show that the algorithm is a 2-approximation algorithm without the edge deletion stage. Note that each vertex v will select the minimum-cost edge incident to it. Call the cost of this edge c_v. Then the cost of the solution generated is at most $\sum_{v\in V} c_v$. Let F^* be the set of edges in the optimal solution. Then

$$\sum_{(u,v)\in F^*} c_{(u,v)} \geq \sum_{(u,v)\in F^*} \max(c_u, c_v) \geq \sum_{(u,v)\in F^*} \frac{c_u + c_v}{2} \geq \frac{1}{2}\sum_{v\in V} c_v.$$

7.1.3 Location-Design and Location-Routing Problems

We can apply the algorithms for non-proper edge-covering problems to solve some problems in network design and vehicle routing. Many of these problems require two levels of decisions. In the first level, the location of special vertices, such as concentrators or switches

in the design of communication networks, or depots in the routing of vehicles, needs to be decided. There is typically a set of possible locations and a fixed cost is associated with each of them. Once the locations of the depots are decided, the second level deals with the design or routing per se. These problems are called location-design or location-routing problems [78].

Several of these problems can be approximated using either Approx-Uncrossable or Approx-Restricted-Uncrossable as discussed in the previous section. We illustrate the ideas involved on one of the simplest location-routing problems. In this problem [79, 78], we need to select depots among a subset D of vertices of a graph $G = (V, E)$ and cover all vertices in V with a set of cycles, each containing a selected depot. The goal is to minimize the sum of the fixed costs of opening our depots and the sum of the costs of the edges of our cycles. In order to approximate this NP-complete problem, we consider an augmented graph $G' = (V \cup D', E')$, which we obtain from G by adding a new copy u' of every vertex $u \in D$ and adding edges of the form (u, u') for all $u \in D$. Edge (u, u') has a cost equal to half the value of the fixed cost of opening a depot at u. Consider the uncrossable edge-covering problem with the uncrossable function by $h(S) = 1$ if $\emptyset \neq S \subseteq V$ and 0 otherwise. Notice that h obeys the maximality property and the property that $h(S) = 0$ implies $h(T) = 0$ for $T \supseteq S$. Thus we can apply either Approx-Uncrossable or Approx-Restricted-Uncrossable to obtain a $(2 - \frac{1}{n})$- or 2-approximation algorithm respectively for this function h. Duplicating and shortcutting the forest obtained can be shown to result in a $(2 - \frac{1}{n})$- or 2-approximation algorithm for the original location-design problem by using analysis similar to that given for the exact cycle partitioning problem.

The same approach works also if, as in the lower constrained cycle partitioning problem, every cycle is required to have at least k vertices. In this case, $h(S) = 1$ if $\emptyset \neq S \subseteq V$ or $0 < |S \cap V| < k$, and 0 otherwise, and once again h obeys all the requisite properties to apply either Approx-Uncrossable or Approx-Restricted-Uncrossable.

7.2 Edge Duplication

So far we have considered edge-covering problems in which each edge can appear in the solution at most once: the integer program (IP) has the explicit constraint that $x_e \in \{0,1\}$ for all edges $e \in E$. Here we consider what happens if we allow an edge e to be included multiple times, up to a *multiplicity* m_e which is possibly infinite. This problem can be modelled by the following integer program, (IP'):

$$
\begin{array}{lll}
\text{Min} & \displaystyle\sum_{e \in E} c_e x_e & \\
\text{subject to:} & & \\
(IP') & \displaystyle\sum_{e \in \delta(S)} x_e \geq f(S) & S \subset V, \\
& x_e \in \{0, 1, \ldots, m_e\} & e \in E.
\end{array}
$$

Replacing an edge of multiplicity m_e by $\min\{f_{\max}, m_e\}$ copies of the edge and applying the algorithm Approx-Weakly-Supermodular yields a solution of cost within a factor of $2\mathcal{H}(f_{\max})$ of optimal. Notice, however, that when $f_{\max}$ is much larger than n, this approach has two significant problems. First, the number of phases of the algorithm is proportional to $f_{\max}$, which is not polynomial. Moreover, the performance guarantee achieved by this approach is $2\mathcal{H}(f_{\max})$, which can be rather weak. In this section, we give two methods for solving proper edge-covering problems with edge duplication which avoid some of these difficulties. The first method applies only if $m_e \geq 2f_{\max}$ for every edge e. It invokes Approx-Proper-0-1 in a sequence of $\lfloor \log f_{\max} \rfloor + 1$ phases and obtains a performance guarantee of $2\lfloor \log f_{\max} \rfloor + 2$. The second method applies to any set of values of m_e. It uses the ellipsoid algorithm for linear programming together with Approx-Weakly-Supermodular to obtain a $2\mathcal{H}(m)$-approximation algorithm.

The algorithm for the first method is given in Figure 7-5. In phase p, we set $h_p(S) = 1$ if $f(S) \geq 2^{\lfloor \log f_{\max} \rfloor + 1 - p}$ and $h_p(S) = 0$ otherwise, then call Approx-Proper-0-1. We make $2^{\lfloor \log f_{\max} \rfloor + 1 - p}$ copies of the edges of the resulting forest F' and add them to the set of edges to be output. The function h_p is proper by Observation 2.0.10. We now prove that the algorithm provides a feasible solution that is within a factor of $2\lfloor \log f_{\max} \rfloor + 2$ of optimal.

PROPER-EDGE-COVER-WITH-DUPLICATES (V, E, c, f)

1 $F_0 \leftarrow \emptyset$

2 $f_{\max} \leftarrow \max\{f(\{v\}) | v \in V\}$

3 **for** $p \leftarrow 1$ **to** $\lfloor \log f_{\max} \rfloor + 1$

4 *Comment: Phase p.*

5 $h_p(S) \leftarrow \begin{cases} 1 & \text{if } f(S) \geq 2^{\lfloor \log f_{\max} \rfloor + 1 - p} \\ 0 & \text{otherwise} \end{cases}$

6 $F' \leftarrow$ APPROX-PROPER-0-1(V, E, c, h_p)

7 Make $2^{\lfloor \log f_{\max} \rfloor + 1 - p}$ copies of each edge in F'

8 $F_p \leftarrow F_{p-1} \cup F'$

9 **return** $F_{f_{\max}}$

Figure 7-5: An approximation algorithm for proper edge-covering problems with edge duplication.

Theorem 7.2.1 Let $Z^*_{IP'}$ be the cost of an optimal solution to (IP'). Then the algorithm PROPER-EDGE-COVER-WITH-DUPLICATES produces a feasible solution for (IP') of cost no more than $(2\lfloor \log f_{\max} \rfloor + 2) Z^*_{IP'}$.

Proof: It is not too hard to see that the algorithm produces a feasible solution: for an arbitrary S such that $f(S) = \rho$, we set $h_p(S) = 1$ in phases $\lfloor \log f_{\max} \rfloor - \lfloor \log \rho \rfloor + 1$ through $\lfloor \log f_{\max} \rfloor + 1$. Hence there is at least one edge in the coboundary of S in the solutions produced by these phases, implying that the final solution has at least

$$\sum_{p=\lfloor \log f_{\max} \rfloor - \lfloor \log \rho \rfloor + 1}^{\lfloor \log f_{\max} \rfloor + 1} 2^{\lfloor \log f_{\max} \rfloor + 1 - p} = 2^{\lfloor \log \rho \rfloor + 1} - 1 \geq \rho$$

edges in the coboundary of S, as desired.

It is also not hard to see that we use at most

$$\sum_{p=1}^{\lfloor \log f_{\max} \rfloor + 1} 2^{\lfloor \log f_{\max} \rfloor + 1 - p} = 2^{\lfloor \log f_{\max} \rfloor + 1} - 1 \leq 2 f_{\max}$$

copies of any edge.

To prove the performance guarantee of the algorithm, we need to relate the cost of the edges selected in phase p to the cost of an optimal solution $\tilde{x}$ to (IP'). Let (LP_p) denote the

linear programming relaxation of the integer program given by the function h_p in phase p, and let Z_p^* be the cost of the optimal solution to (LP_p). By Theorem 4.0.1, we know that the cost of the edges selected in phase p is no more than $2Z_p^*$. Since $\frac{1}{2^{\lfloor \log f_{\max} \rfloor + 1 - p}} \tilde{x}$ is a feasible solution to (LP_p), we know that $Z_p^* \leq \frac{1}{2^{\lfloor \log f_{\max} \rfloor + 1 - p}} Z_{IP'}^*$. Since we use $2^{\lfloor \log f_{\max} \rfloor + 1 - p}$ copies of the solution of phase p, the overall cost of the solution is no more than $2(\lfloor \log f_{\max} \rfloor + 1) Z_{IP'}^*$.

■

This algorithm for decomposing a proper edge-covering problem with unlimited copies of edges into many 0-1 proper edge-covering problems is essentially the same as one given by Agrawal, Klein, and Ravi [2]. Agrawal, Klein, and Ravi show how to use their 2-approximation algorithm for the generalized Steiner tree problem to approximate the survivable network design problem given edge duplication. They achieve the performance guarantee given above. Agrawal et al. based their algorithm on an earlier algorithm of Goemans and Berstimas [52]. Goemans and Bertsimas show how to decompose a subclass of the survivable network design problem into a sequence of Steiner tree problems. If a function for this subclass assumes at most k different values $\rho_0 = 0 < \rho_1 < \cdots < \rho_k$, their algorithm applies a 2-approximation algorithm for the Steiner tree problem k times, and obtains a performance guarantee of $2 \sum_{i=1}^{p} \frac{\rho_i - \rho_{i-1}}{\rho_i}$. A approach similar to Goemans and Bertsimas can also be applied here to proper edge-covering problems, with the same results as in Goemans and Bertsimas. However, in the worst case the algorithm needs $f_{\max}$ phases and is pseudopolynomial. If $f_{\max}$ is polynomial in the input size, then the Goemans/Bertsimas approach can potentially give a better performance guarantee.

Our second method for solving proper edge-covering problems modelled by (IP') is as follows. Consider the linear programming relaxation of (IP') in which the integrality constraints on x_e are replaced by constraints $0 \leq x_e \leq m_e$. We use the ellipsoid method [57] or the convex programming algorithm of Vaidya [123] to solve this linear program. Both of these algorithms require a subroutine to solve the separation problem, as mentioned in Section 5.1; we can solve the separation problem for proper functions by using the strong oracle MAX-VIOLATED. Let x^* denote the optimal fractional solution obtained. We include $\bar{x}_e = \lfloor x_e^* \rfloor$ copies of each edge e in the solution, and then apply APPROX-WEAKLY-SUPERMODULAR to a reduced problem given by the function $f'(S) = f(S) - \bar{x}(\delta(S))$ on the

edge set $E - \{e \in E : \bar{x}_e = m_e\}$. By Lemma 2.0.5, the function f' is weakly supermodular, and by the discussion of Section 5.1, we can implement the strong oracle for this class of functions. Let x'_e be the incidence vector of the edges returned by APPROX-WEAKLY-SUPERMODULAR. It is not hard to see that $(\bar{x}_e + x'_e)$ is a feasible solution to (IP').

Theorem 7.2.2 $\sum_{e \in E} c_e(\bar{x}_e + x'_e) \leq 2\mathcal{H}(m) \sum_{e \in E} c_e x^*_e \leq 2\mathcal{H}(m) Z^*_{IP'}$.

Proof: The second inequality follows from the fact that x^*_e is an optimal solution to the linear programming relaxation of (IP'). Since x^* is feasible, for each $S \subset V$, $x^*(\delta(S)) \geq f(S)$, which implies that $f'_{\max} \leq m$. Observe that $x^* - \bar{x}$ is an optimal solution to the linear relaxation of (IP) given the weakly supermodular function f'. By Theorem 4.0.3, the cost of the solution x' is bounded by $2\mathcal{H}(f'_{\max})$ times the optimum solution to the linear relaxation. In other words, $\sum_{e \in E} c_e x'_e \leq 2\mathcal{H}(f'_{\max}) \sum_{e \in E} c_e(x^* - \bar{x})$, and the theorem follows. ■

Very recently, Aggarwal and Garg [1] have shown how to obtain a $2\mathcal{H}(\ell)$-approximation algorithm for any proper edge-covering problem with edge duplication if $m_e \geq f_{\max}$ for every edge e, where $\ell = |\{v : f(v) \geq 1\}|$. Their algorithm is based on our algorithm APPROX-PROPER-0-1, and uses an interesting scaling variation of it.

7.3 Fixed-Charge Network Design Problems

A common problem that arises in the design of telephone and traffic networks is the *fixed-charge network design problem*. In this problem, we are given a graph $G = (V, E)$, non-negative *design costs* c_e, positive capacities u_e, and non-negative *flow costs* f^i_e on each edge $e \in E$, as well as l commodities, where commodity i has a demand d_i of that must be shipped from a source node $s_i \in V$ to a sink node $t_i \in V$. The objective is to construct a network so that all demands can be satisfied, minimizing the sum of the cost of the network and the cost of the flow in the network (i.e., each unit of commodity i shipped on edge e costs f^i_e). Needless to say, this is a very difficult problem in its full generality, and many special cases have been considered; see Magnanti and Wong [90] for a survey.

Here we will show that our techniques lead to an $2f_{\max}$-approximation algorithm for a variation on the fixed-charge network design problem in which the flow costs are negligible

and the total capacity of the edges in the coboundary of a set S must be at least $f(S)$, for some weakly supermodular function f. We can model this problem by the integer program

$$
\begin{aligned}
&\text{Min} \quad \sum_{e \in E} c_e x_e \\
&\text{subject to:} \\
(FC) \qquad & \sum_{e \in \delta(S)} u_e x_e \geq f(S) && S \subset V, \\
& x_e \in \{0,1\} && e \in E.
\end{aligned}
$$

In the case that all capacities $u_e = u$ for some fixed positive integer u and the function f is proper, the problem reduces to a proper edge-covering problem with the proper function $g(S) = \lceil f(S)/u \rceil$. Applying Approx-Proper gives a $2\mathcal{H}(\lceil f_{\max}/u \rceil)$-approximation algorithm.

If the capacities are not all the same, then we apply Approx-Uncrossable in a sequence of $f_{\max}$ phases, as given in Figure 7-6. The algorithm is almost precisely the same as Approx-Weakly-Supermodular. Let F_{p-1} denote the set of edges we have selected by the end of phase $p-1$. Initially, $F_0 = \emptyset$. In phase p, we consider the remaining deficiency $\Delta_p(S) = f(S) - u(\delta_{F_{p-1}}(S))$, where we use $u(A)$ to denote $\sum_{e \in A} u_e$. As with Approx-Weakly-Supermodular, we guarantee that in phase p the deficiency $\Delta_p(S) \leq f_{\max} - p + 1$ for all sets $S \subset V$, and in this phase we add an edge to the coboundary of all sets with deficiency $\Delta_p(S) = f_{\max} - p + 1$. Thus we call Approx-Uncrossable on the graph (V, E_p), where $E_p = E - F_{p-1}$, with edge costs c, and uncrossable function

$$
h_p(S) \leftarrow \begin{cases} 1 & \text{if } \Delta_p(S) = f_{\max} - p + 1 \\ 0 & \text{otherwise.} \end{cases}
$$

The function h_p is uncrossable by Lemma 2.0.5 and Observation 2.0.1. The resulting set of edges, F', is then added to F_{p-1} to give F_p.

Theorem 7.3.1 The algorithm Approx-Fixed-Charge is a $2f_{\max}$-approximation algorithm for the fixed-charge network design problem with negligible flow costs and cut capacities given by f. If the function f is proper then each phase can be implemented to run in $O(nm' f_{\max} +$

APPROX-FIXED-CHARGE (V, E, c, u, f)

1 $F_0 \leftarrow \emptyset$

2 $f_{\max} \leftarrow f(S)$ for $S \in$ MAX-VIOLATED$(f, \emptyset)$

3 **for** $p \leftarrow 1$ **to** $f_{\max}$

4 *Comment: Phase p.*

5 $\Delta_p(S) \leftarrow f(S) - u(\delta_{F_{p-1}}(S))$ for all $S \subset V$

6 $h(S) \leftarrow \begin{cases} 1 & \text{if } \Delta_p(S) = f_{\max} - p + 1 \\ 0 & \text{otherwise} \end{cases}$

7 $E_p \leftarrow E - F_{p-1}$

8 $F' \leftarrow$ APPROX-UNCROSSABLE(V, E_p, c, h_p)

9 $F_p \leftarrow F_{p-1} \cup F'$

10 **return** $F_{f_{\max}}$

Figure 7-6: The approximation algorithm for a variation of the fixed-charge network design problem.

$n^2 m' + n^2 \omega_f)$ time.

Proof: Section 5.1 argues that we can implement the strong oracle for functions of the form $f(S) - u(\delta_{F_{p-1}}(S))$, where f is proper. By the discussion of Section 5.1.2, this leads to an implementation of a phase that runs in $O(nm' f_{\max} + n^2 m' + n^2 \omega_f)$ time.

We now prove the performance guarantee. In phase p, we incur a cost of $\sum_{e \in F'} c_e$. By Theorem 4.0.2, we know that $\sum_{e \in F_e} c_e \leq 2Z_p^*$, where Z_p^* denotes the optimum value of the integer program (IP) corresponding to the uncrossable function h_p.

Let x^* be an optimal solution to (FC), with value Z_{FC}^*. Observe that by assigning x_e^* for each $e \in E_p$, we get a feasible solution to the integer program (IP) with function h_p and edge set E_p. Hence, $Z_p^* \leq Z_{FC}^*$. Combined with the previous inequality, this shows $\sum_{e \in F'} c_e \leq 2Z_{FC}^*$. In other words, the cost incurred in every phase is bounded by twice the optimal value, which leads to a performance guarantee of $2f_{\max}$. ■

The performance guarantee of $O(f_{\max})$ is essentially tight. Let G be a graph on $k+2$ nodes s, t, and v_i for $i = 1, \ldots, k$. The edges of the graph are (s, v_i) and (v_i, t) for every i, with capacities and costs $u_{(v_i,t)} = 0$, and $c_{(v_i,t)} = k$ for every i, and $c_{(s,v_i)} = c$ and $u_{(s,v_i)} = 1$ for $i \neq k$, and $u_{(s,v_k)} = k$ and $c_{(s,v_k)} = c + \epsilon$ (see Figure 7-7). The proper function f is k for every s-t cut and 0 otherwise. As for the tight example for APPROX-WEAKLY-

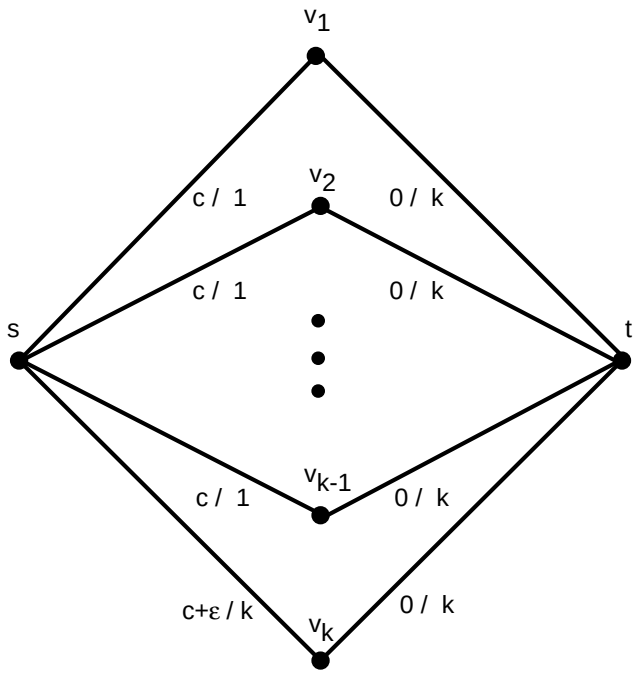

Figure 7-7: Tight example for APPROX-FIXED-CHARGE. Edges are labelled with the edge cost/edge capacity.

SUPERMODULAR in Section 4.3, each call to APPROX-UNCROSSABLE finds the minimum-cost augmenting path from s to t. Therefore, phase p selects $F_i' = \{(s, v_p), (v_p, t)\}$. This results in a solution of value $kc + \epsilon$, whereas the optimum solution is $\{(s, v_k), (v_k, t)\}$ at a cost of $c + \epsilon$.

If $f_{\max}$ becomes larger than m, then we can use a simple greedy algorithm which has performance guarantee of m. Consider the edges in order of decreasing cost, and greedily delete an edge if the remaining graph is a feasible solution. We claim that this greedy algorithm is an m-approximation algorithm. Let e be the first edge on which the greedy solution differs from the optimal solution, and let C be the cost of the edges shared by the greedy and the optimal solution. Then the cost of the optimal solution is at least $C + c_e$, whereas the cost of the greedy solution is at most $C + mc_e$.

Using techniques analogous to the $2\mathcal{H}(m)$-approximation algorithm for edge duplication in Section 7.2, the guarantees of Theorem 7.3.1 extend to a further generalization of the fixed-charge model when an edge e can be included up to m_e times. This problem is modelled

by replacing the constraint $x_e \in \{0,1\}$ in the integer program (FC) by the constraint $x_e \in \{0,1,\ldots,m_e\}$.

Our algorithm for the fixed-charge network design problem can be used to solve a network reinforcement problem recently investigated by Bienstock and Diaz [15]. Given a graph $G = (V,E)$, edge costs c_e, and a weakly supermodular function f, one must find a minimum-cost set of edges such that the graph induced by contracting these edges satisfies f. In other words, the set must include one edge from the coboundary of every set S such that $|\delta(S)| < f(S)$. By using the function $f'(S) = f(S) - |\delta(S)|$ and setting $u_e = f'_{\max}$ for all e, we can apply the algorithm above.

In practice, most fixed-charge network design problems are specified in terms of multicommodity flow, as we stated initially, rather than by specifying a function f on the cuts of the graph. Given a fixed-charge network design problem in which the flow costs are negligible, we observe that in order to be able to ship the commodities, we must construct a network such that $f(S) \geq \sum_{i:|S\cap\{s_i,t_i\}|=1} d_i$. When there is a single commodity, it is known that $f_1(S) = \sum_{i:|S\cap\{s_i,t_i\}|=1} d_i$ is sufficient to be able to ship the commodity, by the max-flow min-cut theorem [38, 35]. Thus the algorithm above leads to a $2f_{\max}$-approximation algorithm in the single commodity case, since f_1 is proper. When there are $l > 1$ commodities, $f_l(S) = c\log^2 l \sum_{i:|S\cap\{s_i,t_i\}|=1} d_i$ for some constant c is known to be sufficient to ship all commodities by recent results in multicommodity flow theory [49, 103]. Unfortunately, since this amount of capacity may not be necessary, it does not translate into a straightforward $O(f_{\max}\log^2 l)$-approximation algorithm for the multicommodity fixed-charge network design problem with negligible flow costs.

7.4 The Prize-Collecting Problems

The prize-collecting traveling salesman problem is a variation of the classical traveling salesman problem (TSP). In addition to the cost on the edges, we have also a penalty π_i on each vertex i. The goal is to find a tour on a subset of the vertices that minimizes the sum of the cost of the edges in the tour and the vertices not in the tour. We consider a variant in which a prespecified root vertex r has to be in the tour; this is without loss

of generality, since we can repeat the algorithm n times, setting each vertex to be the root. This version of the prize-collecting TSP is a special case of a more general problem introduced by Balas [9]. The prize-collecting Steiner tree problem is defined analogously. The standard Steiner tree problem can be seen to be a special case of the prize-collecting Steiner tree problem in which non-terminals have a penalty of zero, while terminals have a very large penalty (e.g., equal to the diameter of the graph).

Bienstock, Goemans, Simchi-Levi and Williamson [16] developed the first approximation algorithms for these problems. The performance guarantees of their algorithms are 5/2 for the TSP (assuming the triangle inequality) and 3 for the Steiner tree problem. These approximation algorithms are not very efficient, however, since they are based upon the exact solution of a linear programming problem.

The prize-collecting problems cannot be modelled by the integer program (IP). However, the techniques used in Approx-Uncrossable and Approx-Proper-0-1 can be modified to give a $(2-\frac{1}{n-1})$-approximation algorithm for both the prize-collecting TSP (under the triangle inequality) and the prize-collecting Steiner tree problem. Moreover, these algorithms are purely combinatorial and do not require the solution of a linear programming problem as in Bienstock et al. [16]. We will focus our attention on the prize-collecting Steiner tree problem, and at the end of the section we will show how the algorithm for the tree problem can be easily modified to yield a prize-collecting TSP algorithm.

7.4.1 The Prize-Collecting Steiner Tree

The prize-collecting Steiner tree can be formulated as the following integer program:

$$\text{Min} \quad \sum_{e \in E} c_e x_e + \sum_{T \subset V; r \notin T} z_T \left(\sum_{i \in T} \pi_i \right)$$

subject to:

$$(PC\text{-}IP) \qquad x(\delta(S)) + \sum_{T \supseteq S} z_T \geq 1 \qquad S \subset V; r \notin S$$

$$\sum_{T \subset V; r \notin T} z_T \leq 1$$

$$x_e \in \{0,1\} \qquad e \in E$$

$$z_T \in \{0,1\} \qquad T \subset V; r \notin T$$

Intuitively, z_T is set to 0 for all T except the set T of all vertices not spanned by the tree of selected edges. A linear programming relaxation (PC-LP) of the integer program can be created by replacing the integrality constraints with the constraints $x_e \geq 0$ and $z_T \geq 0$ and dropping the constraint $\sum_T z_T \leq 1$. In fact, including this constraint does not affect the optimal solution. Suppose there is some problem instance in which every optimal solution that minimizes $\sum_T z_T$ has $\sum_T z_T > 1$. Notice that if $z_A \geq z_B > 0$ and $B \not\subseteq A$, then we can create an equivalent solution by setting $z_{A \cup B} \leftarrow (z_{A \cup B} + z_B)$, $z_A \leftarrow (z_A - z_B)$, and $z_B \leftarrow 0$. Continuing this process eventually yields a solution in which $z_A, z_B > 0$ implies that either $A \subseteq B$ or $B \subseteq A$. Thus there is some smallest set C with $z_C > 0$ such that $z_A > 0$ implies $A \supseteq C$. If $\sum_{T \supseteq C} z_T > 1$, we can decrease z_C without affecting the feasibility of the solution, contradicting the minimality of $\sum_T z_T$.

The LP relaxation (PC-LP) can be shown to be equivalent to the following, perhaps more natural, linear programming relaxation of the prize-collecting Steiner tree problem, which was used by the algorithm of Bienstock et al. [16]:

$$\text{Min} \quad \sum_{e \in E} c_e x_e + \sum_{i \neq r} (1 - y_i)\pi_i$$

subject to:

$$\begin{aligned} x(\delta(S)) &\geq y_i && i \in S; r \notin S \\ x_e &\geq 0 && e \in E \\ y_i &\geq 0 && i \in V; i \neq r. \end{aligned}$$

Given a feasible solution to (PC-LP) with the z variables such that $z_A, z_B > 0$ implies that either $A \subseteq B$ or $B \subseteq A$, we can construct a feasible solution of the same cost by setting $y_i = 1 - \sum_{i \in T} z_T$ and leaving x untouched. Similarly, given a feasible solution to the linear program above with $y_{i_1} \leq y_{i_2} \leq \cdots \leq y_{i_n}$, we can construct a feasible solution to (PC-LP) of the same cost by setting $z_{\{i_1\}} = y_{i_2} - y_{i_1}$, $z_{\{i_1, i_2\}} = y_{i_3} - y_{i_2}$, ..., $z_V = 1 - y_{i_n}$.

The dual of (PC-LP) can be formulated as follows:

$$\text{Max} \quad \sum_{S : r \notin S} y_S$$

subject to:

$$(PC\text{-}D) \qquad \begin{aligned} \sum_{S:e\in\delta(S)} y_S &\le c_e && e \in E \\ \sum_{S\subseteq T} y_S &\le \sum_{i\in T} \pi_i && T \subset V; r \notin T \\ y_S &\ge 0 && S \subset V; r \notin S. \end{aligned}$$

The algorithm for the prize-collecting Steiner tree problem is shown in Figure 7-8. The basic structure of this algorithm is similar to that of Approx-Proper-0-1. The algorithm maintains a forest F of edges, which is initially empty. Hence each vertex v is initially in its own connected component. Unlike the previous algorithms we have looked at, here we have no notion of a "violated" set; the sets that are violated depend on which vertices we choose to include in the Steiner tree. Hence we are going to assign activity or inactivity to particular sets as the algorithm progresses. Initially, all components except the root r are considered active.

The algorithm loops, in each iteration doing one of two things. First, the algorithm may add an edge between two connected components of F. If the resulting component contains the root r, it becomes inactive; otherwise it is active. Second, the algorithm may decide to "deactivate" a component. Intuitively, a component is deactivated if the algorithm decides it is willing to pay the penalties for all vertices in the component. In this case, the algorithm labels each vertex in the component with the name of the component. The main loop terminates when all connected components of F are inactive. Since in each iteration the sum of the number of components and the number of active sets decreases, the loop terminates after at most $2n - 1$ iterations. The final step of the algorithm removes as many edges from F as possible while maintaining two properties. First, all unlabelled vertices must be connected to the root, since these vertices were never in any deactivated component and the algorithm was never willing to pay the penalty for these vertices. Second, if a vertex with label C is connected to the root, then so is every vertex with label $C' \supseteq C$.

As before, the choices of this algorithm are motivated by primal-dual method. We implicitly construct a solution to the dual $(PC\text{-}D)$. Initially all dual variables are set to zero and the set of selected edges F is empty. Let T denote the set of vertices for which we are willing to pay a penalty; T is also empty initially. In each iteration of the main loop,

APPROX-PC-STEINER (V, E, c, π, r)

$F \leftarrow \emptyset$; $T \leftarrow \emptyset$

Comment: Implicitly set $y_S \leftarrow 0$ *for all* $S \subset V$

$\mathcal{C} \leftarrow \{\{v\} : v \in V, v \neq r\}$

$\mathcal{I} \leftarrow \{\{r\}\}$

For each $v \in V$

  Unmark v; $d(v) \leftarrow 0$; $w(\{v\}) \leftarrow 0$

  If $v = r$ then $a(r) \leftarrow 0$ else $a(v) \leftarrow 1$

While $|\mathcal{C}| > 0$

  Find edge $e = (u, v)$ with $u \in C_p \in \mathcal{C}$, $v \in C_q \in \mathcal{C} \cup \mathcal{I}$, $C_p \neq C_q$ that minimizes $\epsilon_1 = \frac{c_e - d(u) - d(v)}{a(u) + a(v)}$

  Find $\tilde{C} \in \mathcal{C}$ that minimizes $\epsilon_2 = \sum_{i \in \tilde{C}} \pi_i - w(\tilde{C})$

  $\epsilon = \min(\epsilon_1, \epsilon_2)$

  $w(C) \leftarrow w(C) + \epsilon$ for all $C \in \mathcal{C}$

  Comment: Implicitly set $y_C \leftarrow y_C + \epsilon$ *for all* $C \in \mathcal{C}$

  For all $v \in C_r \in \mathcal{C}$ do $d(v) \leftarrow d(v) + \epsilon$

  If $\epsilon = \epsilon_2$

    $\mathcal{C} \leftarrow \mathcal{C} - \{\tilde{C}\}$; $\mathcal{I} \leftarrow \mathcal{I} \cup \{\tilde{C}\}$; $T \leftarrow T \cup \tilde{C}$

    $a(v) \leftarrow 0$ for $v \in \tilde{C}$

    Mark all unlabelled vertices of $\tilde{C}$ with label $\tilde{C}$

  else

    $F \leftarrow F \cup \{e\}$

    Delete C_p and C_q from $\mathcal{C}$ and $\mathcal{I}$

    $w(C_p \cup C_q) \leftarrow w(C_p) + w(C_q)$

    If $r \in C_p \cup C_q$

      $\mathcal{I} \leftarrow \mathcal{I} \cup \{C_p \cup C_q\}$

      $a(v) \leftarrow 0$ for all $v \in C_p \cup C_q$

    else

      $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_p \cup C_q\}$

      $a(v) \leftarrow 1$ for all $v \in C_p \cup C_q$

F' is derived from F by removing as many edges as possible but so that the following two properties hold: (1) every unlabelled vertex is connected to r; (2) if vertex v with label C is connected to r, then so is every vertex with label $C' \supseteq C$. In this case, delete all vertices in these C' from T to yield T'.

return F', T'

Figure 7-8: The algorithm for the Prize-Collecting Steiner Tree Problem.

the algorithm performs a dual improvement step by increasing y_C for all active sets C by a value ϵ which is as large as possible without violating the two types of packing constraints of (*PC-D*): $\sum_{S:e\in\delta(S)} y_S \leq c_e$ for all $e \in E$, and $\sum_{S\subseteq T} y_S \leq \sum_{i\in T} \pi_i$ for all $T \subset V$. Increasing the y_C for active sets C by ϵ will cause one of the packing constraints to become tight. If one of the first kind of constraints becomes tight, then it becomes tight for some edge e between two connected components of the current forest F; hence we add this edge to F, and improve the feasibility of the primal solution. Now suppose one of the second kind of constraints becomes tight for some set T. The constraint must have become tight due to an increase in y_{C_i} for some active sets $C_i \subseteq T$. Thus it must also be the case that the second kind of constraint is also tight for some active $C = C_i$; it cannot be the case that $\sum_{S\subseteq C_i} y_S < \sum_{j\in C_i} \pi_j$ for all active $C_i \subseteq T$. In this case, the algorithm chooses to deactivate C, and we add the vertices of C to T, also improving the feasibility of the primal solution. When there are no active sets remaining, the solution $x_e = 1$ for all $e \in F$ and $z_T = 1$ is feasible for (*PC-IP*).

As with our other algorithms, we must modify the primal solution in order to get a good performance guarantee. Intuitively, we would like to ensure that r is connected to all vertices in $V - T$, and delete all other edges. In doing so, we might use edges that connect r to vertices in T, say to a vertex in component C deactivated at some point during the algorithm. Including these edges satisfy the primal constraints for all deactivated $C' \supseteq C$, so we reduce T by removing all the vertices in these C' from T. Reducing T may, in turn, require us to keep more edges connecting vertices in $V - T$ to the root r, possibly allowing us to reduce T further. Thus we remove as many edges as possible to ensure that all vertices in the original set $V - T$ are connected to r, and that if a vertex in some deactivated C is connected to r, then so are all vertices in deactivated $C' \supseteq C$.

We claim that the algorithm shown in Figure 7-8 behaves exactly in the manner described above. The claim follows straightforwardly from the algorithm's construction of y and F, and from the fact that $d(i) = \sum_{S:i\in S} y_S$ and $w(C) = \sum_{S\subseteq C} y_S$ at the beginning of each iteration.

We can now prove the correctness of the algorithm in a manner similar to that of the previous algorithms.

Theorem 7.4.1 Let Z^*_{PCIP} be the value of an optimal solution to the integer program $(PC\text{-}IP)$. Then APPROX-PC-STEINER produces a set of edges F', a set of vertices T', and a feasible solution y to $(PC\text{-}D)$ such that

$$\sum_{e \in F'} c_e + \sum_{i \in T'} \pi_i \leq \left(2 - \frac{1}{n-1}\right) \sum_{S \subset V} y_S \leq \left(2 - \frac{1}{n-1}\right) Z^*_{PCIP}.$$

Proof: It is not hard to see that the algorithm produces a feasible solution to $(PC\text{-}IP)$, since F' has no non-trivial component not containing r and the component containing r is a tree.

By the construction of F', each vertex not spanned by F' (i.e., the vertices in T') lies in some component deactivated at some point during the algorithm. Furthermore, if the vertex was in some deactivated component C, then none of the vertices of C are spanned by F'. Using these observations, plus the manner in which components are formed by the algorithm, we can partition the vertices of T' into disjoint deactivated components $C_1, \ldots, C_k$. These sets are the maximal labels of the vertices in T'. Since each C_j is a deactivated component, it follows that $\sum_{S \subseteq C_j} y_S = \sum_{i \in C_j} \pi_i$, and thus that the inequality to be proven is equivalent to $\sum_{e \in F'} c_e + \sum_j \sum_{S \subseteq C_j} y_S \leq (2 - \frac{1}{n-1}) \sum_{S \subset V} y_S$. In addition, since $c_e = \sum_{S : e \in \delta(S)} y_S$ for each $e \in F'$ by construction of the algorithm, all we need to prove is that

$$\sum_{e \in F'} \sum_{S : e \in \delta(S)} y_S + \sum_j \sum_{S \subseteq C_j} y_S \leq \left(2 - \frac{1}{n-1}\right) \sum_{S \subset V} y_S,$$

or, rewriting terms,

$$\sum_S y_S |\delta_{F'}(S)| + \sum_j \sum_{S \subseteq C_j} y_S \leq \left(2 - \frac{1}{n-1}\right) \sum_{S \subset V} y_S.$$

Using the argument of Section 4.1, we can prove this inequality by showing that at every iteration, the following "total degree" inequality holds:

$$\sum_{S \in \mathcal{C}} |\delta_{F'}(S)| + \sum_j |\{S \in \mathcal{C} : S \subseteq C_j\}| \leq 2|\mathcal{C}| - 1,$$

where $\mathcal{C}$ is the set of all active sets in the current iteration. Since $|\mathcal{C}| \leq n - 1$, the previous

inequality follows. We prove the total-degree inequality by induction on the main loop.

Pick any particular iteration. Let $\mathcal{I}'$ be the inactive components N such that $\delta_{F'}(N) \neq \emptyset$, and let $\mathcal{C}_1$ be the active sets contained in some C_j. Let H be the graph formed by considering the components in $\mathcal{C} \cup \mathcal{I}'$ as vertices and the edges $e \in \delta_{F'}(C)$ for active C as the edges of H. Let $d_N = |\delta_{F'}(N)|$ for any component $N \in \mathcal{C} \cup \mathcal{I}'$. To prove the inequality, we would like to show that

$$\sum_{N \in \mathcal{C}} d_N + |\mathcal{C}_1| \leq 2|\mathcal{C}| - 1.$$

Notice that for all components $N \in \mathcal{C}_1$, $d_N = 0$. Hence if we can show that

$$\sum_{N \in \mathcal{C} - \mathcal{C}_1} d_N \leq 2(|\mathcal{C}| - |\mathcal{C}_1|) - 1,$$

then the proof will be complete.

To do this, we show that all but one of the leaves of H must be correspond to active sets. Suppose that v is an inactive leaf of H, adjacent to edge e, and let I_v be the inactive component corresponding to v. Further suppose that I_v does not contain the root r. Since I_v is inactive and does not contain r, it must have been deactivated. Because I_v is deactivated, no vertex in I_v is unlabelled; furthermore, since v is a leaf, no vertex in I_v can lie on the path between the root and a vertex which must be connected to the root. By the construction of F', then, $e \notin F'$, which is a contradiction. Therefore, there can be at most one inactive leaf in H, which must correspond to the component containing r.

Then

$$\begin{aligned}
\sum_{N \in \mathcal{C} - \mathcal{C}_1} d_N &\leq \sum_{N \in (\mathcal{C} - \mathcal{C}_1) \cup \mathcal{I}'} d_N - \sum_{N \in \mathcal{I}'} d_N \\
&\leq 2(|\mathcal{C}| - |\mathcal{C}_1| + |\mathcal{I}'| - 1) - (2|\mathcal{I}'| - 1) \\
&= 2(|\mathcal{C}| - |\mathcal{C}_1|) - 1
\end{aligned}$$

The inequality holds since all but one of the components in $\mathcal{I}'$ has degree at least two. ■

The algorithm can be implemented in $O(n(n + \sqrt{m \log\log n}))$ time, by using the edge-selection algorithm given for APPROX-PROPER-0-1 and an $O(n^2)$ time edge deletion step.

APPROX-PC-TSP (V, E, c, π, r)

1 $F', T' \leftarrow$ APPROX-PC-STEINER $(V, E, c, \frac{\pi}{2}, r)$

2 Duplicate the edges F' of the Steiner tree returned to form an Eulerian graph R.

3 Shortcut R to form a tour R'.

4 **return** R', T'

Figure 7-9: The algorithm for the Prize-Collecting Traveling Salesman Problem.

Recall that for APPROX-PROPER-0-1 we kept track of an "addition time" for each edge in a priority queue in order to select the edge minimizing the reduced cost ϵ in each iteration. For this algorithm, we must also keep track of the time at which we expect each component to deactivate. We keep a separate priority queue for these deactivation times, and in each iteration of the main loop we select the edge or deactivate the component having the minimum addition time or deactivation time respectively. Each iteration of the main loop requires at most $O(1)$ changes to the elements of the deactivation time priority queue, leading to an overall time bound of $O(n \log n)$ for maintaining this queue. To implement the edge deletion step in $O(n^2)$ time, we first perform a depth-first search from every unmarked vertex to the root, and "lock" all the edges and vertices on this path. We then look at all the deactivated components corresponding to the labels of "locked" vertices or supersets of these deactivated components. If one of these contains an unlocked vertex, we perform a depth-first search from the vertex to the root and lock all the edges and vertices on the path. We continue this process until each locked vertex is in a deactivated component that contains only locked vertices and whose supersets contain only locked vertices. We then eliminate all unlocked edges. This procedure requires at most n $O(n)$ time depth-first searches.

7.4.2 The Prize-Collecting Traveling Salesman Problem

In order to solve the prize-collecting TSP given that edge costs obey the triangle inequality, we use the algorithm shown in Figure 7-9. Note that the algorithm uses the above algorithm for the prize-collecting Steiner tree problem with penalties $\pi'_i = \pi_i/2$. To see that the algorithm is a $(2 - \frac{1}{n-1})$-approximation algorithm, we need to consider the following linear programming relaxation of the problem:

$$\text{Min} \quad \sum_{e \in E} c_e x_e + \sum_{T \subset V; r \notin T} z_T \left(\sum_{i \in T} \pi_i \right)$$

subject to:

$$x(\delta(S)) + 2 \sum_{T \supseteq S} z_T \geq 2 \qquad r \notin S$$

$$x_e \geq 0 \qquad e \in E$$

$$z_T \geq 0 \qquad T \subset V; r \notin T.$$

This linear program is a relaxation of an integer program similar to $(PC\text{-}IP)$ in which $z_T = 1$ for the set of vertices T not visited by the tour, and $z_T = 0$ otherwise. We relax the constraint that each vertex in the tour be visited twice to the constraint that each vertex be visited at least twice. The dual of the linear programming relaxation is

$$\text{Max} \quad 2 \sum_{S: r \notin S} y_S$$

subject to:

$$\sum_{S: e \in \delta(S)} y_S \leq c_e \qquad e \in E$$

$$2 \sum_{S \subseteq T} y_S \leq \sum_{i \in T} \pi_i \qquad T \subset V, r \notin T$$

$$y_S \geq 0 \qquad S \subset V, r \notin S.$$

Notice that this dual is very similar to $(PC\text{-}D)$. The dual solution generated by the algorithm for the prize-collecting Steiner tree for penalties π' will be feasible for the dual program above with penalties π. By duality, $2 \sum_{S \subset V} y_S \leq Z^*_{PCTSP}$, where Z^*_{PCTSP} is the cost of the optimal solution to the prize-collecting TSP. Given a solution F' and T' to the prize-collecting Steiner tree problem, the cost of our solution to the prize-collecting TSP is at most $2 \sum_{e \in F'} c_e + \sum_{i \in T'} \pi_i = 2(\sum_{e \in F'} c_e + \sum_{i \in T'} \pi'_i)$. Theorem 7.4.1 shows that

$\sum_{e \in F'} c_e + \sum_{i \in T'} \pi'_i \leq (2 - \frac{1}{n-1}) \sum_{S \subset V} y_S$, so that

$$2(\sum_{e \in F'} c_e + \sum_{i \in T'} \pi'_i) \leq 2\left(2 - \frac{1}{n-1}\right) \sum_{S \subset V} y_S \leq \left(2 - \frac{1}{n-1}\right) Z^*_{PCTSP}.$$

Thus the cost of the solution found by the algorithm is within $(2 - \frac{1}{n-1})$ of optimal.

7.5 The k-Vertex-Connected Subgraph Problem

So far the problems we have considered have had constraints requiring at least a certain number of edges in the coboundary of each set. We can, however, use the primal-dual methodology to solve problems requiring sets to have at least a certain number of neighboring vertices. In particular, our technique extends to solve a basic problem of this kind, the minimum-cost k-vertex-connected subgraph problem. In this problem, we must find the minimum-cost set of edges such that there are at least k vertex-disjoint paths between every pair of vertices. The minimum-cost 2-vertex-connectivity problem is NP-hard [36], and the only approximation algorithm known is a 3-approximation algorithm for the 2-vertex-connectivity problem due to Khuller and Thurimella [70]. By modifying Approx-Weakly-Supermodular and Approx-Uncrossable, we can obtain a $2\mathcal{H}(k)$-approximation algorithm for the minimum-cost k-vertex-connected subgraph problem for any k.

Menger [92] has shown that a graph is k-vertex connected if and only if it has at least $k+1$ vertices and there is no vertex set $T \subset V$ with $|T| \leq k-1$ such that the graph induced by removing T is disconnected. Define $\delta(A : B)$ to be $\delta(A) \cap \delta(B)$. The latter condition is equivalent to saying that for all $|T| \leq k-1$, for all $S \subset V - T$, $\delta(S : V - T - S) \neq \emptyset$. Thus the minimum-cost k-vertex-connected subgraph problem can be modelled by the following integer program:

$$\begin{aligned}
&\text{Min} && \sum_{e \in E} c_e x_e && \\
&\text{subject to:} && && \\
(kVC) \qquad & && \sum_{e \in \delta(S : V-T-S)} x_e \geq 1 && T \subset V,\ |T| \leq k-1,\ S \subset V - T \\
& && x_e \text{ integer} && e \in E.
\end{aligned}$$

We do not need the constraint $x_e \in \{0,1\}$ in this case, since for any feasible solution x to (kVC) the solution $x'_e = 1$ if $x_e > 0$ and $x'_e = 0$ otherwise is also feasible and of no greater cost. The dual of the linear relaxation of (kVC) is

$$
\begin{aligned}
&\text{Max} \quad \sum_{S,T} y_{S,T} \\
&\text{subject to:} \\
(DVC) \qquad & \sum_{S,T:e\in\delta(S:V-T-S)} y_{S,T} \le c_e \qquad e \in E, \\
& y_{S,T} \ge 0 \qquad\qquad\qquad\qquad T \subset V,\ |T| \le k-1,\ \ S \subset V-T.
\end{aligned}
$$

As in Approx-Weakly-Supermodular, we will augment our solution in phases, starting with an empty set of edges. Our strategy here will be to select edges so that by the end of phase p we have an edge set that is p-vertex-connected. The edge set at the end of phase p will be denoted F_p. As in Approx-Weakly-Supermodular we will augment F_{p-1} to F_p by adding a set of edges F' to the coboundaries of "violated" sets; we select a set of edges F using the primal-dual method, then remove redundant edges to obtain F'.

To define a violated set in this case, we first need some notation. Let $\Gamma_A(S)$ be the vertex neighborhood of S given the set of edges A: that is, the vertices not in S that are adjacent to some vertex in S via an edge in A. We also define the "vertex complement" $\zeta_A(S)$ of S to be $V - S - \Gamma_A(S)$. Let F be the set of edges selected so far in phase p. Unless we specify otherwise, the edge set under consideration for both Γ and ζ will always be $F_{p-1} \cup F$, and thus for notational simplicity we drop the subscript. Then a *violated set* S is one such that $|\Gamma(S)| \le p-1$ and $\zeta(S) \neq \emptyset$. By Menger's Theorem, the set of edges $F_{p-1} \cup F$ is p-vertex-connected if and only if there are no violated sets. Because F_{p-1} is $(p-1)$-vertex-connected, $|\Gamma(S)| = p-1$ for any violated set S.

Instead of looking at all violated sets, we will instead consider only *small* violated sets. Call a set S in phase p small if $|S| \le \lfloor \frac{n-(p-1)}{2} \rfloor$. If there are no small violated sets, then we claim that there will be no violated sets, since for any non-small violated set S, the set $\zeta(S)$ is a small violated set. We can now prove that the small violated sets behave much like the

violated sets of the APPROX-UNCROSSABLE algorithm. We define the *active sets* to be the minimal small violated sets with respect to the edge set $F_{p-1} \cup F$. As before, we can show that in any particular phase p, the active sets are disjoint. To do this we need the fact that $\Gamma_Y(S)$ is submodular for any edge set Y. That is, for any edge set Y and any two sets of vertices A and B, $|\Gamma_Y(A)| + |\Gamma_Y(B)| \geq |\Gamma_Y(A \cup B)| + |\Gamma_Y(A \cap B)|$. We also observe that $\Gamma_Y(A \cap B) - A - B \subseteq \Gamma_Y(A) \cap \Gamma_Y(B)$.

Lemma 7.5.1 If A and B are crossing small violated sets with respect to the edge set $F_{p-1} \cup F$, then $A \cap B$ is a small violated set and either $A \cup B$ or $\zeta(A \cup B)$ is a small violated set.

Proof: Since A and B are small-violated, $|\Gamma(A)| = |\Gamma(B)| = p - 1$. Because A and B are small-violated and cross, we know that $|A \cup B| \leq n - (p-1) - 1$. We would like to show that $\zeta(A \cap B) \neq \emptyset$. Using the relation that $\Gamma(A \cap B) - A - B \subseteq \Gamma(A) \cap \Gamma(B)$, we see that $|(A \cap B) \cup \Gamma(A \cap B)| \leq |A \cup B \cup (\Gamma(A) \cap \Gamma(B))| \leq n - 1$, implying that $\zeta(A \cap B) \neq \emptyset$. Thus by the feasibility of F_{p-1}, $|\Gamma(A \cap B)| \geq p - 1$. By the submodularity of $|\Gamma(S)|$, it follows that $|\Gamma(A \cup B)| \leq p - 1$. Then $|A \cup B \cup \Gamma(A \cup B)| \leq n - 1$, or $\zeta(A \cup B) \neq \emptyset$. Hence the feasibility of F_{p-1} implies $|\Gamma(A \cup B)| \geq p - 1$. Then it follows by submodularity that $|\Gamma(A \cup B)| = |\Gamma(A \cap B)| = p - 1$, and $A \cup B$ and $A \cap B$ are both violated. $A \cap B$ is certainly small-violated, and either $A \cup B$ or $\zeta(A \cup B)$ must be small-violated. ■

Theorem 7.5.2 The minimal small violated sets with respect to the edge set $F_{p-1} \cup F$ are disjoint.

Proof: Suppose there are two active sets A and B which cross. Then by the lemma above, $A \cap B$ is also small-violated, which contradicts the minimality of A and B. ■

In order to find the minimal small violated sets in phase p, we assume the existence of an oracle SMALL-VIOLATED$(p, F_{p-1} \cup F)$ that returns all the active sets (the minimal small violated sets) with respect to the edge set $F_{p-1} \cup F$. Later we will show how to implement the oracle SMALL-VIOLATED in polynomial time.

The overall algorithm is given in Figure 7-10. The algorithm for each phase is similar to APPROX-UNCROSSABLE. In each phase, we select a set of edges F and construct a dual solution for (DVC). Initially F is empty and $y_{S,T} = 0$ for all S, T. The dual

improvement step increases dual variables $y_{C,\Gamma(C)}$ for all active sets C uniformly until $\sum_{S,T:e\in\delta(S:V-T-S)} y_{S,T} = c_e$ for some edge e. In particular, the constraint will become tight for some edge $e \in \delta(C : \zeta(C))$ of some active set C; we then add this edge e to F and continue until there are no small violated (and hence no violated) sets left for the phase. Notice that an increase in a dual variable $y_{C,\Gamma(C)}$ does not increase the sums in the dual constraints for *any* edge from vertices in C to vertices in $\Gamma(C)$. In particular, the sums are not increased for any edge already in $F_{p-1} \cup F$, so an edge from F_{p-1} will never be selected, and once an edge in F is selected, $c_e = \sum_{S,T:e\in\delta(S:V-T-S)} y_{S,T}$ for the rest of the phase. Once there are no small violated sets left, we go through the edges of F in reverse order, removing all edges that do not affect the feasibility of the solution for the pth phase, as in the REGULAR-EDGE-DELETE step of APPROX-UNCROSSABLE.

We now begin to prove that the following theorem, where Z^*_{kVC} is the value of an optimal solution to (kVC).

Theorem 7.5.3 The algorithm APPROX-k-VERTEX-CONN produces a k-vertex-connected set of edges F_k such that

$$\sum_{e\in F_k} c_e \leq 2\mathcal{H}(k) Z^*_{kVC}.$$

Let x^* be the optimal solution to (kVC) of value Z^*_{kVC}. Notice that in phase p, we only increase dual variables for sets $|T| = p-1$. In effect, we provide a dual solution to the linear program

$$\begin{aligned}
&\text{Min} \quad \sum_{e\in E} c_e x_e \\
&\text{subject to:} \\
(kVC_p) \qquad &\sum_{e\in\delta(S:V-T-S)} x_e \geq 1 \qquad T \subset V, |T| = p-1, S \subset V-T \\
&x_e \geq 0 \qquad e \in E.
\end{aligned}$$

Then $\frac{1}{k-p+1}x^*$ is a feasible solution to this linear program: for any sets S, T such that $|T| = p-1$ and $S \subset V-T$, it must be the case that x^* has at least $k-(p-1)$ edges in the cut from S to $V-T-S$. Thus $\frac{1}{k-p+1}x^*(\delta(S : V-T-S)) \geq 1$, and $\sum_{S,T} y_{S,T} \leq \frac{1}{k-p+1}Z^*_{kVC}$ for the feasible dual solution constructed in phase p. If we can show that in

APPROX-k-VERTEX-CONN (V, E, k)

1 $F_0 \leftarrow \emptyset$
2 **for** $p \leftarrow 1$ **to** k
3 *Comment: Begin phase p*
4 $F \leftarrow \emptyset$
5 *Comment: Implicitly set* $y_{S,T} \leftarrow 0$ *for all* $S, T \subset V$
6 $i \leftarrow 0$
7 $d(e) \leftarrow 0$ for all $e \in E - F_{p-1}$
8 $\mathcal{C} \leftarrow$ SMALL-VIOLATED(p, F_{p-1})
9 $a(e) \leftarrow \begin{cases} 2 & \text{if } e \in \delta(C_1 : \zeta(C_1)) \cap \delta(C_2 : \zeta(C_2)) \text{ for } C_1, C_2 \in \mathcal{C}, C_1 \neq C_2 \\ 0 & \text{if } e \notin \delta(C : \zeta(C)) \text{ for any } C \in \mathcal{C} \\ 1 & \text{otherwise} \end{cases}$
10 **while** $|\mathcal{C}| > 0$
11 $i \leftarrow i + 1$
12 *Comment: Begin iteration i*
13 Find edge $e_i = (u, v)$, $a(e_i) \neq 0$, that minimizes $\epsilon = \frac{c_e - d(e)}{a(e)}$
14 For all $e \in E - F_{p-1}$
15 $d(e) \leftarrow d(e) + a(e) \cdot \epsilon$
16 *Comment: Implicitly set* $y_{C,\Gamma(C)} \leftarrow y_{C,\Gamma(C)} + \epsilon$ *for all* $C \in \mathcal{C}$
17 $F \leftarrow F \cup \{e_i\}$
18 $\mathcal{C} \leftarrow$ SMALL-VIOLATED$(p, F_{p-1} \cup F)$
19 Update $a(e)$
20 *Comment: End iteration i*
21 *Comment: Edge deletion stage*
22 $F' \leftarrow F$
23 **for** $j \leftarrow i$ **downto** 1
24 If $F_{p-1} \cup F' - \{e_j\}$ is p-vertex-connected
25 $F' \leftarrow F' - \{e_j\}$
26 $F_p \leftarrow F' \cup F_{p-1}$
27 **return** F_k

Figure 7-10: The algorithm for k-vertex-connectivity.

phase p, $\sum_{e \in F'} c_e \leq 2 \sum_{S,T} y_{S,T}$, then the cost of the overall set of edges F_k is no more than $\sum_{p=1}^{k} \frac{2}{k-p+1} Z^*_{kVC} = 2\mathcal{H}(k) Z^*_{kVC}$, giving the desired performance guarantee.

Now we must show that $\sum_{e \in F'} c_e \leq 2 \sum_{S,T} y_{S,T}$ in phase p. By rewriting $\sum c_e$ we get

$$\sum_{e \in F'} \sum_{S,T : e \in \delta(S : V-T-S)} y_{S,T} \leq 2 \sum_{S,T} y_{S,T}.$$

Rewriting again gives

$$\sum_{S,T} y_{S,T} \cdot |\delta_{F'}(S : V - T - S)| \leq 2 \sum_{S,T} y_{S,T}.$$

To prove this statement, we use the techniques from Section 4.1 to reduce the inequality to proving that at each iteration,

$$\sum_{C \in \mathcal{C}} |\delta_{F'}(C : \zeta(C))| \leq 2|\mathcal{C}|.$$

To prove this, we essentially use the proof of the performance guarantee for Approx-Uncrossable with Regular-Edge-Delete, as in Section 4.1.2.

Define $Y = \bigcup_{C \in \mathcal{C}} \delta_{F'}(C : \zeta(C))$; that is, Y consists of the edges in F' in the coboundary of active sets C and not incident to vertices in $\Gamma(C)$. The set $\zeta(C)$ and $\Gamma(C)$ are defined with respect to the set of edges $F_{p-1} \cup F$, where F is the set of edges chosen up to (but not including) the current iteration. Notice that all the edges in Y must have been added during or after the current iteration. We prove a lemma analogous to Lemma 4.1.2.

Lemma 7.5.4 For each edge $e \in Y$ there exists a witness set $S_e \subset V$ such that

1. $\delta_{F'}(S_e : \zeta(S_e)) = \{e\}$,
2. S_e is small and violated in the current iteration,
3. For each $C \in \mathcal{C}$ either $C \subseteq S_e$ or $C \cap S_e = \emptyset$.

Proof: Any edge $e \in Y$ is also in F', and thus during the edge deletion stage the removal of e causes there to exist some violated set, and hence some small violated set; call this set S. In other words, there can exist no other $e' \in F'$ that is also in $\delta_{F'}(S : \zeta(S))$. This

set S will be the witness set S_e for e, and clearly satisfies (1). Now let F be all the edges added in this phase before the current iteration. To show (2) and (3), notice that when considering edge e in the reverse delete stage, no edge in F had yet been removed. Hence S_e is small-violated even if all the edges of F are included; that is, S_e is small-violated in the current iteration. Property (3) follows by the minimality of the active sets C. ■

We can now prove a lemma analogous to Lemma 4.1.3. Consider a collection of sets S_e satisfying the conditions of the preceding lemma, taken over all the edges e in Y. Recall that such a collection is called a witness family. As in Lemma 4.1.3, we will show that there exists a laminar witness family by uncrossing pairs of sets using Lemma 7.5.1. Because we can replace a pair of sets A, B with a set $\zeta(A \cup B) \not\subseteq A \cup B$, it becomes more difficult to prove that the uncrossing process terminates.

Lemma 7.5.5 If A is a violated set, then $\zeta(\zeta(A)) = A$.

Proof: It is not hard to see that $A \subseteq \zeta(\zeta(A))$. Suppose there exists a vertex $v \in \zeta(\zeta(A)) - A$. Then it must be the case that $v \in \Gamma(A)$, $\zeta(A \cup \{v\}) = \zeta(A)$, and $\Gamma(A \cup \{v\}) = \Gamma(A) - \{v\}$. Since A is violated, $\zeta(A) \neq \emptyset$ and $|\Gamma(A)| = p-1$. But then $\zeta(A \cup \{v\}) \neq \emptyset$ and $|\Gamma(A \cup \{v\})| < p-1$, which contradicts the feasibility of the edge set F_{p-1}. ■

Lemma 7.5.6 Let $\mathcal{S}$ be a collection of small violated sets. Then there exists a laminar family of small violated sets formed by successively replacing a crossing pair of sets A and B with an appropriate choice of $A \cap B$ and $A \cup B$, or $A \cap B$ and $\zeta(A \cup B)$.

Proof: We use a potential function

$$\Phi(\mathcal{S}) = \sum_{S \in \mathcal{S}} \left(|S|^2 + |\zeta(S)|^2 \right)$$

to prove the lemma. If A and B cross and are both small violated sets, then by Lemma 7.5.1 either $A \cap B$ and $A \cup B$, or $A \cap B$ and $\zeta(A \cup B)$ are both small and violated. Let $\mathcal{S}'$ be the collection of sets formed by replacing A and B with the pair of small violated sets. We will show that $\Phi(\mathcal{S}') - \Phi(\mathcal{S}) > 0$. Since uncrossing pairs of sets does not increase the number of sets in the collection, Φ can never grow larger than $2|\mathcal{S}| \cdot n^2$. Thus the uncrossing process must terminate with a laminar family.

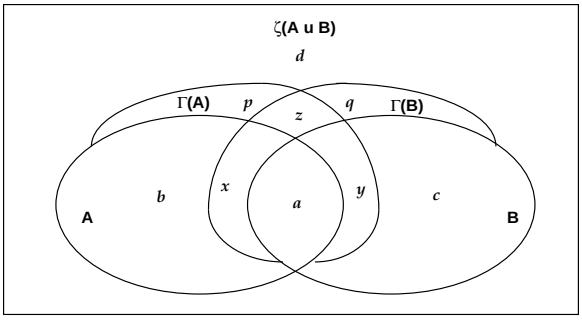

Figure 7-11: Two crossing sets A and B.

Let

$$a = |A \cap B| \qquad b = |A - \Gamma(B) - B| \qquad c = |B - \Gamma(A) - A|$$

$$x = |A \cap \Gamma(B)| \qquad y = |B \cap \Gamma(A)| \qquad z = |\Gamma(A) \cap \Gamma(B)|$$

$$p = |\Gamma(A) - \Gamma(B) - B| \quad q = |\Gamma(B) - \Gamma(A) - A| \qquad d = |\zeta(A \cup B)|$$

(see Figure 7-11). The initial contribution of A to $\Phi(\mathcal{S})$ is $(a+x+b)^2 + (c+q+d)^2$, and the initial contribution of B is $(a+y+c)^2 + (b+p+d)^2$. After uncrossing, the contribution of $A \cap B$ is at least $a^2 + (b+p+d+q+c)^2$ and the contribution of $A \cup B$ (or $\zeta(A \cup B)$, which we treat symmetrically by Lemma 7.5.5) is at least $d^2 + (a+b+c+x+y)^2$. Since all other sets in $\mathcal{S}'$ stay the same, $\Phi(\mathcal{S}') - \Phi(\mathcal{S})$ is at least the difference between these two quantities, which, by algebraic manipulation, is $2((b+p)(c+q) + (x+b)(y+c))$. Since A and B are crossing, $|A-B| > 0$ and $|B-A| > 0$, which implies that $x+b > 0$ and $y+c > 0$. Thus $\Phi(\mathcal{S}') - \Phi(\mathcal{S}) > 0$. ■

We can now prove the following.

Lemma 7.5.7 There exists a laminar witness family.

Proof: By Lemma 7.5.4, there exists a witness family. From this collection of sets we can form a laminar collection of sets as follows. We maintain that all sets S in the collection are small and violated. If the collection is not laminar, there exists a pair of sets A, B that cross. We uncross A and B by replacing them in the collection with either $A \cup B$ and $A \cap B$ or with $\zeta(A \cup B)$ and $A \cap B$. By Lemma 7.5.1, we know that at least one of these

two uncrossings yields two small violated sets. This procedure terminates with a laminar collection by Lemma 7.5.6.

We claim that the resulting laminar collection forms a witness family. This claim can be proven by induction on the uncrossing process. Obviously property (2) holds. Property (3) continues to hold because the uncrossed sets are small violated sets for the current iteration, and must either contain or be disjoint from the minimal small violated sets. Now we must prove (1). Suppose we have two crossing witness sets S_1 and S_2 corresponding to edges e_1 and e_2, and, without loss of generality (by Lemma 7.5.5) suppose we uncross them into $S_1 \cap S_2$ and $S_1 \cup S_2$. We claim that

$$|\delta_{F'}(S_1 : \zeta(S_1))| + |\delta_{F'}(S_2 : \zeta(S_2))| \geq |\delta_{F'}(S_1 \cup S_2 : \zeta(S_1 \cup S_2))| + |\delta_{F'}(S_1 \cap S_2 : \zeta(S_1 \cap S_2))|.$$

This follows from a simple counting argument showing that each edge counted on the right-hand side is accounted for by the same edge on the left-hand side. Because F' covers all small violated sets, we know that $|\delta_{F'}(S_1 \cup S_2 : \zeta(S_1 \cup S_2))| \geq 1$ and $|\delta_{F'}(S_1 \cap S_2 : \zeta(S_1 \cap S_2))| \geq 1$, and so it must be the case that $|\delta_{F'}(S_1 \cup S_2 : \zeta(S_1 \cup S_2))| = |\delta_{F'}(S_1 \cap S_2 : \zeta(S_1 \cap S_2))| = 1$. Then either $e_1 \in \delta'_F(S_1 \cup S_2 : \zeta(S_1 \cup S_2))$ and $e_2 \in \delta_{F'}(S_1 \cap S_2 : \zeta(S_1 \cap S_2))$, or vice versa. ■

The remaining proof of the inequality is essentially identical to the proof in Section 4.1.2, but we include it here for the sake of completeness. Let $\mathcal{S}$ be a laminar witness family. Augment the family with the vertex set V. The family can be viewed as defining a tree H with a vertex v_S for each $S \in \mathcal{S}$ and edge (v_S, v_T) if T is the smallest element of $\mathcal{S}$ properly containing S. To each active set $C \in \mathcal{C}$ we associate the smallest set $S \in \mathcal{S}$ that contains it. We will call a vertex v_S active if S is associated with some active set C. Let $\mathcal{L}(v_S)$ be the collection of sets $C \in \mathcal{C}$ associated with an active vertex v_S.

Lemma 7.5.8 The tree H has at most one inactive leaf.

Proof: Only V and the minimal (under inclusion) witness sets can correspond to leaves. Any minimal witness set is a small violated set, and thus must contain an active set which corresponds to it. Thus only V can correspond to an inactive leaf. ■

Lemma 7.5.9 For any active vertex v_S in H, the degree of v_S is at least $\sum_{C \in \mathcal{L}(v_S)} |\delta_{F'}(C :$

$\zeta(C))|$.

Proof: Note that the one-to-one mapping between the edges of Y and the witness sets implies a one-to-one mapping between the edges of Y and the edges of H: each witness set S defines a unique edge (v_S, v_T) of H, where T contains S. Consider any edge $e \in \delta_{F'}(C : \zeta(C))$ for some $C \in \mathcal{C}$. Let (v_{S_e}, v_T) be the edge defined by the witness set S_e. The active set C must be associated with either v_{S_e} or v_T. By summing over all edges $e \in \delta_{F'}(C : \zeta(C))$ for all active sets C corresponding to an active vertex of H (that is, all $C \in \mathcal{L}(v_S)$), we obtain the lemma. ■

Let H_a denote the set of active vertices in H and let d_v denote the degree of a vertex v. Then,

$$\sum_{v \in H_a} d_v = \sum_{v \in H} d_v - \sum_{v \in H - H_a} d_v \leq 2(|H| - 1) - 2(|H| - |H_a| - 1) - 1 = 2|H_a| - 1.$$

This inequality holds since H is a tree with $|H| - 1$ edges, and since all vertices of $H - H_a$ except for possibly one have degree at least 2. The lemma above implies that $\sum_{C \in \mathcal{C}} |\delta_{F'}(C : \zeta(C))| \leq \sum_{v \in H_a} d_v$, while clearly $|H_a| \leq |\mathcal{C}|$. Thus

$$\sum_{C \in \mathcal{C}} |\delta_{F'}(C : \zeta(C))| \leq 2|\mathcal{C}|,$$

as desired.

We now turn to the problem of implementing the algorithm. As with Approx-Uncrossable, we must show how to implement the strong oracle, how to select the edge minimizing ϵ in each iteration, and how to remove edges. Our implementation techniques from Chapter 5 do not seem to generalize cleanly to this case, so we will merely give a straightforward implementation.

The strong oracle Small-Violated can be implemented using network flow theory. Suppose that in phase p there is a minimal small violated set S with respect to the edge set $F_{p-1} \cup F$. We can determine S as follows. Construct a directed graph $G' = (V', E')$ from the graph $(V, F_{p-1} \cup F)$ by making two copies v', v'' for each $v \in V$, adding directed edges (u'', v') and (v'', u') for each $(u, v) \in F_{p-1} \cup F$, and adding an edge (v', v'') for each $v \in V$.

Consider a vertex $s \in S$ and a vertex $t \in \zeta(S)$. It is known that the value of a maximum s'-t'' flow in G' corresponds to the number of vertex-disjoint paths between s and t in G [98, p. 458]. Furthermore, the minimal mincut in G' will correspond to S. The minimal mincut is given by the vertices reachable from s in the residual graph of the flow.

Thus a straightforward way of implementing SMALL-VIOLATED is to calculate an s-t maximum flow for all pairs of vertices $s, t \in V$, find all the minimal mincuts, then extract all the minimal small violated sets from this collection. Since there will be $O(n^2)$ candidate sets, it will take $O(n^3)$ time to extract the minimal small violated sets. We can, as in Section 5.1.3, cut down the time incurred by the sequence of calls to SMALL-VIOLATED by keeping track of the residual graphs for each network flow problem. At the beginning of phase p, we ensure that we have found a flow of at least p in each graph if such a flow exists. Whenever the algorithm adds an edge, we add the edge to each graph, and see if it makes any more vertices reachable from s. Given the active sets from the previous iteration, we can then extract the new active sets in $O(n^2)$ time. Let $m' = \min(nk, m)$. As in Section 5.1.3, it will take $O(km')$ total time per vertex pair to keep the graph updated at the beginning of each phase, and $O(m')$ time per vertex pair to find the new minimal mincuts in the graph over the course of a phase. Thus implementing SMALL-VIOLATED will take $O(km'n^2)$ time through the course of the algorithm.

To implement the edge selection step, we keep track of a variable $d(e) = \sum_{S,T:e \in \delta(S:V-T-S)} y_{S,T}$ for each edge e. Let $a(e)$ denote the number of sets $C \in \mathcal{C}$ for which $e \in \delta(C : \zeta(C))$. Then in each iteration we search for the edge that minimizes $\epsilon = \frac{c_e - d(e)}{a(e)}$. Because of Lemma 7.5.1, we can prove as in Lemma 3.2.2 that the active sets over all iterations of a phase form a laminar family. Thus we can, as before, use a union-find structure to keep track of the vertices in the current collection $\mathcal{C}$ of active sets. Whenever a new active set C is formed, we use $O(m'\alpha(n,n))$ time to find the vertices in $\Gamma(C)$. Then in each iteration we examine each edge to compute $\frac{c_e - d(e)}{a(e)}$. This takes $O(\alpha(n,n)+k)$ time per edge: $O(\alpha(n,n))$ time to determine the $C \in \mathcal{C}$ to which its endpoints belong and $O(k)$ time to check if the edge is in $\delta(C : \zeta(C))$. It takes $O(n\alpha(n,n))$ time per phase to maintain the union-find structure on the active sets. Thus the overall running time of the edge selection process is $O(mn(\alpha(n,n)+k))$ per phase.

Every time an edge is removed in the edge deletion stage of phase p, we must verify that the remaining graph is still p-vertex connected. Steiglitz, Weiner, and Kleitman [117] have shown that this can be done with $O(pn)$ network flows. Since each flow is in a graph with m' edges, and we need only p augmenting paths per flow, the time needed to compute each flow is $O(pm')$. We check $O(kn)$ edges for deletion over the course of the algorithm, so that the total time used for the edge deletion step is $O(k^3m'n^2)$.

The discussion above leads to the following theorem.

Theorem 7.5.10 The algorithm APPROX-k-VERTEX-CONN runs in $O(k^3m'n^2)$ time and produces a k-vertex-connected set of edges F_k such that

$$\sum_{e \in F_k} c_e \leq 2\mathcal{H}(k) Z^*_{kVC}.$$

CHAPTER 8

Experimental Results

The main criticism which is often formulated with regard to approximation algorithms is that, although they are backed up by a performance guarantee, they might not generate "nearly-optimal" solutions in practice. Indeed, a practitioner will seldom be satisfied with a solution guaranteed to be of cost less than, say, twice the optimum cost; by far, he will prefer a heuristic algorithm which typically generates a solution within, say, 3% of optimality, although this heuristic might from time to time generate a more costly solution. For example, in the context of the traveling salesman problem, computational studies show that the heuristic algorithm of Lin and Kernighan [85] outperforms in practice the algorithm of Christofides [17], although the latter has a performance guarantee of $\frac{3}{2}$: Christofides' algorithm typically comes within 9% of optimal, while Lin-Kernighan comes within 2% [40].

Motivated by these comments, this thesis initiates a computational study of the approximation algorithms presented in the previous chapters. We study our 2-approximation algorithm for minimum-weight perfect matching based on APPROX-PROPER-0-1 (see Section 6.3). We restrict our attention to Euclidean instances where the edge costs are the distances between points under the L_2 or L_∞ norms. The main reason for choosing to study our Euclidean matching approximation algorithm is that a number of heuristics and exact algorithms for Euclidean matching have been designed and implemented, and it is not

difficult to compare the performance of our algorithm to other algorithms, both in terms of running time and quality of solutions produced.

In addition to the original algorithm of Edmonds [31], other algorithms for minimum-weight perfect matching have been devised by Cunningham and Marsh [23], Derigs [25], Grötschel and Holland [56], and Lessard, Rousseau, and Minoux [82]. Various implementations of these algorithms have been studied by Applegate and Cook [5], Cunningham and Marsh [23], Derigs [24, 25, 26], Derigs and Metz [27, 28], and Lessard, Rousseau, and Minoux [82]. Aside from Applegate and Cook's paper, the largest problem studied in these papers was on 1000 vertices; Applegate and Cook solve one example with 101,230 vertices.

Many heuristics for Euclidean perfect matching have been proposed. A survey of many of these heuristics can be found in Avis [8]. Computational studies of some of them have been carried out by Iri, Murota, and Matsui [64] and Jünger and Pulleyblank [67]. The largest instance in these two studies had 10,000 vertices.

The remainder of the chapter is divided into four sections. Section 8.1 discusses the implementation of Approx-Proper-0-1 that we used in our study. The problem with the implementation given in Chapter 5 is that it uses $O(n^2)$ space, which severely restricts the size of the instances that can be approximated. Section 8.2 reviews known results about the behavior of Euclidean optimization problems on random instances. These results prompted our study of particular properties of the approximation algorithm. Section 8.3 gives the results of our study. We conclude with a discussion of the results in Section 8.4.

8.1 Description of an Implementation

In this section, we describe the implementation of Approx-Proper-0-1 that we use in our computational study. The heart of the implementation is a new implementation of the edge selection routine. Recall that Approx-Proper-0-1 maintains variables $d(v)$ for each vertex v, and that for the minimum-weight perfect matching problem we use the proper function $h(S) = |S| (\mathrm{mod}\ 2)$. In each iteration, we select the edge $e = (u, v)$ between connected components C_p and C_q that minimizes $\epsilon = \frac{c_e - d(u) - d(v)}{h(C_p) + h(C_q)}$. The packet implementation of the edge selection step given in Section 5.2.2 is somewhat complicated to code. The

simple implementation described in Section 5.2.1 has two main disadvantages for performing computational experiments. The first is that it uses $\Theta(n^2)$ space; this severely limits the size of the instances that can be solved. The second is that the running time is in fact $\Theta(n^2 \log n)$: the algorithm must add at least $n/2$ edges to obtain a feasible solution, necessitating $\Theta(n^2)$ queue operations. The main theoretical advantage of our new implementation compared to the one given above is that it is much more space efficient, using only $O(n)$ space. Moreover, although its worst-case time complexity appears to be worse than the original implementation, it performs well enough on average to allow us to run relatively large instances. It will require $O(n^2)$ queue operations, and we will see experimentally that it requires $O(n^{1+\epsilon})$ queue operations on random instances, although other factors will now dominate the running time.

Before we go on to describe the main idea behind selecting edges in our implementation, we note that several small tricks are necessary to ensure that other parts of the algorithm do not force the running time to be $\Omega(n^2)$. For example, updating the $d(v)$ variables each iteration would take $\Theta(n^2)$ time. To avoid this, we augment the union-find structure used to keep track of the connected components of the current edge set. Since the $d(v)$ are increased by the same amount for all vertices v in the same component, we increase an offset in the root of the component, and define $d(v)$ to be the sum of the offsets along the path to the root. In addition, we let the increases for a component accumulate and only change the offset when we merge the component with another component, or when we need to calculate $d(v)$ for some vertex v in the component.

Recall that the current time T of the algorithm is the sum of the values of ϵ over the preceding iterations, and the addition time of an edge $e = (u, v)$ is defined to be $T + \frac{c_e - d(u) - d(v)}{h(C_p) + h(C_q)}$. The basic idea of the implementation, suggested to us by David Johnson, is that each component should maintain an estimate of its closest neighboring component under addition times. The corresponding edges are placed in a priority queue with the estimates as the key values. The estimates are maintained in such a way that the shortest edge (under addition times) between two components is always found in any iteration; thus the algorithm can be successfully implemented. The main advantage of this implementation is its space efficiency: we need to keep track of the keys of only $|\mathcal{C} \cup \mathcal{I}|$ edges, where $\mathcal{C} \cup \mathcal{I}$

is the set of components.

More formally, let $l(e)$ denote the current addition time of edge $e = (i, j)$. For two components C_p and C_q in $\mathcal{C} \cup \mathcal{I}$, let $l(C_p, C_q)$ be equal to the smallest addition time of an edge with one endpoint in C_p and the other in C_q. The key of an edge e in the queue will be denoted by $k(e)$ and corresponds to the addition time of that edge when it was added to the queue. By abuse of notation, we let $k(C)$ denote the key of the edge in the queue which was selected by component C.

The implementation works as follows:

- Initially, every vertex calculates its nearest neighbor (under addition times) and puts the corresponding edge in the priority queue with a key value of the addition time.
- Whenever we pull an edge e off the queue, we check if its key value $k(e)$ is no less than its actual addition time $l(e)$. We maintain that whenever this is true, then the edge is the next edge that should be added; that is, it has the smallest addition time. Whenever two components get merged into one, we find its new nearest neighbor under addition times.
- When the key value of the edge e is less than the actual addition time, we then search for the associated component's real nearest neighbor, bounding the search by the correct addition time $l(e)$ of e.

In order to prove that the implementation is correct, we first prove that it maintains an invariant.

Lemma 8.1.1 At any point in the algorithm, for all $C_p, C_q \in \mathcal{C}$, $\min(k(C_p), k(C_q)) \leq l(C_p, C_q)$.

Proof: Certainly the invariant is true initially. Suppose that we insert an edge e selected by component C to the queue. This insertion might be the result of either two components merging into C or the discovery that the edge in the queue corresponding to C has a key less than its addition time. In both cases, the invariant is maintained for any two components C_p and C_q different from C. Moreover, if $C_p = C$ then our choice of the edge to insert

guarantees that $\min(k(C), k(C_q)) \leq k(C) = k(e) = l(e) \leq l(C, C_q)$, implying that the invariant continues to hold. ■

The invariant leads to a proof of correctness.

Theorem 8.1.2 The implementation selects an edge with the smallest addition time in every iteration.

Proof: In each iteration of the algorithm, we must find the edge with the smallest addition time. Let a denote the smallest addition time of this iteration, and let e be the edge at the top of the queue. We will show that whenever $k(e) \geq l(e)$ then $l(e) = a$ and thus the algorithm correctly selects edge e. Whenever $k(e) < l(e)$ we replace the queue element e with another edge e' such that $k(e') = l(e')$. Such a replacement does not affect the distances between components or the other key values in the queue, so we can replace at most $|\mathcal{C}|$ number of edges before we must reach the case that $k(e) \geq l(e)$ for the top element e of the queue.

Suppose $k(e) \geq l(e)$. By the invariant, for any $C_p, C_q \in \mathcal{C} \cup \mathcal{I}$, we have $\min(k(C_p), k(C_q)) \leq l(C_p, C_q)$. But since e is the edge at the top of the queue, $k(e) \leq k(C)$ for all $C \in \mathcal{C} \cup \mathcal{I}$ and, thus, $k(e) \leq l(C_p, C_q)$. The fact that $l(e) \leq k(e)$ now implies that $l(e) \leq l(C_p, C_q)$ for any $C_p, C_q \in \mathcal{C} \cup \mathcal{I}$. In other words, e is an edge with smallest addition time. ■

We now evaluate the worst-case number of queue operations. The argument of the theorem shows that we perform at most $O(n)$ queue operations for each edge selection. This implies that the algorithm performs a total of $O(n^2)$ queue operations.

In order to complete the description of our implementation, we must describe how to find an edge e that represents the nearest neighbor of a component C under addition time. To do this, we use k-d trees of Friedman, Bentley, and Finkel [41] as described in Bentley [13]. A k-d tree is a binary tree that corresponds to a partitioning of a given set of points in d-dimensional space; here we use $d = 2$. The tree is constructed by determining whether the points are spread out most in the x or y direction, then finding the vertical or horizontal line that splits the points in half in that direction. The line determines an internal node of the tree, and the procedure is performed recursively on each half until the number of points remaining is below a certain threshold (usually 5 or 6). Hence each leaf in the tree

corresponds to a bucket containing at most a certain number of points. The tree can be used to perform a search for the nearest neighbor of a given point under a number of norms (including L_1, L_2 and L_∞): at each internal node we first search the subtree containing the given point, then search the other tree iff the nearest neighbor found so far is no closer than the distance from the given point to the line determining the internal node. Friedman et al. show experimentally that this search takes $O(\log n)$ expected time, and Bentley [13] gives a bottom-up variation that seems to take $O(1)$ expected time. Notice that this search method also lends itself to searching for the nearest neighbor within a certain radius.

To find the nearest neighbor of a component C under addition time, we iterate through the vertices in C to find the smallest edge (under addition time) from the vertex to a vertex not in C. We use the smallest edge found so far to bound the search on the next vertex. Let v_i denote the point in the Euclidean plane corresponding to vertex i, and let $\|v_i - v_j\|$ denote the distance between vertices i and j, and hence the cost of edge (i, j). Suppose we are searching from vertex i in component C, the current time is T, and the smallest edge found so far has addition time a. Then since $d(v) \leq T$ for all $v \in V$, any potentially smaller edge must be within distance $(a - T)(h(C) + 1) + d(i) + T$. As long as T remains small, this reasonably restricts the number of points that we have to consider.

As one might suspect, the time spent performing these searches dominates all other operations in our implementation. Therefore, we introduce a few tricks for speeding up these searches. The first trick is that whenever we notice that all the vertices in a subtree of the k-d tree belong to the same component, we label the subtree with the name of that component. Then whenever we search for the nearest neighbor of a vertex in C, we ignore all subtrees labelled C. This trick is useful as C becomes large and most of the neighboring vertices of a given vertex in C are also in C.

Another trick we use is that when searching for the nearest neighbor of a vertex i in C, we can sometimes infer when possible nearest neighbor candidates will be closer to another vertex j in C than i. We say that i is *shadowed* by j. Thus we can disregard these nearest neighbor candidates. To be more formal, consider the four quadrants of the plane formed by using v_i as the origin (see Figure 8-1). Let (x_i, y_i) denote the coordinates of v_i. Let n_i be the Euclidean point $(x_i + d(i), y_i)$ and let e_i be the point $(x_i, y_i + d(i))$.

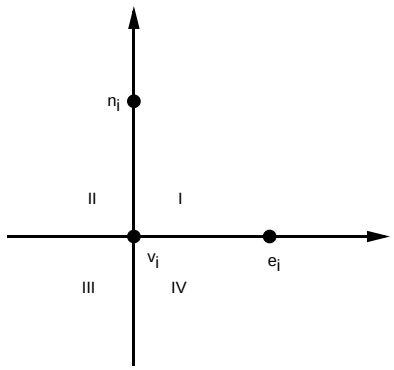

Figure 8-1: The four quadrants around v_i.

Theorem 8.1.3 Suppose there is some other vertex j in C such that j is not in quadrant III (i.e. j is such that $x_j \geq x_i$ or $y_j \geq y_i$), and

$$\|n_i - v_j\| \leq d(j) \text{ and } \|e_i - v_j\| \leq d(j).$$

Then, for any vertex q in quadrant I ($x_q \geq x_i$ and $y_q \geq y_i$) that is in component $C' \neq C$, vertex j is at least as close to vertex q under addition times as is vertex i.

Proof: To show this, we will prove that $\|v_q - v_j\| - d(j) \leq \|v_q - v_i\| - d(i)$. From this it will follow that $\frac{\|v_q - v_j\| - d(j) - d(q)}{h(C)+h(C')} \leq \frac{\|v_q - v_i\| - d(i) - d(q)}{h(C)+h(C')}$.

We consider two cases. First suppose that v_j is in quadrant I. Because vertex q is not in component C, it cannot lie inside the triangle defined by v_i, n_i and e_i. Furthermore, v_q cannot lie inside the triangle defined by v_j, n_i and e_i since both n_i and e_i are within distance $d(j)$ of v_j. As a result, the line segment (v_q, v_i) must intersect either the line segment (n_i, v_j) or the line segment (v_j, e_i) (see Figure 8-2). Assume the former (the other case is similar), and let m be the intersection point. Then

$$\|v_i - v_q\| + \|v_j - n_i\| = \|v_i - m\| + \|m - n_i\| + \|v_q - m\| + \|m - v_j\| \geq \|v_i - n_i\| + \|v_q - v_j\|,$$

which implies that $\|v_q - v_i\| + d(j) \geq \|v_q - v_j\| + d(i)$, as desired.

Now suppose that v_j is in quadrant IV (the case for II is similar). Since $\|v_i - v_j\| \leq \|n_i - v_j\| \leq d(j)$ and v_q is not in the same component as v_i and v_j, v_q cannot be in the triangle

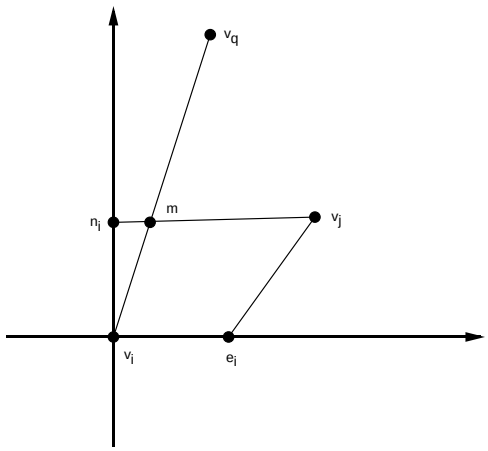

Figure 8-2: Illustration of the case in which v_q is in quadrant I.

defined by v_i, v_j, and n_i. Hence, the line segments (v_q, v_i) and (n_i, v_j) must intersect. Then the proof is the same as above. ■

Thus if vertex i is shadowed as in the statement of the theorem, we need not look for nearest neighbors of i in quadrant I. Obviously, shadowing is symmetric with respect to quadrants. Once the size of a component has increased by a certain amount, we sweep through the component, determining in which quadrants each vertex is shadowed. We store this information in four bits for each vertex and use it when we perform nearest neighbor searches to cut down the scope of the search for each vertex. If a vertex is shadowed in all four quadrants, then we do not perform a search on it at all.

In conclusion, we note that the final step of the algorithm for minimum-weight perfect matching, which transforms the even-sized components into a perfect matching, is not implemented as described in Section 6.3. Indeed, we have observed experimentally that most of these components contain few vertices (see Section 8.3 for details). As a result, for any component having at most 10 vertices, we compute optimally the perfect matching on the vertices spanned.

8.2 Probabilistic Analysis of Euclidean Functionals

Our study of the behavior of the approximation algorithm on random instances is prompted by a wealth of results about the behavior of combinatorial optimization problems on randomly distributed Euclidean instances. In this section, we review the results known about Euclidean matching and other problems, and indicate results that are likely to be true about the approximation algorithm.

In the basic version of the *Euclidean model*, the vertices of the problem instance are distributed independently and uniformly in the unit square $[0,1]^2$ and the Euclidean metric plays the role of cost function. The *functionals* of interest are typically the values of combinatorial optimization problems (such as the traveling salesman, matching or minimum spanning tree problems) on these randomly distributed points. The behavior of these functionals is somewhat independent of the functional itself and, for these reasons, we briefly review some of these probabilistic results for the most studied problem, the traveling salesman problem. We also indicate results likely to hold, although they have never been proved. For a more detailed picture of the field, the reader is referred to [115, 69, 113].

The area of probabilistic analysis under the Euclidean model has its origin in the pioneering paper of Beardwood, Halton, and Hammersley [11]. It characterizes the asymptotic behavior of the value of the traveling salesman problem by proving the existence of a constant β_{TSP} such that

$$\lim_{n\to\infty} \frac{TSP_n}{\sqrt{n}} = \beta_{TSP} \qquad \text{almost surely} \tag{8.1}$$

where TSP_n denotes the value of the optimal tour on $X_1, X_2, \ldots, X_n$ with the X_i's being an infinite sequence of independently and uniformly distributed vertices from $[0,1]^2$. We should point out that the exact value of β_{TSP} is not known. More recently, a careful analysis of the functional has led to the following results and/or conjectures. It is known that $Var\, TSP_n$ is upper bounded by a constant (see Steele [114]): $Var\, TSP_n \leq \frac{16}{\pi} + O(\frac{1}{n})$. Quoting Steele [113],

> it seems inevitable that one has a genuine limit, $\lim_{n\to\infty} Var TSP_n = \sigma^2 > 0$.

It is less certain but (still very likely) that one has a central limit theorem $TSP_n - E[TSP_n] \sim N(0, \sigma^2)$.

Rhee and Talagrand [111] have proved a first step towards this central limit theorem by showing that the tails of TSP_n are Gaussian or subgaussian: there exists K such that, for all t, $Pr\left(|TSP_n - E[TSP_n]| > t\right) \leq Ke^{-t^2/K}$. From the knowledge of $E[TSP_n]$ for a finite value of n, one can derive a bound on the limiting constant β_{TSP}. Indeed (see Jaillet [65]), there exists a constant $\gamma_{TSP} < 9.5$ such that, for all n, $|E[TSP_n] - \beta_{TSP}\sqrt{n}| \leq \gamma_{TSP}$. Furthermore, it seems likely that there exists a limit α_{TSP} such that $\lim_{n\to\infty} |E[TSP_n] - \beta_{TSP}\sqrt{n}| = \alpha_{TSP}$.

Some of these results also hold for other functionals; see Steele [115], Goemans [51], and the references above. However, for the minimum-cost perfect matching problem and its associated functional M, only the asymptotic behavior (8.1) is known to hold (Papadimitriou [97]), although the other results and/or conjectures are likely to be true.

From these results and/or conjectures, we shall implicitly assume for our experimental study that, for several functionals L, L_n is normally distributed with mean $\beta_L\sqrt{n} + \alpha_L$ and variance σ_L^2. These functionals are the values of the minimum-cost perfect matching, and also of F', the perfect matching and the dual solution returned by our approximation algorithm. We shall denote these additional functionals by F', P (for primal) and D (for dual).

Functionals of a more structural nature have also been studied. For example, Steele et al. [116] have shown that there exist constants ν_k such that the number of degree k vertices in a minimum-cost spanning tree divided by n is almost surely equal to ν_k. In a set of experiments Steele et al. estimated these constants to be $\hat{\nu}_1 = .221$, $\hat{\nu}_2 = .566$, $\hat{\nu}_3 = .206$, $\hat{\nu}_4 = .007$, and $\hat{\nu}_5 = .000$. It is known that $\nu_k = 0$ for $k \geq 6$.

8.3 Results

We summarize our main experimental results in a sequence of tables. Table 8.1 contains our results on structured examples. We drew our structured examples from the Traveling Salesman Library, TSPLIB [108]. Tables 8.2 and 8.3 contain our results on random instances;

the first table is for instances using the L_2 norm, and the second is for instances using the L_∞ norm. The columns LBGAP and OPTGAP give the ratio of the cost of the approximate matching to the cost of the dual lower bound and to the cost of an optimal matching respectively. For the purposes of comparison, we used the efficient code of Applegate and Cook [5] to find optimal matchings. We used the variation of the Applegate and Cook code which begins with a fractional 10 nearest neighbor graph. The Time columns specify the running time of the approximation algorithm, while the AC Time column specifies the running time of the Applegate and Cook algorithm. All running times are in CPU seconds on a Silicon Graphics Challenge machine with eight 100Mhz MIPS R4400 processors. The Speedup column gives the ratio of the running time of Applegate and Cook's algorithm to the running time of the approximation algorithm. We summarize our estimates of the parameters β and α in Table 8.4. We include for comparison parameters given for other matching heuristics. Finally, some asymptotic estimates of structural properties of the solutions are given in Table 8.5. Each of these tables is discussed in the following paragraphs.

As mentioned above, the structured examples in Table 8.1 were taken from the TSPLIB. The number in the problem name indicates the number of vertices in the problem. We attempted to use the same procedure as given in Applegate and Cook [5]; namely, if the example contained an odd number of points, we sorted by x and y coordinates, then deleted the last point. We also attempted as much as possible to run the same suite of examples as given in [5]. The solution obtained by the approximation algorithm for the problem r1002 is shown in Figure 8-9.

The random examples in Tables 8.2 and 8.3 were generated on a 2^{20} by 2^{20} grid using the UNIX `random()` function. A single seed was used to generate all the instances of a given size. The first entry of Table 8.2 comes from a sequence of 1000 experiments run separately on a VAX 9000 (11 data points had to be omitted). We used these experiments to get an upper bound on the variance of the matching parameters for F', P, D, and M on the unit square. Using the parameter with the largest variance (F'), we obtained an upper bound of .05399 with 99% confidence. This information was used to decide the number of experiments to perform in order to obtain small confidence intervals on the parameters β. We did not use the 1000 experiments in these parameter estimations.

Problem Name	Norm	LBGAP	OPTGAP	Time	AC Time
r1002	L_2	1.0459	1.0154	3.57	2.69
r2392	L_2	1.0357	1.0098	7.38	11.51
pcb3038	L_2	1.0298	1.0093	30.32	22.26
rl5934	L_2	1.0237	1.0093	326.30	119.85
pla7396	L_2	1.0172	1.0094	461.25	203.11
rl11848	L_2	1.0287	1.0118	944.60	229.82
d18512	L_2	1.0357	1.0164	3651.13	664.93
r20726	L_∞	1.0440	1.0188	718.95	4636.06
pla33810	L_2	1.0214	1.0169	24687.20	1704.09
pla85900	L_∞	1.0152	1.0134	107653.81	6202.22

Table 8.1: Experimental results on TSPLIB instances.

Size	Trials	Ave LBGAP	Ave OPTGAP	Max LBGAP	Max OPTGAP	Ave Time	Ave Speedup
2^{10}	989	1.0369	1.0159	1.0587	1.0346	–	–
2^{10}	64	1.0369	1.0158	1.0615	1.0367	5.09	.92
2^{11}	64	1.0369	1.0160	1.0486	1.0245	21.60	.92
2^{12}	32	1.0372	1.0165	1.0500	1.0265	79.75	1.12
2^{13}	32	1.0361	1.0157	1.0434	1.0208	261.59	1.72
2^{14}	16	1.0368	1.0163	1.0390	1.0189	1330.69	2.17
2^{15}	16	1.0371	1.0165	1.0400	1.0189	7533.67	2.00
2^{16}	8	1.0370	1.0164	1.0379	1.0179	32942.70	2.00
2^{17}	4	1.0374	1.0163	1.0383	1.0170	200820.00	1.87

Table 8.2: Experimental results on random instances using the L_2 norm.

Size	Trials	Ave LBGAP	Ave OPTGAP	Max LBGAP	Max OPTGAP	Ave Time	Ave Speedup
2^{10}	64	1.0440	1.0190	1.0644	1.0382	4.04	.73
2^{11}	64	1.0440	1.0197	1.0564	1.0289	14.86	.87
2^{12}	32	1.0440	1.0201	1.0526	1.0265	53.15	1.27
2^{13}	32	1.0440	1.0197	1.0514	1.0254	246.27	1.73
2^{14}	16	1.0433	1.0195	1.0456	1.0222	885.21	2.79
2^{15}	16	1.0435	1.0196	1.0452	1.0213	4577.87	2.95
2^{16}	8	1.0446	1.0205	1.0470	1.0223	28017.50	2.16
2^{17}	4	1.0445	1.0197	1.0451	1.0203	150841.00	3.19

Table 8.3: Experimental results on random instances using the L_∞ norm.

We summarize our findings on the matching parameters in Table 8.4. Parameters were estimated using a least-squares fit. The "Std Err" column gives the standard error s_b, or the estimated standard deviation, of the estimated parameter. At 99% confidence, the actual parameter is within $\pm 2.576\, s_b$ of the estimated parameter; for example, with 99% confidence β_M is between .31 and .3106. To allow comparisons with our algorithm, we include the β coefficient of several other Euclidean matching heuristics from the literature, including the Serpentine and Spiral-rack heuristics of Iri, Murota, and Matsui [64], the Rectangle and Strip heuristics of Supowit, Plaisted, and Reingold [119], the MST heuristic of Papadimitriou as given in Supowit et al. (and tested in [67]), and the DUST heuristic of Jünger and Pulleyblank [67]. All of these heuristics run in $O(n \log n)$ time, except for the first two, which require $O(n)$ time. We should also mention that there have been other efforts to estimate the matching parameter β_M, where the cost of the matching was assumed to be $\beta_M \sqrt{n}$, rather than $\beta_M \sqrt{n} + \alpha_M$. Scatterplots of the matching cost divided by $\sqrt{n}$ (see Figure 8-3) tend to suggest the existence of this additional term. Because of the omission of α_M, we believe previous estimates are overestimates. The influence of the α_M term is especially strong if the maximum number of vertices in the experiments is small. Papadimitriou [97] conjectured that $\beta_M = .35$, based on some experiments that had at most 200 vertices. Iri et al. [64] noted that their experiments on sizes up to 256 vertices seemed to indicate that $.32 \leq \beta_M \leq .33$ (note that this agrees well with our predicted value for this value of n). Weber and Liebling [125] obtained an estimate $\beta_M \approx .3189$; the largest example

	L_2 norm				L_∞ norm			
	$\hat{\beta}$	Std Err	$\hat{\alpha}$	Std Err	$\hat{\beta}$	Std Err	$\hat{\alpha}$	Std Err
Cost of F'	.3363	.0002	.2720	.0236	.2989	.0002	.2248	.0207
Approx. Matching (P)	.3154	.0001	.2327	.0150	.2807	.0001	.2038	.0149
Opt. Matching (M)	.3103	.0001	.2357	.0127	.2752	.0001	.2052	.0129
Lower bound (D)	.3041	.0001	.2298	.0121	.2688	.0001	.1978	.0123
Serpentine [64]	.585				.545			
Spiral-rack [64]	.495				.450			
Rectangular [119, 109]	.5164				.4288			
Strip [119]	.474				.436			
MST-H [119, 67]	.358				–			
DUST [67]	.338				–			

Table 8.4: Estimates of matching constants for our approximation algorithm (adjusted for the unit square) and other heuristics. The estimates on Rectangular and Strip are analytically determined.

used in their study was on 1,000 vertices. It is also likely that there is some sensitivity to the means of generating the random instances in the estimation of these parameters.

In Table 8.5, we list some asymptotic estimates of the structural properties of solutions. All appear to be linear in n, and we modelled them either as γn or $\gamma n + \eta$ for some constants γ, η. The choice of which model to use was based on whether the residuals of the estimation were skewed for low values of n in the γn model: if so, the additive constant was included. For the most part, the variance of properties associated with the set of edges F tended to be quite high, growing with n^2, while the variance of properties associated with F' tended to be low, growing with $\sqrt{n}$. We illustrate this in Figure 8-4 with scatterplots of the fraction of vertices of degree 1 in both F and F'. We judged the order of growth of the variance for each property by looking at its relative increase from instances of size 2^n to instances of size 2^{n+1}. We used this judgment to properly adjust the least-squares estimation (least-squares requires constant variance in the observations). In looking at the data we observed that unlike the minimum-cost spanning tree it is possible to have vertices of degree 6 or 7 in F, but vertices of degree 7 are extremely rare, and we saw no vertices of degree 8. Similarly, vertices of degree 5 in F' are also extremely rare, and we saw no vertices of higher degree.

We illustrate the behavior of the algorithm on random instances by showing an example

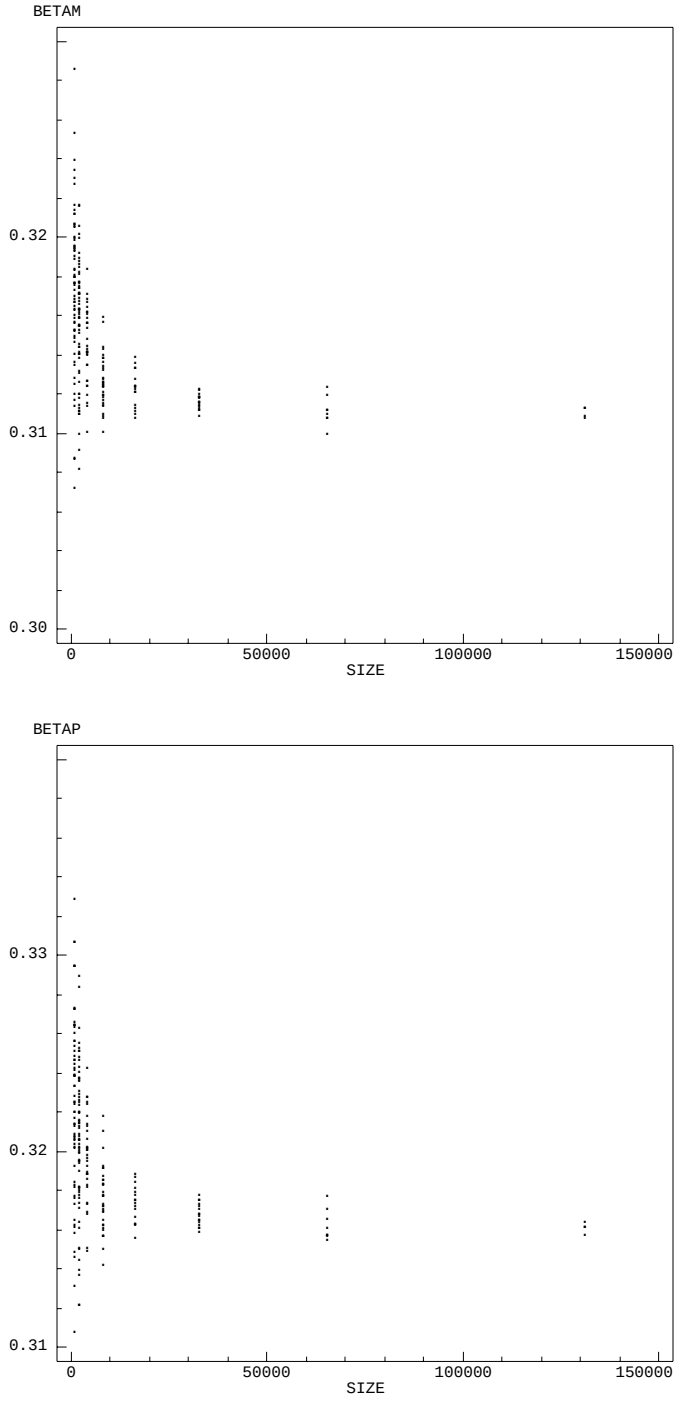

Figure 8-3: Scatterplots of the matching cost (M) divided by $\sqrt{n}$ versus n (top), and the approximate matching cost (P) divided by $\sqrt{n}$ versus n (bottom). Both costs have been scaled down to the unit square.

	L_2 norm		L_∞ norm	
	F	F'	F	F'
Vertices of degree 1	$.311n + 36$	$.954n$	$.320n + 20$	$.950n$
Vertices of degree 2	$.508n - 21$	0	$.494n - 12$	0
Vertices of degree 3	$.166n - 12$	$.046n$	$.167n - 8$	$.050n$
Vertices of degree 4	$.014n - 2$	0	$.018n - 1$	0
Vertices of degree 5	$.001n$	≈ 0	$.001n$	≈ 0
Number of edges	$.931n$	$.546n$	$.936n$	$.550n$
Number of components	$.057n + 26$	$.454n$	$.057n + 15$	$.450n$
Components of size 2	–	$.416n$	–	$.409n$
Components of size 4	–	$.032n$	–	$.034n$
Components of size 6	–	$.005n$	–	$.006n$
Components of size 8	–	$.001n$	–	$.001n$
Components of size 10	–	≈ 0	–	≈ 0

Table 8.5: Estimates of asymptotic properties of solutions.

of the algorithm working on a random 500 vertex instance in Figures 8-5 to 8-8.

We conclude with some observations about the estimated behavior of our implementation. Modelling the number of queue operations as $\lambda n^\mu \epsilon$ (where ϵ is assumed to be a normally distributed error term), we obtained an estimate of $\hat{\mu} = 1.006$, with a standard error small enough to reject the hypothesis that the exponent is 1. The total number of calls to the routine to find the nearest neighbor of a vertex had an exponent $\hat{\mu} = 2.013$ for the L_2 norm instances and $\hat{\mu} = 1.929$ for the L_∞ instances. The running time of the implementation is highly correlated to the number of calls to this routine. We presume this fact leads to a running time of $\Theta(n^\mu \log n)$ for our implementation. Modelling the running time of Applegate and Cook's code as $\lambda n^\mu \epsilon$ gave an estimate of $\hat{\mu} = 2.29$ for the L_2 instances and $\hat{\mu} = 2.41$ for the L_∞ instances. We note that the variances of the running times for both algorithms increased significantly with n, making it difficult to make intelligible estimates of the asymptotic running times.

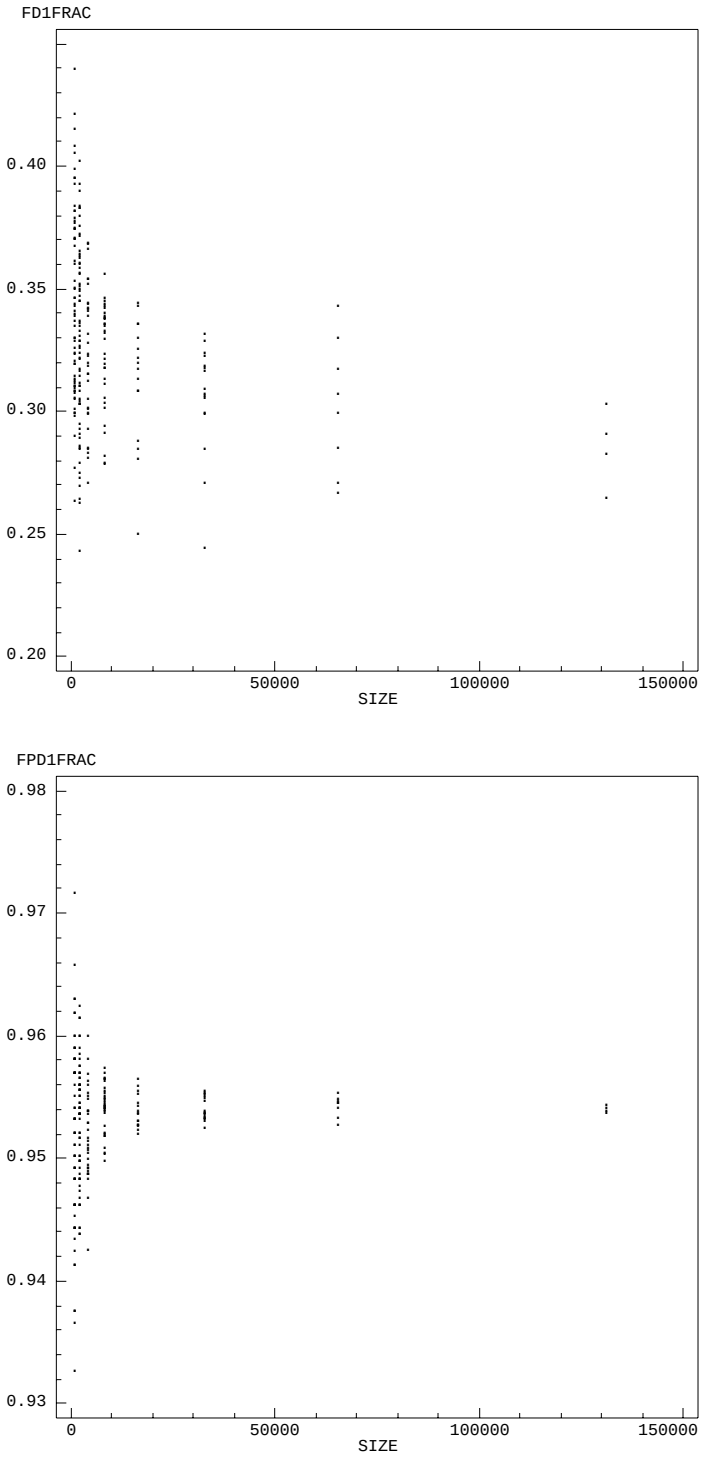

Figure 8-4: Scatterplots of the fraction of vertices of degree 1 in F versus n (top), and the fraction of vertices of degree 1 in F' versus n (bottom).

8.4 Discussion

Empirically, our approximation algorithm seems to deliver perfect matchings which are very close to optimal. As is shown in the tables, the algorithm was never more than 4% away from the optimal solution or 7% away from the lower bound in any of the over 1,400 experiments. One can also see that the algorithm was closer to optimal on structured instances than on random instances. As with any computational study, one might argue that the observed behavior is dependent on the classes of instances being considered. However, in this case, the knowledge of the worst-case performance gives us some peace of mind: our algorithm will never perform embarassingly poorly.

Our study to this point does not yet answer the question of whether the approximation algorithm is a practical alternative to a very good implementation of an exact algorithm or to other heuristics for the matching problem. Our implementation was only faster than Applegate and Cook's code on large random instances; it was slightly slower on small random and structured instances and usually significantly slower on large structured instances. Although Applegate and Cook's code seems to be an exceptionally good implementation of Edmonds' matching algorithm, one would still want an implementation of the approximation algorithm that consistently beat the best exact algorithm by a substantial margin before one could call the algorithm practical. Thus more work needs to be done on the algorithm's implementation. The best implementations of exact matching algorithms, including the Applegate-Cook implementation, usually solve the instance on a sparse subgraph first, then correct the solution afterwards; it might be possible to do something similar with the approximation algorithm.

We should observe that the behavior of Approx-Proper-0-1 for the Euclidean minimum-weight perfect matching problem does not necessarily imply that Approx-Proper will obtain near-optimal solutions for other proper functions. Luckily, the design of Approx-Proper-0-1 should allow us to study other proper functions with range $\{0,1\}$ without much difficulty, by simply changing the weak oracle for f. Researchers at Bellcore are planning to implement Approx-Proper for the survivable network design problem for possible inclusion in their network design software. These further studies should tell us more about

the practicality of Approx-Proper in terms of its running time and the quality of solutions produced.

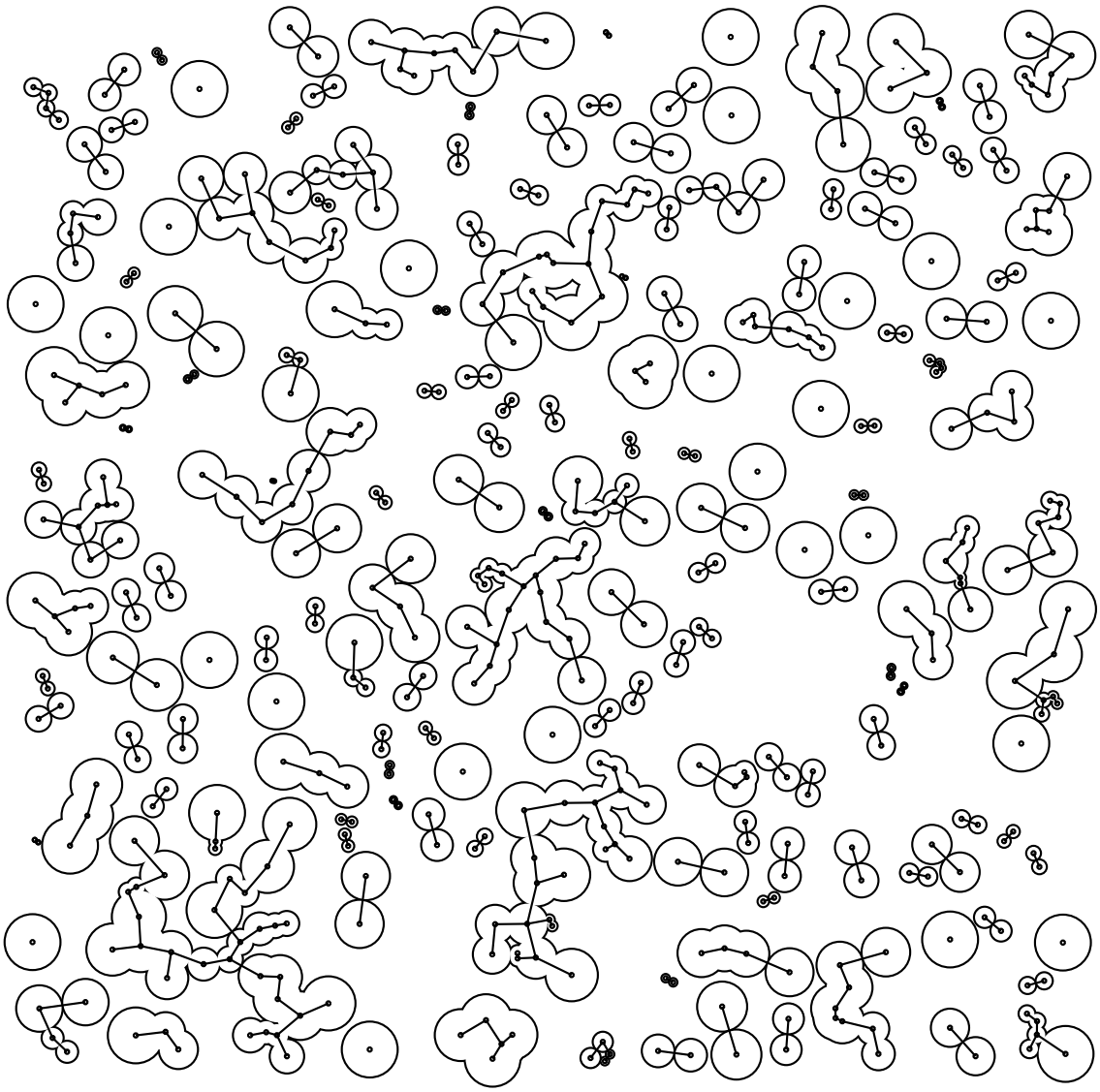

Figure 8-5: Snapshot of the algorithm working on a random instance of 500 vertices.

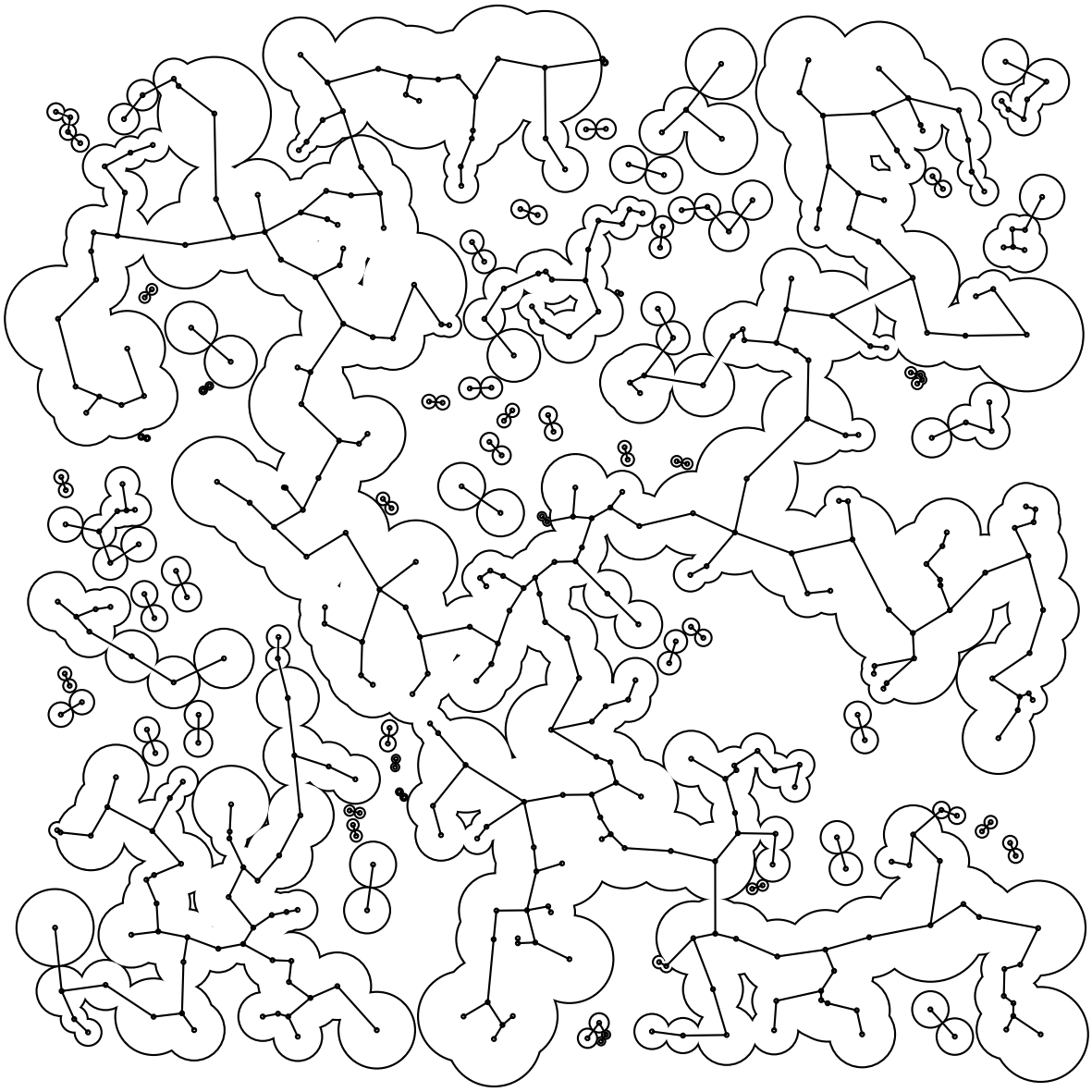

Figure 8-6: The forest F for the random instance of 500 vertices.

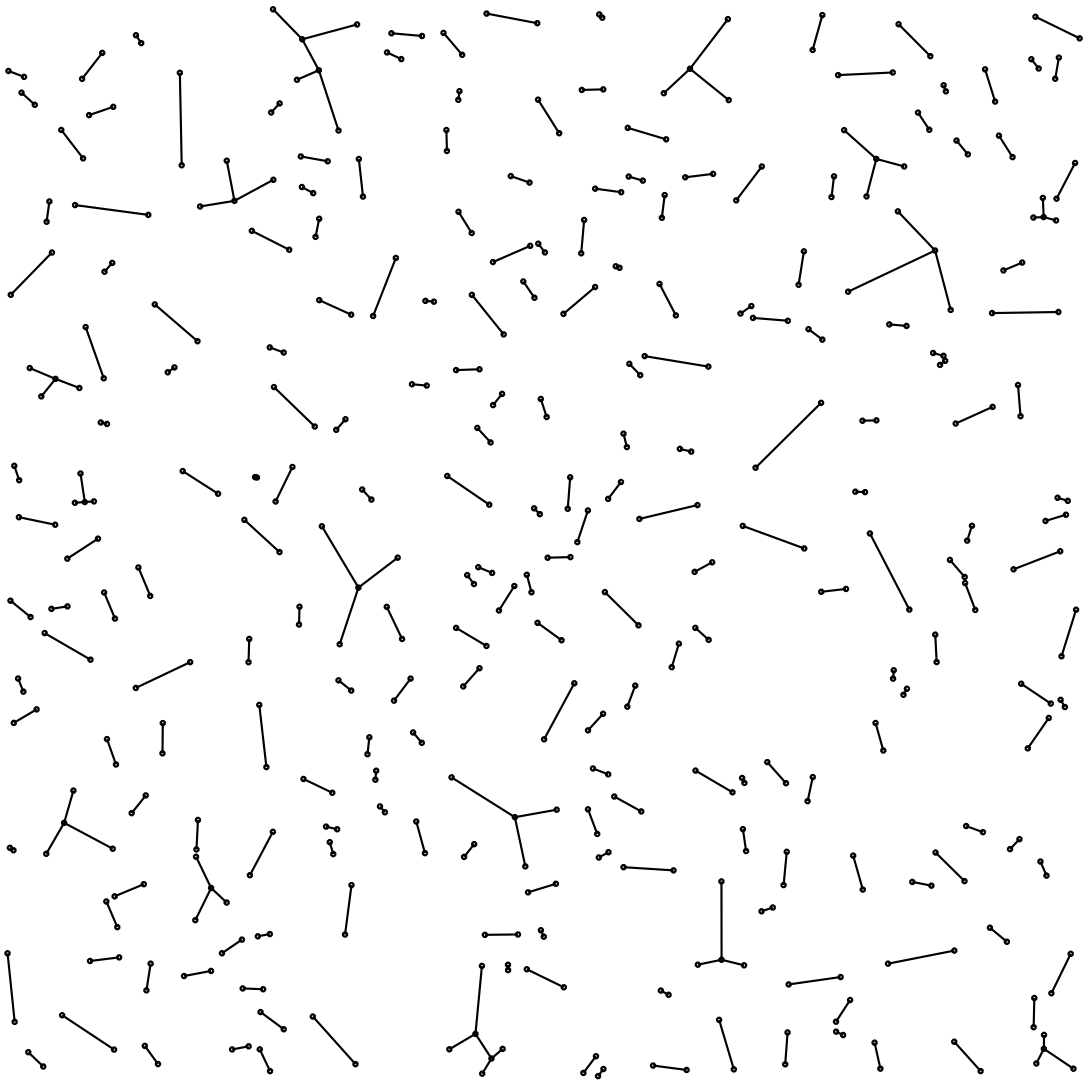

Figure 8-7: The forest F' for the random instance of 500 vertices.

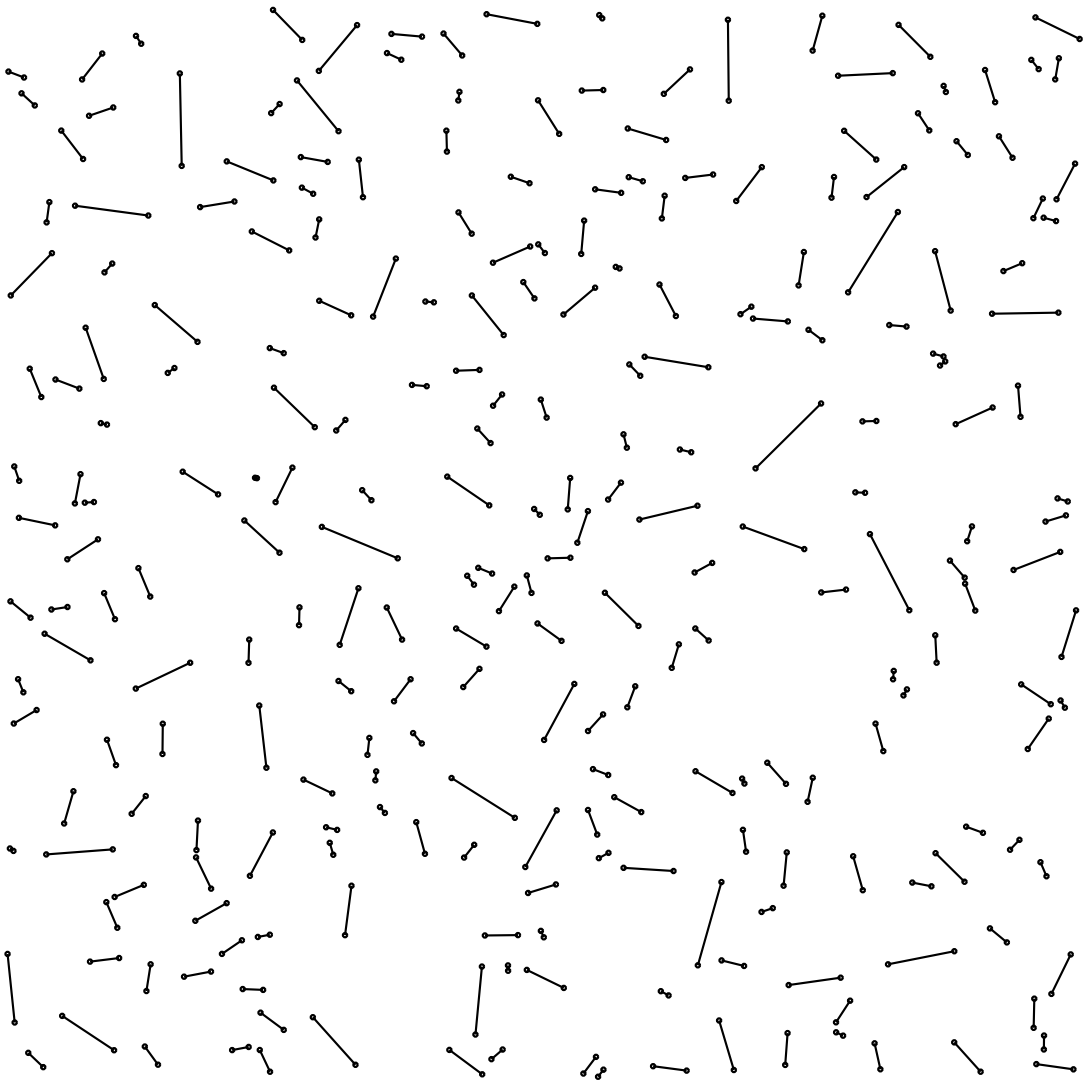

Figure 8-8: Final matching produced by the algorithm for the random instance of 500 vertices.

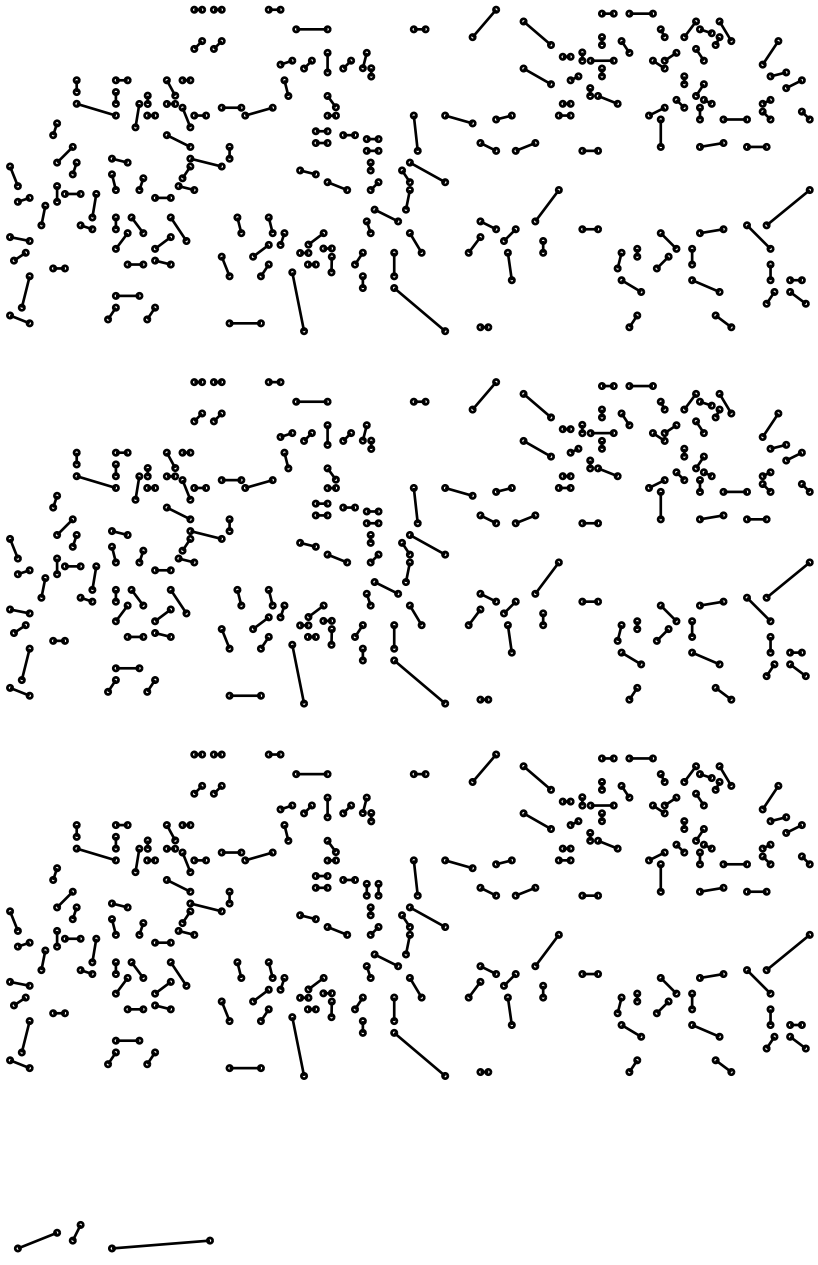

Figure 8-9: The approximation algorithm's solution to the TSPLIB r1002 problem.

CHAPTER 9

Conclusion

The previous chapters have given several different algorithms for a wide variety of graph problems, and so it is important for us to step back and examine the common technique employed by all of the algorithms. Most of the problems involve finding a minimum-cost set of edges that cover each coboundary $\delta(S)$ with some number or capacitated amount $f(S)$ of edges. The first step of our technique reduces these problems into a sequence of subproblems. In each subproblem, we try to find a minimum-cost set of edges that cover each coboundary $\delta(S)$ from a specified collection of sets S at least once. Given a current partial solution E' to the overall problem, we specify this collection as the sets of maximum deficiency $f(S) - |\delta_{E'}(S)|$; that is, those cuts $\delta(S)$ which are currently farthest from the necessary number (or capacity) of edges.

A collection of sets specified in this way often has the interesting property that it is uncrossable; that is, if $h(S) = 1$ for sets S in the collection and $h(S) = 0$ otherwise, then h is uncrossable. We then use the primal-dual methodology to find an approximate solution for the subproblem of covering the collection of sets. In each iteration of the method, we increase the dual variables associated with the minimal violated sets until some primal constraint becomes tight. We add the associated primal variable to our solution, and continue until primal feasibility is reached. By removing unnecessary parts of the primal solution, we can then prove a "total degree" inequality on each dual increase, and thus show

that the primal solution is not too far away in value from the dual solution, implying that it is not too far away from the optimal value.

We have seen that a remarkable number of problems can be approximated using this single framework, from minimum-cost spanning trees to survivable network design problems, from shortest path problems to minimum-cost k-vertex-connected subgraphs. It seems natural to expect that the technique will be extended to still more problems. This might happen by the implementation of the strong oracle for particular weakly supermodular or uncrossable edge-covering problems of interest, or via applications of the technique to related problems, as with the prize-collecting algorithms. In any case, we think that this technique will become a useful part of the algorithm designer's toolkit.

Another consequence of this work is that it highlights the importance of the primal-dual method in algorithm design, particularly for creating approximation algorithms. Before the line of work started in Agrawal, Klein, and Ravi [2] and expanded in this thesis, the primal-dual method for approximation algorithms had been applied mainly to the vertex cover and general set cover problems. Here we see that a particular means of applying the primal-dual method gives good approximation algorithms for edge-covering problems, which form a subclass of the general set cover problem. We believe that the primal-dual method will continue to be useful in designing algorithms for covering problems. It is an interesting open question as to whether it can be applied to other non-covering problems.

Furthermore, the concepts of uncrossable, weakly supermodular, and proper functions seem useful in and of themselves. Ravi, Ragavachari, and Klein [106] and Ravi, Marathe, Ravi, Rosencrantz, and Hunt [105] have shown how to extend work of Fürer and Ragavachari [42] on finding minimum-degree spanning trees to finding minimum-degree connected networks specified by proper functions with $f_{\max} = 1$. Ravi and Klein [104] have shown how a concept akin to the toughness of a graph can be approximated for graphs defined by proper functions. These classes of functions may continue to find use in graph problems other than edge-covering problems.

Many interesting problems present themselves. Is it possible to design a single phase algorithm for some class of edge-covering problems? Can the performance guarantee for proper edge-covering problems be improved to a constant? Is it possible to prove, using

recent lower bound techniques (see [6]), that essentially no such improvement is possible? This seems like an especially interesting direction to try, given the results of Lund and Yannakakis [89] and Bellare, Goldwasser, Lund, and Russell [12] showing that the set cover problem cannot have an approximation algorithm with performance guarantee less than $\frac{1}{8}\log_2 n$ unless $NP \subseteq DTIME(n^{\log\log n})$.

Is it possible to analyze the algorithms probabilistically? For example, it would be interesting to show theoretically that the value of solutions produced by the approximation algorithm for matching does converge almost surely to $\beta\sqrt{n}$ on random Euclidean instances, or to show some theoretical basis for the structural properties of the algorithm on such instances. This would seem to be a difficult task, however.

Can the algorithms be extended to classes beyond uncrossable and weakly supermodular edge-covering problems? This cannot be done in general using our current techniques, because there exist non-uncrossable functions for which the relative gap between the value of an optimal dual solution and an optimal integral solution is large. Consider the problem for which $h(S) = 1$ if a $u \in S$ and $|S| \leq k+1$, and $h(S) = 0$ otherwise. Let $G = (V, E)$ be a star graph, with $V = \{u, v_1, \ldots, v_k\}$, $E = \{(u, v_1), \ldots (u, v_{2k})\}$, and a cost of 1 on all edges. Then the solution $x_e = \frac{1}{k}$ for all $e \in E$ is a feasible solution of value 2 for the linear programming relaxation of (IP), while any $k+1$ edges form an optimal integer solution. Thus any approximation algorithm for a class of problems which includes these functions h will not have a constant performance guarantee if its analysis relies on bounding the gap between an integer solution and a dual lower bound.

The key feature of uncrossable functions is that any dual solution can be made laminar with no decrease in value; that is, the solution can be transformed so that the family of sets S for which $y_S > 0$ is laminar. This property characterizes the uncrossable functions. The theory of laminar sets and supermodular and submodular functions looms large behind the design and analysis of the algorithms presented here, and approximating edge-covering problems without these properties will likely require significantly different techniques.

Bibliography

[1] M. Aggarwal and N. Garg. A scaling technique for better network design. Submitted to the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, 1993.

[2] A. Agrawal, P. Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized Steiner problem on networks. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 134–144, 1991.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[4] D. Applegate. Personal communication, 1993.

[5] D. Applegate and W. Cook. Solving large-scale matching problems. In *DIMACS implementation challenge workshop: Algorithms for network flow and matching*. DIMACS Technical Report 92-4, 1992.

[6] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 14–23, 1992.

[7] S. Arora and S. Safra. Probabilistic checking of proofs; a new characterization of NP. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 2–13, 1992.

[8] D. Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13:475–493, 1983.

[9] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19:621–636, 1989.

[10] R. Bar-Yehuda and S. Even. A linear time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2:198–203, 1981.

[11] J. Beardwood, J. Halton, and J. Hammersley. The shortest path through many points. *Proceedings of the Cambridge Philosophical Society*, 55:299–327, 1959.

[12] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficeint probabilistically checkable proofs and applications to approximation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 294–304, 1993.

[13] J. L. Bentley. K-d trees for semi-dynamic point sets. In *Proceedings of the 6th Annual ACM Symposium on Computational Geometry*, pages 187–197, 1990.

[14] P. Berman and V. Ramaiyer. Improved approximations for the Steiner tree problem. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 325–334, 1992.

[15] D. Bienstock and N. Diaz. Blocking small cuts in a network, and related problems. *SIAM Journal on Computing*, 22:482–499, 1993.

[16] D. Bienstock, M. X. Goemans, D. Simchi-Levi, and D. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 59:413–420, 1993.

[17] N. Christofides. Worst case analysis of a new heuristic for the traveling salesman problem. Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[18] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[19] V. Chvátal. *Linear Programming*. W.H. Freeman and Company, New York, NY, 1983.

[20] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[22] G. Cornuéjols and W. Pulleyblank. A matching problem with side constraints. *Discrete Mathematics*, 29:135–159, 1980.

[23] W. Cunningham and A. M. III. A primal algorithm for optimum matching. *Mathematical Programming Study*, 8:50–72, 1978.

[24] U. Derigs. A shortest augmenting path method for solving minimal perfect matching problems. *Networks*, 11:379–390, 1981.

[25] U. Derigs. Solving large-scale matching problems efficiently: a new primal matching approach. *Networks*, 16:1–16, 1986.

[26] U. Derigs. Solving non-bipartite matching problems via shortest path techniques. *Annals of Operations Research*, 13:225–261, 1988.

[27] U. Derigs and A. Metz. On the use of optimal fractional matchings for solving the (integer) matching problem. *Computing*, 36:263–270, 1986.

[28] U. Derigs and A. Metz. Solving (large scale) matching problems combinatorially. *Mathematical Programming*, 50:113–121, 1991.

[29] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[30] G. Dobson. Worst-case analysis of greedy heuristics for integer programming with non-negative data. *Mathematics of Operations Research*, 7:515–531, 1982.

[31] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards B*, 69B:125–130, 1965.

[32] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[33] J. Edmonds and E. Johnson. Matching: A well-solved class of integer linear programs. In R. Guy et al., editors, *Proceedings of the Calgary International Conference on Combinatorial Structures and Their Applications*, pages 82–92. Gordon and Breach, 1970.

[34] J. Edmonds and E. L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, 5:88–124, 1973.

[35] P. Elias, A. Feinstein, and C. E. Shannon. Note on maximum flow through a network. *IRE Transactions on Information Theory*, IT-2:117–119, 1956.

[36] K. P. Eswaran and R. E. Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5:653–665, 1976.

[37] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is Almost NP-complete. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 2–12, 1991.

[38] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[39] G. N. Frederickson and J. Ja'Ja'. Approximation algorithms for several graph augmentation problems. *SIAM Journal on Computing*, 10:270–283, 1981.

[40] M. Fredman, D. Johnson, L. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 145–154, 1993.

[41] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3:209–226, 1977.

[42] M. Fürer and B. Ragavachari. Approximating the minimum degree spanning tree to within one from the optimal degree. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 317–324, 1992.

[43] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 90–100, 1985.

[44] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 434–443, 1990.

[45] H. N. Gabow, Z. Galil, and T. H. Spencer. Efficient implementation of graph algorithms using contraction. *Journal of the ACM*, 36:540–572, 1989.

[46] H. N. Gabow, M. X. Goemans, and D. P. Williamson. An efficient approximation algorithm for the survivable network design problem. In *Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization*, pages 57–74, 1993.

[47] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph matching problems. Technical Report CU-CS-432-89, University of Colorado, Boulder, 1989.

[48] M. Garey, R. Graham, and D. Johnson. The complexity of computing Steiner minimal trees. *SIAM Journal of Applied Mathematics*, 32:835–859, 1977.

[49] N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 698–707, 1993.

[50] M. Goemans, A. Goldberg, S. Plotkin, D. Shmoys, E. Tardos, and D. Williamson. Improved approximation algorithms for network design problems. Submitted to the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, 1993.

[51] M. X. Goemans. *Analysis of linear programming relaxations for a class of connectivity problems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1990. Also appears as MIT Operations Research Center Technical Report OR 233-90.

[52] M. X. Goemans and D. J. Bertsimas. Survivable networks, linear programming relaxations and the parsimonious property. Technical Report OR 216-90, MIT Operations Research Center, 1990. To appear in *Math. Prog.*, Vol. 60.

[53] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 307–316, 1992.

[54] R. Gomory and T. Hu. Multi-terminal network flows. *SIAM Journal of Applied Mathematics*, 9:551–570, 1961.

[55] R. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[56] M. Grötschel and O. Holland. Solving matching problems with linear programming. *Mathematical Programming*, 33:243–259, 1985.

[57] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithm and Combinatorial Optimization*. Springer-Verlag, Berlin, 1988.

[58] M. Grötschel, C. L. Monma, and M. Stoer. Design of survivable networks. In *Handbook in Operations Research and Management Science*. 1993. To appear.

[59] D. Gusfield and D. Naor. Efficient algorithms for generalized cut trees. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 422–433, 1990.

[60] N. G. Hall and D. S. Hochbaum. A fast approximation algorithm for the multicovering problem. *Discrete Applied Mathematics*, 15:35–40, 1986.

[61] D. S. Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on Computing*, 11:555–556, 1982.

[62] A. J. Hoffman and P. Wolfe. History. In E. Lawler, J. Lenstra, A. R. Kan, and D. Shmoys, editors, *The Traveling Salesman Problem*, chapter 1. John Wiley and Sons, Chichester, 1985.

[63] C. Imielinska, B. Kalantari, and L. Khachiyan. A greedy heuristic for a minimum weight forest problem. Submitted to *Oper. Res. Lett.*, 1993.

[64] M. Iri, K. Murota, and S. Matsui. Heuristics for planar minimum-weight perfect matchings. *Networks*, 13:67–92, 1983.

[65] P. Jaillet. Rates of convergence of quasi-additive smooth Euclidean functionals and application to combinatorial optimization problems. *Mathematics of Operations Research*, 17:964–980, 1992.

[66] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[67] M. Jünger and W. Pulleyblank. New primal and dual matching heuristics. Research Report 91.105, Universität zu Köln, 1991.

[68] R. M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, NY, 1972.

[69] R. M. Karp and J. Steele. Probabilistic analysis of heuristics. In E. Lawler, J. Lenstra, A. R. Kan, and D. Shmoys, editors, *The Traveling Salesman Problem*, chapter 6. John Wiley and Sons, Chichester, 1985.

[70] S. Khuller and R. Thurimella. Approximation algorithms for graph augmentation. *Journal of Algorithms*, 14:214–225, 1993.

[71] S. Khuller and U. Vishkin. Biconnectivity approximations and graph carvings. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 759–770, 1992.

[72] V. King, S. Rao, and R. Tarjan. A faster deterministic maximum flow algorithm. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 157–164, 1992.

[73] P. Klein, A. Agrawal, R. Ravi, and S. Rao. Approximation through multicommodity flow. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 726–737, 1990.

[74] P. Klein and R. Ravi. When cycles collapse: A general approximation technique for constrained two-connectivity problems. In *Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization*, pages 39–55, 1993. Also appears as Brown University Technical Report CS-92-30.

[75] P. N. Klein. A data structure for bicategories, with application to speeding up an approximation algorithm. Unpublished manuscript, 1993.

[76] P. N. Klein, S. Rao, A. Agrawal, and R. Ravi. An approximate max-flow min-cut relation for multicommodity flow, with applications. To appear in *Combinatorica*, 1993.

[77] J. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[78] G. Laporte. Location-routing problmes. In B. L. Golden and A. A. Assad, editors, *Vehicle routing: Methods and studies*, pages 163–197. North-Holland, Amsterdam, 1988.

[79] G. Laporte, Y. Nobert, and P. Pelletier. Hamiltonian location problems. *European Journal of Operations Research*, 12:82–89, 1983.

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

[81] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[82] R. Lessard, J. Rousseau, and M. Minoux. A new algorithm for general matching problems using network flow subproblems. *Networks*, 19:459–479, 1989.

[83] L. Levin. Universal sequential search problems. *Problems of Information Trans. 9*, 3:265–266, 1973.

[84] C.-L. Li, S. T. McCormick, and D. Simchi-Levi. The point-to-point delivery and connection problems: Complexity and algorithms. *Discrete Applied Mathematics*, 36:267–292, 1992.

[85] S. Lin and B. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.

[86] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[87] L. Lovász. Submodular functions and convexity. In A. Bachem, M. Grötschel, and B. Korte, editors, *Mathematical Programming: The State of the Art*, pages 235–257. Springer-Verlag, 1983.

[88] L. Lovász and M. Plummer. *Matching Theory*. North-Holland, Amsterdam, 1986.

[89] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 286–293, 1993.

[90] T. L. Magnanti and R. T. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18:1–55, 1984.

[91] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.

[92] K. Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.

[93] D. Naor, D. Gusfield, and C. Martel. A fast algorithm for optimally increasing the edge-connectivity. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 698–707, 1990.

[94] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, NY, 1988.

[95] T. Nicholson. Finding the shortest route between two points in a network. *Computer Journal*, 9:275–280, 1966.

[96] M. W. Padberg and M. Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7:67–80, 1982.

[97] C. H. Papadimitriou. The probabilistic analysis of matching heuristics. In *Proceedings of the 15th Annual Allerton Conference on Communication, Control, and Computing*, pages 368–378, 1978.

[98] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[99] S. Phillips and J. Westbrook. Online load balancing and network flow. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 402–411, 1993.

[100] J. Picard and M. Queyranne. On the structure of all minimum cuts in a network and applications. *Mathematical Programming Study*, 13:8–16, 1980.

[101] D. A. Plaisted. Heuristic matching for graphs satisfying the triangle inequality. *Journal of Algorithms*, 5:163–179, 1984.

[102] S. A. Plotkin, D. B. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 495–504, 1991.

[103] S. A. Plotkin and E. Tardos. Improved bounds on the max-flow min-cut ratio for multicommodity flows. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 691–697, 1993.

[104] R. Ravi and P. Klein. Approximation through uncrossing. Unpublished manuscript, 1992.

[105] R. Ravi, M. Marathe, S. Ravi, D. Rosenkrantz, and H. H. III. Many birds with one stone: Multi-objective approximation algorithms. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 438–447, 1993.

[106] R. Ravi, B. Ragavachari, and P. Klein. Approximation through local optimality: Designing networks with small degree. In the *Proc. of the 12th Conf. on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, 1992.

[107] R. Ravi and D. P. Williamson. An approximation algorithm for the minimum-cost k-vertex-connected subgraph problem. Unpublished manuscript, 1993.

[108] G. Reinelt. TSPLIB – A traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991.

[109] E. Reingold and K. Supowit. Probabilistic analysis of divide-and-conquer heuristics for minimum weighted Euclidean matching. *Networks*, 13:49–66, 1983.

[110] E. M. Reingold and R. E. Tarjan. On a greedy heuristic for complete matching. *SIAM Journal on Computing*, 10:676–681, 1981.

[111] W. T. Rhee and M. Talagrand. A sharp deviation inequality for the stochastic traveling salesman problem. *Annals of Probability*, 17:1–8, 1989.

[112] M. Sipser. The history and status of the P versus NP question. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 603–618, 1992.

[113] J. Steele. Notes on probabilistic and worst case analyses of classical problems of combinatorial optimization in Euclidean space. Lectures given at Cornell University, 1987.

[114] J. M. Steele. Complete convergence of short paths and Karp's algorithm for the TSP. *Mathematics of Operations Research*, 6:374–378, 1981.

[115] J. M. Steele. Subadditive Euclidean functionals and nonlinear growth in geometric probability. *Annals of Probability*, 9:365–376, 1981.

[116] J. M. Steele, L. A. Shepp, and W. F. Eddy. On the number of leaves of a Euclidean minimal spanning tree. *Journal of Applied Probability*, 24:809–826, 1987.

[117] K. Steiglitz, P. Weiner, and D. Kleitman. The design of minimal cost survivable networks. *IEEE Trans. Cir. Theory*, CT-16:455–460, 1969.

[118] G. Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, San Diego, CA, Third edition, 1988.

[119] K. J. Supowit, D. A. Plaisted, and E. M. Reingold. Heuristics for weighted perfect matching. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 398–419, 1980.

[120] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[121] P. Vaidya. Personal communication, 1991.

[122] P. M. Vaidya. Geometry helps in matching. *SIAM Journal on Computing*, 18:1202–1255, 1989.

[123] P. M. Vaidya. A new algorithm for minimizing convex functions over convex sets. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 338–343, 1989.

[124] O. Vornberger. *Complexity of path problems in graphs*. PhD thesis, Universität-GH-Paderborn, 1979.

[125] M. Weber and T. Liebling. Euclidean matchings problems and the metropolis algorithm. *Zeitschrift für Operations Research*, 30:A85–A110, 1986.

[126] D. P. Williamson and M. X. Goemans. Computational experience with an approximation algorithm on large-scale Euclidean matching instances. Submitted to the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, 1993.

[127] D. P. Williamson, M. X. Goemans, M. Mihail, and V. V. Vazirani. A primal-dual approximation algorithm for generalized Steiner network problems. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 708–717, 1993.

[128] P. Winter. Steiner problem in networks: a survey. *Networks*, 17:129–167, 1987.

[129] L. A. Wolsey. Heuristic analysis, linear programming and branch and bound. *Mathematical Programming Study*, 13:121–134, 1980.

[130] A. Zelikovsky. An 11/6-approximation algorithm for the network Steiner problem. *Algorithmica*, 9:463–470, 1993.

[131] A. Zelikovsky. An approximation algorithm for weighted k-polymatroids and the Steiner tree problem in graphs. In *Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization*, pages 89–98, 1993.