Approximation Algorithms for Multicommodity Flow and Shop Scheduling Problems

by

Clifford Stein

B.S.E., Electrical Engineering and Computer Science
Princeton University
(1987)
S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1989)

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 1992

Signature of Author____
Department of Electrical Engineering and Computer Science
August 10, 1992

Certified by____
David B. Shmoys
Associate Professor of Industrial Engineering and Operations Research,
Cornell University
Thesis Supervisor

Accepted by____
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Approximation Algorithms for Multicommodity Flow and Shop Scheduling Problems

by

Clifford Stein

Submitted to the Department of Electrical Engineering and Computer Science
on August 10, 1992,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

In this thesis, we give efficient approximation algorithms for two classical combinatorial optimization problems: *multicommodity flow problems* and *shop scheduling problems.* The algorithms we develop for these problems yield solutions that are not necessarily optimal, but come with a provable performance guarantee; that is, we can guarantee that the solution found is within a certain percentage of the optimal solution. This type of algorithm is known as an *approximation algorithm.* Our results show that by allowing a small error in the solution of a problem, it is often possible to gain a significant reduction in the running time of an algorithm for that problem.

In Chapter 2, we study the multicommodity flow problem. The multicommodity flow problem involves simultaneously shipping several different commodities from their respective sources to their sinks in a single network so that the total amount of flow going through each edge is no more than its capacity. Associated with each commodity is a demand, which is the amount of that commodity that we wish to ship. Given a multicommodity flow problem, one often wants to know if there is a *feasible* flow, i.e., if it is possible to find a flow that satisfies the demands and obeys the capacity constraints. More generally, we might wish to know the maximum percentage z such that at least z percent of each demand can be shipped without violating the capacity constraints. The latter problem is known as the *concurrent flow problem.* Our algorithms are approximation algorithms that find ϵ-optimal solutions to the concurrent flow problem, that is, solutions in which z is within a $(1-\epsilon)$ factor of the minimum possible value. In particular, we show that for any $\epsilon > 0$, an ϵ-optimal solution to the n-node, m-edge, k-commodity concurrent flow problem can be found by a randomized algorithm in $O(\epsilon^{-3}kmn \log k \log^3 n)$ time and by a deterministic algorithm in $O(\epsilon^{-2}k^2mn \log k \log^3 n)$ time.

Our expected running time is the same (up to polylog factors) as the time needed to compute k maximum-flows, thus giving the surprising result that approximately computing a k-commodity concurrent flow is about as difficult as exactly computing k single-commodity maximum flows. In fact, we formally prove that a k-commodity concurrent flow problem can be approximately solved by approximately solving $O(k \log k \log n)$ minimum-cost flow problems.

The multicommodity flow problem has several important applications. Many classical problems in Operations Research can be phrased as multicommodity flow problems, including: telecommunications problems, import-export problems, freight transport and scheduling, net-

work design, freight assignment in the less-that-truckload trucking industry, traffic planning, and busing students to schools. Multicommodity flow can also be used to find good separators for graphs, yielding divide-and-conquer algorithms for several NP-hard graph problems. In particular, the results in this thesis can be used to give the fastest polylogarithmic approximations to several problems including: VLSI channel routing, minimum cut linear arrangement, minimum area layout, $\sqrt{2}$-bifurcators of a graph, minimum feedback-arc set, graph embedding problems, chordalization of a graph, register sufficiency, minimum deletion of clauses in a $2CNF \equiv$ formula, via minimization, and the edge-deletion graph bipartization problems. These problems will be discussed in Chapter 3. In addition we will show how, in some cases, our algorithms can be adapted to find *integral* solutions. The ability to do so is significant, since the integral multicommodity flow problem is likely to be more difficult than the problem of finding an optimal flow that is not necessarily integral: the former problem is NP-hard whereas the latter is solvable in polynomial time via linear programming.

Not only do our algorithms have provably efficient running times, but they perform well in practice. In Chapter 4 we discuss an implementation of one variant of the algorithm presented in Chapter 2. The results, while preliminary, are rather encouraging. For large problems our implementation significantly outperforms a good simplex-based linear programming code. In fact, we have been able to solve problems that are larger than those that can be solved by good simplex-based codes. In particular, we are able to solve problems in which there are a large number of commodities.

In Chapter 5, we turn to the problem of shop scheduling. We give the first polylogarithmic approximation algorithms for the job-shop problem, flow-shop problem, and several extensions. Our algorithms are randomized and combine techniques from two seemingly disparate fields of study: vector-sum theorems and packet routing algorithms.

In Chapter 6, we show how to make the shop scheduling algorithms deterministic. Our algorithm makes use of some recent extensions of our multicommodity flow techniques and unifies many of the ideas in this thesis, since it is necessary to find an approximately optimal integral solution to a generalized version of the multicommodity flow problem. The algorithms we use are closely related to those used for the multicommodity flow problem.

Keywords: Multicommodity Flow, Scheduling, Combinatorial Optimization, Network Algorithms, Approximation Algorithms, Randomized Algorithms.

Thesis Supervisor: David B. Shmoys
Title: Associate Professor of Industrial Engineering and Operations Research, Cornell University

Acknowledgments

I have been lucky to have had so many wonderful teachers, colleagues and friends. David Shmoys is a wonderful advisor. Even though he left for Cornell three years ago, he made sure that the physical distance did not present any problems, save my increased use of dramamine. He has been generous with his time and and willingness to dispense sound advice, on both technical and non-technical matters. I have learned much about research, scholarship and integrity from David.

I have also learned a great deal from Éva Tardos, especially throughout collaborations over the past few years. I appreciate the "advisor-like" concern she has shown, and her willingness to share her ideas. I also thank both David and Éva for naming their daughter after my wife.

One advantage of having a long-distance advisor was that it gave me the opportunity to work closely with other people. Early in my graduate career, I worked closely with Philip Klein. The first set of results on multicommodity flow was done in collaboration with Philip. He has taught me a great deal about research, writing and determination. I also thank him for the conversations about the implementation work. I am also grateful to Jim Orlin for allowing me to be part of his research group and have enjoyed my collaborations with him. I also thank him for being a reader on my thesis committee. Thanks to Charles Leiserson for being a reader on my thesis committee and for clearing away the many beaurocratic hurdles that arose. From Charles' careful reading and copious comments, I learned more about the art of scientific writing than I had imagined was possible. I also thank Tom Leighton for his support and advice and take this opportunity to state publicly that he is a better shortstop than I have previously given him credit for. I thank Jim, Charles and Tom for recognizing the potential problems of having a long-distance advisor, and for filling various aspects of that role. In particular, they were all generous in suggesting interesting problems to work on.

I've also enjoyed my collaboration with Joel Wein. Joel first got me interested in shop scheduling and has been a constant source of interesting problems and good ideas on how to solve them. I'm also grateful for the information about the Knicks with which he has provided me.

Most of the implementation work was done during the summer I spent at Bell Labs. I learned a great deal working with Peter Shor and Tishya Leong, and I thank David Johnson for his support and advice on the project.

I can't imagine a better academic environment that the Theory Group at MIT. I've learned a great deal from my fellow students and I'd like to thank all of them, particularly Avrim Blum, Tom Cormen, Michel Goemans, Mark Hanson, Bruce Maggs, James Park, Cindy Phillips, Satish Rao, Eric Schwabe, Joel Wein and David Williamson. I've also enjoyed the opportunity to participate in a variety of intramural sports and I thank the members of the Running Time softball team, Execution Time hockey team and the Hoopsorts basketball team for providing a great deal of exercise, entertainment and friendship. I have also been fortunate to share an office with, at various times, Tom Cormen, Sally Goldman, Lalita Jategaonkar, Bruce Maggs and Eric Schwabe. I'd particularly like to thank Tom for his help with the Macintosh and for giving me the advice, "When the ball is hit over you head, turn around, run hard to where it will be, then and only then do you stick your glove out." The support staff has also helped to create a friendly environment. Special thanks to Be Hubbard for everything she does, William Ang for his computer help, David Jones for Latex and Postscript help, and Arlene Benford, Julie Sweedler, Denise Sergent-Leventhal and Becky Bisbee for all their help.

I'm also grateful to Ken Steiglitz and Bob Tarjan for getting me interested in combinatorial optimization and theoretical computer science, and to Esther Rifkin, Ron Mezzadri and Miriam Waks for early encouragement in the fields of mathematics and computer science.

I acknowledge the support of a Bell Labs Graduate Fellowship, a GE Graduate Fellowship, Air Force Contracts AFOSR-86-0078 and AFOSR-89-0271, DARPA contract N00014-89-J-1988, and the National Science Foundation under a PYI awarded to David Shmoys, with matching support from Sun and UPS.

Not only have I gained a great deal of knowledge in the past five years, but more importantly, I have also gained a wife and companion, Rebecca Ivry. I thank Rebecca for standing by me and providing encouragement, support, love and understanding throughout. I am also grateful to my parents Irene and Ira Stein and my sister Amy Stein, for their constant support and love.

Contents

Chapter 1

Introduction

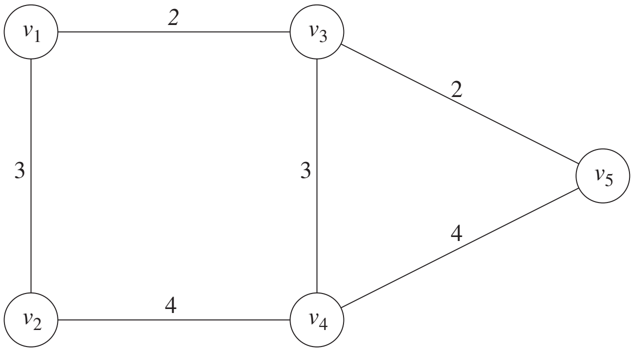
Given a particular combinatorial optimization problem, there are many possible approaches to finding a solution. Perhaps the most common strategy is to rely on a general purpose method, such as linear programming, that solves a large class of problems. The advantages of using such a method are clear – a single computer program that implements this method can solve many different problems. The only effort involved in solving a new problem in this class is to express it in the proper form. However, general purpose methods have their disadvantages as well. By expressing a particular problem as an instance of a more general problem, one may ignore some structure that makes the original problem easier to solve.

In this thesis, we develop algorithms that exploit the particular combinatorial structure of the problem at hand. This approach leads to algorithms that are faster than previously known ones, and which are able to solve larger sized instances.

In particular, we give efficient approximation algorithms for two classes of problems: *multi-commodity flow problems* and *shop scheduling problems.* These are basic and classical problems in combinatorial optimization. The algorithms we develop for these problems yield solutions that are not necessarily optimal, but come with a provable performance guarantee; that is, we can guarantee that the solution found is within a certain percentage of the optimal solution. This type of algorithm is known as an *approximation algorithm.* For many applications, an optimal solution is not needed, either because the application can be solved just as easily with an approximate solution, or because the input data itself may only be accurate up to some fixed precision. Our results show that by allowing a small error in the solution of a problem, it is often possible to gain a significant reduction in the running time of an algorithm for that problem.

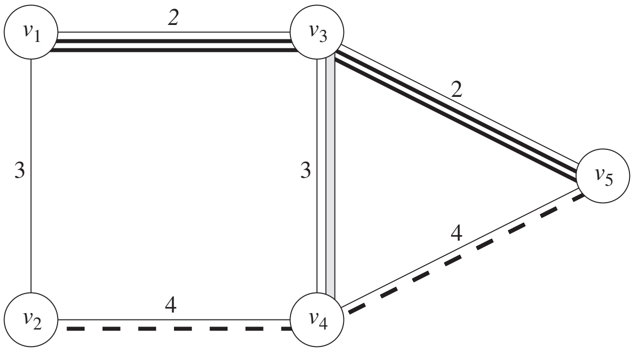
The first problem we study is the *multicommodity flow problem.* The multicommodity flow problem involves simultaneously shipping several different commodities from their respective sources to their sinks in a single network so that the total amount of flow going through each edge is no more than its capacity. Associated with each commodity is a demand, which is the amount of that commodity that we wish to ship. Given a multicommodity flow problem, one often wants to know if there is a *feasible* flow, i.e., if it is possible to find a flow that satisfies the demands and obeys the capacity constraints. More generally, we might wish to know the maximum percentage z such that at least z percent of each demand can be shipped without violating the capacity constraints. The latter problem is known as the *concurrent flow problem*, and is equivalent to the problem of determining the minimum factor by which the capacities can be multiplied so that it is possible to ship 100% of each demand. For our algorithms, it is convenient to state the concurrent flow problem in a different, but equivalent form. We are given a network and a set of commodities. Let the congestion of an edge be the ratio of the total flow on that edge to its capacity. We wish to find a way to route each commodity so that the maximum edge congestion is minimized. We denote the value of the maximum edge congestion by λ and the minimum possible value of λ by λ^*. Our algorithms are approximation algorithms that find ϵ-optimal solutions, that is, solutions in which $\lambda \leq (1+\epsilon)\lambda^*$.

An example of a concurrent flow problem appears in Figure 1.1. The input consists of a network and a specification of the commodities. Each edge is labeled with its capacity. The goal is to find a solution that sends 1 unit of flow between v_2 and v_5, 1 unit of flow between v_3 and v_4 and 2 units of flow between v_1 and v_5. In Figures 1.2 and 1.3, we give two solutions to the concurrent flow problem. In Figure 1.2, it is easy to verify that the demands are all satisfied. Further, the maximum edge congestion $\lambda = 1$, because on all edges the flow is less than or equal to the capacity. In Figure 1.3, the maximum edge congestion $\lambda = \frac{1}{2}$, because on all edges the flow is less than or equal to one half the capacity. As we will prove in Chapter 2, the solution in Figure 1.3 is actually the optimal solution to this problem. Throughout the thesis, we use n, m and k to denote the number of nodes, edges and commodities, we assume that the demands



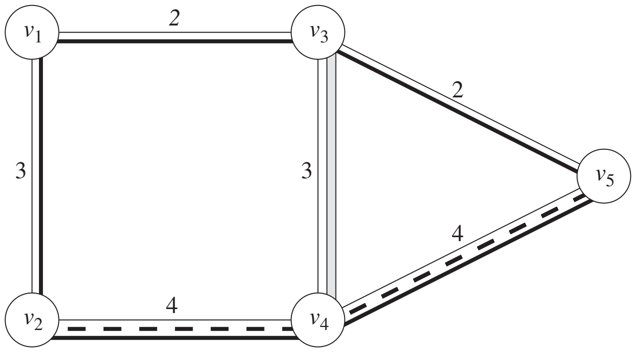
commodity	s	t	demand
1	v_2	v_5	1
2	v_3	v_4	1
3	v_1	v_5	2

Figure 1.1: A sample problem



commodity	s	t	demand	symbol
1	v_2	v_5	1	
2	v_3	v_4	1	
3	v_1	v_5	2	

Figure 1.2: A suboptimal solution



commodity	s	t	demand	symbol
1	v_2	v_5	1	
2	v_3	v_4	1	
3	v_1	v_5	2	

Figure 1.3: An optimal solution

and the capacities are integral, and use D and U to denote the largest demands and capacities, respectively. For the example in Figure 1.1, $n = 5$, $m = 6$, $k = 3$, $D = 2$ and $U = 4$.

In this thesis, we describe the first combinatorial approximation algorithms for the concurrent flow problem. Given any positive ϵ, the algorithms find ϵ-optimal solutions. The running times of the algorithms depend polynomially on ϵ^{-1}, and are significantly better than the running times of previous algorithms when ϵ is a constant. In other words, by trading a small amount of accuracy, we are able to obtain large improvements in the time needed to solve a multicommodity flow problem. As an example of the running times we can achieve, we state one of our results. We define the *simple concurrent flow problem* to be a concurrent flow problem in which each commodity has exactly one source and one sink.

Theorem 1.0.1 For any $\epsilon > 0$, an ϵ-optimal solution for the simple concurrent flow problem can be found by a randomized algorithm in $O(\epsilon^{-3}kmn\log k\log^3 n)$ time and by a deterministic algorithm in $O(\epsilon^{-2}k^2mn\log k\log^3 n)$ time.

Our expected running time is the same (up to polylog factors) as the time needed to compute k maximum-flows, thus giving the surprising result that approximately computing a k-commodity concurrent flow is about as difficult as exactly computing k single-commodity maximum flows. In fact, we formally prove that a k-commodity concurrent flow problem can be approximately solved by approximately solving $O(k \log k \log n)$ minimum-cost flow problems.

The only previously-known algorithms for solving the general concurrent flow problem use linear programming. The concurrent flow problem can be formulated as a linear program in $O(mk)$ variables and $O(nk + m)$ constraints. Any polynomial-time linear programming algorithm can be used to solve the problem optimally. Kapoor and Vaidya [30] gave a method to speed up the matrix inversions involved in Karmarkar-type algorithms for multicommodity flow problems; combining their technique with Vaidya's linear programming algorithm that uses fast matrix multiplication [67] yields a time bound of $O(k^{3.5}n^3m^{.5}\log(nDU))$ for the concurrent flow problem with integer demands and an $O(k^{2.5}n^2m^{.5}\log(n\epsilon^{-1}DU))$ time bound to find an approximate solution. When ϵ is not too small, for example when $\epsilon = \Theta(1)$, the running time of our algorithm is faster for all possible instances of a simple concurrent flow problem.

Before continuing, we emphasize the difference between approximation algorithms and the common approach of solving problems through the use of *heuristics*. Heuristics are procedures that are applied when it is deemed impractical to use an algorithm that always finds the optimal solution. Heuristics typically run much faster than algorithms that find optimal solutions, and while they may often find solutions that are optimal or very close to optimal, there are no guarantees on the quality of the solution found. In contrast, the algorithms in this thesis all come with guarantees.

Our approach to solving the concurrent flow problem can be easily understood in pseudo-economic terms. The essential complexity of the problem arises because the different commodities are all competing for the same scarce resource, the capacities of the edges. In order to model this process, we introduce a *pricing scheme*. Consider a particular flow, say the one that appears in Figure 1.2. We introduce a price on each edge, to represent the *congestion*, or percentage utilization of that edge. If an edge is heavily congested, it has a high price, and if an edge is lightly congested it has a low price. For example, edge v_1v_3, which has congestion $2/2 = 1$, has a higher price than edge v_4v_5, which has congestion $1/4$. Once we have these prices, a particular routing for a commodity has a cost, which is based on the prices of the edges that the commodity is using. Consider commodity 3. It sends flow over the two edges with maximum congestion, i.e., the two highest priced edges. A cheaper way to send its flow might be over the bottom path $v_1v_2v_4v_5$. Our algorithm recognizes this situation and reroutes some of the flow of commodity 3 off its current path and onto the path $v_1v_2v_4v_5$. For example, if half the flow were rerouted, we would obtain the flow depicted in Figure 1.3. After rerouting, the congestion of edges change, and hence the prices change.

While this description is a highly simplified version of our algorithm, it does capture the essential ideas. The key to the efficiency of our algorithm is twofold. First, we can phrase this question of finding a cheap way to route flow as a minimum-cost flow problem, which is a "well-solved" problem. Second, we can show that, for the right choice of parameters, a rerouting procedure does not have to be executed too many times. This is the difficult part of the analysis. We need to show that every iteration of our algorithm makes progress, for some suitably defined notion of progress. We also need to be able to detect when our solution is ϵ-optimal. Note that we are requiring that we can detect when our solution is within a $(1+\epsilon)$ factor of optimal, *without knowing what the optimal value is.* To do this, we develop a notion of *relaxed optimality* and a detection scheme that uses suitably relaxed versions of the complementary slackness conditions of linear programming.

A detailed description of this algorithm, along with algorithms for an important special case, that in which all edges have capacity 1, appears in Chapter 2.

The multicommodity flow problem has several important applications. Many classical problems in Operations Research can be phrased as multicommodity flow problems, including:

- telecommunications problems,
- import-export problems,
- freight transport and scheduling,
- network design,
- freight assignment in the less-that-truckload trucking industry,
- traffic planning, and
- busing students to schools.

Multicommodity flow can also be used to find good separators for graphs, yielding divide-and-conquer algorithms for several NP-hard graph problems. In particular, the results in this thesis can be used to give the fastest polylogarithmic approximations to a number of problems including:

- VLSI channel routing,
- minimum cut linear arrangement,
- minimum area layout,
- $\sqrt{2}$-bifurcators of a graph,
- minimum feedback-arc set,
- graph embedding problems,
- chordalization of a graph,
- register sufficiency,
- minimum deletion of clauses in a $2CNF \equiv$ formula,
- via minimization, and
- the edge-deletion graph bipartization problems.

These problems will be discussed in Chapter 3. In addition we will show how, in some cases, our algorithms can be adapted to find *integral* solutions. The ability to do so is significant, since the integral multicommodity flow problem is likely to be more difficult than the problem of finding an optimal flow that is not necessarily integral: the former problem is NP-hard whereas the latter is solvable in polynomial time via linear programming.

Not only do our algorithms have provably efficient running times, but they perform well in practice. In Chapter 4 we discuss an implementation of one variant of the algorithm presented in Chapter 2. The results, while preliminary, are rather encouraging. For large problems our implementation significantly outperforms a good simplex-based linear programming code. In fact, we have been able to solve problems that are larger than those that can be solved by good simplex-based codes. In particular, we are able to solve problems in which there are a large number of commodities.

In Chapter 5, we turn to the problem of shop scheduling. We give the first polylogarithmic approximation algorithms for the job-shop problem, flow-shop problem, and several extensions. Our algorithms are randomized and combine techniques from two seemingly disparate fields of study: vector-sum theorems and packet routing algorithms.

In Chapter 6, we show how to make the shop scheduling algorithms deterministic. Our algorithm makes use of some recent extensions of our multicommodity flow techniques and unifies many of the ideas in this thesis, since it is necessary to find an approximately optimal integral solution to a generalized version of the multicommodity flow problem. The algorithms we use are closely related to those used for the multicommodity flow problem.

Throughout this thesis, we assume familiarity with the basic concepts of linear and integer programming. While none of our algorithms actually rely on a procedure for linear programming, some of the proofs rely on well-known results about linear programming. We refer the reader who is unfamiliar with linear programming to a basic textbook such as that of Chvátal [11] or Schrijver [55].

We include an glossary of notation.

Chapter 2

Multicommodity Flow Algorithms[1]

2.1 Introduction

The multicommodity flow problem involves simultaneously shipping several different commodities from their respective sources to their sinks in a single network so that the total amount of flow going through each edge is no more than the edge's capacity. Associated with each commodity is a *demand*, which is the amount of that commodity that we wish to ship. Given a multicommodity flow problem, one often wants to know if there is a *feasible* flow, i.e., if it is possible to find a flow that satisfies the demands and obeys the capacity constraints. More generally, we might wish to know the maximum percentage z such that at least z percent of each demand can be shipped without violating the capacity constraints. The latter problem is known as the *concurrent flow problem*, and is equivalent to the problem of determining the minimum ratio by which the capacities must be uniformly increased in order to ship 100% of each demand. For our algorithms, it is convenient to state the concurrent flow problem in a different, but equivalent form. We are given a network and a set of commodities. Let the congestion of an edge be the ratio of the flow on that edge to its capacity. We wish to find a way to route each commodity so that the maximum edge congestion is minimized. We denote the value of the maximum edge congestion by λ and the minimum possible maximum edge congestion by λ^*. Our algorithms are approximation algorithms that find ϵ-optimal solutions,

[1]This chapter contains joint work with Tom Leighton, Fillia Makedon, Serge Plotkin, Éva Tardos and Spyros Tragoudas [42] and joint work with Philip Klein, Serge Plotkin and Éva Tardos [35].

ones in which $\lambda \leq (1+\epsilon)\lambda^*$.

In this chapter, we describe the first combinatorial approximation algorithms for the concurrent flow problem. Given any positive ϵ, the algorithms find an ϵ-optimal solution. The running times of the algorithms depend polynomially on ϵ^{-1} and are significantly better than those of previous algorithms when ϵ is a constant. More specifically, we prove the following result. Throughout, we use n, m and k to denote the number of nodes, edges and commodities, we assume that the demands and the capacities are integral, and use D and U to denote the largest demands and capacities, respectively. We also assume, for now, that each commodity has one source and one sink. We refer to this problem as the simple multicommodity flow problem.

Theorem 2.1.1 For any $\epsilon > 0$, an ϵ-optimal solution for the simple concurrent flow problem can be found by a randomized algorithm in $O(\epsilon^{-3}kmn\log k\log^3 n)$ time and by a deterministic algorithm in $O(\epsilon^{-2}k^2mn\log k\log^3 n)$ time.

A complete table of results appear at the end of this introduction in Figure 2.1.

Our expected running time is the same (up to polylog factors) as the time needed to compute k maximum-flows, thus giving the surprising result that approximately computing a k-commodity concurrent flow is about as difficult as computing k single commodity maximum-flows. In fact, we formally prove that an instance of a k-commodity flow problem can be approximately solved by approximately solving $O(k\log k\log n)$ minimum-cost flow problems.

The running times in the above theorem can be improved when k is large. Let k^* denote the number of different sources. In both the randomized and the deterministic algorithm we can replace k in the running time by k^* at the expense of having to replace one of the $\log n$ terms by a $\log(nU)$. Notice that k^* is at most n for all simple multicommodity flow problems.

As a consequence of our approximation algorithm for the concurrent flow problem, we obtain a *relaxed decision procedure* for multicommodity flow feasibility; that is, given an instance of the multicommodity flow problem, we can either prove that it is infeasible, or give a feasible flow for the problem in which the capacity of each edge increased by a factor of $1+\epsilon$. Since in practice, the input to a multicommodity flow problem may have some measurement error, by making ϵ small enough, we can obtain a procedure for determining feasibility up to the precision of the input data.

An important special case of the concurrent flow problem occurs when all edge capacities are 1. For this special case, we can give even faster algorithms. The algorithms for the case when the edge capacities are 1 solve a series of shortest path problems instead of a series of minimum-cost flow problems. The shortest-path variant of the algorithm performs more iterations than minimum-cost flow version, but each iteration of the shortest path variant runs in less time than the minimum-cost flow based variant. In some cases, the shortest-path based algorithm yields faster algorithms. Historically, this shortest-path variant for the unit-capacity case preceded the general minimum-cost flow based variant. In fact, the original version of the algorithm for the general case used a series of shortest path computations, rather than minimum-cost flow computations. In spite of the fact that a minimum-cost flow can be found via a series of shortest path computations, by doing the minimum-cost flow computations directly, we are able to obtain faster algorithms.

The only previously known algorithms for solving (or approximately solving) the general concurrent flow problem use linear programming. The concurrent flow problem can be formulated as a linear program in $O(mk)$ variables and $O(nk + m)$ constraints. Any polynomial time linear programming algorithm can be used to solve the problem optimally. Kapoor and Vaidya [30] gave a method to speed up the matrix inversions involved in Karmarkar-type algorithms for multicommodity flow problems. Combining their technique with Vaidya's linear programming algorithm that uses fast matrix multiplication [67] yields a time bound of $O(k^{3.5}n^3m^{.5}\log(nDU))$ to obtain an optimal solution to the concurrent flow problem with integer demands and an $O(k^{2.5}n^2m^{.5}\log(n\epsilon^{-1}DU))$ time bound to find an approximate solution.

The only previous combinatorial polynomial approximation algorithms for concurrent flow problems only handle the special case when all the capacities are 1. For this special case, Shahrokhi and Matula [59] gave an algorithm that ran in $O(\epsilon^{-5}nm^7)$ time. Our algorithm is based on this work, so we describe the basic ideas here.

The algorithm starts by finding a flow that satisfies the demands but not the capacity constraints. The algorithm then repeatedly reroutes flow so as to decrease the maximum flow on any edge. To guide the rerouting, they assign *lengths* to each edge and then reroute flow off a path that is long with respect to those lengths onto one that is short with respect to these lengths.

In the unit capacity case, our approach differs from that of Shahrokhi and Matula in several ways. We develop a framework of *relaxed optimality conditions* that allows us to measure the congestion on both a local and a global level, thereby giving us more freedom in choosing which flow paths to reroute at each iteration. We exploit this freedom by using a faster *randomized* method for choosing flow paths. In addition, this framework also allows us to achieve greater improvement as a result of each rerouting.

In the general case, we must first develop the appropriate framework to handle general capacities. We will develop more general relaxed optimality conditions. Also, we are able to reroute an entire commodity during each iteration instead of only a single path of flow. To do this rerouting , we compute a minimum-cost flow in an auxiliary graph and reroute a portion of the flow accordingly. As a consequence, we are able to make much greater progress during each iteration. Of course, the time to run each iteration goes up, but the tradeoff proves to be worthwhile since the improvement obtained in each iteration is large enough so that we need to solve only $O(k \log k \log n)$ minimum-cost flow problems in order to get an approximately optimal solution.

The running times of the presented algorithms depend polynomially on ϵ^{-1}. The deterministic algorithm runs in time proportional to ϵ^{-2} and the randomized one runs in time proportional to ϵ^{-3}. For the randomized algorithm, Goldberg [21] and Grigoriadis and Khachiyan [27] have shown how to improve the dependence on ϵ of the randomized algorithm to ϵ^{-2}.

Our model of computation is the RAM. We shall use the elementary arithmetic operations (addition, subtraction, comparison, multiplication, and integer division), and count each of these as a single step. All numbers occurring throughout the computation will have $O(\log(nU))$ bits. For ease of exposition we shall first use a model of computation that allows exact arithmetic on real numbers and assumes that exponentiation is a single step. We then show how to convert the results to the usual RAM model.

Scenario	Running Time
General, randomized	$O\left(mnk(\epsilon^{-3}+\log k)\min\left\{\log\left(\frac{n^2}{m}\right),\log\log n\right\}\log n\right)$ $O\left(mnk^*(\epsilon^{-3}+\log k^*)\min\left\{\log\left(\frac{n^2}{m}\right),\log\log(nU)\right\}\log(nU)\right)$
General, deterministic	$O\left(mnk^2(\epsilon^{-2}+\log k)\min\left\{\log\left(\frac{n^2}{m}\right),\log\log n\right\}\log n\right)$ $O\left(mnk^{*2}(\epsilon^{-2}+\log k^*)\min\left\{\log\left(\frac{n^2}{m}\right),\log\log(nU)\right\}\log(nU)\right)$
Unit capacity, randomized	$O((k\epsilon^{-1}+m\epsilon^{-3}\log n)(m+n\log n))$ $O(km^{3/2}\log^2 n(\epsilon^{-3}+\log k))$
Unit capacity, deterministic	$O(((k+\epsilon^{-2}m)\log n)(k^*n\log n+m(\log n+\min\{k,k^*(\log d_{\max}+1)\})))$ $O(k^2m^{3/2}\log^2 n(\epsilon^{-2}+\log k))$
Unit capacity, $\epsilon=O(1)$ randomized	$O(m(k+m)\log n)$
Unit capacity, $\epsilon=O(1)$ deterministic	$O(m(k+m)(\log n+\min\{k,k^*(\log d_{\max}+1)\}\log n))$
Unit capacity, unit demand, $\epsilon=O(1)$, randomized	$O(m\log n\log k(m+n\log n+k\log k))$

Figure 2.1: Some of our running times for the multicommodity flow problem. The bounds are for an n-node, m-edge graph with maximum edge capacity U and maximum demand $d_{\max}$. There are k commodities and k^* distinct sources.

2.2 Preliminaries

Throughout this chapter we use the notation $\mathbf{A}^S$, where $\mathbf{A}\in\{\mathbf{R},\mathbf{Z},\mathbf{R}_+,\mathbf{Z}_+\}$ and $S\in\{V,E\}$ to denote an $|S|$-dimensional vector in which each element is a member of the set $\mathbf{A}$. The component of $\beta\in\mathbf{A}^S$ corresponding to, say node v, is denoted by $\beta(v)$.

An instance $\mathcal{I}=(G,u,\mathcal{K})$ of the *simple multicommodity flow problem* consists of an undirected graph $G=(V,E)$ with vertex set V and edge set E, a capacity vector $u\in\mathbf{R}_+^E$, and a specification $\mathcal{K}$ of k commodities, numbered 1 through k, where the specification for commodity i consists of a source-sink pair $s_i,t_i\in V$ and a non-negative demand d_i. We denote the number of distinct sources by k^*, the number of nodes by n, and the number of edges by m. For notational convenience we assume that $m\geq n$, and that the graph G is connected and has no parallel edges. Also, for notational convenience, we arbitrarily direct each edge. If there is an edge directed from v to w, this edge is unique by assumption, and we denote it by vw. We assume that the capacities and the demands are integral, and denote the largest capacity by U and the sum of the demands by D.

A multicommodity flow f consists of k vectors $f_i,\ i = 1, \ldots, k$, where $f_i \in \mathbf{R}^E$. The quantity $f_i(vw)$ represents the *flow* of commodity i on edge vw. If the flow of commodity i on edge vw is oriented in the same direction as edge vw, then $f_i(vw)$ is positive, otherwise it is negative. The signs only serve to indicate the direction of the flows. For each commodity i we require the *conservation constraints*:

$$\begin{aligned}
\sum_{wv\in E} f_i(wv) - \sum_{vw\in E} f_i(vw) &= 0 && \text{for each node } v \notin \{s_i, t_i\}, && (2.1)\\
\sum_{vw\in E} f_i(vw) &= d_i && \text{for } v = s_i, && (2.2)\\
\sum_{wv\in E} f_i(wv) &= d_i && \text{for } v = t_i. && (2.3)
\end{aligned}$$

Note that (2.1) and (2.2) imply (2.3). Alternatively, we can define the *flow* of a commodity in the following way. Let $\mathcal{P}_i$ denote a collection of paths from s_i to t_i in G, and let $f_i(P)$ be a nonnegative value for every path P in $\mathcal{P}_i$ that represents the amount of flow carried by path P. The *value* of the flow thus defined is $\sum_{P\in\mathcal{P}_i} f_i(P)$, which is the total flow delivered from s_i to t_i. The amount of flow through an edge vw is

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

We will use both formulations as convenient.

We define the value of the total flow on edge vw to be $f(vw) = \sum_i |f_i(vw)|$, and say that a multicommodity flow f in G is *feasible* if $f(vw) \le u(vw)$ for all edges vw. (Note that $f(vw)$ is always non-negative.)

We consider the optimization version of this problem, called the *simple concurrent flow problem*, first defined by Shahrokhi and Matula [59]. In this problem the objective is to compute the maximum possible value z such that there is a feasible multicommodity flow with demands $z \cdot d_i$ for $1 \le i \le k$. We call z the *throughput* of the multicommodity flow. An equivalent formulation of the concurrent flow problem is to compute the minimum $\lambda = 1/z$ such that there is a feasible flow with demands d_i and capacities $\lambda \cdot u(vw)$. We shall use the notation $\lambda(vw)$ to denote the *congestion* $f(vw)/u(vw)$ of an edge $vw \in E$, $\lambda = \max_{vw\in E} \lambda(vw)$, and λ^*

to denote the optimal (minimum) value of λ. We can now restate the concurrent flow problem as follows:

Simple Concurrent Flow Problem (restatement) Given an instance $\mathcal{I}$ of the multicommodity flow problem, find a flow that satisfies the conservation constraints (2.1)–(2.3) and minimizes the congestion λ.

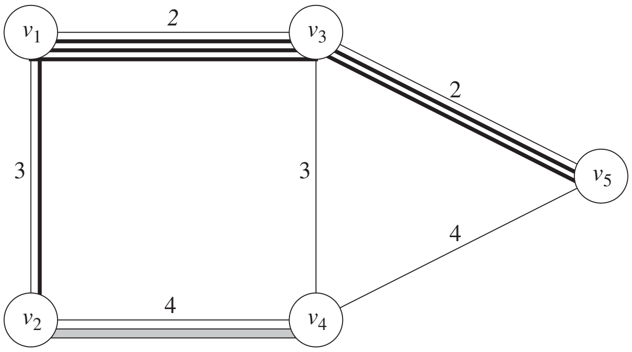
A multicommodity flow f satisfying the demands d_i, $i = 1, \ldots, k$ is *ϵ-optimal* if its congestion λ is at most a factor $(1+\epsilon)$ more than the minimum possible value; that is $\lambda \leq (1+\epsilon)\lambda^*$. The *approximation problem* associated with the concurrent flow problem is to find an ϵ-optimal multicommodity flow f. We shall assume implicitly throughout that ϵ is at least inverse polynomial in n and is at most 1. This assumption is without loss of generality. If $\epsilon > 1$, we can run the algorithm for $\epsilon = 1$. If ϵ^{-1} is greater than any polynomial in n, our algorithms still yield a correct solution. In this case, however, the running times of our algorithms are somewhat greater and will be dominated by the time to solve the problem exactly.

We can extend the results in this chapter to the case where the input graph is directed. In this case we require all edge flows to be non-negative and oriented in the same direction as the corresponding edges in the input graph. The results in this chapter carry through to this case with slight notational changes. Henceforth, we focus only on the undirected case.

The *general multicommodity flow problem* is a natural extension of the simple problem when each commodity may have more than one source and sink. For each commodity i we are given an n-dimensional demand vector $\hat{d}_i \in \mathbf{Z}^V$, where the j^{th} component $\hat{d}_i(v_j)$ denotes the demand for commodity i at node v_j. A negative demand denotes a supply. We require that the total demand equal the total supply, i.e., $\sum_v \hat{d}_i(v) = 0$ and we shall use D_i to denote $\max_v \left\{ |\hat{d}_i(v)| \right\}$. The conservation constraints of equations (2.1)–(2.3) are replaced by the more general conservation constraints:

$$\sum_{wv \in E} f_i(wv) - \sum_{vw \in E} f_i(vw) = \hat{d}_i(v) \quad i = 1, \ldots, k; \quad v \in V. \tag{2.4}$$

Many of our results can be extended to this slightly more general model, although we shall not address this issue in this thesis. The main point in introducing this model is to reduce the number of commodities. We will show that every simple concurrent flow problem is equivalent



commodity	s	t	demand
1	v_1	v_3	1
2	v_2	v_4	1
3	v_1	v_5	2
4	v_1	v_2	1

	$\hat{d}(v_1)$	$\hat{d}(v_2)$	$\hat{d}(v_3)$	$\hat{d}(v_4)$	$\hat{d}(v_5)$	symbol
1	-4	1	1	0	2	
2	0	-1	0	1	0	

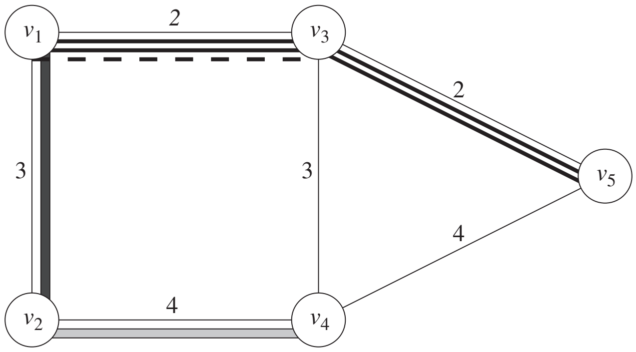
Figure 2.2: The original input, the grouped input and a solution to the grouped problem.

to a general concurrent flow problem with at most n commodities.

For a general concurrent flow problem, it may not be possible to reduce the number of commodities. To simplify the running time bounds, we will assume that the number of commodities is polynomial in n. In particular, we will use that $\log k = O(\log n)$.

We now explain how to convert a simple concurrent flow problem to a general concurrent flow problem with k^* commodities, where k^* is the number of distinct sources: we combine those commodities that share a source. In other words, for each source s we define a demand vector $\hat{d}_s \in \mathbf{Z}^V$ as follows: for each commodity i with $s_i = s$, we set $\hat{d}_s(t_i) = d_i$; we set $\hat{d}_s(s) = -\sum\{d_i : s_i = s\}$; and we set all other demands to zero.

We give an example of combining and uncombining flows in Figures 2.2 and 2.3. In Figure 2.2, the input is a 4 commodity simple concurrent flow problem. Commodities 1,2 and 4 all have v_1 as a source. Hence we can combine these 3 commodities into 1 commodity group, group 1. The demand vector for this group appears in the second table. Node v_1 is a supply node with 4 units of supply, and hence $\hat{d}(v) = -4$. Nodes v_2 and v_3 have demand 1 and node v_5 has



commodity	s	t	demand	symbol
1	v_1	v_3	1	
2	v_2	v_4	1	
3	v_1	v_5	2	
4	v_1	v_2	1	

Figure 2.3: The result of ungrouping the solution in Figure 2.2

demand 2. Node v_4, which had no demand in any of the original commodities, has no demand in the grouped commodity. The second commodity group consists of the original commodity 2. A solution for this grouped commodity is given in the graph. It is easy to check the demands for the two commodity groups are satisfied. Figure 2.3 shows how to convert the solution for the grouped instance in Figure 2.2 into one for the original instance. Commodity group 1 has been split into three commodities. The total amount of flow on each edge is still the same and all the original demands are still satisfied.

Lemma 2.2.1 Consider a simple k-commodity concurrent flow problem and the corresponding k^*-commodity problem defined by combining commodities that share a source.

1. Given the ungrouped problem, the grouped problem can be created in $O(kn)$ time.
2. Any feasible solution to one can be converted to a solution to the other with the same congestion.
3. The conversion of a solution for the k^*-commodity grouped problem to one for the k-commodity ungrouped problem can be done in $O(k^*nm)$ time, or in $O(k^*m \log n)$ time using the dynamic tree data structure.

Proof: The conversion of an instance of the grouped problem from an ungrouped one can be performed, in $O(kn)$ time, by the procedure described above for combining commodities that share a source. The conversion of a solution of the simple concurrent flow with k commodities into a solution of the k^*-commodity problem is straightforward, we simply add the flows that share a common source. Assume that we are given a solution to the general concurrent flow problem with k^* commodities. Decompose the flow of each commodity into paths and cycles and combine the flows on paths that have the same source and sink nodes, disregarding the cycles. This procedure is known as flow decomposition and it is well known how to compute a decomposition in $O(nm)$ time (see, for example [3]) and in $O(m \log n)$ time using the dynamic tree data structure. [64] ■

The sources and sinks play a symmetric role in the (undirected) problem, and hence k^* in the lemma could have been defined as the number of nodes in any subset that contains an endpoint of each commodity. While finding a minimum such node set is NP-complete, we mention this formulation because in some cases it leads to an efficiently computable k^* that is smaller than the one defined above.

Except for the few places in this chapter where we explicitly distinguish between simple and non-simple concurrent flow problems, all our bounds are for a k-commodity non-simple concurrent flow problem, and hence they also apply to a k-commodity simple concurrent flow problem. The only distinction between the two variants will be made in the routine INITIALIZE and its analysis in Lemma 2.4.2, and in the final analysis of our algorithms in Theorems 2.4.21, 2.4.23 and 2.4.32.

The main subroutine of our algorithm is a minimum-cost flow computation (of a single commodity). We use the following, slightly unconventional definition. An instance of a minimum-cost flow problem $\mathcal{M} = (G, u, c, \hat{d}_i)$ consists of a graph $G = (V, E)$ with edge capacities $u \in \mathbf{R}_+^E$, edge costs $c \in \mathbf{R}^E$ and a demand vector $\hat{d}_i$. The cost C_i of a flow f_i is $\sum_{vw \in E} c(vw)\,|f_i(vw)|$. Given a demand vector $\hat{d}_i(v)$, and capacities u, the *minimum-cost flow problem* is the problem of finding a flow of minimum cost that satisfies the conservation constraints (2.4) and has $|f_i(vw)| \leq u(vw)$ for each edge $vw \in E$. We denote the value of the minimum-cost flow by C_i^*. The *residual graph* of a flow f_i, denoted $G_{f_i} = (V, E_{f_i})$ is the directed graph consisting of the set of edges for which $f_i(vw) < u(vw)$ and the reversal of the set of edges for which $f_i(vw) > -u(vw)$. In Section 3.3.3, we will need to work with the linear-programming dual of a minimum-cost flow. The dual variables on the nodes are commonly referred to as *prices*, and are denoted by p. A *price function* is a vector $p \in \mathbf{R}^V$. The *reduced cost* of an edge $vw \in E$ is $c(vw) + p(v) - p(w)$, and $-c(vw) - p(v) + p(w)$ on reverse edges. Linear programming duality implies that a flow f_i is of minimum cost if and only if there exists a price function p, such that the reduced cost of the edges in the residual graph of f_i are nonnegative (complementary slackness conditions).

For initialization, we will need to solve maximum flow problems. We use the following, slightly unconventional definition. An instance of a maximum flow problem $\mathcal{N} = (G, u, s_i, t_i)$ is a graph $G = (V, E)$ with edge capacities $u \in \mathbf{R}_+^E$, and two distinguished nodes, the source s_i and the sink t_i. The maximum flow problem is the problem of finding the maximum value d_i such that there exists a flow $f_i \in \mathbf{R}^E$ that satisfies the conservation constraints (2.1)–(2.3) and has $|f_i(vw)| \leq u(vw)$ for each edge $vw \in E$.

We will also need to solve a variant of the maximum flow problem that we call the *feasible flow* problem. The input to a feasible flow problem $\mathcal{F} = (G, u, \hat{d}_i)$ consists of a graph $G = (V, E)$ with edge capacities u and a demand vector $\hat{d}_i$. The object is to find a flow f_i satisfying the conservation constraints (2.4) and that has $|f_i(vw)| \leq u(vw)$ for each edge $vw \in E$. It is well-known how to convert an instance $\mathcal{F}$ of the feasible flow problem with n nodes and m edges into an instance $\mathcal{M}$ of the maximum flow problem with at most $n+2$ nodes and $m+2n$ edges. Thus both the maximum flow problem and the feasible flow problem can be solved by a maximum flow computation on a graph with $O(n)$ nodes and $O(m)$ edges.

2.2.1 Optimality Conditions

Linear programming duality can also be used to give a characterization of the optimum solution for the concurrent flow problem. Let $\ell \in \mathbf{R}_+^E$ be a nonnegative *length* function. For nodes $v, w \in V$, let $dist_\ell(v, w)$ denote the length of the shortest path from v to w in G with respect to the length function ℓ. The following theorem is a special case of the linear programming duality theorem.

Theorem 2.2.2 For a simple multicommodity flow f satisfying the demands d_i, $i = 1, \ldots, k$ and capacities $\lambda \cdot u(vw)$, $\forall vw \in E$, and any length function ℓ,

$$\begin{aligned}
\lambda \sum_{vw\in E} \ell(vw)u(vw) &\geq \sum_{vw\in E} \ell(vw)f(vw) \\
&= \sum_{i=1}^{k} \sum_{vw\in E} \ell(vw)\,|f_i(vw)| \\
&= \sum_{i=1}^{k} \sum_{P\in\mathcal{P}_i} \ell(P)f_i(P) \\
&\geq \sum_{i=1}^{k} dist_\ell(s_i,t_i)d_i. \qquad (2.5)
\end{aligned}$$

Furthermore, a multicommodity flow f minimizes λ if and only if there exists a nonzero length function ℓ for which the inequalities above all hold with equality.

Theorem 2.2.2 is a characterization of optimality that relates the value of λ to the lengths of the shortest path for each commodity. We shall also use a slightly different characterization, one that relates the value of λ to the costs of minimum-cost flows in appropriately derived graphs. While these characterizations can be proven to be equivalent, by measuring optimality in terms of minimum-cost flows, we are able to develop faster algorithms for the general case.

Let ℓ be a nonnegative length function on the edges, f a multicommodity flow, and λ its congestion. Let C_i be the cost of the current flow for commodity i, using ℓ as the cost function, i.e., $C_i = \sum_{vw\in E} \ell(vw)\,|f_i(vw)|$. For a commodity i, let $C_i^*(\lambda)$ be the value of a minimum-cost flow f_i^* satisfying the demands of commodity i, subject to costs ℓ and capacities $\lambda \cdot u(vw)$, i.e., let f_i^* be a flow that satisfies $|f_i^*(vw)| \leq \lambda \cdot u(vw)$ and minimizes the cost $C_i^*(\lambda) = \sum_{vw\in E} \ell(vw)\,|f_i(vw)|$. For brevity we shall sometimes use C_i^* to abbreviate $C_i^*(\lambda)$.

The following theorem is a restatement of Theorem 2.2.2.

Theorem 2.2.3 For a (general) multicommodity flow f satisfying capacities $\lambda \cdot u(vw)$, and a length function ℓ,

$$\begin{aligned}
\lambda \sum_{vw \in E} \ell(vw) u(vw) &\geq \sum_{vw \in E} \ell(vw) f(vw) \\
&= \sum_{i=1}^{k} \sum_{vw \in E} \ell(vw) \left| f_i(vw) \right| \\
&= \sum_{i=1}^{k} C_i \\
&\geq \sum_{i=1}^{k} C_i^*(\lambda). \qquad (2.6)
\end{aligned}$$

Furthermore, a multicommodity flow f minimizes λ if and only if there exists a nonzero length function ℓ for which the inequalities above all hold with equality.

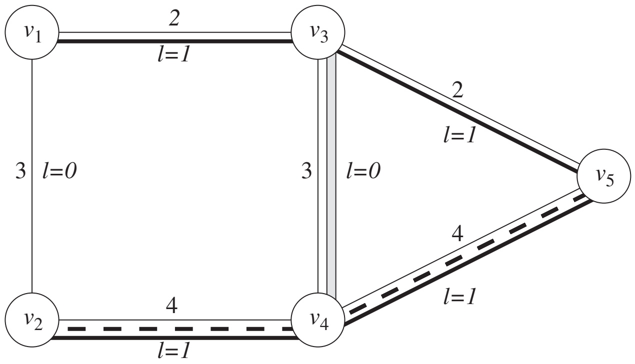
We would like to be able to say that the ratio of the last term and the multiplier of λ in the first term gives a lower bound on the optimal value λ^*. The analogous statement for the inequality (2.5) is obvious, because neither of the two terms depend on λ. In Theorem 2.2.3 the last term, $\sum_{i=1}^{k} C_i^*(\lambda)$, depends on λ. Observe, however, that the minimum cost of a flow subject to capacity constraints $\lambda \cdot u(vw)$ cannot increase if λ increases.

Lemma 2.2.4 Suppose that we have a multicommodity flow satisfying capacities $\lambda \cdot u(vw)$ and ℓ is a length function. Then $\lambda^* \geq \sum_{i=1}^{k} C_i^*(\lambda)/(\sum_{vw \in E} \ell(vw)u(vw))$.

Another well-known characterization of optimality for a linear program is known as the complementary slackness conditions. One way to formulate these conditions for multicommodity flow is to formulate them as conditions on edges and individual commodities.

Theorem 2.2.5 A multicommodity flow f has minimum λ if and only if there exists a nonzero length function ℓ such that

1. for each edge $vw \in E$, either $\ell(vw) = 0$ or $f(vw) = \lambda \cdot u(vw)$, and
2. for each commodity i, $C_i = C_i^*(\lambda)$.



commodity	s	t	demand	symbol
1	v_2	v_5	1	dashed
2	v_3	v_4	1	hollow
3	v_1	v_5	2	solid

Figure 2.4: An optimal solution

The complementary slackness conditions can also be formulated in terms of conditions on edges and paths. The following theorem is equivalent to the definition above, but will turn out to be more useful in the unit capacity case.

Theorem 2.2.6 A multicommodity flow f has minimum λ if and only if there exists a nonzero length function ℓ such that

1. for each edge $vw \in E$ either $\ell(vw) = 0$ or $f(vw) = \lambda$, and
2. for each commodity i and every path $P \in \mathcal{P}_i$ with $f_i(P) > 0$ we have $\ell(P) = dist_\ell(s_i, t_i)$.

We illustrate the concepts of this section by giving a length function that demonstrates the optimality of the flow given in Figure 1.3. In Figure 2.4, we give the flows and length functions. We first check Theorem 2.2.3. The leftmost term,

$$\lambda \sum_{vw \in E} \ell(vw) u(vw) = \frac{1}{2}(1 \cdot 2 + 1 \cdot 2 + 1 \cdot 4 + 1 \cdot 4 + 0 \cdot 3 + 0 \cdot 3) = 6.$$

The middle terms,

$$\sum_{vw \in E} \ell(vw) f(vw) = \sum_{i=1}^{k} \sum_{vw \in E} \ell(vw) \left| f_i(vw) \right| = \sum_{i=1}^{k} C_i = 1 + 1 + 2 + 2 = 6.$$

Finally, in order to compute the last term, we need to compute a minimum-cost flow for each commodity. The value of the minimum-cost flow is equal to the shortest $s_i - t_i$ path for commodity i multiplied by the demand of i. So $C_1^* = 2 \cdot 1$, $C_2^* = 0 \cdot 1$, and $C_3^* = 2 \cdot 2$. Summing, we get 6 and we have shown that at optimality the terms are all equal.

We can also check for optimality using the complementary slackness conditions of Theorem 2.2.5. The first condition is easily verified and the second was verified in the previous paragraph. Hence we have another "proof" that the flow given in Figure 1.3 is optimal.

2.3 Relaxed Optimality Conditions

The conditions in Theorems 2.2.5 and 2.2.6 describe when a solution is *optimal.* The goal of our algorithms, however, is to find a multicommodity flow f and a length function ℓ such that this lower bound is within a $(1+\epsilon)$ factor of optimal, i.e.,

$$\lambda \leq (1+\epsilon)\lambda^* \leq (1+\epsilon)\sum_{i=1}^{k} C_i^*(\lambda)/(\sum_{vw \in E} \ell(vw)u(vw)).$$

In this case, we have proved that f is ϵ-optimal, and ℓ is a particular length function that allows us to verify ϵ-optimality.

Let $\epsilon > 0$ be an error parameter, f a multicommodity flow satisfying capacities $\lambda \cdot u(vw)$, and ℓ a length function. We say that a commodity i is ϵ*-good* if

$$C_i - C_i^*(\lambda) \quad \leq \quad \epsilon C_i + \epsilon \frac{\lambda}{k} \sum_{vw \in E} \ell(vw)u(vw).$$

Otherwise, we say that the commodity is ϵ*-bad.* Intuitively, a commodity is ϵ-good if it is almost as cheap as the minimum cost possible for that commodity or it is at most a small fraction of $\lambda \sum_{vw \in E} \ell(vw)u(vw)$, the total cost of the network. We use this notion in defining a relaxed version of the complementary slackness conditions. We define the following *relaxed optimality conditions* (with respect to a multicommodity flow f that satisfies capacity constraints $\lambda \cdot u(vw)$, a length function ℓ and an error parameter ϵ):

$(R1)$ For each edge $vw \in E$,

either

$$(1+\epsilon) f(vw) \geq \lambda \cdot u(vw)$$

or

$$u(vw)\ell(vw) \leq \frac{\epsilon}{m} \sum_{xy \in E} \ell(xy) u(xy).$$

$(R2)$ $\displaystyle\sum_{i\ \epsilon\text{-bad}} C_i \leq \epsilon \sum_{i=1}^{k} C_i.$

Typically, complementary slackness conditions are used as a way to check whether a solution is optimal. We will use the relaxed optimality conditions to check when a solution is ϵ-optimal. We now show that if these two conditions are satisfied then the gap between the first and last terms in (2.6) is small. We begin by showing that the gap between the first and second terms is small.

Lemma 2.3.1 Suppose that flow f and length function ℓ satisfy Relaxed Optimality Condition R1. Then

$$\lambda \sum_{vw \in E} \ell(vw) u(vw) \leq \left(\frac{1+\epsilon}{1-\epsilon}\right) \sum_{vw \in E} \ell(vw) f(vw).$$

Proof: Let A denote the set of edges for which $(1+\epsilon) f(vw) \geq \lambda \cdot u(vw)$. We can estimate the sum $\lambda \sum_{vw \in E} \ell(vw) u(vw)$, by summing separately over the sets A and E/A, i.e.,

$$\lambda \sum_{vw \in E} \ell(vw) u(vw) = \lambda \sum_{vw \in A} \ell(vw) u(vw) + \lambda \sum_{vw \in E/A} \ell(vw) u(vw).$$

Now we bound the first sum using the first part of Relaxed Optimality Condition R1 and the second sum using the second part of Relaxed Optimality Condition R1. Thus

$$\begin{aligned}
\lambda \sum_{vw \in E} \ell(vw) u(vw) &\leq (1+\epsilon) \sum_{vw \in A} \ell(vw) f(vw) + \lambda \sum_{vw \in E/A} \left(\frac{\epsilon}{m} \sum_{vw \in E} \ell(vw) u(vw)\right) \\
&\leq (1+\epsilon) \sum_{vw \in A} \ell(vw) f(vw) + \lambda m \left(\frac{\epsilon}{m} \sum_{vw \in E} \ell(vw) u(vw)\right) \\
&\leq (1+\epsilon) \sum_{vw \in A} \ell(vw) f(vw) + \lambda \epsilon \sum_{vw \in E} \ell(vw) u(vw).
\end{aligned}$$

This chain of inequalities implies that

$$\begin{aligned}(1-\epsilon)\lambda \sum_{vw\in E} \ell(vw)u(vw) &\leq (1+\epsilon)\sum_{vw\in A} \ell(vw)f(vw)\\ &\leq (1+\epsilon)\sum_{vw\in E} \ell(vw)f(vw).\end{aligned}$$

■

We can now bound the gap between the last two terms in (2.6).

Lemma 2.3.2 Suppose that flow f and length function ℓ satisfy Relaxed Optimality Conditions R1 and R2. Then

$$\sum_{i=1}^{k} C_i \leq (1+5\epsilon)\sum_{i=1}^{k} C_i^*(\lambda).$$

Proof: We can bound the sum $\sum_{i=1}^{k} C_i$ by considering the contribution from ϵ-good and ϵ-bad commodities separately. The total contribution from all ϵ-bad commodities is bounded in Relaxed Optimality Condition 2 by $\epsilon\sum_{i=1}^{k} C_i$. The contribution from each ϵ-good commodity can be bounded using the definition of an ϵ-good commodity, i.e.,

$$C_i \leq \left(\frac{1}{1-\epsilon}\right)\left(C_i^*(\lambda) + \frac{\epsilon\lambda}{k}\sum_{vw\in E} \ell(vw)u(vw)\right).$$

Letting B represent the set of all ϵ-good commodities and summing over both ϵ-good and ϵ-bad commodities, we get that:

$$\begin{aligned}\sum_{i=1}^{k} C_i &\leq \sum_{i\in B} C_i + \sum_{i\notin B} C_i\\ &\leq \left(\frac{1}{1-\epsilon}\right)\sum_{i\in B}\left(C_i^*(\lambda) + \frac{\epsilon\lambda}{k}\sum_{vw\in E} \ell(vw)u(vw)\right) + \epsilon\sum_{i=1}^{k} C_i.\end{aligned}$$

Combining like terms, we obtain

$$(1-\epsilon)\sum_{i=1}^{k} C_i \leq \left(\frac{1}{1-\epsilon}\right)\sum_{i=1}^{k} C_i^*(\lambda) + \left(\frac{1}{1-\epsilon}\right)\sum_{i=1}^{k}\frac{\epsilon\lambda}{k}\sum_{vw\in E} \ell(vw)u(vw) \qquad (2.7)$$

$$\leq \left(\frac{1}{1-\epsilon}\right)\sum_{i=1}^{k} C_i^*(\lambda) + \left(\frac{\epsilon}{1-\epsilon}\right)\lambda \sum_{vw\in E} \ell(vw)u(vw). \tag{2.8}$$

Now we can use Lemma 2.3.1 to bound the second term on the righthand side. This yields

$$(1-\epsilon)\sum_{i=1}^{k} C_i \leq \left(\frac{1}{1-\epsilon}\right)\sum_{i=1}^{k} C_i^*(\lambda) + \left(\frac{\epsilon}{1-\epsilon}\right)\left(\frac{1+\epsilon}{1-\epsilon}\right)\sum_{vw\in E} \ell(vw)f(vw) \tag{2.9}$$

$$= \left(\frac{1}{1-\epsilon}\right)\sum_{i=1}^{k} C_i^*(\lambda) + \left(\frac{\epsilon(1+\epsilon)}{(1-\epsilon)^2}\right)\sum_{i=1}^{k} C_i. \tag{2.10}$$

Combining like terms,

$$\sum_{i=1}^{k} C_i \leq \frac{\frac{1}{1-\epsilon}}{1-\epsilon-\frac{\epsilon(1+\epsilon)}{(1-\epsilon)^2}}\sum_{i=1}^{k} C_i^*(\lambda).$$

A simple algebraic calculation shows that for $\epsilon \leq \frac{1}{9}$, the right side is at most $(1+5\epsilon)\sum_{i=1}^{k} C_i^*(\lambda)$.
■

Theorem 2.3.3 Suppose f, ℓ, and ϵ satisfy the relaxed optimality conditions and $\epsilon < 1/9$. Then f is $O(\epsilon)$-optimal, and in particular, $\lambda \leq (1+9\epsilon)\lambda^*$.

Proof: Combining Lemma 2.3.1 and 2.3.2,

$$\lambda \sum_{vw\in E} \ell(vw)u(vw) \leq \frac{(1+\epsilon)(1+5\epsilon)}{1-\epsilon}\sum_{i=1}^{k} C_i^*(\lambda).$$

Simple algebra and Lemma 2.2.4 prove the theorem. ■

The Unit Capacity Case

For the unit capacity case, we sometimes benefit from using the optimality conditions of Theorem 2.2.2. We can develop similar conditions for relaxed optimality where the optimality is in terms of paths rather than commodities. The development is similar to that for commodities, so our presentation shall be more concise.

Let $0 < \epsilon < 1/12$ be an error parameter, f a multicommodity flow and ℓ a length function.

We say that a path $P \in \mathcal{P}_i$ for a commodity i is *ϵ-short* if

$$\ell(P) - dist_\ell(s_i, t_i) \leq \epsilon \ell(P) + \epsilon \frac{\lambda}{\min\{D, kd_i\}} \sum_{vw \in E} \ell(vw) u(vw).$$

and *ϵ-long* otherwise. The intuition is that a flow path is ϵ-short if it is short in either a relative or an absolute sense, i.e., it is either almost as short as the shortest possible (s_i, t_i)-path or it is at most a small fraction of $\sum_{vw \in E} \ell(vw)u(vw)$. We use this notion in defining *relaxed optimality conditions* for the unit-capacity case (with respect to a flow f, a length function ℓ and an error parameter ϵ). The new relaxed optimality conditions are condition $R1$ defined above and the following variant of condition $R2$,

$$(R2') \sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_i \\ P\ \epsilon\text{-bad}}} f_i(P)\ell(P) \leq \epsilon \sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P).$$

Relaxed Optimality Condition $R2'$ says that the amount of flow that is on ϵ-long paths contributes a small fraction of the sum $f \cdot \ell$.

Lemma 2.3.2 bounds the gap between the first and second terms in (2.5). We now proceed to bound the gap between the last two terms.

Lemma 2.3.4 Suppose f and ℓ and ϵ satisfy the Relaxed Optimality Conditions $R1$ and $R2'$. Then

$$\sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P) \leq (1 + 8\epsilon) \sum_{i=1}^{k} dist_\ell(s_i, t_i) d_i.$$

Proof: We break the sum, $\sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$, into two parts; the sum over ϵ-short paths and the sum over ϵ-long paths. Relaxed optimality condition $R2'$ gives us an upper bound of $\epsilon \sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$ on the sum over the ϵ-long paths. Taking the definition of an ϵ-short path and multiplying both sides by $f_i(P)$ gives us the following bound that applies for ϵ-short paths:

$$\ell(P) f_i(P) \leq \frac{1}{1 - \epsilon} \left(dist_\ell(s_i, t_i) f_i(P) + \frac{\epsilon \lambda f_i(P)}{\min\{D, kd_i\}} \sum_{vw \in E} \ell(vw) u(vw) \right).$$

Let S denote the set of ϵ-short paths. Summing over all ϵ-short paths and using the facts that $\sum_{P\in\mathcal{P}_i\cap S} f_i(P) \le \sum_{P\in\mathcal{P}_i} f_i(P) = d_i$, and $(\min\{D, kd_i\})^{-1} \le D^{-1} + (kd_i)^{-1}$ we get

$$\begin{aligned}
\sum_{P\in S} f_i(P)\ell(P) &\le \frac{1}{1-\epsilon}\sum_{i=1}^{k}\sum_{P\in\mathcal{P}_i\cap S}\left(dist_\ell(s_i,t_i)f_i(P) + \epsilon\frac{\lambda}{\min\{D,kd_i\}}f_i(P)\sum_{vw\in E}\ell(vw)u(vw)\right)\\
&\le \frac{1}{1-\epsilon}\sum_{i=1}^{k} dist_\ell(s_i,t_i)d_i + \frac{\epsilon}{1-\epsilon}\sum_{i=1}^{k}\left(\frac{\lambda d_i}{\min\{D,kd_i\}}\sum_{vw\in E}\ell(vw)u(vw)\right)\\
&\le \frac{1}{1-\epsilon}\sum_{i=1}^{k} dist_\ell(s_i,t_i)d_i + \frac{\epsilon}{1-\epsilon}\sum_{i=1}^{k}\left(\lambda d_i\left(\frac{1}{D}+\frac{1}{kd_i}\right)\sum_{vw\in E}\ell(vw)u(vw)\right)\\
&\le \frac{1}{1-\epsilon}\sum_{i=1}^{k} dist_\ell(s_i,t_i)d_i + \frac{\epsilon}{1-\epsilon}\sum_{i=1}^{k}\left(\frac{d_i}{D}+\frac{1}{k}\right)\lambda\sum_{vw\in E}\ell(vw)u(vw).
\end{aligned}$$

Now observe that there are exactly k commodities and $\sum_{i=1}^{k} d_i = D$, so the last term sums to exactly $\frac{2\epsilon\lambda}{1-\epsilon}\sum_{vw\in E}\ell(vw)u(vw)$. Thus

$$\sum_{P\in S} f_i(P)\ell(P) \le \frac{1}{1-\epsilon}\sum_i dist_\ell(s_i,t_i)d_i + \frac{2\epsilon}{1-\epsilon}\lambda\sum_{vw\in E}\ell(vw)u(vw).$$

Combining the bounds on the sum over ϵ-long and ϵ-short paths,

$$\sum_{i=1}^{k}\sum_{P\in\mathcal{P}_i}\ell(P)f_i(P) \le \frac{1}{1-\epsilon}\sum_i dist_\ell(s_i,t_i)d_i + \frac{2\epsilon}{1-\epsilon}\lambda\sum_{vw\in E}\ell(vw)u(vw) + \epsilon\sum_{i=1}^{k}\sum_{P\in\mathcal{P}_i}\ell(P)f_i(P).$$

Using Lemma 2.3.1 to bound $\lambda\sum_{vw\in E}\ell(vw)u(vw)$ and the equation $\sum_{vw\in E}\ell(vw)f(vw) = \sum_{i=1}^{k}\sum_{P\in\mathcal{P}_i}\ell(P)f_i(P)$, we obtain

$$\sum_{\mathrm{i=1}}^{\mathrm{k}}\sum_{\mathrm{P}\in\mathcal{P}_\mathrm{i}}\ell(\mathrm{P})\mathrm{f_i}(\mathrm{P}) \le \frac{1}{1-\epsilon}\sum_{\mathrm{i}}\mathrm{dist}_\ell(\mathrm{s_i},\mathrm{t_i})\mathrm{d_i} + \frac{2\epsilon}{1-\epsilon}\frac{1+\epsilon}{1-\epsilon}\sum_{\mathrm{i=1}}^{\mathrm{k}}\sum_{\mathrm{P}\in\mathcal{P}_\mathrm{i}}\ell(\mathrm{P})\mathrm{f_i}(\mathrm{P}) + \epsilon\sum_{\mathrm{i=1}}^{\mathrm{k}}\sum_{\mathrm{P}\in\mathcal{P}_\mathrm{i}}\ell(\mathrm{P})\mathrm{f_i}(\mathrm{P}).$$

Combining like terms yields the equation

$$\sum_{i=1}^{k}\sum_{P\in\mathcal{P}_i}\ell(P)f_i(P) \le \frac{\frac{1}{1-\epsilon}}{1-\epsilon-\frac{2\epsilon(1+\epsilon)}{(1-\epsilon)^2}}\sum_{i=1}^{k} dist_\ell(s_i,t_i)d_i.$$

Simple algebra shows that the second term is less than $(1+8\epsilon)\sum_{i=1}^{k} dist_\ell(s_i,t_i)d_i$ if $\epsilon < 1/12$.

```
CONCURRENT(𝓘, ε)
f ← INITIALIZE(𝓘).
while f is not ε-optimal
    f ← DECONGEST(f, ε).
return f
```

Figure 2.5: Algorithm CONCURRENT

■

Combining the previous lemma with Lemma 2.3.1 yields the following theorem.

Theorem 2.3.5 Suppose f and ℓ and ϵ satisfy the Relaxed Optimality Conditions $R1$ and $R2'$ and $\epsilon < 1/12$. Then f is ϵ-optimal, i.e., λ is at most a factor $(1 + 12\epsilon)$ more than the minimum possible.

Proof: Combining Lemma 2.3.1 with Lemma 2.3.4 gives that

$$\lambda \sum_{vw \in E} \ell(vw)u(vw) \leq \left(\frac{(1+8\epsilon)(1+\epsilon)}{1-\epsilon}\right) \sum_{i=1}^{k} dist_\ell(s_i, t_i)d_i.$$

Simple algebra completes the proof. ■

The remainder of this chapter focuses on algorithms that achieve the various relaxed optimality conditions.

2.4 Algorithms for the General Concurrent Flow Problem

In this section we give an algorithm, CONCURRENT, for approximately solving the general concurrent flow problem. In Section 2.4.1, we will bound the time needed in terms of the number of minimum-cost flow computations. For simplicity of presentation, throughout this section we shall use a model of computation that allows the use of exact arithmetic on real numbers and provides exponentiation as a single step. In Section 2.4.2 we will show how to modify our algorithms to work in the standard RAM model. The question of which minimum-cost flow algorithm to use is deferred to Section 2.4.3, in which we show how several different minimum-cost flow algorithms can be used, each of which leads to a different running time.

2.4.1 Solving Concurrent Flow Problems

In this section, we give approximation algorithms for the general concurrent flow problem. We give two algorithms, Concurrent and ScalingConcurrent. The former is the basic algorithm on which we will concentrate on for most of this section. The latter is an algorithm that employs a technique which we call ϵ-scaling and is best when $\epsilon = o(1)$. We defer our discussion of ScalingConcurrent until the end of this section.

We begin with a high level description of our main algorithm Concurrent, which appears in Figure 2.5. Algorithm Concurrent takes as input an instance $\mathcal{I}$ of the concurrent flow problem and an error parameter ϵ, $\epsilon \leq \frac{1}{9}$, and returns an 9ϵ-optimal concurrent flow. The algorithm first calls a procedure Initialize which, given an instance of the concurrent flow problem, returns a $2k$-optimal flow. The remainder of the algorithm consists of a sequence of calls to a procedure called Decongest. Decongest takes as input a flow f that has congestion λ_0 and an error parameter ϵ and returns a flow which is either 9ϵ-optimal or has congestion at most $\lambda_0/2$.

We begin our analysis by bounding the running time of Concurrent in terms of the running time of Initialize and Decongest.

Lemma 2.4.1 Let $T_I = T_I(\mathcal{I})$ be the running time of procedure Initialize, a procedure that returns a $2k$-optimal flow. Let $T_D = T_D(\mathcal{I})$ be the running time of procedure Decongest, a procedure that either returns a 9ϵ-optimal flow or decreases λ by a factor of 2. Then, given an instance $\mathcal{I}$ of a concurrent flow problem, algorithm Concurrent finds an ϵ-optimal solution in $O(T_I + T_D \log k)$ time.

Proof: We first call procedure Initialize to find an initial solution that is $2k$-optimal, i.e., one for which $\lambda \leq 2k\lambda^*$. Each call to Decongest, except for the final one, decreases λ by a factor of 2 and we continue to call Decongest until $\lambda \leq (1+9\epsilon)\lambda^*$. Thus the number of iterations is at most the logarithm of the ratio of the initial and final values, or

$$\log\left(\frac{2k\lambda^*}{(1+9\epsilon)\lambda^*}\right) = O(\log k).$$

■

INITIALIZE($\mathcal{I}, \epsilon$)
for $i = 1 \ldots k$
  if commodity i is simple
  then
    Compute g_i, a maximum flow from s_i to t_i in instance $\mathcal{N} = (G, u, s_i, t_i)$.
(*) $f_i(vw) \leftarrow g_i(vw) \cdot (d_i/|g_i|)\ \forall vw \in E$.
  else
    $\lambda_{\text{low}} \leftarrow \frac{D_i}{mU}$; $\lambda_{\text{high}} \leftarrow nD_i$.
    while $(\lambda_{\text{high}} - \lambda_{\text{low}}) \geq \frac{D_i}{mU}$
      $\lambda_{\text{mid}} \leftarrow (\lambda_{\text{high}} - \lambda_{\text{low}})/2$.
      $u'(vw) \leftarrow u(vw) \cdot \lambda_{\text{mid}}\ \forall vw \in E$.
      if there is a feasible flow in instance $\mathcal{F} = (G, u', \hat{d}_i)$
      then
        $\lambda_{\text{high}} \leftarrow \lambda_{\text{mid}}$
      else
        $\lambda_{\text{low}} \leftarrow \lambda_{\text{mid}}$
    Let $\lambda_i \leftarrow \lambda_{\text{high}}$.
    $u'(vw) \leftarrow u(vw) \cdot \lambda_i\ \forall vw \in E$.
    Let f_i be a feasible flow in instance $\mathcal{F} = (G, u', \hat{d}_i)$.
return f

Figure 2.6: Procedure INITIALIZE

In the remainder of this section we shall describe how to implement the various parts of algorithm CONCURRENT. First, we will describe procedure INITIALIZE, which finds a "good" initial solution to the given concurrent flow problem. Then, we will describe procedure DECONGEST, which takes a flow with congestion λ and produces a new flow that is either 9ϵ-optimal or has congestion at most $\lambda/2$.

Finding an Initial Solution

This section describes procedure INITIALIZE, which takes as input an instance of the concurrent flow problem $\mathcal{I}$ and outputs a flow which is $2k$-optimal. See Figure 2.6. The main idea is that we separately route each commodity i in a good way. The algorithm is broken into two cases. If a commodity i is simple then we find a maximum flow of value $|g_i|$ from s_i to t_i and then scale the flow on each edge by $d_i/|g_i|$. If a commodity is not simple, then a series of maximum flow computations must be performed. We perform a binary search over the range of possible values of λ_i and at each iteration test whether there exists a flow with congestion λ_i. In either case, the flow found for commodity i has congestion $O(\lambda^*)$. Combining all the commodities yields a flow with congestion $O(k\lambda^*)$.

Lemma 2.4.2 Let $T_{\mathrm{MF}} = T_{\mathrm{MF}}(\mathcal{N})$ be the time to compute a maximum flow on instance $\mathcal{N} = (G, u, s, t)$. Then procedure INITIALIZE finds a $2k$-optimal multicommodity flow satisfying demands in $O(k \log(nU) T_{\mathrm{MF}})$ time. Given a simple multicommodity flow problem, INITIALIZE finds a $2k$-optimal multicommodity flow satisfying demands in $O(kT_{\mathrm{MF}})$ time.

Proof: For each $i = 1, \ldots, k$, INITIALIZE finds a flow f_i for the one-commodity concurrent flow problem consisting solely of commodity i. Let λ_i be the congestion of flow f_i and let λ_i^* be the minimum possible value of λ_i. Clearly for each i, $\lambda_i^* \leq \lambda^*$. Assume that INITIALIZE finds a flow with $\lambda_i \leq 2\lambda_i^*$ for each commodity i. Combining the flows for all commodities yields a flow with congestion

$$\sum_{i=1}^{k} \lambda_i \leq \sum_{i=1}^{k} 2\lambda_i^* \leq \sum_{i=1}^{k} 2\lambda^* = 2k\lambda^*.$$

We now show, that for each i, INITIALIZE actually finds such a flow.

Consider first the case when commodity i has a single source and a single sink. The algorithm computes g_i, a maximum flow for instance $\mathcal{N} = (G, u, s, t)$. Let $|g_i|$ be the value of this flow. Then by the maximum-flow minimum-cut theorem [17, 14], there exists a cut with total capacity $|g_i|$. It is easy to see that the smallest amount by which we can multiply the capacity of the cut and have a flow satisfying demands d_i, is $d_i/|g_i|$. Therefore, $\lambda_i^* = d_i/|g_i|$. Further, if we scale the value of the flow on each edge by $d_i/|g_i|$, as in Line (*), we now have a flow that satisfies demands and has congestion $\lambda_i = \lambda_i^*$.

Now consider the case when a commodity is not simple. A single maximum flow computation no longer suffices. However, for a given value of λ_i, say λ_{mid}, it is possible to check whether there is a feasible flow with congestion λ_{mid}. To do so, we multiply each edge capacity by λ_{mid} and then see if there exists a feasible flow in instance $\mathcal{F} = (G, u \cdot \lambda_{\mathrm{mid}}, \hat{d}_i)$. This computation can be carried out via a maximum flow computation in a graph with $O(n)$ nodes and $O(m)$ edges (see, for example, [38]). To find a good value of λ_i, we perform binary search over the range of possible λ_i. The maximum possible value of λ_i is no more than the maximum edge flow divided by the minimum edge capacity. The maximum edge flow is no more than nD_i, the total demand for commodity i. The minimum edge capacity is 1, and hence $\lambda_i \leq nD_i$.

The total amount of flow in the network is D_i and hence some edge must have at least D_i/m flow. The capacities are bounded by U, and hence the minimum possible value λ_i attains is $D_i/(mU)$. Each iteration of the while loop halves the range, and hence in

$$\log\left(\frac{nD_i}{D_i/(mU)}\right) = O(\log(nU))$$

iterations λ_i is within $D_i/(mU)$ of λ_i^*. When we stop we have a flow with congestion at most $\lambda_i^* + D_i/(mU) \leq 2\lambda_i^*$ of optimal. ■

Rerouting Flow

Now, we show how, given a flow, we can iteratively reroute commodities in order to produce a new flow that is closer to optimality. We give a procedure Decongest which takes a flow f with congestion λ_0 and produces a new flow that is either 9ϵ-optimal or has congestion at most $\lambda_0/2$. Decongest consists of a series of iterations of a while loop. We will analyze Decongest by first bounding the number of iterations of the while loop and then bounding the time for one iteration of this while loop. In the remainder of this section, when we use the term iteration we refer to an iteration of this while loop.

Recall that a commodity is called ϵ-bad if its cost is too high. The basic idea is that each iteration of Decongest reroutes an appropriately chosen fraction of the flow of an ϵ-bad commodity onto the edges of a minimum-cost flow associated with this commodity (as described below), in order to reduce congestion. We use a length function $\ell(vw) = e^{\alpha\lambda(vw)}/u(vw)$, where the value of α will be specified later. This length function has the property that the length of an edge vw is a function of the congestion, i.e., the fraction (possibly greater than 1) of the capacity of that edge that is being used. Intuitively, by using lengths as costs in the computation of the minimum-cost flow, we are penalizing edges with high congestion.

One of the important properties of this particular length function is that at the beginning of procedure Decongest, we can choose α so that Relaxed Optimality Condition $R1$ is satisfied and remains satisfied through the execution of procedure Decongest. The act of rerouting flow gradually enforces Relaxed Optimality Condition $R2$. When both conditions are satisfied, Theorem 2.3.5 can be used to infer that f is $O(\epsilon)$-optimal. Alternatively, Decongest

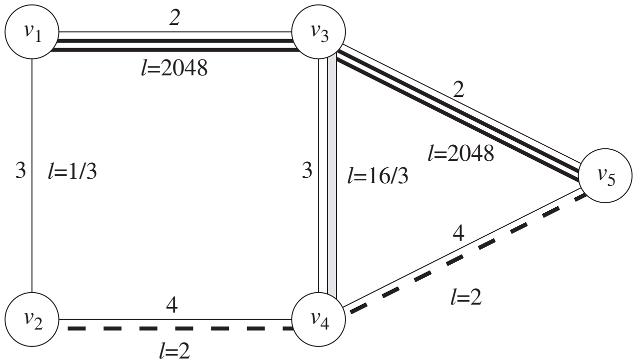
```
DECONGEST(f, ε)
λ0 ← λ; α ← 2(1 + ε)λ0^{-1} ε^{-1} ln(mε^{-1}) .
while λ ≥ λ0/2 and we have not detected that f is 9ε-optimal
        σ ← ε/(8αλ).
        For each edge vw, ℓ(vw) ← e^{αλ(vw)}/u(vw).
(*)     Choose a commodity i as a candidate for rerouting.
        if commodity i is ε-bad
        then
              Formulate an auxiliary minimum-cost flow instance M = (G, λ · u, ℓ, d̂_i).
              Compute f_i^* a minimum-cost flow for M.
              For all vw ∈ E, f_i(vw) ← (1 − σ)f_i(vw) + σf_i^*(vw).
return f
```

Figure 2.7: Procedure DECONGEST

terminates if λ decreases by more than a factor of 2.

More formally, procedure DECONGEST (see Figure 2.7) takes as input a multicommodity flow f with congestion λ_0, where f satisfies the demands, and an error parameter ϵ. In each iteration, we first choose an ϵ-bad commodity i and formulate an auxiliary minimum-cost flow instance $\mathcal{M} = (G, \lambda \cdot u, \ell, \hat{d}_i)$. The demand of each node v in the auxiliary problem is equal to $\hat{d}_i(v)$, and the desired flow $f_i^*(vw)$ is constrained to be between $-\lambda \cdot u(vw)$ and $\lambda \cdot u(vw)$, where λ is the current congestion. The objective is to minimize $C_i^*(\lambda) = \sum_{vw \in E} \ell(vw)\,|f_i^*(vw)|$. Given an optimal solution to this problem, we reroute a fraction $\sigma = \frac{\epsilon}{8\alpha\lambda}$ of the flow f_i onto the edges of f_i^* by setting $f_i(vw) \leftarrow (1-\sigma)f_i(vw) + \sigma f_i^*(vw)$, recompute the length function, and repeat. Upon termination, DECONGEST returns an improved flow f that is either 9ϵ-optimal or has congestion $\lambda \leq \lambda_0/2$.

An example of an iteration of DECONGEST appears in Figures 2.8, 2.9, and 2.10. This example may help provide intuition before proceeding. In Figure 2.8 we have the same flow as in Figure 1.2. In order to highlight the main ideas without using large numbers, we set the values of α and σ somewhat arbitrarily in this example. The algorithm needs to find an ϵ-bad commodity. In order to do so, we first compute the cost of each commodity. This calculation is carried out in the bottom of the figure. Next, we need to compute minimum-cost flows for each commodity. In the example we compute a minimum-cost flow only for commodity 3. The minimum-cost flow problem and its solution are presented in Figure 2.9. We see that the cost of the minimum-cost flow for commodity 3 is much less than the cost of the current flow



commodity	s	t	demand	symbol
1	v_2	v_5	1	— —
2	v_3	v_4	1	▭
3	v_1	v_5	2	▬

$\alpha \leftarrow e^{12 \ln 2} = 2^{12}$

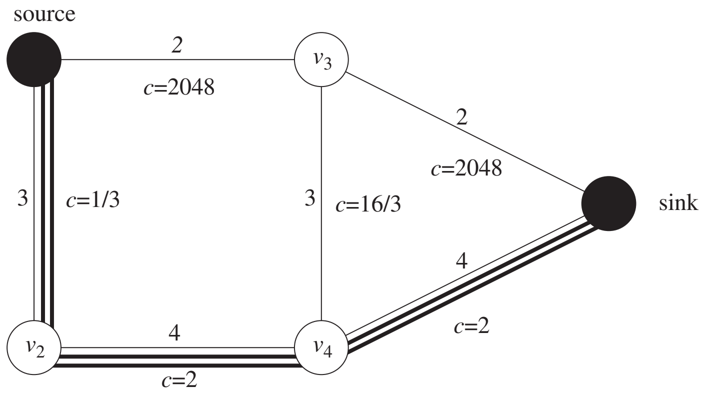
$\sigma \leftarrow \frac{1}{2}$

$l(vw) \leftarrow \frac{e^{\alpha\lambda(vw)}}{u(vw)} = \frac{2^{12\,\lambda(vw)}}{u(vw)}$

$C_i = \sum_{vw} |f_i(vw)|\, l(vw)$

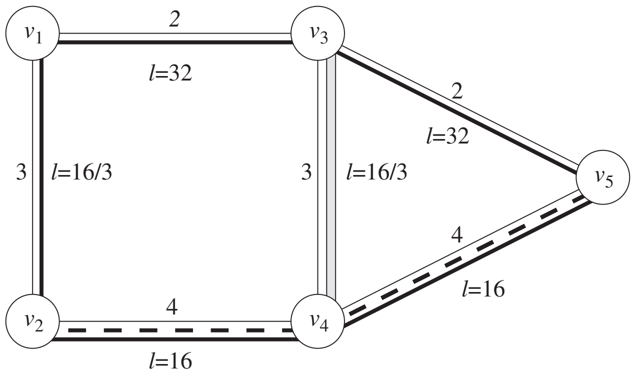
$$
\begin{aligned}
C_1 &= 1(2) + 1(2) &&= 4\\
C_2 &= 1(16/3) &&= 16/3\\
C_3 &= 2(2048) + 2(2048) &&= 8196
\end{aligned}
$$

Figure 2.8: Flows, edge lengths and costs of the current flows



$$C_3^* = 2(4) + 2(4) + 2\left(\frac{1}{3}\right) = \frac{50}{3}$$

Figure 2.9: A minimum-cost flow for commodity 3



commodity	s	t	demand	symbol
1	v_2	v_5	1	
2	v_3	v_4	1	
3	v_1	v_5	2	

$$\begin{aligned} C_1 &= 1(16) + 1(16) &&= 32 \\ C_2 &= 1(16/3) &&= 16/3 \\ C_3 &= 1(32) + 1(32) + \\ &\quad 1(16/3) + 1(16) + 1(16) &&= 304/3 \end{aligned}$$

Figure 2.10: The situation after rerouting

for commodity 3. We could verify that commodity 3 is indeed, ϵ-bad, and hence we should reroute it. Figure 2.10 shows the result of rerouting $\sigma = \frac{1}{2}$ of the flow of commodity 3 onto the edges of the minimum-cost flow for commodity 3. We have also shown the edge lengths and commodity costs. Observe that the cost of commodity 3 has decreased significantly. Also note that although we did not reroute commodity 1, its cost has increased, because commodity 1 is using some of the same edges as commodity 3. This example demonstrates why multicommodity flow problems are difficult: rerouting the flow of one commodity in a better way may result in the flow of another commodity being routed in a worse way.

We now proceed with the analysis. Recall that if Relaxed Optimality Conditions R1 and R2 are satisfied, then the current flow is 9ϵ-optimal. We will first show that R1 is always satisfied. In particular, we now show that if we set $\alpha = 2(1+\epsilon){\lambda_0}^{-1}\epsilon^{-1}\ln(m\epsilon^{-1})$ at the beginning of a call to DECONGEST, then Relaxed Optimality Condition $R1$ is satisfied throughout that call.

Lemma 2.4.3 If f is a multicommodity flow that satisfies demands and $\alpha \geq (1+\epsilon)\lambda^{-1}\epsilon^{-1}\ln(m\epsilon^{-1})$, then f and length function $\ell(vw) = e^{\alpha\lambda(vw)}/u(vw)$ satisfy Relaxed Optimality Condition $R1$.

Proof: We show that if an edge $v'w'$ violates the first part of Relaxed Optimality Condition $R1$ then it must satisfy the second part. If other words if

$$\lambda \cdot u(v'w') > (1+\epsilon)f(v'w'), \tag{2.11}$$

then

$$u(v'w')\ell(v'w') \leq \frac{\epsilon}{m}\sum_{vw\in E}\ell(vw)u(vw),$$

We can use (2.11) to upper bound the length of edge $v'w'$,

$$\ell(v'w') = e^{\alpha f(v'w')/u(v'w')}/u(v'w') \leq e^{\alpha\lambda/(1+\epsilon)}/u(v'w').$$

Let v^*w^* be an edge such that $\lambda(v^*w^*) = \lambda$ and hence $u(v^*w^*)\ell(v^*w^*) = e^{\alpha\lambda}$. Then

$$\frac{\sum_{vw\in E}\ell(vw)u(vw)}{u(v'w')\ell(v'w')} \geq \frac{u(v^*w^*)\ell(v^*w^*)}{e^{\alpha\lambda/(1+\epsilon)}} = \frac{e^{\alpha\lambda}}{e^{\alpha\lambda/(1+\epsilon)}} = e^{\alpha\lambda(1-1/(1+\epsilon))} = e^{\alpha\lambda(\epsilon/(1+\epsilon))}.$$

Now if we plug the lower bound on α into $e^{\alpha\lambda(1-(1/\epsilon))}$, we see that

$$e^{\alpha\lambda(\epsilon/(1+\epsilon))} = e^{2(1+\epsilon)\lambda^{-1}\epsilon^{-1}\ln(m\epsilon^{-1})\lambda(\epsilon/(1+\epsilon))} \geq e^{\ln(m\epsilon^{-1})} \geq \frac{m}{\epsilon},$$

where the penultimate inequality follows by canceling terms. ■

Corollary 2.4.4 Relaxed Optimality Condition R1 is always satisfied throughout a call to procedure DECONGEST.

Proof: At the beginning of procedure DECONGEST, α is set equal to $2(1+\epsilon){\lambda_0}^{-1}\epsilon^{-1}\ln(m\epsilon^{-1})$ and throughout DECONGEST $\lambda > \lambda_0/2$ and hence $\lambda_0^{-1} \geq \lambda^{-1}/2$. Therefore $\alpha \geq (1+\epsilon)\lambda^{-1}\epsilon^{-1}\ln(m\epsilon^{-1})$ throughout. ■

We have just seen that Relaxed Optimality Condition R1 is always satisfied. By the contrapositive of Theorem 2.3.3, if the current flow is not ϵ-optimal, then Relaxed Optimality Condition R2 must be violated. The key to showing the efficiency of our algorithm will be to show that when R2 is not satisfied, then one iteration of DECONGEST makes "progress." Although the overall goal of our algorithm is to make progress by decreasing the congestion λ, each iteration of our algorithm need not actually decrease λ. In order to measure progress of our algorithm, we introduce a potential function $\Phi = \Phi(u, f, \ell)$. We will specify a set of conditions on this potential function that will suffice to derive a good bound on the number of iterations of procedure DECONGEST. We will then give a particular potential function and show that it meets these criterion.

Notice that the termination condition of the while loop is "$\lambda \geq \lambda_0/2$ **and** we have not detected that f is 9ϵ-optimal." It is trivial to detect when $\lambda \leq \lambda_0/2$, and hence we can assume that as soon as $\lambda \leq \lambda_0/2$, the call to DECONGEST terminates. To check whether f is 9ϵ-optimal is not as easy and in fact, if not done carefully, can dominate the running time of DECONGEST. For ease of presentation, we assume, for now, that as soon as the condition of the while loop in DECONGEST is not satisfied, the algorithm detects it. In other words, at the beginning of each iteration, the current flow f is not 9ϵ-optimal. In particular, since Relaxed Optimality Condition R1 is always satisfied, at the beginning of each iteration Relaxed Optimality Condition R2 is not satisfied. For the deterministic algorithms, at the beginning of each iteration, R2 actually is not satisfied whereas for the randomized algorithms, R2 may be satisfied at the beginning of an iteration. In either case, we will eventually show how to remove this assumption.

Let λ_0 be congestion of the initial flow passed to DECONGEST. Let $f^0, \ldots, f^p$ be the sequence of flows at the beginning of each iteration of DECONGEST. We call an iteration j *productive* if the commodity i chosen on line (*) of DECONGEST is ϵ-bad and *unproductive* otherwise.

We now state a trivial lemma which captures the different factors that affect the running time of an implementation of DECONGEST.

Lemma 2.4.5 For a call to DECONGEST, let I_P be the number of productive iterations and I_U be the number of unproductive iterations. Let T_P be the time spent in one productive iteration, and T_U be the time spent in one unproductive iteration. Assume that the procedure terminates as soon as f is 9ϵ-optimal. Then the running time of DECONGEST is

$$O(I_P T_P + I_U T_U). \tag{2.12}$$

■

In most cases, the dominant term is $I_P T_P$, the time spent in productive iterations. We proceed to bound I_P first.

Let $\Phi^0, \ldots, \Phi^p$ be the values that potential function Φ takes on during successive iterations. We call a potential function Φ *useful* if throughout a call to procedure DECONGEST it satisfies the following four conditions:

U1) $\Phi^0 \leq m e^{\alpha \lambda_0}$,

U2) $\Phi^p \geq e^{\alpha \lambda_0 / 2}$,

U3) If iteration j is unproductive then $\Phi^j = \Phi^{j+1}$, $j = 0, \ldots, p-1$.

U4) If iteration j is productive then $\Phi^j - \Phi^{j+1} = \Omega(\frac{\epsilon^2}{k}\Phi)$, $j = 0, \ldots, p-1$.

Note that the existence of a useful potential function actually implies something about the performance of DECONGEST.

We now show that if a potential function is useful, then we can establish a bound on the number of productive iterations on the while loop during one call to DECONGEST.

Lemma 2.4.6 Let Φ be a useful potential function. Then procedure DECONGEST terminates after $O(\epsilon^{-3}k\log n)$ productive iterations. If the initial flow is $O(\epsilon)$-optimal, then DECONGEST terminates after $O(\epsilon^{-2}k\log n)$ productive iterations.

Proof: First we bound the number of times Φ can be reduced by a factor of e throughout a call to DECONGEST. Since Φ is useful, initially, $\Phi^0 \leq me^{\alpha\lambda_0}$ and in the last iteration $\Phi^p \geq e^{\alpha\lambda_0/2}$. By $U3$ and $U4$, Φ never increases. Thus the number of times Φ can decrease by a constant factor is just the logarithm of the ratio of the initial and final values of Φ, which is

$$\log\left(\frac{me^{\alpha\lambda_0/2}}{e^{\alpha\lambda_0}}\right) = O(\alpha\lambda_0 + \log m) = O(\alpha\lambda_0).$$

The last equality follows by plugging in the value of α specified in DECONGEST.

By U4, each productive iteration results in a reduction in Φ of $\Omega(\frac{\epsilon^2}{k}\Phi)$. Since $1-x \leq e^{-x}$, it follows that every $O(k\epsilon^{-2})$ iterations reduce Φ by at least a factor of e.

Multiplying the number of productive iterations it takes to reduce Φ by a factor of e by the number of times Φ can be reduced by a factor of e in order to decrease by a constant factor, we see that DECONGEST executes $O(\alpha\lambda_0 k\epsilon^{-2})$ productive iterations. Plugging in the value of α, we get that the number of productive iterations is

$$O(\alpha\lambda_0 k\epsilon^{-2}) = O(\epsilon^{-1}\lambda^{-1}\ln(n\epsilon^{-1})\lambda_0 k\epsilon^{-2}) = O\left(\epsilon^{-3}\frac{\lambda_0}{\lambda}\ln(n\epsilon^{-1})\right).$$

We have assumed that ϵ is at least inverse polynomial in n, and we maintain that $\lambda \geq \lambda_0/2$, so the number of productive iterations is in fact $O(\epsilon^{-3}k\log n)$.

If the initial flow is $O(\epsilon)$-optimal then we know that $\lambda^* \geq \lambda_0/(1+O(\epsilon))$, so throughout DECONGEST, λ never goes below $\lambda_0/(1+O(\epsilon))$. Thus, we have the tighter bound on the possible range of the potential function of $e^{\alpha(1+O(\epsilon))^{-1}\lambda_0} \leq \Phi \leq me^{\alpha\lambda_0}$. So to decrease the potential function by a constant factor takes

$$O\left(\log\left(\frac{me^{\alpha\lambda_0}}{e^{\alpha(1+O(\epsilon))^{-1}\lambda_0}}\right)\right) = O(\epsilon\alpha\lambda + \log m)$$

productive iterations. Continuing as above, we get that the number of iterations is $O(\epsilon^{-2}k\log n)$.
∎

We use the particular potential function $\Phi = \sum_{vw\in E} u(vw)\ell(vw)$. To complete the proof that DECONGEST terminates after a small number of productive iterations, we need to show that Φ is useful. We begin by showing that it satisfies the first three conditions in the definition of useful.

Lemma 2.4.7 Let $\Phi = \sum_{vw\in E} u(vw)\ell(vw)$. Then throughout one call to DECONGEST, U1 and U2, U3 are satisfied.

Proof: Initially for each edge vw, $\lambda(vw) \leq \lambda_0$, thus

$$u(vw)\ell(vw) = e^{\alpha\lambda(vw)} \leq e^{\alpha\lambda_0},$$

and

$$\Phi^0 = \sum_{vw\in E} u(vw)\ell(vw) \leq \sum_{vw\in E} e^{\alpha\lambda_0} \leq me^{\alpha\lambda_0}.$$

At the beginning of the last iteration, DECONGEST has not terminated, and therefore $\lambda \geq \lambda_0/2$. Thus here must be at least one edge $v'w'$ for which $\lambda(v'w') \geq \lambda_0/2$. Since $\ell(vw)$ and $u(vw)$ are always non-negative, $u(vw)\ell(vw)$ is non-negative for each edge vw and hence

$$\Phi^p = \sum_{vw\in E} u(vw)\ell(vw) \geq u(v'w')\ell(v'w') \geq e^{\alpha\lambda_0/2}.$$

Finally, if iteration j is non-productive, then commodity i is not ϵ-bad and no rerouting takes place. Hence, neither the flow nor the length function changes and Φ remains unchanged. ■

The following lemma establishes that the potential function $\Phi = \sum_{vw\in E} u(vw)\ell(vw)$ satisfies U4. This lemma is the heart of the analysis of DECONGEST.

Lemma 2.4.8 Let $\epsilon \leq 1$ and $\frac{\epsilon}{16\alpha\lambda} \leq \sigma \leq \frac{\epsilon}{8\alpha\lambda}$. Let j be a productive iteration and let i be an ϵ-bad commodity, and let f_i^* be a minimum-cost flow for this commodity as computed in DECONGEST. Let the new flow for commodity i be defined by $f_i(vw) \leftarrow (1-\sigma)f_i(vw) + \sigma f_i^*(vw)$. Then, $\Phi^j - \Phi^{j+1} \geq \Omega(\frac{\epsilon^2}{k}\Phi^j)$.

Proof: Denote by $\ell(vw)$ and $\ell'(vw)$ the length of edge vw before and after rerouting, respectively. Let $\delta(vw)$ denote the increase in flow on vw due to rerouting. Recall that, after rerouting, the flow of the rerouted commodity i on vw is $|(1-\sigma)f_i(vw) + \sigma f_i^*(vw)|$, and hence

$|\delta(vw)| \leq \sigma|f_i^*(vw) - f_i(vw)| \leq \sigma(|f_i(vw)| + |f_i^*(vw)|)$. Moreover, since both f_i and f_i^* have congestion at most λ, $|\delta(vw)| \leq 2\sigma\lambda u(vw)$.

By definition of the length function,

$$\ell'(vw) = e^{\alpha(f(vw)+\delta(vw))/u(vw)}/u(vw) = e^{\alpha f(vw)/u(vw)+\eta}/u(vw),$$

where $\eta = \alpha\delta(vw)/u(vw)$. Observe that $|\eta| \leq 2\alpha\sigma\lambda \leq \epsilon/4 \leq 1/4$. Using the Taylor series, we see that $|\eta| \leq \epsilon/4 \leq 1/4$ implies that for all x, $e^{x+\eta} \leq e^x + \eta e^x + \frac{\epsilon}{2}|\eta|e^x$. Therefore, we have:

$$\begin{aligned} \ell'(vw) &\leq \ell(vw) + \eta\ell(vw) + \frac{\epsilon}{2}|\eta|\ell(vw) \\ &\leq \frac{\alpha\sigma(|f_i^*(vw)| - |f_i(vw)|)}{u(vw)}\ell(vw) + \frac{\epsilon\alpha\sigma(|f_i(vw)| + |f_i^*(vw)|)}{2u(vw)}\ell(vw). \end{aligned}$$

We use this bound to give a lower bound on the decrease in the potential function.

$$\begin{aligned} \Phi^j - \Phi^{j+1} &= \sum_{vw\in E}(\ell(vw) - \ell'(vw))u(vw) \\ &\geq \alpha\sigma\sum_{vw}(|f_i(vw)| - |f_i^*(vw)|)\ell(vw) - \alpha\sigma\frac{\epsilon}{2}\sum_{vw}(|f_i(vw)| + |f_i^*(vw)|)\ell(vw). \end{aligned}$$

By the definitions of C_i and $C_i^*(\lambda)$, $C_i = \sum_{vw\in E} f_i(vw)\,|\ell(vw)|$ and $C_i^*(\lambda) = \sum_{vw\in E} f_i^*(vw)\,|\ell(vw)|$, hence we can rewrite the last bound as

$$\Phi^j - \Phi^{j+1} \geq \alpha\sigma(C_i - C_i^*(\lambda)) - \alpha\sigma\frac{\epsilon}{2}(C_i + C_i^*(\lambda)).$$

Since $C_i^*(\lambda) \leq C_i$, we can bound the last term by $\alpha\sigma\epsilon C_i$. We can use the definition of an ϵ-bad commodity to establish a lower bound on the $C_i - C_i^*(\lambda)$ that appears in the first term on the righthand side. Plugging in we get

$$\Phi^j - \Phi^{j+1} \geq \alpha\sigma(C_i - C_i^*(\lambda)) - \alpha\sigma\epsilon C_i \geq \alpha\sigma\left(\epsilon C_i + \epsilon\lambda\frac{\sum_{vw}\ell(vw)u(vw)}{k}\right) - \alpha\sigma\epsilon C_i = \frac{\alpha\sigma\epsilon\lambda}{k}\Phi^j. \tag{2.13}$$

Plugging in the value of σ from the statement of the lemma, we get that

$$\Phi^j - \Phi^{j+1} \geq \frac{\alpha\sigma\epsilon\lambda}{k}\Phi^j \geq \frac{\alpha\epsilon^2\lambda}{16\alpha\lambda k}\Phi^j = \Omega\left(\frac{\epsilon^2}{k}\Phi^j\right).$$

■

Combining Lemmas 2.4.6, 2.4.7, and 2.4.8 we get the following lemma:

Lemma 2.4.9 Assume that as soon as f is 9ϵ-optimal, DECONGEST terminates. Then $I_P = O(\epsilon^{-3}k\log n)$ if the initial flow is arbitrary and $O(\epsilon^{-2}k\log n)$ if the initial flow is $O(\epsilon)$-optimal.

This bound on I_P holds for both the randomized and the deterministic version of DECONGEST.

Implementations of DECONGEST

We now give two different implementations of DECONGEST, a deterministic implementation and a more efficient randomized one. For each one, we will explain the algorithm and then bound I_U, T_P, and T_U. We will also discuss the issue of detecting when f is 9ϵ-optimal. For all variations, the only computation-intensive part of an iteration of DECONGEST, be it productive or unproductive, is finding an ϵ-bad commodity and computing minimum-cost flows. All the rest can be done in $O(m)$ time. Thus, we will concentrate on finding an ϵ-bad commodity and computing minimum-cost flows. Throughout this section, we treat the minimum-cost flow subroutine as a black box, and shall discuss its implementation later.

Before beginning, we discuss how to perform a termination check in all variants of the algorithm.

Lemma 2.4.10 Let $T_{\mathrm{MCF}} = T_{\mathrm{MCF}}(\mathcal{M})$ be the time to compute a minimum cost flow for instance $\mathcal{M}$. Then given a flow f, we can determine if f is 9ϵ-optimal in $O(k(T_{\mathrm{MCF}}))$ time.

Proof: In order to detect termination, we can compare λ to

$$\sum_{i=1}^{k} C_i^*(\lambda)/(\sum_{vw\in E}\ell(vw)u(vw)), \tag{2.14}$$

the lower bound given by Lemma 2.2.4. The numerator of (2.14) can be computed by computing k minimum-cost flows. The denominator can be computed in $O(m)$ time. ■

Now we describe the straightforward deterministic implementation. The simplest way to find an ϵ-bad commodity is to compute the costs $C_i = \sum_{vw \in E} \ell(vw) \, |f_i(vw)|$ and the costs of the minimum-cost flows C_i^* and compare them to see if commodity i is ϵ-bad. In the worst case we need to check all k commodities. Computing the cost of one commodity takes $O(m)$ time, and therefore the costs of all commodities can be computed in $O(km)$ time. Hence, an iteration can be implemented in the time it takes to perform k minimum-cost flow computations plus $O(km)$ additional time. After each iteration, we can perform a termination check. By Lemma 2.4.10, the time for a termination check is the same as the time for an iteration, and hence including the time spent performing termination checks does not increase the asymptotic running time. Further, since we perform a termination check after each iteration, we claim that every iteration is productive. At the start of an iteration, the flow is not 9ϵ-optimal, and by Lemma 2.4.3, Relaxed Optimality Condition R1 is always satisfied. Thus by the contrapositive of Theorem 2.3.3, Relaxed Optimality Condition R2 must not hold and there must be an ϵ-bad commodity. Since in each iteration, we check each commodity to see if it is ϵ-bad, if one exists, we will find it.

We summarize this discussion in the following lemma:

Lemma 2.4.11 Let $T_{\mathrm{MCF}} = T_{\mathrm{MCF}}(\mathcal{M})$ be the time to compute a minimum cost flow for instance $\mathcal{M}$. Procedure DECONGEST can be implemented to run in $O(\epsilon^{-3}k^2 \log n(T_{\mathrm{MCF}}))$ time. If the initial flow is $O(\epsilon)$-optimal, DECONGEST can be implemented to run in $O(\epsilon^{-2}k^2 \log n(T_{\mathrm{MCF}}))$ time.

Proof: By the above discussion, $T_P = O(kT_{\mathrm{MCF}})$, $I_U = 0$, and as soon as f is 9ϵ-optimal, DECONGEST terminates. Plugging these bounds and the bounds of Lemma 2.4.6 into equation (2.12) yields the lemma. ■

Deterministically, it seems necessary to know the values of the k minimum-cost flows in each iteration. However, by using a simple randomized strategy, we can show that it is necessary to compute only expected $O(\epsilon^{-1})$ minimum-cost flows in each iteration. When $\epsilon^{-1} = o(k)$, for example when ϵ is a fixed constant, this randomized strategy leads to faster algorithms.

We begin by giving the strategy:

Randomized Strategy 1: When choosing a commodity to reroute, choose a commodity with probability proportional to its cost, i.e., Pr[commodity i is chosen] $= C_i/C$, where $C = \sum_{j=1}^{k} C_j$.

Lemma 2.4.12 Suppose f is not 9ϵ-optimal and a commodity i is chosen at random using Randomized Strategy 1. Then $\Pr[i \text{ is } \epsilon\text{-bad}] \geq \epsilon$. If we implement DECONGEST using Randomized Strategy 1 then $E[I_U] = O(\epsilon^{-1} I_P)$, and we can amortize the running times so that $T_U = O(T_{\text{MCF}} + k)$ and $T_P = O(T_{\text{MCF}} + mk)$.

Proof: Let $C = \sum_{j=1}^{k} C_j$. If the flow is not 9ϵ-optimal, then it must be the case that $R2$ is violated, i.e.,

$$\sum_{j\ \epsilon\text{-bad}} C_j > \epsilon C. \tag{2.15}$$

We choose a commodity i with probability proportional to its cost, i.e., with probability C_i/C. The probability that a commodity chosen in this manner is bad is just

$$\Pr[i \text{ is } \epsilon\text{-bad}] \;=\; \sum_{j\ \epsilon\text{-bad}} C_j/C \;>\; \epsilon C/C \;=\; \epsilon,$$

where the inequality follows from (2.15). Thus with probability at least ϵ, we have chosen an ϵ-bad commodity.

In each iteration, the probability of finding an ϵ-bad commodity is at least ϵ, thus in expected $O(\epsilon^{-1})$ iterations, we will find such a commodity. The iteration in which we find an ϵ-bad commodity is productive while the rest are unproductive, so $E[I_U] = O(\epsilon^{-1} I_P)$. Consider a sequence of unproductive iterations followed by one productive iteration. At the beginning of this sequence, we compute, in $O(km)$ time, the values C_i for $i = 1, \ldots, k$. We then execute a sequence of unproductive iterations. Each involves choosing a commodity, which can be performed in $O(k)$ time, and then computing one minimum-cost flow. Thus each one of these iterations takes $O(T_{\text{MCF}} + k)$ time. Finally, we actually execute a productive iteration in $O(T_{\text{MCF}} + k)$ time. We can charge the computation of the commodity costs to this productive iteration, so $T_P = O(mk + T_{\text{MCF}})$. ■

Corollary 2.4.13 Let $T_{\text{MCF}} = T_{\text{MCF}}(\mathcal{M})$ be the time to compute a minimum cost flow for instance $\mathcal{M}$ and assume that DECONGEST terminates as soon as f is 9ϵ-optimal. Then Procedure DECONGEST can be implemented to run in expected $O(\epsilon^{-3}k\log n(\epsilon^{-1}T_{\text{MCF}}+\epsilon^{-1}k+mk))$ time. If the initial flow is $O(\epsilon)$-optimal, in can be implemented to run in expected $O(\epsilon^{-2}k\log n(\epsilon^{-1}T_{\text{MCF}}+\epsilon^{-1}k+mk))$ time.

Proof: Plug the values from Lemma 2.4.12 and 2.4.6 into (2.12). ■

Observe that if $k \leq n$ (this is the case when Lemma 2.2.1 is applied) the time to compute the cost of all current flows is dominated by the time to compute a minimum-cost flow, and the mk and ϵ^{-1} terms disappear from the bound in Corollary 2.4.13. On the other hand, if k is large, then the dominant step may be computing the costs and choosing a commodity. In this case, we can reduce the time by using a somewhat more involved strategy:

Randomized Strategy 2: Pick an edge with probability proportional to the product of the length of the edge and the flow through this edge. Let the chosen edge be vw. Then choose a commodity with probability proportional its flow on edge vw. In other words, let $F = \sum_{vw\in E} f(vw)\ell(vw)$. Choose an edge $vw \in E$ with $\Pr[vw \text{ is chosen}] = \ell(vw)f(vw)/F$. Choose a commodity i with $\Pr[i \text{ is chosen}] = |f_i(vw)|/f(vw)$.

We now show that this strategy still chooses commodity i with probability proportional to its cost and reduces the time for random selection from $O(km)$ to the minimum of $O(m+k)$ and $O(m\log k)$. By doing so, we are actually picking a commodity with probability proportional to its cost, *without ever explicitly computing these costs.*

Lemma 2.4.14 Suppose a commodity i is chosen according to Randomized Strategy 2. Then $\Pr[i \text{ is } \epsilon\text{-bad}] \geq \epsilon$. Assume that we implement DECONGEST using Randomized Strategy 2 and terminate as soon as f is 9ϵ-optimal. Then, for this strategy $E[I_U] = O(\epsilon^{-1}I_P)$.

Proof: Let $F = \sum_{vw \in E} f(vw)\ell(vw)$. The event that i is chosen is just the sum of m independent events, one for each edge. Thus

$$\begin{aligned} \Pr[i \text{ is chosen}] &= \sum_{vw \in E} \Pr[vw \text{ is chosen}] \Pr[i \text{ is chosen} \mid vw \text{ is chosen}] \\ &= \sum_{vw \in E} \frac{\ell(vw) f(vw)}{F} \frac{|f_i(vw)|}{f(vw)} \\ &= \sum_{vw \in E} \frac{|f_i(vw)|\ell(vw)}{F}. \end{aligned}$$

But $\sum_{vw \in E} |f_i(vw)|\ell(vw) = C_i$ and $F = \sum_{i=1}^{k} C_i$, so we are choosing a commodity with exactly the same probability as in Randomized Strategy 1. The lemma follows. ■

We now give two ways to implement Randomized Strategy 2. The first is the straightforward one in which we pick an edge and then pick a commodity going through that edge. In each iteration, we first choose a random number $x \in [0,1]$ and then find the smallest i for which

$$a_i = \sum_{vw=1}^{i} \frac{\ell(vw) f(vw)}{F} \geq x. \tag{2.16}$$

This procedure gives us an edge with the right probability. Given the value of a_{i-1}, a_i can be computed in $O(1)$ time, and thus $a_1, \ldots, a_m$ can be computed in $O(m)$ time. Analogously, we can choose a commodity using this edge vw by choosing a random number $y \in [0,1]$ and then finding the smallest i for which

$$b_i = \sum_{j=1}^{i} \frac{|f_i(vw)|}{f(vw)} \geq y. \tag{2.17}$$

The b_i, $i = 1, \ldots, k$ can be computed in $O(k)$ time. Once we have chosen a commodity, a minimum-cost flow is computed in $O(T_{\mathrm{MCF}})$ time. If it turns out that the commodity is ϵ-bad, it is rerouted. Thus we have shown:

Lemma 2.4.15 An iteration of DECONGEST can be implemented using Randomized Strategy 2 so that $T_U = T_P = O(T_{\mathrm{MCF}} + k + m) = O(T_{\mathrm{MCF}} + k)$.

This yields the following corollary:

Corollary 2.4.16 Let $T_{\mathrm{MCF}} = T_{\mathrm{MCF}}(\mathcal{M})$ be the time to compute a minimum cost flow for instance $\mathcal{M}$and assume that DECONGEST terminates as soon as f is 9ϵ-optimal. Then Procedure DECONGEST can be implemented to run in expected $O(\epsilon^{-4}k \log n(T_{\mathrm{MCF}} + m + k))$ time. If the initial flow is $O(\epsilon)$-optimal, in can be implemented to run in expected $O(\epsilon^{-3}k \log n(T_{\mathrm{MCF}} + m + k))$ time.

Proof: Plug the values from Lemmas 2.4.15 and 2.4.6 into (2.12). ∎

Alternatively, we can use slightly more involved data structures and derive bounds that have depend on $\log k$ rather than k.

Lemma 2.4.17 An iteration of DECONGEST can be implemented using Randomized Strategy 2 so that $T_U = O(T_{\mathrm{MCF}} + \log k + m) = O(T_{\mathrm{MCF}} + \log k)$ and $T_P = O(T_{\mathrm{MCF}} + m \log k)$.

Proof: In the previous strategy, at each iteration, given a random value y, we had to check (2.17) for all k commodities. To perform this computation more efficiently, we can store the values $|f_i(vw)|/f(vw)$ in a balanced binary tree. There is one tree for each edge vw. In tree vw, leaf i, $i = 1, \ldots, k$ contains the value $|f_i(vw)|/f(vw)$. Each internal node of the tree contains the sum of the leaf values in its subtree. It's well known that given such a data structure, the leftmost leaf satisfying (2.17) can be found in $O(\log k)$ time. In order to maintain these data structures, we must update the appropriate trees each time a flow value changes. The only changes occur when flow is rerouted and each rerouting only changes the value of the flow for one commodity on at most m edges. Therefore, the updates associated with one routing step can be accomplished in $O(m \log k)$ time. Each unproductive iteration performs one tree search and one minimum-cost flow, while each productive iteration performs one tree search, one minimum-cost flow and one rerouting. The lemma follows. ∎

We can summarize this in the following corollary:

Corollary 2.4.18 Let T_{MCF} be the time to compute a minimum-cost flow and assume that DECONGEST terminates as soon as f is 9ϵ-optimal. Then Procedure DECONGEST can be implemented to run in expected $O(\epsilon^{-4}k \log n(T_{\mathrm{MCF}} + m \log k))$ time. If the initial flow is $O(\epsilon)$-optimal, in can be implemented to run in expected $O(\epsilon^{-3}k \log n(T_{\mathrm{MCF}} + m \log k))$ time.

Proof: Plug the values from Lemmas 2.4.17 and 2.4.6 into (2.12). ∎

In the analysis above, we have assumed that the algorithm terminates as soon as the flow is 9ϵ-optimal. In order to guarantee this condition, we would have to perform a check after each iteration. However, if we did perform a check after each iteration, we would spend more time performing checks than rerouting flow. If the termination checks are not to dominate, we need a strategy that checks for termination only 1 out of every k iterations. To do so, after each iteration, with probability $1/k$, we perform an termination check.

Lemma 2.4.19 Assume that, with probability $1/k$, after each iteration of DECONGEST we check whether f is 9ϵ-optimal. Then the time bounds given in Corollaries 2.4.13 and 2.4.18 hold without any assumptions about termination.

Proof: We divide the iterations of the algorithm into two sets, those in which the flow is 9ϵ-optimal and those in which it is not.

First we focus on the case when the flow is not 9ϵ-optimal. From Lemmas 2.4.15 and 2.4.17, we see that in all cases, if the current flow f is not 9ϵ-optimal, then one iteration of DECONGEST takes $\Omega(T_{\mathrm{MCF}})$ time. We now add to these iterations, a termination check, with probability $1/k$. By Lemma 2.4.10, a termination check takes $\Omega(kT_{\mathrm{MCF}})$ time, and therefore the expected running time increases by at most a constant factor.

If f is 9ϵ-optimal, then the termination check recognizes it. However, since we only check with probability $1/k$, we expect to execute $O(k)$ iterations in which the flow is 9ϵ-optimal but the termination check does not recognize it. But for both Randomized Strategy 1 and Randomized Strategy 2, $I_P = \Omega(k)$ and $I_U = \Omega(k)$. Therefore adding k additional iterations does not increase the asymptotic running time. ■

We now summarize the main results of this section.

Theorem 2.4.20 Let $T_{\mathrm{MCF}} = T_{\mathrm{MCF}}(\mathcal{M})$ be the time to compute a minimum cost flow for instance $\mathcal{M}$. Then assuming that exponentiation can be done in $O(1)$ time, the following table gives the times for procedure DECONGEST:

	initial flow arbitrary	initial flow $O(\epsilon)$-optimal
Randomized	$O(\epsilon^{-4}k\log n(T_{\mathrm{MCF}}+m\log k))$	$O(\epsilon^{-3}k\log n(T_{\mathrm{MCF}}+m\log k))$
	$O(\epsilon^{-4}k\log n(T_{\mathrm{MCF}}+m+k))$	$O(\epsilon^{-3}k\log n(T_{\mathrm{MCF}}+m+k))$
Deterministic	$O(\epsilon^{-3}k^2\log nT_{\mathrm{MCF}})$	$O(\epsilon^{-2}k^2\log nT_{\mathrm{MCF}})$

Proof: This table summarizes Corollary 2.4.16 and 2.4.18 and Lemma 2.4.11. ■

Putting it all together: A Summary of Algorithm Concurrent

We can now give running times for algorithm Concurrent. We will give two sets of running times, one when the input is a simple k-commodity concurrent flow problem, and one when the input is a non-simple k^*-commodity concurrent flow problem. Given an instance of a simple k-commodity concurrent flow problem, we have two options. One option is to use the bounds for the simple concurrent flow problem. Alternatively, we can use Lemma 2.2.1 to create a k^*-commodity non-simple instance, use the time bounds for the non-simple concurrent flow problem, and then decompose the solution. This procedure takes $O(k^*m\log n+kn)$ time plus the time to solve a non-simple k^*-commodity concurrent flow problem. Although this second method may lead to faster running times, throughout the rest of the chapter, we do not carry this calculation through. Also, we use the randomized running times given in the first line of the chart in Theorem 2.4.20 for simple instances and the second line for non-simple instances. Even though for some simple instances, the second line of randomized running times given in the chart in Theorem 2.4.20 gives faster running times, do not carry through this calculation.

Theorem 2.4.21 Let $T_{\mathrm{MCF}} = T_{\mathrm{MCF}}(\mathcal{M})$ be the time to compute a minimum cost flow for instance $\mathcal{M}$. Then assuming that exponentiation can be performed in $O(1)$ time, the following table gives the running times for Algorithm Concurrent:

	simple instance	non-simple instance
Randomized	$O(\epsilon^{-4}k\log k\log n(T_{\mathrm{MCF}}+m\log k))$	$O(k^*nm\log\left(\frac{n^2}{m}\right)\log(nU)$ $+\epsilon^{-4}k^*\log k^*\log n(T_{\mathrm{MCF}}+m))$
Deterministic	$O(\epsilon^{-3}k^2\log k\log n(T_{\mathrm{MCF}}+m\log k))$	$O(k^*nm\log\left(\frac{n^2}{m}\right)\log(nU)$ $+(\epsilon^{-3}k^{*2}\log k^*\log n(T_{\mathrm{MCF}}+m)))$

Proof: Combine Lemma 2.4.1, 2.4.2, Corollary 2.4.16, Corollary 2.4.18 and the fact that a maximum flow in an n-node m-edge graph can be computed in $O(nm \log(n^2/m))$ time [23]. Note that only in the non-simple case does the time for initialization appear in the final time bounds. ■

A scaling algorithm

The dependence on ϵ given in Theorem 2.4.21 can be reduced somewhat, through a technique we call ϵ-scaling. Instead of calling DECONGEST with the value of ϵ given in the input, we call it with a series of values of ϵ, each $\frac{1}{2}$ of the previous value. The advantage of scaling is that at the beginning of each call to DECONGEST the initial flow is $O(\epsilon)$-optimal. Thus we can employ the bounds on DECONGEST from the second column for the table in Theorem 2.4.20 to obtain faster running times. The ϵ scaling we use is similar to that used by Goldberg and Tarjan in their minimum-cost flow algorithm[22]. The details of our scaling algorithm, SCALINGCONCURRENT, appear in Figure 2.11.

Lemma 2.4.22 Let $T_C(\epsilon) = T_C(\epsilon, \mathcal{I})$ be the running time of CONCURRENT, on input $(\epsilon, \mathcal{I})$. Let $T_D = T_D(\mathcal{I}, \epsilon)$ be the running time of procedure DECONGEST given an $O(\epsilon)$-optimal input flow. Then, given an instance $\mathcal{I}$ of a concurrent flow problem, algorithm SCALINGCONCURRENT finds an ϵ-optimal solution in $O(T_C(\frac{1}{9}) + T_D(\epsilon))$ time.

Proof:

First we find an 1-optimal multicommodity flow using algorithm CONCURRENT, with $\epsilon = 1/9$. The rest of the computation is divided into scaling phases. We start each phase by dividing ϵ by 2. Thus our current flow is 18ϵ-optimal with respect to the new ϵ. The bounds given in the second column of Theorem 2.4.20 imply that the running time of DECONGEST, given an $O(\epsilon)$-optimal solution is proportional to ϵ^{-c}, where c is 2 or 3, depending on whether the algorithm is randomized or deterministic. Since in each subsequent call to DECONGEST, ϵ decreases by a factor of 2, the the running times form a geometric series in ϵ^{-1}. Hence the running time for the series is dominated by twice the time for the last iteration. ■

Goldberg [21] and Grigoriadis and Khachiyan [27] have shown how to reduce the running time of our randomized algorithms by an ϵ^{-1} factor. Goldberg gives a somewhat simplified

```
SCALINGCONCURRENT(I, ε)
f ← CONCURRENT(I, 1/9).
ε' = 1/9
while (ε' ≥ ε)
        ε' ← ε'/2.
        f ← DECONGEST(f, ε').
return f
```

Figure 2.11: Algorithm CONCURRENT

version of our proof that leads to a randomized selection strategy that avoids having to search for an ϵ-bad commodity. Grigoriadis and Khachiyan generalize our algorithm to solve certain types of convex programming problems. Their algorithm, when specialized to the case of solving multicommodity flows, also avoids searching for an ϵ-bad commodity.

We now summarize the results for Algorithm SCALINGCONCURRENT so far:

Theorem 2.4.23 Let $T_{\text{MCF}} = T_{\text{MCF}}(\mathcal{M})$ be the time to compute a minimum cost flow for instance $\mathcal{M}$. Then the following table gives the running times of Algorithm SCALINGCONCURRENT, assuming exponentiation can be implemented in $O(1)$ time:

	simple instance	non-simple instance
Randomized	$O((\epsilon^{-3} + \log k)k \log n(T_{\text{MCF}} + m \log k))$	$O(k^* nm \log\left(\frac{n^2}{m}\right) \log(nU)$ $+(\epsilon^{-3} + \log k^*)k^* \log n(T_{\text{MCF}} + m))$
Deterministic	$O((\epsilon^{-2} + \log k)k^2 \log n T_{\text{MCF}})$	$O(k^* nm \log\left(\frac{n^2}{m}\right) \log(nU)$ $+((\epsilon^{-2} + \log k^*)k^{*2} \log n(T_{\text{MCF}} + m)))$

Proof: Combine Theorem 2.4.21, Corollary 2.4.16, Corollary 2.4.18, Lemma 2.4.22, Theorem 2.4.20 and the fact that a maximum flow in an n-node m-edge graph can be computed in $O(nm \log(n^2/m))$ time [23]. Note that only in the non-simple case does the time for initialization appear in the final bound. ■

Given an instance of a concurrent flow problem, the running time for Algorithm SCALINGCONCURRENT is never greater than that of Algorithm CONCURRENT. For the rest of this chapter, we will quote the running times for algorithm SCALINGCONCURRENT.

2.4.2 Dealing with Exponentiation

In the previous section, we assumed a non-standard model of computation in which exponentiation takes $O(1)$ time. In this section, we show how to implement an iteration of procedure DECONGEST in the standard RAM model of computation, achieving the same time bounds. Our approach is to show that even if we require that all variables be represented using $O(\log(nU))$ bits, we can still, in each iteration, achieve the same decrease in Φ, up to constant factors.

More specifically, we first show that a flow that satisfies a relaxed set of minimum-cost flow constraints suffices to achieve a decrease in Φ of $\Omega(\frac{\epsilon^2}{k}\Phi)$. We then show that a flow satisfying a second set of relaxed constraints can be modified in $O(m)$ time to satisfy the first set of relaxed constraints while having the additional property that the resulting flow can be represented in $O(\log(nU))$ bits per commodity/edge pair. We then give an approximate length function that uses $O(\log(nU))$ bits per edge which can be used in a minimum-cost flow algorithm to produce a flow that satisfies the second set of relaxed constraints.

Each iteration of procedure DECONGEST, as described in Figure 2.7, iteratively computes f_i^*, which is a flow that satisfies the demands of commodity i subject to capacity constraints $\lambda u(vw)$ on each edge vw, and minimizes $C_i^* = \sum_{vw\in E} \ell(vw)\,|f_i^*(vw)|$. Instead, we compute an approximation $\bar{f}_i^*$ to f_i^*. The flow $\bar{f}_i^*$ can have cost somewhat more than the cost of f_i^*, and it may satisfy slightly relaxed capacity constraints. The key to showing that this flow can be used in the algorithm instead of f_i^* is to prove a relaxed version of Lemma 2.4.8.

Lemma 2.4.24 *Let C_i denote the cost of the current flow of commodity i with respect to the current length function, and let $\bar{f}_i^*$ be a flow that satisfies demands of commodity i and the constraints*

$$\left\{ \begin{array}{rcl} \bar{f}_i^*(vw) & \leq & 2\lambda u(vw)\ \forall vw \in E \\ \sum_{vw\in E} \ell(vw)\,|\bar{f}_i^*(vw)| & \leq & C_i^* + \frac{1}{2}(\epsilon C_i + \epsilon\frac{\lambda\Phi}{k}). \end{array} \right. \tag{2.18}$$

Then, if we use $\bar{f}_i^*$ instead of f_i^* in DECONGEST with $\frac{\epsilon}{32\alpha\lambda} \leq \sigma \leq \frac{\epsilon}{16\alpha\lambda}$, we can bound the decrease of the potential function by $\Omega(\frac{\epsilon^2}{k}\Phi)$.

Proof: The difference between this proof and that of Lemma 2.4.8 is as follows. Here we can conclude that $|\delta(vw)| \leq 3\sigma\lambda u(vw)$, and $|\eta| \leq 3\alpha\sigma\lambda \leq 3\epsilon/16$. We use that $|\eta| \leq 3\epsilon/16 \leq 1/4$

implies $e^{x+\eta} \leq e^x + \eta e^x + \frac{3\epsilon}{16}|\eta| e^x$. We modify equation (2.13) appropriately, and conclude that $\Phi - \Phi' \geq \frac{\alpha\sigma\epsilon\lambda\Phi}{4k} = \Omega(\frac{\epsilon^2}{k})$. ■

In fact, we do not find such a flow directly. Instead, we compute a flow that satisfies the somewhat tighter constraints,

$$\begin{cases} \tilde{f}_i^*(vw) & \leq & \frac{3}{2}\lambda u(vw) \;\; \forall vw \in E \\ \sum_{vw\in E} \ell(vw)\left|\tilde{f}_i^*(vw)\right| & \leq & C_i^* + \frac{1}{4}(\epsilon C_i + \epsilon\frac{\lambda\Phi}{k}). \end{cases} \tag{2.19}$$

We then modify this flow slightly so that it satisfies conditions (2.18) and the new flow can be represented by $O(\log(nU))$ bits per edge.

Lemma 2.4.25 Let $\tilde{f}_i^*$ be a flow that satisfies conditions (2.19). Then, in $O(m)$ time, we can convert it into a flow $\bar{f}_i$ that satisfies (2.18) and such that $(1-\sigma)f_i(vw) + \sigma\bar{f}_i^*(vw)$ can be represented in $O(\log(nU))$ bits per edge.

Proof: Given the flow $\tilde{f}_i^*$, we first compute the flow $(1-\sigma)f_i + \sigma\tilde{f}_i^*$ where σ is chosen as in Lemma 2.4.24. We then round the flow on edge vw to an integer multiple of $\nu = \epsilon^2/(128m^2k\alpha)$. The maximum possible flow value is λU. Dividing the range of flow values by ν and substituting for α, we see that the number of possible flow values is

$$\frac{\lambda U}{\epsilon^2/(128m^2k\alpha)} = \frac{256\lambda U m^2 k(1+\epsilon)\ln(m\epsilon^{-1})}{\lambda\epsilon^3}.$$

The number of bits needed is just the logarithm of the number of possible values. Recall that ϵ is inverse polynomial in n, k is polynomial in n and $m = O(n^2)$, thus a flow value can be represented in $O(\log(nU))$ bits. Observe that if we just rounded the flow on each edge vw to the nearest integer multiple of ν, we would have no guarantee that the flow conservation constraints of equation (2.4) or (2.1)– (2.3) are still satisfied. Thus, we must round more carefully. Let T be a spanning tree in the graph. We round the flow on all the non-tree edges to the nearest multiple of ν. This rounded flow does not necessarily satisfy the conservation constraints, so we use the tree edges to correct for the violations we may have introduced. It is easy to see that by computing the flow values on the edges of T in postorder we can carry out this step in $O(m)$ time. Observe that the amount of flow we had to add to any non-tree edge is at most ν

and the amount that we had to add to any tree edge is at most $m\nu$, since the flow on a tree edge may have to correct for the violation across the cut defined by deleting that edge in the tree.

The resulting rounded flow implicitly defines a $\bar{f}_i^*$ as it can be written as $(1-\sigma)f_i + \sigma\bar{f}_i^*$ for an appropriately chosen $\bar{f}_i^*$. The flow $\bar{f}_i^*$ on edge vw is $\tilde{f}_i^*(vw)$ plus σ^{-1} times the rounding error on the edge. We now show that it satisfies the conditions (2.18). The rounding error on any edge is at most $m\nu$, and therefore for each edge, $\bar{f}_i^*(vw) \le \tilde{f}_i^*(vw) + \sigma^{-1}m\nu$. Plugging in the bounds on $\tilde{f}_i^*(vw)$ from (2.19) and the values of σ and ν, we get an upper bound of $\frac{3}{2}\lambda u(vw) + \frac{1}{2}\lambda$. Since $u(vw)$ is integral, we conclude that $\bar{f}_i^*(vw) \le 2\lambda u(vw)$. We bound the cost of $\bar{f}_i^*$ as follows.

$$\begin{aligned}
\textstyle\sum_{vw\in E}\ell(vw)\left|\bar{f}_i^*(vw)\right| &\le \textstyle\sum_{vw\in E}(|\tilde{f}_i^*(vw)| + \sigma^{-1}m\nu)\ell(vw) \\
&\le \left(\textstyle\sum_{vw\in E}\ell(vw)\left|\tilde{f}_i^*(vw)\right|\right) + m(\sigma^{-1}m\nu e^{\alpha\lambda}) \\
&\le C_i^* + \tfrac{1}{4}\left(\epsilon C_i + \tfrac{\epsilon\lambda\Phi}{k}\right) + \tfrac{\epsilon\lambda e^{\alpha\lambda}}{4k} && \text{(by (2.19) and the definitions of } \sigma \text{ and } \nu) \\
&\le C_i^* + \tfrac{1}{2}\left(\epsilon C_i + \tfrac{\epsilon\lambda\Phi}{k}\right) && \text{(using } \Phi \ge e^{\alpha\lambda}).
\end{aligned}$$

Therefore we have satisfied the conditions of the theorem. ■

Combining the previous two theorems we get the following corollary:

Corollary 2.4.26 Let $\tilde{f}_i^*$ be a flow satisfying equations (2.19). Let $\bar{f}_i^*$ be the flow obtained from $\tilde{f}_i^*$ via the procedure described in Lemma 2.4.25 Then if we use $\bar{f}_i^*$ instead of f_i^* in DECONGEST with $\frac{\epsilon}{32\alpha\lambda} \le \sigma \le \frac{\epsilon}{16\alpha\lambda}$, we can bound the decrease of the potential function during one iteration of DECONGEST by $\Omega(\frac{\epsilon^2}{k}\Phi)$, while maintaining flows represented by $O(\log(nU))$ bits per edge. ■

Now we show how to compute a flow that satisfies (2.19). We could do so by finding a minimum-cost flow with respect to the exact length function ℓ. Unfortunately, this length function is exponential in the size of the input, and computing it exactly might take too long. Instead, we will describe how to compute an approximate length function $\tilde{\ell}$, such that the flow that has minimum cost with respect to $\tilde{\ell}$ has cost at most $C_i^* + \epsilon\lambda\Phi/(8k)$ with respect to ℓ. By Corollary 2.4.26, such a flow can be used in order to implement the rerouting step in our algorithm.

The new length function $\tilde{\ell}$ is integral, consists of $O(\log(nU))$ bits per edge, it is approximately related to ℓ by the scalar multiplier $\gamma = \epsilon e^{\alpha\lambda}/(16Umk)$, and it satisfies $\gamma\tilde{\ell}(vw) \leq \ell(vw)$ on each edge vw. We shall show that it takes $O(\log n)$ time to compute $\tilde{\ell}(vw)$ for each edge vw. In the following we will use $\tilde{C}_i$ and $\tilde{C}_i^*$ to denote the current cost and the minimum cost of commodity i with respect to length $\tilde{\ell}$, respectively.

For each edge, first we compute $e^{\alpha(f(vw)/u(vw)-\lambda)}$ approximately so that it has at most $\zeta = \epsilon/(16km)$ additive error, then we multiply the result by $\zeta^{-1}U$, divide by $u(vw)$, take the integer part, and set $\tilde{\ell}(vw)$ to be this value. Using the Taylor series we can compute one bit of e^x in $O(1)$ time. Since $e^{\alpha(f(vw)/u(vw)-\lambda)}$ is at most 1 on each edge, it is sufficient to compute $O(\log(1/\zeta))$ bits to achieve the desired approximation. Computing the approximate length function takes $O(\log(1/\zeta)) = O(\log n)$ time for each edge, and $O(m \log n)$ time in total.

Because of the approximation and the integer rounding, a flow $\tilde{f}_i^*$, which has minimum cost with respect to $\tilde{\ell}$, is not necessarily the minimum-cost flow with respect to ℓ. We will show, however, that a flow that is minimum-cost with respect to $\tilde{\ell}$ satisfies conditions (2.19).

Lemma 2.4.27 Let $\tilde{f}_i^*$ be a flow that is minimum cost with respect to the costs $\tilde{\ell}$ defined above. Then $\tilde{f}_i^*$ has cost (with respect to ℓ) at most $\epsilon\lambda\Phi/(8k)$ more than the actual minimum-cost flow with respect to ℓ.

Proof: Recall that $\gamma = e^{\alpha\lambda}\zeta/U$, and $\zeta = \epsilon/(16mk)$. We bound the difference between ℓ and $\gamma\tilde{\ell}$, a scaled up version of the approximate length function. In computing $\gamma\tilde{\ell}$, we introduce errors in two places. First, when computing $e^{\alpha(\lambda(vw)-\lambda)}$ to a precision of ζ, we introduce an error of ζ. This error gets scaled up by $\zeta^{-1}U/u(vw)$ when we scale up and gets increased by 1 when we round $\tilde{\ell}$ down to an integer. Finally, if we scale $\tilde{\ell}$ back to be compatible with ℓ, the whole error gets scaled by γ. Thus,

$$\ell(vw) - \gamma\tilde{\ell}(vw) \leq \gamma\left(\zeta\left(\frac{\zeta^{-1}U}{u(vw)}\right) + 1\right) = \gamma\left(\frac{U}{u(vw)} + 1\right). \tag{2.20}$$

We defined $\tilde{\ell}$ so that $\gamma\tilde{\ell}(vw) \leq \ell(vw)$ on each edge, and hence we have that

$$\gamma\tilde{C}_i^* \leq C_i^*. \tag{2.21}$$

Using these two equations and the fact that $\Phi \geq e^{\alpha\lambda}$ we get that:

$$\begin{array}{rcll} \sum_{vw\in E} \ell(vw)\left|\tilde{f}_i^*(vw)\right| - C_i^* & \leq & \sum_{vw\in E} \ell(vw)\left|\tilde{f}_i^*(vw)\right| - \gamma\tilde{C}_i^* & \text{(by (2.21))} \\ & \leq & \sum_{vw} |\tilde{f}_i^*(vw)| \left(\ell(vw) - \gamma\tilde{\ell}(vw)\right) & \\ & \leq & \sum_{vw} |\tilde{f}_i^*(vw)| \gamma \left(\frac{U}{u(vw)} + 1\right) & \text{(by (2.20))} \\ & \leq & 2\sum_{vw} \lambda u(vw) \frac{e^{\alpha\lambda}\zeta}{U} \left(\frac{2U}{u(vw)}\right) & \\ & \leq & \frac{2m\lambda\Phi\epsilon}{16km} & \\ & = & \frac{\epsilon\lambda\Phi}{8k}. & \end{array} \tag{2.22}$$

■

Notice that this flow actually satisfies slightly stronger conditions than (2.19). We shall use this stronger condition in Subsection 2.4.3.

In the randomized implementation, where we used the cost of the current flow C_i for the selection of a bad commodity i, we shall now use the rounded cost $\tilde{C}_i$ instead. One can show that the rounding error is small relative to $\sum_i \tilde{C}_i^*$, and therefore using $\tilde{C}_i$ does not significantly decrease the probability that a bad commodity is selected.

To summarize, we have just described how to implement DECONGEST in the RAM model of computation. We first compute an approximation $\tilde{\ell}$ to the length function ℓ. Then we compute the approximate cost of each commodity and choose, either randomly or deterministically, a commodity to reroute. Next, we compute an approximate minimum-cost flow for that commodity with respect to the costs $\tilde{\ell}$. This process gives us an approximate minimum-cost flow that satisfies equations (2.19). We then update the flows for commodity i. Finally, we modify the updated flow as described in Lemma 2.4.25, represent it in $O(\log(nU))$ bits per edge, and start the next iteration. As the above discussion shows, the time to perform the whole computation is $O(m \log n)$ plus the time to compute a minimum-cost flow.

We can now update Theorem 2.4.23 to remove the assumption that exponentiation takes $O(1)$ time.

Theorem 2.4.28 Let $T_{\mathrm{MCF}} = T_{\mathrm{MCF}}(\mathcal{M})$ be the time to compute a minimum cost flow for instance $\mathcal{M}$. Then the following table gives the times to find an ϵ-optimal solution to the concurrent flow problem in the RAM model:

	simple instance	non-simple instance
Randomized	$O((\epsilon^{-3} + \log k)k \log n(T_{\mathrm{MCF}} + m \log k))$	$O(k^* nm \log\left(\frac{n^2}{m}\right) \log(nU)$ $+(\epsilon^{-3} + \log k^*)k^* \log n(T_{\mathrm{MCF}} + m))$
Deterministic	$O((\epsilon^{-2} + \log k)k^2 \log n T_{\mathrm{MCF}})$	$O(k^* nm \log\left(\frac{n^2}{m}\right) \log(nU)$ $+((\epsilon^{-2} + \log k^*)k^{*2} * \log n(T_{\mathrm{MCF}} + m)))$

2.4.3 Implementing One Iteration

In this subsection we consider the problem of choosing the appropriate minimum-cost flow routine to find a minimum-cost flow subject to the costs $\tilde{\ell}(vw)$. In some cases we only compute an approximate minimum-cost flow subject to cost $\tilde{\ell}$ by further rounding the costs before the minimum-cost flow computation. In all cases, however, we find a flow that satisfies (2.19).

Different situations require different choices. For general concurrent flow problems, the best choice seems to be either the algorithm of Goldberg and Tarjan [23] or that of Ahuja, Goldberg, Orlin and Tarjan [2]. For concurrent flow with uniform capacity, we use Gabow and Tarjan's [19] algorithm for the assignment problem. When both the demands and capacities are uniform, we use the algorithm that iteratively computes shortest paths in the residual graph with nonnegative costs discovered independently by Ford and Fulkerson [17] and Yakovleva [71].

First, we consider the general concurrent flow problem.

Lemma 2.4.29 For a commodity i, a minimum-cost flow with respect to $\tilde{\ell}$ can be found in $O(nm \log(n^2/m) \log(nU))$ time.

Proof: The Goldberg-Tarjan minimum-cost flow algorithm runs in $O(nm \log(n^2/m) \log(nC))$ time, where C is the maximum edge cost, assuming that the costs are integral. Recall that we are only using $O(\log nU)$ bits to represent the integral edge costs, and hence $\log(nC) = O(\log nU)$.
■

The above bound can be improved if the capacities are small relative to n^2/m. In this case we round the demands and solve this rounded problem using the double scaling algorithm of

Ahuja, Goldberg, Orlin, and Tarjan [2]. We then satisfy the remaining flow on arbitrary paths. This flow still satisfies (2.19) and the rounding allows us to use a faster algorithm. We will prove the following lemma:

Lemma 2.4.30 For a commodity i, a flow satisfying (2.19) can be found in $O(nm \log(nU) \log\log(nU))$ time.

Proof: Assume without loss of generality that ϵ^{-1} is an integer and define $\mu = \lambda\epsilon/(16nk)$. We round the demands for commodity i to integer multiples of μ such that the absolute value of each demand does not increase, the rounded demands still sum to zero, and the total decrease in the absolute values of the demands is at most $2n\mu$. (Recall that each node may have a positive or a negative demand.) To achieve these constraints, we round all but one of the demands to the next smallest multiple of μ if the demand is positive and to the next largest multiple of μ if the demand is negative. The last demand can then be rounded by at most $n\mu$ to ensure that the rounded demands still sum to 0.

Since the absolute value of the demand for commodity i has not increased at any node, there must exist a flow satisfying these demands with cost at most $\tilde{C}_i^*$, subject to costs $\tilde{\ell}$.

Both the demands and the capacities are integral multiples of μ. If we divide both the demands and the capacities by μ, we get a problem where the maximum capacity of an edge is $\lambda U/\mu = 16Unk\epsilon^{-1}$. We can then use the double scaling algorithm of Ahuja, Goldberg, Orlin and Tarjan [2] for solving the minimum-cost problem with rounded demands. This algorithm takes $O(nm \log(nC) \log\log(nU'))$ time on a graph with maximum capacity U'. Plugging in the value of C from Lemma 2.4.29 and $U' = 16Unk\epsilon^{-1}$ yields the time bound. By Lemma 2.4.27, this procedure gives a flow that satisfies the capacity constraints $\lambda u(vw)$ and has cost at most $\epsilon\lambda\Phi/(8k)$ more than the minimum cost but does not satisfy the demands. We then satisfy the remaining demands by arbitrary paths from nodes with excess to nodes with deficit. The last step increases the flow on an edge by no more than $2n\mu = \epsilon\lambda/(8k) \le \epsilon\lambda u(vw)/(8k)$, and adds a total of no more than $2n\mu \sum_{vw \in E} \ell(vw) \le \lambda\epsilon\Phi/(8k)$ to the cost of the flow subject to costs ℓ.

Combining the minimum-cost flow with the flows on the additional paths, we get a flow that satisfies (2.19) and proves the lemma. ■

In the case of the simple concurrent flow problem we can make the time required for solving the minimum-cost flow problem independent of U.

Lemma 2.4.31 For the simple concurrent flow problem, a flow of a commodity i satisfying (2.19) can be found in the minimum of $O(nm \log n \log(n^2/m))$ and $O(nm \log n \log\log n)$ time.

Proof: We reduce d_i by a factor of $(1-\epsilon/8)$. We then find a flow f'_i that satisfies the reduced demand $d'_i = (1-\epsilon/8)d_i$ and whose cost with respect to $\tilde{\ell}$ is no more than $\epsilon\lambda \sum_{vw\in E} \tilde{\ell}(vw)u(vw)/(16k)$ above the minimum cost. Then, we multiply the flow on each edge by $(1-\epsilon/8)^{-1}$. This process gives a flow that satisfies demands, obeys the slightly increased capacity constraints $(1-\epsilon/8)^{-1}\lambda \cdot u(vw)$, and has cost (subject to ℓ) at most $\epsilon C_i/4 + \epsilon\lambda\Phi/(4k)$ above C_i^*, where Φ is the current potential function value. By Lemma 2.4.25, we can use this flow and still get the same asymptotic improvement in the potential function.

Define $\mu' = \epsilon d_i/(8m)$, and round the capacities $\lambda u(vw)$ used for the minimum-cost flow problem, down to multiples of μ'. One can show that the minimum-cost flow with respect to $\tilde{\ell}$ that satisfies the decreased demand d'_i and rounded capacity, is no more than $\tilde{C}_i^*$.

The demand and capacities in this rounded problem are integer multiples of μ'. Therefore, there exists a minimum-cost flow where the flow on the edges is multiple of μ'. This flow does not use edges whose cost is more than $\tilde{C}_i/\mu'$, and hence these edges can be deleted for the minimum-cost flow computation.

For getting the approximate minimum-cost flow we can work with a further rounded length function. We take $\bar{\ell}(vw)$ to be the integer part of $d_i\tilde{\ell}(vw)/(\lambda U)$. Since after the capacity rounding we consider only edges with $\lambda u(vw) \geq \mu'$, we have

$$\begin{aligned}
\bar{\ell}(vw) &\leq \frac{16\epsilon^{-1}kmU}{\mu'/\lambda} \cdot \frac{d_i}{\lambda U} \\
&= \frac{16\epsilon^{-1}kmd_i}{\mu'} \\
&= O(\epsilon^{-2}km^2).
\end{aligned}$$

Therefore the Goldberg-Tarjan minimum-cost flow algorithm runs in $O(nm\log(n^2/m)\log n)$ time on this problem.

Now we show that the resulting flow, after multiplication by $(1-\epsilon/8)^{-1}$, satisfies (2.19). The minimum-cost flow has a single source and a single sink and non-negative costs. Therefore, no edge carries more than d_i' units of flow. Let $\bar{f}_i^*$ be a minimum-cost flow with respect to $\bar{\ell}$. By an argument similar to the proof of Lemma 2.4.27, the cost of this flow with respect to ℓ is at most $md_i \cdot \lambda U/d_i \cdot \epsilon e^{\alpha\lambda}/(16kmU) \leq \epsilon\lambda\Phi/(16k)$ larger than the cost of $\tilde{f}_i'$ with respect to ℓ, where $\tilde{f}_i'$ is the minimum-cost flow with respect to $\tilde{\ell}$ that satisfies the reduced demand d_i'. Now Lemma 2.4.27 implies that (2.19) is satisfied.

For all but very dense graphs, the double scaling algorithm of Ahuja, Goldberg, Orlin and Tarjan [2] gives a better bound. As we observed no edge carries more than d_i' units of flow in the optimal flow of commodity i. Thus, we can also limit capacities to be no more than d_i'. That is, we can set $u'(vw) = \min\left\{\left\lfloor\frac{\lambda u(vw)}{\mu'}\right\rfloor\mu', d_i'\right\}$. With this modification, the largest capacity is at most $d_i' = O(m\epsilon^{-1}\mu')$. The demand and the capacities are multiples of μ'. Dividing through by the scale factor μ' yields a problem with integral capacities using $O(\log n)$ bits. ■

Combining Theorem 2.4.28 and Lemmas 2.4.2, 2.4.29, 2.4.30 and 2.4.31 we obtain the following theorem that summarizes the results for the general case:

Theorem 2.4.32 For $\epsilon > 0$, algorithm SCALINGCONCURRENT finds an ϵ-optimal solution for the simple concurrent flow problem in the following time bounds:

	simple instance	non-simple instance
Randomized	$O\left((\epsilon^{-3}+\log k)knm\log^2 n\log\left(\frac{n^2}{m}\right)\right)$	$O\left((\epsilon^{-3}+\log k^*)k^*nm\log n\log(nU)\log\left(\frac{n^2}{m}\right)\right)$
	$O\left((\epsilon^{-3}+\log k)knm\log^2 n\log\log n\right)$	$O\left((\epsilon^{-3}+\log k^*)k^*nm\log n\log(nU)\log\log(nU)\right)$
Deterministic	$O\left((\epsilon^{-2}+\log k)k^2nm\log^2 n\log\left(\frac{n^2}{m}\right)\right)$	$O\left((\epsilon^{-2}+\log k^*)k^{*2}nm\log n\log(nU)\log\left(\frac{n^2}{m}\right)\right)$
	$O\left((\epsilon^{-2}+\log k)k^2nm\log^2 n\log\log n\right)$	$O\left((\epsilon^{-2}+\log k^*)k^{*2}nm\log n\log(nU)\log\log(nU)\right)$

2.5 The Unit Capacity Case

An important special case of the concurrent flow problem occurs when the edge capacities are all 1. One way to solve this problem is to use the algorithm for the general case, but with a minimum-cost flow algorithm more suited to graphs with unit capacities. We will discuss this approach in Section 2.5.1. Sections 2.5.2 through 2.5.5 develop and use a framework for solving uniform capacity concurrent flow problems via the solution of a series of *shortest path problems*. As much of the work needed to develop such algorithms is identical to that done in Section 2.4, we omit some of the details. The algorithms are sufficiently different, however, to merit a fairly involved presentation. For the unit capacity case, we only deal with the simple concurrent flow problem. It is possible to modify the results of this section to give algorithms for the non-simple concurrent flow problem, put we do not pursue that here.

2.5.1 Using the Results for the General Case

One approach to solving the unit capacity case is to use the algorithm for the general case. For efficiency, we modify the minimum-cost flow algorithm that we use. This subsection discusses this approach.

If the capacities in the concurrent flow problem are uniform, then the capacities in each minimum-cost flow problem are all equal to λ. In this case, there are more efficient minimum-cost flow algorithms than the ones mentioned in the previous section.

Lemma 2.5.1 For the simple concurrent flow problem with uniform capacities, a flow for a commodity i satisfying (2.19) can be found in $O(m^{3/2} \log n)$ time.

Proof: Since the concurrent flow problem is simple and has unit capacity, the auxiliary minimum-cost flow problem has one source, one sink and edge capacities all equal to λ. The minimum-cost flow problem is guaranteed to have a feasible solution. Therefore, it must be the case that the demand $d_i \leq m\lambda$, the total capacity in the networks. Let

$$\begin{aligned} d'_i &= \left\lfloor \frac{d_i}{\lambda} \right\rfloor \\ &\leq \left\lfloor \frac{m\lambda}{\lambda} \right\rfloor \\ &\leq m, \end{aligned}$$

and

$$u'(vw) = \left\lfloor \frac{u(vw)}{\lambda} \right\rfloor = 1 \ \forall vw \in E.$$

We solve this rounded problem and then route the remaining flow on a single path. By arguments similar to those in Lemma 2.4.30, this procedure yields a flow that satisfies (2.19) . To solve the minimum-cost flow problem with capacities u' and demands d'_i, we can use an algorithm of Gabow and Tarjan [19]. Gabow and Tarjan show how to modify their scaling algorithm for the assignment problem to find a solution to a single-source single-sink minimum-cost flow problem with m edges, all edge capacities equal to 1, and demand at most d'_i, in $O((m + d'_i)^{3/2}\log(nd'_i))$ time. Plugging in the bound $d'_i \leq m$, we obtain the time bound claimed in the statement of the lemma. ■

When both the capacities and demands are uniform and k is relatively large, we can obtain better performance by using the Ford and Fulkerson [17] minimum-cost flow algorithm, which iteratively computes shortest paths in the graph of residual edges with nonnegative costs. Since each capacity is an integer multiple of λ and the lengths are non-negative, a minimum-cost flow of demand d can be computed by $\lceil d/\lambda \rceil$ shortest path computations in networks with nonnegative edge lengths.

The demands are also uniformly equal to 1, thus $\lambda \geq k/m$, since there are at least k units of flow divided between m edges. Therefore, in this case, the number of shortest path computations required for finding a minimum-cost flow is at most $O(m/k + 1)$. Each shortest path can be computed in $O(m + n\log n)$ time [18]. Thus we have shown:

Lemma 2.5.2 For the simple concurrent flow problem with uniform capacities and demands, a flow for a commodity i satisfying (2.19) can be found in $O\left(\frac{m}{k}(m + n\log n)\right)$ time.

By incorporating Lemma 2.5.2 or Lemma 2.5.1 into Theorem 2.4.32, one can derive running time bounds for algorithm ScalingConcurrentfor the special cases when the capacities are uniform and when both the capacities and demands are uniform. Observe that when both the capacities and demands are uniform, the time to find a minimum-cost flow, $O\left(\frac{m}{k}(m + n\log n)\right)$, may be less than $O(m\log k)$, the time to select a commodity. Because of this case alone, the bounds in Theorem 2.4.20 contain the extra $O(m\log k)$ term.

Theorem 2.5.3 For $\epsilon > 0$, an ϵ-optimal solution for the simple concurrent flow problem

- with uniform capacities can be found in expected $O((\epsilon^{-3} + \log k)km^{3/2}\log^2 n)$ time and in $O((\epsilon^{-2} + \log k)k^2m^{3/2}\log^2 n)$ time deterministically,
- with uniform capacities and demands can be found in expected $O((\epsilon^{-3} + \log k)m\log n(m + n\log n + k\log k))$ time and in $O((\epsilon^{-2} + \log k)km\log n(m + n\log n + k\log k))$ deterministically.

2.5.2 Solving Unit Capacity Concurrent Flow Problems

In this section, we rederive algorithms for the special case of the concurrent flow problem with unit capacities. These are based on treating flow as a collection of paths, rather than as a set of commodities and lead to improved running times for some cases. In this section, we describe the procedure REDUCE that is the core of our approximation algorithms, and prove bounds on its running time. Given a multicommodity flow f, procedure REDUCE modifies f until either f becomes ϵ-optimal or λ is reduced below a given target value. The approximation algorithms presented in the next two sections repeatedly call procedure REDUCE to decrease λ by a factor of 2, until an ϵ-optimal solution is found.

As before, for simplicity of presentation, we shall assume for now that the value of the length function $\ell(vw) = e^{\alpha\lambda(vw)}/u(vw) = e^{\alpha f(vw)}$ of edge vw can be computed in one step from $f(vw)$ and represented in a single computer word. In Section 2.5.4 we will remove this assumption and show that it is sufficient to compute an approximate length function, and show that an approximate length function can be computed quickly.

Procedure REDUCE (see Figure 2.12) is the analog of procedure DECONGEST. It takes as input a multicommodity flow f, a target value τ, an error parameter ϵ, and a flow quantum σ_i for each commodity i. We require that each flow path comprising f_i carries flow that is an integer multiple of σ_i. The procedure repeatedly reroutes σ_i units of flow from an ϵ-long path of commodity i to a shortest path for commodity i. We will need a technical *granularity condition* that σ_i is small enough for each i to guarantee that approximate optimality is achievable through such reroutings. In particular, we assume that when REDUCE is called, we have

$$\sigma_i \leq \epsilon^2 \frac{\tau}{3\log(m\epsilon^{-1})} \quad i = 1, \ldots, k. \tag{2.23}$$

Upon termination, the procedure outputs an improved multicommodity flow f such that either λ is less than the target value τ or f is ϵ-optimal. In this section, we assume that $\epsilon \leq \frac{1}{12}$, the bound on ϵ needed to prove Theorem 2.3.5.

In the remainder of this section, we analyze the procedure REDUCE shown in Figure 2.12. First, we show that if the granularity condition is satisfied, the number of iterations in REDUCE is small. Second, we give an even smaller bound on the number of iterations for the case in

REDUCE($f, \tau, \epsilon, \sigma_i$ for $i = 1, \ldots, k$)
$\alpha \leftarrow (1+\epsilon)\tau^{-1}\epsilon^{-1}\log(m\epsilon^{-1})$.
while $\lambda \geq \tau$ **and** f and ℓ are not ϵ-optimal
  For each edge vw, $\ell(vw) \leftarrow e^{\alpha f(vw)}$.
  Call FINDPATH(f, ℓ, ϵ) to find an ϵ-long flow path P and a short path Q with the same endpoints as P.
  Reroute σ_i units of flow from P to Q.
return f.

Figure 2.12: Procedure Reduce.

which the flow f is $O(\epsilon)$-optimal when REDUCE is called. Finally, we will give two algorithms, UNIT and SCALINGUNIT, that bound the number of iterations needed to solve unit capacity concurrent flow problems. The former solves the case when $\epsilon = O(1)$, while the latter solves the case when $\epsilon = o(1)$. As in the general case, for ease of presentation, we assume a model of computation in which exponentiation takes $O(1)$ time and the word size is unbounded. In later sections we will remove this assumption, and discuss the implementation of an iteration of the algorithm.

Bounding the number of iterations of REDUCE

At the beginning of REDUCE, α is set equal to $(1+\epsilon)\tau^{-1}\epsilon^{-1}\log(m\epsilon^{-1})$, which is essentially the same as in the general case. While $\lambda \geq \tau$, the value of α is sufficiently large, so by Lemma 2.4.3 relaxed optimality condition $R1$ is satisfied. If we are lucky and relaxed optimality condition $R2'$ is also satisfied, then it follows that f and ℓ are ϵ-optimal. Now, we show that if $R2'$ is not satisfied, then we can make significant progress. As before, we use $\Phi = \sum_{vw \in E} \ell(vw)u(vw)$ as a measure of progress. In the unit-capacity case, $\Phi = \sum_{vw \in E} \ell(vw) = \sum_{vw \in E} e^{\alpha f(vw)}$.

Lemma 2.5.4 Suppose σ_i and τ satisfy the granularity condition. Then rerouting σ_i units of flow from an ϵ-long path of commodity i to the shortest path with the same endpoints decreases Φ by $\Omega(\frac{\sigma_i \log m}{\min\{D, kd_i\}}\Phi)$.

Proof: Let P be an ϵ-long path from s_i to t_i, and let Q be a shortest (s_i, t_i)-path. Let $A = P - Q$, and $B = Q - P$. The only edges whose length changes due to the rerouting are those in $A \cup B$.

The decrease in Φ is $\ell(A)+\ell(B)-e^{-\alpha\sigma_i}\ell(A)-e^{\alpha\sigma_i}\ell(B)$, which can also be written as

$$(1-e^{-\alpha\sigma_i})(\ell(A)-\ell(B))-(1-e^{-\alpha\sigma_i})(e^{\alpha\sigma_i}-1)\ell(B).$$

The granularity condition, the definition of α, and the assumption that $\epsilon \leq 1/12$, imply that $\alpha\sigma_i \leq \frac{1+\epsilon}{3}\epsilon \leq \epsilon/2 \leq 1/2$. For $0 \leq x \leq \frac{1}{2}$, we have $e^x \geq 1+x$, $e^x \leq 1+\frac{4}{3}x$, and $e^{-x} \leq 1-\frac{2}{3}x$. Thus the decrease is at least

$$\frac{2}{3}\alpha\sigma_i\left(\ell(A)-\ell(B)\right)-(\alpha\sigma_i)\left(\frac{4}{3}\alpha\sigma_i\right)\ell(B). \tag{2.24}$$

Now, observe that $\ell(A)-\ell(B)$ is the same as $\ell(P)-\ell(Q)$, and $\ell(Q)=dist_\ell(s_i,t_i)$. Also, $\ell(B)\leq\ell(P)$. Plugging these bounds into (2.24) yields a lower bound of

$$\frac{2}{3}\alpha\sigma_i\left(\ell(P)-dist_\ell(s_i,t_i)\right)-\left(\frac{4}{3}\alpha^2\sigma_i^2\right)\ell(P). \tag{2.25}$$

But P is ϵ-long, so the quantity in (2.25) must be at least

$$\begin{aligned}
& \frac{2}{3}\alpha\sigma_i\left(\epsilon\ell(P)+\epsilon\frac{\lambda}{\min\{D,kd_i\}}\sum_{vw\in E}\ell(vw)u(vw)\right)-\frac{4}{3}\alpha^2\sigma_i^2\ell(P)\\
= \; & \frac{2}{3}\alpha\sigma_i\epsilon\ell(P)-\frac{4}{3}\alpha^2\sigma_i^2\ell(P)+\frac{2}{3}\alpha\epsilon\lambda\frac{\sigma_i}{\min\{D,kd_i\}}\Phi.
\end{aligned}$$

We have seen that $\frac{\epsilon}{2}\geq\alpha\sigma_i$, which implies that $\frac{2}{3}\epsilon\geq\frac{4}{3}\alpha\sigma_i$, and therefore the first term dominates the second term. Thus, the third term gives a lower bound on the decrease in Φ. Substituting the value of α and using the fact that during the execution of REDUCE we have $\tau\leq\lambda$, we obtain the claim of the lemma. ■

The following theorem bounds the number of iterations in REDUCE.

Theorem 2.5.5 If, for each commodity i, the values τ and σ_i satisfy the granularity condition and we have $\lambda=O(\tau)$ initially then the procedure REDUCE terminates after $O(\epsilon^{-1}\max_i\frac{\min\{D,kd_i\}}{\sigma_i})$ iterations. If in addition, the input flow f is $O(\epsilon)$-optimal, then the procedure REDUCE terminates after $O(\max_i\frac{\min\{D,kd_i\}}{\sigma_i})$ iterations.

UNIT($\mathcal{I}, \epsilon$)
For each commodity i: $\sigma_i \leftarrow d_i$, create a simple path from s_i to t_i and route d_i flow on it.
$\tau \leftarrow \lambda/2$.
while f is not 12ϵ-optimal
 for every i
  until σ_i and τ satisfy the granularity condition
   $\sigma_i \leftarrow \sigma_i/2$.
 $f \leftarrow$ REDUCE(f, τ, ϵ, d).
 $\tau \leftarrow \tau/2$.
return f.

Figure 2.13: Procedure Concurrent.

Proof: The same as the proof of Lemma 2.4.6 with $\max_i \frac{\min\{D, kd_i\}}{\sigma_i} \log m$ substituted for $k\epsilon^{-2}$.
■

In most cases, one iteration of the loop in REDUCE is dominated by the time spent in FINDPATH, so we concentrate on bounding the number of calls to FINDPATH. Although there are some cases in which the call to FINDPATH is not the dominant part of an iteration of REDUCE, we assume for now that it is. In Section 2.5.5, we will see a case when the time spent on calls to FINDPATH is not the dominant step, and will deal with it separately there.

Solving Unit Capacity Concurrent Flow

In this section, we give approximation algorithms for the concurrent flow problem with uniform capacities. We describe two algorithms: UNIT and SCALINGUNIT. Algorithm UNIT is simpler and is best if ϵ is constant. SCALINGUNIT gradually scales ϵ to the right value and is faster for $\epsilon = o(1)$.

The presentation of this section is slightly different than in the general case. Instead of expressing the running times in terms of the number of minimum-cost flow computations, we express it in terms of the number of calls to the procedure FINDPATH. FINDPATH has both a deterministic and a randomized implementation, so we will put off the difference between the two until we discuss FINDPATH in more detail.

Algorithm UNIT (see Figure 2.13) consists of a sequence of calls to procedure REDUCE described in the previous section. The initial flow is constructed by routing each commodity

i on a single flow path from s_i to t_i. Initially, we set $\sigma_i = d_i$. Before each call to REDUCE we divide the flow quantum σ_i by 2 for each commodity where this is needed to satisfy the granularity condition (2.23). Each call to REDUCE modifies the multicommodity flow f so that either λ decreases by a factor of 2 or f becomes ϵ-optimal. In the latter case algorithm CONCURRENT terminates and returns the flow. As we will see, $O(\log m)$ calls to REDUCE will suffice to achieve ϵ-optimality.

Theorem 2.5.6 Let T_F be the time used by by procedure FINDPATH. The algorithm UNIT finds an ϵ-optimal multicommodity flow in $O((\epsilon^{-1} k \log n + \epsilon^{-3} m \log n) T_F)$ time.

Proof: Immediately after the initialization, we have $\lambda \leq D$. To bound the number of phases we need a lower bound on the minimum value of λ. Observe that for every multicommodity flow f, the total amount of flow in the network is D. Every unit of flow contributes to the total flow on at least one of the edges, and hence $\sum_{vw \in E} f(vw) \geq D$. Therefore,

$$\lambda \geq D/m, \tag{2.26}$$

which implies that the number of iterations of the main loop of UNIT is

$$O\left(\log\left(\frac{D/m}{D}\right)\right) = O(\log m).$$

By Theorem 2.5.5 procedure REDUCE executes $O(\epsilon^{-1}\frac{\min\{D,kd_i\}}{\sigma_i})$ iterations during a single call to REDUCE. Throughout the algorithm, for each i, σ_i is either equal to d_i, or is $\Theta(\epsilon^2\tau/\log(m\epsilon^{-1}))$. In the first case,

$$\begin{aligned} \frac{\min\{D, kd_i\}}{\sigma_i} &= \frac{\min\{D, kd_i\}}{d_i} \\ &= \min\left\{\frac{D}{d_i}, k\right\} \\ &\leq k. \end{aligned}$$

In the second case

$$\frac{\min\{D, kd_i\}}{\sigma_i} = \min\{D, kd_i\}\,\epsilon^{-2}\tau^{-1}\log(m\epsilon^{-1})$$

```
ScalingUnit(I, ε)
ε' ← 1/12.
Call Unit(I, ε'), and let f be the resulting flow.
  τ ← τ/2.

while ε' > ε,
  ε' ← ε'/2,
  for every i,
   until σ_i and τ satisfy the granularity condition,
     σ_i ← σ_i/2.
  f ← Reduce(f, τ, ε', σ).
return f.
```

Figure 2.14: Procedure ScalingConcurrent.

$$\leq \quad \epsilon^{-2}\frac{D}{\tau}\log(m\epsilon^{-1}).$$

Thus, the total number of iterations of the loop of REDUCE is $O(\epsilon^{-1}(k + \epsilon^{-2}\frac{D}{\tau}\log(m\epsilon^{-1}))$. The value τ is halved at every iteration, and therefore the total number of calls required for all iterations is $O(\epsilon^{-1}k\log n)$ plus twice the number required for the last iteration of UNIT. It follows from (2.26) that $\tau = \Omega(\frac{D}{m})$, and the total number of iterations of the loop of REDUCE is at most $O(\epsilon^{-1}k\log n + \epsilon^{-3}m\log n)$. ■

If $\epsilon = o(1)$, we use the algorithm SCALINGCONCURRENT, shown in Figure 2.14. The algorithm starts with a large ϵ and then gradually scales ϵ down to the required value. More precisely, algorithm SCALINGUNIT starts by applying algorithm UNIT with $\epsilon = \frac{1}{12}$. SCALINGUNIT then repeatedly divides ϵ by a factor of 2 and calls REDUCE. After the initial call to UNIT, f is 1-optimal, and λ is no more than twice the minimum possible value. Therefore, λ cannot be decreased below $\tau/2$, and each subsequent call to REDUCE returns an ϵ-optimal multicommodity flow (with the current value of ϵ). As in UNIT each call to REDUCE uses the largest flow quantum σ permitted by the granularity condition (2.23).

Theorem 2.5.7 Let T_F be the time taken by procedure FINDPATH, then algorithm SCALINGUNIT finds an ϵ-optimal multicommodity flow in $O((k + m\epsilon^{-2})\log nT_F)$ time.

Proof: As is stated in Theorem 2.5.6, the call to procedure UNIT uses $O((k + m)\log n)$ calls to FINDPATH and returns a 1-optimal multicommodity flow f. Hence, λ is no more than twice the minimum. Therefore all subsequent calls to REDUCE returns an ϵ-optimal multicommodity flow f.

The time required by one iteration is dominated by the call to REDUCE. The input flow f of REDUCE is $24\epsilon'$-optimal, so, by Theorem 2.5.5, REDUCE executes $O(\max_i \frac{\min\{D,kd_i\}}{\sigma_i})$ iterations of FINDPATH. We have seen in the proof of Theorem 2.5.6 that $O(\max_i \frac{\min\{D,kd_i\}}{\sigma_i})$ is at most $O(k+\epsilon'^{-2}m\log n)$. The value of ϵ' is reduced by a factor of 2 in every iteration. So the total number of calls to FINDPATH is

$$\sum_{\epsilon'=\frac{1}{12},\frac{1}{24},\ldots,\epsilon} O(k+\epsilon'^{-2}m\log n) = \sum_{\epsilon'=\frac{1}{12},\frac{1}{24},\ldots,\epsilon} O(k) + \sum_{\epsilon'=\frac{1}{12},\frac{1}{24},\ldots,\epsilon} O(\epsilon^{-2}m\log n).$$

There are at most $\log(\epsilon^{-1}) = O(\log n)$ iterations, so the first term sums to $O(k\log n)$. The second sum is a geometric series and is no more than twice the last term, so it is bounded by $O(\epsilon^{-2}m\log n)$. Combining this bound with the bound on procedure UNIT from Theorem 2.5.6 yields the claim. ■

2.5.3 Implementing One Iteration

We have shown that REDUCE terminates after a small number of iterations. It remains to show that each iteration can be carried out quickly. REDUCE consists of three steps: computing lengths, executing FINDPATH and rerouting flow. Assuming that exponentiation can be performed in $O(1)$ time, computing lengths takes $O(m)$ time. Thus, we focus our attention on the other two steps.

We first consider the time taken by procedure FINDPATH. We shall give three implementations of this procedure. First, we will give a simple deterministic implementation that runs in $O(k^*(m+n\log n)+n\sum_i \frac{d_i}{\sigma_i})$ time, then a more sophisticated implementation that runs in time $O(k^*n\log n + m(\log n + \min\{k, k^*(\log d_{\max}+1)\}))$, and finally a randomized implementation that runs in expected $O(\epsilon^{-1}(m+n\log n))$ time. All of these algorithms use the shortest-paths algorithm of Fredman and Tarjan [18] that runs in $O(m+n\log n)$ time.

To deterministically find a bad flow path, we first compute, for each source node s_i, the length of the shortest path from s_i to every other node v, which takes $O(k^*(m+n\log n))$ time. In the simplest implementation, we then compute the length of every flow path in $\mathcal{P}$ and compare its length to the length of the shortest path to decide if the path is ϵ-long. There can be at most $\sum_i \frac{d_i}{\sigma_i}$ flow paths, each consisting of up to n edges, and hence computing these lengths takes $O\left(n \sum_i \frac{d_i}{\sigma_i}\right)$ time.

To decrease the time required for FINDPATH, we must find an ϵ-long path, if one exists, without computing the length of every path. The following lemma explains how to achieve this:

Lemma 2.5.8 The total time required for deterministically implementing an iteration of REDUCE (assuming that exponentiation is a single step) is $O(k^* n \log n + m(\log n + \min\{k, k^*(\log d_{\max} + 1)\}))$.

If there is an ϵ-long flow path for commodity i then the longest flow path for commodity i must be ϵ-long. Thus, instead of looking for an ϵ-long path in $\mathcal{P}_i$ for some commodity i, it suffices to find an ϵ-long path in the directed graph obtained by taking all flow paths in $\mathcal{P}_i$, and treating the paths as directed away from s_i. In order to see if there is an ϵ-long path, we need to compute the length of the longest path from s_i to t_i in this directed graph. To facilitate this computation, we shall maintain that the directed flow graph is acyclic.

Let G_i denote the flow graph of commodity i. If G_i is acyclic, an $O(m)$ time dynamic programming computation suffices to compute the longest paths from s_i to every other node. Suppose that during an iteration we reroute flow from an ϵ-long path from s_i to t_i, in the flow graph G_i. We must first update the flow graph G_i to reflect this change. Second, the update might introduce directed cycles in G_i, so we must eliminate such cycles of flow. We use an algorithm due to Sleator and Tarjan [63] to implement this process. Sleator and Tarjan gave a simple $O(nm)$ algorithm and a more sophisticated $O(m \log n)$ algorithm for the problem of converting an arbitrary flow into an acyclic flow.

Eliminating cycles only decreases the flows on edges, so it cannot increase Φ. Thus the bound from Theorem 2.5.5 on the number of iterations in REDUCE still holds.

We compute the total time required for each iteration of REDUCE as follows. In order to implement FINDPATH, we must compute a shortest path from s_i to t_i in G and the longest path from s_i to t_i in G_i for every commodity i, so the time required is $O(k^*(m + n \log n) + km)$. Furthermore, after each rerouting, we must update the appropriate flow graph and eliminate cycles. Elimination of cycles takes $O(m \log n)$ time. Combining these bounds gives an $O(k^* n \log n + m(k + \log n))$ bound on the running time of FINDPATH.

In fact, further improvement is possible if we consider the flow graphs of all commodities with the same source and same flow quantum σ_i together. Let $G_{v,\sigma}$ be the directed graph obtained by taking the union of all flow paths $P \in \mathcal{P}_i$ for a commodity i with $s_i = v$ and $\sigma_i = \sigma$, treating each path as directed away from v. If $G_{v,\sigma}$ is acyclic, an $O(m)$ time dynamic programming computation suffices to compute the longest paths from v to every other node in $G_{v,\sigma}$.

During our concurrent flow algorithm all commodities with the same demand have same flow quantum. To limit the different flow graphs that we have to consider we want to limit the number of different demands. By decomposing demand d_i into at most $\lfloor \log d_i \rfloor + 1$ demands with source s_i and sink t_i we can assume that each demand is a power of 2. This way the number of different flow graphs that we have to maintain is at most $k^*(\log d_{\max} + 1)$. ■

Next, we give a randomized implementation of FINDPATH that is much faster when ϵ is not too small; this implementation is similar to the randomized implementation of the general case. If f and ℓ are not ϵ-optimal, then relaxed optimality condition $R2'$ is not satisfied, and thus ϵ-long paths contribute at least an ϵ-fraction of the total sum $\sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$. Therefore, by randomly choosing a flow path P with probability proportional to its contribution to the above sum, we have at least an ϵ chance of selecting an ϵ-long path. Furthermore, we will show that we can select a candidate ϵ-long path according to the right probability in $O(m)$ time. Then we can compute a shortest path with the same endpoints in $O(m + n \log n)$ time, which enables us to determine whether or not P was an ϵ-long path. Thus we can implement FINDPATH in $O(\epsilon^{-1}(m + n \log n))$ expected time.

The contribution of a flow path P to the above sum is just the length of P times the flow on P, so we must choose P with probability proportional to this value. In order to avoid examining all such flow paths explicitly, we use a two-step procedure, as described in the following lemma.

Lemma 2.5.9 If we choose an edge vw with probability proportional to $\ell(vw)f(vw)$, and then we select a flow path among paths through this edge vw with probability proportional to the value of the flow carried on the path, then the probability that we have selected a given flow path P is proportional to its contribution to the sum $\sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$.

Proof: Let $B = \sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$. Select an edge vw with probability $f(vw)\ell(vw)/B$.

Once an edge vw is selected, choose a path $P \in \mathcal{P}_i$ through edge vw with probability $\frac{f_i(P)}{f(vw)}$. Consider a commodity i and a path $P \in \mathcal{P}_i$.

$$\begin{aligned}
Pr(P \text{ chosen}) &= \sum_{vw \in P} Pr(vw \text{ chosen}) \cdot \frac{f_i(P)}{f(vw)} \\
&= \sum_{vw \in P} \frac{f(vw)\ell(vw)}{B} \cdot \frac{f_i(P)}{f(vw)} \\
&= \sum_{vw \in P} \frac{\ell(wv) f_i(P)}{B} \\
&= \frac{f_i(P)\ell(P)}{B}.
\end{aligned}$$

■

Choosing an edge with probability proportional to $\ell(vw)f(vw)$ can easily be done in $O(m)$ time. In order to then choose with the right probability a flow path going through that edge, we need a data structure to organize these flow paths. For each edge we maintain a balanced binary tree with one leaf for each flow path through the edge, labeled with the flow value of that flow path. Each internal node of the binary tree is labeled with the total flow value of its descendent leaves. The number of paths is polynomial in n and ϵ^{-1}, and therefore using this data structure, we can randomly choose a flow path through a given edge in $O(\log n)$ time.

In order to maintain this data structure, each time we change the flow on an edge, we must update the binary tree for that edge, at a cost of $O(\log n)$ time. In one iteration of Reduce the flow only changes on $O(n)$ edges, and therefore the time to do these updates is $O(n \log n)$ per call to FindPath, which is dominated by the time to compute single-source shortest paths.

We have shown that if relaxed optimality condition $R2'$ is not satisfied, then, with probability at least ϵ we can find an ϵ-long path in $O(m + n \log n)$ time. FindPath continues to pick paths until either an ϵ-long path is found or 12ϵ trials are made. Observe that given that f and ℓ are not yet ϵ-optimal (which implies that condition $R2'$ is not yet satisfied), the probability of failing to find an ϵ-long path in 1ϵ trials is bounded by $1/e$. Thus, in this case, Reduce can terminate, claiming that f and ℓ are ϵ-optimal with probability at least $1 - 1/e$. Computing lengths and updating flows can each be done in $O(n \log n)$ time, and thus we get the following bound:

Lemma 2.5.10 One iteration of REDUCE can be implemented to run in expected time $(\epsilon^{-1}(m+n\log n))$ time (assuming that exponentiation is a single step). ■

The randomized algorithm, as it stands, is *Monte Carlo*; there is a non-zero probability that REDUCE erroneously claims to terminate with an ϵ-optimal f. To make the algorithm *Las Vegas* (never wrong, sometimes slow), we introduce a deterministic check. If FINDPATH fails to find an ϵ-long path, REDUCE computes the sum $\sum_i dist_\ell(s_i,t_i)d_i$ to the required precision and compares it with $\lambda\sum_{vw\in E}\ell(vw)u(vw)$ to determine whether f and ℓ are really ϵ-optimal. If not, the loop resumes. The time required to compute the sum is $O(k^*(m+n\log n))$, because at most k^* single-source shortest path computations are required. The probability that the check must be done t times in a single call to REDUCE is at most $(e^{-1})^{t-1}$, so the total expected contribution to the running time of REDUCE is at most $O(k^*(m+n\log n))$.

Recall that the number of iterations of REDUCE is greater than $\max_i \frac{\min\{D,kd_i\}}{\sigma_i}$, which in turn is at least k. Since in each iteration we carry out at least one shortest path computation, the additional time spent on checking does not asymptotically increase our bound on the running time for REDUCE.

2.5.4 Dealing with Exponentiation

To remove the assumption that exponentiation can be performed in $O(1)$ time, we shall do two things. First we shall show that it is sufficient to work with edge lengths $\hat{\ell}(vw)$ that are approximations to the actual lengths $\ell(vw)=e^{\alpha f(vw)}$. We then show that computing these approximate edge lengths does not change the asymptotic running times of our algorithms.

In fact, we show that for large values of ϵ (e.g., when ϵ is a constant), the time required for FINDPATH can be reduced by using approximate lengths. To do so requires two changes: using a different implementation of Dijkstra's algorithm and using a more sophisticated data structure for storing the flow paths going through an edge.

The first step is to note that in the proof of Lemma 2.5.4, we never used the fact that we reroute flow onto a *shortest* path. We only need that we reroute flow onto a *sufficiently short* path. More precisely, it is easy to convert the proof of Lemma 2.5.4 into a proof for the following claim. The conversion is similar to that used to prove Lemma 2.4.24 via Lemma 2.4.8.

Lemma 2.5.11 Suppose that σ_i and τ satisfy the granularity condition and let P be a flow path of commodity i. Let Q be a path connecting the endpoints of P such that the length of Q is no more than $\epsilon\ell(P)/2 + \epsilon\frac{\lambda}{D}\Phi/2$ greater than the length of the shortest path connecting the same endpoints. Then rerouting σ_i units of flow from path P to Q decreases Φ by $\Omega(\frac{\sigma_i}{\min\{D,kd_i\}}\Phi\log m)$). ■

We now show that in order to compute the lengths of paths up to the precision given in this lemma, we only need to compute the lengths of edges up to a reasonably small amount of precision.

By Lemma 2.5.11, the length of a path can have a rounding error of $\epsilon\frac{\lambda\sum_{vw\in E}\ell(vw)u(vw)}{2D}$. Each path has at most n edges, so it suffices to ensure that each edge has a rounding error of

$$\frac{1}{n}(\epsilon\frac{\lambda}{D}\sum_{vw\in E}\ell(vw)u(vw)/2). \tag{2.27}$$

We shall now bound this quantity. The value λ is the maximum flow on an edge and hence must be at least as large as the average flow on an edge, i.e., $\lambda \geq \sum_{vw} f(vw)/m$. Every unit of flow contributes to the total flow on at least one edge, and hence $\sum_{vw} f(vw) \geq D$. Combining with the previous inequality, we get that

$$\lambda/D \geq 1/m. \tag{2.28}$$

The potential function $\sum_{vw\in E}\ell(vw)u(vw)$ is at least as big as the length of the longest edge, i.e.,

$$\sum_{vw\in E}\ell(vw)u(vw) \geq e^{\alpha\lambda}. \tag{2.29}$$

Plugging (2.28) and (2.29) into (2.27), we see that it suffices to compute the length of an edge with an error of at most $\frac{\epsilon}{nm}e^{\alpha\lambda}$. Each edge has a positive length of at most $e^{\alpha\lambda}$ and can be expressed as $e^{\alpha\lambda}\rho$, where $0 < \rho \leq 1$. Thus we need to compute ρ up to an error of $\frac{\epsilon}{nm}$. To do so, we need to compute $O(\log(\epsilon^{-1}nm))$ bits, which by the assumption that ϵ^{-1} is polynomial in n, is just $O(\log n)$ bits.

By using the Taylor series expansion of e^x, we can compute one bit of the length function in $O(1)$ time. Therefore, to compute the lengths of all edges at each iteration of REDUCE, we need $O(m \log n)$ time. We shall see that in the deterministic implementation of REDUCE each iteration takes $\Omega(m \log n)$ time (the time required for cycle canceling). Therefore the time spent on computing the lengths is dominated by the running time of an iteration.

The approximation above depends on the current value of λ, which may change after each iteration. It was crucial that we recomputed the lengths of every edge in every iteration. The time to do so, $O(m \log n)$, would dominate the running time of the randomized implementation of REDUCE. (Recall that the randomized implementation does not do cycle canceling.) Thus, we need to find an approximation that does not need to be recomputed at every iteration. We choose one that does not depend on the current λ and hence only needs to be updated on the $O(n)$ edges on which the flow actually changes. We proceed to describe such an approximation that depends on τ rather than λ.

Throughout REDUCE all edge length are at most $e^{O(\alpha\tau)}$, and at least one edge has length more than $e^{\alpha\tau}$. Therefore, $\sum_{vw \in E} \ell(vw)u(vw)$ is at least $e^{\alpha\tau}$, and by the same argument as for the deterministic case $O(\log n)$ bits of precision suffice throughout REDUCE. When we first call REDUCE, we must spend $O(m \log n)$ time to compute the edge lengths. For each subsequent iteration, we only need to spend $O(n \log n)$ time updating the $O(n)$ edges whose length has changed. Since each iteration of REDUCE is expected to take $O(\epsilon^{-1}(m + n \log n))$ time to compute shortest paths in FINDPATH, the time for updating edges is dominated by the time required by FINDPATH. While it appears that the time to initially compute all the edge lengths may dominate the time spent in one call to REDUCE, as we have seen, whenever any of our algorithms calls REDUCE, it performs $\Omega(\log n)$ iterations. Each iteration is expected to take at most $\Omega(\epsilon^{-1}m)$ time to compute the shortest paths in FINDPATH. Therefore, the time spent on initializing lengths is dominated by the running time of REDUCE.

In describing the randomized version of FINDPATH in Lemma 2.5.9, we assumed we knew the exact lengths. By using the approximate lengths, however, we do not significantly change a path's apparent contribution to the sum $\sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$. Hence, we do not significantly reduce the probability of selecting a bad path.

Thus we have shown that without any assumptions, REDUCE can be implemented deterministically in the same time as is stated in Lemma 2.5.8. Although for the randomized version, there is additional initialization time, for all the algorithms in this chapter, the initialization time is dominated by the time spent in the iterations of REDUCE.

Theorem 2.5.12 The running times required for the deterministic implementations of procedure REDUCE stated in Lemma 2.5.8 hold without the assumption that exponentiation takes $O(1)$ time. The times required by the randomized implementations increase by an additive term of $O(\epsilon^{-1} m \log n)$ without this assumption.

2.5.5 Further implementation details

In this section we show how one can reduce the time per iteration of REDUCE for the case in which ϵ is a constant. First, we show how using approximate lengths can reduce the time required by FINDPATH; we use an approximate shortest-paths algorithm that runs in $O(m + n\epsilon^{-1})$ time. Then, we give improved implementation details for an iteration of REDUCE to decrease the time required by other parts of REDUCE.

We now describe how, given the lengths and an ϵ-long path P from s to t, we can find, in $O(m + n\epsilon^{-1})$ time, a path Q with the same endpoints such that $\ell(Q) \leq dist_\ell(s,t) + \epsilon\ell(P)/2$. First, we discard all edges with length greater than $\ell(P)$, for they can never be in a path that is shorter than P (if P is a shortest path between s and t then P is not an ϵ-long path). Next, on the remaining graph, we compute shortest paths from s using approximate edge-lengths $\tilde{\ell}(v,w) = \frac{\epsilon\ell(P)}{2n}\left\lceil \ell(vw)\frac{2n}{\epsilon\ell(P)}\right\rceil$, thus giving us $dist_{\tilde{\ell}}(s,t)$, an approximation of $dist_\ell(s,t)$, the length of the actual shortest (s,t)-path. There are at most $n-1$ edges on any shortest path, and for each such edge, the approximate length is at most $\frac{\epsilon\ell(P)}{2n}$ more than the actual length. Thus we know that

$$\begin{aligned} dist_{\tilde{\ell}}(s,t) &\leq dist_\ell(s,t) + n\frac{\epsilon\ell(P)}{2n} \\ &= dist_\ell(s,t) + \frac{\epsilon\ell(P)}{2}. \end{aligned}$$

Further, since each shortest path length is an integer multiple of $\frac{\epsilon\ell(P)}{2n}$ and no more than $\ell(P)$, we can use Dial's implementation of Dijkstra's algorithm [13] to compute $dist_{\tilde{\ell}}(s,t)$ in $O(m+n\epsilon^{-1})$ time.

Implementing FindPath with this approximate shortest path computation directly improves the time required by a deterministic implementation of Reduce. The randomized implementation of FindPath with approximate shortest path computation requires $O(\epsilon^{-1}(m+n\epsilon^{-1}))$ expected time. In order to claim that an iteration of Reduce can be implemented in the same amount of time, we must handle two difficulties: updating edge lengths and updating each edge's table of flow paths when flow is rerouted. Previously, these steps took $O(n \log n)$ time, which was dominated by the time for FindPath. We have reduced the time for FindPath, so the time for these steps now dominates. We now show how to carry out these steps in $O(n)$ time. For the first step, we show that a table can be precomputed so that each edge length can be updated in constant time. For the second step, we sketch a three-level data structure that allows selection of a random flow path through an edge in $O(n)$ time, and allows constant-time addition and deletion of flow paths.

Suppose that before computing the length $e^{\alpha f(vw)}$, we were to round $\alpha f(vw)$ to the nearest multiple of ϵ/c, for some constant c. This rounding introduces an additional multiplicative error of $1 + O(\epsilon/c)$ in the length of each edge and hence an additional multiplicative error of $1 + O(\epsilon/c)$ on each path. By arguments similar to those in the previous subsection, however, this process still gives us a sufficiently precise approximation.

Now we show that by rounding in this way, there are a small enough number of possible values for $\ell(vw)$ that we can just compute them all at the beginning of an iteration of Reduce and then compute the length of an edge by simply looking up the value in a precomputed table. The largest value of $\alpha f(vw)$ we ever encounter is $O(\epsilon^{-1} \log n)$. Since we are only concerned with multiples of ϵ/c, there are a total of only $O(\epsilon^{-2} \log n)$ values, we can ever encounter. At the beginning of each iteration, we compute each of these numbers to a precision of $O(\log n)$ bits in $O(\epsilon^{-2} \log^2 n)$ time. Once we have computed all these numbers, we compute the length of an edge by computing $\alpha f(vw)$, truncating to a multiple of ϵ/c, and then looking up the value of $\ell(vw)$ in the table. This process takes $O(1)$ time. Thus for constant ϵ, we are spending $O(\log^2 n + m) = O(m)$ time per iteration.

Now, we address the problem of maintaining, for each edge, the flow paths going through that edge. Henceforth, we will describe the data structure associated with a single edge. First, suppose that all flow paths carry the same amount of flow, i.e., σ_i is the same for each. In this case, we keep pointers to the flow paths in an array. We maintain that the array is at most 1/4 empty. It is then possible to randomly select a flow path in constant expected time as follows; one randomly chooses an index and checks whether the corresponding array entry has a pointer to a flow path. If so, select that flow path. If not, try another index.

One can delete flow paths from the array in constant time. If one maintains a list of empty entries, one can also insert in constant time. If the array gets too full, copy its contents into a new array of twice the size. The time required for copying can be amortized over the time required for the insertions that filled the array. If the array gets too empty, copy its contents into a new array of half the size. The time required for copying can be amortized over the time required for the deletions that emptied the array. (See, for example, [12], for a detailed description of this data structure.)

Now, we consider the more general case in which the flow values of flow paths may vary. In this case, we use a three-level data structure. In the top level, the paths are organized according to their starting nodes. In the second level, the paths with a common starting node are organized according to their ending nodes. The paths with the same starting and ending nodes may be assumed to belong to the same commodity, and hence all carry the same amount of flow. Thus, these paths can be organized using the array as described above.

The first level consists of a list. Each list item specifies a starting node, the total flow of all flow paths with that starting node, and a pointer to the second-level data structure organizing the flow paths with the given starting node. Each second-level data structure also consists of a list. Each item in the second level list specifies an ending node, the total flow of all flow paths with that ending node and the given starting node, and a pointer to the third-level data structure, the array containing flow paths with the given starting and ending nodes.

Now we analyze the time required to maintain this data structure. Adding and deleting a flow path takes $O(1)$ time. Choosing a random flow path with the right probability can be accomplished in $O(n)$ time. First we randomly choose a value between 0 and the total flow through the edge. Then we scan the first-level list to select an appropriate item based on the value. Next we scan the second-level list pointed to by that item, and select an item in the second-level list. Each of these two steps takes $O(n)$ time. Finally, we select an entry in the third-level array. In the third level array, all flows have the same σ_i, thus an entry can be chosen

$O(1)$ expected time by the scheme described for the special case when all flow paths had the same value.

So we have shown that for constant ϵ, each of the three steps in procedure REDUCE can be implemented in $O(m)$ expected time, thus yielding the following lemma.

Lemma 2.5.13 If $\epsilon = O(1)$, then procedure REDUCE can be implemented in expected $O(m)$ time.

Combining the results in this section, we get theorems that summarize our results. First, we consider the case for a constant ϵ. In this case we use algorithm UNIT. Combining Theorem 2.5.6, Lemmas 2.5.8 and 2.5.10, Theorem 2.5.12 and Lemma 2.5.13 we get the following theorem:

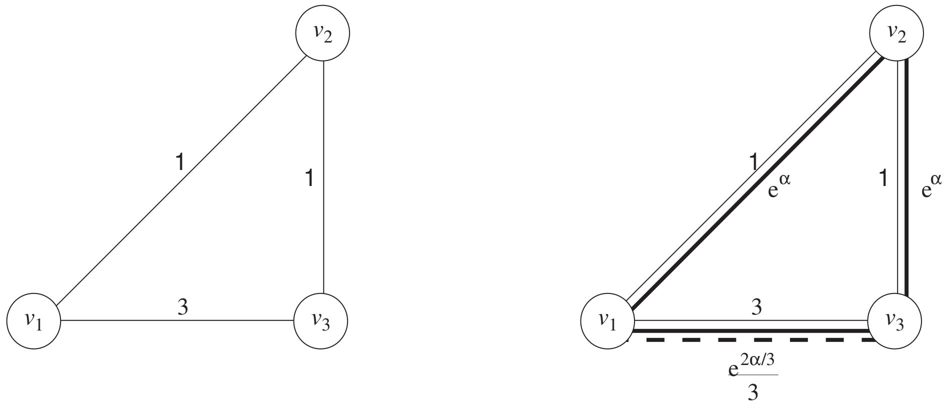
Theorem 2.5.14 For any constant $\epsilon > 0$, algorithm UNIT finds an ϵ-optimal solution for the unit-capacity concurrent flow problem in $O(m(k+m)\log n)$ expected time and in $O(m(k+m)(\log n + \min\{k, k^* \log d_{\max}\})\log n)$ deterministically. ■

Using Theorem 2.5.7, Lemmas 2.5.8 and 2.5.10,and Theorem 2.5.12 we obtain the following bounds on the running time of algorithm SCALINGUNIT:

Theorem 2.5.15 For $0 < \epsilon \leq 1/12$, algorithm SCALINGUNIT finds an ϵ-optimal solution to the unit-capacity concurrent flow problem in expected time $O((k\epsilon^{-1} + m\epsilon^{-3}\log n)(m + n\log n))$ and deterministically in time $O((k + \epsilon^{-2}m)\log n)(k^* n \log n + m(\log n + \min\{k, k^*(\log d_{\max} + 1)\})))$.
■

2.6 Open Problems

The big open questions are whether the dependence on ϵ can be reduced, and whether an algorithm similar to CONCURRENT can be used to get an exact algorithm for the multicommodity flow problem. The dependence on k, n and m is certainly acceptable, since the best algorithms for performing k maximum flows take, up to logarithmic factors, $O(knm)$ time. Yet, the dependence on ϵ is not as satisfying. If we want to get solutions that have accuracy $\epsilon = o(n^{-.1})$, algorithm CONCURRENT takes more time than the exact linear programming algorithms.



commodity	s	t	demand	symbol
1	v_1	v_2	2	———
2	v_1	v_3	1	– – –

Figure 2.15: A bad example for an of our algorithm

The other problem is that of obtaining an exact algorithm. As is done in interior-point linear programming algorithms, we could run our algorithm until it is possible to do a rounding step. In order to achieve the necessary accuracy, however, ϵ must be much too small to be of any practical interest. Are there any approaches that would allow us to round earlier? One drawback of our algorithm is that, given an optimal solution, it is unable to recognize it. Consider the graph in Figure 2.15. This problem has two commodities. Commodity 1 wants to send 2 units of flow between v_1 and v_2 and commodity 2 wants to send 1 unit of flow between v_1 and v_3. Suppose that as an initial solution we choose the routing that appears in the graph on the right. The flow of commodity 1 is represented by heavy lines and the flow for commodity 2 is represented by a dashed line. This solution has $\lambda = 1$, which is optimal. The values of the edge lengths appear along the edges. Consider the two paths for commodity 1. The top path $v_1 v_2$ has cost e^{α}, while the bottom path $v_1 v_3 v_2$ has cost $e^{\alpha} + e^{2\alpha/3}/3$. Thus, the algorithm routes flow off the bottom path and onto the top path. The flow of the top path is now more than the capacity, which causes λ to increase. Thus we no longer have an optimal solution.

The reason that this unfortunate phenomenon occurs is that we require that the dual variables be a predetermined function of the primal variables. There exist problems for which such a solution seems hard to achieve. An algorithm that computes an exact solution would probably need to relax this condition. Perhaps a more fruitful approach is to use our algorithm to get close to the optimal solution and then switch to some other algorithm.

Chapter 3

Applications of Multicommodity Flow[1]

3.1 Introduction

The techniques for solving multicommodity flow problems have a host of applications and extensions. In Section 3.2 we shall discuss a special case where it is possible to find integral flows, and which has applications to a VLSI routing problem. In Section 3.3, we show how the solution to a concurrent flow problem can be used to help find sparse cuts in graphs.

3.2 An Integer Theorem for Multicommodity Flows

In this section we discuss situations in which the techniques presented in Chapter 2 can be used to obtain good *integral* solutions. None of the four algorithms, CONCURRENT, SCALING-CONCURRENT, UNIT, or SCALINGUNIT find flows that are integral. For many applications, it is desirable to have *integral* solutions. In Section 3.2.1, we will discuss an application to a VLSI routing problem, in which the flows represent numbers of wires, and thus we want the flow to take on integral values. In general, we can not obtain results about integral solutions that are as strong as the results we have obtained about non-integral solutions. For some cases

[1]This chapter contains joint work with Tom Leighton, Fillia Makedon, Serge Plotkin, Éva Tardos and Spyros Tragoudas [42] and joint work with Philip Klein, Serge Plotkin and Éva Tardos [35].

of interest, however, we can obtain rather strong results. The ability to do so is interesting because the integer multicommodity flow problem seems to be harder than the non-integral one: the integer problem is NP-hard, while, as we have discussed, the non-integral problem can be solved exactly in polynomial time, using linear programming.

For the remainder of this section, we focus on the case when we have a unit-capacity unit-demand problem in which $\lambda^* = \Omega(\log n)$. Using similar arguments one can obtain integer solutions within some guaranteed factor of optimal for some problems in which the demands and capacities are not uniform, but we will not pursue that direction here. In Chapter 6 we will show how to find integer solutions to a related problem.

The unit-capacity unit-demand problem in which $\lambda^* = \Omega(\log n)$ is one of those studied by Raghavan [50] and Raghavan and Thompson [51, 52]. The problem is to find a solution to a unit-capacity unit-demand concurrent flow problem in which all flows must be integral. They introduced a technique for solving this type of problem known as *randomized rounding.* We describe the idea as it relates to the unit-capacity unit-demand concurrent flow problem. First, ignore the integrality constraints and solve the resulting problem, known as the linear-programming relaxation, using any linear-programming algorithm. Since this problem has unit demands, the flow for each commodity is a collection of paths, each of which carries some amount of flow between 0 and 1. Then interpret the flow on path p as the probability that all flow for that commodity is on path p. Using Chernoff-type bounds, Raghavan and Thompson show that, if $\lambda^* = \Omega(\log n)$, then the resulting flow is such that

$$\lambda \leq \lambda^* + O(\sqrt{\lambda^* \log n}). \tag{3.1}$$

Raghavan later showed how to make this algorithm deterministic. The deterministic algorithm still requires the solution of the linear-programming relaxation and a derandomized version of the randomized rounding.

Our algorithms can be used to replace the linear-programming step, thereby giving a faster algorithm for the problem. However, we can use our techniques to achieve a more interesting and efficient result. By slightly modifying algorithm ScalingUnit, we obtain an algorithm that finds an integral flow satisfying (3.1) *directly.* In other words, we do not need to solve the linear-program relaxation and perform the rounding. Our modification to algorithm ScalingUnit provides an alternative proof of Raghavan and Thompson's result[51]. It also yields a significantly faster algorithm, since the time-consuming step is now approximately solving a concurrent flow problem, rather than exactly solving a linear-program.

To find such an integral flow, we simply run algorithm ScalingUnit with the modification that we never allow any of the σ_i to become non-integral. With this modification, $d_i = 1,\ i = 1, \ldots k$, and therefore $\sigma_i = 1,\ i = 1, \ldots, k$. Thus, we never allow the step $\sigma_i \leftarrow \sigma_i/2$ in Unit or ScalingUnit to be executed. Consequently, if the granularity condition (2.23) becomes false, we terminate the algorithm. We now show that this algorithm gives an integral solution that is sufficiently close to optimal.

Theorem 3.2.1 Assume we run algorithm ScalingUnit on a concurrent flow problem in which all $d_i = 1,\ i = 1, \ldots, k$, but maintain that the flows are integral by terminating whenever we would have executed the step $\sigma_i \leftarrow \sigma_i/2$. If $\lambda^* = \Omega(\log n)$, then this algorithm yields an integral solution with $\lambda = \lambda^* + O(\sqrt{\lambda^* \log n})$.

Proof: ScalingUnit begins with a call to Unit. The call to Unit can terminate in two ways, either with a $\frac{1}{12}$-optimal flow, or because granularity condition (2.23) would become false if σ_i were divided by 2. First suppose the call to Unit terminates because the granularity condition becomes false. At this point, we have

$$\frac{1}{2} \leq \epsilon^2 \frac{\tau}{3\log(m\epsilon^{-1})} \leq 1, \tag{3.2}$$

where τ is the target value for λ. In particular, we have $\tau \geq \lambda/2$ and $\epsilon = \frac{1}{12}$, and therefore $\lambda = O(\log n)$. By our assumption $\lambda^* = \Omega(\log n)$, and thus $\lambda \leq \lambda^* + O(\sqrt{\lambda^* \log n})$.

Now assume that the call to Unit terminates with a $\frac{1}{12}$-optimal flow. We proceed with ScalingUnit. It terminates when the granularity condition becomes false, at which point inequality (3.2) implies that $\epsilon^2 = \Theta((\log m)/\tau)$. The flow f is ϵ-optimal and integral. So $\lambda \leq (1+\epsilon)\lambda^* \leq \lambda^* + O(\lambda^* \sqrt{(\log m)/\tau})$. Since $\tau = \lambda/2 \geq \lambda^*/2$, this bound on λ is at most $\lambda^* + O(\sqrt{\lambda^* \log m})$, as required. ■

Observe that Theorem 3.2.1 gives a direct proof of the theorem of Raghavan and Thompson, in the sense that in order to find an integral flow, we never need to resort to a fractional flow as an intermediate step. Not only does it give a direct proof, but it also gives a faster algorithm. On any input, the modified integral algorithm does no more work than the original version of ScalingUnit. Hence, the running times of Theorem 2.5.15 apply. We can derive even faster running times, however, by reanalyzing the algorithm for this special case.

Theorem 3.2.2 If $\lambda^* = \Omega(\log m)$, a flow such that $\lambda \leq \lambda^* + O(\sqrt{\lambda^* \log n})$ can be found by a randomized algorithm in expected time $O(km \log k + k^{3/2}(m + n \log n)/\sqrt{\log n})$, and by a deterministic algorithm in time $O(k \log k(k^* n \log n + mk^* + m \log n))$.

Proof: We have shown that algorithm ScalingUnit finds the required routing if it is terminated as soon as the granularity condition becomes false with $\sigma = 1$. Now we analyze the time required.

We begin with a call to Unit. Recall that Unit repeatedly calls Reduce, and each call reduces λ by a factor of 2. Since each $d_i = 1$ and there are k commodities, throughout the algorithm $1 \leq \lambda \leq k$. Thus, there are $O(\log k)$ calls to Reduce. By Theorem 2.5.5, each call to Reduce consists of $O(\epsilon^{-1} \max_i \frac{\min\{D, kd_i\}}{\sigma_i})$ calls to findpath. In this case, $O(\epsilon^{-1} \max_i \frac{\min\{D, kd_i\}}{\sigma_i}) = O(\epsilon^{-1} \max_i \min\{k, k\}/1) = O(\epsilon^{-1} k)$. Hence, Unit consists of $O(k \log k)$ calls to findpath, when ϵ is constant. The remainder of ScalingUnit consists of a series of calls to reduce, each with ϵ decreasing by a factor of 2. Hence, the time for the series of calls in the randomized implementation is dominated by the time for the last iteration. The last iteration consists of $O(k)$ calls to findpath with $\epsilon = \Theta(\sqrt{\log m/\tau})$. Since the total amount of flow in the network is k, we have $\tau = O(\lambda) = O(k)$, and thus $\epsilon^{-1} = O(\sqrt{k/\log n})$. In the deterministic implementation, we use the fact that there are at most $O(\log \sqrt{k/\log n}) = O(\log k)$ iterations, for a total bound of $O(k \log k)$ calls to Findpath.

To derive the running times, we just need to plug in the time for findpath. Using Lemma 2.5.8, Lemma 2.5.10, Theorem 2.5.12 and Lemma 2.5.13 for the times for findpath yields the theorem. ∎

3.2.1 Applications to VLSI routing

In this section, we discuss the problem of approximately minimizing channel width in VLSI routing. Often, a VLSI design consists of a collection of modules separated by channels. The modules are connected up by wires that are routed through the channels. For purposes of regularity the channels have uniform width. It is desirable to minimize that width in order to minimize the total area of the VLSI circuit. Raghavan and Thompson [51] give an approximation algorithm for minimizing the channel width. They model the problem as a graph problem in which one must route wires between pairs of nodes in a graph G so as to minimize the maximum number of wires routed through an edge. To approximately solve the problem, they first solve a concurrent flow problem where there is a commodity with demand 1 for each path that needs to be routed. An optimal solution f_{opt} fails to be a wire routing only in that it may consist of paths of *fractional* flow. The value of $|f_{\text{opt}}|$ is certainly a lower bound on the minimum channel width. Raghavan and Thompson give a randomized method for converting the fractional flow f_{opt} to an integral flow, increasing the channel width only slightly. The resulting wire routing f achieves channel width at most

$$|f_{\text{opt}}| + O(\sqrt{|f_{\text{opt}}| \log n}) \tag{3.3}$$

which is at most $w_{\min}+O(\sqrt{w_{\min}\log n})$, where $w_{\min}$ is the minimum width. In fact, the constant implicit in this bound is quite small. Later Raghavan [49] showed how this conversion method can be made deterministic.

Using Theorem 3.2.1, we can directly obtain an integral flow satisfying (3.3) and thus solve the channel routing problem. This method is much faster than the original method of Raghavan and Thompson. Our method does have a somewhat larger constant hidden in the big-O of equation (3.3). However, by changing the constant in the granularity condition, we can get a solution with a much smaller constant than the one given here, although not as small as that of Raghavan and Thompson. In order to get a smaller constant factor in the quality of the approximation, the running time of the algorithm increases by a constant factor.

3.3 Sparse Cuts

Another application of our concurrent flow algorithms is finding sparse cuts in graphs. The computational bottleneck of these algorithms is solving a concurrent flow problem and its linear programming dual. First, we summarize the previous sparse cut approximation results. Then we show our concurrent flow algorithm can be used to find an approximately optimal dual solution to the corresponding concurrent flow problems, in addition to finding a near optimal flow. Finally, we shall give even faster running times for the special case of the sparse cut problem where the input graph G has low maximum degree.

3.3.1 Review of Previous Results on Sparse Cuts

We begin by motivating the need for finding sparse cuts. One good method for solving graph problems is to use a *divide-and-conquer* algorithm, which involves dividing the graph into two pieces, solving the two pieces separately, and then patching the results back together. In many graph problems, the number of nodes determines the difficulty of solving a problem, and the number of edges going between the two pieces of the graph determines the cost of patching the problem together. It is therefore desirable to split the graph into two roughly equal pieces, such that the number of edges going between the two sides is small. Thus, to every (bi)partition of the nodes of a graph, we can assign a value that measures how well we have achieved this goal. While there are many ways we can formulate this metric, we describe one that is particularly useful.

Let G be an undirected graph with capacities u on its edges. For a subset of the nodes A, we use $\bar{A}$ to denote the complement of A. The associated *cut* is the set of edges $\Gamma(A)$ with one endpoint in A and the other in $\bar{A}$. Let $u(\Gamma(A))$ denote the sum of the capacities of the edges in the cut. The metric we use is

$$\beta = u(\Gamma(A))/(|A||\bar{A}|).$$

This value, β, is small when the number of edges crossing the cut is small and when the two sides are balanced. Leighton and Rao [43] gave an $O(\log n)$-approximation algorithm for the problem of minimizing the ratio $\beta = u(\Gamma(A))/(|A||\bar{A}|)$ over all cuts.

Given the ability to find such cuts, many problems have been solved by using a divide-and-conquer approach in the manner described in the first paragraph of this subsection. In particular, this approach has yielded the first polylog-times-optimal approximation algorithms for a wide variety of NP-complete graph problems. Leighton and Rao [43] showed how to use these techniques to find approximately balanced separators. Combining the result of Leighton and Rao with the results of Bhatt and Leighton [9], we obtain algorithms to approximate the minimum cut linear arrangement, minimum area layout and $\sqrt{2}$-bifurcators of a graph. They also showed how to approximate the minimum feedback-arc set. Hansen [28] has shown how to extend these results to approximate some graph embedding problems, and Makedon and Tragoudas [46] have extended some of these results to hypergraphs.

Consider the concurrent flow problem on G with one unit of demand between each pair of nodes. The optimum value λ^* must satisfy

$$\lambda^* \cdot u(\Gamma(A)) \geq d(A, \bar{A}) = |A||\bar{A}| \tag{3.4}$$

for each cut $\Gamma(A)$, where $d(A, \bar{A})$ denotes the sum of all demands across the cut. Therefore, the minimum value of $u(\Gamma(A))/(|A||\bar{A}|)$ over all cuts $\Gamma(A)$ gives an upper bound on $1/\lambda^*$. Leighton and Rao show that this minimum is within an $O(\log n)$ factor of the value $1/\lambda^*$. Their algorithm to find approximately sparsest cuts makes use of this connection. More precisely given a nearly optimal length function (dual variables) they show how to find a partition $A \cup B$ that is within a factor of $O(\log n)$ of the minimum value of λ, and hence of the value of the sparsest cut.

The computational bottleneck of the Leighton and Rao algorithm is computing a nearly optimal λ and the corresponding near-optimal linear programming dual solution for the concurrent flow problem on G with one unit of demand between each pair of nodes. The dual solution is a non-negative length function ℓ that maximizes the ratio $\sum_{v,w} dist_\ell(v, w)/(\sum_{vw \in E} u(vw)\ell(vw))$ (see Theorem 2.2.2). Linear programming duality implies that this maximum is equal to λ^*. Leighton and Rao use a linear programming algorithm to find the length function.

A natural extension is the problem where we are given nonnegative node weights $\nu(v)$ for $v \in V$ in addition to the capacities on the edges. For a subset X of V let $\nu(X)$ denote the sum of the weights on the nodes in X. Consider the extension of the sparsest cut problem to minimizing $u(\Gamma(A))/(\nu(A)\nu(\bar{A}))$ over all cuts. The Leighton and Rao algorithm can be extended to give an $O(\log n)$-approximation algorithm for this problem. The corresponding concurrent flow problem has demand between each pair of nodes, where the demand $d(s, t)$ between nodes s and t equals $\nu(s)\nu(t)$. (If the weights are scaled so that the total node-weight is n, then the main change to the Leighton and Rao algorithm is to select the node s for starting a tree with $\nu(s)$ maximum.)

Klein, Agrawal, Ravi, and Rao [33] extended the Leighton and Rao results to the case of simple concurrent flow problems with integral capacities and arbitrary integral demands. For a source-sink pair (s,t), let $d(s,t)$ denote the corresponding demand. The minimum ratio cut problem is to minimize the ratio $u(\Gamma(A))/d(A,\bar{A})$ over all cuts. By inequality (3.4), the minimum value of the ratio $u(\Gamma(A))/d(A,\bar{A})$ is an upper bound on $1/\lambda^*$. Klein, Agrawal, Ravi, and Rao [33] proved that this upper bound is at most a factor of $O(\log nU \log kD)$ more than $1/\lambda^*$ in general, and they gave an $O(\log nU \log kD)$ approximation algorithm for the minimum cut problem, where U is the maximum capacity and D is the maximum demand. Tragoudas [66] has observed that their algorithm can be modified to give an $O(\log n \log kD)$ factor instead.

Using this result, Klein et al. give approximation algorithms for chordalization of a graph, register sufficiency, minimum deletion of clauses in a $2CNF \equiv$ formula, via minimization and the edge-deletion graph bipartization problems. Later Ravi, Agrawal and Klein [53] used these techniques to give approximation algorithms for interval graph completion and a single-processor scheduling algorithm. Similar to the Leighton-Rao algorithm, the computational bottleneck of their algorithm is solving the dual of the concurrent flow problem, i.e., finding a length function ℓ such that the ratio $\sum_{s,t\in V} d(s,t)dist_\ell(s,t)/\sum_{vw\in E} u(vw)\ell(vw)$ is close to maximum.

3.3.2 Speeding up the Unit-Capacity Case

The computational bottleneck of the method of Leighton and Rao is solving a unit-capacity concurrent flow problem in which there is a demand of 1 between each pair of nodes. In their paper, they appealed to the fact that the concurrent flow problem can be formulated as a linear program, and hence can be solved in polynomial time. A much more efficient approach is to use our unit-capacity approximation algorithm. The number of commodities required is $O(n^2)$. Leighton [40] has discovered a technique to reduce the number of commodities required. He shows that if the graph in which there is an edge connecting each source-sink pair is an expander graph, then the resulting flow problem suffices for the purpose of finding an approximately sparsest cut. (We call this graph the *demand graph*.) In an expander we have:

> For any partition of the node set into A and B, where $|A| \leq |B|$, the number of commodities crossing the associated cut is $\theta(|A|)$.

Therefore, for this smaller flow problem $\lambda = \Omega(|A|/u(\Gamma(A)))$. Since $|B| \geq n/2$, it follows that $n\lambda = \Omega(|A||B|)/u(\Gamma(A)))$. The smaller flow problem essentially "simulates" the original all-pairs problem. Moreover, Leighton and Rao's sparsest-cut algorithm can start with the length function for the smaller flow problem in place of that for the all-pairs problem. Thus Leighton's idea allows one to find an approximately sparsest-cut after solving a much smaller concurrent flow problem. If one is willing to tolerate a small probability of error in the approximation, one can use $O(n)$ randomly selected source-sink pairs for the commodities. It is well known how to randomly select node pairs so that, with high probability, the resulting demand graph is an expander.

By Theorem 2.5.15, algorithm UNIT takes expected time $O(m^2 \log^2 m)$ to find an appropriate solution for this smaller problem. We then find a dual solution and run the rest of the algorithm of Leighton-Rao. The dominant step, however, is the solution of the concurrent flow problem.

Theorem 3.3.1 An $O(\log n)$-factor approximation to the sparsest cut in a graph can be found by a randomized algorithm in $O(m^2 \log^2 m)$ time. ■

3.3.3 Speeding up the General Case

Finding Good Dual Solutions

The algorithms for finding sparse cuts in node-weighted edge-weighted graphs were discovered by Klein, Agrawal, Ravi and Rao [33]. Similar to the algorithm of Leighton and Rao for the unit capacity case, they first approximately solved a concurrent flow problem and then used the edge lengths (dual variables) to guide the second phase of their algorithm. The time consuming step of their algorithm is to solve their concurrent flow problem and find the dual variables. They relied on the fact that a concurrent flow problem can be solved via linear programming. In this section, we will show how to find faster solutions using algorithm SCALINGCONCURRENT. Unfortunately, algorithm SCALINGCONCURRENT returns an ϵ-optimal solution for the formulation of the concurrent flow problem in which optimality is measured in terms of minimum-cost flows. In other words, assume for a moment an infinite precision model of computation. Then, by Lemma 2.2.4, the flow f and length function ℓ returned by algorithm ScalingConcurrent on an instance for which the optimal congestion is λ^* are such that:

$$\frac{\sum_{i=1}^{k} C_i^*(\lambda)}{\sum_{vw \in E} \ell(vw)u(vw)} \geq \frac{\lambda^*}{1+\epsilon}. \tag{3.5}$$

The algorithm of Klein et al. needs a length function $\hat{\ell}$, such that for some constant $\epsilon > 0$, this function satisfies

$$R(\hat{\ell}) = \frac{\sum_{s,t \in V} d(s,t)\mathrm{dist}_{\hat{\ell}}(s,t)}{\sum_{vw \in E} \hat{\ell}(vw)u(vw)} \geq \frac{\lambda^*}{1+\epsilon}\ . \tag{3.6}$$

In order to do so, we use the algorithm ScalingConcurrent to find a length function ℓ. We show that with respect to this length function, we do not necessarily satisfy (3.5) but we can show that

$$\frac{\sum_{i=1}^{k} C_i^*(\lambda)}{\sum_{vw \in E} \ell(vw)u(vw)}$$

is "close" to λ^*. We then show how to modify this length function so that it satisfies (3.6).

Throughout this section we refer to the complementary slackness conditions for the minimum-cost flow problem. These were given in Section 2.2, but we restate them here. Given an instance for a minimum-cost flow problem $\mathcal{M}$ and a feasible flow f_i then f_i is optimal if and only if there exists a price function p such that

$$c_p(vw) \geq 0 \quad \forall vw \in E_{f_i} \tag{3.7}$$

where E_{f_i} is the set of edges in the residual graph G_{f_i}.

First, we consider the concurrent flow problem that directly corresponds to the given minimum-ratio cut problem. We combine all commodities that share a source into a single commodity as suggested in Lemma 2.2.1, which decreases the number of commodities to $k^* \leq n$. We shall index the resulting commodities by their sources. Given an error target ϵ, if our concurrent flow algorithm used the exact length function ℓ, it would compute a flow satisfying capacities $\lambda \cdot u(vw)$ such that

$$Q = \frac{\sum_s C_s^*(\lambda)}{\sum_{vw \in E} \ell(vw)u(vw)} \geq \frac{\lambda^*}{1+\epsilon}\ .$$

But we actually compute flows with respect to an approximate length function $\tilde{\ell}$, described in the proof of Lemma 2.4.27. Let $\tilde{Q}$ denote the corresponding ratio with ℓ replaced by $\tilde{\ell}$ and C_i^* replaced by $\tilde{C}_i^*$. First we show that $\tilde{Q}$ is almost as close to λ^* as Q.

Lemma 3.3.2 Let f be the flow and $\tilde{\ell}$ be the length function returned by algorithm ScalingConcurrent. Then $\tilde{Q} \geq \frac{\lambda^*}{1+2\epsilon}$.

Proof: Let $\gamma = \epsilon \cdot e^{\alpha\lambda}/(16mkU)$. Recall that γ is the factor that approximately relates the real lengths to the approximate lengths. By the way the approximate lengths were computed, $\gamma\tilde{\ell}(vw) \leq \ell(vw)$ for each edge $vw \in E$. Also, by arguments similar to those used to derive (2.22) we have

$$C_i^* - \gamma\tilde{C}_i^* \leq \epsilon\lambda\Phi/(8k) \leq \epsilon\lambda^*\Phi/(4k).$$

Using these two facts, we obtain the following bound on $\tilde{Q}$:

$$\begin{aligned}
\tilde{Q} &= \frac{\gamma\sum_s \tilde{C}_i^*(\lambda)}{\sum_{vw\in E}\gamma\tilde{\ell}(vw)u(vw)} \\
&\geq \frac{\gamma\sum_s \tilde{C}_i^*(\lambda)}{\sum_{vw\in E}\ell(vw)u(vw)} \\
&\geq \frac{\sum_s C_i^*(\lambda) - \epsilon\lambda^*\sum_{vw\in E}\ell(vw)u(vw)/4}{\sum_{vw\in E}\ell(vw)u(vw)} \\
&= \frac{\sum_s C_i^*(\lambda)}{\sum_{vw\in E}\ell(vw)u(vw)} - \epsilon\lambda^*/4 \\
&\geq \frac{\lambda^*}{1+\epsilon} - \frac{\epsilon\lambda^*}{4} \\
&\geq \frac{\lambda^*}{1+2\epsilon}.
\end{aligned}$$

The last inequality follows from $\epsilon \leq \frac{1}{9}$.■

Now, we describe how to modify this length function to produce one that satisfies inequality (3.6). Setting $\hat{\ell} = \tilde{\ell}$ does not necessary work, since $\sum_{s,t\in V} d(s,t)\text{dist}_{\hat{\ell}}(s,t)$ might be significantly smaller than $\sum_s \tilde{C}_s^*(\lambda)$. Instead of using $\tilde{\ell}$ directly, we compute a new length function $\hat{\ell}$. Let λ be the congestion of the flow returned by algorithm ScalingConcurrent. For each commodity s, we compute a minimum-cost flow for instance $\mathcal{M}_s = (G, u\cdot\lambda, \tilde{\ell}, \hat{d}_s)$, that is, a flow that is minimum with respect to costs $\tilde{\ell}$ and capacities $\lambda \cdot u(vw)$ for each commodity. We then use the optimal price function $\tilde{p}_s$ (dual variables from the minimum-cost flow) to adjust $\tilde{\ell}$

by adding to it the sum of the absolute values of reduced costs for edges with negative reduced costs.

Let $\tilde{f}_s^*$ be the minimum-cost flow for instance $\mathcal{M}_s = (G, u\cdot\lambda, \tilde{\ell}, \hat{d}_s)$, and let $\tilde{p}_s$ be the optimal price function for this instance. We use $\ell_s(vw)$ to denote the absolute value of the reduced cost of edge vw if it is negative, and zero otherwise, i.e.,

$$\ell_s(vw) = -\min\{0, \tilde{\ell}(vw) + \tilde{p}_s(v) - \tilde{p}_s(w)\}. \tag{3.8}$$

The complementary slackness conditions for the minimum-cost flow problem (3.7) imply that if $\ell_s(vw) > 0$ then $\tilde{f}_s^*(vw) = \lambda u(vw)$. We define the new length function as $\hat{\ell}(vw) = \tilde{\ell}(vw) + \sum_s \ell_s(vw)$. We need the following lemma to estimate the numerator of $R(\hat{\ell})$.

Lemma 3.3.3 Let $\tilde{f}_s^*$ be the minimum-cost flow for instance $\mathcal{M}_s = (G, u\cdot\lambda, \tilde{\ell}, \hat{d}_s)$, and let $\tilde{p}_s$ be the optimal price function for this instance. Then $\tilde{f}_s^*$ is also the minimum-cost flow for instance $\mathcal{M}'_s = (G, u\cdot\lambda, \tilde{\ell} + \ell^s, \hat{d}_s)$. Further, the cost of $\tilde{f}_s^*$ in instance $\mathcal{M}'_s$ is

$$\sum_{vw\in E} (\tilde{\ell}(vw) + \ell^s(vw))\tilde{f}_s^* = \sum_t d(s,t)\mathrm{dist}_{\tilde{\ell}+\ell_s}(s,t).$$

Proof: We prove the optimality of $\tilde{f}_s^*$ by showing that $\tilde{f}_s^*$ and the price function $\tilde{p}_s$ satisfy the complementary slackness conditions (3.7) for instance $\mathcal{M}'_s$. By the definition of ℓ_s we have that the reduced cost of edge vw, $\tilde{\ell}(vw) + \ell_s(vw) + \tilde{p}_s(v) - \tilde{p}_s(w)$, is nonnegative and it is positive if and only if $\tilde{\ell}(vw) + \tilde{p}_s(v) - \tilde{p}_s(w)$ is positive. By applying the complementary slackness conditions to cost $\tilde{\ell}$, flow $\tilde{f}_s^*$ and prices $\tilde{p}_s$, we see that if this value is positive, then $\tilde{f}_s^*$ is zero, and therefore, $\tilde{f}_s^*$ is minimum-cost for instance $\mathcal{M}'_s$.

Now consider the cost of $\tilde{f}_s^*$ subject to the cost function $\tilde{\ell} + \ell_s$. There are no edges with negative reduced cost, and therefore the cost of the flow is at least $\sum_t d(s,t)(\tilde{p}_s(t) - \tilde{p}_s(s))$. All edges that carry flow have zero reduced cost, which implies that the cost of the flow is equal to $\sum_t d(s,t)(\tilde{p}_s(t) - \tilde{p}_s(s))$ and $\tilde{p}_s(t) - \tilde{p}_s(s) = \mathrm{dist}_{\tilde{\ell}+\ell_s}(s,t)$. ■

Theorem 3.3.4 $R(\hat{\ell}) \geq \frac{\lambda^*}{(1+2\epsilon)}$.

Proof: We shall estimate the numerator of $R(\hat{\ell})$ using Lemma 3.3.3. For a source s we have that

$$\begin{aligned}\sum_t d(s,t)\mathrm{dist}_{\hat{\ell}}(s,t) &\geq \sum_t d(s,t)\mathrm{dist}_{\tilde{\ell}+\ell_s}(s,t)\\ &= \sum_{vw}(\tilde{\ell}(vw)+\ell_s(vw))\tilde{f}_s^*(vw)\\ &= \tilde{C}_s^*(\lambda)+\sum_{vw}\ell_s(vw)\tilde{f}_s^*(vw).\end{aligned}$$

By the complementary slackness conditions, and the definition of ℓ_s in (3.8) we find that if $\ell_s(vw)\neq 0$, then $\tilde{f}_s^*(vw)=\lambda u(vw)$. Summing over all sources yields

$$\sum_{s,t\in V} d(s,t)dist_{\hat{\ell}}(s,t) \geq \sum_s \tilde{C}_s^*(\lambda)+\lambda\sum_{vw}u(vw)\sum_s\ell_s(vw).$$

Dividing the two sides of this inequality by $\sum_{vw}\hat{\ell}(vw)u(vw)$ we have

$$R(\hat{\ell}) \geq \frac{\sum_s \tilde{C}_s^*(\lambda)+\lambda\sum_{vw}u(vw)\sum_s\ell_s(vw)}{\sum_{vw}u(vw)\tilde{\ell}(vw)+\sum_{vw}u(vw)\sum_s\ell_s(vw)}. \tag{3.9}$$

Applying the fact that for positive a, b, x and λ, if $a/b \leq \lambda$ then $(a+\lambda x)/(b+x) \geq a/b$, we see that the left side of inequality (3.9) is at least $\tilde{Q}$ which by Lemma 3.3.2 is at least $\lambda^*/(1+2\epsilon)$. ■

Thus we have shown the following:

Corollary 3.3.5 An ϵ-optimal flow and length function pair (f,ℓ) produced by our concurrent flow algorithm can be translated into a length function $\hat{\ell}$ needed by the minimum-ratio cut algorithms in $O(k^*nm\log(n^2/m)\log(nU))$ time. The dual objective value associated with $\hat{\ell}$ is within a $1-O(\epsilon)$ factor of the optimum.

Proof: The algorithm needs to perform k^* minimum-cost flows. The time for a minimum-cost flow comes from Lemma 2.4.29. ■

We can use the approximate minimum-cost flow computation in Lemma 2.4.30 instead of Lemma 2.4.29. With an argument similar to the above, but somewhat more involved, we replace the $\log(n^2/m)$ in the theorem by a $\log\log(nU)$. We obtain the following corollary.

Corollary 3.3.6 An $O(\log n)$-approximation to the node weighted cut problem with general capacities can be found in $O(n^2 m \log nU \log^2 n \min\{\log(n^2/m), \log\log nU\})$ expected time. An $O(\log n \log kD)$ -approximation to the minimum-ratio cut problem with general demands and capacities can be found in $O(k^* nm \log nU \log k \log n \min\{\log(n^2/m), \log\log nU\})$ expected time.

An analogous theorem can be obtained for finding approximately sparsest cuts in hypergraphs using the concurrent flow algorithm in conjunction with the approximation algorithm of Makedon and Tragoudas [46].

3.3.4 A Faster Algorithm for Low Degree Graphs

In this section, we improve the running time given in Corollary 3.3.6 for low-degree graphs G. The new running time depends on Δ, the maximum degree of any node in the graph G.

We consider the following minimum-ratio cut problem for graphs with unit demands,

Problem MR1: Given an instance $\mathcal{I}$ where each commodity i has $d_i = 1$ and the graph that has an edge between the source and sink of each commodity is a constant degree expander on V. (We call this graph the *demand graph.*)

While Problem MR1 may seem like an obscure special case, it is in fact an important one. The Leighton and Rao [43] algorithm uses the solution of a concurrent flow problem in which the demand graph is the complete graph. As mentioned in Section 3.3.1, one can modify the Leighton and Rao algorithm to use the solution to this new concurrent flow problem and its dual problem to derive an $O(\log n)$ approximation to the minimum-ratio $u(\Gamma(A))/(|A||\bar{A}|)$ over all cuts. To get an idea how the two problems are related consider a cut $\Gamma(A)$ and assume that $|A| \le |\bar{A}|$. Since the demand graph is a constant degree expander, $c|A| \le d(A, \bar{A}) \le \hat{c}|A|$ for some constants c and $\hat{c}$. Therefore,

$$\frac{u(\Gamma(A))}{d(A, \bar{A})} = \Theta\left(\frac{u(\Gamma(A))}{|A|}\right).$$

But since by assumption, $|A| \le |\bar{A}|$, we know that $n/2 \le |\bar{A}| \le n$. Therefore, $u(\Gamma(A))/d(A, \bar{A})$ is $\Theta(n)$ times more than $u(\Gamma(A))/(|A||\bar{A}|)$.

The first step in solving problem MR1 is to round all edge capacities up to integer multiples of a parameter μ in such a way that the ratio $u(\Gamma(A))/(|A||\bar{A}|)$ is not changed by more than a factor of two. Notice that $|\Gamma(A)| \leq \Delta|A|$. We shall use r to denote the maximum of $|\Gamma(A)|/d(A,\bar{A})$ over all cuts $\Gamma(A)$. Notice that $r \leq \Delta/c$, where c is the expansion parameter of the demand graph.

Theorem 3.3.7 Let λ^* be the optimum value of the concurrent flow problem MR1, and let $\mu \leq (r\lambda^*)^{-1}$. If we round each capacity $u(e)$ up to $\hat{u}(e)$, the next integer multiple of μ, then the minimum value of $\hat{u}(\Gamma(A))/(|A||\bar{A}|)$ over all cuts is at most twice of the minimum value of of $\hat{u}(\Gamma(A))/(|A||\bar{A}|)$ over all cuts $\Gamma(A)$.

Proof: For all cuts $\Gamma(A)$, it must be that $\lambda^* u(\Gamma(A)) \geq d(A,\bar{A})$. The rounding error $\hat{u}(\Gamma(A)) - u(\Gamma(A))$ is at most $\mu|\Gamma(A)| \leq |\Gamma(A)|(2r\lambda^*)^{-1} \leq d(A,\bar{A})|/\lambda^* \leq u(\Gamma(A))$. Thus for each cut $\hat{u}(\Gamma(A)|/(|A||\bar{A}|) \leq 2u(\Gamma(A))/(|A||\bar{A}|)$, i.e., the new ratio is at most twice the old ratio. ■

Rounding to integer multiples of μ preserves the minimum-ratio cut up to a factor of 2. If we want to preserve λ^* up to a constant factor we must perform a somewhat finer rounding.

Theorem 3.3.8 Let λ^* be the optimum value of the concurrent flow problem MR1 and let $\mu \leq \epsilon(20r\lambda^* \log mU \log n)^{-1}$. If we round each capacity $u(e)$ up to $\hat{u}(e)$, the next integer multiple of μ, then the minimum congestion $\hat{\lambda}^*$ subject to capacities $\hat{u}$ is at most $\lambda^*/(1+\epsilon)$, where λ^* is the minimum congestion subject to capacities u.

Proof: The idea is to use the $O(\log n \log kD)$ approximation result of Klein, Agrawal, Ravi, and Rao [33] as improved by Tragoudas [66]. Klein et al. show that the minimum value over all cuts $u(\Gamma(A))/d(A,\bar{A})$ is within an $O(\log n \log kD)$ factor of the value of $1/\lambda^*$. Consider the following auxiliary concurrent flow problem. The graph is G with capacities u. For each edge $vw \in E$ there is a demand of value $d(v,w) = \hat{u}(vw) - u(vw)$ from v to w. Observe that the demands in the auxiliary problem are integral and at most μ, and $\log \mu$ is at most $\log(\epsilon mU/(20r \log nU \log n)) \leq 2\log(mU)$. Using the same estimates as in the proof of Theorem 3.3.7 we can conclude that the minimum of $u(\Gamma(A))/d(A,\bar{A})$ over all cuts $\Gamma(A)$ is at most $\epsilon/(20 \log mU \log n)$. By the approximation result of Klein et al. the minimum congestion λ^* for this problem is at most ϵ. That is, the added capacities can be routed in an ϵ-fraction of the original capacities u.

Now consider an optimal flow $\hat{f}$ of congestion $\hat{\lambda}^*$ in the rounded problem. To get a solution in the original problem, we route the part of flow $\hat{f}$ that uses the added capacity in the way this demand is routed in the optimal solution to the auxiliary problem. The additional flow does not increase the congestion by more than a factor of $1+\epsilon$. ■

Next consider the question of how long it takes to solve a rounded concurrent flow problem. For simplicity we shall restrict our attention to the case when ϵ is a constant. The number of commodities is $O(n)$. The capacities in the minimum-cost flow problem are integer multiples of $\lambda\mu$. We shall use algorithm ScalingConcurrent with a suitable choice of minimum-cost flow routine. We shall use the minimum-cost flow algorithm due to Ford and Fulkerson [17] and Yakovleva [71], that repeatedly augments the flow along the shortest path in the residual graph, to solve these problems. Given a concurrent flow with congestion λ, the number of shortest path computations in a minimum-cost flow subroutine is at most the demand divided by the unit of capacity, rounded up, that is, the number of minimum-cost flow computations is at most

$$\left\lceil \frac{1}{\mu\lambda} \right\rceil \leq \mu^{-1}\lambda^{-1} + 1.$$

We use these ideas to solve the minimum-ratio cut and the concurrent flow problem. The $O((\lambda^{-1}\mu^{-1}+1)(m+n\log n))$ time required for solving the minimum-cost flow problem might not dominate the $O(m\log n)$ needed to compute the approximate length function. To simplify the bounds we shall count each minimum-cost flow computation as $O((\lambda^{-1}\mu^{-1}+1)m\log n))$ time. These bounds can be further improved by using the data structures described in Section 2.5.5, but we do not pursue that here.

The running time that we wish to achieve is greater that the time it takes to find an initial flow using the k maximum-flow computations suggested in Lemma 2.4.2. The capacities of this problem are not rounded, therefore we have to use a general maximum-flow algorithm. All such algorithms take, up to logarithmic factors, $\Omega(mn)$ time. An initial flow that is optimal up to a factor of $O(km)$ can be computed in $O(km)$ time by routing each commodity on the path with maximum bottleneck capacity from its source to its sink.

An iteration of the algorithm uses the rounding described Theorem 3.3.7 with $\mu = c(\Delta\lambda_0)^{-1}$. We terminate the iteration if λ decreases below $\lambda_0/2$. At that point we divide λ_0 by 2, and start the next iteration. We use the flow obtained in the previous iteration as our initial flow.

Theorem 3.3.9 An $O(\log n)$-approximation to the minimum ratio $u(\Gamma(A))/(|A||\bar{A}|)$ over all cuts

$\Gamma(A)$ in a graph with capacities u and maximum degree Δ can be computed in $O(nm\Delta\log^3 n)$ expected time.

Proof: By Theorem 2.4.20, we need to perform $O(k \log n \log k)$ minimum-cost flow problems,after initialization. One iteration takes $O(\lambda^{-1}\mu^{-1}m\log n) = O((c\Delta+1)m\log n)$ time, which yields the result. ■

The proof of the following theorem is the same as the proof of the previous theorem with the rounding from Theorem 3.3.7 replaced by that of Theorem 3.3.8 and with $\mu = \epsilon c(20\Delta\lambda_0 \log mU \log n)^{-1})$.

Theorem 3.3.10 For any constant ϵ, an ϵ-optimal solution to a unit demand concurrent flow problem in a graph with maximum degree Δ and with a constant degree expander demand-graph can be computed in $O(nm\Delta\log^4 n \log nU)$ expected time.

In regular graphs $n\Delta = m$, and therefore the running times of the above two algorithms for problem MR1 are, up to polylogarithmic factors, $O(m^2)$.

Chapter 4

Implementing Multicommodity Flow Algorithms[1]

4.1 Introduction

In this chapter we describe an implementation of algorithm ScalingConcurrent. In Section 4.2 we will discuss some of the previous implementations of multicommodity flow algorithms. In Section 4.3 we discuss some of the decisions we made. In Section 4.4, we analyze the running time of our implementation and make some comparisons to a linear programming algorithm.

4.2 Previous Results

All previous implementations of algorithms for the concurrent flow problem with general capacities that we are aware of rely on linear programming. An instance $\mathcal{M}$ of the concurrent flow problem can be expressed as the following linear program:

[1]This chapter describes joint work with Tishya Leong and Peter Shor [44].

minimize λ

subject to

$$
\begin{array}{rcll}
\sum_{wv\in E} f_i(wv) - \sum_{vw\in E} f_i(vw) & = & 0, & \text{for every node } v \notin \{s_i, t_i\}, \\
 & & & i = 1,\ldots,n \\
\sum_{vw\in E} f_i(vw) & = & d_i, & \text{for } v = s_i, i = 1,\ldots,n; \\
\sum_{wv\in E} f_i(wv) & = & d_i, & \text{for } v = t_i, i = 1,\ldots,n; \\
\sum_{i=1}^{k} f_i(vw) & \leq & \lambda \cdot u(vw), & \forall vw \in E; \\
f_i(vw) & \geq & 0, & \forall vw \in E, i = 1,\ldots n
\end{array}
\tag{4.1}
$$

This linear program has $O(mk)$ variables and $O(nk + m)$ constraints. Even for a graph with average vertex degree Δ, there are $O(\Delta nk + mk) = O(mk)$ non-zero entries in the constraint matrix. Thus the size of the linear program is fairly large compared with the size of the input. In particular, both the number of constraints and the number of variables grows linearly in k. The large size of the linear programs makes the general simplex algorithm impractical for all but very small problems. Some algorithms that take advantage of the special structure of multicommodity flow problems have been proposed. These algorithms fall into three main classes: price-directive decomposition, resource-directive decomposition, and partitioning approaches. More recent approaches include interior-point methods [1] and a combinatorial scaling algorithm [54]. All of the aforementioned algorithms solve multicommodity flow problems using one of two different objective functions. Some find a minimum-cost multicommodity flow, while others find a flow that maximizes the total amount of flow in the network. A detailed description of these approaches requires a knowledge of linear programming that is beyond the scope of this thesis. We refer the reader to the surveys of Assad [5] and Kennington [31] and the thesis of Schneur [54] for more information on these approaches.

We are aware of two implementations of algorithms for the unit-capacity unit-demand concurrent flow problem. Shahrokhi and Matula [59] report encouraging results for an implementation of their algorithm. Klein, Kang and Borger [34] have implemented a variant of the algorithm of Klein, Stein and Tardos [36] which is essentially the algorithm ScalingUnit. Initial comparisons to the algorithm of Klein et al. show that, as expected, our algorithm performs fewer iterations, but take more time to perform each iteration.

4.3 An Implementation

We now describe how we have adapted and implemented Algorithm ScalingConcurrent. We made some modifications to the algorithm for the purpose of improving actual performance. We describe the changes we have made and the motivations behind them. We also point out areas in which our modifications could be fine-tuned with further research. First we will focus on a few of the more interesting and important aspects of our implementation.

4.3.1 Grouping Commodities

The grouping of commodities is suggested in Lemma 2.2.1. We place all commodities with the same source into one commodity group and run the algorithm on the commodity groups instead of on the individual commodities. Grouping has two advantages. First, the running time, which varies linearly with k, now depends on the number of commodity groups rather than on the number of commodities. For problems with large numbers of commodities, the favorable dependence on k means a significant reduction in running time. Second, because our algorithm uses $O(km)$ space, commodity grouping also reduces the space requirement by up to a factor of n. In practice, this advantage probably outweighs the previous one. We have been able to solve problems through the use of grouping that were not solvable without grouping, due to the memory limitations of the particular machine. We also note that the minimum-cost flow code that we used is written to handle multiple sources and sinks, so grouping does not create any added complexity. The advantages gained by grouping commodities have also been documented by Schneur [54].

4.3.2 Choosing a Commodity to Reroute

Recall that algorithm ScalingConcurrent uses either a deterministic strategy or a randomized strategy for choosing a commodity group (or a commodity) to reroute. Let f_i be the current flow of commodity group i and let f_i^* be the minimum-cost flow for problem $\mathcal{M} = (G, u \cdot \lambda, \ell, \hat{d}_i)$. Then the deterministic method, described in Lemma 2.4.11, computes the cost $C_i = \sum_{vw \in E} |f_i(vw)|\ell(vw)$ of a commodity group i, its minimum cost $C_i^* = \sum_{vw \in E} |f_i^*(vw)|\ell(vw)$, and the difference $C_i - C_i^*$ between its cost and the minimum cost. The commodity group to be rerouted is the first ϵ-bad one found in a predetermined ordering, in other words, the first one which has a difference $C_i - C_i^*$ greater than $\epsilon C_i + (\epsilon\lambda\Phi)/k^*$. This method requires k^* minimum-cost flow computations per iteration in the worst case. The randomized strategy computes the cost C_i of each commodity group i and randomly chooses a commodity group with probability proportional to cost. This method uses an expected ϵ^{-1} minimum-cost flow computations per iteration. Once every k^* iterations, minimum-cost flows are computed for all the commodity groups, and the congestion λ is checked against the lower bound $\sum_{i=1}^{k^*} C_i^*(\lambda)/\Phi$ to decide if the algorithm should terminate. This check increases the number of minimum-cost flow computations by at most a factor of 2. Our selection strategy draws from both the deterministic and the randomized methods and from the termination check.

To make the most progress per iteration, we attempt to find not only a poorly routed commodity group but the most poorly routed commodity group. We may designate as the most poorly routed commodity group either the group with the highest cost C_i or the group with the largest difference $C_i - C_i^*$ between cost and minimum cost. Using either measure and rerouting larger fractions of flow than the σ in Lemma 2.4.8 we have found that an algorithm that deterministically reroutes the most poorly routed commodity group sometimes gets stuck rerouting a single group over and over with no decrease in the congestion. We have also found that when it does not get stuck, such a deterministic algorithm usually progresses faster than a randomized algorithm. We therefore use a partly deterministic, partly randomized selection strategy in which we alternate between $k^*/2$ iterations of deterministic selection and $k^*/2$ iterations of random selection. By taking advantage of the minimum-cost flow computations performed in the termination check every k^* iterations, we can select commodity groups to reroute without computing extra minimum-cost flows. We reroute, in decreasing order, the $k^*/2$ groups with the greatest difference between cost and minimum-cost followed by $k^*/2$ randomly chosen commodity groups. To prevent domination by a limited number of groups, the random selection weights all commodity groups equally as proposed by Goldberg [21] and Grigoriadis and Khachiyan [27]. Note the savings over the theory here. We compute the costs once and then perform several reroutings. By the time that we actually reroute a commodity, it might be the case that that commodity is no longer ϵ-bad. The savings gained by not having to recompute costs at each iteration more than compensates for the extra reroutings performed.

4.3.3 Implementing the Minimum-cost Flow

Once the algorithm has chosen a commodity group to reroute, it must find an appropriate minimum-cost flow. For this purpose, we use the RELAXT-III minimum-cost flow code of Bertsekas and Tseng [8]. One drawback of the routine we have chosen is that it requires integer capacities, costs, and demands, making preprocessing and postprocessing necessary each time it is called. Another routine might better suit our algorithm, but we concentrate mainly on the number of iterations of our algorithm and treat the minimum-cost flow routine as a black box.

For the costs used to calculate the minimum-cost flow, we use a slightly different length function from that proposed in Algorithm DECONGEST. Instead of setting the length $\ell(vw)$ of each edge $vw \in E$ equal to $e^{\alpha\lambda(vw)}/u(vw)$. we use a length function in which $\ell(vw) = \left\lfloor e^{\alpha(\lambda(vw)-\lambda)+c} \right\rfloor$, where c is a scaling constant that depends on the largest integer the system can handle. (Note the similarity to the techniques used in Section 2.4.2.) We include the terms $-\lambda$ and c because we want to extract real flows from a routine that works only with integers. These terms spread the lengths over the range of viable non-negative integers, giving us the most accurate minimum-cost flow we can procure. We have removed the $u(vw)$ factor so that edges with equally high congestion have equally high cost in the minimum-cost flow. We have found through limited experimentation that this strategy produces minimum-cost flows which better suit our algorithm.

4.3.4 Choosing Constants and Rerouting

As is evident from the analysis of SCALINGCONCURRENT in Chapter 2, the constant α and the fraction σ of flow rerouted greatly affect the running times of the algorithm. The values used in algorithm DECONGEST are very large for α and very small for σ. The constant α can easily exceed 1000, and σ can easily fall below 10^{-5}. For the algorithm to progress at a reasonable rate in practice, given the fixed precision of computers, we need to use smaller values for α and larger values for σ. As the algorithm progresses, however, we need to use a larger α and a smaller σ (see the description of DECONGEST for details). We control the rate of growth of α and σ by means of a scaling factor s. We set α equal to $c' \cdot s/\lambda$, where c' is the constant $c - \log m$. Here c is chosen to be the largest value x such that $m \cdot e^x$, the largest possible value of the potential function, does not cause overflow. One key to making progress is to decide when to decrease s.

To decide how much flow to reroute, we sample the values that the potential function Φ would take after rerouting various fractions of flow. We do not need to restrict ourselves to a value of σ that guarantees improvement in *every* iteration, we only need to choose a value that guarantees us improvement in that particular iteration, thereby allowing for the possibility of rerouting much larger fractions of flow than in procedure REDUCE. In fact, we can try to choose the best possible value for σ, i.e., the one that gives the greatest reduction in Φ. We can find σ efficiently because Φ is a concave function.

More precisely, we take advantage of the following:

Lemma 4.3.1 Let f be a flow and f_i^* be the minimum-cost flow computed by procedure DECONGEST. Let $\Phi(\sigma)$ be the value of the potential function after rerouting a σ fraction of the flow from f_i onto f_i^*. Then $\Phi(\sigma)$ is a concave function with respect to σ.

Proof: We will prove the lemma for the potential function given in Chapter 2. We shall use the notation $\exp(x)$ to denote e^x. Recall that

$$\begin{aligned}\Phi &= \sum_{vw \in E} u(vw)\ell(vw) \\ &= \sum_{vw \in E} \exp\left(\alpha \frac{f(vw)}{u(vw)}\right)\end{aligned}$$

$$= \sum_{vw \in E} \exp\left(\frac{\alpha}{u(vw)} \sum_{i=1}^{k} |f_i(vw)|\right).$$

Thus after rerouting σ units of flow

$$\Phi(\sigma) = \sum_{vw \in E} \exp\left(\frac{\alpha}{u(vw)}\left(\left(\sum_{j \neq i} |f_j(vw)|\right) + |(1-\sigma)f_i(vw) + \sigma f_i^*(vw)|\right)\right). \tag{4.2}$$

Observe that for each edge vw, the quantity $\exp\left(\frac{\alpha}{u(vw)}\left(\sum_{j\neq i} |f_j(vw)|\right)\right)$, which we denote by $Q(vw)$, is independent of σ and is always positive. We use $S(vw)$ to denote the sign of $(1-\sigma)f_i(vw) + \sigma f_i^*(vw)$, i.e., $S(vw) = 1$ if that quantity is positive and -1 otherwise. We can now rewrite equation (4.2) as

$$\Phi(\sigma) = \sum_{vw \in E} Q(vw) \exp\left(\frac{\alpha S(vw)}{u(vw)}((1-\sigma)f_i(vw) + \sigma f_i^*(vw))\right). \tag{4.3}$$

We can now take the first derivative of $\Phi(\sigma)$ with respect to σ,

$$\sum_{vw \in E} \left(Q(vw) \exp\left(\frac{\alpha S(vw)}{u(vw)}((1-\sigma)f_i(vw) + \sigma f_i^*(vw))\right) \frac{\alpha S(vw)}{u(vw)}(-f_i(vw) + f_i^*(vw))\right),$$

and the second derivative

$$\sum_{vw \in E} \left(Q(vw) \exp\left(\frac{\alpha S(vw)}{u(vw)}((1-\sigma)f_i(vw) + \sigma f_i^*(vw))\right) \left(\frac{\alpha S(vw)}{u(vw)}\right)^2 (-f_i(vw) + f_i^*(vw))^2\right). \tag{4.4}$$

Now observe that the multiplicands in (4.4) are all positive, and hence the second derivative is always positive and the function is concave. Note that it is also true for the Φ that is used in practice as this Φ can be written is the Φ here with each term multiplied by a positive constant. ■

Since Φ is concave, we know that it has at most one local minimum. Thus if we search for the minimum by sampling 5 equally spaced point, we know that we can always eliminate 1/4 of the possible values at each iteration. Thus we can efficiently find the minimum, or at least a point close to the minimum. We sample fractions to the precision $.001/s^2$, and we also use this value as a floor $\sigma_{\min}$ on the fraction of flow that can be rerouted. To avoid wasting time rerouting small amounts of flow, we reroute a commodity only if σ is at least as large as $\sigma_{\min}$. We know that we may have to reroute fractions as small as $O(\epsilon/\alpha\lambda)$, and so we must decrease $\sigma_{\min}$ faster than we increase α to lower the minimum value for σ. We begin with s equal to .25 and raise it by .25 whenever the maximum fraction rerouted in k^* iterations is less than $\sigma_{\min}/(s \cdot k^*)$ or whenever the ratio of the congestion λ to its lower bound $\sum_{i=1}^{k^*} C_i^*(\lambda)/\Phi$ increases after k^* iterations. We have found that this strategy works well in most instances but scales α too fast in a few instances, slowing the algorithm too much for practical use. In such cases, we rerun the algorithm, scaling α more slowly. We have not yet discovered the optimal rate at which we should scale α, nor have we discovered exactly when we should scale it. This is the area in which our algorithm would benefit most from further research. Other areas in which it could be further improved include the selection strategy for commodities to reroute and the technique for choosing $\sigma_{\min}$.

4.4 Experimental Results

We have tested our algorithm on a variety of problems and compared its performance to the theoretical bounds. We used two different random network generators, NETGEN[37] and RMFGEN[24]. RMFGEN generates graphs that have a set of square planes with connections between adjacent planes. When we refer to a graph generated by NETGEN, we will indicate the number of planes. When run on random NETGEN and RMFGEN graphs with randomly placed commodities, our algorithm behaved more or less as expected. It took polynomially in ϵ^{-1} more time to get closer to the optimal solution and less than linearly in k^* more time to handle larger numbers of commodities. Furthermore, for large numbers of commodities, our algorithm outperformed the linear programming-based code of Kennington. Our algorithm performed poorly on one real problem provided by the GTE Corporation, but as we explain in Section 4.4.5 we consider this an anomaly arising from a limited number of unusually time-consuming minimum-cost flow computations. This one instance aside, we find our results encouraging and consider it an improvement, in many cases, over the simplex-based algorithms that have preceded it.

4.4.1 Dependence on the Error Parameter

The theory predicts an inverse polynomial dependence of the running time on the error parameter ϵ. More precisely, it states that the number of minimum-cost flow computations depends on ϵ^{-2}. Since our algorithm computes a constant number of minimum-cost flows per iteration, the number of iterations should also depend on ϵ^{-2}. Equivalently, ϵ should depend on $1/\sqrt{\# \text{ of iterations}}$.

We ran our algorithm on various problems and graphed the lowest ϵ achieved against the number of iterations completed. Each run stopped at a final ϵ of .001 or less. To compress the data, we used data points representing ranges of iterations. For each problem, we considered 10 runs and, for each run, the minimum ϵ achieved at each termination check. The aggregate ϵ for a range equaled the average of the minimum ϵ values found at the termination checks falling in the range during each of the 10 runs. We examined a problem with 20 commodities and four problems with 10 commodities using different NETGEN graphs with 50 nodes and 100 edges. We also examined two problems with 10 and 20 commodities, respectively, using an RMFGEN graph with 140 edges and 48 nodes (spread evenly over 12 square planes). To test a large problem, we examined a single run on a large RMFGEN problem with 700 commodities, 2075 edges, and 500 nodes (spread over 20 square frames). For all these problems, we graphed ϵ versus the number of iterations. We also graphed the function $1/\sqrt{\# \text{ of iterations}}$ on which we expected ϵ to depend. As is evident from Figures 4.5 through 4.11, our implementation always performed better than the expected bounds. Some inconclusive attempts at fitting the data to a curve of the form $a * (\# \text{ of iterations})^b + c$ using a regression package lends some additional support to this conclusion as typical values of b were between $-.5$ and -1.

4.4.2 Dependence on the Number of Commodities

With respect to the number of commodities k, our algorithm also seems to conform to the theoretical bounds. Using 10 runs for each data point, we graphed the average number of iterations needed to solve problems with variable numbers of commodities given a fixed graph. In Figure 4.12, we examined four NETGEN graphs with 50 nodes and 100 edges and values of k between 10 and 70. In Figure 4.13, we traced the same values of k using an RMFGEN graph with 140 edges and 48 nodes (spread over 12 square planes). In Figure 4.14, using values of k between 50 and 250, we examined an RMFGEN graph with 752 edges and 192 nodes (spread over 12 square planes). Graphing the number of iterations against the number of commodity groups $k^* \leq k$, we observed that the number of iterations either grew linearly or grew linearly to a peak and then dropped. The drops may result from larger numbers of commodities making it possible to route commodities over a smaller number of paths and over shorter paths. In trying to find flows that give the edges equal congestion, the algorithm has more commodities at its disposal to congest each edge. In any case, the number of iterations grows no more than linearly with the number of commodity groups and therefore no more than linearly with the number of commodities.

4.4.3 Comparison to Other Algorithms

Because the running time of our algorithm grows no more than linearly with the number of commodities, it can effectively solve large concurrent flow problems. To the best of our knowledge, our implementation is the first for an algorithm that finds an ϵ-optimal solution to the general concurrent flow problem. Consequently, comparisons to existing algorithms inherently contain some amount of bias. We have nevertheless compared our algorithm to another as best we could. The fact that our algorithm runs faster than another on a particular problem instance does not necessarily mean our algorithm is faster in general. The comparison reveals sufficiently consistent trends, however, that enable us to draw some general conclusions.

We begin with a brief discussion of the algorithm to which we have compared our algorithm. The algorithm is MCNF85, a special purpose simplex code for multicommodity flow problems written by Kennington [32]. We chose it for two reasons. First, we had access to the code on our machine. Second, and more importantly, previous tests by Adler, Karmarkar, Resende and Veiga [1] demonstrate its efficiency. Adler et al. compared three different codes for multi-commodity flow: MINOS 5.0, which is an advanced implementation of the simplex method [47], MCNF85, and their own interior point method. Their experiments show that the running time of MINOS grows much faster than that of the other two algorithms and that, for the problems they tested, MCNF85 and the interior point algorithm have comparable running times. Thus we concluded that MCNF85 was one of the best codes available at that time. Several people, however, have pointed us towards codes, particularly interior point codes, that may possibly be better than MCNF85 on this class of problems. We are in the process of comparing our algorithm against these other codes and will report the results when they become available.

We faced two obstacles in comparing our algorithm to MCNF85. First, our algorithm finds an approximate solution while MCNF85 finds an exact solution. Since we did not possess the programming skills needed to modify MCNF85 to alleviate this problem, we ran our algorithm to both $\epsilon = .01$ and $\epsilon = .001$ before comparing it to MCNF85. The second difficulty in making the comparison is that the algorithms are designed for different objective functions. By using an objective function of 0 for MCNF85 and a cost of 0 on each edge, we can treat it as an algorithm that determines whether a feasible multicommodity flow exists. We could then call this algorithm $O(\log(n\epsilon^{-1}))$ times to find an ϵ-optimal solution to a concurrent flow problem, but to do so seems too far from the original purpose of the algorithm for fair comparison. Instead, we ran our algorithm to find the maximum z for which there exists a feasible flow satisfying a percentage z of each demand. We then scaled the demands by z to get a problem that we knew to be feasible. This problem corresponds to the problem that MCNF85 would have to solve in the *last* iteration of the binary search procedure defined above. We compared a run of our algorithm to a run of MCNF85 with the input modified as described above. We could better evaluate our algorithm by comparing it to other approximation codes for the same problem. As mentioned above, however, we could not make such a comparison because we do not know of any such codes.

4.4.4 The Results

The results of our experiments appear in Figure 4.1. The experiments in this table were performed on a Silicon Graphics 4D/340S. They show that as the number of commodities increases, the running time of MCNF85 grows much more rapidly than the running time of our algorithm for graphs of all sizes. The difference does not arise simply because we group the commodities (they could incorporate grouping in their algorithm too). Hardly any grouping occurred in the graphs with 500 nodes and 70 or less commodities, and the running time of our algorithm still grew much more slowly than the time for MCNF85. In fact, as discussed above, the running time of our algorithm grows slower than k while rough analysis of the data shows

Problem Specification				Kennington	Our algorithm	
nodes	edges	commodities	generator		$\epsilon = .01$	$\epsilon = .001$
50	100	20	NG	49	20	103
50	100	50	NG	397	35	43
50	100	70	NG	857	29	33
48	140	10	RMF	8	13	13
48	140	20	RMF	24	23	23
48	140	30	RMF	69	18	35
48	140	40	RMF	122	25	38
48	140	50	RMF	216	21	71
48	140	60	RMF	316	40	61
48	140	70	RMF	470	45	62
500	2075	10	RMF	87	831	5230
500	2075	20	RMF	608	1484	2641
500	2075	30	RMF	1831	2625	3881
500	2075	40	RMF	6571	3762	6084
500	2075	50	RMF	15601	4710	7401
500	2075	60	RMF	18449	3819	6201
500	2075	70	RMF	34362	4435	8258
500	2075	700	RMF		22411	
192	748	50	RMF	2702	240	589
192	748	250	RMF	85754	637	1571
49	260	585	none	1373	2472 (estimate)	

Figure 4.1: Running time comparison of our algorithm and Kennington's algorithm. Running times are in seconds on a Silicon Graphics machine. NG is NETGEN and generator RMF is RMFGEN. The last problem is the problem defined in Section 4.4.5

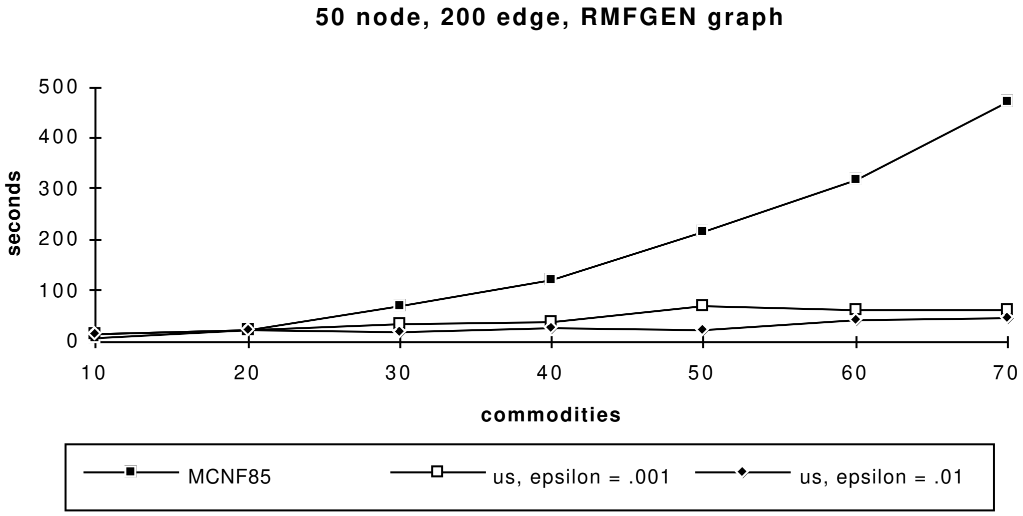


Figure 4.2: A comparison between our algorithm and MCNF85. This problem has 50 nodes and 200 edges.

that the time for MCNF85 grows at least as fast as k^2. We show two examples graphically in Figure 4.2 and 4.3. Since the size of the linear program grows by k^2, this growth is not particularly surprising.

Our algorithm will be able to solve large and previously unsolvable multicommodity flow problems. We have already shown that we can solve a 700 commodity problem faster than MCNF85 can solve a 70 commodity problem. For large graphs with small numbers of commodities, our algorithm is slower than MCNF85. The rapid growth rate of MCNF85, however, with respect to the number of commodities makes our algorithm more desirable for problems with more than a few commodities. As discussed in Chapter 3, one of the motivations for this work comes from multicommodity flow problems that arise in approximating various NP-hard problems. (See [43],[33],[35], and [42] for details.) These problems have large numbers of commodities, i.e., at least as many commodities as the number of nodes. Our algorithm provides a practical means for solving such problems.

4.4.5 An Anomaly

In one case, a problem with 49 nodes, 260 edges, and 585 commodities using actual data from GTE, our algorithm performed much more poorly than the linear programming algorithm.

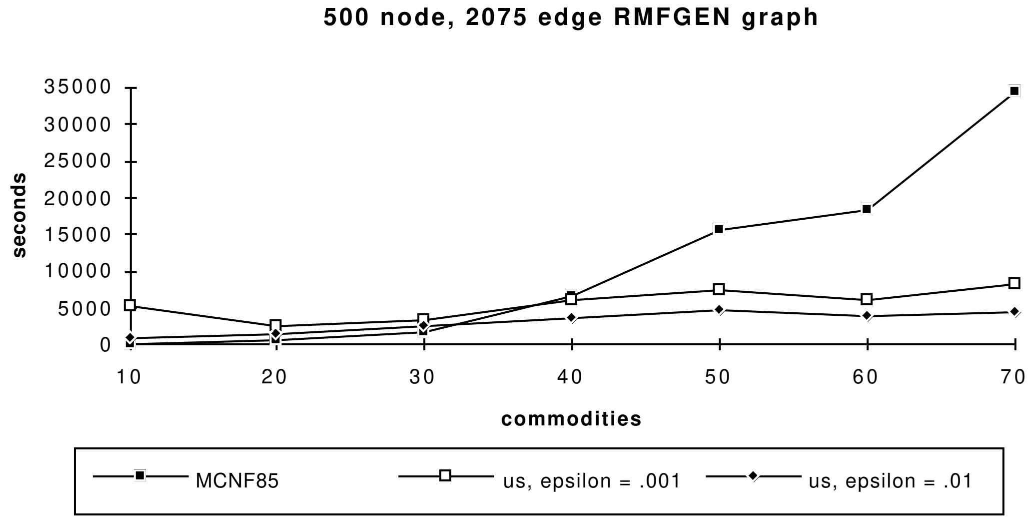


Figure 4.3: A comparison between our algorithm and MCNF85. This problem has 500 nodes and 2075 edges.

Though our algorithm ran for only 3745 iterations, a reasonable number, those iterations took a total of 18.4 hours of CPU time. We attribute this anomaly to inefficiency in the minimum-cost flow routine since minimum-cost flow computations accounted for over 99.8% of the running time. The theory shows that minimum-cost flow computations dominate the running time of the algorithm, but even for the much larger RMFGEN graph with 500 nodes and 1025 edges, minimum-cost flow computations generally took less than 80% of the time. For small graphs, they generally took between 40 and 50 percent of the time. See Figure 4.4 for a more detailed description of the times. The time spent solving the GTE problem was not equally divided between iterations. Iterations including the termination check aside, most iterations took less than 100 milliseconds. Some iterations, however, took hundreds of seconds, up to 1000 times the normal duration.

With the help of several other researchers, we have verified that these are problems on which RELAXT-III takes an inordinately long amount of time. Several people have run these problems on their codes and observed no anomalous behavior, i.e., the running times for this set of problems are all approximately the same. In order to estimate a more realistic running time for this problem, we compute an upper bound on the what the running time would have if we were using the RNET code of Grigoriadis. Joseph Cheriyan [10] has reported that on a

Problem Specification				ϵ	% of time finding
nodes	edges	commodities	generator		Min-cost flows
50	100	20	NG	.001	49.9
50	100	50	NG	.001	50.5
50	100	70	NG	.001	43.6
48	140	30	RMF	.001	44.1
48	140	40	RMF	.001	44.1
48	140	50	RMF	.001	42.4
48	140	60	RMF	.001	44.7
48	140	70	RMF	.001	47.7
500	2075	10	RMF	.01	87.3
500	2075	30	RMF	.01	79.7
500	2075	40	RMF	.01	73.3
500	2075	50	RMF	.01	76.8
500	2075	60	RMF	.01	80.7
500	2075	70	RMF	.01	77.7
192	752	50	RMF	.001	55.8
192	752	250	RMF	.001	59.0
49	260	585	none	.01	**99.8**

Figure 4.4: Percentage of Time that our algorithm spent performing minimum-cost flows. The data is gotten from the UNIX profiling routine **prof**. NG is NETGEN and generator RMF is RMFGEN. The last problem is the problem defined in Section 4.4.5.

representative sample of these minimum-cost flow problems, the running time of RNET on a SPARC2 (which is slower than our machine) never exceeds 0.66 seconds. Using the estimate that 50% (see Figure 4.4) of the time is spent in the minimum-cost flow computations, we arrive at a figure of 2472 seconds as a "reasonable" upper bound on the running time of this instance.

4.5 Conclusions and Open Problems

Our algorithm performs as well as, and often better than, the theoretical bounds. The theory predicts the number of iterations of the algorithm to be $O(\epsilon^{-2}k)$. Our experiments show that the number of iterations often grows slower as a function of ϵ. Our experiments also show that for small k, the number of iterations does increase linearly with k. As k approaches the number of nodes, however, the number of iterations grows at most linearly and sometimes actually decreases.

On the problems we tested, the running time of our algorithm grew much slower as a function of k than that of Kennington's algorithm, thereby implying that our algorithm is preferable to one of the best network simplex based approaches for problems with large numbers of commodities.

The performance of our algorithm was heavily influenced by our choice of when to scale α. We tested several strategies and found that different strategies performed better for different problems. We therefore believe that more work is needed to find a strategy that works well for all problems.

Our algorithm might be improved by using a different minimum-cost flow algorithm. In fact, we do not require the exact solution to a minimum-cost flow but only an approximate solution. An algorithm that is able to find fast approximations to a minimum-cost flow might significantly improve the running time of our algorithm. Also, the minimum-cost flow problems we solve for the same commodity might have similar solutions. Using the solution to the previous problem as a starting point for the new problem might improve the running time.

We are aware of two other implementations of combinatorial algorithms to which we should compare our algorithm. The first, by Shahrokhi and Matula [59], works only for graphs in which every capacity and demand is 1, but it would still be interesting to see how our algorithm compares to theirs on this class of graphs. The second, by Schneur [54], also works by gradually rerouting flow. She has shown that her algorithm runs well on many problems. We would like to compare the algorithms on the same machine and the same problems.

We conclude by mentioning a valuable lesson learned about accuracy. In the theoretical results of Chapter 2, a good deal of technical effort was used to convert the results from a model of computation in which infinite precision is used to a RAM model of computation. It is tempting to view this work as being of purely theoretical interest. After all, modern computers can perform arithmetic operations on real numbers almost as quickly as they can perform arithmetic operations on integers. The single biggest difficulty in implementing this algorithm, however, was dealing with the finite precision of the computer. The exponents of the length functions that we wished to compute were typically too big for the computer. Many decisions had to be made about how to scale and round these numbers. In making these decisions the theoretical work on adapting the algorithm for the RAM model of computation was very useful.

x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\text{\# of iterations}}$.
The bottom curve is the minimum ϵ achieved.

Figure 4.5: NETGEN graph with 50 nodes, 100 edges, and 20 commodities.

x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\# \text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

Figure 4.6: NETGEN graph with 50 nodes, 100 edges, and 10 commodities.

x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\#\text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

Figure 4.7: NETGEN graph with 50 nodes, 100 edges, and 10 commodities.

x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\#\text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

Figure 4.8: NETGEN graph with 50 nodes, 100 edges, and 10 commodities.

x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\text{\# of iterations}}$.
The bottom curve is the minimum ϵ achieved.

Figure 4.9: RMFGEN graph with 48 nodes, 140 edges, and 10 commodities.

x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\# \text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

Figure 4.10: RMFGEN graph with 48 nodes, 140 edges, and 20 commodities.

x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\# \text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

Figure 4.11: RMFGEN graph with 500 nodes, 2075 edges, and 700 commodities.

x-axis is # of commodity groups.
y-axis is # of iterations.
Each curve represents a set of runs on one of four different underlying graphs.

Figure 4.12: NETGEN graphs with 50 nodes, 100 edges, and from 10 through 70 commodities.

x-axis is # of commodity groups.
y-axis is # of iterations.
The curve represents a set of runs on one underlying graph.

Figure 4.13: RMFGEN graphs with 48 nodes, 140 edges, and from 10 through 70 commodities.

x-axis is # of commodity groups.
y-axis is # of iterations.
The curve represents a set of runs on one underlying graph.

Figure 4.14: RMFGEN graphs with 192 nodes, 740 edges, and from 50 through 250 commodities.

Chapter 5

Approximation Algorithms for Shop Scheduling[1]

5.1 Introduction

Shop scheduling refers to a large class of problems that typically arise in a shop, factory or assembly line setting. The shop has m machines, and in the basic environment each machine is different and performs a different function. Each job consists of a set of *operations*, each of which must be processed on a particular machine; a job may have more than one operation on a particular machine. We wish to produce a *schedule* that assigns a period of time to each operation during which it is processed on the appropriate machine. The goal is to minimize the *completion time* of the last operation to complete, while ensuring that no more than one operation is assigned to a machine at any point in time and no two operations of the same job are scheduled simultaneously.

A variety of constraints may be introduced on the order of execution of the operations of the job, and different sorts of constraints yield different well-known versions of the problem. (We focus only on order constraints between the operations of each job, and not between operations of different jobs.) For example, if we impose a strict total order on the order of execution of the operations of a job, the problem is a *job shop* scheduling problem. If the total order is the

[1]This chapter describes joint work with David Shmoys and Joel Wein [60].

same total order for every job, and each job has at most one operation on each machine, we have a *flow shop* scheduling problem. If there is no order at all imposed on the execution of any job's operations, we have an *open shop* problem. It is traditional in the scheduling literature to focus, for the open shop problem, on the case when each job is processed on each machine at most once (since operations on the same machine can be coalesced). We refer to the general shop scheduling problem that does not fall into one of the three above categories as the *dag shop* problem.

In this thesis we concentrate primarily on the job shop scheduling problem, for two reasons. First of all, most of our results for other shop problems can be obtained as easy corollaries of our results for the job shop problem. Second, the job shop problem is probably the most famous and most difficult of all the versions of the problem. It is strongly $\mathcal{NP}$-hard, and moreover, except for the cases when there are two jobs or when there are two machines *and* each job has at most two operations, essentially all special cases of this problem are $\mathcal{NP}$-hard, and typically strongly $\mathcal{NP}$-hard [20, 39]. For example, it is $\mathcal{NP}$-hard even if there are 3 machines, 3 jobs and each operation is of unit length; in this case we can think of the input length as the maximum number of operations in a job, μ.

In addition to this theoretical evidence of the difficulty of the job shop problem, it is also one of the most notoriously difficult $\mathcal{NP}$-hard optimization problems in terms of practical computation, even with very small instances being difficult to solve exactly. A striking example of this difficulty is that a single instance of the problem involving only 10 jobs, 10 machines and 100 operations, which first appeared in a book by Muth and Thompson in 1963, remained unsolved for 23 years despite repeated attempts to find an optimal solution [39]. Today, due to better algorithms and faster machines, instances with 10 jobs and 10 machines seem to be tractable. Applegate and Cook solved ten different 10×10 problems, including the notorious instance mentioned above, in times ranging from 90 seconds to 42 minutes. (It is interesting to note that the instance of Muth and Thompson was one of the easier instances to solve using their technique). Slightly larger instances, however, are still currently intractable; they report instances of size 10×15, 15×20, 15×15 and 10×20 that they were unable to solve [4].

Formal Definition and Previous Results

We formally define the job shop problem as follows. We are given a set $\mathcal{M} = \{m_1, m_2, \ldots, m_m\}$ of machines, a set $\mathcal{J} = \{J_1, \ldots, J_n\}$ of jobs, and a set $\mathcal{O} = \{O_{ij} | i = 1, \ldots, \mu_j, j = 1, \ldots, n\}$ of operations, where κ_{ij} indexes the machine on which operation O_{ij} runs. Thus m is the number of machines, n is the number of jobs, μ_j is the number of operations of job J_j, and $\mu = \max_j \mu_j$. O_{ij} is the ith operation of J_j; it requires processing time on a given machine $m_k \in \mathcal{M}$, where $k = \kappa_{ij}$, for an uninterrupted period of a given length p_{ij}. (In other words, this is a *non-preemptive* model. A model in which operations may be interrupted and resumed at a later time is called a *preemptive* model.) Each machine can process at most one operation at a time, and each job may be processed by at most one machine at a time. If the completion time of operation O_{ij} is denoted by C_{ij}, then the objective is to produce a schedule that minimizes the maximum-completion time, $C_{\max} = \max_{i,j} C_{ij}$; the optimal value is denoted by $C^*_{\max}$.

It is possible to extend this model by associating with each job J_j a *release date* r_j, on which J_j becomes available for processing. A theorem of Shmoys, Wein and Williamson [61] shows that the length of the optimal schedule is no more than twice the length of the optimal schedule for the corresponding problem without release dates. All our results thus apply to this model, with the corresponding bounds multiplied by 2.

The formal definition of the flow, open or dag shop problems are almost the same, except for the following small differences:

- *flow shop:* $\kappa_{ij} = \kappa_{ij'}$, for all i, j, j', and $\kappa_{ij} \neq \kappa_{i'j}$ for all i, i', j.
- *open shop:* The O_{ij} can be processed in *any* order.
- *dag shop:* For each job j we define a partial order on the O_{ij} and require that they be processed in any total order consistent with that partial order.

There are two fundamental lower bounds on the length of an optimum schedule. Since each job must be processed, $C^*_{\max}$ must be at least the maximum total length of any job, $\max_{J_j} \sum_i p_{ij}$, which we shall call the *maximum job length* of the instance, $P_{\max}$. Furthermore, each machine must process all of its operations, and so $C^*_{\max}$ must be at least $\max_{m_k} \sum_{\kappa_{ij}=k} p_{ij}$, which we call the *maximum machine load* of the instance, $\Pi_{\max}$. These lower bounds apply regardless of whether we have a job, flow, open or dag shop problem.

There has been a tremendous amount of literature on shop scheduling problems over the last thirty years [39]. We mentioned earlier that all but the most restrictive versions of the job shop problem are $\mathcal{NP}$-hard, as are the other versions of the problem. When there are at least 3 machines both the open and flow shop problems are $\mathcal{NP}$-hard [39]. When there are just two machines both these problems are known to be in $\mathcal{P}$ [29, 25]. In contrast, the two-machine job shop problem is only known to be polynomial-time solvable if each job has at most two operations, or if each operation has unit size [39].

Despite all the attention, however, surprisingly little has been known about approximation algorithms for shop scheduling problems. In fact, all that was known was the following observation by Gonzales and Sahni:

Theorem 5.1.1 [26] An algorithm $\mathcal{A}$ for the job shop problem that produces a schedule in which at least one machine is running at any point in time is an m-approximation algorithm.

Proof: The length of the schedule produced by such an algorithm $C_{\max}(\mathcal{A})$ is bounded above by $\sum_{i,j} p_{ij}$, since some operation is always being executed. On the other hand, the *average machine load*, $\sum_{i,j} p_{ij}/m$, is a lower bound on the *maximum machine load*, which is a lower bound. The theorem follows directly. ■

Little was also known in the way of negative results, results that indicate it is difficult to approximate these problems. Recently, however, Williamson, Hall, Hoogeveen, Hurkens, Lenstra, and Shmoys [70], extending work by Williamson [69], have shown that unless $\mathcal{P} = \mathcal{NP}$, none of these problems can be approximated arbitrarily closely.

Theorem 5.1.2 [70] Unless $\mathcal{P} = \mathcal{NP}$, there is no polynomial-time algorithm that approximates any of the job shop, flow shop or open shop problems within a factor of less than $\frac{5}{4}$. ■

Despite the lack of knowledge about approximation algorithms with good worst-case relative error guarantees, there are two relevant results that are important to our work. The most interesting approximation algorithms to date for job shop scheduling have appeared primarily in the Soviet literature and are based on a beautiful connection to geometric arguments. This approach was independently discovered by Belov and Stolin [7] and by Sevast'yanov [56] as well as by Fiala [15]. This approach typically produces schedules for which the length can be bounded by $\Pi_{\max} + q(m, \mu)p_{\max}$, where $q(\cdot, \cdot)$ is a polynomial, and $p_{\max} = \max_{ij} p_{ij}$ is the maximum operation length. For the job shop problem, Sevast'yanov [57, 58] gave a polynomial-time algorithm that delivers a schedule of length at most $\Pi_{\max} + O(m\mu^3)p_{\max}$. The bounds obtained in this way do not give good worst-case relative error bounds. Even for the special case of the *flow shop problem*, the best algorithms to date delivered solutions of length $\Omega(mC^*_{\max})$.

Since these results are not well known in the West, yet are important tools for us, we provide here a bit of information about the proof of the flow shop result, which is simpler than the more general job shop result. This simpler presentation of the proof is due to David Shmoys [62].

Theorem 5.1.3 There exists a polynomial time algorithm $\mathcal{A}$ for the flow shop problem that yields a schedule of length bounded above by $C^*_{\max} + m(m-1)p_{\max}$.

Proof:

The proof relies heavily on the following lemma.

Lemma 5.1.4 Let $\{v_1, v_2, \ldots, v_n\}$ be a set of d-dimensional vectors such that $\sum_{j=1}^{n} v_j = 0$. There exists a polynomial-time algorithm that computes a permutation π such that for any $k = 1, \ldots, n$, $||\sum_{j=1}^{k} v_{\pi(j)}|| \leq d \max_j ||v_j||$, where we use $||x||$ to denote the L_1-norm of x.

Without loss of generality, we can assume that the load on each machine is equal to the maximum machine load, namely $\Pi_{\max}$. In this case the completion time of the schedule is $\Pi_{\max} + I$, where I is the amount of idle time on the last machine before it starts processing the last operation of the last job to complete on it. If we choose a permutation π of the n jobs and schedule their operations in that order on every machine, the condition $\sum_{l=1}^{j}(p_{i-1,\pi(l)} - p_{i,\pi(l)}) \leq (m-1)p_{\max}$ yields an upper bound of $m(m-1)p_{\max}$ on I.

Now if we construct a set of n m-dimensional vectors v_j, where $v_j = (p_{1j} - p_{2j}, p_{2j} - p_{3j}, \ldots, p_{m-1,j} - p_{mj})$, the algorithm mentioned in the previous lemma produces the necessary permutation. ■

Another important result on shop scheduling comes, somewhat surprisingly, from the literature on packet routing. Leighton, Maggs and Rao [41] have proposed the following model for the routing of packets in a network: find paths for the packets, and then schedule the transmission of the packets along these paths so that no two packets traverse the same edge simultaneously. The primary objective is to minimize the time by which all packets have been delivered to their destination.

The scheduling problem considered by Leighton, Maggs and Rao is simply the job shop scheduling problem with each processing time $p_{ij} = 1$. They also added the restriction that each path does not traverse any edge more than once, or in scheduling terminology, each job has at most one operation on each machine. This restriction of the job shop problem remains (strongly) $\mathcal{NP}$-hard [39]. The main result of Leighton, Maggs and Rao was to show that for their special case of the job shop problem, there always exists a schedule of length $O(\Pi_{\max} + P_{\max})$. Unfortunately, their result is not algorithmic, as it relies on a nonconstructive probabilistic argument based on the Lovász Local Lemma. They also obtained a randomized algorithm that delivers a schedule of length $O(\Pi_{\max} + P_{\max} \log n)$, with high probability.

We can now state our main theorem.

Theorem 5.1.5 There exists a polynomial-time randomized algorithm for job shop scheduling, that, with high probability, yields a schedule that is of length $O(\frac{\log^2(m\mu)}{\log\log(m\mu)} C^*_{\max})$.

Our techniques are useful not only for the job shop problem, but can easily be extended to the general problem of *dag shop scheduling*. Another important generalization is the situation where, rather than having m different machines, there are m' types of machines, and for each type, there are a specified number of identical machines; each operation, rather than being assigned to one machine, may be processed on any machine of the appropriate type. These problems have significant practical importance, since in real-world shops, we expect that a job need not follow a total order and that the shop has more than one copy of many of its machines. We will give approximation algorithms with the same performance guarantees for this generalization as well.

When m and μ are constants, we can achieve much better approximation guarantees. Specifically, we give a $(2 + \epsilon)$-approximation algorithm for this special case. Finally, we give *parallel* approximation algorithms for all the scheduling models mentioned above and some improved results for the open shop problem.

While all the algorithms that we give are polynomial-time, they are all also rather inefficient.

Most rely on the algorithms of Sevast'yanov, and for example, his algorithm for job shop scheduling takes $O((\mu mn)^2)$ time. As a result, we do not refer explicitly to running times throughout the remainder of this chapter. In Chapter 6, we will carefully consider the running time for a deterministic version of the algorithm presented in this chapter.

The rest of this chapter is organized as follows. In Section 5.2 we extend the basic technique of Leighton, Maggs and Rao to the general job shop problem. In Section 5.3 we show how to scale and reduce the input data so that the techniques of Section 5.2 yield good performance bounds. In Section 5.4 we show how our techniques apply to more general problems. We conclude with a discussion of the open shop problem in Section 5.5 and some open problems in Section 5.6.

5.2 The Basic Algorithm

In this section we extend the technique due to Leighton, Maggs and Rao [41] of assigning random delays to jobs to the general case of non-preemptive job shop scheduling. A valid schedule assigns at most one job to a particular machine at any time, and schedules each job on at most one machine at any time. Their approach, for the special case of unit-size operations and at most one operation of each job on each machine, was to first create a schedule that obeyed only the second constraint, and then build from this a schedule that satisfies both constraints and is not much longer. An outline of their strategy follows:

1. Define the *oblivious* schedule, where each job starts running at time 0 and runs continuously until all of its operations have been completed. This schedule is of length $P_{\max}$, but there may be times when more than one job is assigned to a particular machine.

2. Perturb this schedule by delaying the start of the first operation of each job by a random integral amount chosen uniformly in $[0, \Pi_{\max}/\log n]$. The resulting schedule, with high probability, has no more than $O(\log n)$ operations assigned to any machine at any time.

3. Reschedule each unit of time t into $O(\log n)$ units of time during which each of the $O(\log n)$ operations scheduled for time t is processed. The resulting (valid) schedule is of length $O(P_{\max}\log n + \Pi_{\max})$.

Our strategy builds upon this framework of Leighton, Maggs and Rao. Whereas Step 1 is the same and Step 2 differs in only a few technical details, the essential difficulty in obtaining the generalization is in Step 3.

2. Perturb this schedule by delaying the start of the first operation of each job by a random integral amount chosen uniformly in $[0, \Pi_{\max}]$. The resulting schedule, with high probability, has no more than $O(\frac{\log(n\mu)}{\log\log(n\mu)})$ jobs assigned to any machine at any time.

3. "Spread" this schedule so that at each point in time all operations currently being processed have the same size, and then "flatten" this into a schedule that has at most one job per machine at any time.

For the analysis of Step 2, we assume that $p_{\max}$ is bounded above by a polynomial in n and μ. In the next section we will show how to remove this assumption. As is usually the case, we assume that $n \geq m$; analogous bounds can be obtained when $n < m$.

Lemma 5.2.1 Given a job shop instance in which $p_{\max}$ is bounded above by a polynomial in n and μ, the strategy of delaying each job an initial integral amount chosen randomly and uniformly from $[0, \Pi_{\max}]$ and then processing its operations in sequence yields an (invalid) schedule that has length at most $\Pi_{\max} + P_{\max}$ and, with high probability, has no more than $O(\frac{\log(n\mu)}{\log\log(n\mu)})$ jobs scheduled on any machine during any unit of time.

Proof: Fix a time t and a machine m_i; consider p = Prob[at least τ units of processing are scheduled on machine i at time t]. There are at most $\binom{\Pi_{\max}}{\tau}$ ways to choose τ units of processing from all those required on m_i. If we focus on a particular one of these τ units and a specific time t, then the probability that it is scheduled at time t is at most $1/\Pi_{\max}$, since we selected a delay uniformly at random from among $\Pi_{\max}$ possibilities. If all τ units are from different jobs, then the probability that they are all scheduled at time t is at most $(\frac{1}{\Pi_{\max}})^{\tau}$ since the delays are chosen independently. Otherwise, the probability that all τ are scheduled then is 0, since it is impossible. Therefore,

$$p \leq \binom{\Pi_{\max}}{\tau}\left(\frac{1}{\Pi_{\max}}\right)^{\tau}$$

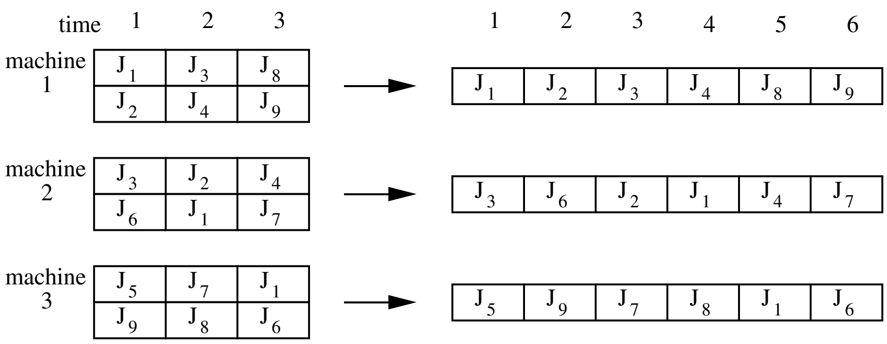


Figure 5.1: Flattening a schedule in the case with unit length operations.

$$
\begin{aligned}
&\leq \left(\frac{e\Pi_{\max}}{\tau}\right)^{\tau}\left(\frac{1}{\Pi_{\max}}\right)^{\tau} \\
&\leq \left(\frac{e}{\tau}\right)^{\tau}.
\end{aligned}
$$

If $\tau = k\frac{\log(n\mu)}{\log\log(n\mu)}$, then $p < (n\mu)^{-(k-1)}$. To bound the probability that *any* machine at *any* time has more than $k\frac{\log(n\mu)}{\log\log(n\mu)}$ jobs using it, multiply p by $P_{\max} + \Pi_{\max}$ for the number of time units in the schedule, and by m for the number of machines. Since we have assumed that $p_{\max}$ is bounded by a polynomial in n and μ, $P_{\max} + \Pi_{\max}$ is as well; choosing k large enough yields that, with high probability, no more than $k\frac{\log(n\mu)}{\log\log(n\mu)}$ jobs are scheduled for any machine during any unit of time. ■

In the special case of unit-length operations treated by Leighton, Maggs and Rao, a schedule $\mathcal{S}$ of length L that has at most c jobs scheduled on any machine at any unit of time can trivially be "flattened" into a valid schedule of length cL by replacing one unit of $\mathcal{S}$'s time with c units of time in which we run each of the jobs that was scheduled for that time unit. (See Figure 5.1.)

For *preemptive* job shop scheduling, where the processing of an operation may be interrupted, each unit of an operation can be treated as a unit-length operation and a schedule that has multiple operations scheduled simultaneously on a machine can easily be flattened into a valid schedule. This strategy is not possible for *non-preemptive* job shop scheduling, and in fact it seems to be more difficult to flatten the schedule in this case. We give an algorithm that takes a schedule of length L with at most c operations scheduled on one machine at any time and produces a schedule of length $O(cL \log p_{\max})$.

Lemma 5.2.2 Given a schedule $\mathcal{S}_0$ of length L that has at most c jobs scheduled on one machine during any unit of time, there exists a polynomial-time algorithm that produces a valid schedule of length $O(cL \log p_{\max})$.

Proof: To begin, we round each processing time p_{ij} up to the next power of 2 and denote the rounded times by p'_{ij}; that is, $p'_{ij} = 2^{\lceil \log_2 p_{ij} \rceil}$. Let $p'_{\max} = \max_{ij} p'_{ij}$. From $\mathcal{S}_0$, one can obtain a schedule $\mathcal{S}$ that uses the modified p'_{ij} and is at most twice as long as $\mathcal{S}_0$. Furthermore, an optimal schedule for the new problem is no more than twice as long as an optimal schedule for the original problem.

A *block* is an interval of a schedule with the property that each operation that begins during this interval has length no more than that of the entire interval. (Note that this does not mean that the operation finishes within the interval.) We can divide $\mathcal{S}$ into $\left\lceil \frac{L}{p'_{\max}} \right\rceil$ consecutive blocks of size $p'_{\max}$. We will give a recursive algorithm that reschedules – "spreads" – each block of size p (where p is a power of 2) into a sequence of schedule *fragments* of total length $p \log p$. The operations scheduled in a fragment of length T all have length T and start at the beginning of the fragment. This algorithm takes advantage of the fact that if an operation of length p is scheduled to begin in a block of size p, then that job is not scheduled on any other machine until after this block. Therefore, that operation can be scheduled to start after all the smaller operations in the block have finished.

To reschedule a block B of size $p'_{\max}$, we first construct the final fragment (which has length $p'_{\max}$), and then we construct the preceding fragments by recursive calls of the algorithm. For each operation of length $p'_{\max}$ that begins in B, reschedule that operation to start at the beginning of the final fragment, and delete it from B. Now each operation that still starts in B has length at most $p'_{\max}/2$, so B can be subdivided into two blocks, B_1 and B_2, each of size $p'_{\max}/2$, and we can recurse on each. See Figure 5.2.

The recurrence equation that describes the total length of the fragments produced from a block of size T is $f(T) = 2f(\frac{T}{2}) + T; f(1) = 1$. Thus $f(T) = \Theta(T \log T)$, and each block B in $\mathcal{S}$ of size $p'_{\max}$ is spread into a schedule of length $p'_{\max} \log p'_{\max}$. By spreading the schedule $\mathcal{S}$, we

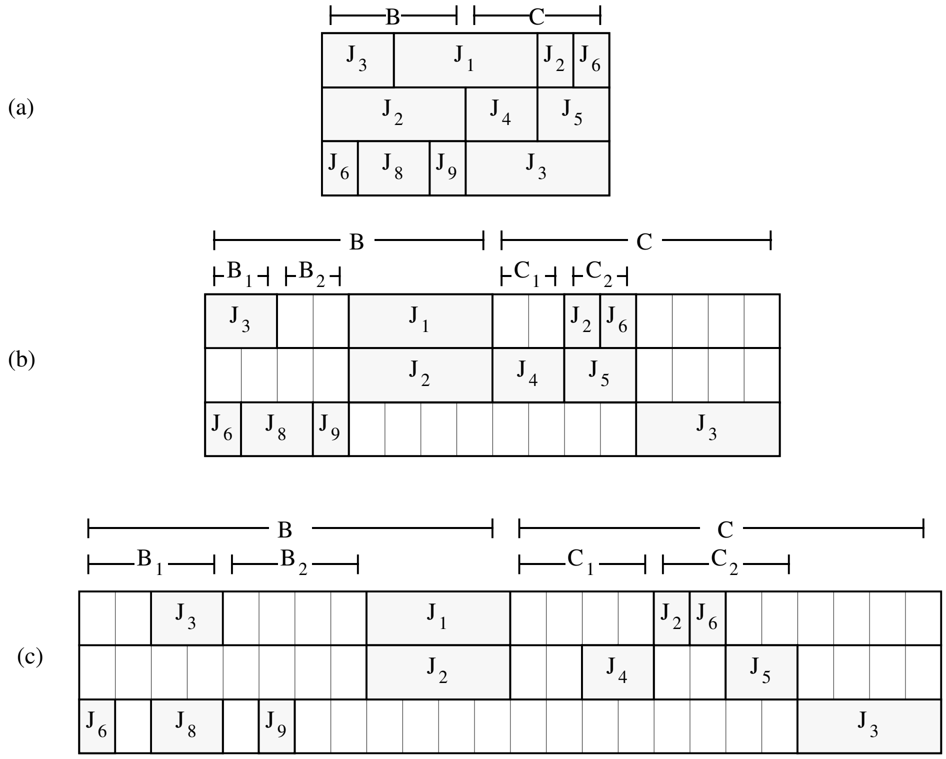


Figure 5.2: (a) The initial greedy schedule of length 8. $p'_{\max} = 4$. (b) The first level of spreading. All jobs of length 4 have been put in the final fragments. We must now recurse on B_1 and B_2 with $p'_{\max} = 2$. (c) The final schedule of length $8 \log_2 8 = 24$.

produce a new schedule $\mathcal{S}'$ that satisfies the following conditions:

1. At any time in $\mathcal{S}'$, all operations scheduled have the same length. Furthermore, any two operations either start at the same time or do not overlap.

2. If $\mathcal{S}$ has at most c jobs scheduled on one machine at any time, then $\mathcal{S}'$ has at most c jobs scheduled on one machine at any time as well.

3. $\mathcal{S}'$ schedules a job on at most one machine at any time.

4. $\mathcal{S}'$ does not schedule the ith operation of job J_j until the first $i-1$ are completed.

Condition 1 is satisfied by each pair of operations on the same machine by the definition of spreading and it is satisfied by each pair of operations on different machines because the division of time into fragments is the same on all machines. To prove condition 2, note that operations of length T that are scheduled at the same time on the same machine in the expanded schedule started in the same block of size T on that machine. Since they all must have been scheduled during the last unit of time of that block, there can be at most c of them.

To prove condition 3, note that if a job is scheduled by $\mathcal{S}'$ on two machines simultaneously, then it must have been scheduled by $\mathcal{S}$ to start two operations of length T in the same block of length T on two different machines. Consequently, it was scheduled by $\mathcal{S}$ on two machines during the last unit of time of that block, which violates the properties of $\mathcal{S}$.

Finally, we verify condition 4 by first noting that if two operations of a job are in different blocks of size $p'_{\max}$ in $\mathcal{S}$, then they are certainly rescheduled in the correct order. Therefore it suffices to focus on the schedule produced from one block. Within a block, if an operation is rescheduled to the final fragment, then it is the last operation for that job in that block. Therefore $\mathcal{S}'$ does not schedule the ith operation of job J_j until the first $i-1$ are completed.

The schedule $\mathcal{S}'$ can easily be flattened to a schedule that obeys the constraint of one job per machine at any time, since c operations of length T that start at the same time can just be executed one after the other in total time cT. Since what we are doing is effectively synchronizing the entire schedule block by block, it is important when flattening the schedule to make each machine wait enough time for all machines to process all operations of that fragment length, even if some machines have no operations of that length in that fragment.

The schedule $\mathcal{S}'$ has length $L \log p'_{\max}$; therefore the flattened schedule has length $Lc \log p'_{\max}$.
■

We note in passing that the inclusion of release dates into the problem does not affect the quality of our bounds at all. The release dates can either be directly included into probabilistic analysis of lemma 5.2.1, or we can view each release date as one additional initial operation on some (imaginary) machine.

5.3 Reducing the Problem

In the previous section we showed how to produce, with high probability, a schedule of length

$$O\left((\Pi_{\max} + P_{\max})\frac{\log(n\mu)}{\log\log(n\mu)}\log p_{\max}\right),$$

under the assumption that $p_{\max}$ was bounded above by a polynomial in n and μ. Since

$$\Pi_{\max} + P_{\max} = O(\max\{\Pi_{\max}, P_{\max}\})$$

this schedule is within a factor of $O(\frac{\log(n\mu)}{\log\log(n\mu)}\log p_{\max})$ of optimal. In this section, we first remove the assumption that $p_{\max}$ is bounded above by a polynomial in n and μ by showing that we can reduce the general problem to that special case while only sacrificing a constant factor in the approximation, thereby yielding an $O(\frac{\log^2(n\mu)}{\log\log(n\mu)})$–approximation algorithm. Then we how how to sacrifice another constant factor to reduce to the special case that n is polynomially bounded in m and μ. Combining these two results, we conclude that we can reduce the general job shop problem to the case where n and $p_{\max}$ are polynomially bounded in m and μ, while changing the performance guarantee by only a constant.

5.3.1 Reducing $p_{\max}$

First we show that we can reduce the problem to one where $p_{\max}$ is bounded by a polynomial in n and μ. Let $\omega = |\mathcal{O}|$ be the total number of required operations. Note that $\omega \leq n\mu$. Round down each p_{ij} to the nearest multiple of $p_{\max}/\omega$, denoted by p'_{ij}. Now there are at most ω distinct values of p'_{ij} and they are all multiples of $p_{\max}/\omega$. Therefore we can treat the p'_{ij} as integers in $\{0, \ldots, \omega\}$; a schedule for this problem can be trivially rescaled to a schedule $\mathcal{S}'$ for the actual p'_{ij}. (Note that assigning $p'_{ij} = 0$ does not mean that this operation does not exist; instead, it should viewed as an operation that takes an arbitrarily small amount of time.) Let L denote the length of $\mathcal{S}'$. We claim that $\mathcal{S}'$ for this reduced problem can be interpreted as a schedule for the original operations that has length at most $L + p_{\max}$. When we adjust the p'_{ij} up to the original p_{ij}, we add an amount that is at most $p_{\max}/\omega$ to each p'_{ij}. Since the length of a schedule is determined by a critical path through the operations and there are ω operations, we add a total amount of at most $p_{\max}$ to the length of the schedule; thus, the new schedule has length at most $L + p_{\max} \leq L + C^*_{\max}$. Therefore we have rounded a general instance $\mathcal{I}$ of the job shop problem to an instance $\mathcal{I}'$ that can be treated as having $p_{\max} = O(n\mu)$; further, a schedule for $\mathcal{I}'$ yields a schedule for $\mathcal{I}$ that is no more than $C^*_{\max}$ longer. Thus, we have proved the following lemma:

Lemma 5.3.1 There exists a polynomial-time algorithm that transforms any instance of the job shop scheduling problem into one with $p_{\max} = O(n\mu)$ with the property that a schedule for the modified instance of length $kC^*_{\max}$ can be converted in polynomial time to a schedule for the original instance of length $(k+1)C^*_{\max}$.

5.3.2 Reducing the Number of Jobs

To reduce an arbitrary instance of job shop scheduling to one with a number of jobs polynomial in m and μ we divide the jobs into big and small jobs. We say that job J_j is *big* if it has an operation of length more than $\Pi_{\max}/(2m\mu^3)$; otherwise, we call the job *small*. For the instance consisting of just the short jobs, let $\Pi'_{\max}$ and $p'_{\max}$ denote the maximum machine load and operation length, respectively. Using the algorithm of [58] described in the introduction, we can, in time polynomial in the input size, produce a schedule of length $\Pi'_{\max} + 2m\mu^3 p'_{\max}$ for this instance. Since $p'_{\max}$ is at most $\Pi_{\max}/(2m\mu^3)$ and $\Pi'_{\max} \leq \Pi_{\max}$, we get a schedule that has length no more than $2\Pi_{\max}$. Thus, an algorithm that produces a schedule for the long jobs that is within a factor of k of optimal yields a $(k+2)$-approximation algorithm. Note that there can be at most $2m^2\mu^3$ long jobs, since otherwise there would be more than $m\Pi_{\max}$ units of processing to be divided amongst m machines, which contradicts the definition of $\Pi_{\max}$. Thus we have shown:

Lemma 5.3.2 There exists a polynomial-time algorithm that transforms any instance of the job shop scheduling problem into one with $O(m^2\mu^3)$ jobs with the property that a schedule for the modified instance of length $kC^*_{\max}$ can be converted in polynomial time to a schedule for the original instance of length $(k+2)C^*_{\max}$.

From the results of the previous two sections we can conclude that:

Theorem 5.3.3 There exists a polynomial-time randomized algorithm for job shop scheduling, that, with high probability, yields a schedule that is of length $O(\frac{\log^2(m\mu)}{\log\log(m\mu)}C^*_{\max})$.

Proof: In Section 2 we showed how to produce a schedule of length

$$O\left((\Pi_{\max}+P_{\max})\frac{\log(n\mu)}{\log\log(n\mu)}\log p_{\max}\right)$$

under the assumption that $p_{\max}$ was bounded above by a polynomial in n and μ. From Lemmas 5.3.1 and 5.3.2 we know that we can reduce the problem to one where n and $p_{\max}$ are polynomial in m and μ, while adding only a constant to the factor of approximation. Since now $\log p_{\max} = O(\log(m\mu))$ and $\log n = O(\log(m\mu))$, our algorithm produces a schedule of length $O(\frac{\log^2(m\mu)}{\log\log(m\mu)}C^*_{\max})$. ■

Note that when μ is bounded by a polynomial in m, the bound only depends on m. In particular, this implies the following corollary:

Corollary 5.3.4 There exists a polynomial-time randomized algorithm for flow shop scheduling, that, with high probability, yields a schedule that is of length $O(\frac{\log^2 m}{\log\log m}C^*_{\max})$. ■

We now briefly address the issue of a parallel version of our shop scheduling algorithm. $\mathcal{NC}$ is the class of problems that can be solved using a polynomial number of processors and polylogarithmic time. $\mathcal{RNC}$ is the class that, in addition, allows each processor to general a $\log n$ bit random number at each step. Except for the use of Sevast'yanov's algorithm, all these techniques can be carried out in $\mathcal{RNC}$ We assign one processor to each operation. The rounding in the proof of Lemma 5.2.2 can be done in $\mathcal{NC}$. We set the random delays and inform each processor about the delay of its job. By summing the values of p_{ij} for all of its job's operations, each processor can calculate where its operation is scheduled with the delays and then where it is scheduled in the recursively spread-out schedule. These sums can be calculated via parallel prefix operations. With simple $\mathcal{NC}$ techniques we can assign to each operation a rank among all those operations that are scheduled to start at the same time on its machine, and thus flatten the spread out schedule to a valid schedule.

Corollary 5.3.5 There exists a $\mathcal{RNC}$ algorithm for job shop scheduling, that, with high probability, yields a schedule that is of length $O(\frac{\log^2(n\mu)}{\log\log(n\mu)}C^*_{\max})$.

5.3.3 A Fixed Number of Machines

Sevast'yanov's algorithm for the job shop problem can be viewed as a $(1+m\mu^3)$-approximation algorithm, which when m and μ are constant, is an $O(1)$-approximation algorithm; that is, it delivers a solution within a constant factor of the optimum. The technique of partitioning the set of jobs by size can be applied to give a much better performance guarantee in this case. Now, call a job J_j *big* if there is an operation O_{ij} with $p_{ij} > \epsilon\Pi_{\max}/(m\mu^3)$, where ϵ is an arbitrary positive constant. There are at most $m^2\mu^3/\epsilon$ big jobs, and since m, μ and ϵ are fixed, the number of jobs is constant.

Now use Sevast'yanov's algorithm to schedule all the small jobs. The resulting schedule is of length at most $(1+\epsilon)C^*_{\max}$. There are only a constant (albeit a huge constant) number of ways to schedule the big jobs. Therefore the best one can be selected in polynomial time and executed after the schedule of the short jobs. The additional length of this part is no more than $C^*_{\max}$.

Thus we have shown:

Theorem 5.3.6 For the job shop scheduling problem where both m and μ are fixed, there is a polynomial-time algorithm that produces a schedule of length $\leq (2+\epsilon)C^*_{\max}$.

5.4 Applications to More General Scheduling Problems

The fact that the quality of our approximations is based solely on the lower bounds $\Pi_{\max}$ and $P_{\max}$ makes it quite easy to extend our techniques to the more general problem of *dag shop scheduling*. We define $\Pi_{\max}$ and $P_{\max}$ exactly the same way, and $\max\{\Pi_{\max}, P_{\max}\}$ remains a lower bound for the length of any schedule. We can convert this dag shop scheduling problem to a job shop problem by selecting for each job an arbitrary total order that is consistent with its partial order. $\Pi_{\max}$ and $P_{\max}$ have the same values for both problems. Therefore, a schedule of length $\rho \cdot (\Pi_{\max} + P_{\max})$ for this job shop instance is a schedule for the original dag shop scheduling instance of length $O(\rho C^*_{\max})$.

A further generalization to which our techniques apply is where, rather than m different machines, we have m' types of machines, and for each type we have a specified number of identical machines of that type. Instead of requiring an operation to run on a particular machine, an operation now may run on any one of these identical copies. The value $P_{\max}$ remains a lower bound on the length of any schedule for this problem. The value $\Pi_{\max}$, which was a lower bound for the job shop problem must be replaced, since we do not have a specific assignment of operations to machines, and the sum of the processing times of all operations assigned to a type is *not* a lower bound. Let S_i, $i = 1, \ldots m'$, denote the sets of identical machines, and let $\Pi(S_i)$ be the sum of the lengths of the operations that run on S_i. Our strategy is to convert this problem to a job shop problem by assigning operations to specific machines in such a way that the maximum machine load is within a constant factor of the fundamental lower bounds for this problem. To obtain a lower bound on the maximum machine load, the best we can do is to distribute the operations evenly across machines in a set, and thus

$$\Pi_{\text{avg}} = \max_{S_i} \frac{\Pi(S_i)}{|S_i|}$$

is certainly a lower bound on the maximum machine load. Furthermore, we can not split operations, so $p_{\max}$ is also a lower bound. We will now describe how to assign operations to machines so that the maximum machine load of the resulting job shop scheduling problem is at most $2\Pi_{\text{avg}} + p_{\max}$. A schedule for the resulting job shop problem of length $\rho \cdot (\Pi_{\max} + P_{\max})$ yields a solution for the more general problem of length $O(\rho \cdot (\Pi_{\text{avg}} + P_{\max}))$. Sevast'yanov [58] used a somewhat more complicated reduction to handle a slightly more general setting.

For each operation O_{ij} to be processed by a machine in S_k, if $p_{ij} \geq \Pi(S_k)/|S_k|$, assign O_{ij} to one machine in S_k. There are certainly enough machines in S_k to make this assignment and all such operations contribute at most $p_{\max}$ to the maximum machine load. Those operations not yet assigned each have length at most $\Pi(S_k)/|S_k|$ and have total length $\leq \Pi(S_k)$. Therefore, these operations can be assigned easily to the remaining machines so that less than $2\Pi(S_k)/S_k$ processing units are assigned to each machine. Combining these two bounds, we get an upper bound of $2\Pi_{\mathrm{avg}} + p_{\max}$ on the maximum machine load which is within a constant factor of the lower bound of $\max\{\Pi_{\mathrm{avg}}, p_{\max}\}$.

Theorem 5.4.1 There exists a polynomial-time randomized algorithm for dag shop scheduling with identical copies of machines that, with high probability, yields a schedule that is of length at most $O(\frac{\log^2(m\mu)}{\log\log(m\mu)}C^*_{\max})$.

Corollary 5.4.2 There exists an $\mathcal{RNC}$ algorithm for dag shop scheduling with identical copies of machines that, with high probability, yields a schedule that is of length at most $O(\frac{\log^2(n\mu)}{\log\log(n\mu)}C^*_{\max})$.

5.5 The Open Shop Problem

Recall that in the open shop problem the operations of a job can be executed in any order. Fiala [16] has shown that if $\Pi_{\max} \geq (16m\log m + 21m)p_{\max}$, then $C^*_{\max}$ is just $\Pi_{\max}$, and there is a polynomial-time algorithm to find an optimal schedule, but in general this problem is strongly $\mathcal{NP}$-Complete. We will show that, in contrast to the job and flow shop problems, a simple greedy strategy yields a fairly good approximation to the optimal open shop schedule. Consider the algorithm that, whenever a machine is idle, assigns to it any job that has not yet been processed on that machine and is not currently being processed on another machine. Anná Racsmány [6] has observed that this greedy algorithm delivers a schedule of length at most $\Pi_{\max} + (m-1)p_{\max}$. We can adapt her proof to show that, in fact, the greedy algorithm delivers a schedule that is no longer than a factor of 2 times optimal. In fact Wein [68], has shown that even with release dates the greedy algorithm is a 2-approximation algorithm. He has also shown that this bound is fairly tight, since he can produce schedules of length $(2 - \frac{1}{m})$ times optimal. We include his proof here.

Theorem 5.5.1 The greedy algorithm for the open shop problem is a 2-approximation algorithm, even when each job J_j has an associated *release date* r_j on which it becomes available for processing.

Proof: Consider the machine m_k that finishes last in the greedy schedule. This machine is active sometimes, idle sometimes, and finishes by completing some job J_j. Since the schedule is greedy, whenever m_k is idle, J_j is either being processed by some other machine or has not yet been released. Therefore, the idle time is at most $\sum_{m_i \neq m_k} p_{ij} + r_j < P_j + r_j$. Thus, machine m_k is processing for at most $\Pi_{\max}$ units of time and is idle for less than $P_j + r_j$ units of time. This implies $C_{\max} < \Pi_{\max} + P_j + r_j$. However, the value $P_j + r_j$ is a lower bound on the length of the schedule, since no processing of job J_j could start until time r_j. ■

Using a slightly different (non-greedy) strategy, we can derive another algorithm that achieves a schedule of length $O(C^*_{\max} \log n)$. This algorithm is also easily parallelizable, thus putting the problem of finding an $O(\log n)$-approximation to the open shop scheduling problem in $\mathcal{NC}$.

We define the *jobs graph*, which is a bipartite graph that represents an instance of the open shop problem. One side of the bipartition contains m nodes, one for each machine, whereas the other side contains n nodes, one for each job. If job J_j has an operation on machine i then the jobs graph contains an edge between the respective nodes.

First consider the case when all operations have the same size, ℓ. Let Δ be the maximum degree of any node in the remaining jobs graph. Then $\ell\Delta$ is a lower bound on the length of the optimal schedule for this problem. However, since this graph is bipartite with maximum degree Δ, it can be edge-colored using exactly Δ colors. So we edge-color the graph, and then schedule the operations in each color class separately, thereby producing a schedule of length $\ell\Delta$, which is optimal. As long as there is at least one processor per operation, this coloring can be done in NC using the edge-coloring algorithm of Lev, Pippinger, and Valiant [45].

We can extend this algorithm to one that solves the general open shop problem by first using the techniques of Section 5.3.1 to reduce the problem to the case where all operations have sizes polynomial in n, and then by rounding the operation sizes so they are all powers of 2. Now there are only $O(\log n)$ different operation sizes. We schedule each one separately, using the edge-coloring based strategy described in the previous paragraph. The schedule we get for any particular ℓ is optimal for that operations of that size, and hence each of the $O(\log n)$ schedules we produce has length $O(C^*_{\max})$. Concatenating these schedules together, and observing that the rounding can easily be done in $\mathcal{NC}$, we obtain the following theorem:

Theorem 5.5.2 An open shop schedule of length $O(C^*_{\max} \log n)$ can be found in $\mathcal{NC}$.

We can also use the results on open shop to get a simple bound for general dag shop scheduling. For certain classes of problems this approach gives better bounds than those given in 5.3.3. Consider the case when the constraints for each job form a dag. We will refer to each job as a *node* of the dag. We define the *depth* of a node to be the distance of that node from the root and the *depth* of a dag to be the length of the longest root-leaf path in the dag. If each job j has an associated dag D_j, let d_j be the depth of D_j.

Theorem 5.5.3 Given a dag scheduling problem, let $d = \max_j d_j$, the maximum depth of any of the dags. Then there exists an algorithm that finds a schedule of length $O(dC_{\max})$.

Proof: For each dag D_j, let D^i_j be the set of all operations at level i in the dag. Our algorithm consists of d iterations. In iteration i, we consider the scheduling problem consisting of, for each job j, all jobs in D^i_j. The key observation is that in a dag, all jobs in any level are independent, i.e., there are no precedence constraints among them. Hence the scheduling problem for each i is an open shop problem. Further, since each of these problems is a subproblem of the original dag scheduling problem, by Theorem 5.5.1, there certainly exists a schedule for the jobs at level i of length $2C^*_{\max}$, where $C^*_{\max}$ is the length of the optimal schedule for the *original* dag scheduling problem. Concatenating the d schedules yields a schedule satisfying the conditions of the lemma. ∎

If d is constant, this approach yields constant factor approximation. Moreover, if each level has about $1/d$ of the processing of each job and $1/d$ of the processing of each machine, we also get a constant factor approximation, regardless of how many levels we have.

5.6 Conclusions and Open Problems

We have given the first polynomial-time polylog-approximation algorithms for minimizing the maximum completion time for the problems of job shop scheduling, flow shop scheduling, dag shop scheduling and a generalization of dag shop scheduling in which there are groups of identical machines. The most basic question to be pursued is the development of approximation algorithms with even better performance guarantees. It is our belief that the $O(\log p_{\max})$ factor that is introduced by the techniques of Section 5.2 can be improved upon, perhaps even by a simple greedy method. Such methods have proved frustratingly difficult to analyze, however. The other logarithmic factor in the performance bound seems much more difficult to improve upon.

An interesting consequence of our results is the following observation about the structure of shop scheduling problems. Assume we have a set of jobs that need to run on a set of machines. We know that any schedule for the associated open shop problem must have length $\Omega(\Pi_{\max} + P_{\max})$. Furthermore, we know that no matter what type of partial ordering we impose on the operations of each job we can produce a schedule of length $O((\Pi_{\max} + P_{\max})\frac{\log^2 m}{\log\log m})$. Hence for any instance of the open shop problem, we can impose an arbitrary partial order on the operations of each job and increase the length of the optimal schedule by a factor of no more than $O(\frac{\log^2 m}{\log\log m})$.

An interesting combinatorial question is, "Can the imposition of a partial order really make the optimal schedule that much longer than $O(\Pi_{\max} + P_{\max})$?" In other words, how good are $\Pi_{\max}$ and $P_{\max}$ as lower bounds? We have seen that in two interesting special cases—job shop scheduling with unit-length operations and open shop scheduling, there is a schedule of length $O(\Pi_{\max} + P_{\max})$. Does there always exist an $O(\Pi_{\max} + P_{\max})$ schedule for the general job, flow or dag shop scheduling problems?

Beyond this, there are several interesting questions raised by this work, including:

- Do there exist parallel algorithms that achieve the approximations of our sequential algorithms? For the general job shop problem achieving these approximations seems hard, since we rely heavily on the algorithm of Sevast'yanov. For open shop scheduling, however, a simple sequential algorithm achieves a factor of 2, whereas the best $\mathcal{NC}$ algorithm that we have achieves only an $O(\log n)$-approximation. As a consequence of the results above, all one would need to do is to produce any greedy schedule.

- Are there simple variants of the greedy algorithm for open shop scheduling that achieve better performance guarantees? For instance, how good is the algorithm that always selects the job with the maximum total (remaining) processing time?

- Our algorithms, while polynomial-time algorithms, are inefficient. Are there significantly more efficient algorithms that have the same performance guarantees?

Chapter 6

Derandomizing Shop Scheduling Via Flow Techniques

In this chapter, we show how the shop scheduling algorithms of the previous chapter can be made deterministic. One approach, which appears in [60], is to use a derandomized version of the randomized rounding techniques of Raghavan and Thompson [52], which are alluded to in Chapter 3. While this approach yields a polynomial-time algorithm, the polynomial is rather large, since the bottleneck step is the solution of a large linear program. Recently, Plotkin, Shmoys and Tardos [48] have generalized the multicommodity flow approximation algorithms of Chapter 2 to show how to approximate a large class of packing linear programs. In this chapter we shall use their results to obtain an algorithm that is much more efficient than the randomized rounding approach. The key will be to phrase the problem of choosing initial delays as the solution of a packing integer program. We then show how to find an integral solution to the linear relaxation of this program that is close to the optimal solution. Besides yielding a faster algorithm, this approach also yields a direct method for finding, in polynomial time, an approximate solution to a certain integer program. Previously, such solutions could only be found via the indirect method of randomized rounding. Our algorithm will imply the main result of this chapter: a deterministic version of the shop scheduling algorithm with almost the same performance guarantee as the randomized algorithm of Chapter 5.

6.1 A Deterministic Approximation Algorithm

In this section, we "derandomize" the results of the Chapter 5 , i.e., we give a deterministic polynomial-time algorithm that finds a schedule of length $O(\log^2(m\mu)C^*_{\max})$. Of all the components of the algorithm of Theorem 5.3.3, the only step that is not already deterministic is the step that chooses a random initial delay for each job so that, with high probability, no machine is assigned too many jobs at any one time. The reduction to the special case in which n and $p_{\max}$ are bounded by a polynomial in m and μ is entirely deterministic, and so we can focus on that case alone. We give an algorithm that deterministically assigns delays to each job so as to produce a schedule in which each machine has $O(\log(m\mu))$ jobs running at any one time. We then apply Lemma 5.2.2 to produce a schedule of length $O(\log^2(m\mu)C^*_{\max})$. The bound of $O(\log(m\mu))$ jobs per machine is not as good as the probabilistic bound of $O(\frac{\log(m\mu)}{\log\log(m\mu)})$. We do not know how to achieve a bound of $O(\frac{\log(m\mu)}{\log\log(m\mu)})$ deterministically[1]. By a proof nearly identical to that of Lemma 5.2.1, however, we can show that in order to achieve this weaker bound of $O(\log(m\mu))$ jobs per machine, we now only need to choose delays in the range $[0, \Pi_{\max}/\log(m\mu)]$. In fact, the reduced range of delays yields a schedule of length $O(P_{\max}\log^2(m\mu) + \Pi_{\max}\log(m\mu))$ which is within an $O(\log(m\mu))$ factor of optimal if $P_{\max} = O(\Pi_{\max}/\log(m\mu))$.

We now state the problem formally:

Problem 6.1.1 Deterministically assign a delay to each job in the range $[0, \Pi_{\max}/\log(m\mu)]$ so as to produce a schedule with no more than $O(\log(m\mu))$ jobs on any machine at any time.

The rest of this chapter focuses on solving Problem 6.1.1.

6.2 The Framework

In this section, we describe the framework of Plotkin, Shmoys and Tardos[48] for approximately solving packing linear programs. We will be somewhat vague in our description, since it is only meant to convey the main ideas of their work. In the next section, we will be more formal and prove the results, citing from [48] as needed.

[1] A recent result by Srinivasan [65] describes how to achieve this bound using different techniques.

Recall the concurrent flow problem that we solved in Chapter 2. We now state it as a linear program:

$$
\begin{array}{lll}
\text{minimize } \lambda & & \\
\text{subject to} & & \\
\displaystyle\sum_{wv\in E} f_i(wv) - \sum_{vw\in E} f_i(vw) = 0, & \text{for each node } v \notin \{s_i, t_i\}, & \\
& i = 1, \ldots, n; & (6.1) \\
\displaystyle\sum_{vw\in E} f_i(vw) = d_i, & \text{for } v = s_i, i = 1, \ldots, n; & (6.2) \\
\displaystyle\sum_{wv\in E} f_i(wv) = d_i, & \text{for } v = t_i, i = 1, \ldots, n; & (6.3) \\
\displaystyle\sum_{i=1}^{k} f_i(vw) \leq \lambda \cdot u(vw), & \forall vw \in E; & (6.4) \\
f_i(vw) \geq 0, & \forall vw \in E, i = 1, \ldots n. & (6.5)
\end{array}
$$

Consider algorithm Concurrent. It initially find a flow that is a feasible solution to this linear program, for some value of λ. Each iteration finds a new flow (f_i^*, the minimum-cost flow), and then takes a convex combination of the new and old flow, thereby producing a flow that is still a feasible solution, but for a smaller value of λ. Informally, each iteration can be thought of as tightening constraint (6.4).

Plotkin, Shmoys and Tardos [48] have shown that the techniques used for multicommodity flow problem in Chapter 2 apply to a much wider class of problems. They have given approximation algorithms for a large class of *packing linear programs*. A packing linear program is one that can be expressed in the following form:

$$
\begin{array}{lll}
\text{minimize } \lambda & & \\
\text{subject to} & & \\
x \in P; & & (6.6) \\
Ax \leq \lambda b; & & (6.7) \\
x \geq 0; & & (6.8)
\end{array}
$$

where A is an $p \times q$ non-negative matrix, b is an p-dimensional non-negative vector, and P is a convex set in $\mathbf{R}^q_+$. We use a_i to denote the i^{th} row of A and use b_i to denote the i^{th} entry in b. Such linear programs are called packing programs because constraints (6.7) define the problem of packing a convex combination of vectors subject to "capacity" constraints b.

Let λ^* denote the minimum possible value of λ. The main result of [48] is that if a packing linear program satisfies certain technical conditions, then a solution with $\lambda \leq (1+\epsilon)\lambda^*$ can be found in polynomial time. Not all linear programs of the form above satisfy these technical conditions, but several important applications, including minimum-cost multicommodity flow, unrelated parallel machine scheduling and the Held-Karp lower bound for the traveling salesman problem do satisfy these conditions.

We now show that the multicommodity flow problem can be expressed as a packing linear program. Constraints (6.1) through (6.3) correspond to equation (6.6), where P is just the set of convex combinations of all flows satisfying flow conservation. Constraint (6.7) corresponds to inequality (6.4), the capacity constraints. The correspondence between equations (6.5) and (6.8) is straightforward.

Our multicommodity flow algorithm requires a minimum-cost flow subroutine for each iteration. The packing algorithm requires a subroutine OPT that

> Given an m-dimensional vector $y \geq 0$, finds $\tilde{x} \in P$ such that $c\tilde{x} = \min(cx : x \in P)$ where $c = y^T A$.

Again, if we let y be the edge lengths ℓ in the multicommodity flow algorithm, the subroutine OPT is just a minimum-cost flow algorithm.

We do not wish to spend much time on the general case. The reader is referred to [48] for a host of algorithms and applications. To understand how the algorithm works, we state the main routine, IMPROVE-PACKING which appears in Figure 6.1. Note the similarities with DECONGEST. In particular, λ, the maximum edge congestion is now $\max_i a_i x/b_i$, the maximum amount by which the constraints (6.7) are violated. The constants α and σ are chosen similarly to the way they are in DECONGEST, but σ depends on a new parameter $\rho = \max_i \max_{x \in P} a_i x/b_i$, the *width* of P relative to $Ax \leq b$. The algorithm computes a cost y_i for each constraint. It then calls a routine that finds a minimum-cost point subject to these costs. Finally, the new solution

IMPROVE-PACKING
$\lambda_0 \leftarrow \max_i a_i x/b_i$; $\alpha \leftarrow 4{\lambda_0}^{-1}\epsilon^{-1}\ln(2m\epsilon^{-1})$; $\sigma \leftarrow \frac{\epsilon}{4\alpha\rho}$.
while $\max_i a_i x/b_i \geq \lambda_0/2$ **and** x and y do not satisfy suitable relaxed optimality conditions
(1) For each $i = 1, \ldots, m$: set $y_i \leftarrow \frac{1}{b_i} e^{\alpha a_i x/b_i}$.
(2) Find a minimum-cost point $\tilde{x} \in P$ for costs $c = y^T A$.
(3) Update $x \leftarrow (1-\sigma)x + \sigma\tilde{x}$.
return x

Figure 6.1: Procedure IMPROVE-PACKING

is set equal to a convex combination of the old and minimum-cost solutions. Performing one iteration of this algorithm leads to a significant decrease in a potential function $\Phi = y^T b$. Given these similarities, one can see how much of the basic analysis follows along the same lines as that of the multicommodity flow algorithm.

We also discuss two extensions to the framework described above that will be needed in the next section. The first occurs when the polytope P can be expressed as a product of polytopes of smaller dimension, i.e., $P = P^1 \times \cdots \times P^k$. A vector x is now partitioned, in some way, into a series of vectors $x^1, \ldots, x^k$ and $x \in P$ if and only if $x^i \in P^i$, $i = 1 \ldots k$. The set of inequalities $Ax \leq b$ can be written as

$$\sum_i A^i x^i \leq b. \tag{6.9}$$

The multicommodity flow problem can be formulated in this way. To give an concrete example of this formulation, we explain the correspondence. With respect to the polytope P, the multicommodity flow variables are $f(vw)$, the total amount of flow on edge vw. We can decompose $P = P^1 \times \cdots \times P^k$ with P^i representing the polytope of all feasible flows for commodity i. As we have seen, each flow $f(vw)$ can be decomposed into flows of the individual commodities on each edge. Therefore, we have a series of vectors $f^1, \ldots, f^k$, where each vector has m components, one for each edge, representing $f_i(vw)$. The constraints (6.1) through (6.3) are already written individually for each commodity, and the capacity constraints, which sum the total flow of all commodities on each edge are of the form given in equation (6.9). The optimization routine is the same for each i, namely a minimum-cost flow routine for commodity i.

For the multicommodity flow problem, each component of the vector $x \in P$, say x_e, is partitioned into k parts, x_e^1 through x_e^k, such that for each $x^i \in P^i$ and for each e, $x_e = \sum_{i=1}^k x_e^i$. This particular partition of x is not unique. Assume that x has $r \cdot s$ components. It is possible to partition x into r length-s vectors such that $x^i = (x_1^i, \ldots x_s^i) = (x_{i\cdot(s-1)+1}, \ldots, x_{i\cdot s})$. We use the latter partitioning on the job scheduling problem.

Other definitions for the product of polytope representation follow. In particular, we use a_j^i to denote the j^{th} row of A^i. Let ρ^i denote the width of P^i relative to $A^i x^i \leq b$, $i = 1, \ldots k$. Observe that $\rho = \sum_i \rho^i$. Instead of a subroutine OPT, we have a series of k subroutines. The i^{th} subroutine minimizes cx^i subject to $x^i \in P^i$ for costs $c = y^t A^i$.

In [48], the main motivation for introducing this formulation is to allow the use of randomness. Where algorithm Concurrent randomly chooses a commodity i to reroute, Plotkin, Shmoys and Tardos randomly choose a polytope P^i over which to optimize. For the purposes of shop scheduling, we are concerned only with a deterministic algorithm, and so the main reason for introducing this formulation is to simplify our algorithm. By separating out the different polytopes, an iteration of Improve-Packing can be applied to one job at a time.

The other extension we need to make deals with integral solutions. In Problem 6.1.1, we need to choose *integral delays* for each job. The procedure Improve-Packing makes no attempt to maintain an integral solution. However, Improve-Packing can be modified to obtain an integral solution. We proceed to outline this modification, which is similar to the one used in Section 3.2.

First, we need to maintain that the point $\tilde{x}$ returned by the optimization routine is integral. We make use of the fact that the optimization routines of interest have the property that there always exists an optimal integral solution. Hence, without loss of generality, we can restrict the search to integral solutions. For multicommodity flow, the optimization routine has integral solutions, since it is well-known that a minimum-cost flow problem with integral data always has an optimal integral solution. As we shall see, for shop scheduling, it is also true that the optimization routine always has an optimal integral solution. Second, we have to ensure that $(1-\sigma)x + \sigma\tilde{x}$ is integral, which is accomplished by maintaining that all components x_i are integral multiples of the current value of σ. The modifications needed in the analysis are similar to those needed in Section 3.2. In particular, it is still possible to show that even with the restriction to integral solutions, a potential function $\Phi = y^T b$ decreases in each iteration, and the running time remains that same as that of the non-integral version, up to constant factors. In addition, by analysis similar to that used in Theorem 3.2.1, we can obtain the following theorem:

Theorem 6.2.1 (Plotkin, Shmoys, Tardos[48]) Let $\overline{\rho} = \max_i \rho^i$, and let $\overline{\lambda} = \max\{\lambda^*, (\overline{\rho}/d)\log M\}$, where M is the number of packing constraints and d is a parameter such that each component of each x^i is comprised of integral multiples of d. There exists an integral solution to $\sum_i A^i x^i \leq \lambda b$ with $x^i \in P^i$ for $i = 1, \ldots, k$ and $\lambda \leq \lambda^* + O(\sqrt{\overline{\lambda}(\overline{\rho}/d)\log(Mkd)})$. Repeated calls to the deterministic integral version of IMPROVE-PACKING find such a solution $(\overline{x}, \overline{\lambda})$ using $O(d\rho/\overline{\rho} + \rho\log(M)/\overline{\lambda} + k\log(d\rho/\overline{\rho}))$ calls to each of the k subroutines. Further, throughout the execution, $\epsilon = \Omega(\sqrt{\overline{\rho}\log(Mkd)/(d\lambda^*)})$.

For the remainder of this chapter, we focus specifically on the application to shop scheduling and quote results from [48] as needed.

6.3 The Solution

We now turn to the solution of Problem 6.1.1. Since we introduce initial delays in the range $[0, \Pi_{\max}/\log(m\mu)]$, the resulting schedule has length $\ell = P_{\max} + \Pi_{\max}/\log(m\mu)$. We can represent the processing of a job J_j with a particular initial delay d by an $(\ell \cdot m)$-length $\{0,1\}$-vector where each position corresponds to a machine at a particular time. The position corresponding to machine m_i and time t is 1 if m_i is processing job J_j at time t, and 0 otherwise. For each job J_j and each possible delay d, there is a vector V_{jd} that corresponds to assigning delay d to J_j.

Let π_j be the set of vectors $\{V_{j1}, \ldots, V_{jd_{\max}}\}$, where $d_{\max} = \Pi_{\max}/\log(m\mu)$, and let $V_{jk}(i)$ be the i^{th} component of V_{jk}. Given the set $\Lambda = \{\pi_1, \ldots, \pi_n\}$ of sets of vectors, Problem 6.1.1 can be stated as the problem of choosing one vector from each π_j (denoted V_j^*), such that

$$\left\| \sum_{j=1}^{n} V_j^* \right\|_\infty = O(\log(m\mu)).$$

In words, we wish to ensure that at any time on any machine, the number of jobs using that machine is $O(\log(m\mu))$.

We can reformulate this problem as a $\{0,1\}$–integer program. Let x_d^j be the indicator variable used to indicate whether V_{jd} is selected from π_j. The vector x has length $n \cdot d_{\max}$ where the i^{th} entry is denoted $x_{((i+1)\bmod d_{\max})-1}^{\lceil i/d_{\max}\rceil}$. Thus the problem of assigning delays to the jobs so as to minimize congestion can be phrased as the following integer program (ICONJ):

minimize λ

subject to

$$\sum_{d=1}^{d_{\max}} x_d^j = 1, \qquad j = 1,\ldots,n; \tag{6.10}$$

$$\sum_{j=1}^{n}\sum_{d=1}^{d_{\max}} V_{jd}(i)x_d^j \leq \lambda\log(m\mu), \qquad i = 1,\ldots,\ell\cdot m; \tag{6.11}$$

$$x \in \{0,1\}^{n\times d_{\max}}. \tag{6.12}$$

Note that we put $\log(m\mu)$ on the righthand side because we know, by Lemma 5.2.1 that there exists a solution where the maximum number of jobs on any machine is $O(\log(m\mu))$. Thus $\lambda^* = O(1)$. However, λ^* can be as small as $1/\log(m\mu)$. While our algorithm may find such a solution, the best that we can show that it always finds a solution with $\lambda = O(1)$.

The linear programming relaxation of ICONJ is just equations (6.10), (6.11) and the constraints that $x \geq 0$. Note that the constraints $x \leq 1$ would be redundant, given (6.10). In order to show that the linear programming relaxation of ICONJ is a packing linear program we will show how to express it in the form given in the definition of a packing linear program.

Constraint (6.11) is clearly in the form $Ax \leq \lambda b$ where b is a vector in which each component is equal to $\log(m\mu)$ and each column of A correspond to one vector V_{jk}. Next we consider the constraints (6.10). First, we see that these can be decomposed into n different constraints, one for each job J_j. Thus the polytope P, defining all constraints (6.10) can be decomposed into n polytopes $P = P^1 \times \cdots \times P^n$, where polytope P^j corresponds to job J_j. Now we focus on a particular polytope $P^{j'}$. The constraint $\sum_{d=1}^{d_{\max}} x_d^{j'} = 1$ just says that for all vectors $x^{j'}$, $x^{j'} \in P^{j'}$ must lie in the $d_{\max}$-dimensional unit simplex. Each vertex of simplex $P^{j'}$ corresponds to choosing a particular delay for job j'.

With the problem phrased in these terms, we can now use Theorem 6.2.1 to bound the number of iterations of an integral version of IMPROVE-PACKING needs to find a solution to ICONJ with $\lambda = O(1)$. We will deal with the implementation of an iteration later. Note that this is a typical use of the framework of Plotkin, Shmoys and Tardos; one bounds the number of iterations in a fairly standard manner but then must implement one iteration in a problem-specific fashion.

Lemma 6.3.1 An integral solution to ICONJ can be found such that $\lambda = O(1)$, using $O(n \log(m\mu))$ iterations of a deterministic integral version of IMPROVE-PACKING. Further throughout the execution, $\epsilon = \Omega(1)$.

Proof: We must compute the particular values of the various parameters in Theorem 6.2.1. The width

$$\overline{\rho} = \max_j \rho^j = \max_j \max_i \max_{x \in P^j} (a_i^j x / b_i^j).$$

From constraints (6.10), $a_i^j x_i^j \leq 1$ for all i, j and $b_i^j = \log(m\mu)$ for all i, j, and therefore $\overline{\rho} = 1/\log(m\mu)$. The polytope P is the crossproduct of n identical polytopes, and hence $\rho = n\overline{\rho}$. The number polytopes $k = n$ and $d = 1$, since all variables are integral. Next, we bound the number of constraints. There is one constraint for every possible time unit for every possible job. There are m jobs and $P_{\max} + \Pi_{\max}/\log(m\mu)$ possible time units. Hence

$$M = m(P_{\max} + \Pi_{\max}/\log(m\mu)). \tag{6.13}$$

From Lemma 5.3.1 we know that the maximum operation size $p_{\max} = O(n\mu)$ and from Lemma 5.3.2, the number of jobs $n = O(m^2\mu^3)$. The maximum machine load, $\Pi_{\max}$, is maximized if all operations are of length $p_{\max}$ and fall on one particular machine, thus $\Pi_{\max} \leq n\mu p_{\max} = O(n^2\mu^2)$. The maximum job length, $P_{\max}$, is at most $\mu p_{\max}$, so $P_{\max} = O(n\mu^2)$. Putting these bounds together, we obtain

$$\begin{aligned} M = m(P_{\max} + \Pi_{\max}/\log(m\mu)) &= O\left(m\left(n\mu^2 + \frac{n^2\mu^2}{\log(m\mu)}\right)\right) \\ &= O\left(m\left(m^2\mu^5 + \frac{m^4\mu^8}{\log(m\mu)}\right)\right) = O(m^5\mu^8). \end{aligned}$$

We will use below that $\log M = O(\log(m\mu))$. Now we turn to the quality of the solution.

$$\lambda' = \max(\lambda^*, (\overline{\rho}/d)\log M) = \max(O(1), \frac{\log M}{\log(m\mu)}) = O(1).$$

In addition, since we know by Lemma 5.2.1 that there exists a solution with $\lambda = O(1)$, $\lambda^* = O(1)$. Thus

$$\begin{aligned}
\lambda &\leq \lambda^* + O(\sqrt{\lambda'(\overline{\rho}/d)\log(Mkd)}) \\
&= O(1) + O\left(\sqrt{O(1)\left(\frac{1}{\log(m\mu)}\right)\log(n^2\mu^2m^2\mu^3)}\right) \\
&= O(1).
\end{aligned}$$

For the running time we need

$$\begin{aligned}
&O(d\rho/\overline{\rho} + \rho\log(M)/\overline{\lambda} + k\log(d\rho/\overline{\rho})) \\
&\qquad = O(n + \frac{n\log M}{\log(m\mu)} + n\log n) \\
&\qquad = O(n\log n) \\
&\qquad = O(n\log(m\mu))
\end{aligned}$$

calls.

Finally, the error $\epsilon = \Omega(\sqrt{\overline{\rho}}\log(Mkd)/d\lambda^*) = \Omega(1)$. ■

We now turn to the time needed to implement one iteration. We will show that an iteration can be implemented efficiently in the RAM model of computation. While it appears that the algorithms in [48] can be implemented in the RAM model using techniques similar to those used in Section 2.4.2 of this thesis, the computation is not explicitly done in [48]. Here we perform the necessary computations for the case of program ICONJ. The first step of IMPROVE-PACKING computes the costs $y_i = \frac{1}{b_i}e^{\alpha a_i x/b_i}$. As before, the difficulty here is that we have to compute exponential functions. In order to have an efficient algorithm, we must bound the precision needed in our computation.

We now show that we need only $O(\log(m\mu))$ bits of precision for each component of y. We use the following theorem of [48], which is similar to Lemma 2.4.24.

Theorem 6.3.2 [48] Let $C_p(y)$ be the value returned by OPT, the minimization subroutine. If we replace OPT by an algorithm that finds a point $\tilde{x}$ such that

$$y^T A\tilde{x} \leq (1+\frac{\epsilon}{2})C_p(y) + \frac{\epsilon}{2}\lambda y^T b \tag{6.14}$$

for any $y \geq 0$, then all the bounds on the running time of the algorithm still hold. In particular, a potential function $\Phi = y^T b$ still decreases by the same amount, up to constant factors.

While this theorem stipulates that an approximate $\tilde{x}$ is sufficient, it does not explain how to find one efficiently. We now show that such an $\tilde{x}$ can be found. The idea is to compute an approximate set of dual variables $\tilde{y}$, such that exact optimization with respect to $\tilde{y}$ produces an $\tilde{x}$ satisfying inequality (6.14).

The approximate dual variables $\tilde{y}$ are integral and consist of $O(\log(m\mu))$ bits per edge. We introduce two parameters γ, an amount by which each approximate dual variable is scaled, and ζ, the number of bits of accuracy in each approximate dual variable. We set $\gamma = \epsilon \cdot e^{\alpha\lambda}/(n\mu p_{\max})$, and will compute approximate dual variables $\tilde{y}$ so that $\gamma\tilde{y} \leq y$ and each component of $\tilde{y}$ can be represented in ζ bits. It will take $O(\log(m\mu))$ time to compute one component of $\tilde{y}$.

For each component y_i, first we compute $e^{\alpha(a_i x/b_i - \lambda)}$ approximately so that it has at most $\zeta = \epsilon/(4\mu p_{\max})$ additive error, then we multiply the result by ζ^{-1}, take the integer part, and set $\tilde{y}_i$ to be this value. Using the Taylor series, we can compute one bit of e^x in $O(1)$ time. Since $e^{\alpha(a_i x/b_i - \lambda)}$ is at most 1 on each edge, it is sufficient to compute $O(\log(1/\zeta))$ bits to achieve the desired approximation. Therefore each $\tilde{y}_i$ can be computed in $O(\log(1/\zeta)) = O(\log(m\mu/\epsilon))$ time. From Theorem 6.2.1, we know that $\epsilon = \Omega(1)$, hence $O(\log(m\mu/\epsilon))$ is just $O(\log(m\mu))$.

Because of the approximation and the integer rounding, the vector $\tilde{x}$, which is of minimum cost with respect to $\tilde{y}$, is not necessarily minimum-cost with respect to y. However, we now show that an $\tilde{x}$ that is minimum-cost with respect to $\tilde{y}$ satisfies conditions (6.14).

Lemma 6.3.3 Let $\tilde{y}$ be a set of dual variables computed as above. Then a point $\tilde{x} \in P^j$ minimizing $\tilde{y}^T A^j \tilde{x}$ satisfies inequality (6.14), where $C_p(y)$ is computed with respect to exact dual variables y. Further, each $\tilde{y}_i$ can be represented in $O(\log(m\mu))$ bits.

Proof: The idea is to show that if we minimize with respect to the approximate dual variables

$\tilde{y}$, the resulting solution, $\tilde{x}$, is "close" to the true minimum solution x. We will first bound the maximum possible difference between any component of y and the corresponding component of $\tilde{y}$. We will then translate this into a bound on the difference between $\tilde{c}^j = \tilde{y}^T A^j$ and $c^j = y^T A^j$. Finally, we will show that the both x and $\tilde{x}$ have a very special structure: they have exactly one component equal to 1 and the rest equal to 0. This will allow us to bound the difference between $y^T A\tilde{x}$ and $C_p(y)$ and hence prove the lemma.

We now proceed with the details. Recall that $\gamma = e^{\alpha\lambda}\zeta/\log(m\mu)$ and $\zeta = \epsilon/(4\mu p_{\max})$. We bound the difference between y and $\gamma\tilde{y}$, the approximate dual variables scaled to have the same units as y. In computing $\gamma\tilde{y}$, we introduce errors in several places. When computing $e^{\alpha(a_i x/b_i - \lambda)}$ to a precision of ζ, we introduce an error of at most ζ. This error is multiplied by by ζ^{-1} and may be increased by 1 when we round $\tilde{y}$ down to an integer. Finally, when we scale $\tilde{y}$ to have the same units as y, the entire error gets scaled by γ. Thus,

$$\begin{aligned} y_i - \gamma\tilde{y}_i &\leq \gamma\left(\zeta\left(\zeta^{-1}\right) + 1\right) \\ &= 2\gamma. \end{aligned} \tag{6.15}$$

Now consider $\tilde{c}^j = \tilde{y}^T A^j$. Each column of A^j corresponds to a vector V_{jd} for some d and hence is a vector of length $\ell \cdot m$ that has at most $\mu p_{\max}$ ones. Thus each entry of c^j is the sum of at most $\mu p_{\max}$ entries of $\tilde{y}$. By inequality (6.15), we know that $\gamma\tilde{y}_i$ differs from y_i by at most 2γ. Hence the difference between an entry of $\gamma\tilde{c}^j$, say $\gamma\tilde{c}^j_r$, and the corresponding entry of $c^j = y^T A^j$ is at most $2\gamma\mu p_{\max}$. Consider the problem of finding an $\tilde{x} \in P^j$ that minimizes $\tilde{c}^j x$. (The case for $c^j x$ is identical.) The vector $\tilde{c}^j$ is non-negative. Let $\tilde{c}^j_z$ be the component of $\tilde{c}^j$ with minimum value. Recall that constraints (6.10) require that $\sum_{d=1}^{d_{\max}} x^j_d = 1$. Then it is easy to see that the $\tilde{x}^j \in P^j$ that minimizes $\tilde{c}^j \tilde{x}^j$ has $x^j_z = 1$ and all other components of x^j equal to 0. This setting is the vector $\tilde{x}$. Thus for the $\tilde{x}$ that minimizes $\tilde{c}^j x^j$, we have that

$$\begin{aligned} c^j\tilde{x} - \gamma\tilde{c}^j\tilde{x} &= (c^j - \gamma\tilde{c}^j)\tilde{x} \\ &= c^j_z - \gamma\tilde{c}^j_z \\ &\leq 2\gamma\mu p_{\max}, \end{aligned} \tag{6.16}$$

where the last inequality follows from the discussion above. But

$$2\gamma\mu p_{\max} = \frac{2e^{\alpha\lambda}\epsilon\mu p_{\max}}{4\mu p_{\max}\log(m\mu)} = \frac{\epsilon}{2}\frac{e^{\alpha\lambda}}{\log(m\mu)}. \tag{6.17}$$

We know that $e^{\alpha\lambda} \le y^T b$, since at least one component of y is equal to $\frac{e^{\alpha\lambda}}{\log(m\mu)}$, all components are non-negative and $b_i = \log(m\mu)\ \forall i$. Also, since some job must execute on some machine, $\lambda \ge \frac{1}{\log(m\mu)}$. Combining inequalities (6.16) and (6.17) with these two bounds we get that

$$c^j\tilde{x} - \gamma\tilde{c}^j\tilde{x} \le \frac{\epsilon}{2}\lambda y^t b. \tag{6.18}$$

But $\gamma\tilde{c}^j\tilde{x}$ just selects the minimum element of $\tilde{c}^j$. Since componentwise, $\gamma\tilde{c}^j \le c^j$, we know that $\gamma\tilde{c}^j\tilde{x} \le C_P(y)$, which together with inequality (6.18) implies inequality (6.14). ■

So, without affecting the performance of our algorithm, we can use approximate costs. Note that we have also shown that the subroutine OPT applied to this problem always has an optimal solution that is integral. Further, we have a nice combinatorial characterization of this routine, since it reduces to finding the minimum of a set of numbers. Yet, in order to have an efficient algorithm, more work needs to be done. Since the number of entries in the matrix A and the vectors y and b are extremely large polynomials in m and μ, we would like to avoid using straightforward matrix-vector and vector-vector multiplications, since they take too much time. We can obtain a more efficient algorithm by taking advantage of the structure of the problem and noticing that between two iterations, not too many variables change.

In the remainder of this chapter, we will first show that in each iteration of Improve-Packing, a small number of the components of y change. We will then show that if a small number of the components of y change then a small number of the components of c change. Further, we will show how to compute an entry of c in less time than the naive method of multiplying y^T by a column in A. We will then show, that by using a heap data structure, we can efficiently compute $\min(y^T A^j x^j : x^j \in P^j)$ and thus be able to conclude that an iteration of Improve-Packing can be implemented efficiently.

Lemma 6.3.4 In each iteration of Improve-Packing, $O(n\mu)$ components of y change. Further, these changes can be computed in $O(n\mu\log(m\mu))$ time.

Proof: Consider the evaluation of the statement $x^j \leftarrow (1-\sigma)x^j + \tilde{x}^j$. Since we maintain that $\sigma = 1$ throughout, this reduces to $x^j \leftarrow \tilde{x}^j$. In other words, we take job J_j and change its assignment delay from some value d' to some other value d'', set $x^j_{d'} = 0$ and $x^j_{d''} = 1$. By changing the value of one variable x^j_d, we affect the value of at most $P_{\max}$ dual variables, because setting $x^j_d = 1$ implies that job j runs for at most $P_{\max}$ time units starting at time d. Since each y_i corresponds to a particular machine at a particular time, only the $P_{\max}$ components of y that correspond to machine-time pairs in the schedule implied by x^j_d are affected. Thus we must recompute at most $2P_{\max}$ components of y.

Once we know which elements to compute, Lemma 6.3.3 tells us that each one can be computed in $O(\log(m\mu))$ time. To identify the components of y to recompute, we simulate the execution of job J_j with delay d. We simply walk through the corresponding schedule for that job and for each machine-time pair that the job uses, and update the corresponding dual variable.

■

We now turn to the second and third steps of Improve-Packing. Step 2 involves finding a minimum-cost point $x^j \in P^j$ for costs $c^j = y^T A^j \;\; \forall j = 1, \ldots, n$ and step 3 involves updating the solution.

We first show that in an iteration, not too many of the components of c change. As we saw in Lemma 6.3.4, the only components of y that change are those associated with machines running in the time intervals $(d', d' + P_{\max} - 1)$ and $(d'', d'' + P_{\max} - 1)$. Each component of c, c^j_d, is associated with starting job J_j at time d then running for up to $P_{\max}$ units. In fact, c^j_d is equal to the sum of the components of y associated with the machine-time pairs on which job J_j is active. Since all the components of y that changed are in the two intervals $(d', d' + P_{\max} - 1)$ and $(d'', d'' + P_{\max} - 1)$, the only components of c^j that can change must be associated with jobs running in those intervals. But the only jobs can be running in those intervals are those that receive initial delays in the range $(d' - P_{\max}, d' + P_{\max} - 1)$ or the range $(d'' - P_{\max}, d'' + P_{\max} - 1)$. There is one component of c^j associated with each possible delay, and hence for a particular job J_j, there are a total of at most $4P_{\max}$ changes overall. Summing over all the jobs, we get a total of $4nP_{\max}$ possible changes.

We could recompute these $4nP_{\max}$ components of c by taking the dot product of two length

$d_{\max}$ vectors. However, we can perform this computation more efficiently.

Lemma 6.3.5 Assume that we have computed the new values of the components of y as in Lemma 6.3.4. Then the correct values of $c = c^1, \ldots, c^n$ can be computed in $O(n^2\mu^3)$ time.

Proof: Assume that we have already computed c^j_d and we wish to compute c^j_{d+1}. Recall the definition of V_{jd}. It has a component for every time on every machine and is one when the component associated with a particular machine-time pair is busy. We can express

$$c^j_d = y^T A^j_d = y^T V_{jd}$$

and

$$c^j_{d+1} = y^T A^j_{d+1} = y^T V_{j(d+1)}.$$

Additionally $V_{j(d+1)}$ corresponds to running the operations of job J_j on the same sequence of machines as V_{jd}, only one time unit later. Therefore, the two vectors V_{jd} and $V_{j(d+1)}$ differ in at most 2μ positions. Hence, c^j_{d+1} can be computed from c^j_d using $O(\mu)$ additions, assuming we know in which components c^j_d and c^j_{d+1} differ. Since the matrix A is fixed throughout the algorithm, we can use a preprocessing phase to construct a series of lists X^j_d, one for each c^j_d, $j = 1, \ldots, n$, $d = 2, \ldots, d_{\max}$. Each X^j_d contains a list of the up to 2μ positions in which V_{jd} differs from $V_{j(d-1)}$, along with an indication of whether the difference is that $V_{jd} = 1$ and $V_{j(d-1)} = 0$ or vice versa. There is no predecessor of c^j_1, but we can precompute X^j_1 $j = 1, \ldots, n$, the list of positions in which $V_{j1} = 1$, to speed up the computation of c^j_1.

Thus for each j, we may have to compute c^j_1, which takes $O(\mu p_{\max})$ time, and then $4P_{\max}$ more values, each of which take $O(\mu)$ time. Summing over all jobs, we get a total time of $O(n(\mu p_{\max} + \mu P_{\max})) = O(n\mu P_{\max}) = O(n^2\mu^3)$. ■

As mentioned before, given a vector c^j, the problem of finding an $\tilde{x}$ that minimizes $c^j\tilde{x}$ for a job J_j consists of choosing the minimum component of c^j. Thus, for each job J_j, we maintain a heap H^j consisting of the $d_{\max}$ values of $c^j_d, d = 1, \ldots, n$. We also maintain a list $\mathcal{L}^j$, where the i^{th} component of $\mathcal{L}^j$, $\mathcal{L}^j_i$, contains a pointer to the position that c^j_i occupies in heap H^j. This data structure allows us to insert an element, delete an arbitrary element and find the minimum value, all in $O(\log d_{\max}) = O(\log(m\mu))$ time. Thus we can minimize $c^j\tilde{x}$ in $O(\log(m\mu))$ time

1. Let $x^j_{d'}$ and $x^j_{d''}$ be the variables changed in the previous iteration.
2. Compute y_i as described in Lemma 6.3.4.
3. Let $\mathcal{D} = (d' - P_{\max}, d' + P_{\max} - 1) \cup (d'' - P_{\max}, d'' + P_{\max} - 1)$ the set of possible initial delays affected.
4. Recompute c^j_d $j = 1, \ldots, n$; $\forall d \in \mathcal{D}$, making the appropriate heap changes.
5. Compute the minimum value in each heap. Let the minimum value in H_j be $c^j_{i_j}$ $j = 1, \ldots, n$.
6. Compute the decrease in $\Phi = y^T b$ associated with replacing the current assignment for job J_j with a minimum-cost assignment, $j = 1, \ldots, n$.
7. Let j' be the job that maximizes the decrease in Φ. Let $\tilde{x}$ be the vector with a 1 in position $i_{j'}$ and 0 elsewhere. Let $x^{j'} \leftarrow \tilde{x}$.

Figure 6.2: One iteration of the algorithm

by choosing the minimum element out of the heap.

Of course, we must update the heap when costs change. Each change can be implemented as a delete followed by an insert. However, as we saw in Lemma 6.3.5, there are $O(P_{\max})$ changes in the costs for each job for a total of $O(nP_{\max})$ heap operations.

We can now restate the algorithm for one iteration of IMPROVE-PACKING, specialized to Problem 6.1.1. This algorithm appears in Figure 6.2 and is a summary and formalization of the ideas presented in this section. Note that, in spite of the phrasing of the problem as a linear program, at no point do we have to perform any matrix operations.

Lemma 6.3.6 The algorithm in Figure 6.2 executes an iteration of IMPROVE-PACKING in $O(n^2\mu^2(\mu + \log(m\mu)))$ time.

Proof: The correctness follows from Lemma 6.3.3 and the discussion of this section. We proceed to bound the running time. By Lemma 6.3.4, Step 2 takes $O(n\mu \log(m\mu))$ time. By Lemma 6.3.5, updating c takes $O(n^2\mu^3)$ time and requires $O(nP_{\max})$ heap operations that take $O(n^2\mu^2 \log(m\mu))$ time. Step 5 requires one operation in each of n heaps, for a total of $O(n \log(m\mu))$ time. For Step 6, we need, for each job, to compute the new Φ that would occur after reassigning that job. Since $\Phi = y^T b = (y^T 1) \log(m\mu)$, Φ can be computed by summing all the components of y. To compute the change associated with one job , we simulate Steps

1 and 2 of the algorithm and then compare the sum of the values of the $O(P_{\max})$ components of y that have changed. We can perform this computation in $O(n\mu\log(m\mu))$ time per job and $O(n^2\mu\log(m\mu))$ time overall. Step 7 can be performed in $O(n)$ time. Putting these steps together yields the bound of the lemma. ■

Combining this lemma with Lemma 6.3.1 yields the following theorem:

Theorem 6.3.7 Problem 6.1.1 can be solved deterministically in $O(n^3\mu^2\log(m\mu)(\mu+\log(m\mu)))$ time.

Finally, combining this theorem with the results of Chapter 5 we get the following theorem. The other bottleneck in the shop scheduling algorithm is the algorithm of Sevast'yanov [56] that takes $O((\mu mn)^2)$ time.

Theorem 6.3.8 There exists a deterministic algorithm for job shop scheduling that finds a schedule of length $O(\log^2(m\mu)\,C^*_{\max})$ in $O(n^2m^2\mu^2 + n^3\mu^2\log(m\mu)(\mu+\log(m\mu)))$ time.

The running time in Theorem 6.3.8 compares quite favorably with the previous best bound for this problem. The previous bounds involved using the linear programming algorithm of Vaidya [67], which takes $O(n^{10.5}\mu^7\log(m\mu))$ for this problem, combined with a deterministic version of the randomized rounding of Raghavan and Thompson [52] and Raghavan [50].

Glossary of Notation

A glossary of nation, containing symbols that are used frequently, follows. It is divided into three sections. The first contains notation used in Chapters 2, 3 and 4 to describe multicommodity flows. The second contains notation used in Chapters 5 and 6 to describe shop scheduling. The third contains notation used only in Chapter 6. Each section is alphabetized, with all Roman letters preceding all Greek letters.

Chapters 2, 3 and 4

Symbol	Meaning
C_i	the cost of the flow for commodity i
$C_i^*, C_i^*(\lambda)$	the cost of the minimum-cost flow for commodity i
$c(vw)$	the cost of edge vw
D	the sum of demands
D_i	$\max_v \left\{ \lvert \hat{d}_i(v) \rvert \right\}$
d_i	the demand for commodity i
$\hat{d}_i(v_j)$	the demand for commodity i at node v_j
$d(A, \bar{A})$	the demand for commodities with one endpoint in A and the other in $\bar{A}$
$\mathrm{dist}_\ell(v, w)$	the length of the shortest path from v to w with respect to ℓ
$d_{\max}$	the maximum demand
$\mathcal{F}$	an instance of a feasible flow problem
$\bar{f}_i^*$	an approximately minimum-cost flow for commodity i
$\tilde{f}_i^*$	an approximately minimum-cost flow for commodity i that can be represented in $O(\log(nU))$ bits

$f(vw)$	the flow on edge vw
$f_i(P)$	the flow of commodity i on path P
$f_i(vw)$	the flow of commodity i on edge vw
$G = (V, E)$	a graph with vertex set V and edge set E
$G_f = (V, E_f)$	the residual graph with vertex set V and edge set E_f
$\mathcal{I}$	an instance of a multicommodity flow problem
I_P	the number of productive iterations during a call to DECONGEST
I_U	the number of unproductive iterations during a call to DECONGEST
$\mathcal{K}$	a specification of k commodities
k	number of commodities
k^*	the number of distinct sources
$\ell(vw)$	the length of edge vw
$\mathcal{M}$	an instance of a minimum-cost flow problem
m	number of edges in the network
$\mathcal{N}$	an instance of a maximum flow problem
n	number of nodes in the network
P	a path
$\mathcal{P}_i$	a collection of paths from s_i to t_i
$p(v)$	the price of node v
s_i	the source of commodity i
T_{MCF}	the time to find a minimum-cost flow
T_P	the time taken by one productive iteration during a call to DECONGEST
T_U	the time taken by one unproductive iteration during a call to DECONGEST
t_i	the sink of commodity i
U	maximum capacity
$u(vw)$	the capacity of edge vw
z	a percentage to scale demands by
α	a parameter in the exponent of the value of ℓ
$\Gamma(A)$	the set of edges with exactly one endpoint in node set A, the cut associated with A.
γ	an amount by which each approximate length is scaled

ϵ	an error parameter; a measure of solution accuracy
λ	the maximum edge congestion; congestion
λ^*	the minimum possible value of λ
λ_0	the congestion at the start of procedure DECONGEST
Φ	the potential function
Φ^i	the value of the potential function at the beginning of iteration i of DECONGEST
σ	the fraction of flow to reroute
σ_i	the fraction of flow of commodity i to reroute
τ	a target value of λ
ζ	the number of bits of accuracy in each approximate length

Chapters 5 and 6

Symbol	Meaning
C_{ij}	the completion time of operation O_{ij}
$C_{\max}$	$\max_{i,j} C_{ij}$
$C^*_{\max}$	the minimum possible completion time
$\mathcal{J} = \{J_1, \ldots, J_n\}$	the set of jobs
$\mathcal{M} = \{m_1, m_2, \ldots, m_m\}$	the set of machines
m	the number of machines
n	the number of jobs
$\mathcal{O} = \{O_{ij} \mid i = 1, \ldots, \mu_j, j = 1, \ldots, n\}$	the set of operations
$P_{\max}$	the maximum job length
p_{ij}	the processing time of the i^{th} operation of job J_j
${p'}_{ij}$	p_{ij} rounded up to the next power of 2
r_j	the release date of job J_j
S_i	a set of identical machines
ϵ	an error parameter
κ_{ij}	the machine on which operation O_{ij} runs

μ	the maximum number of operations in any job
μ_j	the number of operations of job J_j
ω	the total number of operations
$\Pi_{\max}$	the maximum machine load

Chapter 6

Symbol	Meaning
A	a $p \times q$ non-negative matrix
a_i	the i^{th} row of A
b_i	i^{th} entry in b
$C_p(y)$	a minimum cost point subject to costs y
c	costs, $c = y^T A$
$\tilde{c}^j$	approximate costs, $\tilde{c}^j = \tilde{y}^T A^j$
d	a value that each component of x^i is an integral multiple of
d', d''	delay values
H^j	a heap associated with job J_j
$\mathcal{L}^j$	pointers to values in H^j
M	the number of packing constraints
P	a polytope, a convex set in $\mathbf{R}_+^q$
V_{jd}	the vector associated with assigning delay d to job J_j
X_d^j	a precomputed list to speed the computation of c_d^j
$\tilde{x}$	an approximate solution to OPT
x^i	a point in polytope P^i
y	dual variables
γ	an amount by which each approximate dual variable is scaled
λ	$\max_i a_i x / b_i$
Φ	a potential function
ρ	$\max_i \max_{x \in P} a_i x / b_i$, the *width* of P relative to $Ax \leq b$

ρ^i	the width of P^i relative to $A^i x^i \leq b$
ζ	the number of bits of accuracy in each approximate dual variable

Bibliography

[1] I. Adler, N. Karmarkar, M. Resende, and G. Veiga. An implementation of Karmarkar's algorithm for linear programming. *Mathematical Programming*, 44:297–335, 1989.

[2] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. Finding Minimum-Cost Flows by Double Scaling. *Mathematical Programming*, 53:243–266, 1992.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows. In G.L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Handbook in operations research and management science, Volume 1: Optimization*, pages 211–360. North-Holland, Amsterdam, 1990.

[4] D. Applegate and B. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal of Computing*, 3:149–156, 1991.

[5] A. A. Assad. Multicommodity network flows - a survey. *Networks*, 8:37–91, 1978.

[6] I. Bárány and T. Fiala. Többgépes ütemezési problémák közel optimális megoldása. *Szigma–Mat.–Közgazdasági Folyóirat*, 15:177–191, 1982.

[7] I.S. Belov and Ya. N. Stolin. An algorithm in a single path operations scheduling problem. In *Mathematical Economics and Functional Analysis [In Russian]*, pages 248–257. Nauka, Moscow, 1974.

[8] D. P. Bertsekas and P. Tseng. RELAXT-III: A new and improved version of the RELAX code. Technical Report LIDS–P–1990, MIT, July 1990.

[9] S. N. Bhatt and F. T. Leighton. A framework for solving VLSI graph layour problems. *Journal of Computer and System Sciences*, 28, 1984.

[10] J. Cheriyan, October 1991. Private communication.

[11] V. Chvátal. *Linear Programming.* W. H. Freeman and Company, 1983.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[13] R. Dial. Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM*, 12:632–633, 1969.

[14] P. Elias, A. Feinstein, and C. E. Shannon. Note on maximum flow through a network. *IRE Transactions on Information Theory IT-2*, pages 117–199, 1956.

[15] T. Fiala. Közelítő algorithmus a három gép problémára. *Alkalmazott Matematikai Lapok*, 3:389–398, 1977.

[16] T. Fiala. An algorithm for the open-shop problem. *Mathematics of Operations Research*, 8(1):100–109, 1983.

[17] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canad. J. Math.*, 8:399–404, 1956.

[18] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[19] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18:1013–1036, 1989.

[20] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[21] A. V. Goldberg. A Natural Randomization Strategy for Multicommodity Flow and Related Algorithms. *Information Processing Letters*, 42:249–256, 1992.

[22] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *Journal of the ACM*, 36:873–886, 1898.

[23] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[24] D. Goldfarb and M. Grigoriadis. A computational comparison of the Dinic and Network Simplex methods for maximum flow. *Annals of Operations Research*, 13:83–123, 1988.

[25] T. Gonzalez and S. Sahni. Open shop scheduling to minimize finish time. *Journal of the ACM*, 23:665–679, 1976.

[26] T. Gonzalez and S. Sahni. Flowshop and jobshop schedules: complexity and approximation. *Operations Research*, 26:36–52, 1978.

[27] M. D. Grigoriadis and L. G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM Journal on Optimization*, 4(1):86–107, February 1994.

[28] M. D. Hansen. Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 604–610. IEEE, October 1989.

[29] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, pages 61–68, 1954.

[30] S. Kapoor and P. M. Vaidya. Fast algorithms for convex quadratic programming and multicommodity flows. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 147–159, 1986.

[31] J. Kennington. A survey of linear cost multicommodity network flows. *Operations Research*, 26:206–236, 1978.

[32] J. Kennington. A primal partitioning code for solving multicommodity flow problems (version 1). Technical Report Techincal Report 79009, Deptartment of Industrial Engineering and Operations Research, Southern Methodist University, 1979.

[33] 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.

[34] P. Klein, S. Kang, and J. Borger. Approximating concurrent flow with uniform demands and capacities: an implementation. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science: Volume 12, Network Flows and Matching*, October 1991. To appear.

[35] P. Klein, S. A. Plotkin, C. Stein, and É. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, 23(3):466–487, June 1994.

[36] P. Klein, C. Stein, and É. Tardos. Leighton-Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 310–321, May 1990.

[37] D. Klingman, A. Napier, and J. Stutz. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science*, 20:814–821, 1974.

[38] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[39] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S.C. Graves, A.H.G. Rinnooy Kan, and P.H. Zipkin, editors, *Handbooks in Operations Research and Management Science, Vol 4., Logistics of Production and Inventory*, pages 445–522. North-Holland, 1993.

[40] F. T. Leighton, November 1989. Private communication.

[41] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[42] T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 101–111, 1991. To appear in JCSS.

[43] 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.

[44] T. Leong, P. Shor, and C. Stein. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science: Volume 12, Network Flows and Matching*, chapter Implementation of a combinatorial multicommodity flow algorithm. October 1991.

[45] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C-30:93–100, 1981.

[46] F. Makedon and S. Tragoudas. Approximating the minimum net expansion: Near optimal solutions to circuit partitioning problems. In *Procedings of the 1990 Workshop on Graph Theoretic Concepts in Computer Science*, June 1990.

[47] B.A. Murtaugh and M.A. Saunders. MINOS 5.0 user's guide. Technical Report Technical Report 83-20, Systemns Optimizaiton Laboratory, Stanford University, 1983.

[48] S. Plotkin, D. B. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. To appear in Math of Operations Research, 1995.

[49] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 10–18, 1986.

[50] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

[51] P. Raghavan and C. D. Thompson. Provably good routing in graphs: regular arrays. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 79–87, 1985.

[52] P. Raghavan and C. D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987.

[53] R. Ravi, A. Agrawal, and P. Klein. Ordering problems approximated: single-processor scheduling and interval graph completion. In *Proceedings of the 1991 ICALP Conference*, pages 751–762, 1991.

[54] R. Schneur. *Scaling algorithms for multicommodity flow problems and network flow problems with side constraints.* PhD thesis, MIT, Cambridge, MA, February 1991.

[55] A. Schrijver. *Theory of linear and integer programming.* John Wiley and Sons, 1986.

[56] S. V. Sevast'yanov. On an asymptotic approach to some problems in scheduling theory. In *Abstracts of papers at 3rd All-Union Conf. of Problems of Theoretical Cybernetics [in Russian]*, pages 67–69. Inst. Mat. Sibirsk. Otdel. Akad. Nauk SSSR, Novosibirsk, 1974.

[57] S.V. Sevast'yanov. Efficient construction of schedules close to optimal for the cases of arbitrary and alternative routes of parts. *Soviet Math. Dokl.*, 29(3):447–450, 1984.

[58] S.V. Sevast'yanov. Bounding algorithm for the routing problem with arbitrary paths and alternative servers. *Kibernetika*, 22(6):74–79, 1986. Translation in Cybernetics 22, pages 773-780.

[59] F. Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37:318 – 334, 1990.

[60] D. B. Shmoys, C. Stein, and J. Wein. Improved approximation algorithms for shop scheduling problems. *SIAM Journal on Computing*, 23(3):617–632, June 1994.

[61] D. B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 1994. To appear.

[62] D.B. Shmoys. Personal communication, 1990.

[63] D D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[64] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[65] A. Srinivasan. A generalization of Chernoff-Hoeffding bounds, with applications. Unpublished Manuscript, 1992.

[66] S. Tragoudas. *VLSI partitioning approximation algorithms based on multicommodity flow and other techniques*. PhD thesis, University of Texax at Dallas, 1991.

[67] P. M. Vaidya. Speeding up linear programming using fast matrix multiplication. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 332–337, 1989.

[68] J. Wein. *Algorithms for Scheduling and Network Problems*. PhD thesis, MIT, Cambridge, MA, August 1991.

[69] D. P. Williamson. The non-approximability of shop scheduling. Unpublished Manuscript, 1991.

[70] D. P. Williamson, L. Hall, J. A. Hoogeven, C. A. J. Hurkens, J. K. Lenstra, and D. B. Shmoys. Short shop schedules. Unpublished Manuscript, 1993.

[71] M.A. Yakovleva. A problem on minimum transportation cost. In V.S. Nemchinov, editor, *Applications of Mathematics in Economic Research*, pages 390–399. Izdat. Social'no-Ekon. Lit., Moscow, 1959.