An Implementation of a Combinatorial Approximation Algorithm for Minimum-Cost Multicommodity Flow

Andrew Goldberg[1] Jeffrey D. Oldham[2] Serge Plotkin[3]
Cliff Stein[4]

1997 December 15

[1]NEC Research Institute, Inc., Princeton, NJ 08540, `avg@research.nj.-nec.com`.

[2]Department of Computer Science, Stanford University, Stanford, CA 94305-9045, `oldham@cs.stanford.edu`. Research partially supported by a National Science Foundation Graduate Research Fellowship, ARO Grant DAAH04-95-1-0121, and NSF Grants CCR-9304971 and CCR-9307045.

[3]Department of Computer Science, Stanford University, Stanford, CA 94305-9045, `plotkin@cs.stanford.edu`. Research supported by Research supported by ARO Grant DAAH04-95-1-0121, NSF Grants CCR-9304971 and CCR-9307045, and a Terman Fellowship.

[4]Department of Computer Science, Dartmouth College, Hanover, NH 03755, `cliff@cs.dartmouth.edu`. Research partly supported by NSF Award CCR-9308701 and NSF Career Award CCR-9624828. Some of this work was done while this author was visiting Stanford University.

Abstract

The *minimum-cost multicommodity flow problem* involves simultaneously shipping multiple commodities through a single network so that the total flow obeys arc capacity constraints and has minimum cost.

Multicommodity flow problems can be expressed as linear programs, and most theoretical and practical algorithms use linear-programming algorithms specialized for the problems' structures. Combinatorial approximation algorithms in [GK95, KP95b, PST95] yield flows with costs slightly larger than the minimum cost and use capacities slightly larger than the given capacities. Theoretically, the running times of these algorithms are much less than that of linear-programming-based algorithms.

We combine and modify the theoretical ideas in these approximation algorithms to yield a fast, practical implementation solving the minimum-cost multicommodity flow problem. Experimentally, the algorithm solved our problem instances (to 1% accuracy) two to three orders of magnitude faster than the linear-programming package `CPLEX` [CPL95] and the linear-programming based multicommodity flow program `PPRN` [CN96].

1 Introduction

The *minimum-cost multicommodity 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 minimum-cost multicommodity flow problem [KP95a, KV86, Vai89] exactly 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 and has cost that is within $1+\epsilon$ of the optimum. Algorithms for computing approximate solutions to the multicommodity flow problem were developed in [LMP$^+$95] (no-cost case) and [GK95, KP95b, PST95] (minimum-cost case). Theoretically, these algorithms are much faster than interior-point method based algorithms for constant ϵ. The algorithm in [LMP$^+$95] was implemented [LSS93] and was shown that indeed it often outperforms the 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 fast.

In this paper we describe `MCMCF`, our implementation of a combinatorial approximation algorithm for the minimum-cost multicommodity flow problem. A direct implementation of [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 implementation with `CPLEX` [CPL95] and `PPRN` [CN96]. (Several other efficient minimum-cost multicommodity 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, `MCMCF` was 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 `MCMCF`'s 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, `MCMCF` repeatedly chooses a commodity and then computes 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 algorithm iterates 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 an efficient implementation:

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 σ that maximizes 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 "warm-start" each minimum-cost flow computation. Excepting the warm-start idea, our primal simplex code is similar to that of Grigoriadis [Gri86].

In the rest of this paper, we first introduce the theoretical ideas behind the implementation. After introducing the problem instances we used to test our implementation, we discuss the various choices in translating the theoretical ideas into practical performance. Then, we present experimental data showing that `MCMCF`'s running time's dependence on the accuracy ϵ is smaller than theoretically predicted and its dependence on the number k of commodities is close to what is predicted. We conclude by showing that the combinatorial-based implementation solves our instances two to three orders of magnitude faster than two simplex-based implementations. In the future work, we will also show that a slightly modified `MCMCF` solves the *concurrent flow problem*, i.e., the optimization version of the no-cost multicommodity flow problem, two to twenty times faster than Leong et al.'s approximation implementation [LSS93].

2 Theoretical Background

2.1 Definitions

The *minimum-cost multicommodity flow problem* consists of a directed network $G = (V, A)$, a positive arc capacity function u, a nonnegative arc cost function c, and a specification (s_i, t_i, d_i) for each commodity i, $i \in$

$\{1, 2, \ldots, k_0\}$. Nodes s_i and t_i are the *source* and the *sink* of commodity i, and a positive number d_i is its *demand*.

A *flow* is a nonnegative arc function f. A *flow f_i of commodity i* is a flow obeying conservation constraints and satisfying its demand d_i. 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 $c \cdot f = \sum_{a \in A} c(a) f(a)$.

Given a problem, a flow f, and a budget B, the *congestion* of arc a is $\lambda_a = f(a)/u(a)$, and the *congestion* of the flow is $\lambda_A = \max_a \lambda_a$. The *cost congestion* is $\lambda_c = c \cdot f / B$, and the *total congestion* is $\lambda = \max\{\lambda_A, \lambda_c\}$. A *feasible problem instance* has a flow f with $\lambda_A \leq 1$.

Our implementation approximately solves the minimum-cost multicommodity 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 *$\epsilon$-optimal cost*, i.e., if B^* is the minimum cost of any feasible flow, f's cost is at most $(1+\epsilon)B^*$. Because we can choose ϵ arbitrarily small, we can find a solution arbitrarily close to the optimal.

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

2.2 The Algorithmic Framework

Our algorithm is mostly based on [KP95b]. Roughly speaking, the approach in that paper is as follows. The algorithm 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 $O(\epsilon)$-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 Radzik [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 [GK95] decreased the dependence on ϵ to ϵ^{-2}.

Since these minimum-cost algorithms compute a multiflow having arc cost at most a budget bound B, we use binary search on B to determine an ϵ-optimal cost. The arc cost of the initial flow gives 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. Lower bound computations (see Section 4.1) increase the lower bound and the algorithm decreases the congestion and cost until an ϵ-optimal flow is found.

3 The Problem Instances

To test the `MCMCF` implementation, we modified single-commodity maximum-flow and minimum-cost flow problem generators to produce minimum-cost multicommodity 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 have maximum arc congestions of 0.60.

The problem generator MULTIRMFGEN (abbreviated RMFGEN and based on [Bad91]) produces two-dimensional *frames* (grids) with arcs connecting a random permutation of nodes in adjacent frames. The intraframe arcs have capacities of 6400, 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.

The generator MULTIGRIDGEN (abbreviated MULTIGRID and based on [LO91]) produces two-dimensional grids with a limited number of additional arcs connecting randomly chosen nodes. 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.

We wrote TRIPARTITE to produce difficult-to-solve graphs to test `MCMCF`'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 a node permutation connects adjacent layers. All arcs have random costs in the range $[1, 1000]$. Commodities' sources and sinks are at opposite ends of the graph.

The real-world GTE problem instance has 49 nodes, 260 arcs, and 585 commodities.

All data obtained using a Sun UltraSparc-2 except where otherwise indicated.

4 Translating Theory into Practice

The algorithmic framework described in the previous section is theoretically efficient, but a direct implementation requires orders of magnitude larger 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 4.1. 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 4.3.

4.1 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], 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]. Since the problem instance is assumed to be feasible, the computation indicates when the current guess for the minimum flow cost is too low.

The weak duality inequalities yield a lower bound. Using the notation from [PST95],

$$\lambda \sum_r y_r \geq \sum_{\text{comm. } i} C_i(\lambda_A) \geq \sum_{\text{comm. } i} C_i^*(\lambda_A). \tag{1}$$

For commodity i, $C_i(\lambda_A)$ represents the cost of the current flow f_i with respect to arc capacities $\lambda_A u$ and the cost function $y^t A'$, where A' is a $(m+1)$-by-km matrix. The first m rows implement the arc capacity constraints while the last row implements the arc cost function c. $C_i^*(\lambda_A)$ is the minimum-cost flow. For all choices of dual variables and $\lambda_A \geq 1$, $\lambda^* \geq \sum_i C_i^*(1)/\sum_r y_r \geq \sum_i C_i^*(\lambda_A)/\sum_r y_r$. Thus, this ratio serves as a lower bound on the optimal congestion λ^*.

4.2 Computing the Step Size σ

While, as suggested by the theory, using a fixed step size σ 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. Brent's method and similar strategies, e.g., see [LSS93], are natural

problem	ϵ	time (seconds) Newton	fixed	ratio	number of MCFs Newton	fixed	ratio
rmfgen1	0.12	0.7	7.9	11.5	238	4789	20.1
rmfgen1	0.06	0.9	45.4	52.2	294	28534	97.1
rmfgen1	0.03	2.5	180.6	70.8	878	114390	130.3
rmfgen1	0.01	11.2	1334.2	119.4	4102	879963	214.5
rmfgen2	0.01	63.9	3842.7	60.1	13261	1361037	102.6
rmfgen3	0.01	257.5	15202.9	59.0	17781	1683061	94.7
multigrid1	0.01	3.0	95.3	31.6	1375	77936	56.7

Table 1: Computing an (almost) optimal step size reduces the running time and number of minimum-cost flow (MCF) computations by two orders of magnitude. Data obtained on a Sun Enterprise 3000.

strategies to maximize the function's reduction. We implemented Brent's method [PFTV88], but the special structure of the potential function allows us to compute the function's first and second derivatives. Thus, we can use the Newton-Raphson method [PFTV88, Rap90], which is faster.

Given the current flow f and the minimum-cost flow f_i^* for commodity i, the potential function $\phi(\sigma)$ is a convex function (with positive second derivative) of the step size σ. Over the range of possible choices of $\sigma \in [\sigma_{min}, 1]$, the potential function's minimum occurs either at the endpoints or at one interior point. Since the function is a sum of exponentials, the first and second derivatives $\phi'(\sigma)$ and $\phi''(\sigma)$ are easy to compute.

Using the Newton-Raphson method reduces the running time by two orders of magnitude compared with using a fixed step size. (See Table 1.) 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 was even greater.

4.3 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. Thus, we developed heuristics for slowly increasing its value. There are two different theoretical explanations for α than can be used to develop two different heuristics.

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.

problem	ϵ	number of MCFs adaptive	fixed	ratio	time (seconds) adaptive	fixed	ratio
GTE	0.01	2559	15804	6.17	3.63	22.50	6.20
rmfgen3	0.01	3659	9999	2.73	55.71	160.75	2.89
rmfgen4	0.10	7998	4575	1.75	114.05	59.24	1.93
rmfgen4	0.05	6884	25144	3.65	91.94	343.30	3.73
rmfgen4	0.03	18483	56511	3.06	238.04	738.48	3.10
rmfgen5	0.03	17125	65069	3.80	353.75	1341.10	3.79
multigrid2	0.01	1659	1266	0.76	41.61	47.22	1.13

Table 2: Adaptively choosing α requires fewer minimum-cost flow (MCF) computations than using the theoretical, fixed value of α. Data obtained on a Sun Enterprise 3000.

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.

Plotkin, Shmoys, and Tardos [PST95] use the weak duality inequalities (Eq. 1 of Section 4.1) upon which we base a different heuristic. The product of the gaps bounds the distance between the potential function and optimal flows. The algorithm's improvement is proportional to the size of the right gap, and 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 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 α values are frequently much lower than those from [PST95]. Using this heuristic rather than using the theoretical value of $\ln(3m)/\epsilon$ [KP95b] usually decreases the running time by a factor of between two and six. See Table 2.

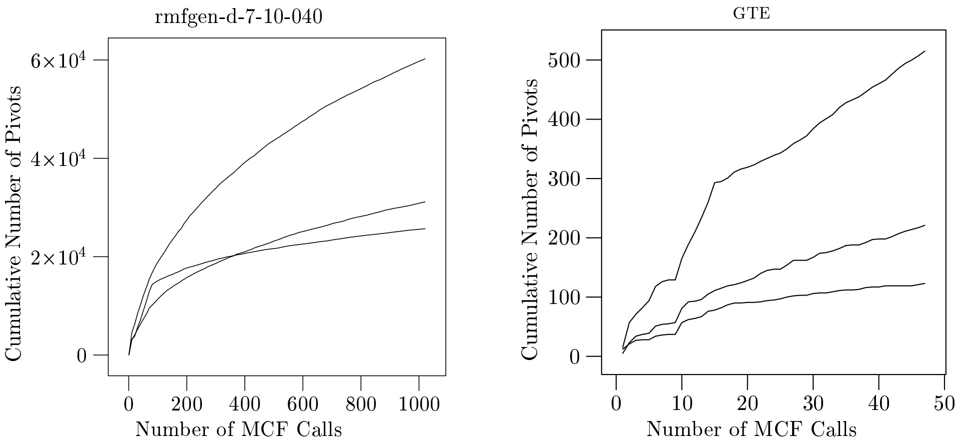

Figure 1: The cumulative number of pivots as a function of the number of MCF calls for three different commodities in two problem instances.

4.4 "Restarting" The Minimum-Cost Flow Subroutine

Theoretically, `MCMCF` 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. Each pivot maintains the flow's feasibility and can decrease its cost. The simplex algorithm can start with any feasible flow and any spanning tree. 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. Using the previously-found minimum-cost flow requires $O(km)$ additional storage. Moreover, it is frequently unusable because it is infeasible with respect to the capacity constraints than using the current flow. In contrast, using the current flow requires no additional storage, this flow is known to be feasible, and starting from this flow experimentally requires a very small number of pivots.

Fairly quickly, the number of pivots per MCF iteration becomes very small. See Figure 1. For the 2121-arc rmfgen-d-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,

problem instance	ϵ	restarting time (sec)	no restarting time (sec)	ratio
rmfgen-d-7-10-020	0.01	464	744	1.60
rmfgen-d-7-10-240	0.01	2130	3152	1.47
rmfgen-d-7-12-240	0.01	2990	4839	1.61
rmfgen-d-7-14-020	0.01	1060	1564	1.47
rmfgen-d-7-14-040	0.01	1694	2521	1.48
rmfgen-d-7-14-080	0.01	3823	5145	1.34
rmfgen-d-7-14-160	0.01	3958	5319	1.34
rmfgen-d-7-14-240	0.01	4496	6172	1.37
rmfgen-d-7-14-320	0.01	4514	6491	1.43
rmfgen-d-7-16-240	0.01	6779	9821	1.44
multigrid-032-032-128-0080	0.01	89	148	1.67
multigrid-064-064-128-0160	0.01	906	2108	2.32

Table 3: Restarting the minimum-cost flow computations from the current flow and the previous spanning tree reduces the running time by at least 25%. Data obtained on a Pentium Pro.

the average numbers are 8, 3, and 1, i.e., at most three percent of the arcs.

Instead of using a commodity's current flow and its previous spanning tree, a minimum-cost flow computation could start from an arbitrary spanning tree and flow. On the problem instances we tried, warm-starting reduces the running time by a factor of about 1.3 to 2. See Table 3. Because optimal flows are not unique, the number of MCF computations differ, but the difference of usually less than five percent.

5 Experimental Results

5.1 Dependence on the Approximation Factor ϵ

The approximation algorithm `MCMCF` yields an ϵ-optimal flow. Plotkin, Shmoys, and Tardos [PST95] solve the minimum-cost multicommodity flow problem using shortest-paths as a basic subroutine. Karger and Plotkin [KP95b] decreased the running time by 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}. Grigoriadis and Khachiyan [GK95] improved the [KP95b] technique, reducing the ϵ-dependence back to ϵ^{-2}. `MCMCF` implements the

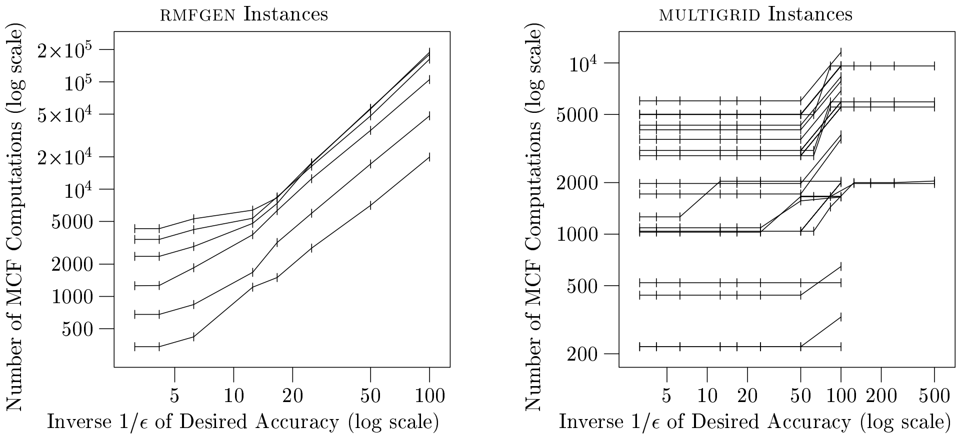

Figure 2: The number of minimum-cost flow computations as a function of $1/\epsilon$ for RMFGEN instances is $O(\epsilon^{-1.5})$ and for MULTIGRID instances.

linear-cost term, but experimentation showed the minimum-cost flows' arc costs were bounded even without using the linear-cost term. Furthermore, the running time usually decreases when omitting the term.

The implementation exhibits smaller dependence than the worst-case no-cost multicommodity flow dependence of $O(\epsilon^{-2})$. We believe the implementation's searching for an almost-optimal step size and its regularly computing lower bounds decreases the dependence. Figure 2 shows the number of minimum-cost flow computations as a function of the desired accuracy ϵ. Each line represents a problem instance solved with various accuracies. On the log-log scale, a line's slope represents the power of $1/\epsilon$. For the RMFGEN problem instances, the dependence is about $O(\epsilon^{-1.5})$. For most MULTIGRID instances, we solved to a maximum accuracy of 1% but for five instances, we solved to an accuracy of 0.2%. These instances depend very little on the accuracy; `MCMCF` 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. `MCMCF` is able to take advantage of these many different routes, while, as we will see in Section 6, some linear-programming based implementations have more difficulty.

5.2 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. Theo-

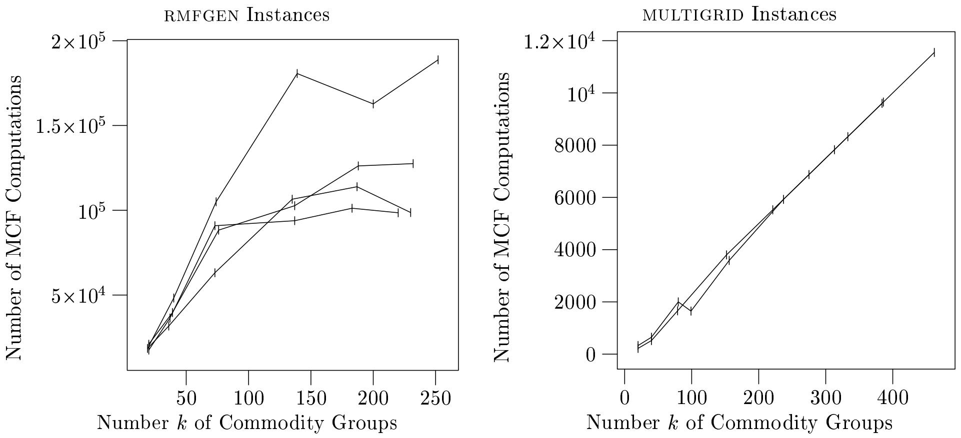

Figure 3: The number of minimum-cost flow computations as a function of the number k of commodity groups for RMFGEN and MULTIGRID instances.

retically, the algorithm performs $\tilde{O}(\epsilon^{-3}k)$ minimum-cost flow computations, as described in Section 2.2. These upper bounds (ignoring the ϵ dependence and logarithmic dependences) match the natural lower bound 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.

Figure 3 shows the number of minimum-cost flow computations as a function of the number k of commodity groups. Each line represents a fixed network with various numbers of commodity groups. The MULTIGRID figure shows a dependence of approximately $25k$ for two 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.

6 Comparisons with Other Implementations

6.1 The Other Implementations: CPLEX and PPRN

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

CPLEX (version 4.0.9) yields exact solutions to multicommodity flow linear programs. When forming the linear programs, we group the commodities since MCMCF computes these groups at run-time. CPLEX's dual simplex method yields a feasible solution only upon completion, while the primal method, in principle, could 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 the 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.

6.2 Dependence on the Number k of Commodity Groups

The combinatorial algorithm MCMCF solves our problem instances 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 Figure 4) 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 the left half of Figure 4, 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 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 the right half of Figure 4.)

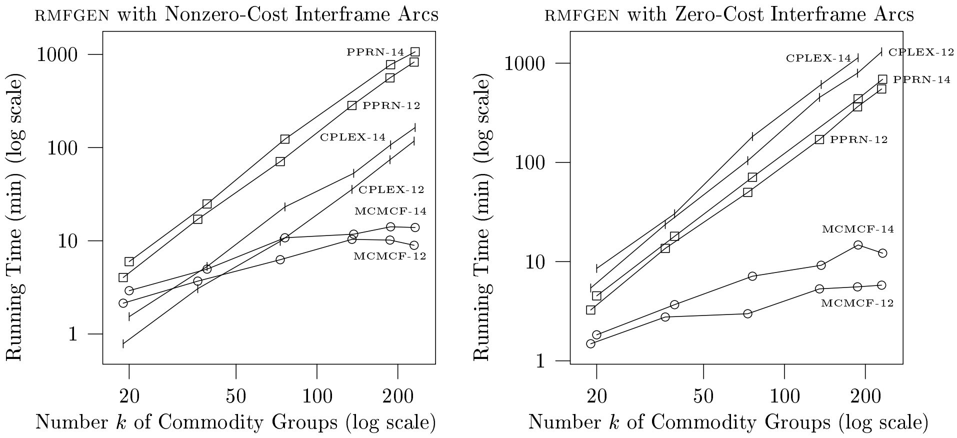

Figure 4: The running time in minutes as a function of the number k of commodity groups for two different RMFGEN networks with twelve and fourteen frames. `CPLEX`'s and `PPRN`'s dependences are larger than `MCMCF`'s.

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 shows `MCMCF`'s running time using a log-log scale for two different networks: the smaller one having 1025 nodes and 3072 arcs and the larger one having 4097 nodes 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`.

6.3 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 nodes 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 6. (`MCMCF` solved the problem instances to two-percent accuracy.) As described in Section 4.4, the minimum-cost flow routine needs only a few pivots before a solution is found. `CPLEX`'s and `PPRN`'s dependences are much higher. (For the sixty-four frame problem, `PPRN` required 2890 minutes so it

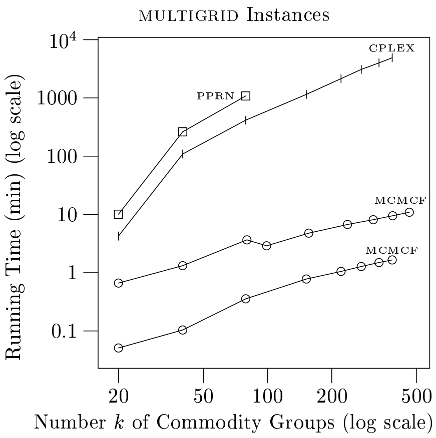

Figure 5. The running time in minutes as a function of the number k of commodity groups for MULTIGRID problem instances.

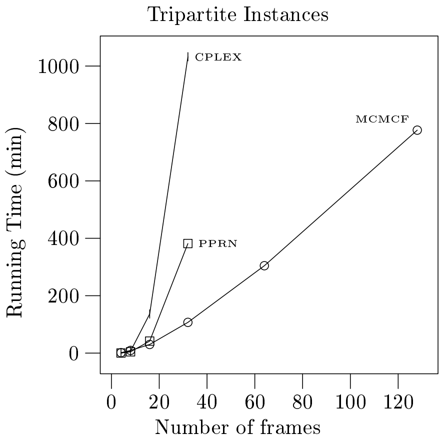

Figure 6. The running time in minutes as a function of the number of frames for TRIPARTITE problem instances.

was omitted from the figure.)

7 Concluding Remarks

For the problem classes we studied, `MCMCF` solved minimum-cost multicommodity 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].

We conclude with two unanswered questions. Since our implementation never needs to use the linear-cost term [KP95b], it is interesting to prove whether the term is indeed unnecessary. Also, it is interesting to try to prove the experimental $O(\epsilon^{-1.5})$ dependence of Section 5.1.

References

[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.

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

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

[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. Using the `CPLEX` callable library. Software Manual, 1995.

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

[GK95] Michael D. Grigoriadis and Leonid G. Khachiyan. Approximate minimum-cost multicommodity flows in $\tilde{O}(\epsilon^{-2}knm)$ time. Technical Report 95-13, Rutgers University, May 1995.

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

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

[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.

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

[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.

[KP95a] Anil Kamath and Omri Palmon. Improved interior point algorithms for exact and approximate solution of multicommodity flow problems. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, volume 6, 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.

[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.

[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.

[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.

[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.

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

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

[Vai89] 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.