MULTICOMMODITY AND GENERALIZED FLOW ALGORITHMS: THEORY AND PRACTICE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jeffrey David Oldham
August 1999

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Serge Plotkin
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dan Boneh

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Cliff Stein

Approved for the University Committee on Graduate Studies:

Abstract

We present several simple, practical, and fast algorithms for linear programs, concentrating on network flow problems. Since the late 1980s, researchers developed different combinatorial approximation algorithms for fractional packing problems, obtaining the fastest theoretical running times to solve multicommodity minimum-cost and concurrent flow problems. A direct implementation of these multicommodity flow algorithms was several orders of magnitude slower than solving these problems using a commercial linear programming solver. Through experimentation, we determined which theoretically equivalent constructs are experimentally efficient. Guided by theory, we designed and implemented practical improvements while maintaining the same worst-case complexity bounds. The resulting algorithms solve problems orders of magnitude faster than commercial linear programming solvers and problems an order of magnitude larger.

We also present simple, combinatorial algorithms for generalized flow problems. These problems generalize ordinary network flow problems by specifying a flow multiplier $\mu(a)$ for each arc a. Using multipliers permit a flow problem to model transforming one type into another, e.g., currency exchange, and modification of the amount of flow, e.g., water evaporation from canals or accrual of interest in bank accounts. First, we show the generalized shortest paths problem can be solved using existing network flow ideas, i.e., by combining the Bellman-Ford-Moore shortest path framework and Megiddo's parametric search. Second, we combine this algorithm with fractional packing frameworks to yield the first polynomial-time combinatorial approximation algorithms for the generalized versions of the nonnegative-cost minimum-cost flow, concurrent flow, multicommodity maximum flow, and multicommodity nonnegative-cost minimum-cost flow problems. These algorithms show that generalized concurrent flow and multicommodity maximum flow have strongly polynomial approximation algorithms.

Contents

List of Figures

List of Tables

List of Algorithms

List of Notations

A summary of notation follows. Entries are alphabetized, ordering Greek symbols according to their Roman equivalent. Nonalphabetic symbols are listed at the beginning, arranged according to their TeX name. The page number refers to the first use of the notation.

notation	explanation	page
$[\mathit{predicate}]$	Iverson notation yields 1 if *predicate* is true and 0 otherwise	73
$v \leadsto w$	zero or more arcs in a path from v to w	73
$\nabla\phi(f)$	gradient of the potential function	14
$\odot$	closed semiring extension operator	77
$\bar{1}$	closed semiring extension identity	77
$\oplus$	closed semiring summary operator	77
(v, w)	a directed arc with tail vertex v and head vertex w	8
$v \to w$	directed arc (v, w)	73
$\bar{0}$	closed semiring summary identity	77
α	constant used in the potential function ϕ	14
A	set of a graph's arcs	8
A'	matrix of constraints for multicommodity cost-bounded flow linear program	9
$\mathcal{B}$	maximum permitted flow cost	9
$\mathcal{B}^*$	minimum possible cost of a feasible flow	10
B	largest integer to represent arc capacities, demands, and flow multipliers	69
C	maximum arc cost	14
c	arc cost function specifying cost of a unit flow on each arc (v, w)	8

notation	explanation	page
$c \cdot f$	dot product of the cost and flow vectors over the implicit arc set A, i.e., the flow's cost	9
c^{π}	reduced arc cost function specifying reduced arc cost for every arc	77
$c^{\pi}(v,w)$	reduced cost of arc (v,w)	77
d_i	desired amount of flow from commodity i's source to its sink	8
ϵ	desired accuracy for the solution	10
f	flow, i.e., nonnegative arc function specifying amount of flow on each arc; sometimes overloaded to represent a multicommodity flow	8
f_i	flow of a particular commodity i	8
f_i^*	minimum-cost flow for commodity i, ignoring all other commodities	14
$f(v,w)$	flow on arc (v,w); sometimes overloaded to mean the sum of all commodities' flows on the arc	8
G	directed graph	8
K	the set of grouped commodities	8
k	the number of (grouped) commodities	10
k_0	the number of simple (ungrouped) commodities in the input	10
λ	maximum congestion over all constraints	9
λ_A	maximum arc congestion	9
λ_c	cost congestion, i.e., the ratio $c \cdot f / \mathcal{B}$ of the flow's cost to the maximum permitted cost	9
$\lambda_c(\sigma)$	cost congestion considered as a function of the step size σ	19
$\lambda_{(v,w)}$	arc congestion, i.e., ratio of flow $f(v,w)$ to its capacity $u(v,w)$	9
$\lambda_{(v,w)}(\sigma)$	arc congestion considered as a function of the step size σ	19
$\mu(v,w)$	flow multiplier for arc (v,w)	65
MCF	minimum-cost flow	23
m	the number of arcs in A	8
n	the number of vertices in V	8

notation	explanation	page
$O(f)$	notation indicating the asymptotic growth of function f	13
$\tilde{O}(f)$	asymptotic notation omitting logarithmic terms	12
π	all vertex potentials, i.e., all dual variables	77
$\pi(v)$	potential, i.e., dual variable, of vertex v	77
ϕ	potential function value for an implicit flow	14
$\phi(f)$	potential function value for flow f	14
$\phi(\sigma)$	potential function considered as a function of the step size σ	19
r	particular arc capacity or flow cost constraint	14
σ	step size, i.e., the fraction of flow that is rerouted	14
σ_{min}	smallest step size to consider	19
S	closed semiring domain	77
s_i	source vertex with positive net outflow for commodity i	8
t_i	sink vertex with positive net inflow for commodity i	8
U	maximum arc capacity	14
u	arc capacity function specifying maximum flow $u(v,w)$ for each arc (v,w)	8
V	set of a graph's vertices	8
y_r	dual variable corresponding to constraint r	14

Chapter 1

Introduction

We present practical and theoretically fast algorithms for multicommodity flow and generalized flow problems. While these network flow problems can be solved as linear programs, our combinatorial algorithms exploit the problems' network flow structure to yield algorithms that are faster in practice and in theory.

Ordinary network flow problems specify a network, origin and destination vertices, supplies and demands, and arc characteristics such as costs and capacities. Given these constraints, algorithms solve questions like "What is the maximum amount of flow from the origins to the destinations given the arc capacities?" and "What is the cheapest way to satisfy the supplies and demands?". These problem statements assume that 1) the network has only one type of object flowing through it and 2) that objects are conserved as they flow through the network. Thus, any flow can travel from any origin to any destination neither increasing or decreasing in amount.

Multicommodity flow problems model simultaneously shipping several different types of objects (commodities) through a network. For example, each telephone call in a telecommunications network should originate and terminate at a specified origin and destination, not just the origin or destination of any telephone call currently in the network. Even though each telephone call is treated as a separate commodity, all the calls must simultaneously share the same network capacity constraints. Thus, multicommodity flow problems answer questions such as "Given a set of commodities' origins and destinations, what is the maximum amount of flow given the arc capacities?" and "Given a set of commodities, what is the cheapest way to satisfy their supplies and demands?" Problems which can be modeled as multicommodity flow problems include telecommunication problems, traffic planning, import-export problems, and freight transport and scheduling.

Generalized flow problems permit flows to be multiplicatively modified as they travel through arcs. For example, breakage of items during transport can be modeled using flow multipliers with value less than one, e.g., 90%. Thus, one out of every ten shipped items will be broken. Gainy flow multipliers greater than one, e.g., 1.05, can model accrual of interest or exchange rates. Flow is conserved at the vertices so the amount of flow entering a vertex equals the amount leaving a vertex. Thus, generalized flow problems answer questions such as "Given the flow multipliers, what is the maximum amount of flow that can leave the origins while obeying the arc capacities?" and "Given the flow multipliers, what is the cheapest way to remove the supplies?" Many problems can be modeled as generalized flow problems including power transmission, water canal shipment, and investment decision problems.

We present and analyze algorithms for the multicommodity and generalized flow problems. Algorithmic analysis usually determines worst-case time bounds, providing an upper bound on the algorithm's running time as a function of the problem input. In this thesis, we create and analyze algorithms with respect to three measures: worst-case performance, empirical performance, and ease of implementation. Most analyses determine worst-case performance bounds because they are frequently easier to determine than empirical or average-case performance. They have the additional advantage that no assumptions about realistic problem distributions are required. That is, an answer is guaranteed within the given time bound for all problem inputs. Worst-case and empirical performance need not be related. For example, some simplex algorithms can require worst-case exponential time, e.g., [Zad73a, Zad73b], but are frequently the fastest methods to solve linear programs [Bix98]. Although empirical analysis better indicates how an algorithm performs in practice than worst-case analysis, it is limited by the need to test algorithms on a representative class of inputs. Combining worst-case and empirical analyses, we combine the advantages of each analysis, reducing the disadvantages: an algorithm can be empirically tested on a class of inputs knowing that its performance is still guaranteed on instances not empirically tested.

Algorithms with good worst-case performance are only of theoretical interest if they are too difficult to implement. Thus, we seek to design new algorithms either by combining simple building algorithmic blocks such as binary search and the Newton-Raphson method or by creating algorithmic frameworks likely to be useful for a wide variety of algorithms.

Most of the algorithms we present are *fully polynomial-time approximation schemes*.

That is, for any desired accuracy ϵ, an algorithm computes a solution within a $1+\epsilon$ factor of the optimal solution (if minimizing or within a $1-\epsilon$ if maximizing) in worst-case time that is polynomial in the problem input and $1/\epsilon$. For many applications, an ϵ-optimal solution suffices because either an exact solution is not needed or the input data itself is accurate to some fixed precision.

We adopt a network-flow perspective throughout the thesis. For example, all problems will be stated in terms of networks, flows, and arc capacities and costs. We however still make extensive use of linear programming concepts: dual variables, complementary slackness conditions, weak duality inequalities.

In the first part of the thesis, we present new practical combinatorial approximation algorithms for the multicommodity minimum-cost flow and the concurrent flow problems. Based on the combinatorial fractional packing frameworks of [KP95b, LMP$^+$95], these potential-function algorithms compute a solution by repeatedly choosing a commodity and computing a single-commodity minimum-cost flow in an auxiliary graph. This graph's arc costs are exponential functions of the current flow. A fraction of the commodity's flow is then rerouted to the corresponding minimum-cost flow. Each rerouting hopefully decreases the potential function. While theoretically fast, a direct implementation is very slow in practice. To create a practical algorithm, we experimentally evaluated theoretically equivalent techniques and determined practical ways to compute the values of several parameters. Our methods to compute the parameters were motivated by algorithmic analysis while maintaining the theoretical invariants. Among the issues crucial for achieving an efficient implementation were

- computing the constant used by the exponential functions to increase each iteration's progress,
- determining when the algorithm has an ϵ-optimal flow, i.e., when to terminate,
- computing the step size that maximizes the potential function reduction each iteration, and
- reducing the time to compute minimum-cost flows.

The resulting implementation solves our problem instances two to three orders of magnitude faster than two simplex-based implementations. It also solves much larger problem instances than previously possible. This provides strong evidence that other algorithms based on fractional packing frameworks can be practical.

In the second part of the thesis, we present combinatorial algorithms for the generalized flow problems. For the generalized shortest paths problems, we present a guess-and-verify algorithm with running time that exactly matches the fastest existing algorithm. Each iteration consists of guessing the answer and verifying, using a Bellman-Ford-Moore computation, whether the guess is larger than, equal to, or smaller than the minimum cost. Combining this comparison subroutine with Megiddo's parametric search technique (or binary search) yields the algorithm. Combining this generalized shortest path algorithm with fractional packing frameworks similar to those used for multicommodity flows, we created combinatorial approximation algorithms for many generalized flow problems. These approximation algorithms were the first combinatorial algorithms for the generalized minimum-cost flow, generalized concurrent flow, generalized multicommodity maximum-flow, and generalized multicommodity minimum-cost flow problems. (For these algorithms, arc costs must be nonnegative.) The approximation algorithms for generalized concurrent flow and multicommodity maximum-flow run in strongly-polynomial time.

Chapter 2

Multicommodity Flows

2.1 Introduction

The *multicommodity minimum-cost flow problem* involves simultaneously shipping multiple commodities through a single network so the total flow obeys the arc capacity constraints and has minimum cost. The problem occurs in many contexts where different items share the same resource, e.g., communication networks, transportation, and scheduling problems [AMO93, HL96, HO96].

Traditional methods for solving minimum-cost and no-cost multicommodity flow problems are linear-programming based [AMO93, Ass78, CN96, KH80]. Using the ellipsoid [Kha80] or the interior-point [Kar84] methods, linear-programming problems can be solved in polynomial time. Theoretically, the fastest algorithms for solving the multicommodity minimum-cost flow problem exactly [KP95a, KV86, Vai89a] use the problem structure to speed up the interior-point method.

In practice, solutions to within, say 1%, often suffice. More precisely, we say that a flow is *ϵ-optimal* if it overflows the capacities by at most $1+\epsilon$ factor, has cost that is within $1+\epsilon$ of the optimum, and ϵ is nonnegative. Algorithms for computing approximate solutions to the multicommodity flow problem were developed in [LMP+95] (no-cost case) and [GK96a, KP95b, PST95] (minimum-cost case). Theoretically, these *approximation algorithms* are much faster than interior-point method based algorithms for constant accuracy ϵ. The algorithm in [LMP+95] was implemented [LSS93] and was shown that

[0]Chapters 2–5 contain joint work with Andrew V. Goldberg, Serge Plotkin, and Cliff Stein [GOPS97, GOPS98].

indeed it often outperforms more traditional approaches. Prior to our work, it was not known whether the combinatorial approximation algorithms for the minimum-cost case can be implemented to run quickly.

In the next few chapters, we describe `MCMCF` and `MCF`, our combinatorial approximation algorithms for the multicommodity minimum-cost flow and concurrent flow problems, respectively. A direct implementation of the theoretically fast algorithm of Karger and Plotkin [KP95b] yielded a correct but practically slow implementation. Much experimentation helped us select among the different theoretical insights of [KP95b, LMP$^+$95, LSS93, PST95, Rad95] to achieve good practical performance.

We compare our algorithms' performance with `CPLEX` [CPL95] and `PPRN` [CN96]. (Several other efficient multicommodity minimum-cost flow implementations, e.g., [ARVK89], are proprietary so we were unable to use these programs in our study.) Both are based on the simplex method [Dan63] and both find exact solutions. `CPLEX` is a state-of-the-art commercial linear programming package, and `PPRN` uses a primal partitioning technique to take advantage of the multicommodity flow problem structure.

Our results indicate that the theoretical advantages of approximation algorithms over linear-programming-based algorithms can be translated into practice. On the examples we studied, our algorithms were several orders of magnitude faster than `CPLEX` and `PPRN`. For example, for 1% accuracy, it was up to three orders of magnitude faster. Our implementation's dependence on the number of commodities and the network size is also smaller, and hence we are able to solve larger problems.

We would like to compare our codes' running times with modified `CPLEX` and `PPRN` programs that yield approximate solutions, but it is not clear how to make the modifications. Even if we could make the modifications, we would probably need to use `CPLEX`'s primal simplex to obtain a feasible flow before an exact solution is found. Since its primal simplex is an order of magnitude slower than its dual simplex for the problem instances we tested, the approximate code would probably not be any faster than computing an exact solution using dual simplex.

To find an ϵ-optimal multicommodity flow, our algorithms repeatedly choose a commodity and then compute a single-commodity minimum-cost flow in an auxiliary graph. The arc costs in this auxiliary graph are exponential functions of the current flow. The base of the exponent depends on a parameter α, which our implementation chooses. A fraction σ of the commodity's flow is then rerouted to the corresponding minimum-cost flow. Each rerouting decreases a certain potential function. The algorithms iterate this process until it finds an ϵ-optimal flow.

As we have mentioned above, a direct implementation of [KP95b], while theoretically fast, is very slow in practice. Several issues are crucial for achieving efficient algorithms:

Exponential Costs: The value of the parameter α, which defines the base of the exponent, must be chosen carefully: Using a value that is too small will not guarantee any progress, and using a value that is too large will lead to very slow progress. Our adaptive scheme for choosing α leads to significantly better performance than using the theoretical value. Importantly, this heuristic does not invalidate the worst-case performance guarantees proved for algorithms using fixed α

Stopping Condition: Theoretically, the algorithm yields an ϵ-optimal flow when the potential function becomes sufficiently small [KP95b]. Alternative algorithms, e.g., [PST95], explicitly compute lower bounds. Although these stopping conditions lead to the same asymptotic running time, the latter one leads to much better performance in our experiments.

Step Size: Theory specifies the rerouting fraction σ as a fixed function of α. Computing the step size σ maximizing the exponential potential function reduction decreases the running time. We show that is it possible to use the Newton-Raphson method [Rap90] to quickly find a near-optimal value of σ for every rerouting. Additionally, a commodity's flow usually differs from its minimum-cost flow on only a few arcs. We use this fact to speed up these computations.

Minimum-Cost Flow Subroutine: Minimum-cost flow computations dominate the algorithm's running time both in theory and in practice. The arc costs and capacities do not change much between consecutive minimum-cost flow computations for a particular commodity. Furthermore, the problem size is moderate by minimum-cost flow standards. This led us to decide to use the primal network simplex method. We use the current flow and a basis from a previous minimum-cost flow to "restart" each minimum-cost flow computation. Excepting the restart idea, our primal simplex code is similar to that of Grigoriadis [Gri86].

In the rest of this chapter, we define the various multicommodity flow problems, briefly describe some applications, review various algorithmic techniques to solve them, and then sketch the algorithms. We conclude this chapter with a description of the problem instances on which we tested our algorithms. In subsequent chapters, we describe the issues crucial for achieving efficient algorithms; present experimental results indicating the algorithms' dependences on the number of commodities, the network size, and the desired accuracy; and then compare our implementations with other implementations.

2.2 Definitions

The *multicommodity minimum-cost flow problem* consists of a directed network $G = (V, A)$, a positive arc capacity function $u : A \to \mathbf{R} > 0$, a nonnegative arc cost function $c : A \to \mathbf{R} \geq 0$, and a specification (s_i, t_i, d_i) for each commodity i, $i \in \{1, 2, \ldots, k_0\}$. Vertices s_i and t_i are the *source* and the *sink* of commodity i, and a positive number d_i is its *demand*. The goal is to find a flow for all commodity satisfying their demands and arc capacity constraints while minimizing the total cost.

We let n and m represent the number of vertices and arcs, respectively. To ease notation, we denote an arc (v, w) using its tail and head vertices even though our algorithms permit multiple arcs between the same pair of vertices. As described in Section 3.6, we reduce the number of commodities by grouping together commodities with the same sources or sink vertices. Let K denote the set of grouped commodities i, $i \in \{1, 2, \ldots, k\}$.

A *multicommodity flow* is a nonnegative arc function $f : K \times A \to \mathbf{R} \geq 0$. A *flow* $f_i : A \to \mathbf{R} \geq 0$ of commodity i is a flow obeying conservation constraints and satisfying its demand d_i, i.e., for each vertex $v \in V$,

$$\sum_{\{w:(v,w)\in A\}} f_i(w, v) - \sum_{\{w:(w,v)\in A\}} f_i(v, w) = \begin{cases} -d_i & \text{if } v = t_i \\ d_i & \text{if } v = s_i \\ 0 & \text{otherwise.} \end{cases}$$

We define the total flow $f(v, w)$ on arc (v, w) by $f(v, w) = \sum_{i=1}^{k_0} f_i(v, w)$. Depending on context, the symbol f represents both the (multi)flow $(f_1, f_2, \ldots, f_k)$ and the total flow $f_1 + f_2 + \cdots + f_k$, summed arc-wise. The cost of a flow f is the dot product $c \cdot f =$

$\sum_{(v,w)\in A} c(v,w)f(v,w)$. Written as a linear program,

$$\begin{aligned}
&\min \sum_{\text{comm. } i} \sum_{\text{arcs } (v,w)\in A} c(v,w) f_i(v,w) \\
&\text{s.t. } (\forall \text{ comm. } i)(\forall v \in V) \left(\sum_{\{w:(v,w)\in A\}} f_i(v,w) - \sum_{\{w:(w,v)\in A\}} f_i(w,v) = d_i(v) \right) \\
&\quad (\forall \text{ arcs } (v,w) \in A) \left(\sum_{\text{comm. } i} f_i(v,w) \le u(v,w) \right) \\
&\quad (\forall \text{ comm. } i)(\forall \text{ arcs } (v,w) \in A)(f_i(v,w) \ge 0).
\end{aligned}$$

It is frequently more convenient to assume a fixed cost bound and write this *multicommodity cost-bounded flow problem* using congestions. Given a problem instance, a flow f, and a cost budget $\mathcal{B}$, the *congestion of arc* (v,w) is $\lambda_{(v,w)} = f(v,w)/u(v,w)$ and the *congestion of the flow* is $\lambda_A = \max_{(v,w)} \lambda_{(v,w)}$. The *cost congestion* is $\lambda_c = c \cdot f/\mathcal{B}$, and the *total congestion* is $\lambda = \max\{\lambda_A, \lambda_c\}$. A *feasible problem instance* has a flow f with $\lambda_A \le 1$. We can write this as a linear program.

$$\text{Min } \lambda \tag{2.1}$$

$$\text{s.t. } (\forall \text{ comm. } i)(\forall v \in V) \left(\sum_{\{w:(v,w)\in A\}} f_i(v,w) - \sum_{\{w:(w,v)\in A\}} f_i(w,v) = d_i(v) \right)$$

$$(\forall \text{ arcs } (v,w) \in A) \left(\sum_{\text{comm. } i} f_i(v,w) \le \lambda u(v,w) \right) \tag{2.2}$$

$$\sum_{\text{comm. } i} \sum_{\text{arcs } (v,w)\in A} c(v,w) f_i(v,w) \le \lambda \mathcal{B} \tag{2.3}$$

$$(\forall \text{ comm. } i)(\forall \text{ arcs } (v,w) \in A)(f_i(v,w) \ge 0).$$

We can also write this using matrix and vector notation: $\min \lambda$ such that $A'f \le \lambda$ and f is a multiflow satisfying all commodities' demands. We view f as a vector with length km. A' is an $(m+1)$-by-km matrix. The first m rows correspond to the arc capacity constraints (2.2) while the last row corresponds to the arc cost function c (2.3). Note that we have divided each constraint by $u(v,w)$ and $\mathcal{B}$, respectively.

Our implementation approximately solves the multicommodity minimum-cost flow problem. Given an accuracy $\epsilon > 0$ and a feasible multicommodity flow problem instance, the algorithm finds an *ϵ-optimal flow* f with *ϵ-optimal congestion*, i.e., $\lambda_A \leq (1+\epsilon)$ and *ϵ-optimal cost*, i.e., if $\mathcal{B}^*$ is the minimum cost of any feasible flow, f's cost is at most $(1+\epsilon)\mathcal{B}^*$. Because we can choose ϵ arbitrarily small, we can find a solution arbitrarily close to the optimal.

Our implementation also solves the *concurrent flow problem*. The problem has the same input as the multicommodity minimum-cost flow problem except omitting the cost function and seeking to maximize the fraction of each commodity's demands that are satisfied given the arc capacity constraints. Equivalently, the goal is to minimize the multiple of the arc capacity constraints to ensure all demands are satisfied. The problem is the same as the cost-bounded linear program listed above but omitting the cost constraint (2.3).

We combine commodities with the same source vertices to form commodities with one source and (possibly) many sinks (see Section 3.6 and [LSS93, Sch91]). Thus, the number k of commodity groups may be smaller than the number k_0 of simple commodities in the input.

These multicommodity flow definitions can be generalized in several different ways. For the *multicommodity maximum-flow problem*, commodity demands are not specified, and the goal is to maximize the total flow, over all commodities, while obeying the arc capacity constraints. Costs and capacities can vary according to commodity. Additional side constraints can be added. Commodities can have multiple sources and sinks. Versions for undirected graphs can also be defined. For experimental purposes, we concentrate on the multicommodity minimum-cost and concurrent flow problems for directed graphs as presented above.

2.3 Applications

Multicommodity flow problems find applications in a wide variety of application contexts including transportation, logistics, network design, communication networks, urban planning, and scheduling. We will briefly survey a few applications. Interested readers are referred to the surveys found in [Ken78] and [Sch91, Section 2.1] and the list of applications in [AMO93, Chapter 17].

In many applications, commodities moving through a network can be differentiated because they are different physical goods and/or because they have different points of origin and destination. An example of the former is a world grain model developed to study the effects of U.S. port capacities [BBM82]. The network consists of demand and supply vertices, vertices representing different world regions and ports, and arcs indicating shipments. Commodities model grains, e.g., corn, wheat, rice, and soybeans, and the joint arc-capacity constraints model port capacities. An example of the latter type of modeling is a communication network. Vertices represent origins and destinations of messages and arcs represent transmission lines. A commodity and its demand models the traffic between particular origin and destination vertices. Each transmission line has a fixed capacity. The goal is to route the messages in the cheapest way possible while obeying the arc capacities, i.e., the multicommodity minimum-cost flow problem.

A scheduling application of multicommodity flow problems is freight assignment in the less-than-truck-load (LTL) industry. LTL carriers schedule trucks' pickups and deliveries to minimize the total service time to satisfy all their customers. Once routes are fixed, the problem can be modeled as a multicommodity flow problem. Each commodity represents a shipment with a particular origin and destination. The vehicle routes define the network over which shipments are to be serviced. At exchange depots (modeled by vertices), shipments can be transferred. The arc capacity constraints correspond to truck capacities. For more information, see Barnhart's Ph.D. thesis [Bar88].

2.4 Previous Work

As we noted earlier, researchers have developed several different classes of algorithms to solve multicommodity flow problems. Since these problems can be written as linear programs, techniques to solve general linear programs such as the simplex method suffice. Specialized algorithms take advantage of the problem's special structure. These include price-directive decomposition, resource-directive decomposition, partitioning methods, and specialized interior-point methods. Combinatorial approaches first appeared in the late 1980s. Here we only briefly sketch the main ideas of each approach, referring the interested reader to the textbook chapter [AMO93, Chapter 17] and surveys such as [Ass78] and [Ken78].

Price-directive decomposition techniques assign Lagrangian multipliers, i.e., prices, to the joint arc capacity constraints and move them into the objective function. Thus, the problem decomposes into k separate minimum-cost flow problems, which are easy to solve.

The difficulty is finding appropriate prices. Lagrangian relaxation update the prices using the technique of subgradient optimization. If an arc's capacity is violated, its price is increased by an amount proportional to the amount of violation. If an arc is underutilized, its price is reduced by a proportional amount, ensuring the multiplier remains nonnegative. Dantzig-Wolfe decomposition [DW60, FF58] decomposes the problem into "easy" single-commodity constraints and "hard" joint arc-capacity constraints. Each iteration, a price-setting master linear program produces prices. Then the separate minimum-cost flow (or shortest path) computations occur.

Resource-directive decomposition techniques allocate the joint arc capacities among the various commodities. Thus, a problem decomposes into k separate minimum-cost flow problems, which are easy to solve. The difficulty is deciding how to allocate the resources. To decide this, a nonlinear program is used. Consider the effect of a resource allocation vector r (that obeys the joint arc-capacity constraints) on the cost of the solution. We seek the vector r that minimizes the total cost. Using ideas similar to subgradient optimization, the vector is modified by taking a step in the direction minimizing r.

Partitioning methods partition the linear program's basis into one spanning tree basis per commodity and one basis for the joint arc-capacity constraints. The method then proceeds in a way similar to the simplex method. Variables are pivoted in and out of the basis, but solving the associated problems can be done using the significantly smaller bases.

Whether multicommodity flow problems could be solved within polynomial time remained an open question until Khachiyan presented a polynomial-time algorithm for linear programs in 1979 [Kha80]. In 1984, Karmarkar [Kar84] presented a different interior point algorithm. Kapoor and Vaidya [KV86, KV96] gave a method for speeding up the matrix inversions in Karmarkar-type algorithms for multicommodity flow problems. Combining their technique with Vaidya's linear programming algorithm that uses fast matrix multiplication [Vai89a] yields a time bound of $\tilde{O}(k^{2.5}m^{0.5}n^2 \log \epsilon^{-1})$ to solve multicommodity minimum-cost flow problems. (The $\tilde{O}()$ notation omits uninteresting logarithmic terms.) Kamath and Palmon [KP95a] improved the running time for some parameter values for approximate and exact solutions by modeling the problem as a quadratic optimization problem.

Instead of viewing multicommodity flow problems as sets of linear inequalities to optimize, one can view the problem as a combinatorial problem on a graph. Shahrokhi and Matula [SM90] presented a combinatorial polynomial-time approximation algorithm for the concurrent flow problem with unit arc capacities. The running time was $O(\epsilon^{-5}nm^7)$. Klein et al. [KPSET94] reduced the running time, but Leighton et al. [LMP$^+$95] showed how to solve the concurrent flow problem with unrestricted arc capacities. The main idea was to compute prices for the joint arc-capacity constraints using exponential functions. Single-commodity minimum-cost flows are computed using the gradient of the sum of these exponential functions as arc costs. Some flow is rerouted along these minimum-cost flow paths, reducing the sum. A randomized choice of commodities requires $\tilde{O}(\epsilon^{-3}k)$ minimum-cost flows, while a deterministic choice requires $\tilde{O}(\epsilon^{-2}k^2)$ flows. See also Cliff Stein's thesis [Ste92]. Researchers subsequently focussed on the choice of commodities. Goldberg [Gol92] showed choosing the commodities uniformly randomly reduces the running time by a factor of ϵ^{-1}. Independently, Grigoriadis and Khachiyan [GK94] achieved the same result. Radzik [Rad95] showed a deterministic round robin ordering, i.e., fixing an order and repeatedly visiting each commodity according to that order, requires the same running time (up to logarithmic factors).

Karger and Plotkin [KP95b] and Grigoriadis and Khachiyan [GK96a] turned their attention to combinatorial algorithms for the multicommodity minimum-cost flow problem. To ensure convergence in the presence of a cost function, Karger and Plotkin added a linear term to the gradient cost function, increasing the complexity by a factor of ϵ^{-1}. The latter researchers reduced the complexity back to $O(\epsilon^{-2}kmn)$.

Researchers have developed and tested several codes for multicommodity flow problems. Kennington [Ken78] and Ali et al. [ABF$^+$84] found that specialized code were two to five times faster than general-purpose linear programming code. Leong, Shor, and Stein [LSS93, Ste92] implemented the combinatorial concurrent flow algorithm of Leighton et al. [LMP$^+$95], illustrating that the combinatorial technique permits solving problems previously considered too large to solve. Castro and Nabona [CN96] and Castro [Cas98] implemented primal partitioning and interior point algorithms, respectively, further enlarging the size of solvable problems. In the mean time, the running time of the `CPLEX` general linear programming package has been decreasing at a rapid rate during the past few years, beating specialized codes for some multicommodity flow instances.

Algorithm 1 Outline for Our Multicommodity Flow Algorithms

$f \leftarrow$ initial flow satisfying commodities' demands
while f is not ϵ-optimal **do**
  for all constraints r **do**
    $y_r \leftarrow e^{\alpha(\lambda_r - 1)}$
  $\phi(f) \leftarrow \sum_r y_r$
  Choose a commodity i to reroute.
  arc costs $c' \leftarrow \nabla\phi(f)$ restricted to commodity i
  $f_i^* \leftarrow$ minimum-cost-flow$_i(c')$
  $f_i \leftarrow (1-\sigma)f_i + \sigma f_i^*$

2.5 The Algorithmic Framework

Our algorithms are mostly based on [KP95b]. Roughly speaking, the approach in that paper is as follows. The algorithm (Algorithm 1) first finds an initial flow satisfying demands but which may violate capacities and may be too expensive. The algorithm repeatedly modifies the flow until it becomes ϵ-optimal. Each iteration, the algorithm first computes the theoretical values for the constant α and the step size σ. It then computes the dual variables $y_r = e^{\alpha(\lambda_r - 1)}$, where r ranges over the arcs A and the arc cost function c, and a potential function $\phi(f) = \sum_r y_r$. The algorithm chooses a commodity i to reroute in a round robin order, as in [Rad95]. It computes, for that commodity, a minimum-cost flow f_i^* in a graph with arc costs related the gradient $\nabla\phi(f)$ of the potential function and arc capacities $\lambda_A u$. The commodity's flow f_i is changed to the convex combination $(1-\sigma)f_i + \sigma f_i^*$. An appropriate choice of values for α and σ lead to $\tilde{O}(\epsilon^{-3}nmk)$ running time. Grigoriadis and Khachiyan [GK96a] decreased the dependence on ϵ to ϵ^{-2}.

Since these minimum-cost algorithms compute a multiflow having arc cost at most a budget bound $\mathcal{B}$, we use binary search on $\mathcal{B}$ to determine an ϵ-optimal cost. First we use our concurrent flow algorithm to determine if the problem ignoring the cost is feasible, i.e., the demands can be satisfied using the given arc capacities. If this no-cost problem is not feasible, the problem with costs is also infeasible. Thus, we can assume that the minimum arc congestion $\lambda_A^* \leq 1$. To perform binary search, we use the cost of the initial flow as the initial lower bound because the flow is the union of minimum-cost single-commodity flows with respect to the arc cost function c and arc capacities u and guess an upper bound of mCU, where C and U denote the maximum arc cost and the maximum arc capacity, respectively. Lower bound computations (see Section 3.1) increase the lower bound and the algorithm decreases the congestion and cost until an ϵ-optimal flow is found.

Theorem 1 ([KP95b, Theorem 3.2]) *The multicommodity minimum-cost flow problem can be ϵ-optimally solved in $\tilde{O}(\epsilon^{-3}knm)$ time.*

For the concurrent flow problem, we wish to determine a congestion within a factor $1+\epsilon$ of the minimum possible arc congestion. Given a feasible flow and a lower bound of zero, lower bound computations repeatedly increase the lower bound while the algorithm decreases the congestion.

In Chapter 3, we more fully describe the different choices for the details of the algorithm, but first we describe the instances used to experimentally evaluate the different choices.

2.6 The Problem Instances

To test our implementation, we modified single-commodity maximum-flow and minimum-cost flow problem generators to produce multicommodity minimum-cost flow problems. We also wrote a new problem generator. Different algorithms may find solving different problem families' instances harder or easier. All instances' demands are scaled to yield feasible problems. We arbitrarily chose a maximum arc congestions of 0.60. In this section, we describe the problem instance characteristics.

The problem generator MULTIRMFGEN (abbreviated RMFGEN, written by Badics [Bad91], and based on [GG88]) produces two-dimensional *frames* (grids) with arcs connecting a random permutation of vertices in adjacent frames. The intraframe arcs have large capacities, while the interframe arcs have uniformly random capacity from the range $[1, 100]$. All arc costs are uniformly randomly sampled from the range $[1, 100]$. Commodities' sources and sinks are randomly chosen. Each frame in a problem denoted rmfgen-a-b-k has a square $a \times a$ grid of vertices such that each vertex is connected to each of its three (or two if on the border of the grid) neighbors by a directed arc with capacity $100a^2$. It has b frames and k_0 (ungrouped) commodities. All interframe arcs have the same direction.

The generator MULTIGRIDGEN (abbreviated MULTIGRID and based on [LO91]) produces two-dimensional grids with a limited number of additional arcs connecting randomly chosen vertices. The additional arcs have higher costs uniformly chosen from the range $[200, 2000]$ and fixed capacities of 2000 units versus the $[0, 200]$ cost range and $[20, 200]$ capacity range for the grid arcs. Commodities' sources and sinks are randomly chosen vertices. A problem denoted multigrid-a-b-k indicates an $a \times a$ grid of vertices. Arcs in odd- and even-numbered columns (rows) have opposite direction. b additional (possibly multiple) arcs and k_0 (ungrouped) commodities complete the problem.

We wrote TRIPARTITE to produce difficult-to-solve graphs to test our program's dependence on the network's size. Problem instances have a given number of layers, each consisting of a tripartite graph (A, B, C). Complete bipartite graphs connect A with B and B with C, while the ith vertex of C connects to the ith vertex of A in the next layer. The bipartite sets of arcs have roughly equal capacity, while the capacity of each of the interlayer arcs is small excepting one with between 50 and 75% of the total interlayer capacity. All arcs have random costs in the range $[1, 1000]$. Commodities' sources and sinks are at opposite ends of the graph. A problem denoted tripartite-a-k indicates a layers and k_0 (ungrouped) commodities.

The real-world GTE problem instance has 49 vertices, 260 directed arcs, and 585 (ungrouped) commodities. Presumably, the problem models a telecommunication network of the GTE Corporation. (The problem was obtained via Cliff Stein from Schneur [Sch91].) All directed arcs come in pairs $v \rightarrow w$ and $w \rightarrow v$. Thirty-four of the 130 pairs have capacity of 10000 with the others having capacity of 320. There is no easily discernible pattern to the commodities except all commodity sources are numerically less than the sinks. There is no easily discernible pattern to the arc costs.

Except where otherwise indicated, all data was obtained using a 300 MHz Sun UltraSparc-2 with 512 megabytes of main memory and running SunOS 5.5.1.

Chapter 3

Translating Theory into Practice

The algorithmic framework described in Section 2.5 is theoretically efficient, but a direct implementation requires orders of magnitude more running time than commercial linear-programming packages [CPL95]. Guided by the theoretical ideas of [KP95b, LMP$^+$95, PST95], we converted the theoretically correct but practically slow implementation to a theoretically correct and practically fast implementation. In some cases, we differentiated between theoretically equivalent implementation choices that differ in practicality, e.g., see Section 3.1. In other cases, we used the theory to create heuristics that, in practice, greatly reduce the running time, but, in the worst case, do not have an effect on the theoretical running time, e.g., see Section 3.3.

Because there are too many different choices to evaluate all possible combinations, we experimentally evaluate each choice individually. For example, when comparing schemes to choose step sizes, we use the best termination scheme and the best scheme for α.

3.1 The Termination Condition

Theoretically, a small potential function value and a sufficiently large value of the constant α indicates the flow is ϵ-optimal [KP95b, Lemma 2.1], but this pessimistic indicator leads to poor performance. Instead, we periodically compute the lower bound on the optimal congestion λ^* found in [LMP$^+$95, PST95]. As we saw in Section 2.5, our algorithm's goal is to either decrease the congestion to within a $1+\epsilon$ factor of our current lower bound or show that the lower bound is much smaller than λ^*.

First, we consider the multicommodity cost-bounded problem. The weak duality inequalities [Chv83, Chapter 9] yield a lower bound. Using the matrix-vector notation for the multicommodity cost-bounded flow problem,

$$\lambda' \sum_r y_r \geq \sum_{\text{comm. } i} (y^t A') f_i(\lambda'_A) \geq \sum_{\text{comm. } i} (y^t A') f_i^*(\lambda'_A). \tag{3.1}$$

Here the flow $f_i(\lambda'_A)$ obeys arc capacities $\lambda'_A u$ and $f_i^*(\lambda'_A)$ represents commodity i's minimum-cost flow with respect to cost function $y^t A'$. To prove the left inequality, multiply the matrix-vector notation by the vector of dual variables. The second inequality follows because the cost $(y^t A') f_i(\lambda'_A)$ of the current flow cannot be less than the minimum-cost flow $(y^t A') f_i^*(\lambda'_A)$.

Since the inequalities are also true for the optimal solution with congestion λ^* and flow obeying arc capacities $1u$,

$$\lambda^* \geq \frac{\sum_{\text{comm. } i} (y^t A') f_i^*(1)}{\sum_r y_r} = \frac{\sum_{\text{comm. } i} (y^t A') f_i^*(1)}{\phi}. \tag{3.2}$$

We use this inequality to compute a lower bound. To do so, first, we compute the cost function $y^t A'$. Then we compute k minimum-cost flows, sum their costs, and divide by the potential function $\phi = \sum_r y_r$. Because it requires k minimum-cost flow computations, computing this lower bound is computationally expensive. Thus, we compute a lower bound once for every $O(k)$ iterations. Experimentally, we determined that $3k$ works well.

The computation for the concurrent flow is very similar. Since this problem does not have costs, the matrix A' has one fewer rows. Since the inequalities are true for the optimal solution with congestion λ^* and flow obeying arc capacities $\lambda^* u$,

$$\lambda^* \geq \frac{\sum_{\text{comm. } i} (y^t A') f_i^*(\lambda^*)}{\sum_r y_r}.$$

Since we do not know the optimal congestion λ^*, we use $\lambda \geq \lambda^*$ and

$$\lambda^* \geq \frac{\sum_{\text{comm. } i} (y^t A') f_i^*(\lambda)}{\sum_r y_r}$$

since increasing the arc capacities cannot increase the cost of the minimum cost flows.

3.2 Computing the Step Size σ

While, as suggested by the theory, using a fixed step size σ_{min} to form the convex combination $(1-\sigma)f_i + \sigma f_i^*$ suffices to reduce the potential function, our algorithm computes σ to maximize the potential function reduction.

Given the current flow f and the minimum-cost flow f_i^* for commodity i, the potential function $\phi(\sigma)$ is a function of the step size σ. To see this, we write the formula for $\phi(\sigma)$.

$$\phi(\sigma) = \sum_r \exp(\alpha(\lambda_r(\sigma) - 1)) \tag{3.3}$$

For arc capacity constraints,

$$\lambda_{(v,w)}(\sigma) = \frac{f(v,w) + \sigma(f_i^*(v,w) - f_i(v,w))}{u(v,w)}. \tag{3.4}$$

For the cost constraint,

$$\lambda_c(\sigma) = \frac{c \cdot f + \sigma c \cdot (f_i^*(v,w) - f_i(v,w))}{\mathcal{B}}. \tag{3.5}$$

The values for all variables (f, f_i^*, u, c, and $\mathcal{B}$) except the step size σ are fixed.

Furthermore, it is a convex function. That is, it has positive second derivative. The first and second derivatives are

$$\phi'(\sigma) = \sum_r \left(\alpha \frac{\partial \lambda_r(\sigma)}{\partial \sigma}\right) \exp(\alpha(\lambda_r(\sigma) - 1)) \tag{3.6}$$

and

$$\phi''(\sigma) = \sum_r \left(\alpha \frac{\partial \lambda_r(\sigma)}{\partial \sigma}\right)^2 \exp(\alpha(\lambda_r(\sigma) - 1)), \tag{3.7}$$

where

$$\frac{\partial \lambda_{(v,w)}(\sigma)}{\partial \sigma} = \frac{f_i^*(v,w) - f_i(v,w)}{u(v,w)}$$

and

$$\frac{\partial \lambda_c(\sigma)}{\partial \sigma} = \frac{c \cdot (f_i^*(v,w) - f_i(v,w))}{\mathcal{B}}.$$

Since the second derivative is positive, the function's minimum occurs at either endpoint of the range $[\sigma_{min}, 1]$ or at exactly one point in the interior.

To minimize the potential function, we considered four different methods: using the fixed step size σ_{min} suggested by theory and computing an optimal step size using the golden search, Brent's, and the Newton-Raphson methods. After describing these methods, we present experimental comparisons.

We could use any method that requires only potential function evaluations at various step sizes. We considered both the golden search method and Brent's method [PFTV88]. The golden search method is based on the provably optimal Fibonacci search method [AW66]. After bracketing the minimum, the bracket's size is repeatedly reduced by the reciprocal of the golden ratio until a small enough bracket is found. Brent's method uses inverse parabolic interpolation to determine the new interval. Given a bracket and a point inside the bracket, a parabola is fit to the points, and the parabola's minimum is determined. A new bracket and interior point are determined. If the method determines the parabolic interpolations are not converging, it falls back upon the golden search method. Since our convex function $\phi(\sigma)$ has a nice shape, Brent's method is likely to perform well.

Computing the derivatives requires just a few more multiplications and additions per choice of σ so methods using these could also prove practical ways to minimize the potential function. We considered the Newton-Raphson method [PFTV88, Rap90]. To find the zero of a function, the method computes a series of points hopefully converging to the zero. Given an initial point, the method approximates the function using a tangent line to the function at the given point. The intersection of the tangent line with the x-axis determines a new point, with the process repeating. Thus, each iteration requires evaluating the function and its derivative at a given point. To use this root-finding method to minimize $\phi(\sigma)$, we differentiate the potential function with respect to σ to obtain a function with a zero. Thus, each iteration of the Newton-Raphson method requires computation of the first and second derivatives of the potential function.

Computing the derivatives (3.6) and (3.7) requires just a few more multiplications and additions per choice of σ than just computing the potential using (3.3). Computing the latter requires computing λ_r which, in turn, requires knowing $\frac{\partial \lambda_r(\sigma)}{\partial \sigma}$. These are independent of the step size so they can be precomputed. Thus, the additional work of computing the first and second derivatives consists of multiplying by the partial derivatives and their squares.

In practice, the Newton-Raphson method outperforms Brent's method despite its additional computation. Table 3.1 shows experimental results. For the problem instances we considered, Newton's method almost always require less total computation time even when using Brent's method required fewer minimum-cost flow computations. This is because Brent's method usually considered more possible step sizes than the Newton-Raphson method.

Even though it requires much more computation per step size, using the Newton-Raphson reduces the total running time by two orders of magnitude compared with using the fixed theoretical step size. See Table 3.2. As the accuracy increases, the reduction in running time for the Newton-Raphson method increases. As expected, the decrease in the number of minimum-cost flow computations is even greater. Thus, we were able to take advantage of the simple structure of our potential function to quickly compute first and second derivatives. Using a more complicated potential function may increase the time to compute the derivatives, eliminating any time savings.

3.3 Choosing α

The algorithm's performance depends on the value of α. The larger its value, the more running time the algorithm requires. Unfortunately, α must be large enough to produce an ϵ-optimal flow. There are two different theoretical explanations for α that can be used to develop two different heuristics: One is based on the potential function's value. The other uses the weak duality inequalities (3.1). We experimental evaluated heuristics based on both.

Karger and Plotkin [KP95b] choose α so that, when the potential function is less than a constant factor of its minimum, the flow is ϵ-optimal.

Plotkin, Shmoys, and Tardos [PST95] use the weak duality inequalities

$$\lambda' \sum_r y_r \geq \sum_{\text{comm. } i} (y^t A') f_i(\lambda'_A) \geq \sum_{\text{comm. } i} (y^t A') f_i^*(\lambda'_A). \tag{3.1}$$

upon which we base a different heuristic. Left to right, the terms represent the product

	time (seconds)			number of MCFs		
problem	Newton	Brent	ratio	Newton	Brent	ratio
Multicommodity Minimum-Cost Flow Problems						
multigrid-032-128-0080	41	51	1.24	1659	1659	1.00
multigrid-064-128-0160	509	582	1.14	3588	3588	1.00
rmfgen-7-10-020	258	477	1.85	17781	19714	1.10
rmfgen-7-10-240	1238	1534	1.24	103522	100177	0.96
rmfgen-7-12-240	1516	2201	1.45	103411	113973	1.10
rmfgen-7-14-020	511	576	1.13	21885	22181	1.01
rmfgen-7-14-040	822	904	1.10	38676	36570	0.94
rmfgen-7-14-080	1815	2121	1.17	90075	88540	0.98
rmfgen-7-14-160	1887	2218	1.17	100673	101337	1.00
rmfgen-7-14-240	2149	2853	1.33	122132	134399	1.10
rmfgen-7-14-320	2125	2700	1.27	120958	122819	1.01
rmfgen-7-16-240	2987	4072	1.36	146695	153053	1.04
Concurrent Flow Problems						
GTE	24.3	30.0	1.23	31044	30238	0.97
multigrid-032-128-0080	22.3	20.0	0.90	3555	2923	0.82
multigrid-064-128-0160	184.0	202.8	1.10	5772	5772	1.00
rmfgen-7-10-020	12.7	12.1	0.95	1540	1240	0.81
rmfgen-7-10-240	56.6	60.3	1.06	6069	6039	1.00
rmfgen-7-12-240	68.4	73.1	1.07	6342	6351	1.00
rmfgen-7-14-020	14.4	16.6	1.15	1060	1060	1.00
rmfgen-7-14-040	38.5	44.3	1.15	3315	3315	1.00
rmfgen-7-14-080	58.6	64.6	1.10	4940	4940	1.00
rmfgen-7-14-160	66.0	86.4	1.31	5086	6713	1.32
rmfgen-7-14-240	81.5	88.5	1.09	6267	6232	0.99
rmfgen-7-14-320	99.0	115.9	1.17	7705	8584	1.11
rmfgen-7-16-240	99.1	105.7	1.07	6653	6633	1.00
tripartite-04-16	5.9	6.0	1.02	789	728	0.92
tripartite-08-16	10.1	15.4	1.53	599	795	1.33
tripartite-16-16	32.8	39.6	1.21	848	912	1.08
tripartite-32-16	110.9	157.2	1.42	976	1104	1.13
tripartite-64-16	337.0	360.2	1.07	976	976	1.00

Table 3.1: Newton-Raphson method outperforms Brent's method for most problems. All problems solved to 1% accuracy. Multicommodity minimum-cost flow data obtained on a Sun Enterprise 3000, and concurrent flow data obtained on a Sun UltraSparc-2.

problem	ϵ	time (seconds) Newton	time (seconds) fixed	time (seconds) ratio	number of MCFs Newton	number of MCFs fixed	number of MCFs ratio
Multicommodity Minimum-Cost Flow Problems							
multigrid-008-016-100	0.01	3.0	95.3	31.6	1375	77936	56.7
rmfgen-4-8-16	0.12	0.7	7.9	11.5	238	4789	20.1
rmfgen-4-8-16	0.06	0.9	45.4	52.2	294	28534	97.1
rmfgen-4-8-16	0.03	2.5	180.6	70.8	878	114390	130.3
rmfgen-4-8-16	0.01	11.2	1334.2	119.4	4102	879963	214.5
rmfgen-4-12-20	0.01	63.9	3842.7	60.1	13261	1361037	102.6
rmfgen-7-10-20	0.01	257.5	15202.9	59.0	17781	1683061	94.7
Concurrent Flow Problems							
GTE	0.01	24.3	837.2	34.4	31044	1401552	45.1
multigrid-032-128-0080	0.01	22.3	34.2	1.5	3555	6083	1.7
multigrid-064-128-0160	0.01	184.0	12028.5	65.3	5772	597948	103.5
rmfgen-7-10-240	0.04	45.1	463.1	10.2	4605	49959	10.8
rmfgen-7-10-240	0.02	50.4	708.5	14.0	5307	80703	15.2
rmfgen-7-10-240	0.01	56.6	1015.0	17.9	6069	123159	20.3
rmfgen-7-16-240	0.01	99.1	1388.5	14.0	6653	110772	16.6
tripartite-16-16	0.04	15.5	468.8	30.2	409	10685	26.1
tripartite-16-16	0.02	22.6	810.0	35.8	594	15632	26.3
tripartite-16-16	0.01	32.8	1328.2	40.4	848	22747	26.8

Table 3.2: Computing an optimal step size reduces the running time and number of minimum-cost flow (MCF) computations by up to two orders of magnitude. Multicommodity minimum-cost flow data obtained on a Sun Enterprise 3000, and concurrent flow data obtained on a Sun UltraSparc-2.

		number of MCFs			final α		
problem	ϵ	potlLB	noChange	ratio	potlLB	noChange	ratio
GTE	0.10	2464	2561	0.96	39.3186	29.3186	1.34
GTE	0.05	7190	4396	1.64	87.3186	41.3186	2.11
GTE	0.02	20009	14644	1.37	217.319	55.3186	3.93
sm18	0.10	28655	28655	1.00	59.8016	59.8016	1.00
ngk70	0.05	689	689	1.00	19.4076	19.4076	1.00
ngk70	0.01	919	2184	0.42	189.408	33.4076	5.67
rmf500k60	0.05	1824	2806	0.65	137.473	45.4727	3.02

Table 3.3: Increasing α when the potential function's value becomes too small (potlLB) does not yield consistent improvement over increasing its value only when no commodities' flows change (noChange). Data obtained on a Pentium II computer for concurrent flow problems.

of the congestion and the potential function, the cost of the multiflow, and the sum of the single-commodity minimum-cost flows. The product of the inequalities' gaps bounds the difference between the potential function and optimal flows. The algorithm's improvement is proportional to the size of the right gap, i.e., the difference between a commodity's current and minimum costs. Increasing α decreases the left gap's size. Choosing α too large, however, can impede progress because progress is proportional to the step size σ which itself depends on how closely the potential function's linearization approximates its value. Thus, larger α reduces the step size.

Our heuristic based on the dual inequality gaps attempts to balance the left and right gaps. More precisely, it chooses α dynamically to ensure the ratio of inequalities

$$\frac{(\lambda \sum_r y_r / \sum_{\text{comm. } i} C_i(\lambda_A)) - 1}{(\sum_{\text{comm. } i} C_i(\lambda_A) / \sum_{\text{comm. } i} C_i^*(\lambda_A)) - 1}$$

remains balanced. We increase α by factor β if the ratio is larger than 0.5 and otherwise decrease it by γ. After limited experimentation, we decided to use the golden ratio for both β and γ.

The heuristic of starting with a small α and increasing it when the potential function's value is too small experimentally failed to decrease significantly the running time. We considered a heuristic (denoted "potlLB") of increasing α whenever the potential function was less than $5m$ or when no commodity's flow was rerouted during one round robin through the commodities. We compared this with the very conservative policy (denoted "noChange") of increasing α whenever no commodity's flow was rerouted during one round robin. See

		number of MCFs			final α		
problem	ϵ	adaptive	fixed	ratio	adaptive	fixed	ratio
GTE	0.01	2256	15723	6.97	227.4	666.3	2.93
multigrid-032-128-80	0.01	1659	1277	0.77	28.4	108.3	3.81
rmfgen-7-10-20	0.01	3394	10271	3.03	112.9	291.9	2.59
rmfgen-7-10-240	0.10	4575	6139	1.34	46.4	87.6	1.89
rmfgen-7-10-240	0.05	6966	22579	3.24	86.5	175.2	2.02
rmfgen-7-10-240	0.03	16287	53006	3.25	125.0	291.9	2.33
rmfgen-7-14-160	0.03	18221	64842	3.56	113.3	303.4	2.68

Table 3.4: Adaptively choosing α usually requires fewer minimum-cost flow (MCF) computations than using the theoretical, fixed value of α for multicommodity minimum-cost flow problems.

Table 3.3 for experimental data. Neither scheme consistently decreases the number of minimum-cost flow computations. In fact, for the same problem and various accuracies, the winning scheme varies. Neither scheme was as successful as the one based on the dual inequalities.

Using the adaptive heuristic based on the dual inequalities rather than using the fixed theoretical value of $\ln(3m)/\epsilon$ [KP95b] usually decreases the running time by a factor of between two and six. See Table 3.4. Furthermore, the α values are frequently much lower. Besides reducing the running time, using the adaptive scheme has another advantage. During the first few iterations, initial congestions can be large. Exponentiating the product of these congestions and a large α easily yields values that exceed the range of floating-point numbers. Thus, even if one uses the fixed theoretical α value, one should start with a smaller initial value until the congestions become small enough.

We developed the heuristic and chose the parameters for multicommodity minimum-cost flow problems. Using the same heuristic for concurrent flow problems usually increases the number of minimum-cost flow computations by about fifty percent compared with using the fixed theoretical value. See Table 3.5. Once again the multigrid problems are easily solved so the adaptive scheme hinders progress. The GTE instance, which is solved twelve times faster under the adaptive scheme, illustrates that choosing α carefully is important.

In Table 3.6, we explore the tradeoffs between all combinations of adaptive vs. fixed α scheme and optimal vs. fixed step size for concurrent flow problems. For all problem instances, using the optimal step size is better than using the fixed theoretical step size. If a fixed step size of $\epsilon/(4\alpha)$ is used, the adaptive scheme to choose α performs better for

		number of MCFs			final α		
problem	ϵ	adaptive	fixed	ratio	adaptive	fixed	ratio
GTE	0.01	31044	388415	12.51	66.4	659.3	9.93
multigrid-032-128-0080	0.01	3555	747	0.21	202.6	297.2	1.47
multigrid-064-128-0160	0.01	5772	1716	0.30	198.4	414.3	2.09
rmfgen-7-10-240	0.04	4605	2893	0.63	46.4	62.6	1.35
rmfgen-7-10-240	0.02	5307	4357	0.82	75.4	428.8	5.69
rmfgen-7-10-240	0.01	6069	4357	0.72	122.4	759.7	6.21
rmfgen-7-16-240	0.01	6653	5097	0.77	122.4	532.9	4.35
tripartite-16-16	0.04	409	306	0.75	46.4	123.9	2.67
tripartite-16-16	0.02	594	498	0.84	198.4	493.0	2.48
tripartite-16-16	0.01	848	626	0.74	321.4	989.9	3.08

Table 3.5: Adaptively choosing α usually requires more minimum-cost flow (MCF) computations than using the theoretical, fixed value of α for concurrent flow problems, but it does decrease the α values.

		number of MCFs					
		adaptive α			fixed α		
		optimal	fixed		optimal	fixed	
problem	ϵ	step	step	ratio	step	step	ratio
GTE	0.04	4272	94848	22.20	39526	692574	17.52
GTE	0.02	11952	407168	34.07	116566	3449078	29.59
GTE	0.01	31044	1401552	45.15	388415	14159774	36.46
multigrid-032-128-0080	0.01	3555	6083	1.71	747	11333	15.17
multigrid-064-128-0160	0.01	5772	597948	103.59	1716	137766	80.28
rmfgen-7-10-240	0.08	3843	32391	8.43	2893	112221	38.79
rmfgen-7-10-240	0.04	4605	49959	10.85	2893	144429	49.92
rmfgen-7-10-240	0.02	5307	80703	15.21	4357	202257	46.42
rmfgen-7-10-240	0.01	6069	123159	20.29	4357	288633	66.25
rmfgen-7-14-020	0.01	1060	10260	9.68	820	15204	18.54
rmfgen-7-14-040	0.01	3315	23042	6.95	1582	45475	28.75
rmfgen-7-14-080	0.01	4940	84284	17.06	2964	123198	41.56
rmfgen-7-14-160	0.01	5086	123437	24.27	3145	215725	68.59

Table 3.6: For concurrent flow problems, choosing the optimal step size obviates most of need to choose appropriate values of α.

all problem instances except multigrid-064-128-0160. When computing optimal step sizes, the theoretical link between the step size and α is broken, and the main algorithmic role of the latter is to control the magnitude of the dual variables. Experimentally, none of the problem instances except GTE depend significantly on the value of α although using the fixed value for α usually requires fewer minimum-cost flow computations.

Further experimental and theoretical work to determine an optimal scheme to determine the value of α is needed. Theoretical analysis indicates the running time strongly depends on its value. Experimentally, the use of an optimal step size reduces its importance to, apparently, less than one order of magnitude except for a few problem instances, e.g., GTE.

3.4 Restarting The Minimum-Cost Flow Subroutine

Theoretically, the algorithm can use any minimum-cost flow subroutine. In practice, the repeated evaluation of single-commodity problems with similar arc costs and capacities favor an implementation that can take advantage of *restarting* from a previous solution. We show that using a primal network simplex implementation allows restarting and thereby reduces the running time by one-third to one-half.

To solve a single-commodity problem, the primal simplex algorithm repeatedly *pivots* arcs into and out of a spanning tree until the tree has minimum cost. (See, e.g., [Gri86] or [AMO93, Chapter 11].) Each pivot maintains the flow's feasibility and can decrease its cost. The simplex algorithm can start with any spanning tree and any feasible flow. Since the cost and capacity functions do not vary much between MCF calls for the same commodity, we can speed up the computation using the previously-computed spanning tree. This requires $O(n)$ storage per commodity.

For the flow, there are two natural choices: restarting from the previously computed minimum-cost flow or using the current flow. Using the previously-computed minimum-cost flow requires $O(m)$ additional storage per commodity. Moreover, it is sometimes unusable because it is frequently infeasible with respect to the capacity constraints. For this to occur three things must happen:

- during the previous minimum-cost flow computation for this commodity, the minimum-cost flow had larger arc congestion than the commodity's flow,
- for the previous computation, the optimal step size was less than one, and

		time (seconds)			number of MCFs		
problem instance	ϵ	restart	no rest.	ratio	restart	no rest.	ratio
GTE	0.001	35310	56830	1.61	43095	50444	1.17
rmfgen-7-10-020	0.01	87570	180130	2.06	17544	17544	1.00
rmfgen-7-10-240	0.01	436550	825420	1.89	101199	105012	1.04
rmfgen-7-12-240	0.01	613430	993500	1.62	113883	104940	0.92
rmfgen-7-14-020	0.01	174650	314070	1.80	20991	20953	1.00
rmfgen-7-14-040	0.01	293300	496140	1.69	39663	38259	0.96
rmfgen-7-14-080	0.01	645140	1070890	1.66	88293	85687	0.97
rmfgen-7-14-160	0.01	697260	1197790	1.72	102685	103709	1.01
rmfgen-7-14-240	0.01	837080	1395920	1.67	126164	125156	0.99
rmfgen-7-14-320	0.01	842820	1313730	1.56	127519	118178	0.93
rmfgen-7-16-240	0.01	1206910	2013800	1.67	162725	158600	0.97
multigrid-032-128-0080	0.01	21160	37510	1.77	1659	1659	1.00
multigrid-064-128-0160	0.01	274520	800760	2.92	3588	4836	1.35
tripartite-04-16	0.03	42300	47160	1.11	8144	85281	1.05

Table 3.7: Restarting the minimum-cost flow computations from the current flow and the previous spanning tree usually reduces the running time for multicommodity minimum-cost flow problems by at least one-third.

- the multiflow's arc congestion must have decreased since the previous computation.

In contrast, using the current flow requires no additional storage, and this flow is always feasible.

Even when restarting minimum-cost flow subroutines, occasional computations must be started from an arbitrary basis and flow. This occurs for each commodity's first minimum-cost flow. It can also occur when computing lower bounds for the multicommodity minimum-cost flow problem. The lower bound computation (3.2) computes minimum-cost flows with respect to arc capacities $1u$, but a commodity's current flow may have congestion greater than one.

Restarting a commodity's minimum-cost flow computations from its current flow and its previous spanning tree rather than an arbitrary spanning tree and flow usually reduces the running time by a factor of about 1.5 to 2 on the problem instances we tried. See Table 3.7. Because minimum-cost flows are not unique, the number of MCF computations differ, but the difference of usually less than five percent.

Most of the time savings occurs because restarting experimentally requires a very small number of pivots. Fairly quickly, the number of pivots per MCF iteration becomes very

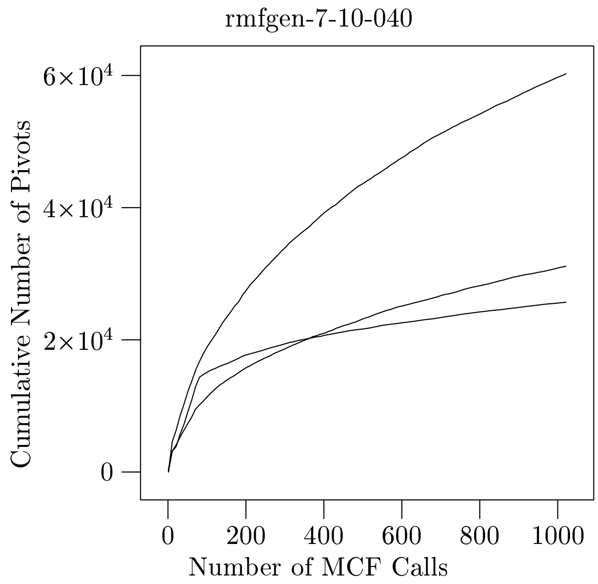

Figure 3.1: The cumulative number of pivots as a function of the number of MCF calls for three different commodities in the rmfgen-7-10-040 multicommodity minimum-cost flow instance.

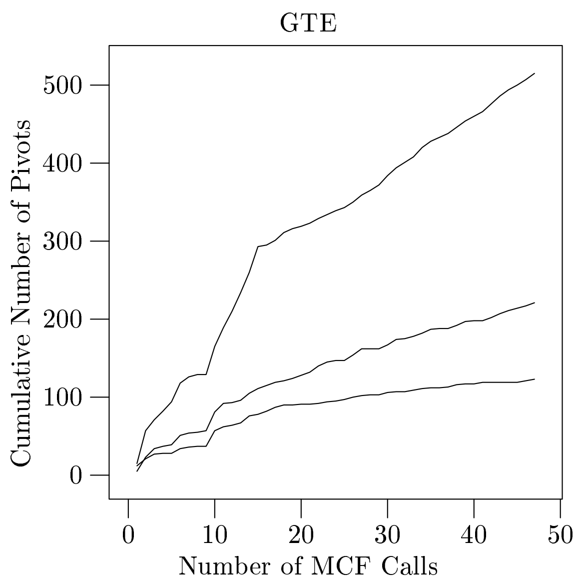

Figure 3.2: The cumulative number of pivots as a function of the number of MCF calls for three different commodities in the GTE multicommodity minimum-cost flow instance.

problem instance	ϵ	no. of MCFs	initializations	other	percentage
GTE	0.001	43095	48	0	0.11
rmfgen-7-10-020	0.01	17544	20	14	0.19
rmfgen-7-10-240	0.01	101199	183	2	0.18
rmfgen-7-12-240	0.01	113883	187	2	0.17
rmfgen-7-14-020	0.01	20991	20	28	0.23
rmfgen-7-14-040	0.01	39663	39	20	0.15
rmfgen-7-14-080	0.01	88293	76	17	0.11
rmfgen-7-14-160	0.01	102685	137	3	0.14
rmfgen-7-14-240	0.01	126164	188	3	0.15
rmfgen-7-14-320	0.01	127519	232	2	0.18
rmfgen-7-16-240	0.01	162725	200	4	0.13
multigrid-032-128-0080	0.01	1659	79	0	4.76
multigrid-064-128-0160	0.01	3588	156	0	4.35
tripartite-04-16	0.03	8144	16	11	0.33

Table 3.8: The number of minimum-cost flows that must be computed from an arbitrary basis and flow is small. See the text for an explanation of the column headings. (Data for multicommodity minimum-cost flow problems.)

small. See Figures 3.2 and 3.1. For the 2121-arc rmfgen-7-10-040 instance, the average number of pivots are 27, 13, and 7 for the three commodities shown. Less than two percent of arcs served as pivots. For the 260-arc GTE problem, the average numbers are 8, 3, and 1, i.e., at most three percent of the arcs.

As noted above, even when restarting MCF computations, occasional computations must be started from an arbitrary basis because the commodity's current flow overflows the subroutine's arc capacities or the subroutine has not previously computed any flow. Table 3.8 illustrates how infrequently this happens. The left data column indicates the total number of minimum-cost flow computations, while the next column indicates the number of subroutine initializations, which equals the number of commodities. The next column indicates the small number of times the current flow exceeds the subroutine's arc capacities, while the rightmost column indicates the percentage of the minimum-cost flows that required using an arbitrary basis and flow.

3.5 Data Structures Sensitive to Flow Changes

In the previous section, we took advantage of the repeated evaluation of single-commodity problems with similar arc costs and capacities by restarting from the previous basis and

problem instance	ϵ	change-insensitive	change-sensitive	ratio
GTE	0.01	8.4	4.7	1.80
multigrid-032-064-0100	0.01	120.3	81.7	1.47
rmfgen-7-10-020	0.01	922.7	488.6	1.89

Table 3.9: The running time decreases significantly using data structures and algorithms with running time proportional to the number of arcs with nonzero flow differences. Data obtained on a Pentium Pro for these multicommodity minimum-cost flow problem instances.

the current flow. In this section, we modify our data structures and algorithms to take advantage of the observation that most minimum-cost flow computations change few arcs' flows.

A commodity's flow and its minimum-cost flow usually differ on few arcs for two reasons. One is that the set of reasonable length paths connecting a commodity's source and sink vertices use only a small fraction of the graph. For example, when traveling between Sydney and Perth, one need not usually consider paths through Lima. Secondly, as the algorithm progresses, the difference between the current flow and the minimum-cost flows is experimentally a few highly utilized arcs. The flows on the other arcs do not change appreciably.

Instead of recording the entire minimum-cost flow with m arcs, we store only nonzero differences between the minimum-cost flow and the current flow. We tag each nonzero difference with the associated arc's number. In the worst case, storing the flow differences and the associated arc numbers requires twice as much space as storing just the flow. Formulae 3.4 and 3.5 indicate the step size computation's running time is dependent on the number of arcs with nonzero flow differences, not the total number of arcs. The running time is further reduced because, once the desired step size is determined fewer dual variables need updating.

The modified techniques have running time dependent on the number of arcs with nonzero flow differences, not m. This simple data structure and algorithmic change experimentally reduces the running time by a factor of between 1.5 and 2. See Table 3.9.

3.6 Grouping Commodities

Since single-commodity minimum-cost flow computations dominate the algorithm's running time, grouping together commodities which can be computed using the same minimum-cost flow computation can significantly decrease the required time. In this section, we describe how we group commodities.

We can group together commodities with the same source or sink vertices. Each grouped commodity can have multiple sources or multiple sinks, each with its own supply or demand, but minimum-cost flow problems can handle these. When the algorithm completes, the flow can be decomposed to satisfy the original commodities. (For the theoretical proof of these claims, see [Ste92, Lemma 2.2.1].)

Grouping guarantees the number of commodity groups is at most the number n of vertices. This reduces the algorithm's space requirements because the multiflow has size proportional to the number of commodities. Also, the minimum-cost flow subroutine stores one spanning tree per commodity.

One can use a minimum vertex cover in a bipartite graph to compute the minimum number of commodity groups. On one side of the bipartite graph are n vertices representing all source vertices. The other side has n vertices representing all sink vertices. Each commodity is represented by an arc connecting its source and sink vertices. Each selected vertex in the vertex cover represents a commodity group. Each arc, i.e., commodity, associates itself with its chosen vertex, choosing arbitrarily if both of its endpoints are selected. Since it is a vertex cover, every arc, i.e., commodity, participates in some group. Since it is minimal, the smallest number of commodity groups is required. A minimum vertex cover in a bipartite graph is equivalent a maximum cardinality matching in the same graph (e.g., see [AMO93, Theorem 6.9]).

A matching in a bipartite graphs can easily be computed using a maximum flow or even a minimum-cost flow so our implementation could easily have grouped commodities this way. Instead, we greedily grouped commodities with the same source vertices, deferring optimal grouping to a production version of the code. For our problem instances, this greedy grouping requires at most one-fifth more commodity groups than the optimal grouping. See Table 3.10.

Experimentally using grouped commodities reduces the running time by approximately the ratio of the number of commodities to the number of commodity groups. See Table 3.11.

problem instance	k_0	grouping sources only	optimal grouping	ratio ratio
GTE	585	48	47	1.02
rmfgen-4-12-100	100	74	69	1.07
rmfgen-4-12-200	200	134	114	1.18
rmfgen-4-12-250	250	147	129	1.14
rmfgen-4-12-300	300	173	148	1.17
rmfgen-4-12-400	400	202	169	1.20
rmfgen-4-12-500	500	215	185	1.16
rmfgen-4-12-600	600	227	201	1.13
rmfgen-4-12-700	700	166	157	1.06

Table 3.10: Grouping commodities according to common sources is close to the minimum possible number of commodity groups.

	commodities			time (seconds)			number of MCFs		
problem instance	k_0	k	ratio	k_0	k	ratio	k_0	k	ratio
multigrid-032-128-0040	40	40	1.00	17.0	16.2	1.05	680	680	1.00
multigrid-032-128-0080	80	79	1.01	28.6	27.3	1.05	1040	1027	1.01
multigrid-032-128-0160	160	152	1.05	52.9	48.6	1.09	2080	1976	1.05
multigrid-032-128-0240	240	221	1.09	77.7	69.5	1.12	3120	2873	1.09
multigrid-032-128-0320	320	275	1.16	94.9	83.1	1.14	4160	3575	1.16
multigrid-032-128-0400	400	333	1.20	166.0	144.1	1.15	6846	5707	1.20
multigrid-032-128-0480	480	384	1.25	203.7	156.8	1.30	10106	6550	1.54
rmfgen-7-16-020	20	20	1.00	724.7	724.2	1.00	20260	20260	1.00
rmfgen-7-16-040	40	40	1.00	1446.0	1437.5	1.01	46615	46615	1.00
rmfgen-7-16-080	80	74	1.08	3097.7	3189.0	0.97	106390	108013	0.98
rmfgen-7-16-160	160	139	1.15	5926.6	5136.5	1.15	217760	183063	1.19

Table 3.11: Grouping commodities reduces the running time in an almost linear fashion. $k_0$denotes the original number of commodities, and k denotes the number of grouped commodities. These multicommodity minimum-cost flow problems were solved to 1% accuracy.

Previous researchers [LSS93, Sch91] found similar results.

3.7 Choosing Commodities to Reroute

Since single-commodity minimum-cost flow computations dominate the algorithm's running time, careful choice of which commodities to reroute could significantly decrease the required time. The choice of commodities has received extensive attention by theoreticians. In practice, there are three issues to explore: how much do individual commodities' potential function reductions vary during the algorithm's computation, can we predict which commodities permit the largest reduction, and do the predictions actually decrease the total running time.

Each iteration, the algorithm seeks to reduce the potential function as much as possible. Rerouting a commodity with a large difference between the costs of its current flow and its minimum-cost flow can possibly reduce the potential function significantly. Leighton et al. [LMP+95] reroute commodities with large cost differences. Deterministically deciding which commodity to improve can require one minimum-cost flow (MCF) computation per commodity before a "bad commodity" is rerouted. A randomized algorithm chooses commodities according to the cost differences, requiring only $O(\epsilon^{-1})$ MCFs before a bad commodity is found. Goldberg [Gol92] showed choosing the commodities uniformly randomly reduces the running time by a factor of ϵ^{-1}. Independently, Grigoriadis and Khachiyan [GK94] achieved the same result. Radzik [Rad95] showed a deterministic round robin ordering, i.e., fixing an order and repeatedly visiting each commodity according to that order, requires the same running time (up to logarithmic factors).

We now turn to evaluating which of these theoretical results is practical. First, we illustrate that the amount by which an individual commodity reduces the potential function varies during the algorithm's computation. See Figures 3.3 and 3.4. Here the problems were solved using round robin order. Each iteration, i.e., round, all the commodities are ranked according to their fraction of the reduction of the potential function. Percentiles are computed so the commodity with the largest reduction is in the 100th percentile and commodities with the lowest reduction are in the 0th percentile. Circle radii are proportional to the percentiles. Visiting all commodities with circles with a certain radius or larger during one iteration will yield some fixed fraction of the total potential reduction. The commodity with the largest radius had the greatest potential function reduction. Note the commodities

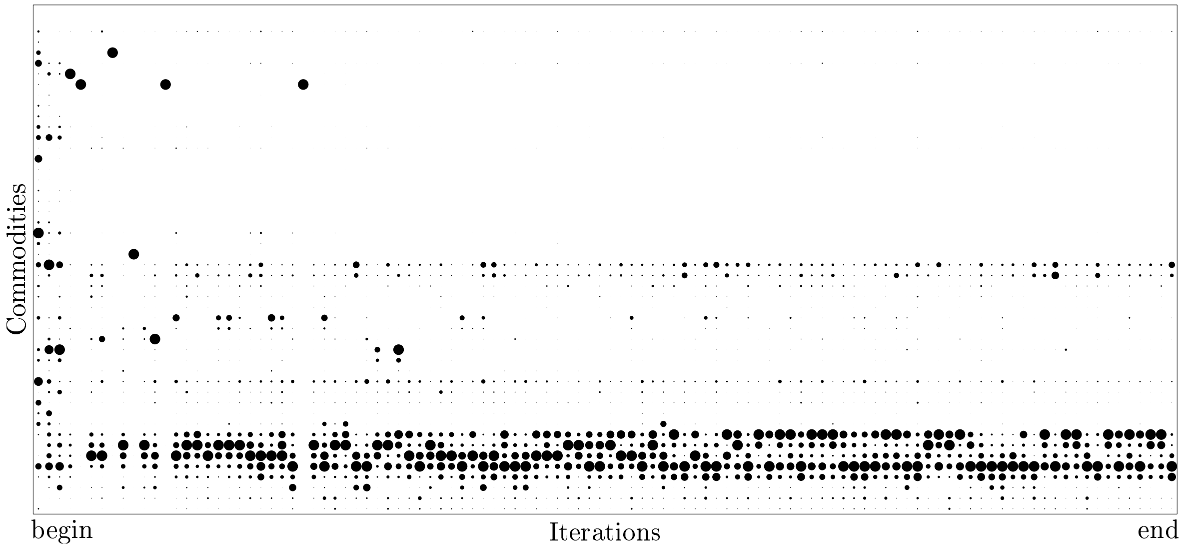

Figure 3.3: The potential function reduction from rerouting various commodities of concurrent GTE. A larger radius indicates the commodity's potential reduction was larger. See the text for a more precise description. To save space, only every fourth iteration is indicated.

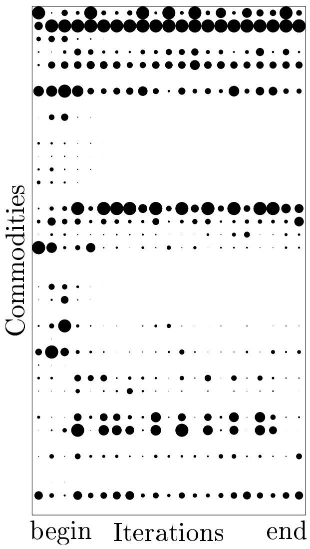

Figure 3.4: The potential function reduction from rerouting various commodities of concurrent rmfgen-7-14-040. A larger radius indicates the commodity's potential reduction was larger. See the text for a more precise description.

are arbitrarily ordered, and the total potential reduction per iteration can vary greatly as α changes.

Both figures indicate a subset of commodities permit the majority of the potential reduction. As the number of iterations increases, the subset of commodities with significant potential reduction does not change much per iteration. For GTE, it appears at least half of the commodities yield almost no reduction. Identifying these commodities could reduce the running time by at least a factor of two. The importance of individual commodities can vary per iteration, even alternating between important and less important. See, e.g., the topmost commodity of the second figure. A few commodities usually permit small reductions but occasionally permit larger reductions. For example, see the two commodities near the middle of the first figure.

Thus, prediction schemes to identify commodities permitting significant potential reduction need to identify the subset that relatively consistently permit significant reductions, make these predictions using more than just the previous iteration's reductions, and permit occasional inclusion of other commodities.

Leighton et al. [LMP$^+$95] reroute commodities with large cost differences so one possible prediction scheme would be to predict which commodities have large cost differences. (Doing so is computationally expensive because checking whether a commodity has large cost difference ostensibly requires computing a minimum-cost flow, but we will ignore this practicality issue for now.) Table 3.12 indicates the effect of using this idea, *assuming we know the exact cost differences*. That is, we have omitted the MCFs used to compute the cost differences. Given this exact knowledge, we visit the commodities with the largest cost differences and accounting for the specified fraction of the total cost difference. For example, for 75%, we visit commodities whose cost differences total 75% of the total cost difference. "exact" indicates only the commodity with the largest cost difference is visited. "RR" indicates visiting all commodities in round robin order. Useful minimum-cost flows indicate visited commodities for which rerouting reduced the potential function. Each useless MCF, a commodity's flow was not rerouted because doing so did not decrease the potential function.

MULTIGRID instances are easy for the algorithm to solve so choosing important commodities has almost no influence. For RMFGEN instances, rerouting a subset of the commodities each iteration measurably reduced the total number of MCFs. For two instances, 75% was the best fraction while for the other approximately 50% was better. In all cases,

problem instance	fraction	useful MCFs	useless MCFs	total MCFs
multigrid-008-016-0100	exact	495	0	495
multigrid-008-016-0100	1%	495	0	495
multigrid-008-016-0100	25%	486	20	506
multigrid-008-016-0100	50%	489	23	512
multigrid-008-016-0100	75%	492	29	521
multigrid-008-016-0100	100%	425	70	495
multigrid-008-016-0100	RR	426	69	495
rmfgen-4-8-16	exact	1784	0	1784
rmfgen-4-8-16	1%	1784	0	1784
rmfgen-4-8-16	25%	1782	0	1782
rmfgen-4-8-16	50%	1750	0	1750
rmfgen-4-8-16	75%	1809	0	1809
rmfgen-4-8-16	100%	2075	1373	3448
rmfgen-4-8-16	RR	1797	1159	2956
rmfgen-4-12-020	exact	8791	0	8791
rmfgen-4-12-020	1%	8791	0	8791
rmfgen-4-12-020	25%	9487	0	9487
rmfgen-4-12-020	50%	8995	0	8995
rmfgen-4-12-020	75%	7580	2	7582
rmfgen-4-12-020	100%	8564	2618	11182
rmfgen-4-12-020	RR	7885	2204	10089
rmfgen-7-16-080	25%	59630	0	59630
rmfgen-7-16-080	50%	56343	0	56343
rmfgen-7-16-080	75%	45772	0	45772
rmfgen-7-16-080	100%	56297	19337	75634
rmfgen-7-16-080	RR	59780	21223	81003

Table 3.12: Given perfect knowledge commodities' cost differences, the algorithm reroutes the commodities accounting for the given fraction of the total possible potential reduction. "exact" and "RR" indicate visiting only the commodity with the largest potential reduction and round robin ordering, respectively. These multicommodity minimum-cost flow problems were solved to 1% accuracy.

	time (seconds)			number of MCFs		
problem instance	RR	predict	ratio	RR	predict	ratio
multigrid-032-128-0020	4720	7790	1.65	140	221	1.58
multigrid-032-128-0040	9430	17540	1.86	280	486	1.74
multigrid-032-128-0080	18770	31180	1.66	711	890	1.25
multigrid-032-128-0160	33540	60680	1.81	1368	1769	1.29
multigrid-032-128-0240	54050	58720	1.09	2431	2452	1.01
multigrid-032-128-0320	65130	70100	1.08	3025	3050	1.01
multigrid-032-128-0400	98380	120270	1.22	4384	4585	1.05
multigrid-032-128-0480	118010	122330	1.04	5792	5078	0.88
rmfgen-7-16-020	981560	1085560	1.11	36833	26825	0.73
rmfgen-7-16-040	2175290	2516530	1.16	92809	69581	0.75
rmfgen-7-16-080	4576120	5206030	1.14	204791	137591	0.67
rmfgen-7-16-160	6950650	7706330	1.11	329148	217725	0.66
rmfgen-7-16-240	4702680	6695010	1.42	243471	198759	0.82
rmfgen-7-16-320	5953270	7886460	1.32	311910	224808	0.72

Table 3.13: The performance of the commodity prediction scheme versus round robin ("RR"). These multicommodity minimum-cost flow problems were solved to 1% accuracy.

visiting commodities accounting for 100% of the cost differences performed comparably to round robin. Visiting all the commodities and using round robin order differ because the former rearranges commodities' order and omits visiting commodities that have nonpositive cost difference when the cost difference is determined. After other commodities are rerouted, the cost difference may change.

Practical schemes that use predictions to determine which commodities to visit cannot use too computational time to make the predictions. For example, we should reroute every commodity for which we compute a minimum-cost flow (as long as it reduces the potential function). Thus, explicitly computing cost differences is unlikely to be beneficial. Furthermore restricting the scheme's computation is the observation that restarting minimum-cost flows reduces their computation time (Section 3.4).

We tried a variety of schemes to approximate commodities' cost differences. Even though we were able to more accurately predict which commodities to reroute, the increased overhead of computing these commodities increased the running time compared with a simple round robin scheme. The most successful scheme first computed minimum-cost flows for all the commodities, saving the flow vector. (We actually merged this computation with the lower bound computation, which also requires computing these MCFs.) Using the current cost vector, each commodity's cost difference is computed, the commodities are sorted according to increasing cost differences, and the algorithm attempts to reroute the commodities with the largest cost differences and accounting for fifty percent of the total difference. As long as at least one of these commodities changes, we repeat the ranking, sorting, and visiting. If none changes, the remaining commodities are visited until one changes. The loop ends if no commodity changes or a lower bound calculation is required because enough iterations have occurred. Table 3.13 shows the prediction scheme more accurately predicts commodities for all RMFGEN and some MULTIGRID instances. Unfortunately, the overhead of computing the cost differences increases the total running time compared with a simple round robin scheme. Thus, we concluded that visiting the commodities in a round robin regardless of their cost differences performs best.

3.8 The Cost Function

For a concurrent flow problem, using a gradient as the cost vector theoretically suffices. To solve the multicommodity minimum-cost flow problem, Plotkin, Shmoys, and Tardos [PST95] use shortest-paths as a basic subroutine. Karger and Plotkin [KP95b] decreased the running time by a factor of m/n using minimum-cost flow subroutines and adding a linear-cost term to the gradient to ensure each flow's arc cost is bounded. This change increases the ϵ-dependence of [PST95] by $1/\epsilon$ to ϵ^{-3}. Intuitively, a minimum-cost flow that minimizes arc congestion may have very large arc cost. A linear term proportional to the arc costs prevents this possibility.

In practice, no problem instance we tested required this linear term for convergence. Additionally, we tried multiplying the linear term by various constants and were unable to discern any pattern. In fact, using the linear-cost term increased the running time.

Chapter 4

Experimental Results

In the previous chapter, we differentiated among the various theoretically equivalent ideas, treating each choice in isolation. In this chapter, we experimentally evaluate the result of combining the best choices. We determine the algorithms' dependences on the approximation factor ϵ, the number k of commodities, and the problem size. We also give data illustrating the algorithms' execution, showing how the algorithms use their running time and how the congestions change during execution.

Data from both multicommodity minimum-cost flow and concurrent flow problems are presented. As we shall see, the presence of arc costs can significantly affect a problem instance's difficulty.

4.1 Dependence on the Approximation Factor ϵ

Our approximation algorithms yield ϵ-optimal flows. As described in Section 3.8, the theoretical dependence of Karger and Plotkin's multicommodity minimum-cost flow algorithm [KP95b] is ϵ^{-3}, while Grigoriadis and Khachiyan [GK96a] reduced the ϵ-dependence to ϵ^{-2}.

The multicommodity minimum-cost flow implementation exhibits various dependences smaller than the worst-case theoretical concurrent flow dependence of ϵ^{-2}. We believe the implementation's searching for an optimal step size and its regularly computing lower bounds decreases the dependence. Figures 4.1–4.4 show the number of minimum-cost flow computations as a function of the desired accuracy ϵ. Each line represents a problem instance solved to various accuracies. On a log-log scale, a line's slope represents the exponent

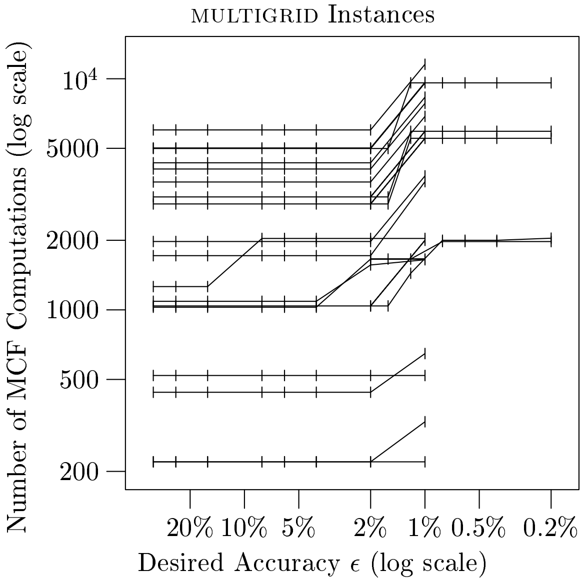

Figure 4.1: The number of minimum-cost flow computations as a function of ϵ for several multicommodity minimum-cost MULTIGRID instances.

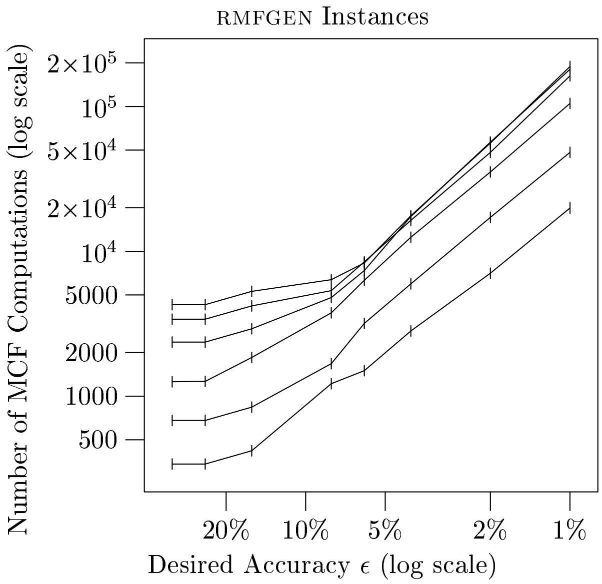

Figure 4.2: The number of minimum-cost flow computations as a function of ϵ for several multicommodity minimum-cost RMFGEN instances.

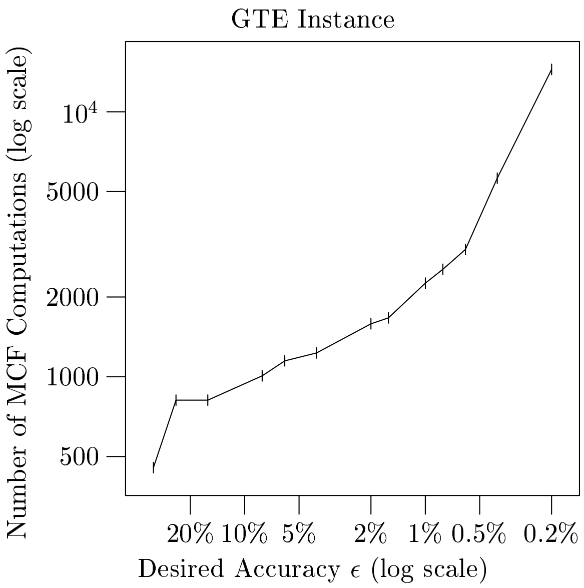

Figure 4.3: The number of minimum-cost flow computations as a function of ϵ for the multicommodity minimum-cost GTE instance.

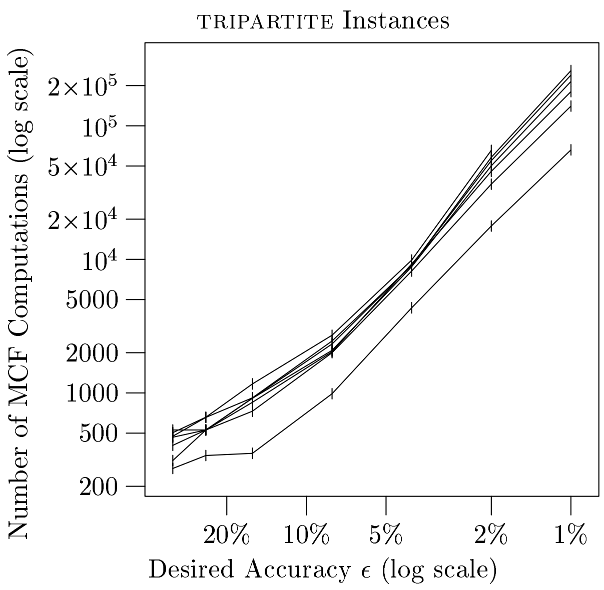

Figure 4.4: The number of minimum-cost flow computations as a function of ϵ for several multicommodity minimum-cost TRIPARTITE instances.

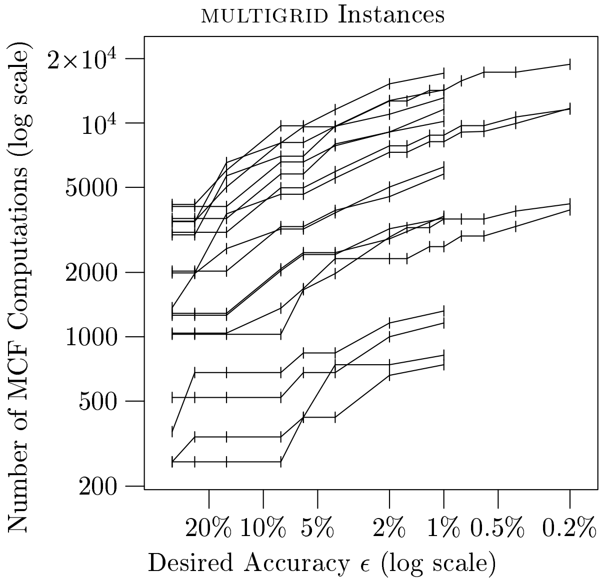

Figure 4.5: The number of minimum-cost flow computations as a function of ϵ for several concurrent flow MULTIGRID instances.

of $1/\epsilon$. For most MULTIGRID instances, we solved to an accuracy of 1%, but for five instances we solved to an accuracy of 0.2%. All these instances depend very little on the accuracy; our algorithm yields the same flows for several different accuracies. Intuitively, the grid networks permit so many different routes to satisfy a commodity that very few commodities need to share the same arcs. Our implementation is able to take advantage of these many different routes, while, as we will see in the next chapter, some linear-programming based implementations have more difficulty. For the RMFGEN problem instances, the dependence is proportional to $\epsilon^{-1.3}$. While the GTE problem instance was relatively easy to solve, the TRIPARTITE instances required time approximately proportional to $\epsilon^{-1.9}$. By construction, interframe arcs' varying arc costs cause contention for cheap, but not necessarily high capacity, arcs.

Compared with multicommodity minimum-cost flow, the concurrent flow instances of MULTIGRID and GTE concurrent flow instances are harder, while RMFGEN and TRIPARTITE are easier. See Figures 4.5–4.8. With a dependence approximately proportional to $\epsilon^{-0.4}$, MULTIGRID are still easy to solve, but the absence of arc costs means flow through the high capacity arcs connecting randomly chosen vertices is more likely. RMFGEN's and

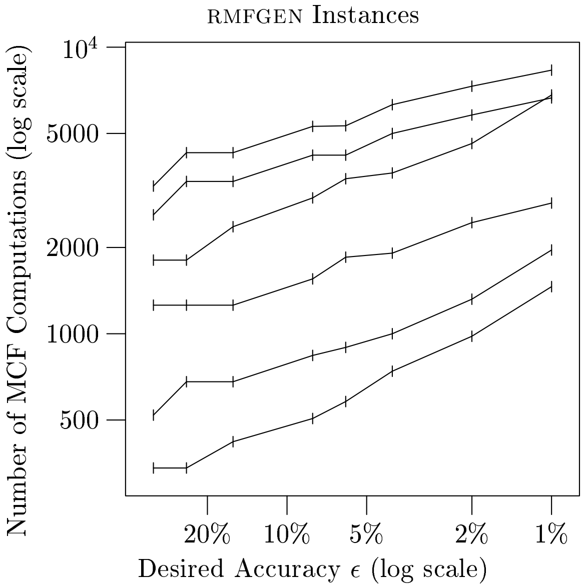

Figure 4.6: The number of minimum-cost flow computations as a function of ϵ for several concurrent flow RMFGEN instances.

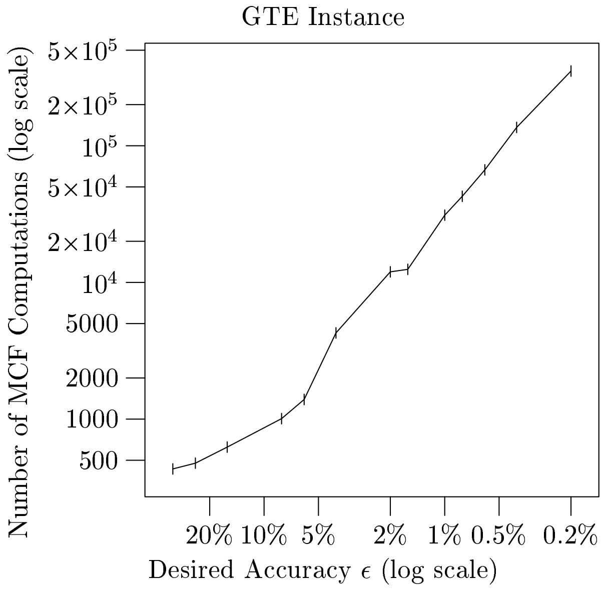

Figure 4.7: The number of minimum-cost flow computations as a function of ϵ for the concurrent flow GTE instance.

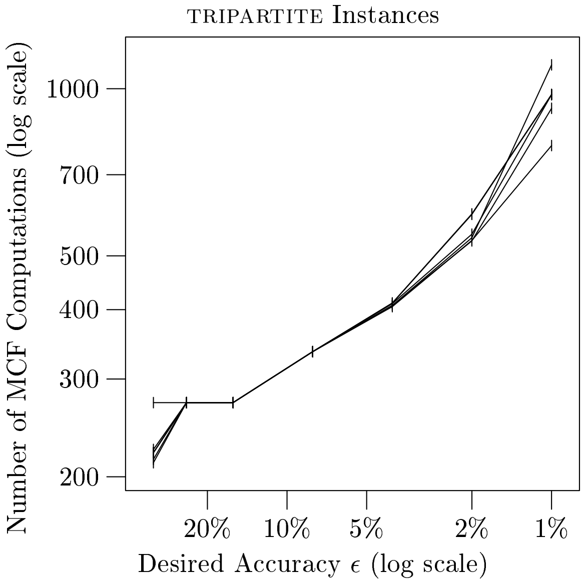

Figure 4.8: The number of minimum-cost flow computations as a function of ϵ for several concurrent flow TRIPARTITE instances.

TRIPARTITE's dependences are about the same ($\epsilon^{-0.4}$) and significantly smaller than for the minimum-cost versions. In both instances, the interframe arcs cause contention, but not having to route through only inexpensive arcs reduces the entire problems' difficulty.

Details of the Problem Instances We briefly indicate which instances were used. MULTIGRID instances include multigrid-032-064-010, multigrid-032-128-k for $k = 20$, 40, 80, 160, 240, 320, 400, and 480, and multigrid-064-128-k for the same k. RMFGEN instances included rmfgen-7-16-k for $k = 20$, 40, 80, 160, 240, and 320. TRIPARTITE instances included tripartite-a-16 for a equaling powers of two from 4 to 256.

4.2 Dependence on the Number k of Commodity Groups

The experimental number of minimum-cost flow computations and the running time of the implementation match the theoretical upper bounds. Theoretically, the algorithm performs $\tilde{O}(\epsilon^{-3}k)$ minimum-cost flow computations, as described in Section 2.5. These upper bounds (holding ϵ constant and ignoring logarithmic dependences) match a natural lower bound

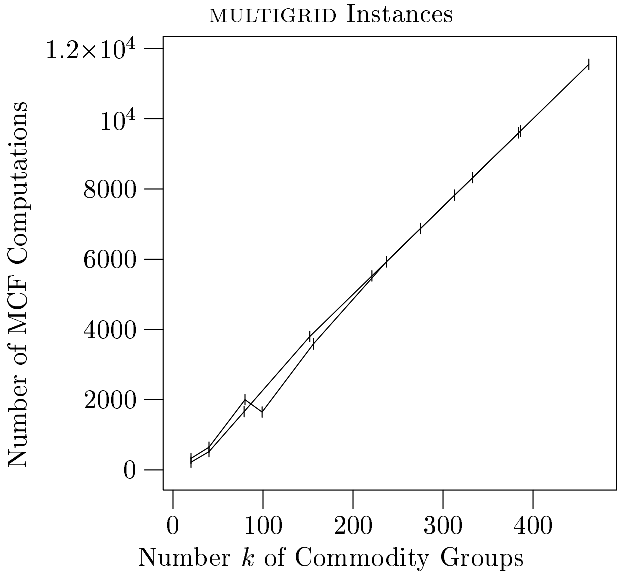

Figure 4.9: The number of minimum-cost flow computations as a function of the number k of commodities for several multicommodity minimum-cost MULTIGRID instances.

where the joint capacity constraints are ignored and the problem can be solved using k single-commodity minimum-cost flow problems. In practice, the implementation requires at most a linear (in k) number of minimum-cost flows.

Figures 4.9 and 4.10 show the numbers of minimum-cost flow computations as a function of the number k of commodity groups for multicommodity minimum-cost flow instances. Each line represents a fixed network with various numbers of commodity groups. The MULTIGRID figure shows a dependence of approximately $26k$ for two different networks. For the RMFGEN instances, the dependence is initially linear but flattens and even decreases. As the number of commodity groups increases, the average demand per commodity decreases because the demands are scaled so the instances are feasible in a graph with 60% of the arc capacities. Furthermore, the randomly distributed sources and sinks are more distributed throughout the graph reducing contention for the most congested arcs. The number of minimum-cost flows depends more on the network's congestion than on the instance's size so the lines flatten.

The comparable concurrent flow figures (Figures 4.11 and 4.12) show analogous behavior. The number of minimum-cost flow computations per commodity is approximately $38k$ and

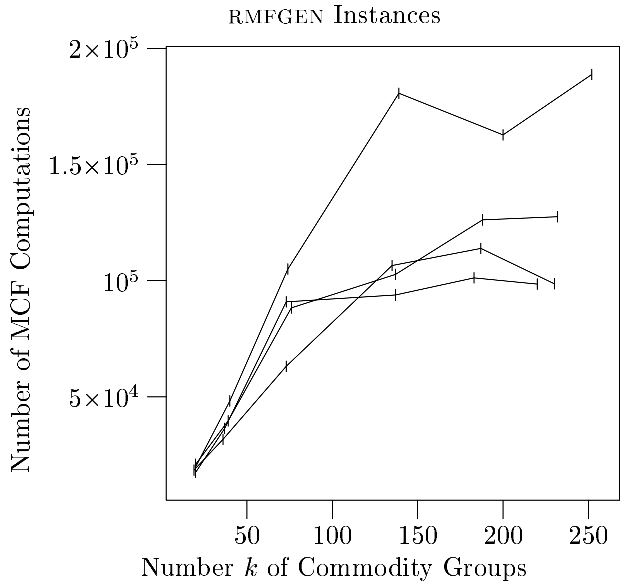

Figure 4.10: The number of minimum-cost flow computations as a function of the number k of commodities for several multicommodity minimum-cost RMFGEN instances.

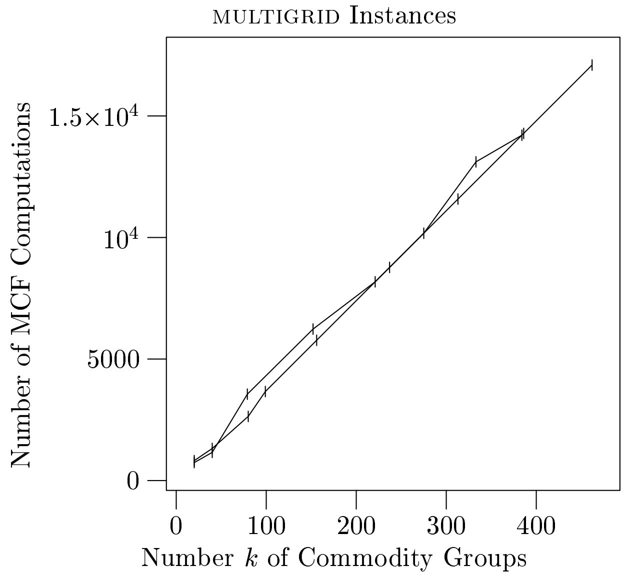

Figure 4.11: The number of minimum-cost flow computations as a function of the number k of commodities for several concurrent MULTIGRID instances.

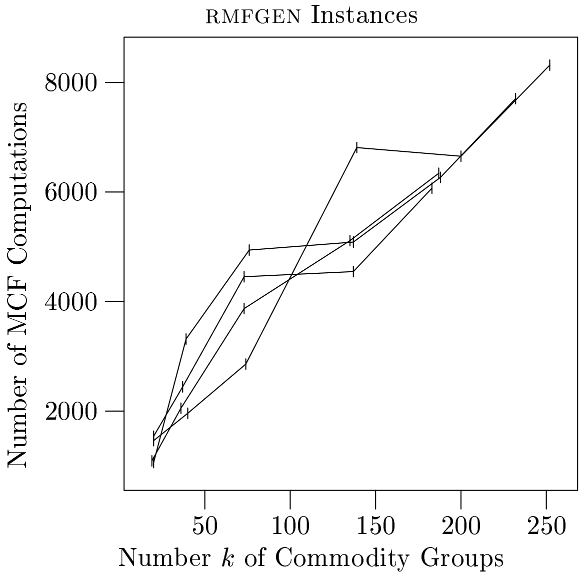

Figure 4.12: The number of minimum-cost flow computations as a function of the number k of commodities for several concurrent RMFGEN instances.

$30k$ for MULTIGRID and RMFGEN respectively. As we saw in the previous section, concurrent MULTIGRID instances are slightly harder to solve and concurrent RMFGEN instances are much easier.

Details of the Problem Instances The MULTIGRID instances are the same as in the previous section. RMFGEN instances were rmfgen-7-b-k for $b = 10$, 12, 14, and 16 and $k = 20$, 40, 80, 160, 240, and 320. All were solved to $\epsilon = 1\%$.

4.3 Dependence on the Network Size

To test the implementations' dependences on the problem size, we used TRIPARTITE problem instances with increasing numbers of frames. Each frame has fixed size so the number of vertices and arcs is linearly related to the number of frames. Figures 4.13 and 4.14 indicate dependences proportional to $n^{1.9}$ and $n^{1.5}$ for multicommodity minimum-cost flow and concurrent flow, respectively. Each line corresponds to a set of increasing large problem instance solved to the same ϵ. The dependence, close to the theoretical dependence of $\tilde{O}(n^2)$, is consistent for increasing problem size and different accuracies ϵ.

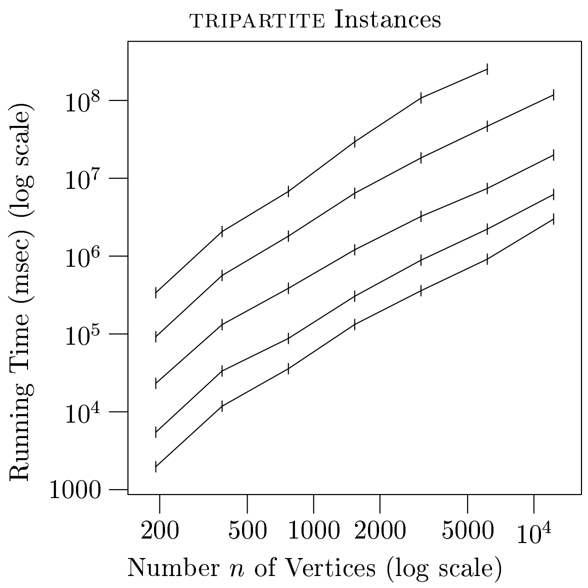

Figure 4.13: Running time as a function of network size, measured as a function of the number n, for several multicommodity minimum-cost flow TRIPARTITE instances.

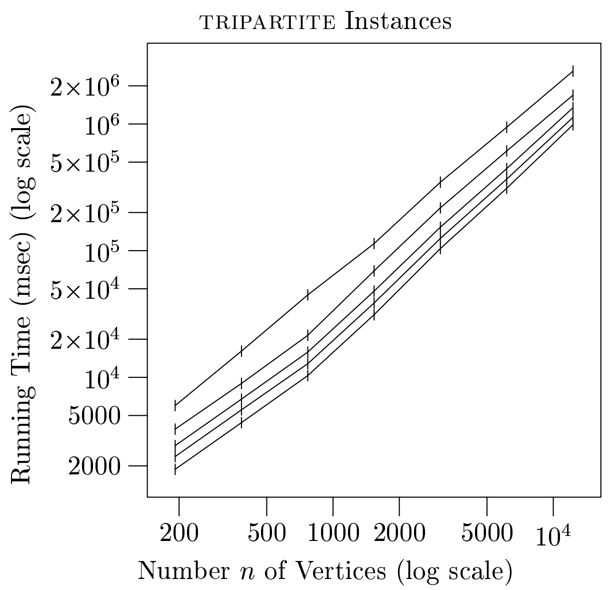

Figure 4.14: Running time as a function of network size, measured as a function of the number n, for several concurrent flow TRIPARTITE instances.

		Main Loop					MCF Computations	
			flow	lower	step			lower
problem	init	MCF	change	bound	calc	exp	reroute	bound
Multicommodity Minimum-Cost Flow Problems								
GTE	2.62	46.12	19.91	13.95	8.53	9.40	77.83	22.17
multigrid-032-128-0320	15.38	68.67	7.26	12.73	1.45	3.04	84.45	15.55
rmfgen-7-16-240	0.18	57.83	8.11	18.30	3.36	1.72	75.75	24.25
tripartite-04-16	0.02	52.94	15.90	17.90	9.05	4.39	74.98	25.02
tripartite-16-16	0.01	53.52	20.09	18.03	14.89	6.42	74.86	25.14
Concurrent Flow Problems								
GTE	0.10	43.41	36.84	14.63	29.92	13.96	74.54	25.46
multigrid-032-128-0320	3.23	65.54	8.66	18.36	2.10	2.32	78.00	22.00
rmfgen-7-16-240	1.93	65.89	9.23	21.25	5.48	2.82	75.49	24.51
tripartite-04-16	1.03	57.83	18.00	19.51	12.74	6.76	75.10	24.90
tripartite-16-16	1.38	60.57	16.74	18.83	12.11	6.24	76.57	23.43

Table 4.1: The percentage of the algorithms' running time among its various tasks. See the text for an explanation. All problems were solved to 1% accuracy on an UltraSparc-2.

Details of the Problem Instances The TRIPARTITE instances included tripartite-a-16 for a equaling powers of two from 4 to 256 and $\epsilon = 1$, 2, 4, 8, and 16%.

4.4 Distribution of Running Times

As presented in [KP95b], the algorithm spends most of its time computing single-commodity minimum-cost flows (MCFs) and most of the remaining time computing the exponentials in the potential function and gradient. The modifications of Chapter 3 increased the running time of choosing a step size from a constant to using the Newton-Raphson method, of choosing α from a constant to that of a heuristic scheme, and of determining termination from that of an if-expression to computation of all commodities' MCFs. Despite these modifications, the algorithms' running time is still dominated by MCF computations.

Table 4.1 shows the distribution of running time among the algorithms' various tasks. "Init" indicates the percentage of time spent computing the initial flow, while the next few columns account for the program's main loop of

- computing a minimum-cost flow (MCF),
- changing the flow by computing a step size and then modifying it and the potential function, and

- occasionally computing a lower bound.

Although included in the "flow change" percentage, the time for the Newton-Raphson method to compute the optimal step size is listed separately as is the time to compute all the programs' exponentials. Since both attempting to reroute flow and computing the lower bound require MCF computations, we indicate the fraction of the computations for each.

(Numbers do not necessarily sum to one hundred percent because a few unimportant tasks have been omitted and profiling's accuracy is limited. Profiling also introduces additional memory use that can affect the collected data. Furthermore, the data was collected on computers with other simultaneous users unlike the other data in this paper.)

Easy problems spend a larger fraction of time computing MCFs to reroute and finding initial flows and less time changing the flow and computing lower bounds. See, for example, the MULTIGRID and RMFGEN instances. Modifying the flow requires less time because computing an optimal step size is easier. Since the step size routine starts its search using the step size from the commodity's previous iteration, it will require fewer iterations if the step sizes do not vary much. Easy problems permit routing much flow per iteration, i.e., use relatively large step sizes. Since a lower bound is computed every $3m$ iterations, we expect these computations to account for up to 25% of the MCFs. The minimum-cost multigrid-032-128-0320 requires so few iterations to solve that the MCFs to reroute is significantly larger than 75%.

Computing the optimal step size usually takes less than one-sixth of the total running time despite its requirement to perform a large number of exponentials. (For GTE, the calculation requires almost one-third of the running time, but the benefit is a decrease in running time by a factor of thirty. See Table 3.2.) We reduced the number through the use of the Newton-Raphson method (Section 3.2), data structures sensitive to the number of arc flow changes (Section 3.5), and data structures speeding computation of congestions, the potential, and the gradient. To reduce the time to exponentiate, Leong et al. [LSS93, Section 3.3] wrote an approximation routine requiring only addition and multiplication [Ste99], but our program's need for a faster routine was much smaller and no portable routine we wrote was faster.

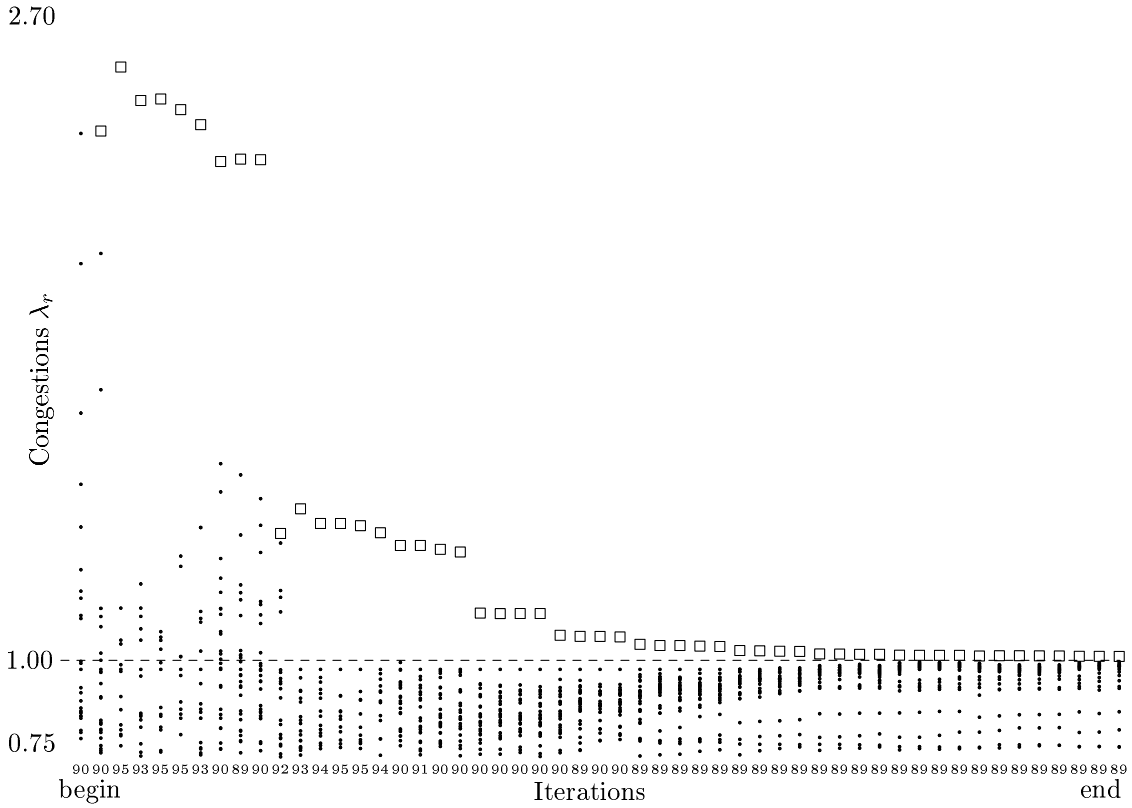

Figure 4.15: Constraint congestions when solving GTE. Squares indicate the cost congestion λ_c, while dots indicate arc congestions $\lambda_{(v,w)}$. Congestions less than 0.75 are omitted with the omitted percentage indicated. Only every twenty-fifth iteration is shown.

4.5 Congestion Change

In this section, we illustrate how the constraint congestions λ_r change during the algorithm's execution. When the algorithm begins, congestions can be quite large, but, when the algorithm terminates, they are at most $1 + \epsilon$.

Figure 4.15 displays constraint congestions as a function of time while solving the multicommodity minimum-cost flow GTE problem to an accuracy of 1%. To improve the figure's clarity, only every twenty-fifth iteration of the 1302 iterations are displayed. Also, the percentage of congestions less than 0.75 are represented by the decimal number at the bottom of each iteration. Dots represent arc capacity congestions $\lambda_{(v,w)}$ while squares represent the cost congestions λ_c.

Initially, the cost congestion drops rapidly as the improved guesses for the minimum possible cost occur. Throughout the algorithm, the cost congestion remains the largest congestion so the algorithm terminates when $\lambda_c \leq 1 + \epsilon$. During most of the execution, the capacity congestions remain less than 1.00. Initially, this is not true, but they decrease rapidly except for a few iterations when an abnormally small guess for the minimum cost temporarily cause them to increase. During the latter two-thirds of the execution, the fraction of congestions less than 0.75 remains fairly constant at 89–90%. The other arcs become increasingly saturated as the solution's cost decreases.

The behavior exhibited by GTE is fairly typical. The cost congestion usually is the only congestion greater than 1.00. A small subset of arcs have congestions in the range 0.90–1.00. As the cost decreases, these congestions move closer to 1.00 but rarely exceed it.

Chapter 5

Comparisons with Other Implementations

We compared our multicommodity minimum-cost flow implementation MCMCF and our concurrent flow implementation MCF with a commercial linear-programming package CPLEX [CPL95] and the primal partitioning multicommodity flow implementation PPRN [CN96].

After describing the other implementations, we present multicommodity minimum-cost flow data and then concurrent flow data. We close with a comparison of solving large problem instances.

5.1 Multicommodity Minimum-Cost Flow Comparisons

5.1.1 The Other Implementations: CPLEX and PPRN

We compared our algorithms (solving to 1% accuracy) with the commercial linear-programming package CPLEX [CPL95] and the primal partitioning multicommodity flow implementation PPRN [CN96].

CPLEX (version 4.0.9) is among the fastest available general linear programming packages and implements several different algorithms, all yielding exact (up to eight decimal digits) solutions. To determine which algorithm to use, we solved a few multicommodity minimum-cost flow problem instances using the various algorithms. See Table 5.1. (We also tried several different parameter values not shown in the table.) CPLEX's dual simplex method yields a feasible solution only upon completion, while the primal method, in principle, could

problem	primal simplex	dual simplex	network simplex
GTE	12.06	4.13	3.61
multigrid-016-032-0100	6039.42	851.06	5609.77
rmfgen-7-10-040	3470.34	193.9	653.68

Table 5.1: Running times (seconds) for various simplex algorithms to solve various multicommodity minimum-cost flow problem instances. Problems solved on a Sun Enterprise 3000.

be stopped to yield an approximation. Despite this fact, we compared `MCMCF` with `CPLEX`'s dual simplex method because it is an order of magnitude faster than its primal simplex for the problems we tested.

`PPRN` [CN96] specializes the primal partitioning linear programming technique to solve multicommodity problems. The primal partitioning method splits an instance's basis into bases for the commodities and another basis for the joint capacity constraints. Network simplex methods then solve each commodity's subproblem. More general linear-programming matrix computations applied to the joint capacity basis combine these subproblems' solutions to solve the problem.

Both `CPLEX` and `PPRN` yield exact solutions. We would like to compare our codes' running times with modified programs, but it is not clear how to make the modifications. Also, our codes group commodities at runtime, which can measuredly reduces the running time. (See Table 3.11.) Thus, when forming these programs' inputs, we group the commodities but do not include this in the running time.

5.1.2 Dependence on the Number k of Commodity Groups

Our multicommodity minimum-cost flow algorithm `MCMCF` solves our problem instances up to two to three orders of magnitude faster than the linear-programming-based implementations `CPLEX` and `PPRN`. Furthermore, its running time depends mostly on the network structure and much less on the arc costs' magnitude.

We solved several different RMFGEN networks (see Figures 5.1 and 5.2) with various numbers of commodities and two different arc cost schemes. Even for instances having as few as fifty commodities, `MCMCF` required less running time. Furthermore, its dependence on the number k of commodities was much smaller. For Figure 5.1, the arc costs were randomly chosen from the range $[1, 100]$. For these problems, `CPLEX`'s running time is roughly quadratic in k, while `MCMCF`'s is roughly linear. Although for problems with few

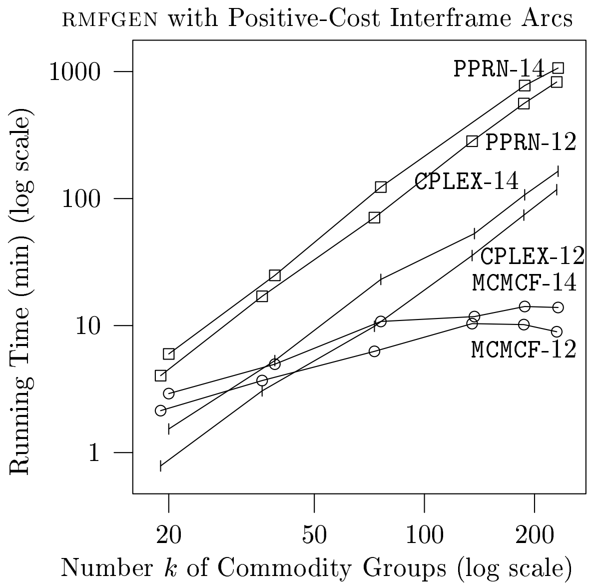

Figure 5.1: The running time in minutes as a function of the number k of commodity groups for two different RMFGEN networks with twelve (12) and fourteen (14) frames. Interframe arc costs are positive. `CPLEX`'s and `PPRN`'s dependences are larger than `MCMCF`'s.

commodities, `CPLEX` is somewhat faster, for larger problems `MCMCF` is faster by an order of magnitude. `PPRN` is about five times slower than `CPLEX` for these problems. Changing the cost of interframe arcs significantly changes `CPLEX`'s running time. (See Figure 5.2.) Both `MCMCF`'s and `PPRN`'s running times decrease slightly. The running times' dependences on k do not change appreciably.

`MCMCF` solves MULTIGRID networks two to three orders of magnitude faster than `CPLEX` and `PPRN`. Figure 5.3 shows `MCMCF`'s running time using a log-log scale for two different networks: the smaller one having 1025 vertices and 3072 arcs and the larger one having 4097 vertices and 9152 arcs. `CPLEX` and `PPRN` required several days to solve the smaller network instances so we omitted solving the larger instances. Even for the smallest problem instance, `MCMCF` is eighty times faster than `CPLEX`, and its dependence on the number of commodities is much smaller. `PPRN` is two to three times slower than `CPLEX` so we solved only very small problem instances using `PPRN`.

Details of the Problem Instances The RMFGEN problem instances included rmfgen-7-{12,14}-k for k = 20, 40, 80, 160, 240, and 320. The MULTIGRID instances included

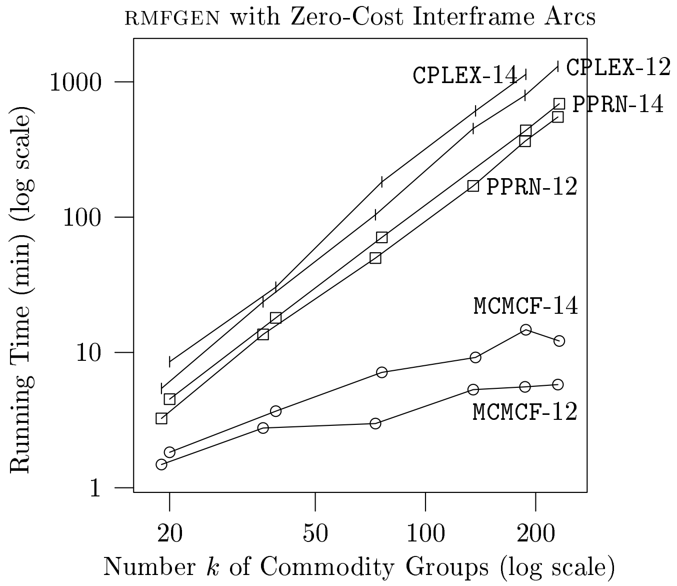

Figure 5.2: The running time in minutes as a function of the number k of commodity groups for two different RMFGEN networks with twelve (12) and fourteen (14) frames. Interframe arc costs are zero.

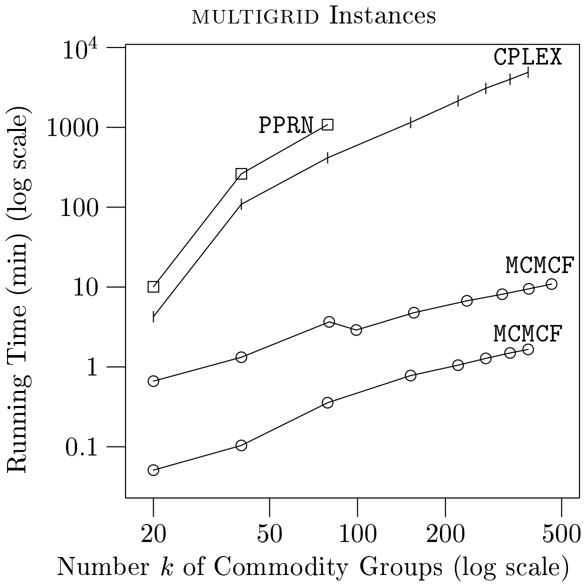

Figure 5.3: The running time in minutes as a function of the number k of commodity groups for two different MULTIGRID networks with approximately 32^2 and 64^2 vertices.

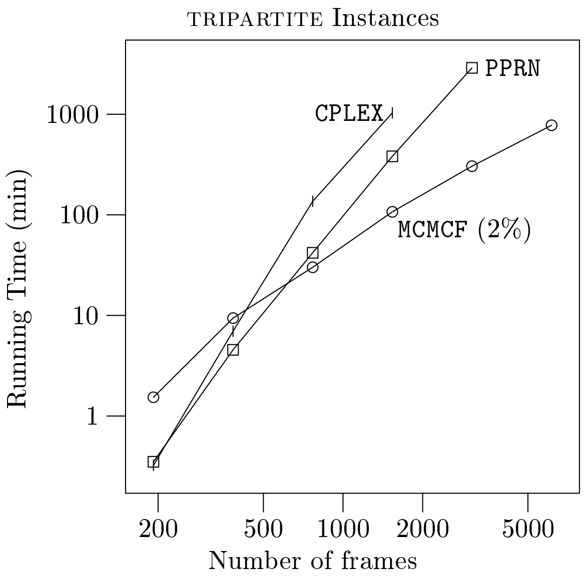

Figure 5.4: The running time in minutes as a function of the number of frames for TRIPARTITE problem instances.

multigrid-032-128-k for $k = 20$, 40, 80, 160, 240, 320, 400, and 480, and multigrid-064-128-k for the same k.

5.1.3 Dependence on the Network Size

To test the implementations' dependences on the problem size, we used TRIPARTITE problem instances with increasing numbers of frames. Each frame has fixed size so the number of vertices and arcs is linearly related to the number of frames. For these instances, `MCMCF`'s almost linear dependence on problem size is much less than `CPLEX`'s and `PPRN`'s dependences. See Figure 5.4. (`MCMCF` solved the problem instances to two-percent accuracy.) As described in Section 3.4, the minimum-cost flow routine needs only a few pivots before a solution is found. `CPLEX`'s and `PPRN`'s dependences are much higher.

Details of the Problem Instances The TRIPARTITE instances included tripartite-a-16 for a equaling powers of two from 4 to 128.

	Time (seconds)				
problem	primal	dual	network/primal	network/dual	interior
GTE	8	31	7	41	24
multigrid-016-032-0100	503	4059	422	1319	3421
multigrid-032-128-0080	7012	6910	5463	omitted	omitted
rmfgen-7-10-040	339	1913	223	965	14871
rmfgen-7-14-080	3529	20345	2212	8872	32727
tripartite-16-08	25	59	28	64	19

Table 5.2: Running times (seconds) for various simplex algorithms to solve various concurrent flow problem instances. Problems solved on a Sun Enterprise 3000.

5.2 Concurrent Flow Comparisons

5.2.1 The Other Implementation: `CPLEX`

As for the multicommodity minimum-cost flow problem, we compare our implementation `MCF` with `CPLEX`. For these problems, `CPLEX` runs fastest starting with the network simplex algorithm and then switching to primal simplex is faster than using just dual simplex. See Table 5.2.

The concurrent flow problem can also be solved using binary search on λ^* and testing the feasibility of problem instances. For testing feasibility, neither `CPLEX` nor `PPRN` were competitive so we did not pursue such experiments. Since we do not otherwise know how to solve concurrent flow problems using `PPRN`, we will compare only `CPLEX` with `MCF`.

5.2.2 Dependence on the Number k of Commodity Groups

As for the multicommodity minimum-cost flow instances, the combinatorial algorithm `MCF` solves our concurrent flow problem instances up to two to three orders of magnitude faster than the linear-programming-based implementations `CPLEX`. See Figures 5.5 and 5.6. For all networks, `MCF` starts an order of magnitude faster for the smallest number of commodities and has smaller dependence. This indicates that our implementations can solve problems with more commodities than previously possible.

The problem instances are the same as in Section 5.1.2 except the arc costs have been omitted.

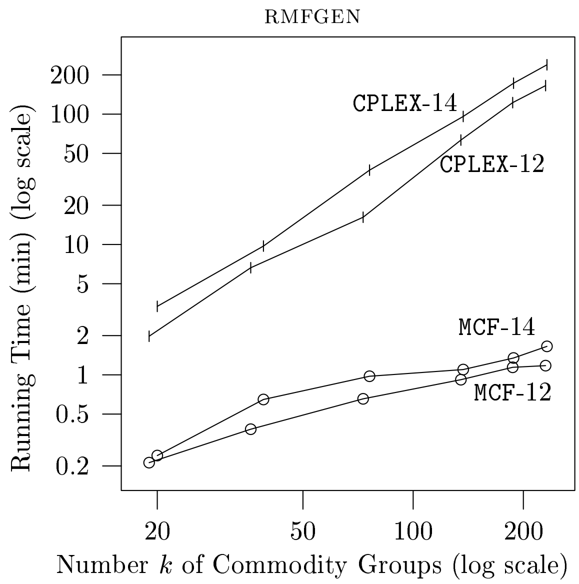

Figure 5.5: The running time in minutes as a function of the number k of commodity groups for two different RMFGEN concurrent flow networks with twelve (12) and fourteen (14) frames.

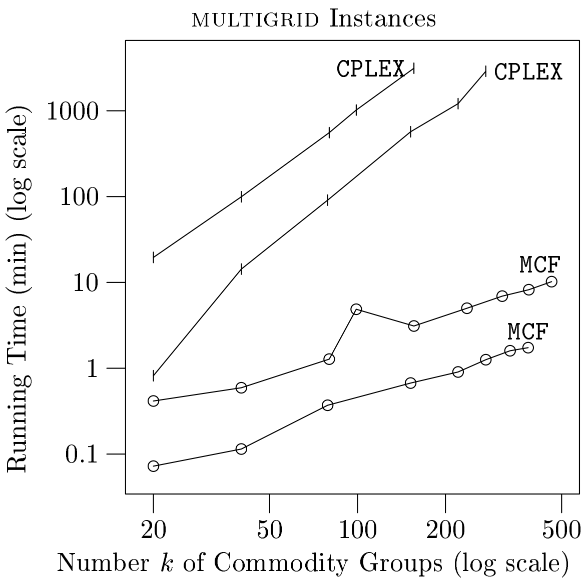

Figure 5.6: The running time in minutes as a function of the number k of commodity groups for two different MULTIGRID concurrent flow networks with approximately 32^2 and 64^2 vertices.

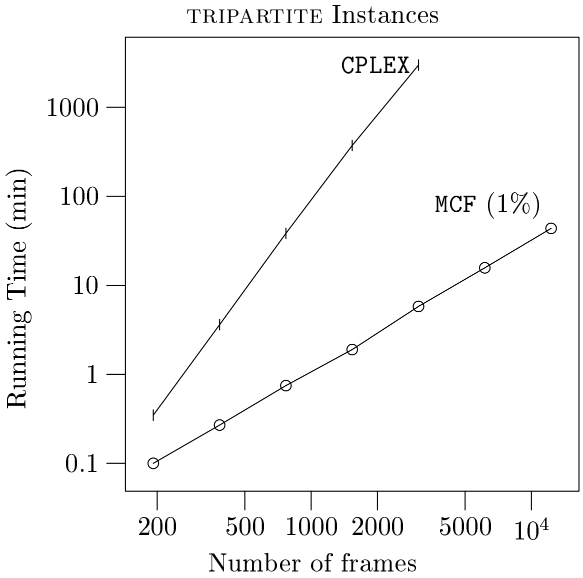

Figure 5.7: The running time in minutes as a function of the number of frames for TRIPARTITE problem instances.

5.2.3 Dependence on the Network Size

Figure 5.7 shows MCF outperforms CPLEX on all the problem instances we tested. It starts half an order of magnitude faster and has smaller dependence on the network's size. The absence of arc costs makes these problems significantly easier to solve than their multicommodity minimum-cost versions.

The problem instances are the same as in Section 5.1.3 except the arc costs have been omitted.

5.3 Large Multicommodity Flow Problems

Up to this point, our emphasis has been solving problems quickly. We now turn to solving multicommodity flow problems that require large amounts of space to specify. Specifying a multicommodity flow problem with single-source, single-sink commodities requires only a constant amount of information for each arc and each commodity, i.e., $O(k + m)$ space. A linear program specifying the flow conservation and arc capacity inequalities requires $O(k(n + m))$ space. We expect linear program solvers to require at least this much space

			Time (seconds)			
n	m	k_0	50%	10%	5%	CPLEX
1636	3235	66095	44	44	53	29927
6780	13485	128183	1835	3150	4280	4Gb memory not enough

Table 5.3: Characteristics and running times for two chip layout concurrent flow problem instances. CPLEX 5.0.1 used on a Sun UltraEnterprise.

when solving the problems. Although not designed to minimize its space requirements, our implementation requires $O(km)$ space to store the flow vector, $O(m)$ shared space, and $O(kn)$ additional space for all the minimum-cost flow objects. The space requirements do not increase or decrease because of matrix fill-in or related phenomena.

Our implementation to be able to solve larger problems than CPLEX. For our linear program corresponding to multigrid-064-128-k, $k = 240$, 320, 400, and 480, CPLEX required more space than the 1.5Gb available virtual memory. MCMCF required at most 217Mb space.

Andre Rohe of the Forschungsinstitut für Diskrete Mathematik at the University of Bonn supplied large multicommodity flow problem instances derived from chip layout problems for which he sought only 100%-approximations. See Table 5.3 for the problem characteristics, noting the number of commodities listed is before grouping. For the two problems, solving 5% approximations took just under one minute and just over one hour, respectively. For the smaller problem, CPLEX required five hundred times longer to compute an exact answer. For the larger problem, it could not allocate enough virtual memory to read the problem instance. Despite being able to solve very large problem instances, further research into reducing our algorithms' space requirements is needed.

5.4 Concluding Remarks

For the problem classes we studied, MCMCF and MCF solved multicommodity minimum-cost flow and concurrent flow problems significantly faster than state-of-the-art linear-programming-based programs. This is strong evidence that the approximate problem is simpler, and that combinatorial-based methods, appropriately implemented, should be considered for this problem. We believe many of these techniques can be extended to other problems solved using the fractional packing and covering framework of [PST95].

Chapter 6

Generalized Shortest Paths

6.1 Introduction to Generalized Flow Problems

Ordinary network flow models require flow conservation on all arcs: The amount of flow entering an arc equals the amount of flow leaving the arc. Generalized network flow models modify this conservation by associating a flow multiplier $\mu(v, w)$ with each arc (v, w). For each unit of flow sent from vertex v along the arc, $\mu(v, w)$ units of flow arrive at w. Using flow multipliers permits two types of modeling not possible with canonical models. Flow multipliers can represent transformations from one type of object to another. For example, Hong Kong dollars can be converted into South African rands, and trees can be converted into reams of paper. Multipliers can also modify the amount of flow. Thus, one can model evaporation from a network of water canals and breakage caused during transport through a delivery network.

The generalized flow model has been studied [Dan63, Jew62] since before the publication of Ford and Fulkerson's network flows book [FF62] defined flows as an area of research, but the only previously known combinatorial polynomial-time algorithms solved the generalized versions of the shortest path and maximum flows problems. In this chapter, we combine a dynamic-programming approach similar to the Bellman-Ford algorithm [Bel58, For56] with Megiddo's parametric search technique [Meg79] to solve the single-source generalized shortest path problem (GSP), also called the *restricted generalized uncapacitated transshipment problem*. Previous approaches [AC91, CM94b, HN94] solved the dual problem as a linear program with two variables per inequality and then converted the solution back to the original primal problem. Our approach has exactly the same running time but is simpler, directly uses the given graph, avoids the dual-to-primal conversion, and requires less space.

Using the GSP algorithm as a subroutine in the Garg-Könemann and Grigoriadis-Khachiyan fractional packing frameworks [GK96b, GK98], we obtain fully polynomial-time approximation schemes for all variants of generalized network flow problems with nonnegative costs. Excepting the generalized shortest path and the generalized maximum-flow problems [CM94b, GJ96, GJO97, GPT91, Rad93, Rad98, TW98], these are the first combinatorial polynomial-time algorithms known to the author. Furthermore, the generalized concurrent flow and the generalized multicommodity maximum-flow flow algorithms extend the class of problems for which strongly polynomial algorithms (assuming a fixed approximation factor) are known.

Generalized Shortest Path Problems In the single-source generalized shortest path problem, one is given a directed graph with arc costs, arc multipliers, and a source vertex s. The objective is to find the cheapest flow removing one unit from the source vertex. Flow is conserved at all vertices except the source vertex, but the flow multipliers modify the flow amount. Since all the multipliers are positive, the solution cannot consist of a loopless path. Instead, it must be an *augmented path*, i.e., a path connected to a lossy cycle. The flow multiplier of such a cycle is less than one. If one unit of flow enters a lossy cycle, traversing the cycle once will yield less than one unit of flow. Repeated traversals will "consume" the flow. (The reader is encouraged to think of a flow as a static function obeying flow conservation constraints at vertices and arc multipliers along arcs, not an entity flowing through a network. This is because vertex flow conservation constraints require more flow through an augmented path's lossy cycle than the amount of flow "reaching" the cycle through the path from the source vertex.)

We show that the vertex potentials of the dual linear problem form a left-distributive, but not right-distributive, closed semiring. Fully-distributive closed semirings are algebraic structures underlying ordinary path algorithms, e.g., the Floyd-Warshall algorithm, in directed graphs [AHU74, CLR90]. However, our dynamic-programming (and the Bellman-Ford) algorithm requires only left-distributivity. Given an initial value for the source's vertex potential, one dynamic-programming computation will indicate whether the potential was smaller than, equal to, or greater than the problem's optimal value. Using this comparison subroutine, we can perform binary search on the problem's optimal value to solve the problem, but the flow multipliers prevent a strongly polynomial running time.

To obtain a $O(mn^2 \log m)$ strongly polynomial running time, we use Megiddo's parametric search technique [Meg79] to compute the optimal value. Instead of representing the vertex potentials as numbers, we use lines that are a function of the source vertex's potential. Throughout the algorithm, each iteration narrows the possible range for the optimal value and consists of one dynamic-programming iteration. An iteration may invoke the comparison subroutine, which narrows the range for the optimal value. At the algorithm's termination, the instance's optimal value is returned.

Since Khachiyan's polynomial-time algorithm for solving linear programs [Kha80], the generalized shortest path problem has had a polynomial-time algorithm. Prior to his work, Charnes and Raike [CR66] showed how to solve the GSP using a variant of Dijkstra's algorithm [Dij59], but problem instances are restricted to have nonnegative costs and flow multipliers at most one. The goal is to obtain one unit of flow at a sink, not a source, vertex. In 1973, Bakó [Bak73] presented a closed semiring for use with the Bellman-Ford algorithm. Although the algorithm permits arbitrary arc costs, flow multipliers are still restricted to be at most one and the goal is to obtain flow at a sink, not a source, vertex. Despite the fact that most ordinary shortest path algorithms are based on closed semirings, Bakó's result is the only known closed semiring with both left- and right-distributivity for generalized shortest path problems.

In 1991, Adler and Cosares published a specialized polynomial-time algorithm for the most general GSP problem having arbitrary costs, arbitrary positive flow multipliers, and a source vertex [AC91]. They showed how to convert a feasible solution to the GSP's dual linear program into an optimal solution to the primal linear program. Their algorithm's running time is dominated by the time to find a feasible solution to a linear program having only two variables per inequality (abbreviated 2VPI or TVPI). They used Megiddo's algorithm [Meg83]. Cohen and Megiddo [CM94a] improved the 2VPI algorithm, decreasing the running time for GSP to $O(mn^2(\log m + \log^2 n))$ [CM94b]. Hochbaum and Naor [HN94] used Fourier-Motzkin elimination [Sch86, Section 12.2] to speed the 2VPI solution. When combined with Adler and Cosares's result, this yields the fastest known running time of $O(mn^2 \log m)$. All of these algorithms use Aspvall and Shiloach's subroutine [AS80] to compute the range of feasibility for variables. Each iteration of this Bellman-Ford-like subroutine reduces the possible range for either the upper or lower bound of one variable. After computing feasibilities for all dual variables, these GSP algorithms use [AC91] to convert to an optimal primal solution. In contrast, we present a left-distributive closed semiring that can be directly used in a Bellman-Ford algorithm to reduce the known range for the GSP's optimal solution. It then becomes obvious this Bellman-Ford algorithm can be combined with Megiddo's parametric search technique [Meg79].

The resulting algorithm, with running time exactly matching that of Hochbaum and Naor, is likely to be practical. The algorithm can operate directly on the graph's data structure as given. No vertex contractions or arc additions or deletions are required. The algorithm's inner loop can be programmed by inserting the left-distributive closed semiring's operations in a templatized Bellman-Ford subroutine. Combining this with binary search yields an easy implementation, likely to work well in practice. Otherwise, the Bellman-Ford code can be modified for parametric search. The algorithm's hidden constants are likely to be smaller than the 2VPI approach because the Bellman-Ford subroutine's inner loop has only one, not four, case statement and the algorithm computes only one number, not a lower and upper bound, for each vertex.

Other Generalized Flow Problems Two recent frameworks [GK96b, GK98] permit us to use a generalized shortest path (GSP) algorithm to produce fully polynomial approximation schemes for all other generalized flow problems with nonnegative costs: generalized maximum flow, generalized nonnegative-cost minimum-cost flow, generalized concurrent flow, generalized multicommodity maximum flow, and generalized multicommodity nonnegative-cost minimum-cost flow. (Tables 6.1 and 6.2 on pages 71–72 contain the schemes' running times.) The frameworks permit computing approximate solutions by repeatedly solving GSP problems with different arc costs. These arc costs reflect the current violation of the problem's capacity and cost constraints. The Garg-Könemann framework [GK98] uses a greedy approach. Each iteration yields an augmented path routing as much flow as the path's capacity constraints permit. The arc costs are exponentially related to the ratio of their flow to capacity. The Grigoriadis-Khachiyan framework [GK96b] repeatedly reroutes flow until it is ϵ-optimal. A potential function, which is closely related to the sum of the capacity and cost constraints' dual variables, yields the arc costs for a GSP iteration. The current flow is replaced by its convex combination with the GSP flow scaled to satisfy a source vertex's supply. Similar frameworks [KP95b, PST95] have proven practical giving a good indication these frameworks will yield practical algorithms. Leong et al. [LSS93] solved concurrent flow problems, while the multicommodity minimum-cost flow implementation of Goldberg et al. [GOPS98] solves instances (to one-percent accuracy) up to three orders of magnitude faster than other algorithms.

Running times for the approximation schemes and other extant generalized flow algorithms are listed in Tables 6.1 and 6.2. Both exact and approximation algorithms are included. All the running times of the interior point and several combinatorial algorithms depend on the size of the input numbers. B is the largest integer in the problem's input assuming the arc capacities and vertex supplies/demands are represented as integers and the flow multipliers are ratios of integers. C and U represent the largest arc cost and capacity, respectively.[1] Excepting generalized maximum flow, this paper's algorithms have the smallest running time for certain parameter values.

Generalized Flow Algorithms and Strong Polynomiality The running times of *strongly polynomial algorithms* depend only on the size of the instance's underlying structure, not the number of bits required to represent the input data. Tardos [Tar86] presented a strongly polynomial algorithm for linear programs with bounded constraint matrix entries. Generalized flow problems can have arbitrary size flow multipliers so her algorithm does not solve these problems, but they are among the next natural set of problems to consider. Except for the generalized shortest paths problem, there are no known strongly polynomially algorithms for generalized flow problems. Furthermore, exact answers can have a polynomial number of digits so approximate solutions are usually more desirable. In 1994, Cohen and Megiddo [CM94b] presented a fully strongly polynomial approximation scheme for the generalized maximum flow problem. A *fully polynomial approximation scheme* is a family of approximation algorithms such that, for any desired accuracy $\epsilon > 0$, there is an algorithm returning a solution within a $1+\epsilon$ factor of the optimal and having a running time polynomial in the input size and ϵ^{-1}. Adding the adjective "strongly" indicates the running time depends only on the input parameters, not the individual values, assuming a fixed accuracy ϵ. We present the first fully strongly polynomial approximation schemes for the generalized nonnegative-cost cost-bounded flow, generalized multicommodity nonnegative-cost cost-bounded flow, generalized concurrent flow, and generalized multicommodity maximum flow problems.

[1]Only polynomial-time algorithms published since 1985 have been included. Exact running times have been listed when presented by either the authors or subsequent references. Otherwise $\tilde{O}()$ notation omitting uninteresting polylogarithmic terms of m and n is used. Running times for approximation algorithms are presented when known.

The interior point running times are reported differently by different authors. We rely on [Rad98].

Outline of the Generalized Flow Chapters In the next section, we formally define the single-source generalized shortest path problem and prove its solution consists of an augmented path. In the subsequent section, we present the dual linear program, and, in Section 6.4, a left-distributive closed semiring is presented. Subsequently, we present a dynamic-programming comparison subroutine indicating whether the source vertex's initial potential is smaller than, equal to, or larger than the problem instance's optimal value. Using the comparison subroutine, binary-search and geometric-search algorithms can solve the GSP problem. In Section 6.7, the comparison subroutine is modified using Megiddo's parametric search technique to yield a strongly polynomial-time algorithm. In the next chapter, we derive approximation algorithms for all the other generalized flow problems.

single-source generalized shortest paths (GSP)		
Adler and Cosares [AC91]	$O(mn^3 \log m)$	2VPI [Meg83] + transformation
Cohen and Megiddo [CM94b]	$O(mn^2(\log m + \log^2 n))$	2VPI [CM94a] + [AC91]
Hochbaum and Naor [HN94]	$O(mn^2 \log m)$	Fourier-Motzkin 2VPI+ [AC91]
this paper	$O(mn^2 \log m)$	dynamic prog. + [Meg79]
	$O(mn(\log C + n \log B))$	dynamic prog. + binary search
generalized maximum flow		
interior point algorithms	see generalized concurrent flow with $k = 1$	
Goldberg, Plotkin, and Tardos [GPT91]	$O(mn^2(m + n \log n) \log n \log B)$	repeated MCFs
	$O(m^2 n^2 \frac{\log n \log^2 B}{\log(n^2/m) + \log\log n + \log\log B})$	Fat-Path
	$O(mn^2 \log n \log B \log \epsilon^{-1})$	Fat-Path
Radzik [Rad93]	$O(m^2 n \log^2 m \log \epsilon^{-1})$	capacity scaling
Cohen and Megiddo [CM94b]	$O(m^3 n^2 (\log m + \log^2 n) \log B)$	2VPI + packing
	$O(m^2 n^2 (\log m + \log^2 n) \log \epsilon^{-1})$	2VPI + packing
Goldfarb and Jin [GJ95]	$O(mn^2(m + n \log n) \log B)$	dual simplex version of [GJ96]
Goldfarb and Jin [GJ96]	$O(mn^2(m + n \log n) \log B)$	improvement on MCF [GPT91]
Goldfarb, Jin, and Orlin [GJO97]	$O(m^2(m + n \log n) \log B)$	scaling
Goldfarb, Jin, and Lin [GJL98]	$O(m^2(m + n \log n) \log B)$	dual simplex version of [GJO97]
Radzik [Rad98]	$O(m^2(m + n \log n \log(n \log B)) \log B)$	Fat-Path + approx. cycle cancel
	$O(mn^2 \log n \log B + m(m + n \log n \log(n \log B)) \log(n\epsilon^{-1}))$	
Tardos and Wayne [TW98]	$O(m^2(m + n \log n \log(n \log B)) \log B)$	flow multiplier scaling
	$O(mn^2 \log n \log B + m(m + n \log n \log(n \log B)) \log(n\epsilon^{-1}))$	
this paper	$O(\epsilon^{-2} m^2 n^2 \log m \log n)$	GSP + [GK98]
	$O(\epsilon^{-2} m^2 n \log n \log(\epsilon^{-1}(\epsilon^{-1} \log m + n \log B)))$	GSP + [GK98]
generalized minimum-cost flow		
interior point algorithms	see generalized concurrent flow with $k = 1$	
Wayne [Way99]	$O(m^3 n^2 \log m \log B)$	min-ratio cycle canceling
	$O(m^2 n^2 \log m \log \epsilon^{-1})$	min-ratio cycle canceling
this paper (nonnegative costs)	$O((\epsilon^{-2} \log \epsilon^{-1} + \log m) m^2 n \log(\epsilon^{-1}(\log(m/\epsilon) + n \log B)) \cdot$ $\log((\log(mCU) + n \log B)/\epsilon))$	GSP + [GK96b]

Table 6.1: Running times for many generalized single-commodity flow algorithms. See the text for an explanation.

generalized concurrent flow		
Kapoor and Vaidya [KV86]	$O(k^{2.5}m^{1.5}n^{2.5}\log B)$	interior point
Vaidya [Vai89b]	$O(k^{2.5}m^{1.5}n^{2}\log B)$	interior point + [KV86]
Murray [Mur92]	$O(k^{2.5}m^{1.5}n^{2}\log B)$	interior pt w/o fast matrix mult.
Kamath and Palmon [KP95a]	$O(k^{2.5}m^{1.5}n^{2}\log B)$	interior point
this paper	$O((\epsilon^{-2}\log\epsilon^{-1}+\log m)m(kmn^2\log m+\log\log(m/\epsilon)))$	GSP + [GK96b]
	$O((\epsilon^{-2}\log\epsilon^{-1}+\log m)km^2n\log(\epsilon^{-1}(\log(m/\epsilon)+n\log B)))$	
generalized multicommodity maximum flow		
interior point algorithms	same as for generalized concurrent flow	
this paper	$O(\epsilon^{-2}km^2n^2\log m\log n)$	GSP + [GK98]
	$O(\epsilon^{-2}km^2n\log n\log(\epsilon^{-1}(\epsilon^{-1}\log m+n\log B)))$	GSP + [GK98]
generalized multicommodity minimum-cost flow		
interior point algorithms	same as for generalized concurrent flow	
this paper (nonnegative costs)	$O((\epsilon^{-2}\log\epsilon^{-1}+\log m)km^2n\log(\epsilon^{-1}(\log(m/\epsilon)+n\log B))\cdot$	
	$\log((\log(mCU)+n\log B)/\epsilon))$	GSP + [GK96b]

Table 6.2: Running times for many generalized multicommodity flow algorithms. See the text for an explanation.

6.2 Single-Source Generalized Shortest Path Problem

In this section, we define the single-source generalized shortest path problem, presenting a simple example. Then, we show any feasible instance with bounded minimum-cost has an augmented path solution. We finish the section by computing an upper bound on the number of bits in the optimal solution.

The *single-source generalized shortest path problem* (GSP), also called the *restricted generalized uncapacitated transshipment problem*, is to find a minimum-cost flow function obeying flow conservation, obeying the arc multipliers, and starting at a source vertex. The input consists of

- a directed graph $G = (V, A)$,
- an arc multiplier function $\mu : A \to \mathbf{R} > 0$,
- an arc cost function $c : A \to \mathbf{R}$, and
- a source vertex $s \in V$.

The resulting flow function $f : A \to \mathbf{R} \geq 0$ must obey flow conservation at the vertices, the multiplier function, remove one unit of flow from the source vertex s, and minimize the flow's cost. Flow conservation ensures that the flow into a vertex equals the flow out of the vertex except the source vertex's net flow must be one unit out. The flow multipliers can change the amount of flow in the network. For example, if three units of flow enter an arc with a flow multiplier μ of four, twelve units exit the arc. By convention, the cost of a flow on an arc is the product of the arc's cost and the flow *entering* the arc, as opposed to leaving the arc. A flow's cost is the sum, over all arcs, of these products.

We denote the number of arcs and vertices by m and n, respectively. To ease notation, we sometimes denote an arc (v, w) using its tail and head vertices even though our algorithms permit multiple arcs between the same pair of vertices. (For ordinary flow problems, all but the cheapest such arc may be removed, unlike for generalized flow problems where arcs have both costs and multipliers.) A (possibly empty) path is sometimes represented $v \rightsquigarrow w$ with an individual arc represented as $v \to w$. Iverson notation [*predicate*] is 1 if *predicate* is true and 0 otherwise [Knu92].

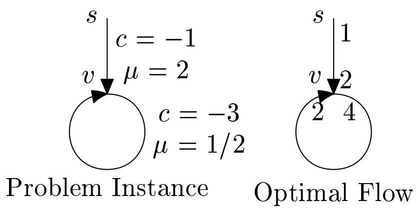

Figure 6.1: A generalized shortest path problem with a negative-cost cycle and bounded minimum cost. Arc costs and multipliers are labeled c and μ, respectively. The optimal flow, removing one unit from the source s, is indicated at arcs' tails and heads.

Writing the GSP requirements as a linear program:

$$\begin{aligned}
&\text{Minimize} \quad \sum_{\text{arcs } (v,w)\in A} c(v,w)f(v,w) \\
&\text{subject to } (\forall \text{ vertices } v \in V)\left(\sum_{\{w:(v,w)\in A\}} f(v,w) - \sum_{\{w:(w,v)\in A\}} \mu(w,v)f(w,v) = [v = s]\right) \qquad (6.1)\\
&\qquad (\forall \text{ arcs } (v,w) \in A)(f(v,w) \geq 0).
\end{aligned}$$

The constraints' equalities ensures flow is conserved at vertices except the source vertex.

The GSP has no sink vertices (vertices with positive net inflow) and exactly one source vertex s with unit supply. Conceptually, sink vertices can be eliminated by adding a zero-cost self-looping arc with flow multiplier less than one. Thus, any flow reaching the sink will be consumed by the arc. If multiple source vertices are present, the problem can be solved for each source and then the solutions can be combined. Also, without loss of generality, we assume all vertices are reachable from the source vertex s.

An Example Figure 6.1 illustrates a very simple GSP instance and its solution. The problem instance has two negative cost arcs, one being a loop with a multiplier less than one. The optimal flow at arcs' tails and heads is indicated in the right figure. The unit of flow removed from the source vertex s is multiplied to two units by the $s \to v$ arc. The flow obeys the multiplier of $v \to v$. Flow is conserved at v because two units enter from the $s \to v$ arc, two units enter from the cycle arc, and four units leave on the cycle arc. The flow's cost is $-1 \cdot 1 + -3 \cdot 4 = -13$. Notice the instance has a solution even though the minimum cost is negative.

Definitions Flow multipliers and costs can be defined for any walk. The *flow multiplier $\mu(W)$ of a walk W* is the product of its arcs' flow multipliers. The definition ensures flow conservation at its vertices. A *lossy cycle C* has flow multiplier $\mu(C)$ less than one. *Breakeven* and *gainy* cycles have multipliers equal to and greater than one, respectively. The *cost of a walk* is the cost of sending a unit of flow along the walk starting at its initial vertex. For example, the cost of the path $v \to w \to x$ is $1 \cdot c(v \to w) + \mu(v \to w)c(w \to x)$. The multiplier and cost of an empty walk is 1 and 0, respectively.

An *augmented path $s \rightsquigarrow v \rightsquigarrow w \to v$* is a nonempty path $s \rightsquigarrow v \rightsquigarrow w$ with an *extra arc* $w \to v$ forming a lossy cycle $v \rightsquigarrow w \to v$. An augmented path is a solution to the GSP because its path transports the source's unit supply to a lossy cycle which "consumes" the flow reaching it.

Optimal Solutions We now discuss optimal and infeasible solutions for the GSP. Any generalized flow can be decomposed into five types of primitive elements [GTT90, Theorem 1.7.2]:

1. a path with source and sink vertices,
2. a gainy cycle with a path leading to a sink vertex,
3. a path connecting a source vertex to a lossy cycle, i.e., an augmented path,
4. a breakeven cycle, and
5. a gainy cycle connected to a lossy cycle, i.e., a *bicycle*.

As we modeled the GSP, we seek a minimum-cost flow removing one unit of flow from the source vertex such that flow at all other vertices is conserved. Thus, we have the following lemma.

Lemma 2 *For any feasible single-source generalized shortest path problem instance with bounded minimum-cost, there exists a solution consisting of an augmented path.*

Proof: Given the five types of primitive flows, we need not consider the first two types because they violate flow conservation because they have sink vertices. If a solution consists of two or more augmented paths, sending all the flow on a cheapest augmented path also suffices. Bicycles and breakeven cycles do not remove any flow from the source vertex. If a solution includes a positive-cost bicycle or a breakeven cycle, excluding it will yield a cheaper solution. Zero-cost bicycles and breakeven cycles can be also be omitted. If a negative-cost bicycle or breakeven cycle is present, its flow can be increased any arbitrary amount without violating the problem's definition so the instance has unbounded minimum-cost. □

We now derive bounds on the number of bits necessary to represent a problem instance's optimal cost. First, we present a formula for an augmented path's cost.

Lemma 3 *The cost of an augmented path $s \rightsquigarrow v \rightsquigarrow w \to v$ with one unit of flow at its source vertex is*

$$c(s \rightsquigarrow v) + \frac{c(v \rightsquigarrow w \to v)}{1 - \mu(v \rightsquigarrow w \to v)}.$$

Proof: First, we note that specifying a path and the amount of flow at its initial vertex uniquely determines the flow on each path arc because flow is conserved at each of the path's internal vertices.

Second, we derive a formula for the flow $f(v)$ leaving the junction vertex v. Because flow is conserved at the vertex,

$$\mu(s \rightsquigarrow v) + \mu(v \rightsquigarrow w \to v) f(v) = f(v)$$

or

$$f(v) = \frac{\mu(s \rightsquigarrow v)}{1 - \mu(v \rightsquigarrow w \to v)}.$$

(Since the cycle is lossy, the denominator is nonzero.)

Since a flow's cost is a linear function of the flow, the claim follows. □

A problem instance's solution can be represented using at most $O(\log C + n \log B)$ bits, assuming all arc costs are integral and can be represented using at most $\lg C$ bits. Let all flow multipliers be represented as rational numbers with integral numerators and denominators having absolute value at most B. For simplicity, let us concentrate on an upper bound for the instance's solution. The maximum cost of a path is $C(\frac{B^n - 1}{B - 1})$, which is bounded by $O(CB^n)$. Using the previous lemma, the number of solution bits is asymptotically bounded by $\lg(CB^n)$ plus the number of bits needed to represent the largest possible $1 - \mu(v \rightsquigarrow w \to v)$. Since the latter addend has asymptotically at most $2n \lg B$ bits, the claim follows.

Lemma 4 *Any bounded solution to a GSP problem instance can be represented using* $O(\log C + n \log B)$ *bits.*

6.3 The Dual Linear Program

To prove the correctness of the dynamic-programming comparison subroutine (Subroutine 2) of Section 6.5, we introduce the dual linear program. We also define reduced costs.

The GSP dual linear program is

$$\begin{aligned}&\text{maximize } \pi(s)\\&\text{subject to } (\forall \text{ arcs } (v,w) \in A)\,(\pi(v) - \mu(v,w)\pi(w) \le c(v,w))\end{aligned}$$

where $\pi(v)$ represents v's dual variable and can attain any real value. The objective function is so simple because the flow is conserved at all vertices except the source vertex s. By the duality theorem of linear programming (see, e.g., [Sch86, Corollary 7.1g]), $\pi(s)$ equals the cost of the minimum-cost augmented path. Thus, determining the maximum value of $\pi(s)$ determines the cost of the minimum-cost augmented path.

Just as for ordinary shortest path, we define the *reduced cost* $c^\pi(v,w)$ of an arc (v,w) as

$$c^\pi(v,w) = c(v,w) + \mu(v,w)\pi(w) - \pi(v).$$

The dual program's constraints can be written as requiring nonnegative reduced costs for all arcs. Complementary slackness conditions (see, e.g., [Sch86, Section 7.9]) imply flow on an arc can be positive only if its reduced cost is zero.

6.4 The Left-Distributive Closed Semiring

In this section, we present a left-distributive closed semiring used in the dynamic-programming comparison subroutine (Subroutine 2) of the next section.

The algebraic structure called a *closed semiring* is a system $(S, \oplus, \odot, \bar{0}, \bar{1})$, where S is a set of elements and $\oplus$ and $\odot$ are binary operators on S satisfying these properties:

1. $(S, \oplus, \bar{0})$ and $(S, \odot, \bar{1})$ are monoids and $\bar{0}$ is an annihilator with respect to $\odot$.

2. $\oplus$ is commutative and idempotent.

3. $\odot$ distributes over $\oplus$.

4. The sum with respect to $\oplus$ of a countable number of elements exists and is unique. Associativity, commutativity, and idempotence apply to all finite and infinite sums.

5. $\odot$ distributes over all countable sums.

A *monoid* is a closed set with an operator and an identity. An *annihilator* $\bar{0}$ is an element such that its product with any other element yields $\bar{0}$.

These algebraic structures form the basis for many shortest path algorithms [AHU74, CLR90]. For example, the Floyd-Warshall algorithm [Flo62, War62], transitive closure algorithms, and the Bellman-Ford dynamic-programming algorithm [Bel58, For56] are all based on these structures. (Algorithms based on semirings usually use algebraic techniques. For example, the dynamic-programming algorithm is based on the Jacobi iteration method [Jac45].)

No closed semiring for generalized flows is known. Instead we present a *left-distributive closed semiring*. The definition of closed semirings requires both left and right distributivity:

$$\begin{aligned} \text{left distributivity:} &\quad (b \oplus c) \odot a = (b \odot a) \oplus (c \odot a) \\ \text{right distributivity:} &\quad a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c), \end{aligned}$$

but we will require only left-distributivity for both finite and infinite sums.

The domain of the generalized flow left-distributive closed semiring $(S, \oplus, \odot, \bar{0}, \bar{1})$ is $S = (\mathbf{R}, \mathbf{R} > 0) \cup \{(-\infty, 0)\}$. It has ordered pairs (c, μ), corresponding to the costs and flow multipliers of paths. The *extension operator* $\odot$ computes the cost and flow multiplier of the concatenation of two paths:

$$(c_1, \mu_1) \odot (c_2, \mu_2) = (c_1 + \mu_1 c_2, \mu_1 \mu_2).$$

This operation is analogous to *reversed functional composition*, i.e., the functional composition of the second operand's line with the first operand's line. The extension identity $\bar{1}$ is $(0, 1)$, i.e., the cost and flow multiplier of an empty path.

The *summary operator* $\oplus$ is relative to a fixed value ϖ.

$$(c_1, \mu_1) \oplus (c_2, \mu_2) = \operatorname{argmax}((\varpi - c_1)/\mu_1, (\varpi - c_2)/\mu_2)$$

That is, it returns the element corresponding to the path with the larger value $(\varpi - c)/\mu$. If a multiplier is zero, the division is defined to yield $-\infty$. The corresponding summary identity is $\bar{0} = (-\infty, 0)$ is less than all other elements.

We now sketch the proof that this algebraic structure $(S, \oplus, \odot, \bar{0}, \bar{1})$ for a fixed value ϖ is a left-distributive closed semiring. $(S, \oplus, \bar{0})$ is a monoid because it is closed, argmax is associative, and $\bar{0}$ is an identity for argmax. Likewise, $(S, \odot, \bar{1})$ is a monoid because the cost and multiplier of the concatenation of two paths is in the set, $\odot$ is associative, and $\bar{1}$ is an identity. Extending a path with $\bar{0}$ yields $\bar{0}$ so it is an annihilator. (We define a product or sum with $-\infty$ yields $-\infty$.)

To prove that $\oplus$ is commutative and idempotent, we note this is true of argmax. Furthermore, the argmax of any countable number of elements exists and is unique, and it is associative, commutative, and idempotent for all countable sums.

To prove left-distributivity, let us assume that

$$(c_1, \mu_1) \oplus (c_2, \mu_2) = (c_1, \mu_1),$$

i.e., that

$$(\varpi - c_1)/\mu_1 \geq (\varpi - c_2)/\mu_2.$$

Thus,

$$\begin{aligned} ((c_1, \mu_1) \oplus (c_2, \mu_2)) \odot (c_3, \mu_3) &= (c_1, \mu_1) \odot (c_3, \mu_3) \\ &= (c_1 + \mu_1 c_3, \mu_1 \mu_3). \end{aligned}$$

On the other hand,

$$\begin{aligned} ((c_1, \mu_1) \odot (c_3, \mu_3)) \oplus ((c_2, \mu_2) \odot (c_3, \mu_3)) &= (c_1 + \mu_1 c_3, \mu_1 \mu_3) \oplus (c_2 + \mu_2 c_3, \mu_2 \mu_3) \\ &= \operatorname{argmax}\left(\frac{\varpi - (c_1 + \mu_1 c_3)}{\mu_1 \mu_3}, \frac{\varpi - (c_2 + \mu_2 c_3)}{\mu_2 \mu_3}\right) \\ &= \operatorname{argmax}\left(\frac{\frac{\varpi - c_1}{\mu_1} - c_3}{\mu_3}, \frac{\frac{\varpi - c_2}{\mu_2} - c_3}{\mu_3}\right) \\ &= (c_1 + \mu_1 c_3, \mu_1 \mu_3). \end{aligned}$$

Thus, left distributivity holds. It is easy to see this extends to left-distributivity over countable sums.

The system is not right-distributive. For example, consider $\varpi = 0$, $a = (4,1)$, $b = (1,0.5)$, and $c = (2.01,1)$.

$$\begin{aligned}(4,1) \odot \big((1,0.5) \oplus (2.01,1)\big) &= (4,1) \odot (1,0.5)\\ &= (5,0.5)\end{aligned}$$

while

$$\begin{aligned}\big((4,1) \odot (1,0.5)\big) \oplus \big((4,1) \odot (2.01,1)\big) &= (5,0.5) \oplus (6.01,1)\\ &= (6.01,1).\end{aligned}$$

6.5 The Dynamic-Programming Comparison Subroutine

In this section, we present a dynamic-programming comparison subroutine for the GSP using the generalized flow left-distributive closed semiring presented in the previous section. Given a guess ϖ, the algorithm indicates whether the guess is smaller than, equal to, or larger than the cost of the minimum-cost augmented path. If correct, it also yields the optimal path.

The algorithm is very similar to the Bellman-Ford shortest path algorithm [Bel58, For56]. (For textbook presentations of that algorithm, see, e.g., [CLR90, Section 25.3] or [Law76, Section 3.6].) The algorithm initially assigns default *vertex potentials* $\pi(v)$, i.e., values, to each vertex v. The algorithm works by repeatedly increasing vertex potentials. At the end of the ith iteration, the algorithm has a tree of arcs with zero reduced cost maximizing the vertex potentials subject to the tree's depth being at most i. Since the longest path in a directed graph has $n-1$ arcs, the iterations terminate with a tree containing all the vertices connected by arcs with zero reduced cost. Then, the algorithm checks the arcs' reduced costs to determine whether the guess was smaller than, equal to, or larger than the optimal cost. If any arc has negative reduced cost, then the guess was too big. If a depth-first search reveals no augmented path exists in the subgraph of zero-reduced-cost arcs, then the guess was too small. Otherwise, the guess was correct.

Subroutine 2 Dynamic-Programming Comparison Subroutine for the GSP

Input: a guess ϖ for the cost $\pi^*(s)$ of the minimum-cost augmented path with initial vertex s

Output: the guess ϖ is smaller than, equal to, or larger than the minimum cost $\pi^*(s)$

// Initialization
  for all vertices v **do**
    $\pi^0(v) \leftarrow \begin{cases} \bar{1} & \text{if } s = v \\ \bar{0} & \text{otherwise} \end{cases}$

// Recurrences
  for iterations $i = 1, \ldots, n-1$ **do**
    for all vertices v **do**
      $\pi^i(v) \leftarrow \pi^{i-1}(v) \oplus \bigoplus_{\text{arcs } w \to v} \left(\pi^{i-1}(w) \odot (c(w \to v), \mu(w \to v))\right)$

// Check for correctness of guess ϖ.
  if there exists an arc with negative reduced cost **then**
    return "$\varpi > \pi^*(s)$"
  else if there does not exist an augmented path in the subgraph of zero reduced cost arcs **then**
    return "$\varpi < \pi^*(s)$"
  else
    return "$\varpi = \pi^*(s)$"

We now turn to proving the subroutine's correctness, running time, and space requirements. First, we will present a lemma relating the potentials of the vertices on a path of zero-reduced-cost arcs.

Lemma 5 *Given any path $v \rightsquigarrow w$ having only arcs with zero reduced cost and vertex potential $\pi(v) = \varpi$, the vertex potential $\pi(x)$ for vertex x on the path is*

$$\frac{1}{\mu(v \rightsquigarrow x)} \left(\varpi - c(v \rightsquigarrow x)\right).$$

Proof: To prove the claim, we use the recursive definition of a path. For an empty path, its multiplier is 1 and its cost is 0 and x equals v so the claim is true.

Consider a path $v \rightsquigarrow w \to x$. Since the last arc has zero reduced cost, $c(w \to x) +$

$\mu(w \to x)\pi(x) - \pi(w) = 0$, i.e.,

$$\begin{aligned}
\pi(x) &= \frac{1}{\mu(w \to x)} \left(\pi(w) - c(w \to x)\right) \\
&= \frac{1}{\mu(w \to x)} \left(\frac{1}{\mu(v \leadsto w)} \left(\varpi - c(v \leadsto w)\right) - c(w \to x) \right) \\
&= \frac{1}{\mu(v \leadsto w)\mu(w \to x)} \left(\varpi - \left(c(v \leadsto w) + \mu(v \leadsto w) c(w \to x)\right)\right) \\
&= \frac{1}{\mu(v \leadsto w \to x)} \left(\varpi - c(v \leadsto w \to x)\right).
\end{aligned}$$

The second equality follows from the inductive hypothesis and the last equality follows from the definitions of flow multipliers and costs for paths. □

Lemma 6 *Consider an augmented path $s \leadsto v \leadsto w \to v$ having only arcs with zero reduced cost except possibly the extra arc $w \to v$. Let $\pi(s)$ denote the path's cost. If vertex s has potential ϖ, then,*

$$\begin{aligned}
c^\pi(w \to v) < 0 &\iff \varpi > \pi(s) \\
c^\pi(w \to v) = 0 &\iff \varpi = \pi(s) \\
c^\pi(w \to v) > 0 &\iff \varpi < \pi(s).
\end{aligned}$$

Proof: Using Lemma 5, the reduced cost $c^\pi(w \to v)$ equals

$$\begin{aligned}
&c(w \to v) + \mu(w \to v)\pi(v) - \pi(w) \\
&\quad = c(w \to v) + \mu(w \to v) \frac{1}{\mu(s \leadsto v)} \left(\varpi - c(s \leadsto v)\right) - \frac{1}{\mu(s \leadsto w)} \left(\varpi - c(s \leadsto w)\right) \\
&\quad = \left(\frac{\mu(w \to v)}{\mu(s \leadsto v)} - \frac{1}{\mu(s \leadsto w)} \right) \varpi + (\text{terms independent of } \varpi) \\
&\quad = \left(\frac{\mu(w \to v)}{\mu(s \leadsto v)} - \frac{1}{\mu(s \leadsto v)\mu(v \leadsto w)} \right) \varpi + (\text{terms independent of } \varpi) \\
&\quad = \frac{\mu(w \to v)}{\mu(s \leadsto v)} \left(1 - \frac{1}{\mu(v \leadsto w \to v)} \right) \varpi + (\text{terms independent of } \varpi).
\end{aligned}$$

As shown in Section 6.3, the reduced cost is zero for $\varpi = \pi(s)$. Since $v \leadsto w \to v$ is lossy, the coefficient of ϖ is negative. Thus, using $\varpi > \pi(s)$ decreases the reduced cost and using $\varpi < \pi(s)$ increases the reduced cost. □

Lemma 7 *The dynamic-programming comparison subroutine (Subroutine 2) yields a correct answer.*

Proof: The algorithm requires only left-distributivity, not right-distributivity. This is because the right operand of the recurrence equation's extension operator $\odot$ is a single arc, not the summary of more than one path. As we proved in the previous section, we are using a left-distributive closed semiring so, by the correctness of the Bellman-Ford algorithm, the dynamic-programming subroutine guarantees all arcs of the minimum-cost augmented path except possibly the extra arc have zero reduced cost. Using the previous lemma, this extra arc's reduced cost will be negative if and only if ϖ is too large. If the guess is too small for the minimum-cost augmented path, it is too small for all augmented paths. Thus, the subgraph of arcs with zero reduced cost will be cycle-free. Otherwise, all arcs in the minimum-cost augmented path have zero reduced cost, and ϖ is the problem's optimal value. The minimum-cost augmented path is any augmented path in the subgraph of arcs with zero reduced cost and can be found using depth-first search. □

Lemma 8 *The dynamic-programming comparison subroutine (Subroutine 2) requires $O(mn)$ time and $O(m + n)$ space.*

Proof: Each iteration requires a constant number of operations per vertex and each arc participates in exactly one operation. Each vertex's current and previous potential needs to be stored and the arc costs and multipliers must be stored. Scanning all arcs' reduced costs indicates if any arc has negative reduced cost. A depth-first subroutine can discover if there exists a zero reduced-cost augmented path. □

6.6 Weakly-Polynomial Algorithms

Combining the comparison subroutine with binary search yields an algorithm requiring $O(\log C + n \log B)$ iterations using the bound derived in Lemma 4.

Lemma 9 *An exact solution to the generalized shortest path problem can be computed in $O(mn(\log C + n \log B))$ time and $O(m+n)$ space, assuming all arc costs have integral value between $-C$ and C and all flow multipliers can be represented as rational numbers using integers with absolute value at most B.*

Fleischer and Wayne [WF99] use *geometric search* [Has92] to produce a faster fully polynomial approximation scheme when the optimal solution is nonnegative. Suppose an approximation within a factor of $1+\epsilon$ is desired and the optimal solution is known to be in the range $[L, U]$. One invocation with a guess of zero will reveal whether the optimal answer is positive. Assuming it is, we can now perform binary search over the range $[\log_{1+\epsilon} L, \log_{1+\epsilon} U]$. Each invocation corresponds to a guess of the geometric mean $\sqrt{LU} = (1+\epsilon)^{(1/2)(\log_{1+\epsilon} L + \log_{1+\epsilon} U)}$ and halves the logarithmic range. Since $\log(\log(U/L)/\epsilon)$ iterations are required, the following lemma holds.

Lemma 10 ([WF99]) *Assuming the optimal solution is nonnegative, an ϵ-optimal solution can be computed in $O(mn \log(\epsilon^{-1}(\log C + n \log B)))$ time and $O(m+n)$ space.*

All of these running times are *weakly polynomial*, i.e., they depend on the value of the arc costs and arc flow multipliers. In the next section, we will combine the dynamic-programming comparison subroutine with ideas from Megiddo's parametric search to obtain a strongly polynomial algorithm for the GSP.

6.7 Parametric Search for the Optimal Value

We present a strongly polynomial algorithm for the single-source generalized shortest path problem (GSP). The algorithm sketched in the previous section consisted of binary search to choose ϖ with an inner loop indicating whether a particular guess was too small, correct, or too large. Our strongly polynomial algorithm will interleave searching for the optimal value with recursive calls. This will eliminate the dependence of the number of dynamic-programming comparisons on the flow multipliers and the arc costs.

We represent each vertex potential (c, μ) as a line $(\varpi - c)/\mu$, where ϖ denotes the source vertex's potential. Initially, we know the answer $\pi^*(s)$ is either in a finite range $[b, e]$ or the problem has unbounded minimum cost. Each iteration will narrow the range by invoking the dynamic-programming comparison subroutine for several values in the range.

Before describing the algorithm, let us consider the summary operator $\oplus$. Each dynamic-programming recurrence requires using $\oplus$, and this operator requires having a value ϖ sufficiently close to the optimal value $\pi^*(s)$ to compute the correct answer. Since each vertex potential is a line, the two vertex potentials correspond to two lines in the positive-slope (ϖ, π) Cartesian plane. We wish to determine which lines has the maximum value

Algorithm 3 Strongly Polynomial GSP Algorithm

Input: directed graph G and source vertex s
Output: cost of the minimum-cost augmented path
// Initialization
  $[b, e] \leftarrow$ largest possible range for the instance's minimum cost $\pi^*(s)$
  for all vertices v **do**
    $\pi^0(v) \leftarrow \begin{cases} \bar{1} & \text{if } s = v \\ \bar{0} & \text{otherwise} \end{cases}$
// Recurrences
  for iterations $i = 1, \ldots, n-1$ **do**
    for all vertices v **do**
      $IP(v) \leftarrow$ sorted sequence of intersection points from
        $\pi^{i-1}(v) \oplus \bigoplus_{\text{arcs } w \to v} \left(\pi^{i-1}(w) \odot (c(w \to v), \mu(w \to v))\right)$
    $IP \leftarrow$ mergesort of all $IP(v)$ {IP is an ordered list of intersection points.}
    Binary search with Subroutine 2 on IP to determine an intersection-free interval containing $\pi^*(s)$.
    $[b, e] \leftarrow$ intersection-free interval containing $\pi^*(s)$
    for all vertices v **do**
      Determine $\pi^i(v)$ as the linear function of the interval containing $[b, e]$.
  return b

for $\varpi = \pi^*(s)$, but we do not know $\pi^*(s)$. If the lines intersect, they do so only at one point. Using the dynamic-programming comparison subroutine at this intersection point will indicate on which side of the point the optimal value $\pi^*(s)$ resides and which line has larger value. (If we are lucky, we will discover the optimal value.)

The parametric algorithm (Algorithm 3) has similar structure to the dynamic comparison subroutine (Subroutine 2). First, vertex potentials are assigned default values and then $n-1$ iterations occur. Each vertex's new potential is computed using the summary operator $\oplus$. Instead of computing them sequentially, all of the intersection points IP are collected together. Using binary search and Subroutine 2, a new range for ϖ is determined such that no vertices' potentials intersect within the range. Thus, their potentials can be resolved, and the iteration ends. After all the iterations, the smallest value in $[b, e]$ is the answer. One more dynamic-programming subroutine invocation using this value will yield the augmented path.

Lemma 11 *The parametric search Algorithm 3 solves the GSP problem and requires $O(mn^2 \log m)$ time and $O(m+n)$ space.*

Proof: The correctness follows from the correctness of the dynamic programming subroutine and the invariant that, at the end of the ith iteration, the range for $[b, e]$ permits resolving all summary operations for trees with zero reduced arcs and having depth at most i.

To analyze the running time, we will denote a vertex v's degree $\delta(v)$. $IP(v)$ indicates all the intersection points required to resolve the summary operator for vertex v's recurrence equation

$$\pi^i(v) = \pi^{i-1}(v) \oplus \bigoplus_{\text{arcs } w \to v} \left(\pi^{i-1}(w) \odot (c(w \to v), \mu(w \to v))\right).$$

We analyze the running time of each of the $n-1$ iterations. Computing the intersection points, $IP(v)$ requires $O(\delta(v) \log \delta(v))$ time because an ordering can be defined on linear functions [Meg79, Appendix]. Thus, the first line requires $\sum_{v \in V} O(\delta(v) \log \delta(v)) \leq O(m \log m)$ time. The mergesort of n lists each having length at most m requires $O(m \log n)$ time. Since the merged list IP has at most $m + n$ intersection points, $O(\log m)$ dynamic-programming comparisons are required. The last step requires $O(m)$ time.

In addition to the $O(m+n)$ space requirements for the dynamic-programming comparison subroutine, this algorithm requires storing the number of intersection points, which is $m + n$. □

Chapter 7

Generalized Flow Approximation Algorithms

Using the single-source generalized shortest path (GSP) algorithms of the previous chapter, we develop fully polynomial-time approximation schemes for the generalized versions of maximum flow, nonnegative-cost minimum-cost flow, concurrent flow, multicommodity maximum flow, and multicommodity nonnegative-cost minimum-cost flow problems. For all but generalized maximum flow, these approximation schemes yield the first[1] polynomial-time algorithms not based on interior-point methods. Using the strongly polynomial algorithm (Algorithm 3) yields running times independent of the flow multipliers μ for all but the minimum-cost problems. As we noted on page 69, representing exact answers to many generalized flow problems can require a polynomial number of bits. Thus, fully polynomial approximation schemes are frequently sought. The only generalized problem known to have a strongly polynomial exact algorithm is the GSP. In 1994, Cohen and Megiddo [CM94b] presented a fully strongly polynomial approximation scheme for the generalized maximum flow problem. For the generalized concurrent flow and multicommodity maximum flow approximation schemes, we present the first strongly polynomial approximation schemes, for which the running time is strongly polynomial assuming a fixed accuracy. We also show that using the geometric search GSP approximate algorithm of Lemma 10 reduces the running times by a factor of n but adding a dependence on the size of the input data.

[1] Subsequent to this work, Kevin Wayne published a polynomial-time, combinatorial algorithm for the generalized minimum-cost flow problem [Way99].

To create the approximation schemes, we combine two different packing frameworks [GK96b, GK98] with generalized shortest path algorithms. Both fractional packing frameworks iteratively improve their solution by solving a generalized shortest path each iteration. This shortest path is combined with the previous solution to yield an improved solution. In addition to maintaining the current primal solution, the packing frameworks also have approximate values for the dual variables. These m variables specify the cost of each arc for the GSP computations. The frameworks differ according to how the flows are combined and how costs are computed. It is an open research question whether there exists one packing framework that will solve all these problems. For example, consider the generalized nonnegative-cost minimum-cost flow problem. Using the Grigoriadis-Khachiyan framework [GK96b], one can approximate the cost while ensuring all demands are met. Using the Garg-Könemann framework [GK98] which tries to maximize flow, one can instead send as much as flow as a given cost bound permits. The author will address the former and not the latter approximations.

The remainder of this chapter is split into four sections. Generalized problems with nonnegative costs are solved using the Grigoriadis-Khachiyan framework [GK96b]. The generalized concurrent flow problem is also solved using the same framework. Generalized maximum flow problems are solved using the Garg-Könemann framework [GK98]. In the last section, we consider incorporating problems with more constraints and also acyclic problems.

7.1 Problems with Nonnegative Costs

First, we will consider the generalized minimum-cost flow and generalized multicommodity minimum-cost flow problems restricted to nonnegative costs. Of all generalized flow problems, the *generalized minimum-cost flow problem* appears most frequently in the generalized flow literature. For example, an early generalized flow paper [Jew62] and the generalized flow chapter in Ahuja et al. [AMO93] concentrate on this problem.

In addition to GSP's input, the nonnegative-cost version requires an arc capacity function $u : A \to \mathbf{R} > 0$, a supply function $d : V \to \mathbf{R} > 0$, and restricts costs to be nonnegative. The goal is to find a flow of minimum cost that removes the desired amount from the source vertex and obeys the capacity constraints and flow multipliers. A flow on an arc obeys the arc's capacity constraint if the flow entering the arc is less than or equal to the arc's capacity. Writing these constraints as a linear program,

$$\begin{aligned}
&\text{Minimize} \quad \sum_{\text{arcs } (v,w)\in A} c(v,w)f(v,w) \\
&\text{subject to } (\forall \text{ vertices } v\in V)\left(\sum_{\{w:(v,w)\in A\}} f(v,w) - \sum_{\{w:(w,v)\in A\}} \mu(w,v)f(w,v) = d(v)\right) \\
&\qquad (\forall \text{ arcs } (v,w)\in A)(0\le f(v,w)\le u(v,w)).
\end{aligned}$$

When stating running times, we assume capacities and costs are integral and bounded by U and C, respectively.

Just as for the generalized shortest path problem, solutions consist of augmented paths. Since the arc capacity functions may restrict the flow on an augmented path, more one than such path may be necessary to remove all the source's supply.

An ϵ-approximate solution to the generalized minimum-cost flow problem exactly satisfies the supplies but may use arc capacities $(1+\epsilon)u$ and the flow's cost may be a factor $1+\epsilon$ larger than the minimum possible cost for a flow strictly obeying the arc capacity constraints.

With loss of generality, we consider problems with only one source vertex and without sink vertices. When solving these problems exactly, each sink vertex t with demand $d(t)$ can be replaced by a zero-cost arc with capacity $d(t)$ leading to a lossy cycle with infinite capacity and zero cost. Multiple sources each with supply $d(s)$ can be replaced by a supersource vertex with demand $\sum_s d(s)$ connected to each source s by an arc of zero cost and capacity $d(s)$. The approximation scheme yields flows at most $(1+\epsilon)u$ so these transformations no longer hold. The amount of flow reaching a sink t may exceed its demand by $\epsilon d(t)$. Similarly, some sources may send up to $1+\epsilon$ times their supply into the network. Instead of using the transformation, each source can be treated as a separate commodity to form a multicommodity flow instance. The approximation scheme ensures each source sends exactly its supply into the network, but the running time and space requirements are multiplied by the number of sources.

The Grigoriadis-Khachiyan fractional packing framework, based on Lagrangian decomposition methods, assumes a problem's constraints can be split into "difficult" coupling constraints and "easy" constraints [GK96b]. Removing the coupling constraints yields independent problems consisting only of "easy" constraints. Given any solution satisfying

Algorithm 4 ϵ-Generalized Multicommodity Nonnegative-Cost Cost-Bounded Flow Algorithm

Input: desired accuracy ϵ and initial flow f satisfying supplies
Output: flow f satisfying supplies and obeying constraints up to $1+\epsilon$ factor
loop
  Compute arc costs $c(f)$ using a potential function of constraints under the current flow f.
  for all commodities i **do**
    $f_i^* \leftarrow \mathrm{GSP}_i(c(f))$
    Scale f_i^* to satisfy supply d_i.
  if the difference between f and f^*'s costs is small enough **then**
    return f
  else
    $f \leftarrow$ convex combination $(1-\sigma)f + \sigma f^*$

these constraints, a logarithmic potential function summarizes the coupling constraints' values. A highly violated constraint contributes much more to the potential function than slightly violated or unviolated constraints. To improve a solution, prices are computed using the potential function and a minimum-cost solution obeying the easy constraints is computed. The solution is updated using the convex combination of the current solution and the minimum-cost solution. Grigoriadis and Khachiyan prove a polynomial number of iterations are needed to produce an $1+\epsilon$ optimal solution.

Theorem 12 ([GK96b, Thm. 3]) *For a given accuracy $\epsilon \in (0,1]$, an ϵ-optimal solution x to the fractional packing problem with m coupling constraints and k independent problems requires $O(m(\epsilon^{-2}\log\epsilon^{-1} + \log m))$ iterations. Each iteration consists of k minimum-cost optimizations solved to an accuracy of $O(1+\epsilon)$ and $O(m\log\log(m/\epsilon))$ other operations. The space requirements are $O(km)$ in addition to the space requirements for the k optimizations.*

We solve the generalized nonnegative-cost minimum-cost flow problem by solving the generalized nonnegative-cost cost-bounded flow problem using the Grigoriadis-Khachiyan framework. The *generalized cost-bounded flow problem* is the same as the generalized minimum-cost flow problem except the latter's objective function is replaced by a constraint bounding the maximum permitted flow cost. The cost-bounded problem's goal is to find a feasible flow, i.e., one obeying the arc capacities and having cost at most the bound.

Lemma 13 *A combinatorial ϵ-approximation algorithm (Algorithm 4 specialized to one commodity) solves the generalized nonnegative-cost cost-bounded flow problem in* $O((\epsilon^{-2}\log\epsilon^{-1} + \log m)m(mn^2\log m + \log\log(m/\epsilon)))$ *time and* $O(m+n)$ *space. It can be solved in weakly polynomial time in* $O((\epsilon^{-2}\log\epsilon^{-1} + \log m)m(mn\log(\epsilon^{-1}(\log(m/\epsilon) + n\log B)) + \log\log(m/\epsilon)))$ *time and the same space.*

Proof: We solve the nonnegative-cost cost-bounded problem using the Grigoriadis-Khachiyan framework. For pseudocode, see Algorithm 4 restricted to just one commodity. The $m+1$ coupling constraints consist of the m arc capacity constraints $f(v,w) \leq u(v,w)$ and the cost constraint $\sum_{\text{arcs } (v,w)} c(v,w)f(v,w) \leq \mathcal{B}$. The independent problem consists of the n flow conservation constraints and the m nonnegativity flow constraints, i.e., a generalized shortest path polyhedron. Since an exact generalized shortest path can be computed in $O(mn^2\log m)$ time, the total running time is $O((\epsilon^{-2}\log\epsilon^{-1}+\log m)m(mn^2\log m + \log\log(m/\epsilon)))$. The space requirements are $O(m+n)$, the same as for the GSP.

To obtain a weakly polynomial time bound, we can reduce the running time by approximately a factor of n using the approximate GSP algorithm of Lemma 10. Since the algorithm invokes the GSP subroutine with nonnegative arc costs of $O(\log(m/\epsilon))$ bits, the running time of each GSP is $O(mn\log(\epsilon^{-1}(\log(m/\epsilon) + n\log B)))$. $\square$

To compute a minimum-cost solution, one can use the cost-bounded algorithm and binary search on the range of possible costs. mCU is an upper bound, but the smallest positive cost may have $O(\log C + n\log B)$ bits. This requires $O((\log(mCU) + n\log B)/\epsilon)$ iterations. Only $O(\log((\log(mCU) + n\log B)/\epsilon))$ iterations are required using geometric search, which we described in Section 6.6.

Lemma 14 *A combinatorial ϵ-approximation algorithm (Algorithm 4 specialized to one commodity) solves the generalized nonnegative-cost minimum-cost flow problem in* $O((\epsilon^{-2}\log\epsilon^{-1}+\log m)m^2 n\log(\epsilon^{-1}(\log(m/\epsilon)+n\log B))\log((\log(mCU)+n\log B)/\epsilon))$ *time and* $O(m+n)$ *space.*

The *generalized multicommodity minimum-cost flow problem* models sharing of network resources by several different commodities. The generalized minimum-cost flow problem specifies one source vertex and its supply, while the multicommodity version specifies k sources each with its own supply. All of the commodities' flows simultaneously share the network while the total flow, over all the commodities, must still obey the arc capacity constraints and minimizing the total cost. Written as a linear program,

$$
\begin{aligned}
\text{Min} \quad & \sum_{\text{commodities } i} \; \sum_{\text{arcs } (v,w)\in A} c_i(v,w) f_i(v,w) \\
\text{s.t.} \quad & (\forall i)(\forall v \in V) \left(\sum_{\{w:(v,w)\in A\}} f_i(v,w) - \sum_{\{w:(w,v)\in A\}} \mu(w,v) f_i(w,v) = d_i(v)[v \text{ is } i\text{'s source}] \right) \\
& (\forall \text{ arcs } (v,w) \in A) \left(\sum_{\text{commodities } i} f_i(v,w) \le u(v,w) \right) \qquad (7.1) \\
& (\forall \text{ comm. } i)(\forall \text{ arcs } (v,w) \in A)(f_i(v,w) \ge 0)
\end{aligned}
$$

$f_i(v,w)$ denotes the flow of commodity i on arc (v,w).

We use the Grigoriadis-Khachiyan framework to solve the multicommodity flow problem. Removing the joint arc capacity constraints (7.1) yields independent problems of computing each commodity's generalized shortest path. Thus, the algorithm iteratively improves each commodity, as shown in Algorithm 4. The time and space requirements increase by a factor of k.

Lemma 15 *An ϵ-approximate solution to the generalized multicommodity minimum-cost flow problem can be found in $O((\epsilon^{-2}\log\epsilon^{-1} + \log m)km^2 n \log(\epsilon^{-1}(\log(m/\epsilon) + n\log B))$ $\log((\log(mCU) + n\log B)/\epsilon))$ time and using $O(k(m+n))$ space.*

7.2 Generalized Concurrent Flow

The *generalized concurrent flow problem* models fair sharing of network resources while trying to remove as much of each user's supply as possible. The objective function is to maximize the fraction of all the commodities' supplies that can be simultaneously satisfied given the network's arc capacity constraints. Alternatively, the objective function is to minimize the multiple of the arc capacities necessary to simultaneously satisfy all the commodities' demands. Written as a linear program,

$$\text{Min } \lambda$$

$$\text{s.t. } (\forall i)(\forall v \in V)\left(\sum_{\{w:(v,w)\in A\}} f_i(v,w) - \sum_{\{w:(w,v)\in A\}} \mu(w,v) f_i(w,v) = d_i(v)[v \text{ is } i\text{'s source}]\right)$$
$$(\forall \text{ arcs } (v,w) \in A)\left(\sum_{\text{commodities } i} f_i(v,w) \leq \lambda u(v,w)\right)$$
$$(\forall \text{ comm. } i)(\forall \text{ arcs } (v,w) \in A)(f_i(v,w) \geq 0)$$

We use the Grigoriadis-Khachiyan framework to solve the generalized concurrent flow problem.

Lemma 16 *The generalized concurrent flow problem can be solved within a factor of* $1+\epsilon$ *of the optimal answer in* $O((\epsilon^{-2}\log\epsilon^{-1} + \log m)m(kmn^2\log m + \log\log(m/\epsilon)))$ *time and space* $O(k(m+n))$. *Using approximate GSP computations, it can also be solved in* $O((\epsilon^{-2}\log\epsilon^{-1} + \log m)km^2n\log(\epsilon^{-1}(\log(m/\epsilon) + n\log B)))$ *weakly polynomial time.*

7.3 Maximizing Flow

We solve the generalized maximum flow and generalized multicommodity maximum flow problems using the framework of Garg and Könemann [GK98]. The *generalized maximum flow problem* is to maximize the net flow out of a source vertex. That is, given a directed graph $G = (V, A)$, an arc multiplier function $\mu : A \to \mathbf{R} > 0$, an arc capacity function $u : A \to \mathbf{R} > 0$, and a source vertex $s \in V$, find a flow function $f : A \to \mathbf{R} \geq 0$ obeying flow conservation at all vertices except the source, the multiplier function, and the capacity function while maximizing the flow out of the source vertex. An ϵ-optimal solution is one within at least a $1-\epsilon$ factor of the optimal value while obeying the capacity constraints.

The *generalized multicommodity maximum flow problem* models sharing of the network by several different commodities. Each commodity has its own source vertex. The goal is to maximize the total flow out of the source vertices. Note the problem is distinct from its single-commodity version only if the arcs have different arc multipliers or capacities for

Algorithm 5 ϵ-Generalized Maximum Flow Algorithm

$(\forall$ arcs $(v,w))(c(v,w) \leftarrow \iota)$ $\{\iota$ is a very small positive number.$\}$
flow $f \leftarrow 0$
loop
  $f^* \leftarrow \mathrm{GSP}(c)$
  Scale f^* so all arc capacities obeyed and at least one capacity constraint becomes tight.
  $(\forall$ arcs $(v,w))(c(v,w) \leftarrow c(v,w)(1+\alpha f^*(v,w)/u(v,w)))$
  if $c \cdot f^* \geq 1$ **then**
    return f scaled down to be feasible.
  $f \leftarrow f + f^*$

different commodities. Writing these requirements as a linear program,

$$\begin{aligned}
\text{Max} \quad & \sum_{\text{commodities } i} |f_i| \\
\text{s.t.} \quad & (\forall i)(\forall v \in V)\left(\sum_{\{w:(v,w)\in A\}} f_i(v,w) - \sum_{\{w:(w,v)\in A\}} \mu_i(w,v) f_i(w,v) = d_i(v)[v \text{ is } i\text{'s source}]\right) \\
& (\forall \text{ arcs } (v,w) \in A)\left(\sum_{\text{commodities } i} f_i(v,w) \leq u(v,w)\right) \\
& (\forall \text{ comm. } i)(\forall \text{ arcs } (v,w) \in A)(f_i(v,w) \geq 0)
\end{aligned}$$

$|f_i|$ denotes the flow of commodity i out of its source vertex. $\mu_i(v,w)$ denotes the flow multiplier for commodity i on arc (v,w).

Multicommodity maximum flow differs from multicommodity minimum-cost flow and concurrent flow because an optimal solution may have no flow for some commodities. The latter two problems require that each commodity carries a specified amount of flow (or a multiple of the specified amount).

Analogous to the Grigoriadis-Khachiyan framework, the Garg-Könemann framework assumes a problem's constraints can be split into difficult "coupling" constraints and "easy" constraints. Removing the difficult constraints yields independent problems consisting only of "easy" constraints. Instead of rerouting flow to inexpensive paths, the framework augments the flow along inexpensive paths. As constraints become more saturated, the costs of the arcs that participate in those constraints increase. Since flow is augmented along minimum-cost paths, saturated arcs are used only if necessary. While the algorithm runs the capacity constraints may be violated, but the final flow is scaled to be feasible.

Theorem 17 ([GK98, Thm. 3.1]) *There is an $(1-\epsilon)$-approximation algorithm for the packing linear program problem with running time $O(\epsilon^{-2}k\kappa T_{sp}\log L)$ and space requirements $O(\kappa m+S_{sp})$, where T_{sp} and S_{sp} are the time and space requirements to compute a minimum-cost solution with nonnegative arc costs. κ and L denote the number of constraints and the largest possible number of arcs in a minimum-cost solution.*

We solve the generalized maximum flow and the generalized multicommodity maximum-flow problems using the Garg-Könemann framework.

Lemma 18 (strongly polynomial) *An ϵ-optimal solution to the generalized maximum flow problem can be found in $O(\epsilon^{-2}m^2n^2\log m\log n)$ time and $O(m+n)$ space. The generalized multicommodity maximum flow problem can be solved in $O(\epsilon^{-2}km^2n^2\log m\log n)$ time and $O(k(m+n))$ space.*

Proof: Algorithm 5 outlines the Garg-Könemann [GK98] algorithm for the single commodity problem. The algorithm maintains arc costs that are almost exponential in the current flow's use of the arc capacities. Using these arc costs, a GSP is computed and scaled so at least one arc becomes saturated. This scaled flow is added to the existing flow. When the cost of the GSP is large enough, the current flow is scaled to be feasible and is returned. To solve the multicommodity version, each iteration computes a GSP for each commodity and the cheapest one is scaled and added to the current flow.

Using a generalized shortest path in the previous theorem shows the running time is $O(\epsilon^{-2}km^2n^2\log m\log n)$ and the space requirements are $O(k(m+n))$. Restricting to one commodity ($k=1$) gives the time and space requirements for the generalized maximum-flow problem. □

As for the other generalized flow problems, we can use the geometric search approximation algorithm to reduce the running time. Fleischer and Wayne show that an $1+\epsilon$-approximate GSP solution suffices for the Garg-Könemann framework [WF99]. The framework produces nonnegative arc costs in the range $[\delta,1]$, where $\delta=\Theta(m^{-1/\epsilon})$. By Lemma 10, an approximate GSP solution can be computed in $O(mn\log(\epsilon^{-1}(\epsilon^{-1}\log m+n\log B)))$ time and $O(m+n)$ space.

Lemma 19 (weakly polynomial) *An ϵ-optimal solution to the generalized maximum flow problem can be found in $O(\epsilon^{-2}m^2n\log n\log(\epsilon^{-1}(\epsilon^{-1}\log m+n\log B)))$ time and $O(m+n)$*

space. The generalized multicommodity maximum flow problem can be solved in $O(\epsilon^{-2}km^2n\log n\log(\epsilon^{-1}(\epsilon^{-1}\log m + n\log B)))$ *time and* $O(k(m+n))$ *space.*

The running times of Radzik and Tardos and Wayne's approximation algorithms [Rad93, TW98] for the generalized maximum flow problem dominate the strongly and weakly polynomial respectively, running times in the previous lemmas, but we have included the algorithm for completeness.

Lemma 20 *The generalized nonnegative-cost cost-bounded flow, generalized concurrent flow, generalized multicommodity maximum flow, and generalized multicommodity nonnegative-cost cost-bounded flow problems can be* ϵ*-approximately solved in strongly polynomial time.*

Proof: This follows directly from Lemmas 13, 15, 16, and 18. □

7.4 Extensions

In the previous sections, we used fractional packing frameworks to solve the generalized versions of canonical network flow problems. Both frameworks assumed a problem's constraints can be categorized as either difficult constraints or easy constraints, and we used generalized shortest path algorithms to solve the easy constraints. In this section, we describe various constraints we can incorporate into our model, and we show that the ability to solve generalized shortest path problems more quickly permits solving these problems more quickly.

Augmenting the Constraints We can augment the constraints in our models. For example, our presentation of multicommodity flow problems had only joint capacity constraints requiring the total flow of all the commodities on each arc to be bounded by the arc's capacity. Arc capacities for individual commodities, e.g., $u_i(v,w) \leq 7$, and for subsets of commodities, e.g., $u_i(v,w) + u_{i+1}(v,w) \leq 9$ can also be incorporated. Cost constraints bounding the total flow cost, a particular commodity's cost, or the cost of some subset of flows can be added. Each additional constraint increases the running time.

Lemma 21 *Consider solving one of the problems in this chapter but having* κ *constraints. If the Grigoriadis-Khachiyan framework is used to solve the problem,* $O((\epsilon^{-2}\log\epsilon^{-1} +$

Algorithm 6 Generalized Shortest Path Algorithm for Directed Acyclic Graphs

Input: directed acyclic graph G and source vertex s
Output: generalized shortest paths from s to all vertices

```
Topologically sort the vertices of G.
for all vertices v do
  π(v) ← ∞
π(s) ← 0
for all vertices v in topologically sorted order do
  for all arcs v → w do
    π(w) ← min(π(w), (π(v) + c(v → w))/μ(v → w))
```

$\log\kappa)\kappa(kmn^2\log m + \log\log(\kappa/\epsilon)))$ *time is required. Using the approximate GSP subroutine, the running time is* $O((\epsilon^{-2}\log\epsilon^{-1} + \log\kappa)\kappa(kmn\log(\epsilon^{-1}(\log(\kappa/\epsilon) + n\log B)) + \log\log(\kappa/\epsilon)))$. *If the Garg-Könemann framework is used,* $O(\epsilon^{-2}k\kappa\log n(mn^2\log m))$ *time is required. A weakly polynomial time bound of* $O(\epsilon^{-2}k\kappa\log n(mn\log(\epsilon^{-1}(\log(m/\epsilon) + n\log B))))$ *is also possible.*

Since each commodity's generalized shortest path calculation is independent of the other commodities' computations, each commodity's flow multipliers can differ. For example, $\mu_i(v,w) = 0.5$ while $\mu_{i+1}(v,w) = 1$. Using differing flow multipliers does not increase the running time.

Faster Shortest Path If a problem's structure permits faster solution of generalized nonnegative-cost shortest path problems, the running times of the approximation algorithms decrease proportionally. For example, many generalized shortest path problems have only lossy or breakeven arcs. For this case, a variant of Dijkstra's algorithm can solve generalized shortest path problems in $O(m + n\log n)$ time [CR66, WF99]. Using this faster subroutine substitutes $m + n\log n$ for the $mn^2\log m$ term in the running times [WF99]. (The careful reader will have noted that we now have generalized flow problems with sources and sinks. The goal is to bring one unit to the sink from any source in the cheapest way possible.)

We can solve generalized shortest path problems in directed acyclic graphs in $O(m+n)$ time and the same space, further reducing the approximation algorithms' running times.

Lemma 22 *The generalized shortest path problem in a directed acyclic graph with arbitrary costs and flow multipliers can be solved in* $O(m + n)$ *time and space. See Algorithm 6 for the algorithm.*

Charnes and Raike [CR66] presented the main idea of the algorithm in 1966 without noting the use of topological sort permits a linear-time algorithm. The main idea is that a vertex's potential $\pi(v)$ is the minimum cost to send one unit of flow from the source to v.

7.5 Conclusions

We presented a strongly polynomial algorithm for the single-source generalized shortest path problem (GSP), which is a generalized version of the shortest path problem. The algorithm combines a left-distributive closed semiring, a dynamic-programming comparison subroutine, and parametric search for the optimal solution. The algorithm is simpler and requires less space than previous polynomial-time algorithms. These algorithms were based on general polynomial-time linear programming algorithms or explicitly solving the dual problem and translating to a primal solution using the algorithm of Adler and Cosares [AC91].

We also demonstrated that combining fractional packing frameworks and generalized shortest path algorithms permits solving other generalized network flow problems: generalized maximum flow, generalized nonnegative-cost minimum-cost flow, generalized concurrent flow, generalized multicommodity maximum flow, and generalized multicommodity nonnegative-cost minimum-cost flow problems. Excepting the maximum flow problem, these are the first polynomial-time algorithms not based on interior-point methods. They also show that generalized concurrent flow and generalized multicommodity maximum flow have strongly polynomial approximation algorithms.

Bibliography

[ABF$^+$84] I. Ali, D. Barnett, K. Farhangian, J. Kennington, B. Patty, B. Shetty, B. McCarl and P. Wong. Multicommodity network problems: Applications and computations. *IIE Transactions*, 16(2):127–134, June 1984.

[AC91] I. Adler and S. Cosares. A strongly polynomial algorithm for a special class of linear programs. *Operations Research*, 39(6):955–960, November–December 1991.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1974.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[ARVK89] Ilan Adler, Mauricio G. C. Resende, Geraldo Veiga, and Narendra Karmarkar. An implementation of Karmarkar's algorithm for linear programming. *Mathematical Programming A*, 44(3):297–335, 1989.

[AS80] Bengt Aspvall and Yossi Shiloach. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. *SIAM Journal on Computing*, 9(4):827–845, November 1980.

[Ass78] A. A. Assad. Multicommodity network flows—A survey. *Networks*, 8(1):37–91, Spring 1978.

[AW66] Mordecai Avriel and Douglass J. Wilde. Optimality proof for the symmetric Fibonacci search technique. *Fibonacci Quarterly*, 4:265–269, 1966.

[Bad91] Tamas Badics. GENRMF. `ftp://dimacs.rutgers.edu/pub/netflow/generators/network/genrmf/`, 1991.

[Bak73] A. Bakó. On the determination of the shortest path in a network having gains. *Mathematische Operationsforschung und Statistik*, 4(1):63–68, January–February 1973.

[Bar88] C. Barnhart. *A Network-Based Primal-Dual Solution Methodology for the Multicommodity Network Flow Problem*. PhD thesis, Department of Civil Engineering, MIT, Cambridge, MA, 1988.

[BBM82] D. Barnett, J. Brinkley, and B. McCarl. The effects of U.S. port capacity constraints on national and world grain shipments. Technical report, Purdue Agricultural Experiment Station, Purdue University, 1982.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[Bix98] Robert E. Bixby. Personal communication, October 1998.

[Cas98] Jordi Castro. A specialized interior point algorithm for multicommodity network flows. Submitted to *SIAM Journal on Optimization*, July 1998.

[Chv83] Vašek Chvátal. *Linear Programming*. W. H. Freeman and Company, New York, 1983.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[CM94a] Edith Cohen and Nimrod Megiddo. Improved algorithms for linear inequalities with two variables per inequality. *SIAM Journal on Computing*, 23(6):1313–1347, December 1994.

[CM94b] Edith Cohen and Nimrod Megiddo. New algorithms for generalized network flows. *Mathematical Programming*, 64(3):325–336, May 1994.

[CN96] J. Castro and N. Nabona. An implementation of linear and nonlinear multicommodity network flows. *European Journal of Operational Research*, 92(1):37–53, July 1996.

[CPL95] CPLEX Optimization, Inc., Incline Village, NV. *Using the CPLEX Callable Library*, 4.0 edition, 1995.

[CR66] A. Charnes and W. M. Raike. One-pass algorithms for some generalized network problems. *Operations Research*, 14:914–924, September–October 1966.

[Dan63] George Bernard Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DW60] George B. Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, January–February 1960.

[FF58] L. R. Ford and D. R. Fulkerson. A suggested computation for maximal multicommodity network flows. *Management Science*, 5:97–101, 1958.

[FF62] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[Flo62] Robert W. Floyd. Algorithm 245 (SHORTEST PATH). *Communications of the Association for Computing Machinery*, 5(6):345, 1962.

[For56] L. R. Ford, Jr. Network flow theory. Technical Report P-923, The Rand Corporation, Santa Monica, CA, August 1956.

[GG88] D. Goldfarb and M. D. Grigoriadis. A computational comparison of the Dinic and network simplex methods for maximum flow. *Annals of Operations Research*, 13(1–4):83–123, 1988.

[GJ95] D. Goldfarb and Z. Jin. A polynomial dual simplex algorithm for the generalized circulation problem. Technical report, Department of Industrial Engineering and Operations Research, Columbia University, 1995.

[GJ96] Donald Goldfarb and Zhiying Jin. A faster combinatorial algorithm for the generalized circulation problem. *Mathematics of Operations Research*, 21(3):529–539, August 1996.

[GJL98] D. Goldfarb, Z. Jin, and Y. Lin. A polynomial dual simplex algorithm for the generalized circulation problem. Technical report, Department of Industrial Engineering and Operations Research, Columbia University, 1998.

[GJO97] Donald Goldfarb, Zhiying Jin, and James B. Orlin. Polynomial-time highest-gain augmenting path algorithms for the generalized circulation problem. *Mathematics of Operations Research*, 22(4):793–802, November 1997.

[GK94] Michael D. Grigoriadis and Leonid G. Khachiyan. Fast approximate schemes for convex programs with many blocks and coupling constraints. *SIAM Journal on Optimization*, 4(1):86–107, February 1994.

[GK96a] Michael D. Grigoriadis and Leonid G. Khachiyan. Approximate minimum-cost multicommodity flows in $\tilde{O}(\epsilon^{-2}knm)$ time. *Mathematical Programming*, 75(3):477–482, 16 Dec 1996.

[GK96b] Michael D. Grigoriadis and Leonid G. Khachiyan. Coordination complexity of parallel price-directive decomposition. *Mathematics of Operations Research*, 21(2):321–340, May 1996.

[GK98] Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *Symposium on Foundations of Computer Science*, volume 39, pages 300–309, Los Alamitos, CA, November 1998. IEEE Computer Society.

[Gol92] A. V. Goldberg. A natural randomization strategy for multicommodity flow and related algorithms. *Information Processing Letters*, 42(5):249–256, 03 July 1992.

[GOPS97] Andrew Goldberg, Jeffrey D. Oldham, Serge Plotkin, and Cliff Stein. An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow. Technical Report CS-TR-97-1600, Stanford University, December 1997. http://theory.stanford.edu/~oldham/publications/MCMCF-TR/MCMCF-TR.ps.

[GOPS98] Andrew V. Goldberg, Jeffrey D. Oldham, Serge Plotkin, and Cliff Stein. An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow. In Robert E. Bixby, E. Andrew Boyd, and Roger Z. Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, volume 1412 of *Lecture Notes in Computer Science*, pages 338–352, Berlin, 1998. Springer. Technical Report available as http://theory.stanford.edu/˜oldham/-publications/MCMCF-TR/MCMCF-TR.ps.

[GPT91] Andrew V. Goldberg, Serge A. Plotkin, and Éva Tardos. Combinatorial algorithms for the generalized circulation problem. *Mathematics of Operations Research*, 16(2):351–381, May 1991.

[Gri86] M.D. Grigoriadis. An efficient implementation of the network simplex method. *Mathematical Programming Study*, 26:83–111, 1986.

[GTT90] Andrew V. Goldberg, Éva Tardos, and Robert E. Tarjan. Network flow algorithms. In Bernhard Korte, László Lovász, Hans Jürgen Prömel, and Alexander Schrijver, editors, *Paths, Flows, and VLSI-Layout*, number 9 in Algorithms and Combinatorics, pages 101–164. Springer-Verlag, Berlin, 1990.

[Has92] Refael Hassin. Approximate schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17(1):36–42, February 1992.

[HL96] Randolph W. Hall and David Lotspeich. Optimized lane assignment on an automated highway. *Transportation Research—C*, 4C(4):211–229, August 1996.

[HN94] Dorit S. Hochbaum and Joseph (Seffi) Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM Journal on Computing*, 23(6):1179–1192, December 1994.

[HO96] Ali Haghani and Sei-Chang Oh. Formulation and solution of a multi-commodity, multi-modal network flow model for disaster relief operations. *Transportation Research—A*, 30A(3):231–250, May 1996.

[Jac45] C. G. J. Jacobi. Über eine neue auflösungsart der bei der methode der kleinsten quadrate vorkommenden linearen gleichungen. *Astronomische Nachrichten*, 22:297–303, 1845.

[Jew62] William S. Jewell. Optimal flow through networks with gains. *Operations Research*, 10(4):476–499, July–August 1962.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[Ken78] Jeff L. Kennington. A survey of linear cost multicommodity network flows. *Operations Research*, 26(2):209–236, March–April 1978.

[KH80] Jeff L. Kennington and Richard V. Helgason. *Algorithms for Network Programming.* John Wiley & Sons, New York, 1980.

[Kha80] L. G. Khachiyan. Polynomial algorithms in linear programming. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki (Journal of Computational Mathematics and Mathematical Physics)*, 20(1):51–68, January–February 1980.

[Knu92] Donald E. Knuth. Two notes on notation. *American Mathematical Monthly*, 99(5):403–422, May 1992.

[KP95a] Anil Kamath and Omri Palmon. Improved interior point algorithms for exact and approximate solution of multicommodity flow problems. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 502–511. Association for Computing Machinery, January 1995.

[KP95b] David Karger and Serge Plotkin. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. In *Symposium on the Theory of Computing*, volume 27, pages 18–25. Association for Computing Machinery, ACM Press, May 1995.

[KPSET94] Philip Klein, Serge Plotkin, Clifford Stein, and Éva 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.

[KV86] Sanjiv Kapoor and Pravin M. Vaidya. Fast algorithms for convex quadratic programming and multicommodity flows. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, volume 18, pages 147–159. Association for Computing Machinery, 1986.

[KV96] Sanjiv Kapoor and Pravin M. Vaidya. Speeding up Karmarkar's algorithm for multicommodity flows. *Mathematical Programming*, 73(1):111–127, 30 April 1996.

[Law76] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Saunders College Publishing, Fort Worth, 1976.

[LMP+95] Tom Leighton, Fillia Makedon, Serge Plotkin, Clifford Stein, Éva Tardos, and Spyros Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences*, 50(2):228–243, April 1995.

[LO91] Y. Lee and J. Orlin. GRIDGEN. `ftp://dimacs.rutgers.edu/pub/netflow/-generators/network/gridgen/`, 1991.

[LSS93] Tishya Leong, Peter Shor, and Clifford Stein. Implementation of a combinatorial multicommodity flow algorithm. In David S. Johnson and Catherine C. McGeoch, editors, *Network Flows and Matching*, volume 12 of *Series in Discrete Mathematics and Theoretical Computer Science*, pages 387–405. American Mathematical Society, 1993.

[Meg79] Nimrod Megiddo. Combinatorial optimization with rational objective functions. *Mathematics of Operations Research*, 4(4):414–424, November 1979.

[Meg83] Nimrod Megiddo. Towards a genuinely polynomial algorithm for linear programming. *SIAM Journal on Computing*, 12(2):347–353, May 1983.

[Mur92] Steven M. Murray. *An Interior Point Approach to the Generalized Flow Problem with Costs and Related Problems.* PhD thesis, Stanford University, August 1992. Department of Operations Research.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, 1988.

[PST95] Serge A. Plotkin, David B. Shmoys, and Éva Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20(2):257–301, May 1995.

[Rad93] Tomasz Radzik. Approximate generalized circulation. Technical Report 93-2, Cornell Computational Optimization Project, Ithaca, NY, January 1993.

[Rad95] Tomasz Radzik. Fast deterministic approximation for the multicommodity flow problem. In *Symposium on Discrete Algorithms*, volume 6, pages 486–492. Association for Computing Machinery and Society for Industrial and Applied Mathematics, January 1995.

[Rad98] Tomasz Radzik. Faster algorithms for the generalized network flow problem. *Mathematics of Operations Research*, 23(1):69–100, February 1998.

[Rap90] Joseph Raphson. *Analysis Æquationum Universalis, seu, Ad Æquationes Algebraicas Resolvendas Methodus Generalis, et Expedita*. Prostant venales apud Abelem Swalle, London, 1690.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, 1986.

[Sch91] Rina Schneur. *Scaling Algorithms for Multicommodity Flow Problems and Network Flow Problems with Side Constraints*. PhD thesis, MIT, Cambridge, MA, February 1991.

[SM90] F. Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37(2):318–334, April 1990.

[Ste92] Clifford Stein. *Approximation Algorithms for Multicommodity Flow and Shop Scheduling Problems*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, September 1992.

[Ste99] Clifford Stein. Personal communication, 12 June 1999.

[Tar86] Éva Tardos. A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research*, 34(2):250–256, March–April 1986.

[TW98] Éva Tardos and Kevin D. Wayne. Simple generalized maximum flow algorithms. In R. E. Bixby, E. A. Boyd, and R. Z. Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, volume 1412 of *Lecture Notes in Computer Science*, pages 310–324, Berlin, June 1998. Springer.

[Vai89a] Pravin M. Vaidya. Speeding up linear programming using fast matrix multiplication. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, volume 30, pages 332–337. IEEE Computer Society Press, 1989.

[Vai89b] Pravin M. Vaidya. Speeding-up linear programming using fast matrix multiplication. In *Thirtieth Annual Symposium on Foundations of Computer Science*, volume 30, pages 332–337, Los Alamitos, CA, 30 October–01 November 1989. IEEE Computer Society Press.

[War62] Stephen Warshall. A theorem on boolean matrices. *Journal of the Association for Computing Machinery*, 9(1):11–12, January 1962.

[Way99] Kevin D. Wayne. A polynomial combinatorial algorithm for generalized minimum cost flow. In *Symposium on the Theory of Computing*, volume 31, pages 11–18. ACM Press, 01–04 May 1999.

[WF99] Kevin D. Wayne and Lisa Fleischer. Faster approximation algorithms for generalized flow. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages S981–982, January 1999. http://www.cs.princeton.edu/˜wayne/packing.ps.

[Zad73a] N. Zadeh. A bad network problem for the simplex method and other minimum cost flow algorithms. *Mathematical Programming*, 5:255–266, 1973.

[Zad73b] N. Zadeh. More pathological examples for network flow problems. *Mathematical Programming*, 5(2):217–224, 1973.

Index