DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

Implementation of a Combinatorial Multicommodity Flow Algorithm

TISHYA LEONG AND PETER SHOR AND CLIFFORD STEIN

December 11, 1992

ABSTRACT. The multicommodity flow problem involves simultaneously shipping multiple commodities through a single network so that the total amount of flow on each edge is no more than the capacity of the edge. This problem can be expressed as a large linear program, and most known algorithms for it, both theoretical and practical, are linear programming algorithms designed to take advantage of the structure of multicommodity flow problems. The size of the linear programs, however, makes it prohibitively difficult to solve large multicommodity flow problems.

In this paper, we describe and examine a multicommodity flow implementation based on the recent combinatorial approximation algorithm of Leighton et al. [**13**]. The theory predicts that the running time of the algorithm increases linearly with the number of commodities. Our experiments verify this behavior. The theory also predicts that the running time increases as the square of the desired precision. Our experiments show that the running time increases at most this fast, and often slower. We also compare our combinatorial implementation against two different linear programming-based codes. First we compare our code to that of of Kennington [**10**], which is a network simplex code known to perform well on multicommodity flow problems. For many problems, our combinatorial algorithm outperforms this simplex-based linear programming algorithm. More precisely, as the number of commodities increases, the running time of our algorithm grows much more slowly than that of Kennington's linear programming-based algorithm. Second, we compared our code to an interior point code of Karmarkar and Ramakrishnan. Here too, we achieved similar, but less dramatic results. Our results suggest that our algorithm may be able to solve larger multicommodity flow problems than have been solved in the past.

1991 *Mathematics Subject Classification.* Primary 90B10; Secondary 68Q25, 68A10.

Research of the first author was done at AT&T Bell Laboratories.

Support for the third author provided by NSF PYI Award CCR-89-96272 with matching support from UPS and Sun and by an AT&T Bell Laboratories Graduate Fellowship. Most of this work was done while at AT&T Bell Laboratories.

1. Introduction

The multicommodity flow problem involves simultaneously shipping multiple commodities through a single network so that the total amount of flow on each edge is no more than the capacity of the edge. This problem can be expressed as a large linear program, and most known algorithms for it, both theoretical and practical, are linear programming algorithms designed to take advantage of the structure of multicommodity flow problems. The size of the linear programs, however, makes it prohibitively difficult to solve large multicommodity flow problems.

Recently, Leighton, Makedon, Plotkin, Stein, Tardos, and Tragoudas [**13**] proposed a combinatorial approximation algorithm for the multicommodity flow problem. This algorithm has a faster theoretical running time than the best theoretical linear programming algorithms [**7, 18**]. Also, the running time of this algorithm is dominated by the computation of minimum-cost flows, which are often efficiently computable in practice. These two facts caused Leighton et al. to conjecture that the algorithm would perform well in practice. In this paper, we describe an implementation based on the algorithm of Leighton et al. and investigate its behavior. We show that the algorithm performs at least as well as the theory predicts. We also compare our combinatorial implementation against two different linear programming-based codes. First we compare our code to that of Kennington [**10**], which is a network simplex code known to perform well on multicommodity flow problems. For many problems, our combinatorial algorithm outperforms this simplex-based linear programming algorithm. More precisely, as the number of commodities increases, the running time of our algorithm grows much more slowly than that of Kennington's linear programming-based algorithm. Second, we compared our code to a state-of-the-art interior point code of Karmarkar and Ramakrishnan, called ADP. While the difference with ADP was not as dramatic as with Kennington, we still consistently outperformed this code. The number of tests performed, however, was too small to quantify this comparison any further.

1.1. Background. The input to a multicommodity flow problem consists of an n node, m edge graph and k commodities, each with a source, a sink, and a demand. The corresponding linear program has $O(mk)$ variables and $O(nk+m)$ constraints. Even for a graph with average vertex degree Δ, there are $O(\Delta nk + mk) = O(mk)$ non-zero entries in the constraint matrix. The large size of the linear programs makes the general simplex algorithm impractical for all but very small problems. Some algorithms which take advantage of the special structure of multicommodity flow problems have been proposed. These algorithms fall into three main classes: price-directive decomposition, resource-directive decomposition, and partitioning approaches. (See the surveys of Assad [**2**] and Kennington [**8**] and the thesis of Schneur [**16**] for more information on these approaches.) More recent approaches include interior-point methods [**1**] and a combinatorial scaling algorithm [**16**]. All of the aforementioned algorithms solve multicommodity flow problems using one of two different objective functions. Some find a minimum-cost multicommodity flow, while others find a flow which maximizes the total amount of flow in the network.

In 1986, Shahrokhi and Matula proposed a different objective function [**17**]. They defined the *concurrent flow problem*, which involves finding the maximum z such that there exists a flow which satisfies a percentage z of every demand without exceeding the capacity of any edge. This problem can be formulated as a linear program of the same size as that for the multicommodity flow problem, and it is strictly more general than the multicommodity flow problem. The concurrent flow problem is equivalent to the problem of finding the minimum $\lambda = 1/z$ such that there exists a flow which satisfies all demands while using no more than λ times the capacity of each edge.

Shahrokhi and Matula gave a *fully polynomial approximation scheme* for this problem in the special case in which all the capacities and demands are 1. For a given flow f, let λ_f be its *congestion*, i.e., the maximum over all edges of the ratio between the flow on an edge and the capacity of that edge. Their basic approach is first to route flow on an arbitrary path, ignoring capacities, and then gradually to reroute small amounts of flow from highly congested edges onto lightly congested edges. In $O(\epsilon^{-5}nm^7)$ time, the algorithm finds an ϵ-*optimal* flow, so named because its congestion lies within a $(1+\epsilon)$-factor of the minimum possible congestion. Shahrokhi and Matula implemented their algorithm and tested it on small examples. For the same problem, Klein, Plotkin, Stein, and Tardos later proposed a faster algorithm with an expected running time of $O(\epsilon^{-3}\min\{n,k\}(m+n\log n))$ [**12**].

Building on this framework, Leighton et al. proposed an algorithm for solving the general concurrent flow problem with arbitrary capacities and demands [**13**]. They also start with an arbitrarily routed flow and gradually improve it by rerouting individual commodities to move flow from highly congested edges to lightly congested edges. To reroute flow, they compute minimum-cost flows in suitably defined auxiliary graphs. They show that the algorithm performs approximately $O(\epsilon^{-2}k)$ minimum-cost flow computations in finding an ϵ-optimal multicommodity flow. The algorithm runs in expected $O(\epsilon^{-3}nmk\log^4 n)$ time. Based on the small number of iterations and the fact that minimum-cost flows can often be solved efficiently in practice, they conjectured that their algorithm might work well in practice. In this paper, we provide support for this conjecture.

2. The Underlying Theory

In this section, we define the concurrent flow problem, relate it to the multicommodity flow feasibility problem, and summarize the algorithm of Leighton et al. from which we derive our algorithm.

Consider an undirected graph $G = (V, E)$ with a positive capacity $u(vw)$ for each edge $vw \in E$. Consider also a set of commodities numbered 1 through k, where each commodity i is specified by a source-sink pair $s_i, t_i \in V$ and a positive demand d_i. For each commodity i, we ship an amount proportional to its demand d_i from its source s_i to its sink t_i. This gives us a *single commodity flow* f_i specified by a set of edge flows $f_i(vw)$ on the edges $vw \in E$, where each edge has an arbitrary direction to keep track of which way the flows travel across it. A positive edge flow $f_i(vw) > 0$ denotes a forward flow of commodity i with respect to the direction of edge vw, while a negative flow $f_i(vw) < 0$ denotes a backwards flow. A *multicommodity flow* f consists of k single commodity flows, one for each commodity. In a multicommodity flow f, the total flow $f(vw)$ on each edge $vw \in E$ equals the sum $\sum_{i=1}^{k} |f_i(vw)|$ of the single commodity flows on that edge.

A multicommodity flow achieves *demand satisfaction* if it ships an amount of each commodity equal to its demand from its source to its sink. It obeys the *capacity constraints* if no flow $f(vw)$ on an edge $vw \in E$ exceeds the capacity $u(vw)$ of the edge. A *feasible* multicommodity flow achieves demand satisfaction while obeying the capacity constraints. The *multicommodity flow feasibility problem* is to determine if a feasible flow exists.

Our algorithm solves a more general problem, the *concurrent flow problem*. Given any multicommodity flow f (which need not obey the capacity constraints), each edge $vw \in E$ has a congestion $\lambda_f(vw)$ equal to the ratio $f(vw)/u(vw)$ of total flow to capacity. The congestion λ_f of the flow is the maximum of these edge congestions. It represents an amount by which we can scale the capacities while achieving demand satisfaction, setting the adjusted capacity $u'(vw)$ of each edge $vw \in E$ equal to $\lambda_f \cdot u(vw)$. The concurrent flow problem is to find the lowest possible congestion, which we call the optimal congestion λ^*.

In solving the concurrent flow problem, we also solve the multicommodity flow feasibility problem. If the optimal congestion is greater than 1, the capacities must be raised to achieve demand satisfaction, and so a feasible flow does not exist. If the optimal congestion is less than or equal to 1, all demands can be met given the original capacities, and a feasible flow does exist. Alternatively, we can solve a concurrent flow problem by solving a logarithmic number of multicommodity flow problems. We perform binary search on the congestions of the multicommodity flows and determine the cutoff between feasible and infeasible flows. The congestion at this cutoff is the solution λ^*.

We now summarize the algorithm of Leighton et al. for approximately solving the concurrent flow problem. Given an error parameter $\epsilon > 0$, the algorithm finds an ϵ-optimal flow, i.e., a flow for which $\lambda_f \leq (1+\epsilon)\lambda^*$. Because we can make ϵ arbitrarily small, we can find a solution arbitrarily close to optimal.

The algorithm begins with a flow f that achieves demand satisfaction but ignores the capacity constraints. Leighton et al. show that if the flow is not ϵ-optimal, i.e., if $\lambda_f > (1+\epsilon)\lambda^*$, then there exists at least one "poorly routed" commodity. They then show that by rerouting a fraction of the flow of a poorly routed commodity onto the edges of a minimum-cost flow for that commodity in an appropriately derived auxiliary graph, they cause a decrease in a potential function Φ, which we will define later. Finally, they show that as the potential function decreases, the congestion gradually decreases. The algorithm iteratively reroutes flow, decreasing the potential function and the congestion until the congestion is within a $(1+\epsilon)$-factor of optimal.

The basic idea behind rerouting is to move flow off of highly congested edges. The algorithm achieves this by assigning long lengths to highly congested edges and short lengths to lightly congested edges. Given these lengths, which correspond to the linear programming dual variables, a flow which uses edges of long length marks a poorly routed commodity. Using the lengths as costs, the algorithm finds a minimum-cost flow onto which it reroutes a fraction of the flow of a poorly routed commodity. Leighton et al. assign lengths according to a length function in which $\ell(vw) = e^{\alpha\lambda_f(vw)}/u(vw)$, where α is a constant and $\lambda_f(vw)$ is the congestion on edge vw given the current multicommodity flow f. This length function, being exponential in $\lambda_f(vw)$, clearly penalizes highly congested edges. The minimum-cost flow therefore favors lightly congested edges and utilizes them to the extent allowed by their adjusted capacities. To prevent large increases in the congestion, the capacities have been scaled by the congestion, the capacity of each edge vw in the auxiliary graph being set at $\lambda_f \cdot u(vw)$. After finding a minimum-cost flow, the algorithm reroutes a fraction σ of the current multicommodity flow onto the edges of the minimum-cost flow to create a new multicommodity flow. Choosing the potential function $\Phi = \sum_{vw\in E} u(vw)\ell(vw)$, they can show that the number of iterations of the algorithm is not too large.

In the algorithm as formulated by Leighton et al., the constant α plays a pivotal role. In order to enforce two relaxed optimality conditions which ensure the algorithm's eventual success, Leighton et al. use the values $\alpha = 2(1+\epsilon_0)\lambda_f^{-1}\epsilon_0^{-1}\ln(m\epsilon_0^{-1})$ and $\sigma = \epsilon_0/(8\alpha\lambda_f)$, where $\epsilon_0 \geq \epsilon$ is an error parameter which gradually approaches ϵ and λ_f is the congestion of the current multicommodity flow. Because the fraction σ of flow rerouted depends inversely on α, a smaller α means more flow is rerouted, resulting in a faster decrease of the congestion. However, the value of α also limits how close the algorithm can come to finding the optimal solution. The algorithm is guaranteed to find an ϵ-optimal solution only when α is sufficiently large. As the algorithm progresses, ϵ_0 and λ_f decrease, making α increase. Progress slows as smaller fractions of flow are rerouted, but these choices for α and σ guarantee a solution within a $(1+\epsilon)$-factor of optimal.

We expect the running time of the algorithm to increase as the error parameter ϵ decreases and as the number of commodities k increases. In fact, Leighton et al. prove that given any $\epsilon > 0$, the randomized version of their algorithm finds an ϵ-optimal solution using an expected $O(k(\log k + \epsilon^{-3})\log n)$ minimum-cost flow computations, while the deterministic version uses $O(k^2(\log k + \epsilon^{-2})\log n)$ minimum-cost flow computations. Goldberg [**5**] and Grigoriadis and Khachiyan

[**6**] have shown how to reduce the number of computations used by the randomized version to $O(k(\log k + \epsilon^{-2}) \log n)$. The running time therefore depends polynomially on ϵ^{-1} and linearly on the number of commodities. In the following sections, we describe an implementation based on the algorithm of Leighton et al., and we compare the running times of our implementation to these theoretical bounds.

3. Our Implementation

We now describe how we have adapted and implemented the algorithm of Leighton et al. Where they have made certain choices in the interest of proving the theoretical bounds, we modify the algorithm for the purpose of improving actual performance. We describe the changes we have made and the motivations behind them. We also point out areas in which our modifications could be fine-tuned with further research.

3.1. Grouping Commodities. First, we group the commodities as suggested by Leighton et al. We place all the commodities with the same source into one commodity group and run the algorithm on the commodity groups instead of on the individual commodities. Under this strategy, the number of commodity groups k' cannot exceed the number of nodes n, and rerouting one commodity group corresponds to rerouting all the commodities in the group, an operation made possible by a minimum-cost flow routine that can handle multiple sinks. The running time, which varies linearly with k, now depends on the number of commodity groups rather than the number of commodities. For problems with large numbers of commodities, this means a significant reduction in running time. Because our algorithm uses $O(km)$ space, commodity grouping also reduces the space requirement, making it possible to run larger problems. The advantages gained by grouping commodities have also been documented by Schneur [**16**].

3.2. Choosing a Commodity to Reroute. Leighton et al. propose both a deterministic strategy and a randomized strategy for choosing a commodity group (or a commodity) to reroute. Let $f_{i'}$ be the current flow of commodity group i' and $f^*_{i'}$ be the minimum cost flow of commodity group i' in an auxiliary graph in which each edge vw has a capacity $u'(vw) = \lambda_f \cdot u(vw)$ and a cost of $\ell(vw)$. Then, the deterministic method computes the cost $C_{i'} = \sum_{vw \in E} |f_{i'}(vw)| \ell(vw)$ of a commodity group i', its minimum cost $C^*_{i'} = \sum_{vw \in E} |f^*_{i'}(vw)| \ell(vw)$, and the difference $C_{i'} - C^*_{i'}$ between its cost and minimum cost. The commodity group to be rerouted is the first in a predetermined ordering which has a difference $C_{i'} - C^*_{i'}$ greater than $\epsilon_0 C_{i'} + (\epsilon_0 \lambda_f \Phi)/k'$. This method requires k' minimum-cost flow computations per iteration in the worst case. The randomized strategy computes the cost $C_{i'}$ of each commodity group i' and randomly chooses a commodity group with probability proportional to its cost. This method uses an expected ϵ_0^{-1} minimum-cost flow computations per iteration. Once every k' iterations, minimum-cost flows are computed for all the commodity groups, and the congestion λ_f is checked against the lower bound $\sum_{i'=1}^{k'} C_{i'}^*(\lambda_f)/\Phi$ to decide if the algorithm should terminate. This check increases the number of minimum-cost flow computations by at most a factor of 2. Our selection strategy draws from both the deterministic and the randomized methods of Leighton et al. and from the termination check.

To make the most progress per iteration, we attempt to find not only a poorly routed commodity group but the most poorly routed commodity group. We may designate as the most poorly routed commodity group either the group with the highest cost $C_{i'}$ or the group with the largest difference $C_{i'} - C_{i'}^*$ between cost and minimum cost. Using either measure and rerouting larger fractions of flow than the σ of Leighton et al., we have found that an algorithm which deterministically reroutes the most poorly routed commodity group sometimes gets stuck rerouting a single group over and over with no improvement of the congestion. We have also found that when it does not get stuck, such a deterministic algorithm usually progresses faster than a randomized algorithm. We therefore use a partly deterministic, partly randomized selection strategy in which we alternate between $k'/2$ iterations of deterministic selection and $k'/2$ iterations of random selection. By taking advantage of the minimum-cost flow computations performed in the termination check every k' iterations, we can select commodity groups to reroute without computing extra minimum-cost flows. We reroute, in decreasing order, the $k'/2$ groups with the greatest difference between cost and minimum-cost followed by $k'/2$ randomly chosen commodity groups. To prevent domination by a limited number of groups, the random selection weights all commodity groups equally as proposed by Goldberg [**5**] and Grigoriadis and Khachiyan [**6**].

3.3. Handling the Minimum-cost Flow. Once the algorithm has chosen a commodity group to reroute, it must find an appropriate minimum-cost flow. For this purpose, we use the RELAXT-III minimum-cost flow code of Bertsekas and Tseng [**3**]. One drawback of the routine we have chosen is that it requires integer capacities, costs, and demands, making preprocessing and postprocessing necessary each time it is called. Another routine might better suit our algorithm, but we concentrate on the number of iterations of our algorithm and treat the minimum-cost flow routine as a black box.

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

3.4. Choosing Constants. As noted earlier, the constant α and the fraction σ of flow rerouted greatly affect the running times of the algorithm. Leighton et al. use very large values for alpha and very small values for σ. Their α can easily exceed 1000, and their σ can easily fall below 10^{-5}. For the algorithm to progress at a reasonable rate in practice, we need smaller values for α and larger values for σ. However, when these values begin to block progress, we must raise α and lower σ. We do this by means of a scaling factor s. We set α equal to $c \cdot s/\lambda_f$, where c is the constant $19.1 - \log m$. To decide how much flow to reroute, we sample the values that the potential function Φ would take after the rerouting of various fractions of flow. We reroute the fraction σ_f which gives the lowest Φ. We can find σ_f efficiently because Φ has a positive second derivative with respect to σ, allowing a binary-type search. We sample fractions to the precision $.001/s^2$, and we also use this value as a floor $\sigma_{\min}$ on the fraction of flow that can be rerouted. To avoid wasting time rerouting small amounts of flow, we reroute a commodity only if σ_f is at least as large as $\sigma_{\min}$. We know from **[13]** that we may have to reroute fractions as small as $O(\epsilon/\alpha\lambda_f)$, and so we must decrease $\sigma_{\min}$ faster than we increase α to lower the minimum value for σ_f. We begin with s equal to .25 and raise it by .25 whenever the maximum fraction rerouted in k' iterations is less than $\sigma_{\min}/(s \cdot k')$ or whenever the ratio of the congestion λ_f to its lower bound $\sum_{i'=1}^{k'} C^*_{i'}(\lambda_f)/\Phi$ increases after k' iterations. We have found that this strategy works well in most instances but scales α too fast in a few instances, slowing the algorithm too much for practical use. In such cases, we rerun the algorithm, scaling α more slowly. Our current heuristic for doing so is to check whether the congestion does not decrease for 2000 consecutive iterations. If this is the case, we consider the algorithm to be "stuck" and decrease the parameter s by a factor of 2. This causes α to grow more slowly. We have not yet discovered the optimal rate at which we should scale α, nor have we discovered exactly when we should scale it. This is the area in which our algorithm would benefit most from further research. Other areas in which it could be further improved include the selection strategy for commodities to reroute and the technique for choosing $\sigma_{\min}$.

4. Experimental Results

We have tested our algorithm on a variety of problems and compared its performance to the theoretical bounds. We used two different random network generators, NETGEN and RMFGEN. When run on random NETGEN and RMFGEN graphs with randomly placed commodities, our algorithm behaved more or less as expected. It took polynomially more time to get closer to the optimal solution and less than linearly more time to handle larger numbers of commodities. Furthermore, for large numbers of commodities, our algorithm outperformed the linear programming-based code of Kennington. It also consistently outperformed the interior point linear programming based codes of Karmarkar and Ramakrishnan. Our algorithm performed poorly on one real problem provided by the GTE Corporation, but we consider this an anomaly arising from a limited number of unusually time-consuming minimum-cost flow computations. This one instance aside, we find our results encouraging and consider it an improvement, in many cases, over the simplex-based algorithms which have preceded it.

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

We ran our algorithm on various problems and graphed the lowest ϵ achieved against the number of iterations completed. Each run stopped at a final ϵ of .001 or less. To compress the data, we used data points representing ranges of iterations. For each problem, we considered 10 runs and, for each run, the minimum ϵ achieved at each termination check. The aggregate ϵ for a range equaled the average of the minimum ϵ values found at the termination checks falling in the range during each of the 10 runs. We examined a problem with 20 commodities and four problems with 10 commodities using different NETGEN graphs with 50 nodes and 100 edges. We also examined two problems with 10 and 20 commodities, respectively, using an RMFGEN graph with 140 edges and 48 nodes (spread evenly over 12 square planes). While we only ran this particular instance once, due to the large number of iterations, we expect that the variance should be reduced. To test a large problem, we examined a single run on a large RMFGEN problem with 700 commodities, 2075 edges, and 500 nodes (spread over 20 square planes). For all of these problems, we graphed ϵ versus the number of iterations. We also graphed the function $1/\sqrt{\#\text{ of iterations}}$ on which we expected ϵ to depend. As is evident from Figures 3 through 9, our implementation always performed better than the expected bounds. We note that the fact that, in Figure 8, ϵ crosses the function $1/\sqrt{\#\text{ of iterations}}$ is not a contradiction of the theoretical predictions. The predictions are for $O(1/\sqrt{\#\text{ of iterations}})$, and with an appropriately chosen constant the lines would not cross. Some inconclusive attempts at fitting the data to a curve of the form $a * (\#\text{ of iterations})^b + c$ using a regression package lends some additional support to this conclusion as typical values of b were between $-.5$ and -1.

4.2. Dependence on the Number of Commodities. With respect to the number of commodities k, our algorithm also seems to conform to the theoretical bounds. Using 10 runs for each data point and disregarding the shortest and the longest of these runs, we graphed the average number of iterations needed to solve problems with variable numbers of commodities given a fixed graph. In Figure 10, we examined four NETGEN graphs with 50 nodes and 100 edges and values of k between 10 and 70. In Figure 11, we traced the same values of k using an RMFGEN graph with 140 edges and 48 nodes (spread over 12 square planes). In Figure 12, using values of k between 50 and 250, we examined an RMFGEN graph with 752 edges and 192 nodes (spread over 12 square planes). Graphing the number of iterations against the number of commodity groups $k' \leq k$, we observed that the number of iterations either grew linearly or grew linearly to a peak and then dropped. The drops may result from larger numbers of commodities making it possible to route commodities over shorter paths. In trying to find flows which give the edges equal congestions, the algorithm has more commodities at its disposal to congest each edge. In any case, the number of iterations grows no more than linearly with the number of commodity groups and therefore no more than linearly with the number of commodities.

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

We begin with a brief discussion of the first algorithm to which we have compared our algorithm. The algorithm is MCNF85, a special purpose simplex code for multicommodity flow problems written by Kennington [**10**], and we chose it for two reasons. First, we had access to the code on our machine. Second, and more importantly, previous tests by Adler, Karmarkar, Resende and Veiga [**1**] demonstrate its efficiency. Adler et al. compared three different codes for multicommodity flow: MINOS 5.0, an advanced implementation of the simplex method [**15**], MCNF85, and their own interior point method. Their experiments show that the running time of MINOS grows much faster than that of the other two algorithms and that, for the problems they tested, MCNF85 and the interior point algorithm have comparable running times. Thus we concluded that MCNF85 was one of the best codes available at that time.

We have also compared our algorithm to a state-of-the-art interior point code called ADP[**9**]. ADP is an approximate dual projective interior point code, written by Karmarkar and Ramakrishnan, and is the latest variant of the interior point algorithm. The algorithm alternates between objective steps and centering steps; the motivation of the algorithm is to stay very close to the "central trajectory" of the polytope. The objective steps are dual affine scaling steps while the centering steps are reciprocal-estimates-improvement step. The implementation of the algorithm took about three years, and uses the iterative technique of preconditioned conjugate gradient at each step to compute the improving direction. The code is especially suited for solving very large linear programs with a particular structure, since in many cases only an approximate solution to the linear system is needed to solve the problem.

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

4.3.1. *The Results.* The results of our experiments appear in Figure 1. The experiments in this table were performed on a Silicon Graphics 4D/340S. They show that as the number of commodities increases, the running time of MCNF85 grows much more rapidly than the running time of our algorithm for graphs of all sizes. The difference does not arise simply because we group the commodities (they could incorporate grouping in their algorithm too). Hardly any grouping occurred in the graphs with 500 nodes and 70 or less commodities, and the running time of our algorithm still grew much more slowly than the time for MCNF85. In fact, as discussed above, the running time of our algorithm grows slower than k while rough analysis of the data shows that the time for MCNF85 grows at least as fast as k^2. Since the size of the linear program grows by k^2, this growth is not particularly surprising.

They also show that our running times are consistently smaller than those of the interior point code ADP. In contrast to the other linear-programming based algorithm, MCNF85, the dependence of the running time on the number of commodities does not appear to grow quadratically. Yet, our algorithm consistently runs faster than ADP. We note that only a small number of tests were performed with ADP and that they did not include the largest problems that we have. We hope to be able to perform more tests in order to allow us to draw more significant conclusions.

Our algorithm will be able to solve large and previously unsolvable multicommodity flow problems. We have already shown that we can solve a 700 commodity problem faster than MCNF85 can solve a 70 commodity problem. For large graphs with small numbers of commodities, our algorithm is slower than MCNF85. However, the rapid growth rate of MCNF85 with respect to the number of commodities makes our algorithm more desirable for problems with more than a few commodities. We note that one of the motivations for this work comes from multicommodity flow problems which arise in approximating a number of NP-hard problems. (See **[14]**,**[11]**,**[12]**, and **[13]** for details.) These problems have large numbers of commodities, i.e., at least as many commodities as the number of nodes. Our algorithm provides a practical means for solving such problems.

4.4. An Anomaly. In one case, a problem with 49 nodes, 260 edges, and 585 commodities using actual data from GTE, our algorithm performed much more poorly than the linear programming algorithm. Though our algorithm ran for only 3745 iterations, a reasonable number, those iterations took a total of 18.4 hours of CPU time. We attribute this anomaly to inefficiency in the minimum-cost flow routine since minimum-cost flow computations accounted for over 99.8% of the running time. The theory shows that minimum-cost flow computations dominate the running time of the algorithm, but even for the much larger RMFGEN graph with 500 nodes and 1025 edges, minimum-cost flow computations generally took less than 80% of the time. For small graphs, they generally took between 40 and 50 percent of the time. See Figure 2 for a more detailed description of the times. The time spent solving the GTE problem was not equally divided between iterations. Iterations including the termination check aside, most iterations took less than 100 milliseconds. Some iterations, however, took hundreds of seconds, up to 1000 times the normal duration.

With the help of a number of other researchers, we have verified that these are problems on which RELAXT-III takes an inordinately long amount of time. A number of people have run these problems on their codes and observed no anomalous behavior, i.e., the running times for this set of problems are all approximately the same. In order to estimate a more realistic running time for this problem, we will compute an upper bound on the what the running time would have if we were using the RNET code of Grigoriadis. Joseph Cheriyan **[4]** has reported that on a representative sample of these minimum-cost flow problems, the running time of RNET on a SPARC2 (which is slower than our machine) never exceeds 0.66 seconds. Using the estimate that 50% (see Figure 2) of the

Problem Specification	MCNF85	ADP		Our algorithm	
(n, m, k, GEN)		$\epsilon = .01$	$\epsilon = .001$	$\epsilon = .01$	$\epsilon = .001$
(50, 100, 20, NG)	49	30	39	20	103
(50, 100, 50, NG)	397	118	153	35	43
(50, 100, 70, NG)	857	199	284	29	33
(48, 140, 10, RMF)	8			13	13
(48, 140, 20, RMF)	24	41	56	23	23
(48, 140, 30, RMF)	69			18	35
(48, 140, 40, RMF)	122			25	38
(48, 140, 50, RMF)	216	250	300	21	71
(48, 140, 60, RMF)	316			40	61
(48, 140, 70, RMF)	470	328	347	45	62
(500, 2075, 10, RMF)	87			831	5230
(500, 2075, 20, RMF)	608	2220	2832	1484	2641
(500, 2075, 30, RMF)	1831			2625	3881
(500, 2075, 40, RMF)	6571			3762	6084
(500, 2075, 50, RMF)	15601	7500	9300	4710	7401
(500, 2075, 60, RMF)	18449			3819	6201
(500, 2075, 70, RMF)	34362	19860	13200	4435	8258
(500, 2075, 700, RMF)	B			22411	
(192, 748, 50, RMF)	2702			240	589
(192, 748, 250, RMF)	85754			637	1571
(49, 260, 585, none	1373			2472 (estimate)	

FIGURE 1. Running time comparison of our algorithm and Kennington's algorithm. Running times are in seconds on a Silicon Graphics machine. NG is NETGEN and generator RMF is RMFGEN. The last problem is the problem defined in Section 4.4 B stands for a breakdown in the computer and a blank signifies that the test was not performed.

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

FIGURE 2. Percentage of Time that our algorithm spent performing minimum-cost flows. The data is gotten from the UNIX profiling routine **prof**. NG is NETGEN and generator RMF is RMFGEN. The last problem is the problem defined in Section 4.4.

time is spent in the minimum-cost flow computations, we arrive at a figure of 2472 seconds as a "reasonable" upper bound on the running time of this instance. It is intriguing that the one anomaly occurred on the one real-world instance. This gives more evidence to the belief that every effort should be made to find real-world instances on which to test programs.

5. Conclusions

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

On the problems we tested, the running time of our algorithm grew much slower as a function of k than that of Kennington's algorithm. This implies that our algorithm is preferable to one of the best network simplex based approaches for problems with large numbers of commodities. Our code also performed well against ADP, a state-of-the-art interior point code.

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

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

We are aware of two other implementations of combinatorial algorithms to which we should compare our algorithm. The first, by Shahrokhi and Matula [**17**], works only for graphs in which every capacity and demand is 1, but it would still be interesting to see how our algorithm compares to theirs on this class of graphs. The second, by Schneur [**16**], also works by gradually rerouting flow. She has shown that her algorithm runs well on many problems. We would like to compare the algorithms on the same machine and the same problems.

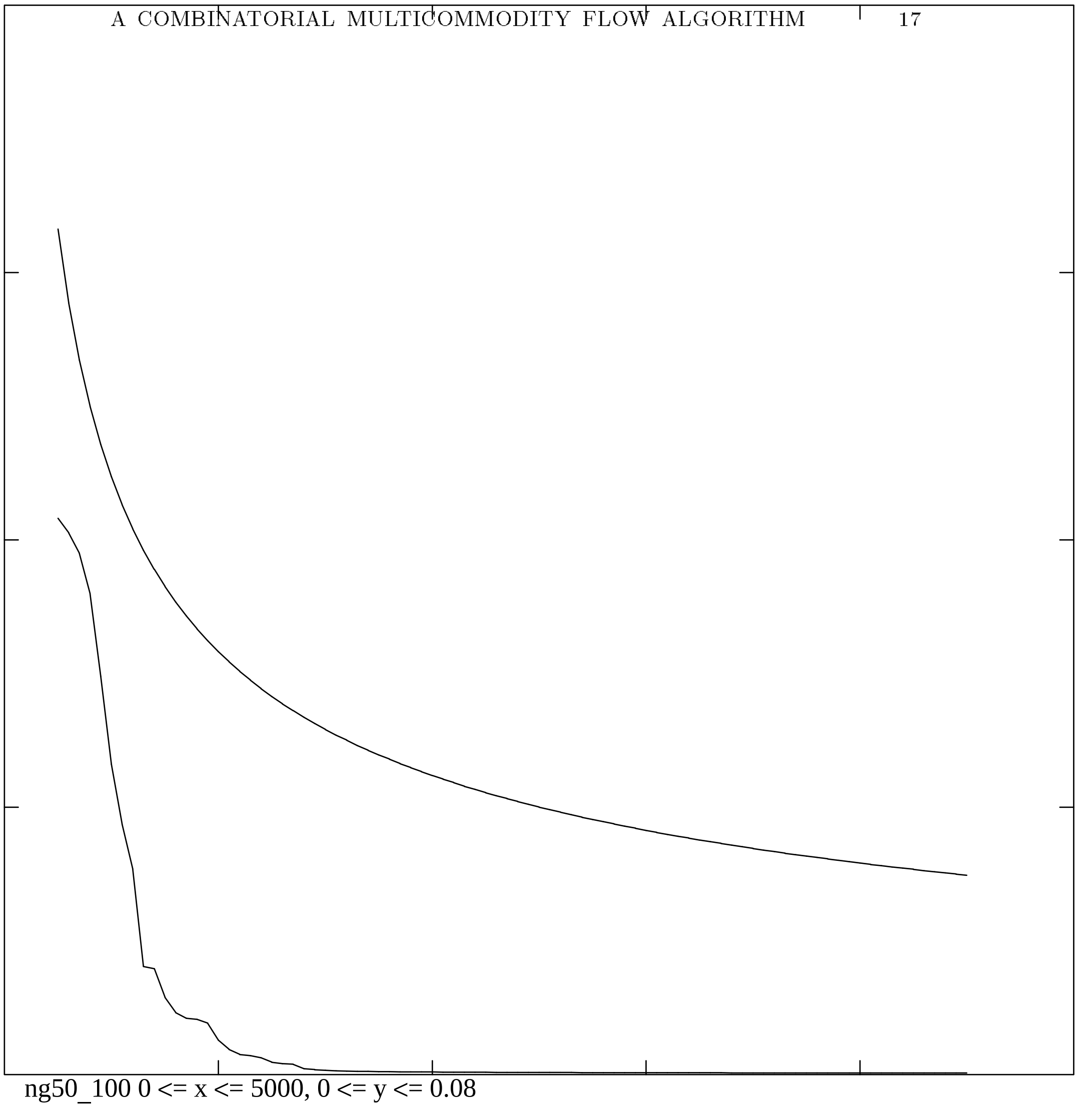
Acknowledgments

Several people provided us with code and with multicommodity flow problems that were very useful in our work. We thank Dimitri Bertsekas for providing us with his minimum-cost flow code and Mauricio Resende for providing us with the multicommodity flow code MCNF85. We thank K. Ramakrishnan and Mauricio Resende for performing the interior point experiments. We thank David Johnson for an explanation of the code ADP. We thank Rina Schneur and Farhad Shahrokhi for providing us with problem instances and Cathy McGeoch and Mauricio Resende for providing us with problem generators. We thank Debbie Lam and Tom Leighton for work on the early stages of this project. We thank

David Johnson, Philip Klein, Cathy McGeoch, and Rina Schneur for helpful discussion and Perry Fizzano for a careful reading of a draft of this paper.

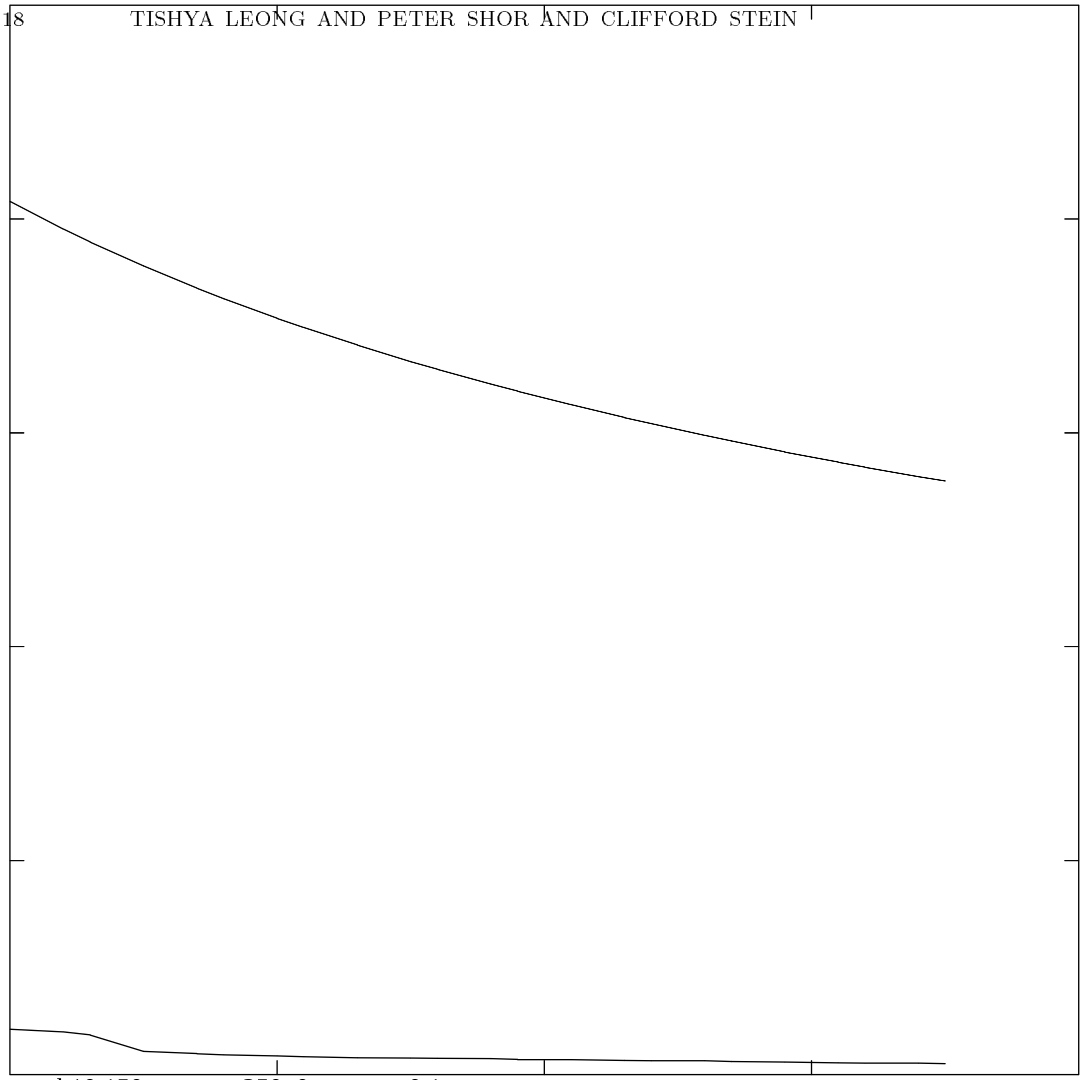
References

1. I. Adler, N. Karmarkar, M. Resende, and G. Veiga. An implementation of Karmarkar's algorithm for linear programming. *Mathematical Programming*, 44:297–335, 1989.
2. A. A. Assad. Multicommodity network flows - a survey. *Networks*, 8:37–91, 1978.
3. D. P. Bertsekas and P. Tseng. RELAXT-III: A new and improved version of the RELAX code. Technical Report LIDS–P–1990, MIT, July 1990.
4. J. Cheriyan, October 1991. Private communication.
5. A. V. Goldberg, Personal communication. Jan., 1991.
6. M. D. Grigoriadis and L. G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. Technical Report DCS-TR-273, Department of Computer Science, Rutgers University, New Brunswick, NJ, March 1991.
7. S. Kapoor and P. M. Vaidya. Fast algorithms for convex quadratic programming and multicommodity flows. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 147–159, 1986.
8. J. Kennington. A survey of linear cost multicommodity network flows. *Operations Research*, 26:206–236, 1978.
9. N.K. Karmarkar and K. G. Ramakrishnan. Computational results of an interior point algorithm for large scale linear programming. *Mathematical Programming*, 52:555–586 1991.
10. J. Kennington. A primal partitioning code for solving multicommodity flow problems (version 1). Technical Report Techincal Report 79009, Department of Industrial Engineering and Operations Research, Southern Methodist University, 1979.
11. P. Klein, A. Agrawal, R. Ravi, and S. Rao. Approximation through multicommodity flow. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 726–727, 1990.
12. P. Klein, S. A. Plotkin, C. Stein, and É. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. Technical Report 961, School of Operations Research and Industrial Engineering, Cornell University, 1991. A preliminary version of this paper appeared in *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 310–321, 1990.
13. T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 101–111, 1991.
14. T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431, 1988.
15. B.A. Murtaugh and M.A. Saunders. MINOS 5.0 user's guide. Technical Report Technical Report 83-20, Systems Optimization Laboratory, Stanford University, 1983.
16. R. Schneur. *Scaling algorithms for multicommodity flow problems and network flow problems with side constraints*. PhD thesis, MIT, Cambridge, MA, February 1991.
17. F. Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37:318 – 334, 1990.
18. P. M. Vaidya. Speeding up linear programming using fast matrix multiplication. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 332–337, 1989.



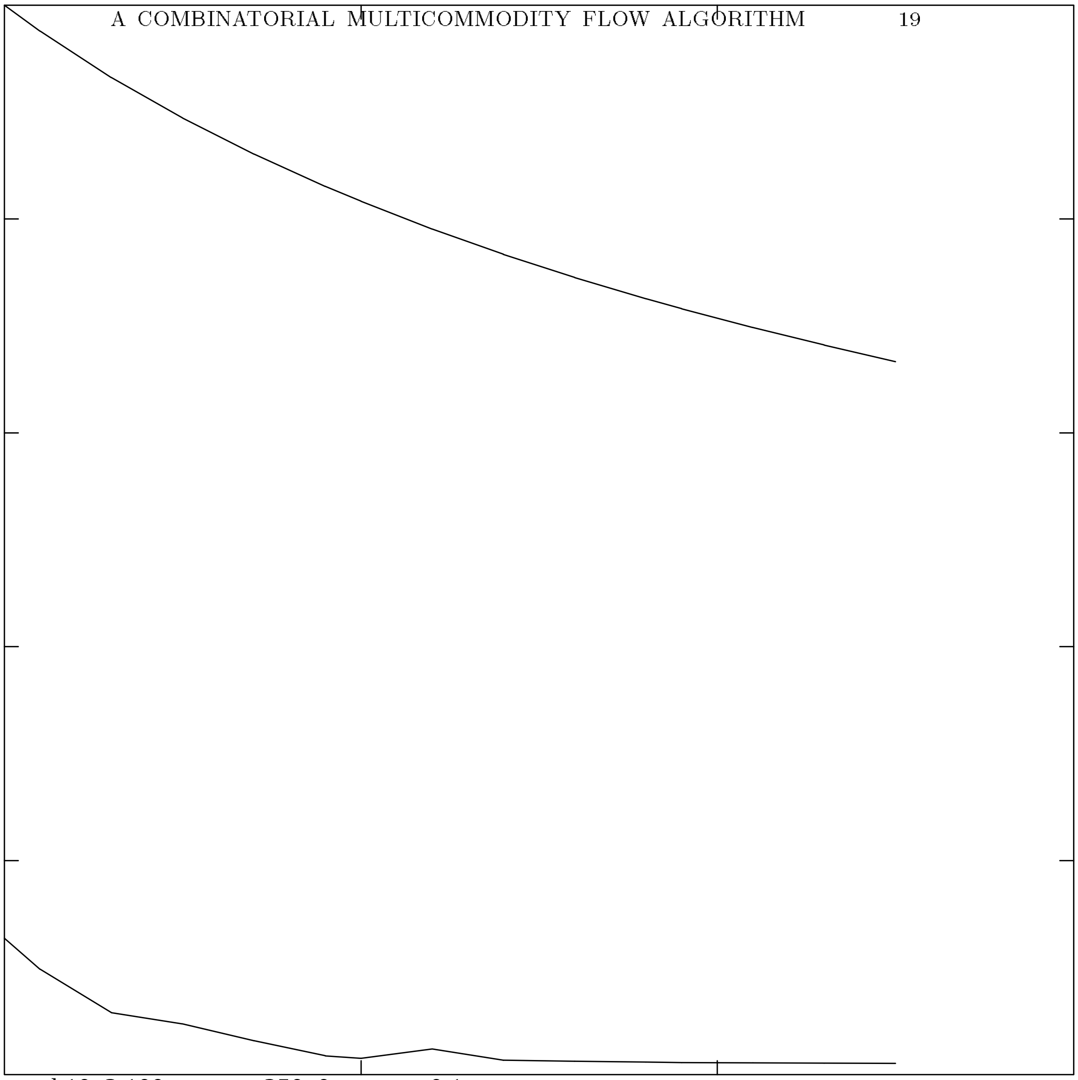
x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\#\text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

FIGURE 3. NETGEN graph with 50 nodes, 100 edges, and 20 commodities.



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

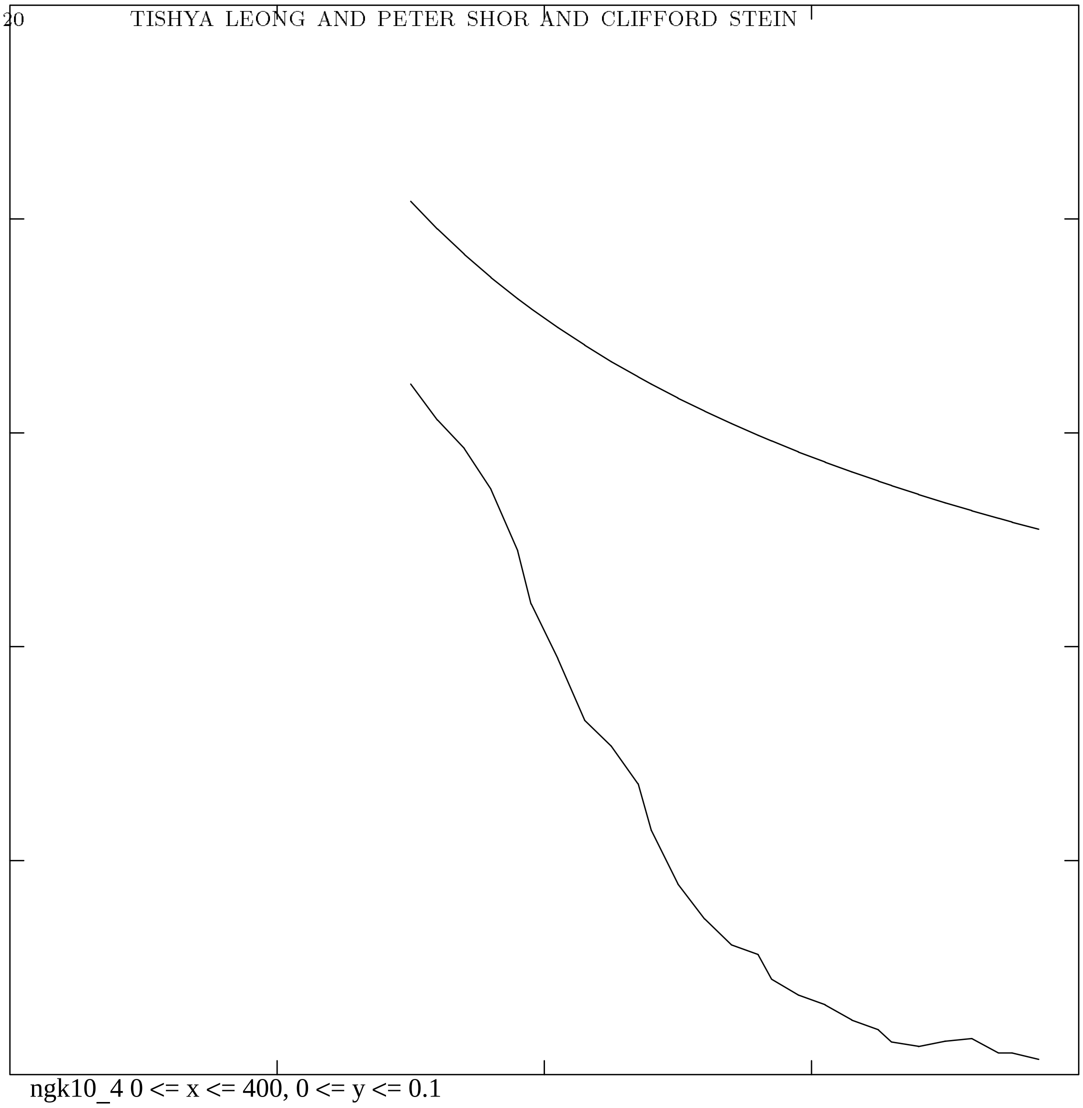
FIGURE 4. NETGEN graph with 50 nodes, 100 edges, and 10 commodities.



ngk10_3 100 <= x <= 250, 0 <= y <= 0.1

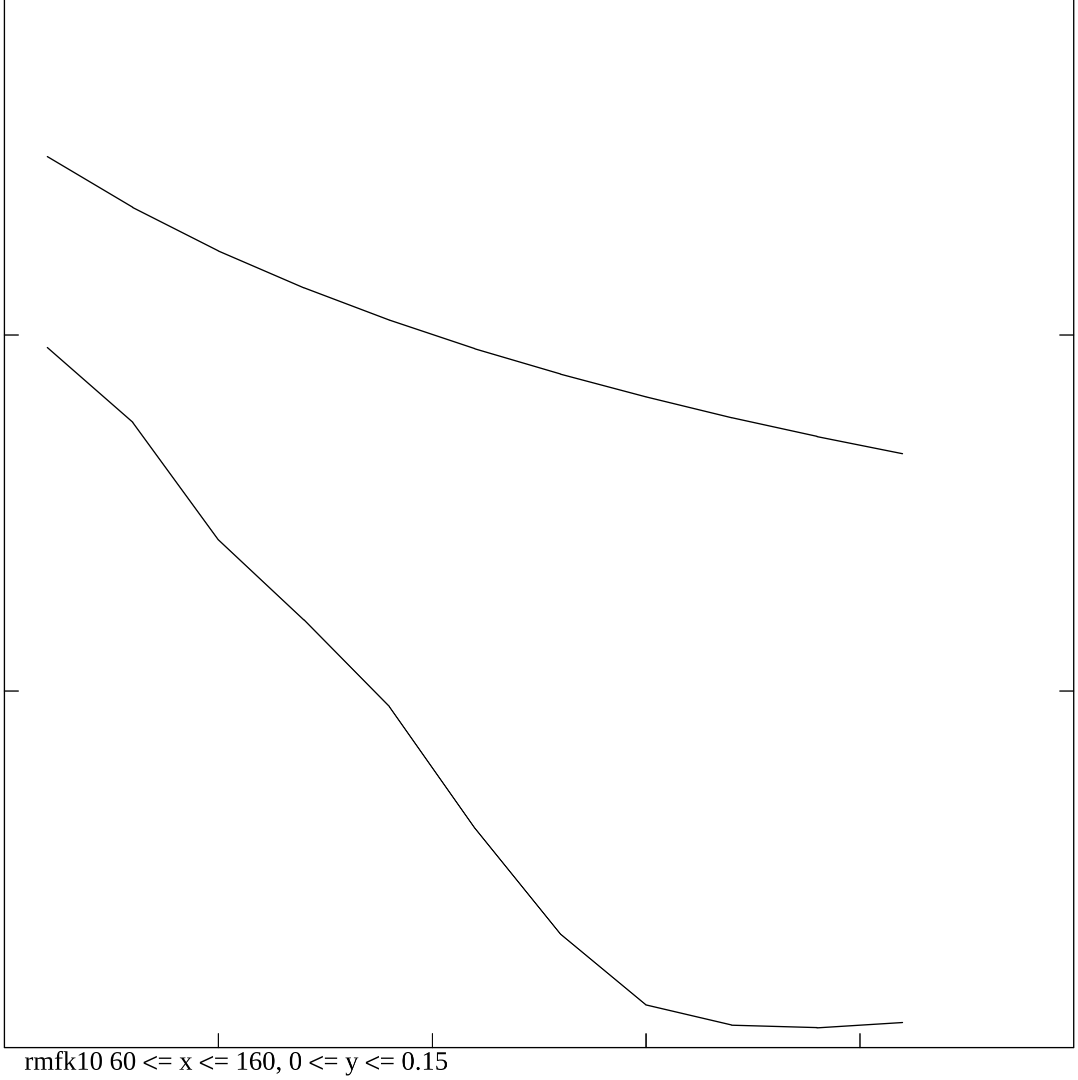
x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\#\text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

FIGURE 5. NETGEN graph with 50 nodes, 100 edges, and 10 commodities.



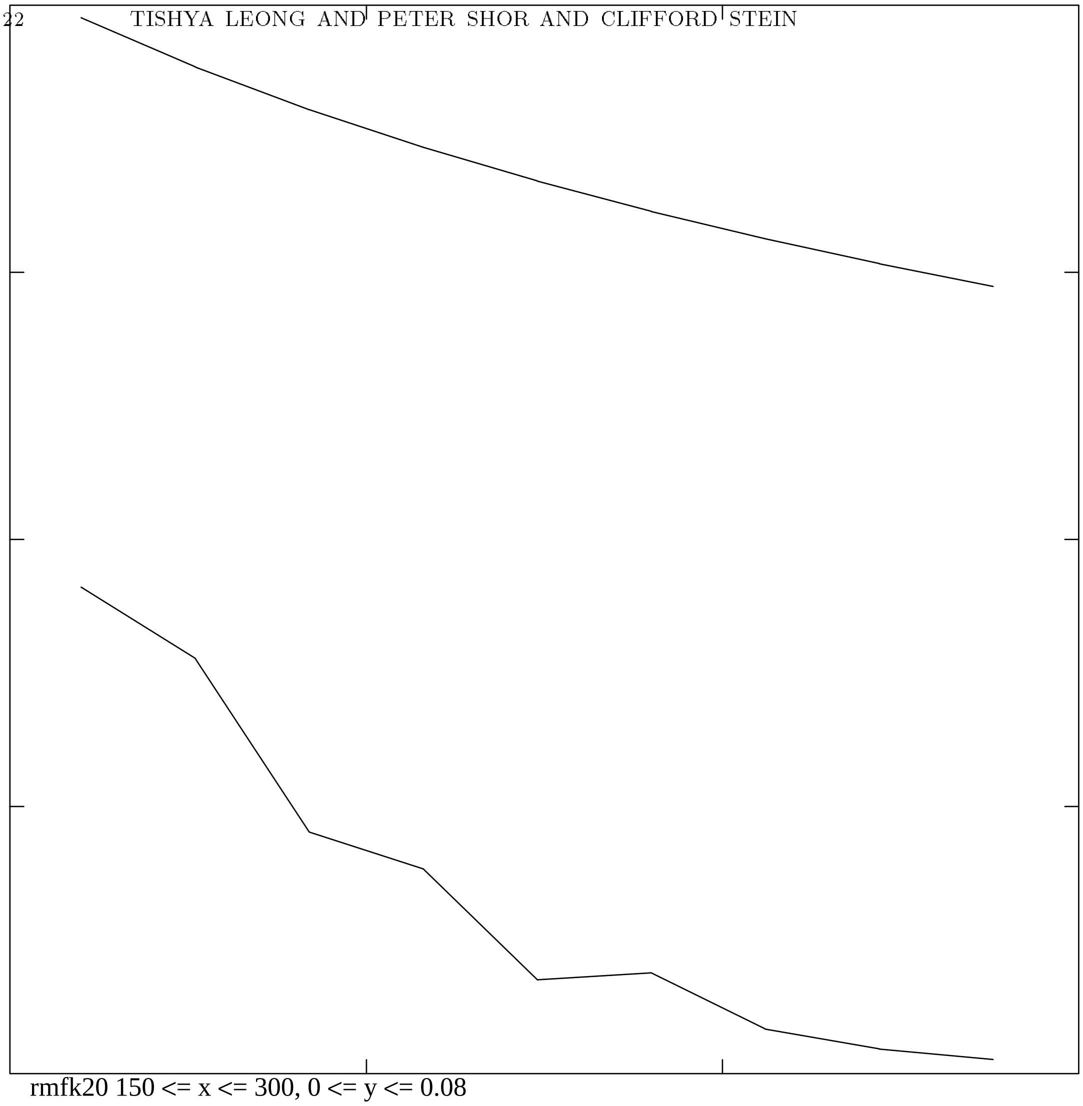
x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\#\text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

FIGURE 6. NETGEN graph with 50 nodes, 100 edges, and 10 commodities.



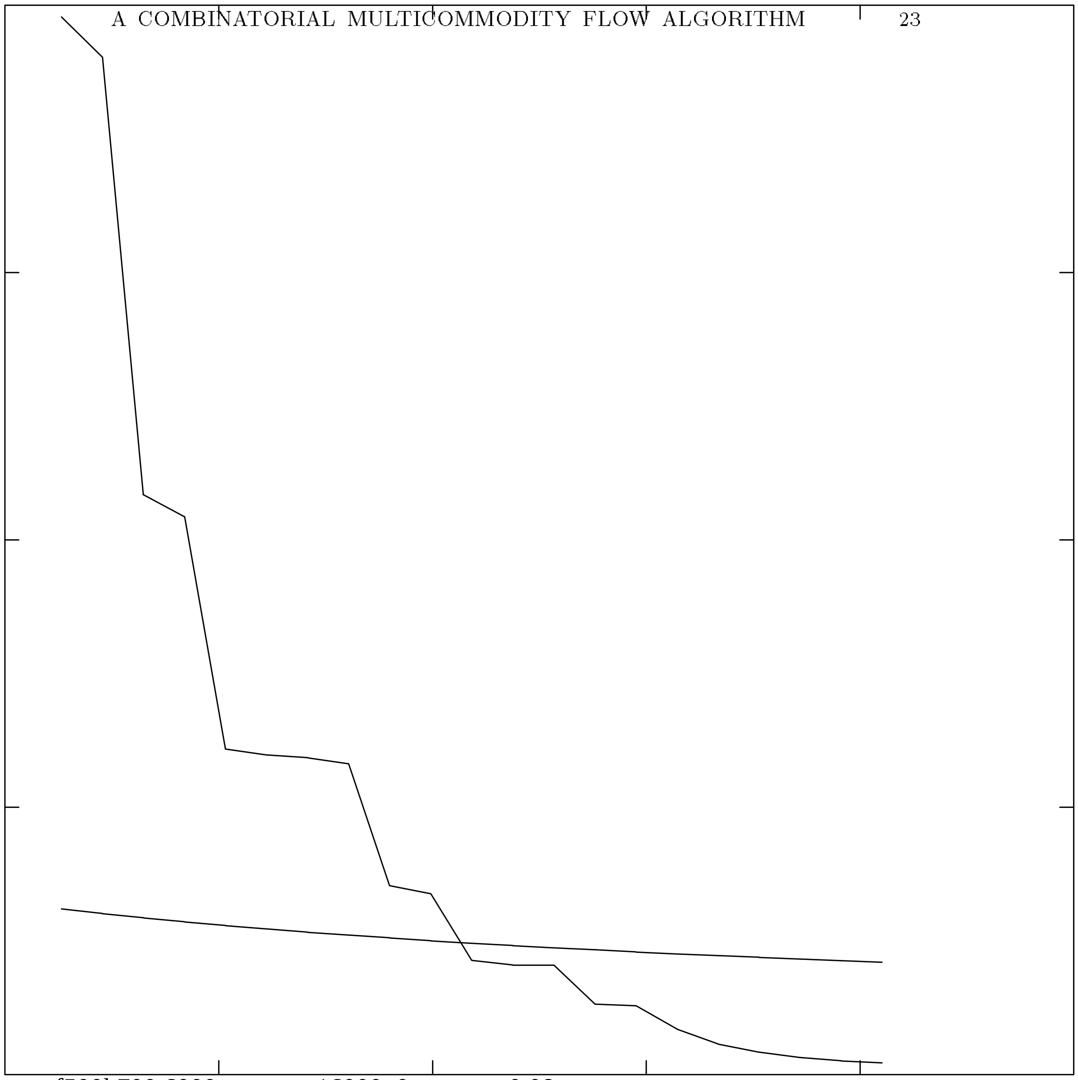
x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\# \text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

FIGURE 7. RMFGEN graph with 48 nodes, 140 edges, and 10 commodities.



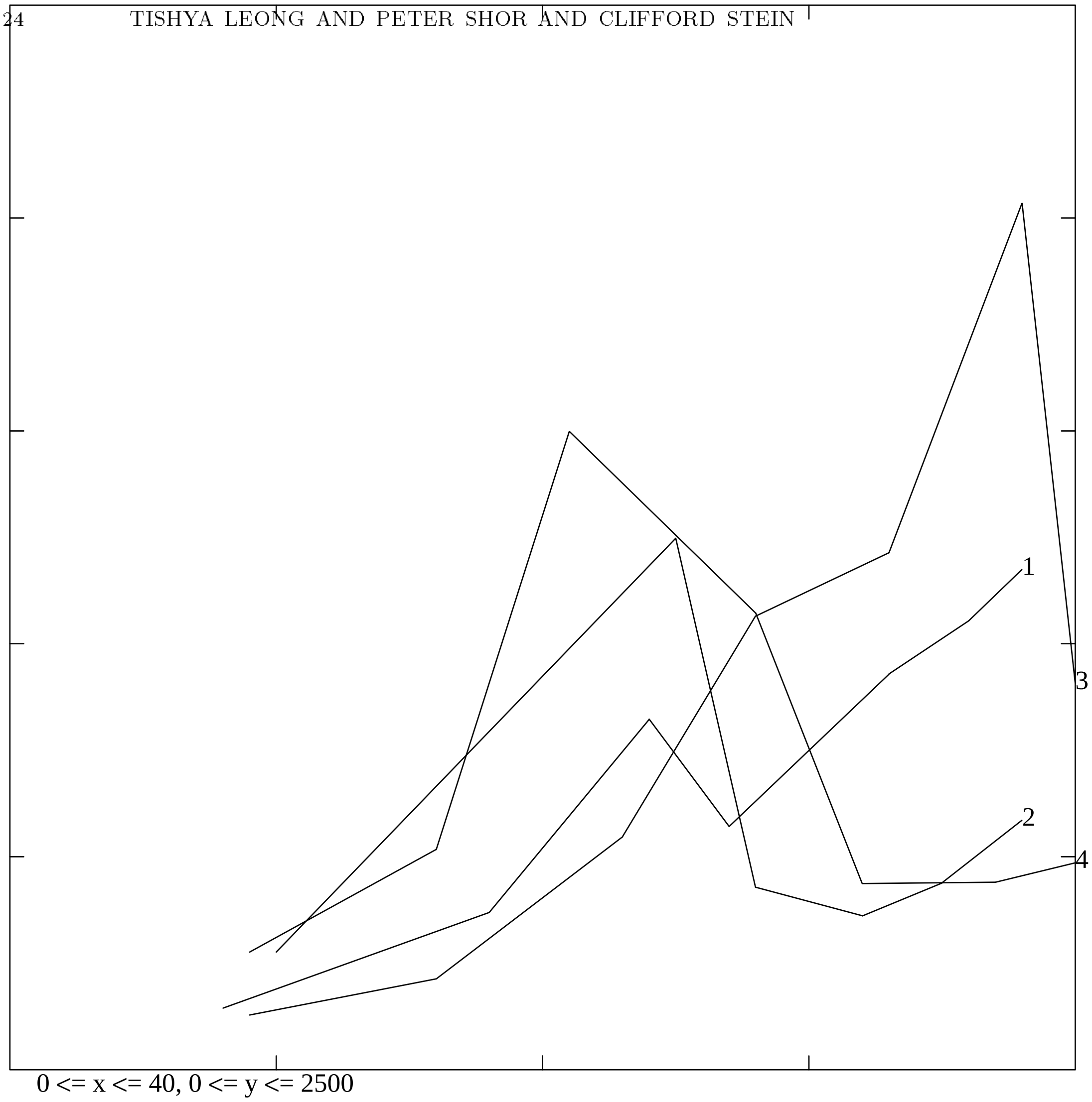
x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\#\text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

FIGURE 8. RMFGEN graph with 48 nodes, 140 edges, and 20 commodities.



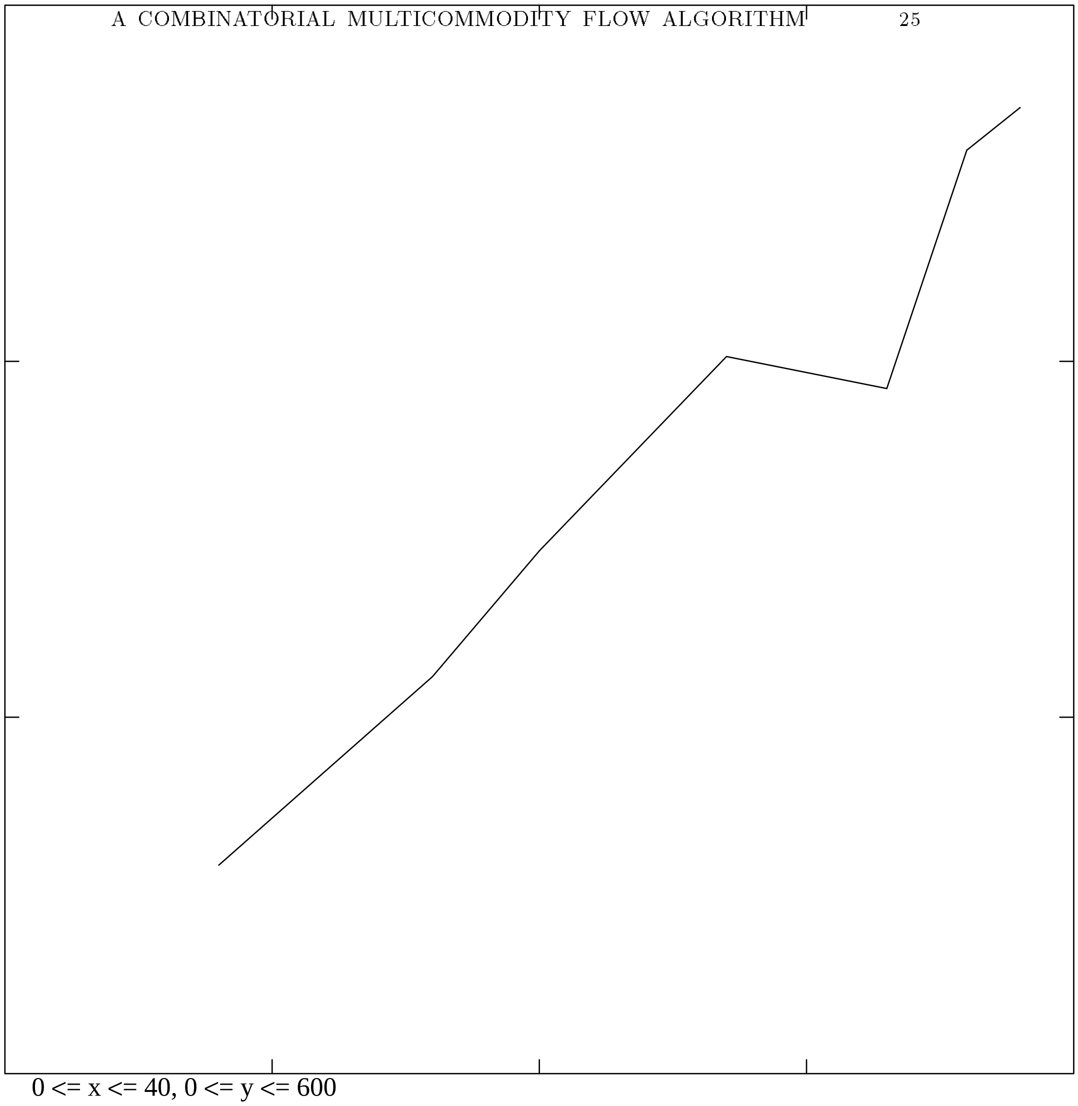
x-axis is # of iterations.
y-axis is ϵ.
The top curve is $1/\sqrt{\#\text{ of iterations}}$.
The bottom curve is the minimum ϵ achieved.

FIGURE 9. RMFGEN graph with 500 nodes, 2075 edges, and 700 commodities.



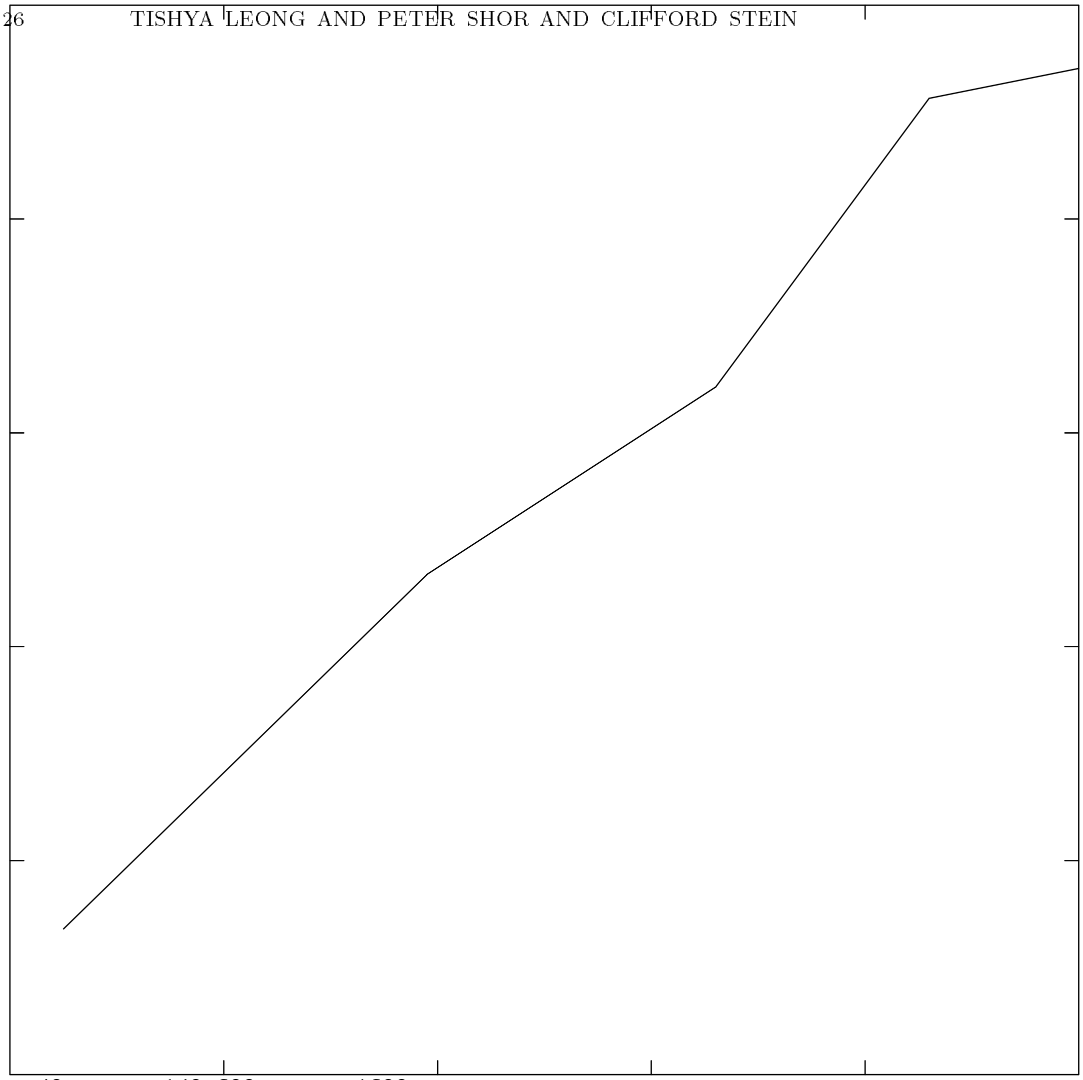
x-axis is # of commodity groups.
y-axis is # of iterations.
Each curve represents a set of runs on one of four different underlying graphs.

FIGURE 10. NETGEN graphs with 50 nodes, 100 edges, and from 10 through 70 commodities.



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

FIGURE 11. RMFGEN graphs with 48 nodes, 140 edges, and from 10 through 70 commodities.



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

FIGURE 12. RMFGEN graphs with 192 nodes, 740 edges, and from 50 through 250 commodities.

Department of Mechanical Engineering, Stanford University, Palo Alto, CA
E-mail address: taleong@leland.stanford.edu

AT&T Bell Laboratories, Murray Hill, NJ
E-mail address: shor@research.att.com

Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH
E-mail address: cliff@cs.dartmouth.edu