A Simple Local-Control Approximation Algorithm for Multicommodity Flow

Baruch Awerbuch *   Tom Leighton †

Abstract

In this paper, we describe a very simple $(1+\varepsilon)$-approximation algorithm for the multicommodity flow problem. The algorithm runs in time that is polynomial in N (the number of nodes in the network) and ϵ^{-1} (the closeness of the approximation to optimal). The algorithm is remarkable in that it is much simpler than all known polynomial time flow algorithms (including algorithms for the special case of one-commodity flow). In particular, the algorithm does not rely on augmenting paths, shortest paths, min-cost paths, or similar techniques to push flow through a network. In fact, no explicit attempt is ever made to push flow towards a sink during the algorithm.

Because the algorithm is so simple, it can be applied to a variety of problems for which centralized decision making and flow planning is not possible. For example, the algorithm can be easily implemented with local control in a distributed network and it can be made tolerant to link failures.

In addition, the algorithm appears to perform well in practice. Initial experiments using the DIMACS generator of test problems indicate that the algorithm performs as well as or better than previously known algorithms, at least for certain test problems.

*Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Supported by Air Force Contract AFOSR F49620-92-J-0125, NSF contract CCR-9114440, DARPA Contract N00014-92-J-1799, and a special grant from IBM.

†Department of Mathematics and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. This research was supported in part by Air Force Contract AFOSR F49620-92-J-0125 and DARPA Contracts N00014-91-J-1698 and N00014-92-J-1799.

1 Introduction

The *multicommodity flow problem* consists of shipping several different commodities from their respective sources to their sinks through a common network so that the total flow going through each edge does not exceed its capacity. Associated with each commodity is a *demand*, which is the amount of that commodity that we wish to ship through the network. Given a multicommodity flow problem, we would like to know if there is a *feasible flow*, i.e. a way of shipping the commodities that satisfies the demands as well as the capacity constraints. More generally, we might also like to know the maximum value z such that at least z percent of each demand can be shipped without exceeding the capacity constraints. The latter problem is known as the *concurrent flow problem*, and is equivalent to the problem of determining the minimum ratio by which the capacities must be increased in order to satisfy 100% of all of the demands.

Multicommodity flow problems arise in a wide variety of contexts and have been extensively studied during the past several decades. For example, many product distribution, traffic planning, and scheduling problems can be expressed and solved as a multicommodity flow problem. In addition, it has recently been discovered [9, 8] that a wide variety of NP-hard problems (such as graph partitioning, minimum feedback arc set, minimum cut linear arrangement, minimum 2D area layout, via minimization, and optimal matrix arrangement for nested disection) can be approximately solved using multicommodity flow algorithms. Many packet routing and communication problems can also be expressed as multicommodity flow problems but since no local-control, on-line algorithms for multicommodity flow were previously known, flow techniques are not commonly used to solve such problems.

Not surprisingly, the prior literature on flow problems is extensive. Much of the past work centers on the much simpler problem of 1-commodity flow (also known as the *max-flow* problem). A survey of the many 1-commodity algorithms can be found in [6]. Most of these algorithms rely on finding augmenting paths to increase the flow from source to sink. An exception is the recent algorithm of Goldberg and Tarjan [6] (which is based on an algorithm of Karzanov [7]). The latter algorithm maintains a preflow on the network and pushes local flow excess toward the sink along what is estimated to be a shortest path. The best of these algorithms run in $\tilde{O}(NM)$ steps, where N is the number of nodes in the network and M is the number of edges in the network.

There has been much less progress on the multicommodity flow problem, perhaps because handling K commodities seems to be much more difficult than handling one commodity. All exact algorithms for multicommodity flow are based on linear programming, all have horrendous running times (even though polynomial), and none are used for large networks in practice.

The situation is somewhat better for approximation algorithms, however. In particular, Vaidya [12] developed a $(1+\varepsilon)$-approximation algorithm for the min-cost multicommodity flow problem based on linear programming that uses (roughly) $O(K^{\frac{7}{2}}NM^{\frac{5}{2}}\log(DU\varepsilon^{-1}))$ steps where K is the number of commodities, N and M are as before, D is the largest demand, and U is the largest edge capacity. More recently, Leighton et al [10] discovered a purely combinatorial $(1+\varepsilon)$-approximation algorithm based on 1-commodity min-cost flows that runs in $O(K^2NM\varepsilon^{-2}\log K\log^3 N)$ steps. (By using randomization, the running time of the latter algorithm can be improved by a factor of K).

All of the known multicommodity flow approximation algorithms are fairly complicated to describe (and to analyze) and none is well-suited for applications in contexts requiring local control and/or local decision-making. For example, none of the algorithms are amenable to implementation on a fault-prone distributed network. In fact, of the many 1-commodity flow algorithms known, only the Goldberg-Tarjan algorithm is implementable in a distributed network, and even there, the algorithm needs to maintain shortest path information and it does not tolerate undetectable faults or dynamic changes in the network structure.

In this paper, we describe a very simple approximation algorithm for the multicommodity flow problem. The algorithm is based on a simple "edge balancing" technique. In particular, the algorithm attempts to send a commodity across an edge $e=(u,v)$ if there is more of the commodity queued at u than there is queued at v. Contention for capacity is resolved by shipping the commodity which has the largest disparity in queue size across e. No attempt is made to find augmenting paths, shortest paths, min-cost paths, or even any path from a node to a sink. Commodities are simply entered at their respective sources, according to their demands, emptied from their sinks when present, and otherwise locally balanced across each edge.

To simplify presentation, the version of the algorithm that we describe in this extended abstract is composed of "parallel rounds". At the start of each round, d_i units of commodity i are added to the source for commodity i for $1 \le i \le K$. Then flow is pushed across each edge in an attempt to balance each commodity across each edge (up to the limits imposed by capacity). (We will explain precisely how to do this in Section 2.) Then any commodity present at the appropriate sink is removed from the network.

The key to the success of the algorithm is that the amount of flow contained in all the queues stays bounded over time (provided that there exists a feasible flow with demands of $(1+\varepsilon)d_i$ for each commodity i). Thus, when the algorithm is run for a large number of rounds, the flow that remains in the system will become very small compared to the flow that was pumped into the system and we can compute an approximate solution to the original flow problem by seeing where the flow went during the course of the algorithm. In distributed network applications, we never need to compute the approximate solution explicitly – the commodities just flow through the network to their destinations on their own (without having any idea where the destination is in the network) and we are guaranteed to get near-optimal throughput.

The algorithm is similar in spirit to a physical network containing fluids. For each fluid, there is a source and a drain. As fluid enters at the source, pressure builds and the fluid spreads through the network. Eventually, each fluid reaches its sink and a fluid path is set up between source and sink. The key to the analysis is to show that no fluid can be blocked by the others for any serious period of time, and to show that the cumulative heights of all the fluids stays bounded over time. It is intuitively clear that these properties hold for one-commodity flow problems, but much less clear that they should hold for multicommodity flow problems. (In fact, we will describe a simple 4-commodity problem in Section 3.2.1 for which the maximum concurrent flow is 25%, but for which no flow ever reaches the correct sink. Hence, the threat of deadlock or blocking must be taken seriously.)

The algorithm is also similar in spirit to an algorithm proposed by Dennis [5] in 1964. In particular, Dennis showed how to model a multicommodity flow problem as a collection of tightly coupled electrical networks (one for each commodity). He then argued that the optimal flow could be found by measuring the current flows in the various networks. An important part of this work is the claim that the electrical system converges to a steady state (although no time bounds for convergence are established). Although the construction in Dennis' paper results in a highly non-linear system of equations, Dennis also observed that the linear approximation to this system of equations suggests an algorithm for multicommodity flow in which some sort of "edge-balancing" technique is used to find the optimal flow. The running time of this algorithm was not made clear, however.

Edge balancing techniques have also been used for solving problems in fault-prone networks, such as end-to-end communication [4, 2, 1] and load balancing [3].

The main contribution of this paper is the introduction of a simple multicommodity flow approximation algorithm for which we are able to establish a polynomial bound on the running time. In particular, our algorithm takes at most $O(M^3 K^{5/2} L \varepsilon^{-3} \log K)$ steps to find a feasible flow (provided that there exists a feasible flow when the demands are increased by a factor of $1+\varepsilon$), where L is the length of the longest flow path. The algorithm can also be used to find a $(1-\varepsilon)$-optimal solution to the concurrent flow problem by using binary search.

It is certainly worth noting that the upper bound on running time just stated is asymptotically *inferior* to the best previously known multicommodity flow approximation algorithm. So why have we taken the time to write the paper? There are four reasons, as follows.

1. We suspect that the true running time of the algorithm (or a close variation) is much better—we just haven't proved it yet. In fact, we believe that the true running time may eventually be shown to be competitive with the algorithms developed in [10], at least for certain classes of flow problems. By establishing the potentially inferior bound contained in the paper as a benchmark, we hope to inspire the search for better bounds.

2. The algorithm is very simple and intuitive. It is even simpler than the known algorithms for 1-commodity flow. Understanding this algorithm would seem to be an important step towards a better understanding of flow problems in general.

3. The algorithm appears to work well empirically. In particular, Mark Tsimelzon [11] has run simulation tests for DIMACS test problems and has found that several variations of the new algorithm are comparable or superior in running time to the best known algorithms. Although the results are too preliminary to include here, we believe that the new algorithm may eventually prove to be of practical value.

4. Unlike all previously known algorithms for multicommodity flow, the new algorithm can be used for routing in environments where global control is not possible and where routing decisions need to be made locally. In contrast, all previously known algorithms require some amount of coordination and/or global control. Hence, multicommodity flow techniques can now be brought to bear on problems like message routing in networks, and vice versa.

The remainder of the paper is divided into sections, as follows. In Section 2, we provide a formal description of the algorithm. In Section 3, we analyze the behavior of the algorithm and prove that it runs in polynomial time. We also mention a nasty example for which there is no feasible solution but for which commodities gang up on each other in a way that keeps any flow for any commodity from ever reaching its sink. In Section 4, we conclude with some remarks and open questions.

2 The Algorithm

2.1 The Static Flow Problem

In a typical instance of a multicommodity flow problem, we are given a network with N nodes, M edges, and K commodities. Each edge e has a capacity $c(e)$ and each commodity i has a demand d_i. For simplicity, we will assume that each commodity has a single source and a single sink. (The algorithm can be easily modified to handle the case of multiple sources and sinks for each commodity.) In what follows, we will show how to set up flow paths from each source to each sink so that we can ship d_i units of commodity i from the ith source to the ith sink for $1 \le i \le K$ without violating capacity constraints. We will assume that there exists a feasible flow for the corresponding problem with demands $(1+3\varepsilon)d_i$ for commodity i, where $\varepsilon > 0$ is a parameter of our choosing. The algorithm can then be easily modified to obtain an approximation algorithm for the static concurrent flow problem.

2.2 The Continuous Flow Problem

In order to find a solution to the static flow problem just described, we will focus on a continuous version of the flow problem in which $(1+\varepsilon)d_i$ units of commodity i are pumped into the source for commodity i at the start of each *round.* During the round, we are allowed to move commodities through edges so that edge capacities are not exceeded and so that each unit of flow moves across just one edge in any round. Flow that reaches the correct sink is removed from the network at the end of each round.

An algorithm for the continuous flow problem just described will be considered to be *stable* if the total amount of flow residing in the network at any time remains bounded (by a function depending on K and M). In Section 2.4, we will describe an algorithm for the continuous multicommodity flow problem that is stable if there exists a feasible solution to the static flow problem with demands $\{(1+3\varepsilon)d_i\}$.

2.3 Reducing a Static Problem to a Continuous Problem

Any stable algorithm for a continuous flow problem with demands $\{(1+\varepsilon)d_i\}$ can be easily used to find a feasible solution to a static flow problem with demands $\{d_i\}$. We simply run the continuous algorithm until the amount of each commodity residing in the queues is at most an $\dfrac{\varepsilon}{(1+\varepsilon)}$ fraction of the total amount of that commodity that has been pumped into the network. The number of rounds R needed to reach this point depends on the upper bound on flow residing in queues.

After R rounds, we will have input $(1+\varepsilon)d_iR$ units of commodity i into the network. At most εd_iR units of commodity i remain in the network, and so d_iR units of commodity i have been pumped through the network. Since the flow has been pumped through in R rounds, an average of d_i units of commodity i is shipped per round. Hence, we can obtain a solution to the static problem by taking the history of the continuous solution and averaging (i.e., dividing by R). The total time needed to find the static solution will then be R times the time needed to implement each round of the continuous algorithm.

2.4 The Continuous Flow Algorithm

In what follows, we will describe the algorithm for the continuous version of the problem in which $(1+\varepsilon)d_i$ units of commodity i are pumped into the network at each step. We will assume that the network is directed (although we will concentrate our analysis on the undirected case where each simple edge with capacity c is replaced with the (oppositely) directed edges with capacity c). The analysis for the general directed case will be sketched in Section 3.2.3.

For technical reasons, we will not allow flow for commodity i to cross an edge e if $c(e) \le \dfrac{\varepsilon d_i}{M}$. It is easy to check that if there is a feasible solution to the unrestricted version of the static problem with demands $\{(1+3\varepsilon)d_i\}$, then there is a feasible solution to the $(c(e) \le \dfrac{\varepsilon d_i}{M})$-restricted version of the static problem with demands $\{(1+2\varepsilon)d_i\}$.

In the continuous flow algorithm, we will maintain a queue for each commodity at the end of each edge. The algorithm proceeds in rounds where each round consists of the following four phases.

Phase 1: *Add new flow to the sources.* In particular, add $(1+\varepsilon)d_i\delta_i^{-1}$ units of flow to each of the δ_i queues for commodity i at the source for commodity i $(1 \le i \le K)$, where δ_i is defined to be the number of edges incident to the source node for commodity i. Since the source has δ_i queues (one for each edge), we are adding a total of $(1+\varepsilon)d_i$ units of commodity i to the source for i.

Phase 2: *Push flow across each edge so as to balance the queues as much as possible.* More precisely, for each edge e, define

$$\Delta_i(e) = q_i(tail(e)) - q_i(head(e))$$

where $q_i(tail(e))$ denotes the height of the queue for commodity i at the tail of e and $q_i(head(e))$ denotes the height of the queue for commodity i at the head of e. To balance the queues across e, we will begin by pushing flow for that commodity i for which $\Delta_i(e)d_i^{-2}$ is maximized. The normalization by d_i^{-2} is needed to keep high-volume (i.e., high-demand) commodities from swamping out lower-volume commodities.

As commodity i is pushed across e, $\Delta_i(e)$ changes, and it may become desirable to push flow for other commodities. In general, we will push f_i units of commodity i across e, for each i, where $f_1, f_2, \ldots, f_K$ are chosen to maximize

$$\sum_{1\le i\le K} f_i(\Delta_i(e) - f_i)d_i^{-2}$$

subject to the constraints:

$$f_i \geq 0 \text{ for all } 1 \leq i \leq K, \tag{1}$$

$$\sum_{1 \leq i \leq K} f_i \leq c(e), \tag{2}$$

$$f_i = 0 \text{ if } c(e) \leq \varepsilon d_i / M \tag{3}$$

(We will show how to compute the optimal f_i's for each edge shortly.)

Phase 3: *Remove flow from sinks.* Zero out all queues for commodity i at the sink for i ($1 \leq i \leq K$).

Phase 4: *Rebalance at nodes.* Reallocate each commodity within each node so that the queues for commodity i are all equal within each node ($1 \leq i \leq K$).

2.4.1 Balancing Flow Across Edges

In Phase 2, we need to find $f_1, f_2, \ldots, f_K$ so that

$$\sum_{1 \leq i \leq K} f_i(\Delta_i(e) - f_i) d_i^{-2} \tag{4}$$

is maximized subject to the constraints in Equations 1–3. The reason for choosing the f_i's in this manner will become clearer in Section 3 when we analyze the performance of the algorithm. (Basically, choosing the f_i's in this way guarantees that the commodities with the biggest discrepancy across an edge will have priority, subject to a normalizing factor of d_i^{-2}. (The $-f_i$ term comes from the fact that the initial discrepancy lessens as we start moving flow.)

Using elementary calculus, it can be shown that for the optimal solution, there exists an $s \geq 0$ such that

$$f_i = \max\left(0, \frac{\Delta_i(e) - s d_i^2}{2}\right)$$

for all i such that $c(e) > \frac{\varepsilon d_i}{M}$. (If $c(e) \leq \frac{\varepsilon d_i}{M}$, then f_i is automatically set to 0.) This is because in the optimal solution, all nonzero f_i must have the same marginal contribution (call it s) to the sum in Equation 4 and this means that $(\Delta_i(e) - 2f_i)d_i^{-2} = s$ if $f_i > 0$.

The optimal value of s is the minimum $s \geq 0$ such that

$$\sum_{i : d_i < \frac{Mc(e)}{\varepsilon}} \max\left(0, \frac{\Delta_i(e) - s d_i^2}{2}\right) \leq c(e) \ .$$

This value can be found by sorting the values of $\Delta_i(e) d_i^{-2}$ for which $\Delta_i(e) > 0$ and $c(e) > \frac{\varepsilon d_i}{M}$ and using binary search to find the maximal set S^+ of such commodities for which

1) if $i \in S^+, j \notin S^+$, and $c(e) > \varepsilon d_j / M$, then $\Delta_i(e) d_i^{-2} > \Delta_j(e) d_j^{-2}$, and

2) $\displaystyle\sum_{i \in S^+} \frac{\Delta_i(e) - s d_i^2}{2} \leq c(e)$, where $\displaystyle s = \min_{i \in S^+} \frac{\Delta_i(e)}{d_i^2}$.

We can then compute s by setting

$$s = \max\left(0, \frac{\sum_{i \in S^+} \Delta_i(e) - 2c(e)}{\sum_{i \in S^+} d_i^2}\right).$$

3 Analysis of Running Time

3.1 The Cost for Each Round

Each round of the continuous algorithm can easily be implemented in $O(MK \log K)$ steps. Phases 1 and 3 each take $K\delta$ steps steps where δ is the maximum node degree. Phase 4 takes $2MK$ steps. The only nontrivial phase is Phase 2 which requires $O(K \log K)$ steps for each edge due to the sorting operation and the binary search process used to construct S^+. Hence, Phase 2 requires requires $O(MK \log K)$ steps, which dominates the running time.

3.2 The Number of Rounds Required

The problem of upper bounding the cumulative queue-sizes is more challenging. In what follows, we will show that at most $O(M^2 K^{3/2} L \varepsilon^{-2} d_i)$ units of commodity i is ever in the network at any time, no matter how long the algorithm is run. (L is a parameter that bounds the maximum path length of any flow from source to sink in a feasible solution. At worst $L \leq N$, but often L is much smaller.) Since d_i units of commodity i enter the system during each step, we can use the analysis from Section 2.3 to conclude that we need only run the continuous algorithm for $O(M^2 K^{3/2} L \varepsilon^{-3})$ rounds in order to find a feasible solution to the static flow problem. Hence, we will be able to solve the static problem with demands $\{d_i\}$ in a total of $O(M^3 K^{5/2} L \varepsilon^{-3} \log K)$ steps.

3.2.1 A Nasty Example

The problem of upper bounding queue sizes is not completely trivial, as is evidenced by the following simple example. In the example, the network consists of a ring with four nodes, $\{v_1, v_2, v_3, v_4\}$, four edges, and four commodities. The source for commodity i is v_i and the sink is v_{i+2} (with indices computed modulo 4). The demand for each commodity is 4 and the capacity of each edge (in each direction) is 1.

It is clear that there is a feasible solution to the preceding problem that ships 25% of each commodity. Unfortunately, using the continuous algorithm described in Section 2, no part of any flow will ever reach the correct sink. This is because commodity i will dominate the edges leading from v_i for each i and so no flow will ever get to take the second step to its destination.

Fortunately, we will show in what follows that such a deadlocking phenomena cannot occur if there is a feasible solution to the flow problem that can ship a factor $(1 + 3\varepsilon)$ more of each commodity, and we will be able to attain nearly 100% throughput while remaining stable.

3.2.2 Proof of Stability

To prove that the continuous algorithm is stable, we will use a simple potential function argument where the potential of an item of flow roughly corresponds to its height in a queue. More precisely, we define the potential of the system at any point in time to be

$$\Gamma = \sum_{e \in E} \sum_{1 \le i \le K} \left(\frac{q_i(head(e))}{d_i} \right)^2 + \left(\frac{q_i(tail(e))}{d_i} \right)^2$$

where, as before, $q_i(head(e))$ is the height of the queue for commodity i at the head of edge e.

In what follows, we will show that if the potential in the system is small at the beginning of a round, then it does not increase by more than a small amount during the round. Even more importantly, if the potential is large at the beginning of a round, then it decreases during the round. By combining these two facts, we will be able to show that the potential stays bounded and that the algorithm is stable.

We first analyze the increase in potential that is incurred during Phase 1. Let q_i denote the size of the δ_i queues for commodity i at the source for i at the beginning of the round. (At the beginning of the round, the queues for each commodity are balanced within each node.) During Phase 1, each of these δ_i queues increases in size from q_i to $q_i + \dfrac{(1+\varepsilon)d_i}{\delta_i}$ for $1 \le i \le K$. The total increase in potential is thus

$$\begin{aligned}
&\sum_{1 \le i \le K} \delta_i \left[\left(\frac{q_i + (1+\varepsilon) d_i \delta_i^{-1}}{d_i} \right)^2 - \left(\frac{q_i}{d_i} \right)^2 \right] \\
&= \sum_{1 \le i \le K} \left[\frac{2(1+\varepsilon) q_i}{d_i} + \frac{(1+\varepsilon)^2}{\delta_i} \right] \\
&\le 2(1+\varepsilon) \sum_{1 \le i \le K} q_i d_i^{-1} + (1+\varepsilon)^2 K \ . \qquad (5)
\end{aligned}$$

Since Phases 2–4 cannot increase the potential, the value in Equation 5 serves as an upper bound on the amount by which the potential of the system can increase during any round.

We next show that the potential decreases by a significant amount during Phase 2. This will be the heart of the argument.

Because there exists a feasible solution to the static problem with demands $\{(1+3\varepsilon)d_i\}$, there is a feasible solution to the static problem with demands $\{(1 + 2\varepsilon)d_i\}$ for which no commodity i uses an edge e with capacity $c(e) \le \dfrac{\varepsilon d_i}{M}$. Hence, for this solution, there exist M elementary flow paths $P_{i,j}$ and volumes $g_{i,j}$ for each commodity, for which:

1. $P_{i,j}$ is a simple path from the source for commodity i to the sink that carries $g_{i,j}$ flow for $1 \le i \le K$, $1 \le j \le M$,
2. $\sum_{1 \le j \le M} g_{i,j} = g_i = (1 + 2\varepsilon)d_i$,
3. $\sum_{1 \le i \le K} \sum_{1 \le j \le M} g_{i,j}(e) \le c(e)$ for all edges e where $g_{i,j}(e) = g_{i,j}$ if $e \in P_{i,j}$ and $g_{i,j}(e) = 0$ otherwise, and
4. $g_{i,j}(e) = 0$ if $c(e) \le \dfrac{\varepsilon d_i}{M}$.

Define L to be the length of the longest $P_{i,j}$, and let $g_i(e) = \sum_{1 \le j \le M} g_{i,j}(e)$. In what follows, we analyze the drop in potential that would occur if we sent $g_i(e)$ units of commodity i across edge e for each i and e. (Note that sending $g_i(e)$ units of commodity i across e may result in pushing flow uphill or in a negative queue height at the tail of e, but that is OK for the purposes of the analysis. The algorithm, of course, would never do such a thing.)

For some commodity i and edge e, the drop in potential obtained by pushing $g_i(e)$ units of commodity i across e is easily computed to be

$$\frac{2g_i(e)(\Delta_i(e) - g_i(e))}{d_i^2} \ .$$

Summing over i, we find that the potential drop for the queues at either end of the edge is

$$\sum_{1 \le i \le K} \frac{2g_i(e)(\Delta_i(e) - g_i(e))}{d_i^2} \ . \qquad (6)$$

Since $g_i(e)$ satisfies the conditions on f_i in Equations 1 – 3, and since the flows f_i chosen by the algorithm maximize Equation 4, we can conclude that the potential in queues for e drops by (at least) the amount in Equation 6 during Phase 2 (even though the algorithm has no knowledge of the $g_i(e)$'s). Hence, the overall potential drops by at least

$$\sum_{e\in E}\sum_{1\le i\le K}\frac{2g_i(e)(\Delta_i(e)-g_i(e))}{d_i^2}$$

during Phase 2.

Simplifying the preceding expression, we find that the potential drop is at least

$$\begin{aligned}
&\sum_{1\le i\le K} 2d_i^{-2}\sum_{e\in E} g_i(e)\Delta_i(e) - \sum_{1\le i\le K} 2d_i^{-2}\sum_{e\in E} g_i(e)^2 \\
&\ge \sum_{1\le i\le K} 2d_i^{-2}\sum_{1\le j\le M} g_{i,j}\sum_{e\in P_{i,j}}\Delta_i(e) \\
&\quad - \sum_{1\le i\le K} 2d_i^{-2}g_i^2L \\
&= \sum_{1\le i\le K} 2d_i^{-2}\sum_{1\le j\le M} g_{i,j}[q_i+(1+\varepsilon)d_i\delta_i^{-1}] \\
&\quad -2(1+2\varepsilon)^2KL \\
&= \sum_{1\le i\le K} 2d_i^{-2}g_i[q_i+(1+\varepsilon)d_i\delta_i^{-1}] \\
&\quad -2(1+2\varepsilon)^2KL \\
&= \sum_{1\le i\le K} 2(1+2\varepsilon)d_i^{-1}[q_i+(1+\varepsilon)d_i\delta_i^{-1}] \\
&\quad -2(1+2\varepsilon)^2KL \\
&= 2(1+2\varepsilon)\sum_{1\le i\le K}\frac{q_i}{d_i}-2(1+2\varepsilon)^2KL \\
&\quad +2(1+2\varepsilon)(1+\varepsilon)\sum_{1\le i\le K}\delta_i^{-1}\ .
\end{aligned}$$

Since Phases 3 and 4 cannot increase the potential, the overall change in potential during the round is at most

$$-2\varepsilon\sum_{1\le i\le K}\frac{q_i}{d_i}+2(1+2\varepsilon)^2KL\ .$$

Thus, the potential decreases if

$$\sum_{1\le i\le K}\frac{q_i}{d_i}>(1+2\varepsilon)^2\varepsilon^{-1}KL\ .$$

Even if

$$\sum_{1\le i\le K}\frac{q_i}{d_i}\le(1+2\varepsilon)^2\varepsilon^{-1}KL,$$

however, there will still be a decrease in potential if any particular queue is large. In particular, assume that

$$\sum_{1\le i\le K}\frac{q_i}{d_i}\le(1+2\varepsilon)^2\varepsilon^{-1}KL$$

and that some queue for commodity j at node v_0 has height q_j^*. From the analysis of Phase 1, we know that the increase in potential due to new flow is at most

$$\begin{aligned}
&2(1+\varepsilon)(1+2\varepsilon)^2\varepsilon^{-1}KL+(1+\varepsilon)^2K \\
&\quad\le\ 2(1+2\varepsilon)^3\varepsilon^{-1}KL\ .
\end{aligned}$$

During Phase 2, we will get a decrease in potential that is at least as large as the potential drop that would result from a flow of magnitude $\frac{\varepsilon d_j}{M}$ from v_o to the sink for commodity j. (Notice that there is a path from v_0 to the sink with capacity at least $\frac{\varepsilon d_j}{M}$ since we can backtrack along the path from v_0 to the source for j and then from the source to the sink. This is the place where we use the fact that the underlying graph is undirected. We show how to overcome this assumption in Section 3.2.3.) Following the methods used previously, we can show that the decrease in potential due to the large queue is at least

$$\frac{2g[q_j^*-(N+L)g]}{d_j^2}$$

where $g=\frac{\varepsilon d_j}{M}$. Hence the potential will decrease during the round if

$$\frac{2\varepsilon d_j}{M}\left(q_j^*-\frac{2N\varepsilon d_j}{M}\right)d_j^{-2}>2(1+2\varepsilon)^3\varepsilon^{-1}KL\ .$$

Thus, the potential decreases if

$$\frac{q_j^*}{d_j}>(1+2\varepsilon)^3\varepsilon^{-2}KLM+\frac{2N\varepsilon}{M}\ . \qquad (7)$$

We have now shown that whenever there is a queue for commodity j with height q_j^* satisfying Equation 7, then the potential of the system drops during the round. (If $\sum_{1\le i\le k}\frac{q_i}{d_i}$ is large, we use the first argument. If it is small, we use the second argument.) If for each j, every queue for commodity j has height at most

$$\begin{aligned}
&\left[(1+2\varepsilon)^3\varepsilon^{-2}KLM+\frac{2N\varepsilon}{M}\right]d_j \\
&\quad\le\ (1+2\varepsilon)^4\varepsilon^{-2}KLMd_j
\end{aligned}$$

then the total potential will be at most

$$\Gamma_1 = 2MK\left[(1+2\varepsilon)^4\varepsilon^{-2}KLM\right]^2 .$$

Hence, if the potential exceeds Γ_1 at the beginning of a round, it will decrease during the round. Since the potential can increase by at most $2(1+2\varepsilon)^3\varepsilon^{-1}KL$ during any round (recall that if $\sum q_i/d_i \geq (1+2\varepsilon)^2\varepsilon^{-1}KL$ then the potential decreases), this means that the maximum potential in the system at any time is at most

$$\begin{aligned}\Gamma_{max} &\leq 2MK\left[(1+2\varepsilon)^4\varepsilon^{-2}KLM\right]^2 \\ &\quad +(1+2\varepsilon)^3\varepsilon^{-1}KL \\ &\leq 2MK\left[(1+3\varepsilon)^4\varepsilon^{-2}KLM\right]^2 .\end{aligned}$$

This means that the amount of commodity j in the system at any time is at most W_j where

$$2M\left(\frac{W_j}{2Md_j}\right)^2 \leq 2MK\left[(1+3\varepsilon)^4\varepsilon^{-2}KLM\right]^2 .$$

Hence,

$$\frac{W_j}{d_j} \leq 2(1+3\varepsilon)^4\varepsilon^{-2}K^{3/2}LM^2 .$$

In order for the amount of commodity j remaining in the network not to exceed an $\dfrac{\varepsilon}{1+\varepsilon}$ fraction of the total amount of commodity j that has been pumped into the network, we thus need to run the algorithm for at most

$$2(1+3\varepsilon)^4\varepsilon^{-3}K^{\frac{3}{2}}LM^2 = O(K^{\frac{3}{2}}LM^2\varepsilon^{-3})$$

rounds, as claimed. This means that we can find a feasible solution to the original static flow problem with demands $\{d_i\}$ in

$$O(K^{\frac{5}{2}}M^3L\varepsilon^{-3}\log K) \qquad (8)$$

steps.

3.2.3 Analysis for Directed Networks

The preceding algorithm also works for directed networks, although the running time degrades by an $O(\sqrt{M})$ factor and the proof of stability becomes more complicated. Below, we sketch the modifications that are necessary to prove the result for directed networks.

The main difficulty in the directed case is that flow can be deposited into nodes for which there is no path to the correct destination. Hence, the argument for the case when the potential is large because there is a large queue somewhere in the network becomes problematic. (For example, it may be the case that this queue cannot be decreased.)

In order to overcome this difficulty, we will partition the flow associated with each queue into two parts: an *included* part and an *excluded* part. The amount of flow that is excluded for each queue will be determined at the end of each round. In particular, we will exclude as much flow as possible, subject to the following constraints:

1. for $1 \leq i \leq K$, no flow is excluded for commodity i at the destination for i,
2. for all $e \in E$ and $1 \leq i \leq K$, the amount of flow excluded from the queue for commodity i at $tail(e)$ is no more than the the amount of flow excluded from the queue for commodity i at $head(e)$,
3. for all $v \in V$ and $1 \leq i \leq K$, the amount of flow excluded from queues for commodity i at node v is the same, and
4. the amount of flow excluded from any queue is no more than the total flow in the queue.

It is not difficult to show that the excluded flow for each queue is well-defined, and that it never decreases from round to round. It can also be shown that whenever the overall potential decreases as a result of flow being pushed across an edge, the included potential (i.e., the potential calculated based only on the included flow) decreases by at least as much. Moreover, the increase in potential due to the injection of new flow at the sources is no worse for included potential than it is for overall potential, since the two measures are the same for nodes (such as sources) for which there are paths to correct destinations. Hence, most all of the analysis that was developed for overall potential in the previous section applies equally well for included potential.

The only case which must be dealt with differently is the case of a large queue. In particular, we need to deal with the case when there is node with a queue Q for commodity j with included potential q^{**} and overall potential q^*, but for which every reachable queue for commodity j has overall potential exceeding $q^* - \dfrac{q^{**}}{2}$. In this case, however, we will be able to exclude $\frac{q^{**}}{2}$ potential from Q at the end of the round, thereby decreasing the included potential by a sufficient amount.

As a consequence, we can conclude that the maximum included potential in the system at any time is at most $O(\varepsilon^{-4}K^3L^2M^3)$. Since no potential is ever excluded from a source queue, this means that each source queue for commodity j never exceeds size $O(\varepsilon^{-2}K^{3/2}LM^{3/2}d_j)$, which means that no queue for commodity j anywhere in the network can ever exceed this size. Hence,

$$\frac{W_j}{d_j} \leq O(\varepsilon^{-2}K^{3/2}LM^{\frac{5}{2}}) ,$$

which means that the algorithm terminates in

$$O(K^{\frac{5}{2}}LM^{\frac{7}{2}}\varepsilon^{-3}\log K)$$

steps, as claimed.

4 Remarks & Open Questions

Almost surely, the running time bound for the algorithm can be substantially improved. The key to improvement is to find a better way of handling the case when $\sum_{1\leq i\leq K}\frac{q_i}{d_i}$ is small. Intuitively, we should be in great shape when the heights of queues at the sources are low. Yet this is precisely the scenario that causes us trouble. If we could assume that the heights at the sources never exceed the limit that causes the potential to drop during the round, then we might be able to show that the overall running time is $O(KLMN\log K)$, which is much better.

We also suspect that the algorithm can be made substantially more robust. For example, the techniques developed in Section 3.2.3 can be used to show that the algorithm works in distributed networks where the capacity of edges can diminish over time. (We simply must be sure to exclude more potential when this happens.) Whether or not the algorithm can be proved to work in a network where demands change over time or where edge capacities can decrease *and* increase over time remains an interesting open question (although we suspect that the algorithm will continue to work well in such environments). It would also be interesting to know if the algorithm works in completely asynchronous networks.

Finally, it would also be interesting to know if there is a variation of the algorithm that is useful in a continuous setting where there is no feasible flow. In particular, is there a natural local-control algorithm that is guaranteed to route as much flow as possible to correct destinations?

Acknowledgments

We are grateful to Andrew Goldberg, Serge Plotkin, Eva Tardos and Laszlo Lovasz for helpful discussions, and to Mark Tsimelzon and Bilal Khan for computer simulations of the algorithm.

References

[1] Yehuda Afek, Baruch Awerbuch, Eli Gafni, Yishay Mansour, Adi Rosen, and Nir Shavit. Polynomial end-to-end communication. unpublished manuscript, 1992.

[2] Yehuda Afek, Eli Gafni, and Adi Rosen. Slide - a technique for communication in unreliable networks. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 35–46, August 1992.

[3] William Aiello, Baruch Awerbuch, Bruce Maggs, and Satish Rao. Approximate load balancing on dynamic and synchronous networks. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 632–641, May 1993.

[4] Baruch Awerbuch, Yishay Mansour, and Nir Shavit. End-to-end communication with polynomial overhead. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 358–363, 1989.

[5] Jack B. Dennis. Distributed solution of network programming problems. In *IEEE Transactions of the Professional Technical Group on Communications Systems*, volume CS-12, number 2, pages 176–184, June 1964.

[6] A.V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[7] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl.*, 15:434–437, 1974.

[8] P. Klein, A. Agrawal, R. Ravi, and S. Rao. Approximation through multicommodity flow. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 726–727, 1990.

[9] F. T. Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommod ity flow problems with applications to approximation algorithms. In *29th Annual Symposium on Foundations of Computer Science, IEEE*, pages 422–431, 1988.

[10] T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problem. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 101–111, May 1991.

[11] Mark Tsimelzon. Bachelor thesis, MIT Lab. for Computer Science, 1993.

[12] P.M. Vaidya. Speeding up linear programming using fast matrix multiplication. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 332–337, 1989.