Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts

Philip Klein*
Computer Science Department
Brown University
Providence, RI

Serge Plotkin†
Department of Computer Science
Stanford University
Stanford, CA

Clifford Stein‡
Laboratory for Computer Science
MIT
Cambridge, MA

Éva Tardos§
School of Operations Research
Cornell University
Ithaca, NY

Abstract

In this paper, we describe new algorithms for approximately solving the concurrent multicommodity flow problem with uniform capacities. Our algorithms are much faster than previously known algorithms. Besides being an important problem in its own right, the uniform-capacity concurrent flow problem has many interesting applications. Leighton and Rao used uniform-capacity concurrent flow to find an approximately "sparsest cut" in a graph, and thereby approximately solve a wide variety of graph problems, including minimum feedback arc set, minimum cut linear arrangement, and minimum area layout. However, their method appeared to be impractical, as it required solving a large linear program. We show that their method might be practical by giving an $O(m^2 \log m)$ expected-time randomized algorithm for their concurrent flow problem on an m-edge graph. Raghavan and Thompson used uniform-capacity concurrent flow to approximately solve a channel width minimization problem in VLSI. We give an $O(k^{3/2}(m + n \log n))$ expected-time randomized algorithm and an $O(k \min\{n, k\}\,(m + n \log n) \log k)$ deterministic algorithm for this problem when the channel width is $\Omega(\log n)$, where k denotes the number of wires to be routed in an n-node, m-edge network.

*Research partially supported by ONR grant N00014-88-K-0243 and DARPA grant N00039-88-C0113 at Harvard University. Additional support provided by NSF Research Initiation Award CCR-901-2357, and by . the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA order 6320, amendment 1.

†Research supported by NSF Research Initiation Award CCR-900-8226, by U.S. Army Research Office Grant #DAAL-03-91-G-0102, and by ONR Contract N00014–88–K–0166.

‡Support provided by NSF PYI Award CCR-89-96272 with matching support from UPS and Sun and by an AT&T Bell Laboratories Graduate Fellowship.

§Research supported in part by a Packard Research Fellowship and by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through NSF grant DMS-8920550.

1 Introduction

The multicommodity flow problem involves shipping several different commodities from their respective sources to their sinks in a single network with the total amount of flow going through an edge limited by its capacity. The amount of each commodity we wish to ship is called the *demand* for that commodity. An optimization version of this problem is the *concurrent flow* problem in which the goal is to find the maximum percentage z such that at least z percent of each demand can be shipped without violating the capacity constraints. Here we consider the concurrent flow problem with unit capacities. Observe that in this case, the problem is equivalent to the problem of finding a flow (disregarding capacities) that minimizes the maximum total flow on any edge (the *congestion*). Let m, n, and k be, respectively, the number of edges, nodes, and commodities for the input network.

In this paper, we give algorithms that, for any positive ϵ, find a solution whose congestion is no more than $(1+\epsilon)$ times the minimum congestion. Our algorithms significantly improve the time required for finding such approximately optimal solutions.

One contribution of this paper is the introduction of a randomization technique useful in iterative approximation algorithms. This technique enables each iteration to be carried out much more quickly than by using known deterministic methods.

Part of our motivation in developing algorithms for concurrent flow derives from two important applications, finding sparsest cuts and finding a VLSI routing that minimizes channel width.

Leighton and Rao [11] showed how to use the solution to a unit-capacity concurrent flow problem to find an approximate "sparsest cut" of a graph. As a consequence, they and other researchers have developed polylog-times-optimal approximation algorithms for a wide variety of graph problems, including minimum area VLSI layout, minimum cut linear arrangement, minimum feedback arc set [11], optimal linear and higher-dimensional arrangement [5], minimum chordal fill [7], and single-processor scheduling [14].

The computational bottleneck of the method of Leighton and Rao is solving a unit-capacity concurrent flow problem with $O(n)$ commodities, each with unit demand. They appealed to linear programming techniques to show that the problem can be solved in polynomial time. The new approximation algorithm greatly improves the resulting running time.

Theorem 1.1 For any fixed $\epsilon > 0$, a $(1+\epsilon)$-factor approximation to the unit-capacity, unit-demand concurrent flow problem can be found by a randomized algorithm in $O((k+m)m\log m)$ expected time, where the constant depends on ϵ.

As an application of this result we substantially reduce the time required for Leighton and Rao's method.

Theorem 1.2 An $O(\log n)$-factor approximation to the sparsest cut in a graph can be found by a randomized algorithm in expected $O(m^2 \log m)$ time.

The previous best running time of $O(n^{4.5}\sqrt{m}\log n)$ [18], is obtained by using linear programming techniques and fast matrix multiplication.

Another application of our approximation algorithm is to VLSI routing in graphs. Raghavan and Thompson [13] and Raghavan [12] considered the problem of routing two-terminal nets (essentially wires) in a graph so as to approximately minimize the *channel width*, *i.e.*, the maximum number of nets routed through an edge. The computational bottleneck in their algorithms is solving a unit-capacity concurrent flow problem. Their algorithms require a better than constant ϵ approximation to the concurrent flow problem. In fact, the algorithm of Theorem 1.1 is a *fully polynomial approximation* algorithm, *i.e.* its running time depends polynomially on ϵ^{-1}.

Algorithm type	Running Time
Randomized, fixed ϵ	$O(m(k+m)\log n)$
Deterministic, fixed ϵ	$O(mk(k+m)\log n)$
Randomized, $1 < \epsilon < \frac{1}{\mathrm{poly}(n)}$	$O(\epsilon^{-3}m(k+m)\log^2 n)$
Deterministic, $1 < \epsilon < \frac{1}{\mathrm{poly}(n)}$	$O(\epsilon^{-2}mk(k+m)\log^2 n)$

Table 1: Upper bounds on the running times of our algorithms. The actual bounds are slightly better.

Theorem 1.3 For any positive $\epsilon < 1$ that is at least inverse polynomial in n, a $(1+\epsilon)$-factor approximation to the unit-capacity concurrent flow problem can be found by a randomized algorithm in expected time $O((\epsilon^{-1}k + \epsilon^{-3}m)(m\log n + n\log^2 m))$ and by a deterministic algorithm in time $O((k+\epsilon^{-2}m)k(m\log m + n\log^2 m))$ time.

An application of the algorithm of Theorem 1.3 is a significant improvement in the time needed to solve Raghavan and Thompson's problem.

Theorem 1.4 If $w_{\min}$ denotes the minimum achievable channel width and $w_{\min} = \Omega(\log m)$, a routing of width $w_{min} + O(\sqrt{w_{min}\log n})$ can be found by a randomized algorithm in expected time $O(k^{3/2}(m + n\log n))$, and by a deterministic algorithm in time $O(k\min\{n,k\}(m+n\log n)\log k)$.

Our algorithms compare favorably to previous work. The concurrent flow problem can be formulated as a linear program in $O(mk)$ variables and $O(m+nk)$ constraints (see, for example [15]). Linear programming can be used to solve the problem optimally in polynomial time. Kapoor and Vaidya [6] gave a method to speed up the matrix inversions involved in Karmarkar type algorithms for multicommodity flow problems; combining their technique with Vaidya's new linear programming algorithm using fast matrix multiplication [18] yields a time bound of $O(k^{3.5}n^3\sqrt{m}\log(nD))$ for the unit-capacity concurrent flow problem with integer demands (where D denotes the sum of the demands) and an $O(\sqrt{m}k^{2.5}n^2\log(n\epsilon^{-1}D))$ bound for the approximation problem.

Shahrokhi and Matula [15] gave a combinatorial fully polynomial approximation scheme for the unit-capacity concurrent flow problem (which they called the concurrent flow problem with uniform capacities). Their algorithm runs in $O(nm^7\epsilon^{-5})$ time.

Our approach to solving concurrent flow problems is a modification of the framework originated by Shahrokhi and Matula [15]. The idea is to use a length function on the edges to reflect congestion, and iteratively reroute flow from long (more congested) paths to short (less congested) paths. Our approach differs from that of Shahrokhi and Matula in several ways. We develop a framework of *relaxed optimality conditions* that allows us to measure the congestion on both a local and a global level, thereby giving us more freedom in choosing which flow paths to reroute at each iteration. We exploit this freedom by using a faster *randomized* method for choosing flow paths. In addition, this framework also allows us to achieve greater improvement as a result of each rerouting. In Table 1, we give upper bounds on the running times for our algorithms. Our actual bounds are slightly better than those in the table, and are given in more detail in the remainder of the paper. Note that by use of various combinations of our techniques, we can obtain slightly better bounds than those stated in Theorems 1.1 and 1.3.

An earlier version of this paper has appeared in [9]. In the earlier version the case when both the capacities and the demands are uniform was considered separately from the more general case when only the capacities are assumed to be uniform. The earlier version presented a fast algorithm for the first case, and a factor of $\epsilon^{-1}m$ slower one for the more general case. In this version we extend the algorithm for the uniform demand case to work for the more general case with at most a logarithmic slowdown.

2 Preliminaries and Definitions

In this section we define the concurrent flow problem, introduce our notation, and give some basic facts regarding the problem. Concurrent flow is a variant of multicommodity flow, and we start by giving a formal definition of the latter.

The multicommodity flow problem is the problem of shipping several different commodities from their respective sources to their sinks in a single network, while obeying capacity constraints. More precisely, an instance of the multicommodity flow problem consists of an undirected graph $G = (V, E)$, a non-negative capacity $cap(vw)$ for every edge $vw \in E$, and a specification of k commodities, numbered 1 through k. The specification for commodity i consists of a source-sink pair $s_i, t_i \in V$ and a non-negative integer demand $d(i)$. We will denote the maximum demand by $d_{\max}$, the total demand $\sum_i d(i)$ by D, the number of nodes by n, the number of edges by m, and the number of different sources by k^*. Notice that $k^* \leq n$. For notational convenience we assume that $m \geq n$, and that the graph G has no parallel edges. If there is an edge between nodes v and w, this edge is unique by assumption, and we denote it by vw. Note that vw and wv denote the same edge.

A *flow* f_i in G from node s_i to node t_i can be defined as a collection of paths from s_i to t_i, with associated real values. Let $\mathcal{P}_i$ denote a collection of paths from s_i to t_i in G, and let $f_i(P)$ be a nonnegative value for every P in $\mathcal{P}_i$. The *value* of the flow thus defined is $\sum_{P \in \mathcal{P}_i} f_i(P)$, which is the total flow delivered from s_i to t_i. The amount of flow through an edge vw is

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

A *feasible multicommodity flow* f in G consists of a flow f_i from s_i to t_i of value $d(i)$ for each commodity $1 \leq i \leq k$. We require that $f(vw) \leq cap(vw)$ for every edge $vw \in E$, where we use $f(vw) = \sum_{i=1}^{k} f_i(vw)$ to denote the total amount of flow on the edge vw.

We consider the optimization version of the multicommodity flow problem, called the *concurrent flow problem*, and first defined by Shahrokhi and Matula [15]. In this problem the objective is to compute the maximum possible value z such that there is a feasible multicommodity flow with demands $z \cdot d(i)$ for every $1 \leq i \leq k$. We call z the *throughput* of the multicommodity flow. An equivalent formulation of the concurrent flow problem is to compute the maximum z such that there is a feasible flow with demands $d(i)$ and capacities $cap(vw)/z$.

In this paper we shall focus exclusively on the special case of *unit capacities*, in which all edge-capacities are equal. The problem of finding a maximum throughput z can be reformulated in this special case as follows: ignore capacities, and find a multicommodity flow f that satisfies the demands and minimizes $|f| = \max_{vw \in E} \{f(vw)\}$, the maximum total flow on any edge.

A multicommodity flow f satisfying the demands $d(i)$ is *ϵ-optimal* if $|f|$ is at most a factor $(1+\epsilon)$ more than the minimum possible $|f|$. The *approximation problem* associated with the unit-capacity concurrent flow problem is to find an ϵ-optimal multicommodity flow f. We shall assume implicitly throughout that ϵ is at least inverse polynomial in n and at most $1/10$. These assumptions are not very restrictive as they cover practically every case of interest. To find an ϵ-optimal flow where $\epsilon \geq 1/10$, one can just find a $1/10$-optimal flow. To find an ϵ-optimal flow when $1/\epsilon$ is greater than any polynomial in n, one can run our algorithm. It will work for arbitrarily small ϵ, however, the running time will be slower than the time bounds given, as we will need to manipulate numbers whose size is exponential in the input. However, if this amount of accuracy is desired, it is more sensible and efficient to use any polynomial time linear programming algorithm to solve the problem exactly.

One can define the analogous problem for directed graphs. Our algorithms, and the corresponding time bounds, easily extend to the directed case by replacing (undirected) edges by (directed) arcs and paths by directed paths. Henceforth, we will concentrate only on the undirected case.

Linear programming duality gives a characterization of the optimum solution to the concurrent flow problem. Let $\ell : E \to \mathbf{R}$ be a nonnegative *length* function. For nodes $v, w \in V$ let $dist_\ell(v, w)$ denote the length of the shortest path from v to w in G with respect to the length function ℓ. For a path P we shall use $\ell(P)$ to denote the length of P. We shall use $|\ell|_1$ to denote $\sum_{vw \in E} \ell(vw)$, the sum of the length of the edges. The following theorem is a special case of the linear programming duality theorem (see, for example, [15]).

Theorem 2.1 For a multicommodity flow f satisfying the demands $d(i)$ and a length function ℓ,

$$|f||\ell|_1 \geq \sum_{vw \in E} f(vw)\ell(vw) = \sum_{i=1}^{k} \sum_{vw \in E} \ell(vw) f_i(vw) = \sum_{i=1}^{k} \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P) \geq \sum_{i=1}^{k} dist_\ell(s_i, t_i) d(i). \quad (1)$$

Furthermore, a multicommodity flow f minimizes $|f|$ if and only if there exists a nonzero length function ℓ for which all of the above terms are equal.

The optimality (complimentary slackness) conditions given by linear programming can be reformulated in terms of conditions on edges and paths.

Theorem 2.2 A multicommodity flow f has minimum $|f|$ if and only if there exists a nonzero length function ℓ such that

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

The goal of our algorithms is to solve the approximation problem, i.e. to find a multicommodity flow f and a length function ℓ such that the largest term, $|f||\ell|_1$, in (1) is within a $(1 + \epsilon)$ factor of the smallest term, $\sum_i dist_\ell(s_i, t_i)d(i)$. In this case, we say that f and ℓ are ϵ-optimal. Note that if f and ℓ are ϵ-optimal then clearly f is ϵ-optimal. In fact, a multicommodity flow f is ϵ-optimal if and only if there exists a length function ℓ such that f and ℓ are ϵ-optimal.

3 Relaxed optimality conditions

Theorems 2.1 and 2.2 give two (apparently) different characterizations of exact optimality. Our goal is to find a flow that satisfies a relaxed version of Theorem 2.1. In order to do so, we will introduce a relaxed version of Theorem 2.2, the complimentary slackness conditions of linear programming. We will then show that these *relaxed optimality conditions* are sufficient to show that the first and last terms in (1) are within a $(1 + \epsilon)$ factor, and hence the flow f is ϵ-optimal. Our notion of relaxed optimality is analogous to the notion of ϵ-optimality used by Goldberg and Tarjan in the context of the minimum-cost flow problem [4].

Let $\epsilon > 0$ be an error parameter, f a multicommodity flow and ℓ a length function. Throughout this section we shall use ϵ' to denote $\frac{\epsilon}{7}$. We say that a path $P \in \mathcal{P}_i$ for a commodity i is *ϵ-good* if

$$\ell(P) - dist_\ell(s_i, t_i) \leq \epsilon' \ell(P) + \epsilon' \frac{|f|}{\min\{D, kd(i)\}} |\ell|_1$$

and ϵ-*bad* otherwise. The intuition is that a flow path is ϵ-good if it is short in either a relative or an absolute sense, *i.e.* it is either almost as short as the shortest possible (s_i, t_i)-path or it is at most a small fraction of $|\ell|_1$. We use this notion in defining the following *relaxed optimality conditions* (with respect to a flow f, a length function ℓ and an error parameter ϵ):

R1) For every edge $vw \in E$ either $\ell(vw) \leq \frac{\epsilon'}{m}|\ell|_1$ or $f(vw) \geq |f|/(1+\epsilon')$.

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

The first condition says that every edge either has a length which is a small fraction of the sum of the lengths of all edges or is almost saturated. The second condition says that the amount of flow which is on ϵ-bad paths, i.e. long paths, contributes a small fraction of the sum $f \cdot \ell$.

The next two lemmas show that the relaxed optimality conditions are sufficient to imply ϵ-optimality. We will first show that Condition R1 implies that the first two terms in (1) are close. Then we will show that the two conditions together imply that the first and last terms in (1) are close. Thus we can conclude that the relaxed optimality conditions are sufficient to imply ϵ-optimality.

Lemma 3.1 Suppose a multicommodity flow f and a length function ℓ satisfy relaxed optimality condition $R1$. Then

$$(1-\epsilon')|f||\ell|_1 \leq (1+\epsilon') \sum_{vw} f(vw)\ell(vw). \tag{2}$$

Proof: We estimate $|f||\ell|_1 = \sum_{vw} |f|\ell(vw)$ in two parts. The first part is the sum of the terms contributed by edges that satisfy $|f| \leq (1+\epsilon')f(vw)$. This part of the sum is clearly at most $(1+\epsilon')\sum_{vw} f(vw)\ell(vw)$. If vw is an edge whose contribution is not counted in the first part then, by assumption, $\ell(vw) \leq \frac{\epsilon'}{m}|\ell|_1$. Therefore, the sum of all other terms is at most $\epsilon'|f||\ell|_1$. Thus, $|f||\ell|_1 \leq (1+\epsilon')\sum_{vw} f(vw)\ell(vw) + \epsilon'|f||\ell|_1$. This implies the lemma. ∎

Theorem 3.2 Suppose f and ℓ and ϵ satisfy the Relaxed Optimality Conditions R1 and R2. Then f is ϵ-optimal, *i.e.* $|f|$ is at most a factor $(1+\epsilon)$ more than the minimum possible.

Proof: We need to estimate the ratio of the terms in inequality (1) of Theorem 2.1. Lemma 3.1 estimates the ratio of the first two terms. We shall use this in estimating the ratio of the first and the last term.

Consider the penultimate term in (1). We break this sum, $\sum_i \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$, into two parts; the sum over ϵ-good paths and the sum over ϵ-bad paths. Relaxed optimality condition R2 gives us an upper bound of $\epsilon' \sum_i \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$ on the sum over the ϵ-bad paths, and the definition of an ϵ-good path gives us the following bound on the sum over the ϵ-good paths:

$$\sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_i \\ P\ \epsilon\text{-good}}} f_i(P)\ell(P) \leq (1-\epsilon')^{-1} \sum_i \sum_{P \in \mathcal{P}_i} (dist_\ell(s_i,t_i) f_i(P) + \epsilon' \frac{|f|}{\min\{D, kd(i)\}} |\ell|_1 f_i(P)).$$

Observing that $(\min\{D, kd(i)\})^{-1} \leq D^{-1} + (kd(i))^{-1}$ and $\sum_{P \in \mathcal{P}_i} f_i(P) = d(i)$, we can bound the sum over the ϵ-good paths by

$$\sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_i \\ P\ \epsilon\text{-good}}} f_i(P)\ell(P) \quad \leq \quad (1-\epsilon')^{-1} \left(\sum_i dist_\ell(s_i,t_i) d(i) + \sum_i \epsilon' |f||\ell|_1 d(i) \left(\frac{1}{D} + \frac{1}{kd(i)} \right) \right)$$

$$= \quad (1-\epsilon')^{-1}\left(\sum_i dist_\ell(s_i,t_i)d(i) + \sum_i \epsilon'|f||\ell|_1\left(\frac{d(i)}{D}+\frac{1}{k}\right)\right)$$

Now observe that there are exactly k commodities and $\sum_i d(i) = D$, so the last term sums to exactly $2\epsilon'|f||\ell|_1$. This gives that

$$\sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_i \\ P\ \epsilon\text{-good}}} f_i(P)\ell(P) \leq (1-\epsilon')^{-1}\left(\sum_i dist_\ell(s_i,t_i)d(i) + 2\epsilon'|f||\ell|_1\right).$$

Combining the bounds on the sum over ϵ-bad and ϵ-good paths we get

$$\sum_i \sum_{P\in\mathcal{P}_i} \ell(P)f_i(P) \leq (1-\epsilon')^{-1}\sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_i \\ P\ \epsilon\text{-good}}} f_i(P)\ell(P) \leq (1-\epsilon')^{-2}\sum_i dist_\ell(s_i,t_i)d(i) + 2(1-\epsilon')^{-2}\epsilon'|f||\ell|_1.$$

By the middle equations in Theorem 2.1, $\sum_i \sum_{P\in\mathcal{P}_i} \ell(P)f_i(P)$ is equal to $\sum_{vw\in E} f(vw)\ell(vw)$. Lemma 3.1 gives a bound on $\sum_{vw\in E} f(vw)\ell(vw)$ in terms on $|f||\ell|_1$. Combining these inequalities and rearranging terms we get

$$\left(\frac{(1-\epsilon')^2}{1+\epsilon'} - \frac{2\epsilon'}{1-\epsilon'}\right)|f||\ell|_1 \leq \frac{1}{1-\epsilon'}\sum_i dist_\ell(s_i,t_i)d(i).$$

Combining the fractions and dropping low order terms we get that

$$|f||\ell|_1 \leq \frac{1+\epsilon'}{1-5\epsilon'}\sum_i dist_\ell(s_i,t_i)d(i).$$

The assumption that $\epsilon \leq 1/10$ implies that $\epsilon' \leq 1/70$, which in turn implies that the factor $\frac{1+\epsilon'}{1-5\epsilon'}$ is less than $(1+7\epsilon') = (1+\epsilon)$. We combine this bound with inequality (1) to complete the proof. ∎

In the next two sections, we will focus on algorithms which achieve the relaxed optimality conditions.

4 Generic rerouting

In this section, we describe the procedure Reduce that is the core of our approximation algorithms, and prove bounds on its running time. Given a multicommodity flow f, procedure Reduce modifies f until either f becomes ϵ-optimal or $|f|$ is reduced below a given target value. The approximation algorithms presented in the next two sections repeatedly call procedure Reduce to decrease $|f|$ by a factor of 2, until an ϵ-optimal solution is found.

The basic step in our algorithms is choosing a flow path and rerouting some flow from this path to a "better" path. This step closely resembles the basic step in the algorithm of Shahrokhi and Matula [15]. The main differences are in the way we choose the paths and in the amount of flow that is rerouted at each iteration.

The key idea is to measure how good the current flow is by using the notion of ϵ-optimality, described in the previous section. Given a flow f and a value α to be determined later, we use a length function defined by $\ell(vw) = e^{\alpha f(vw)}$, which reflects the congestion of the edge vw. In other words, the length of an edge depends on the flow carried by the edge. Given an input ϵ, our algorithms

REDUCE($f, \tau, \epsilon, \sigma_i$ for $i = 1, \ldots, k$)
$\alpha \leftarrow (7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7m\epsilon^{-1})$.
While $|f| \geq \tau$ **and** f and ℓ are not ϵ-optimal,
  For each edge vw, $\ell(vw) \leftarrow e^{\alpha f(vw)}$.
  Call FINDPATH(f, ℓ, ϵ) to find an ϵ-bad flow path P and a short path Q with the same endpoints as P.
  Reroute σ_i units of flow from P to Q.
Return f.

Figure 1: Procedure Reduce.

gradually update f until f and ℓ (defined by the above formula) become ϵ-optimal. Each update is done by choosing an ϵ-bad flow path, rerouting some flow from this path to a much shorter path (with respect to ℓ), and recomputing the length function. We will prove below that the parameter α in the definition of length can be selected so that relaxed optimality condition *R1* is always satisfied. Through iterative reroutings of flow, we gradually enforce relaxed optimality condition *R2*. When both relaxed optimality conditions are satisfied then Theorem 3.2 can be used to infer that f is ϵ-optimal.

For simplicity iof presentation, we shall assume for now that the value of the length function $\ell(vw) = e^{\alpha f(vw)}$ at an edge vw can be computed in one step from $f(vw)$ and represented in a single computer word. In Section 4.3 we will remove this assumption and sho that it is sufficient to compute an approximation to this value, and show that the time required for computing a sufficiently good approximation does not change the asymptotic running times of our algorithms.

Procedure REDUCE (see Figure 1), takes as input a multicommodity flow f, a target value τ, an error parameter ϵ, and a flow quantum σ_i for each commodity i. We require that each flow path comprising f_i carries flow that is an integer multiple of σ_i. The procedure repeatedly reroutes σ_i units of flow from an ϵ-bad path of commodity i to a shortest path. We will need a technical *granularity condition* that σ_i is small enough for every i to guarantee that approximate optimality is achievable through such reroutings. In particular, we assume that upon invocation of REDUCE, for every commodity i we have

$$\sigma_i \leq \epsilon^2 \frac{\tau}{102\log(7m\epsilon^{-1})} \tag{3}$$

Upon termination, the procedure outputs an improved multicommodity flow f such that either $|f|$ is less than the target value τ or f is ϵ-optimal. (Recall that we have assumed that $\epsilon \leq 1/10$.)

In the remainder of this section, we analyze the procedure REDUCE shown in Figure 1. First, we show that throughout REDUCE f and ℓ satisfy relaxed optimality condition *R1*. Second, we show that if the granularity condition is satisfied, the number of iterations in REDUCE is small. Third, we give an even smaller bound on the number of iterations for the case in which the flow f is $O(\epsilon)$-optimal upon invocation of REDUCE. This bound will be used in Section 5 to analyze an ϵ-scaling algorithm presented there. Fourth, we describe efficient implementations of procedure FINDPATH.

4.1 Bounding the number of iterations of REDUCE

Lemma 4.1 If f is a multicommodity flow, and $\alpha \geq (7+\epsilon)|f|^{-1}\epsilon^{-1}\log(7m\epsilon^{-1})$, then the multicommodity flow f and the length function $\ell(vw) = e^{\alpha f(vw)}$ satisfy relaxed optimality condition *R1*.

Proof: Assume $|f| - f(v,w) \geq \frac{\epsilon}{7} f(v,w)$ for an edge $vw \in E$ and let ϵ' denote $\frac{\epsilon}{7}$. Observe that

$|\ell|_1 \geq e^{\alpha|f|}$. Hence, we have

$$\frac{|\ell|_1}{\ell(v,w)} \geq \frac{e^{\alpha|f|}}{e^{\alpha f(v,w)}} \geq \frac{e^{\alpha|f|}}{e^{\alpha|f|(1+\epsilon')^{-1}}} = e^{\alpha|f|\epsilon'(1+\epsilon')^{-1}}.$$

We can use the bound on α in the statement of the lemma to conclude that this last term is at least $\frac{7m}{\epsilon}$. Thus, $\ell(vw) \leq \frac{\epsilon}{7m}|\ell|_1$. ▮

At the beginning of REDUCE, α is set equal to $(7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7m\epsilon^{-1})$. As long as $|f| \geq \tau$, the value of α is sufficiently large, so by Lemma 4.1, relaxed optimality condition *R1* is satisfied. If we are lucky and relaxed optimality condition *R2* is also satisfied, then it follows that f and ℓ are ϵ-optimal. Now we show that if *R2* is not satisfied, then we can make significant progress. Like Shahrokhi and Matula, we use $|\ell|_1$ as a measure of progress.

Lemma 4.2 Suppose σ_i and τ satisfy the granularity condition. Then rerouting σ_i units of flow from an ϵ-bad path of commodity i to the shortest path with the same endpoints decreases $|\ell|_1$ by $\Omega(\frac{\sigma_i}{\min\{D,kd(i)\}}|\ell|_1 \log m)$.

Proof: Let P be an ϵ-bad path from s_i to t_i, and let Q be a shortest (s_i,t_i)-path. Let $A = P - Q$, and $B = Q - P$. The only edges whose length changes due to the rerouting are those in $A \cup B$. The decrease in $|\ell|_1$ is $\ell(A) + \ell(B) - e^{-\alpha\sigma_i}\ell(A) - e^{\alpha\sigma_i}\ell(B)$, which can also be written as

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

The granularity condition, the definition of α, and the assumption that $\epsilon \leq 1/10$, imply that $\alpha\sigma_i \leq \frac{7+\epsilon}{102}\epsilon \leq \epsilon/7 \leq 1/70$. For $0 \leq x \leq \frac{1}{70}$, we have $e^x \geq 1+x$, $e^x \leq 1 + \frac{141}{140}x$, and $e^{-x} \leq 1 - \frac{139}{140}x$. Thus the decrease is at least

$$\frac{139}{140}\alpha\sigma_i\left(\ell(A)-\ell(B)\right) - (\alpha\sigma_i)\left(\frac{141}{140}\alpha\sigma_i\right)\ell(B).$$

Now, observe that $\ell(A)-\ell(B)$ is the same as $\ell(P)-\ell(Q)$, and $\ell(Q) = dist_\ell(s_i,t_i)$. Also $\ell(B) \leq \ell(P)$. This gives a lower bound of

$$\frac{139}{140}\alpha\sigma_i\left(\ell(P) - dist_\ell(s_i,t_i)\right) - \left(\frac{141}{140}\alpha^2\sigma_i^2\right)\ell(P).$$

But P is ϵ-bad, so this must be at least

$$\begin{aligned} & \frac{139}{140}\alpha\sigma_i(\epsilon'\ell(P) + \epsilon'\frac{|f|}{\min\{D,kd(i)\}}|\ell|_1) - \frac{141}{140}\alpha^2\sigma_i^2\ell(P) \\ = \; & \frac{139}{140}\alpha\sigma_i\epsilon'\ell(P) - \frac{141}{140}\alpha^2\sigma_i^2\ell(P) + \frac{139}{140}\alpha\epsilon'|f|\frac{\sigma_i}{\min\{D,kd(i)\}}|\ell|_1. \end{aligned}$$

We have seen that $\frac{7+\epsilon}{102}\epsilon \geq \alpha\sigma_i$, which implies that $139\epsilon' \geq 141\alpha\sigma_i$ and therefore the first term dominates the second term. Thus the third term gives a lower bound on the decrease in $|\ell|_1$. Substituting the value of α and using the fact that during execution of REDUCE we have $\tau \leq |f|$, yields the claim of the lemma. ▮

The following theorem bounds the number of iterations in REDUCE.

Theorem 4.3 If, for every commodity i, τ and σ_i satisfy the granularity condition and $|f| = O(\tau)$ initially then the procedure REDUCE terminates after $O(\epsilon^{-1}\max_i \frac{\min\{D,kd(i)\}}{\sigma_i})$ iterations.

Proof: Theorem 3.2 implies that if f and ℓ satisfy the relaxed optimality conditions then they are ϵ-optimal. By Lemma 4.1, relaxed optimality condition *R1* is maintained throughout all iterations. The fact that f is not yet ϵ-optimal implies that condition *R2* is not yet satisfied. Hence there exists an ϵ-bad path for FINDPATH to find. A single rerouting of flow from an ϵ-bad path of commodity i to a shortest path results in a reduction in $|\ell|_1$ of at least $\Omega(\frac{\sigma_i}{\min\{D,kd(i)\}}|\ell|_1 \log(m\epsilon^{-1}))$. Since $1-x < e^{-x}$, it follows that every $O(\max_i \frac{\min\{D,kd(i)\}}{\sigma_i} \log^{-1}(m\epsilon^{-1}))$ iterations reduce $|\ell|_1$ by at least a constant factor.

Next we bound the number of times $|\ell|_1$ can be reduced by a constant factor. Let f' denote the input multicommodity flow. For every edge vw, $f'(vw) \le |f'|$. Hence after we first assign lengths to edges, the value of $|\ell|_1$ is at most $me^{\alpha|f'|}$. The length of every edge remains at least 1 so $|\ell|_1$ is always at least m. Therefore, $|\ell|_1$ can be reduced by a factor of e at most $\alpha|f'|$ times, which is $O(\epsilon^{-1}\log(m\epsilon^{-1}))$ by the assumption that $f = O(\tau)$ and the value of α. This proves that REDUCE terminated in the claimed number of iterations. ∎

Theorem 4.4 Suppose that the input flow f is $O(\epsilon)$-optimal, σ and τ satisfy the granularity condition, and $|f| = O(\tau)$ initially. Then the procedure REDUCE terminates after $O(\max_i \frac{\min\{D,kd(i)\}}{\sigma_i})$ iterations.

Proof: Again let f' denote the input multicommodity flow. The assumption that f' is $O(\epsilon)$-optimal implies that $|f'| \le (1+O(\epsilon))|f|$ for *every* multicommodity flow f. Therefore, the value of $|\ell|_1$ is never less than $e^{(1+O(\epsilon))^{-1}\alpha|f'|}$. As in Theorem 4.3, the initial value of $|\ell|_1$ is at most $me^{\alpha|f'|}$, so the number of times $|\ell|_1$ can be reduced by a constant factor is $O(\alpha\epsilon|f'| + \log m)$, which is $O(\alpha\epsilon|f'|)$ by the choice of α and τ. The theorem then follows as in the proof of Theorem 4.3. ∎

4.2 Implementing an Iteration of Reduce

We have shown that REDUCE terminates after a small number of iterations. It remains to show that each iteration can be carried out quickly. REDUCE consists of three steps—computing lengths, executing FINDPATH and rerouting flow. We discuss computing lengths in Section 4.3. In this section, we discuss the other two steps.

We now consider the time taken by procedure FINDPATH. We will give three implementations of this procedure. First, we will give a simple deterministic implementation that runs in $O(k^*(m + n\log n) + n\sum_i \frac{d(i)}{\sigma_i})$ time, then a more sophisticated implementation that runs in time $O(k^* n \log n + m(\log n + \min\{k, k^* \log d_{\max}\}))$, and finally a randomized implementation that runs in expected $O(\epsilon^{-1}(m + n\log n))$ time. All of these algorithms use the shortest-paths algorithm of Fredman and Tarjan [3] that runs in $O(m + n\log n)$ time.

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

To decrease the time required for FINDPATH we have to find an ϵ-bad path, if one exists, without computing the length of so many paths. Observe that if there is an ϵ-bad flow path for commodity i then the longest flow path for commodity i must be ϵ-bad. Thus, instead of looking for an ϵ-bad path in $\mathcal{P}_i$ for some commodity i, it suffices to find an ϵ-bad path in the directed graph obtained by taking all flow paths in $\mathcal{P}_i$, and treating the paths as directed away from s_i. In order to see if there is an ϵ-bad path we need to compute the length of the longest path from s_i to t_i in this directed graph. To facilitate this computation we shall maintain that the directed flow graph is acyclic.

Let G_i denote the flow graph of commodity i. If G_i is acyclic, an $O(m)$ time dynamic programming computation suffices to compute the longest paths from s_i to every other node. Suppose that in an iteration we reroute flow from an ϵ-bad path from s_i to t_i, in the flow graph G_i. We must first update the flow graph G_i to reflect this change. Second, the update might introduce directed cycles in G_i, so we must eliminate such cycles of flow. We use an algorithm due to Sleator and Tarjan [16] to implement this process. Sleator and Tarjan gave a simple $O(nm)$ algorithm and a more sophisticated $O(m \log n)$ algorithm for the problem of converting an arbitrary flow into an acyclic flow.

Note that eliminating cycles only decreases the flows on edges, so it cannot increase $|\ell|_1$. Thus our bound on the number of iterations in REDUCE still holds.

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

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

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

Lemma 4.5 The total time required for deterministically implementing an iteration of REDUCE (assuming that exponentiation is a single step) is $O(k^* n \log n + m(\log n + \min\{k, k^* \log d_{\max}\}))$.

Next, we give a randomized implementation of FINDPATH that is much faster when ϵ is not too small; this implementation seems simple enough to be practical. If f and ℓ are not ϵ-optimal, then relaxed optimality condition *R2* is not satisfied, and thus ϵ-bad paths contribute at least an $\frac{\epsilon}{7}$-fraction of the total sum $\sum_i \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$. Therefore, by randomly choosing a flow path P with probability proportional to its contribution to the above sum, we have at least an $\frac{\epsilon}{7}$ chance of selecting an ϵ-bad path. Furthermore, we will show that we can select a candidate ϵ-bad path according to the right probability in $O(m)$ time. Then we can compute a shortest path with the same endpoints in $O(m + n \log n)$ time. This enables us to determine whether or not P was an ϵ-bad path. Thus we can implement FINDPATH in $O(\epsilon^{-1}(m + n \log n))$ expected time.

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

Lemma 4.6 If we choose an edge vw with probability proportional to $\ell(vw) f(vw)$, and then we select a flow path among paths through this edge vw with probability proportional to the value of the flow carried on the path, then the probability that we have selected a given flow path P is proportional to its contribution to the sum $\sum_i \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$.

Proof: Let $B = \sum_i \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$. Select an edge vw with probability $f(vw)\ell(vw)/B$. Once an edge vw is selected, choose a path $P \in \mathcal{P}_i$ through edge vw with probability $\frac{f_i(P)}{f(vw)}$. Consider a commodity i and a path $P \in \mathcal{P}_i$.

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

∎

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

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

We have shown that if relaxed optimality condition *R2* is not satisfied, then, with probability of at least $\epsilon/7$, we can find an ϵ-bad path in $O(m + n\log n)$ time. FINDPATH continues to pick paths until either an ϵ-bad path is found or $7/\epsilon$ trials are made. Observe that given that f and ℓ are not yet ϵ-optimal (which implies that condition *R2* is not yet satisfied), the probability of failure to find an ϵ-bad path in $7/\epsilon$ trials is bounded by $1/e$. Thus, in this case, REDUCE can terminate, claiming that f and ℓ are ϵ-optimal with probability of at least $1 - 1/e$. Computing lengths and updating flows can each be done in $O(n \log n)$ time, thus we get the following bound:

Lemma 4.7 One iteration of REDUCE can be implemented randomly in time $(\epsilon^{-1}(m + n\log n))$ time (assuming that exponentiation is a single step).

The randomized algorithm as it stands is *Monte Carlo*; there is a non-zero probability that REDUCE erroneously claims to terminate with an ϵ-optimal f. To make the algorithm *Las Vegas* (never wrong, sometimes slow), we introduce a deterministic check. If FINDPATH fails to find an ϵ-bad path, REDUCE computes the sum $\sum_i dist_\ell(s_i, t_i)d(i)$ to the required precision and compares it with $|f||\ell|_1$ to determine whether f and ℓ are really ϵ-optimal. If not, the loop resumes. The time required to compute the sum is $O(k^*(m + n\log n))$, because at most k^* single-source shortest path computations are required. The probability that the check must be done t times in a single call to REDUCE is at most $(e^{-1})^{t-1}$, so the total expected contribution to the running time of REDUCE is at most $O(k^*(m + n\log n))$.

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

We conclude this section with a theorem summarizing the running time of REDUCE for some cases of particular interest. For all of these bounds the running time is computed by multiplying the appropriate time for an iteration of REDUCE by the appropriate number of iterations of REDUCE. These bounds depend on the assumption that exponentiation is a single step. In Subsection 4.3 we shall show that the same bounds can be achieved without this assumption. We shall also give a more efficient implementation for the case when ϵ is a constant.

Theorem 4.8 Let $f = O(\tau)$ and τ and σ_i satisfy the granularity condition. Let $H(k,d,\sigma) = \max_i \frac{\min\{D, kd(i)\}}{\sigma_i}$ and let $\hat{k} = \min\{k, k^* \log d_{\max}\}$. Then the following table contains running times for various implementations of procedure REDUCE (assuming that exponentiation is a single step).

	Randomized Implementation	Deterministic Implementation
$1 \le \epsilon \le \frac{1}{poly(n)}$	$O\left(\epsilon^{-2}(m + n\log n)H(k,d,\sigma)\right)$	$O\left(\epsilon^{-1}H(k,d,\sigma)[k^* n \log n + m(\log n + \hat{k})]\right)$
$1 \le \epsilon \le \frac{1}{poly(n)}$, f is $O(\epsilon)$-opt.	$O\left(\epsilon^{-1}(m + n\log n)H(k,d,\sigma)\right)$	$O\left(H(k,d,\sigma)[k^* n \log n + m(\log n + \hat{k})]\right)$

4.3 Further implementation details

In this section, we will show how to get rid of the assumption that exponentiation can be performed in a single step. We will also give a more efficient implementation of the procedure REDUCE for the case when ϵ is fixed.

4.3.1 Removing the assumption that exponentiation can be performed in $O(1)$ time

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

The first step is to note that in the proof of Lemma 4.2, we never used the fact that we reroute flow onto a *shortest* path. We only need that we reroute flow onto a *sufficiently short* path. More precisely, it is easy to convert the proof of Lemma 4.2 into a proof for the following claim.

Lemma 4.9 Suppose σ_i and τ satisfy the granularity condition and let P be a flow path of commodity i. Let Q be a path connecting the endpoints of P such that the length of Q is no more than $\epsilon' \ell(P)/2 + \epsilon' \frac{|f|}{D} |\ell|_1/2$ greater than the length of the shortest path connecting the same endpoints. Then rerouting σ_i units of flow from path P to Q decreases $|\ell|_1$ by $\Omega(\frac{\sigma_i}{\min\{D, kd(i)\}} |\ell|_1 \log m)$.

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

By Lemma 4.9, the length of a path can have a rounding error of $\epsilon' \frac{|f|}{D} |\ell|_1/2$. Each path has at most n edges, so it will suffice to ensure that each edge has a rounding error of $\frac{1}{n}(\epsilon' \frac{|f|}{D}|\ell|_1/2)$. We will now bound this quantity. $|f|$ is the maximum flow on an edge and hence must be at least as large as the average flow on an edge, i.e. $|f| \ge \sum_{vw} f(vw)/m$. Every unit of flow contributes to the total flow on at least one edge, and hence $\sum_{vw} f(vw) \ge D$ and combining with the previous equation, we get that $|f|/D \ge 1/m$. $|\ell|_1$ is at least as big as the length of the longest edge, i.e. $|\ell|_1 \ge e^{\alpha |f|}$. Plugging in these bounds we see that it suffices to compute with an error of at most $\frac{\epsilon'}{nm} e^{\alpha|f|}$. Each edge has a positive length of at most $e^{\alpha|f|}$ and can be expressed as $e^{\alpha|f|}\rho$, where $0 < \rho \le 1$. Thus we need to compute ρ up to an error of $\frac{\epsilon'}{nm}$. To do so, we need to compute $O(\log(\epsilon^{-1} nm))$ bits which by the assumption that ϵ is inverse polynomial in n is just $O(\log n)$ bits.

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

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

Throughout REDUCE all edge length are at most $e^{O(\alpha\tau)}$, and at least one edge has length more than $e^{\alpha\tau}$. Therefore, $|\ell|_1$ is at least $e^{\alpha\tau}$, and by the same argument as for the deterministic case $O(\epsilon^{-1} \log n)$ bits of precision suffice throughout REDUCE. When we first call REDUCE, we must spend $O(\epsilon^{-1} m \log n)$ time to compute all the edge lengths. For each subsequent iteration, we only need to spend $O(\epsilon^{-1} n \log n)$ time updating the $O(n)$ edges whose length has changed. Since each iteration of REDUCE is expected to take $O(\epsilon^{-1}(m + n \log n))$ time to compute shortest paths in FINDPATH, the time for updating edges is dominated by the time required by FINDPATH. While it appears that the time to initially compute all the edge lengths may dominate the time spent in one invocation of REDUCE, we shall see in Section 5 that whenever any of our algorithms calls REDUCE, it will have at least $\Omega(\log n)$ iterations. Each iteration is expected to take at least $\Omega(\epsilon^{-1} m)$ time to compute the shortest paths in FINDPATH. Therefore, the time spent on initializing lengths will be dominated by the running time of REDUCE.

Note that in describing the randomized version of FINDPATH in Lemma 4.6, we assumed we knew the exact lengths. However, by using the approximate lengths we do not significantly change a path's apparent contribution to the sum $\sum_i \sum_{P \in \mathcal{P}_i} \ell(P) f_i(P)$. Hence we do not significantly reduce the probability of selecting a bad path.

Thus we have shown that without any assumptions, REDUCE can be implemented deterministically in the same time as is stated in Theorem 4.8. Although for the randomized version, there is additional initialization time, for all the algorithms in this paper that initialization time is dominated by the time spent in the iterations of REDUCE.

Theorem 4.10 The times required for the deterministic implementations of procedure REDUCE stated in Theorem 4.8 hold without the assumption that exponentiation is a single step. The time required by the randomized implementations increases by an additive factor of $O(\epsilon^{-1} m \log n)$ without this assumption.

4.3.2 Further improvements for fixed ϵ

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

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

$$dist_{\tilde{\ell}}(s,t) \leq dist_\ell(s,t) + n\frac{\epsilon'\ell(P)}{2n} = dist_\ell(s,t) + \frac{\epsilon'\ell(P)}{2}$$

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

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

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

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

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

One can delete flow paths from the array in constant time. If one maintains a list of empty entries, one can also insert in constant time. If the array gets too full, copy its contents into a new array of twice the size. The time required for copying can be amortized over the time required for the insertions that filled the array. If the array gets too empty, copy its contents into a new array of half the size. The time required for copying can be amortized over the time required for the deletions that emptied the array. (See, for example, [1], for a detailed description of this data structure.)

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

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

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

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

Theorem 4.11 Let $f = O(\tau)$ and τ and σ_i satisfy the granularity condition. Let $H(k,d,\sigma) = \max_i \frac{\min\{D,kd(i)\}}{\sigma_i}$ and let $\hat{k} = \min\{k, k^* \log d_{\max}\}$. For any constant $\epsilon > 0$ the procedure REDUCE can be implemented in randomized $O(mH(k,d,\sigma))$ and in deterministic $O(H(k,d,\sigma)m(\log n + \hat{k}))$ time.

5 Concurrent flow algorithms

In this section, we give approximation algorithms for the concurrent flow problem with uniform capacities. We describe two algorithms: CONCURRENT and SCALINGCONCURRENT. CONCURRENT is simpler and is best if ϵ is constant. SCALINGCONCURRENT gradually scales ϵ to the right value, and is faster for small ϵ.

Algorithm CONCURRENT (see Figure 2) consists of a sequence of calls to procedure REDUCE described in the previous section. The initial flow is constructed by routing each commodity i on a single flow path from s_i to t_i. Initially, we set $\sigma_i = d(i)$. Before each call to REDUCE we divide the flow quantum σ_i by 2 for every commodity where this is needed to satisfy the granularity condition (3). Each call to REDUCE modifies the multicommodity flow f so that either $|f|$ decreases by a factor of 2 or f becomes ϵ-optimal. (The procedure REDUCE can set a global flag to indicate whether it has concluded that f is ϵ-optimal.) In the latter case our algorithm can terminate and return the flow. As we will see, $O(\log m)$ calls to REDUCE will suffice to achieve ϵ-optimality.

Theorem 5.1 The algorithm CONCURRENT finds an ϵ-optimal multicommodity flow in $O((\epsilon^{-1}k + \epsilon^{-3}m)(k^* n \log n + m(\log n + \min\{k, k^* \log d_{\max}\}))\log n)$, or in expected time $O((k\epsilon^{-2} + m\epsilon^{-4})(m + n \log n)\log n)$.

Proof: Immediately after the initialization we have $|f| \leq D$. To bound the number of phases we need a lower bound on the minimum value of $|f|$. Observe that for every multicommodity flow f, the total amount of flow in the network is D. Every unit of flow contributes to the total flow on at

$\text{Concurrent}(G, \epsilon, \{d(i), (s_i, t_i) \ : \ 1 \le i \le k\})$
For each commodity i: $\sigma_i \leftarrow d(i)$, create a simple path from s_i to t_i and route $d(i)$ flow on it.
$\tau \leftarrow |f|/2$.
While f is not ϵ-optimal,
  For every i,
    Until σ_i and τ satisfy the granularity condition,
      $\sigma_i \leftarrow \sigma_i/2$.
  Call $\text{Reduce}(f, \tau, \epsilon, d)$.
  $\tau \leftarrow \tau/2$.
Return f.

Figure 2: Procedure Concurrent.

least one of the edges, and hence $\sum_{vw \in E} f(vw) \ge D$. Therefore,

$$|f| \ge D/m. \tag{4}$$

This implies that the number of iterations of the main loop of Concurrent is at most $O(\log m)$. By Theorems 4.3 and 4.8, procedure Reduce invoked during a single iteration of Concurrent first spends $O(m \log n)$ time initializing edge lengths and then executes $O(\epsilon^{-1} \frac{\min\{D, kd(i)\}}{\sigma_i})$ iterations. Throughout the algorithm σ_i is either equal to $d(i)$, or is $\Theta(\epsilon^2 \tau / \log(m\epsilon^{-1}))$ for every i. In the first case,

$$\frac{\min\{D, kd(i)\}}{\sigma_i} = \frac{\min\{D, kd(i)\}}{d(i)} = \min\left\{\frac{D}{d(i)}, k\right\} \le k.$$

In the second case

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

Thus the total number of iterations of the loop of Reduce is at most $O(\epsilon^{-1}(k + \epsilon^{-2}\frac{D}{\tau}\log(m\epsilon^{-1}))$, and the time spent on the initialization the edge length is dominated. The value τ is halved at every iteration, therefore the total number of calls required for all iterations is at most $O(\epsilon^{-1} k \log n)$ plus twice the number required for the last iteration of Concurrent. It follows from (4) that τ is $\Omega(\frac{D}{m})$, and the total number of iterations of the loop of Reduce is at most $O(\epsilon^{-1} k \log n + \epsilon^{-3} m \log n)$. ∎

Consider the special case when ϵ is constant. We use the version of Reduce implemented with an approximate shortest path computation, and apply the bounds of Theorem 4.11 combined with a proof similar to that of Theorem 5.1 to get the following result:

Theorem 5.2 For any constant $\epsilon > 0$, an ϵ-optimal solution for the unit-capacity concurrent flow problem can be found in $O(m(k+m)\log^2 n)$ expected time by a randomized algorithm and in $O(m(k + m)(\log n + \min\{k, k^* \log d_{\max}\}) \log n)$ time by a deterministic algorithm.

If ϵ is less than a constant, we use the algorithm ScalingConcurrent, shown in Figure 3. It starts with a large ϵ, and then gradually scales ϵ down to the required value. More precisely, algorithm ScalingConcurrent starts by applying algorithm Concurrent with $\epsilon = \frac{1}{10}$. Scaling-Concurrent then repeatedly divides ϵ by a factor of 2 and calls Reduce. After the initial call to Concurrent, f is $\frac{1}{10}$-optimal, *i.e.* $|f|$ is no more than twice the minimum possible value. Therefore, $|f|$ cannot be decreased below $\tau/2$, and every subsequent call to Reduce returns an ϵ-optimal multicommodity flow (with the current value of ϵ). As in Concurrent, each call to Reduce uses the largest flow quantum σ permitted by the granularity condition (3).

$\underline{\text{ScalingConcurrent}(G, \epsilon', \{d(i), (s_i, t_i)\ :\ 1 \le i \le k\})}$
$\epsilon \leftarrow \frac{1}{10}$.
Call Concurrent$(G, \epsilon, \{d(i), (s_i, t_i)\ :\ 1 \le i \le k\})$, and let f be the resulting flow.
  $\tau \leftarrow \tau/2$.

While $\epsilon > \epsilon'$,
  $\epsilon \leftarrow \epsilon/2$,
  For every i,
   Until σ_i and τ satisfy the granularity condition,
    $\sigma_i \leftarrow \sigma_i/2$.
  Call Reduce$(f, \tau, \epsilon, \sigma)$.
Return f.

Figure 3: Procedure ScalingConcurrent.

Theorem 5.3 The algorithm ScalingConcurrent finds an ϵ-optimal multicommodity flow in expected time $O((k\epsilon^{-1} + m\epsilon^{-3}\log n)(m + n\log n))$.

Proof: As is stated in Theorem 5.2, the call to procedure Concurrent takes $O(km\log n + m^2\log m)$ time, and returns a multicommodity flow f that is $\frac{1}{10}$-optimal, hence $|f|$ is no more than twice the minimum. Therefore every subsequent call to Reduce returns an ϵ-optimal multicommodity flow f.

The time required by one iteration is dominated by the call to Reduce. The input flow f of Reduce is 2ϵ-optimal, so, by Theorems 4.8 and 4.10, the time required by the randomized implementation of Reduce is $O(\epsilon^{-1}(m + n\log n)\max_i \frac{\min\{D, kd(i)\}}{\sigma_i})$. We have seen that $\max_i \frac{\min\{D, kd(i)\}}{\sigma_i}$ is at most $O(k + \epsilon^{-2}m\log m)$. The value of ϵ is reduced by a factor of two in every iteration. Therefore, the total time required for all iterations is at most twice the time required by the last iteration. The last iteration takes $O((k + \epsilon^{-2}m\log n)(\epsilon^{-1}(m + n\log n)))$ time, which proves the claim. ∎

Consider an implementation of Concurrent or ScalingConcurrent with the deterministic version of Reduce. The time required by FindPath does not depend on ϵ, so we can not claim that the time is bounded by at most twice the time required for the last call to Reduce. Since there are at most $\log \epsilon^{-1}$ iterations, we have the following theorem.

Theorem 5.4 An ϵ-optimal solution to the unit-capacity concurrent flow problem can be found deterministically in time $O(km\log^2 n + (k\log\epsilon^{-1} + \epsilon^{-2}m\log n)(k^* n\log n + m(\log n + \min\{k, k^*\log d_{\max}\})))$.

6 Two applications

In this section we describe two applications of our unit-capacity concurrent flow algorithm. The first application is to efficiently implement Leighton and Rao's sparsest cut approximation algorithm [11], and the second application is to approximately minimize channel width in VLSI routing; the second problem was considered by Raghavan and Thompson [13] and Raghavan [12].

We start by reviewing the result of Leighton and Rao concerning finding an approximately sparsest cut in a graph. For any partition of the nodes of a graph G into two sets A and B, the associated *cut* is the set of edges between A and B, and $\delta(A, B)$ denotes the number of edges in that cut. A cut is *sparsest* if $\delta(A, B)/(|A||B|)$ is minimized. Leighton and Rao [11] gave an $O(\log n)$-approximation algorithm for finding the sparsest cut of a graph. By applying this algorithm they obtained polylog-times-optimal approximation algorithms for a wide variety of NP-complete graph problems, including minimum feedback arc set, minimum cut linear arrangement, and minimum area layout.

Leighton and Rao exploited the following connection between sparsest cuts and concurrent flow. Consider an *all-pairs* multicommodity flow in G, where there is a unit of demand between every pair of nodes. In a feasible flow f, for any partition $A \cup B$ of the nodes of G, a total of at least $|A||B|$ units of flow must cross the cut between A and B. Consequently, one such edge must carry at least $|A||B|/\delta(A, B)$ flow for the sparsest cut $A \cup B$. Leighton and Rao prove an approximate max-flow, min-cut theorem for the all-pairs concurrent flow problem, by showing that in fact this lower bound for $|f|$ is at most an $O(\log n)$ factor below the minimum value. Their approximate sparsest-cut algorithm makes use of this connection. More precisely given a nearly optimal length function (dual variables) they show how to find a partition $A \cup B$ that is within a factor of $O(\log n)$ of the minimum value of $|f|$, and hence of the value of the sparsest cut.[1]

The computational bottleneck of their method is solving a unit-capacity concurrent flow problem, in which there is a demand of 1 between every pair of nodes. In their paper, they appealed to the fact that concurrent flow can be formulated as a linear program, and hence can be solved in polynomial time. A much more efficient approach is to use our unit-capacity approximation algorithm. The number of commodities required is $O(n^2)$. Leighton [10] has discovered a technique to reduce the number of commodities required. He shows that if the graph in which there is an edge connecting each source-sink pair is an expander graph, then the resulting flow problem suffices for the purpose of finding an approximately sparsest cut. (We call this graph the *demand graph.*) In an expander we have:

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

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

By Theorem 5.2, algorithm CONCURRENT takes expected time $O(m^2 \log^2 m)$ to find an appropriate solution for this smaller problem.

Theorem 6.1 An $O(\log n)$-factor approximation to the sparsest cut in a graph can be found by a randomized algorithm in $O(m^2 \log^2 m)$ time.

The second application we discuss is approximately minimizing channel width in VLSI routing. Often a VLSI design consists of a collection of modules separated by channels; the modules are connected up by wires that are routed through the channels. For purposes of regularity the channels have uniform width. It is desirable to minimize that width in order to minimize the total area of the VLSI circuit. Raghavan and Thompson [13] give an approximation algorithm for minimizing the channel width. They model the problem as a graph problem in which one must route wires between pairs of nodes in a graph G so as to minimize the maximum number of wires routed through an edge. To approximately solve the problem, they first solve a concurrent flow problem where there is a commodity with demand 1 for each path that needs to be routed. An optimal solution f_{opt} fails

[1]Their algorithm also works for edge-weighted graphs; weights translate to edge capacities in the corresponding concurrent flow problem.

to be a wire routing only in that it may consist of paths of *fractional* flow. However, the value of $|f_{\rm opt}|$ is certainly a lower bound on the minimum channel width. Raghavan and Thompson give a randomized method for converting the fractional flow $f_{\rm opt}$ to an integral flow, increasing the channel width only slightly. The resulting wire routing f achieves channel width

$$|f| \leq |f_{\rm opt}| + O(\sqrt{|f_{\rm opt}| \log n}) \tag{5}$$

which is at most $w_{\min} + O(\sqrt{w_{\min} \log n})$, where $w_{\min}$ is the minimum width. In fact, the constant implicit in this bound is quite small. Later Raghavan [12] showed how this conversion method can be made deterministic.

The computational bottleneck is, once again, solving a unit-capacity concurrent flow problem. Theorems 5.3 and 5.4 are applicable, and yield good algorithms. But if $w_{\min}$ is $\Omega(\log n)$, we can do substantially better.[2] In this case, a modified version of our algorithm SCALINGCONCURRENT directly yields an integral f satisfying (5), although the big-Oh constant is not as good as that of [13].

Consider the procedure SCALINGCONCURRENT. It consists of two parts. First the procedure CONCURRENT is called with $\epsilon = \frac{1}{10}$ to achieve $\frac{1}{10}$-optimality. Next, SCALINGCONCURRENT repeatedly calls REDUCE, reducing the error parameter ϵ by a factor of two every iteration, till the required accuracy is achieved. The demands are the same for every commodity, hence σ_i is independent of i, and we shall denote it by σ.

We claim that if $w_{min} = \Omega(\log n)$ then σ, which is initially 1 for this application, need never be reduced. Consequently, there remains a single path of flow per commodity, and the randomized conversion method of Raghavan and Thompson becomes unnecessary. We show that these paths constitute a routing with width $w_{\min} + O(\sqrt{w_{\min} \log n})$.

First suppose the call to CONCURRENT terminates because the granularity condition becomes false. At this point, we have that

$$1 > \epsilon^2 \tau / (51 \log(7m\epsilon^{-1})). \tag{6}$$

We have that $\tau \geq |f|/2$ and $\epsilon = \frac{1}{10}$, and therefore $|f| = O(\log n)$. By our assumption that $w_{\min} = \Omega(\log n)$, and hence $|f| \leq w_{\min} + O(\sqrt{w_{\min} \log n})$.

Now assume that the call to CONCURRENT terminates with a $\frac{1}{10}$-optimal flow. We proceed with SCALINGCONCURRENT. It terminates when the granularity condition becomes false, at which point inequality (6) implies that $\epsilon^2 = O((\log m)/\tau)$. The flow f is ϵ-optimal and integral. So $|f| \leq w_{\min} + O(w_{\min}\sqrt{(\log m)/\tau})$. Since $\tau = |f|/2 \geq w_{\min}/2$, this bound on $|f|$ is at most $w_{\min} + O(\sqrt{w_{\min} \log m})$, as required.

Theorem 6.2 If $w_{\min}$ denotes the minimum possible width and $w_{\min} = \Omega(\log m)$, a routing of width $w_{min} + O(\sqrt{w_{min} \log n})$ can be found by a randomized algorithm in expected time $O(km \log n \log k + k^{3/2}(m + n \log n)/\sqrt{\log n})$, and by a deterministic algorithm in time $O(k \log k(k^* n \log n + mk^* + m \log n))$.

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

We have $D = k$ and $d(i) = 1$ for every i, and throughout the algorithm we have $\sigma_i = 1$ for every i. The number of calls to REDUCE during CONCURRENT is $O(\log k)$ (initially $|f| \leq k$, and it never gets below 1 with $\sigma = 1$). Therefore the number iterations of the loop of REDUCE required during

[2]This is the case of most interst, for if $w_{\min}$ is $o(\log n)$, then the error term in (5) dominates $w_{\min}$.

CONCURRENT is $O(k \log k)$. Next we proceed with SCALINGCONCURRENT. The number of iterations is at most $O(\log k)$, because ϵ is reduced by a factor of two each iteration, ϵ starts at $\frac{1}{10}$ and never gets below $1/k$. Each iteration is a call to REDUCE, which in turn results in $O(k)$ iterations of the loop of REDUCE.

The time required by one iteration of the loop deterministically is $O(k^* n \log n + m(k^* + \log m))$, and the total time to find a good routing of wires is $O(k \log k(k^* n \log n + mk^* + m \log n))$.

The expected time required by the randomized implementation of REDUCE is $O(m \log n + k\epsilon^{-1}(m + n \log n))$. The total expected time required by CONCURRENT is $O(mk \log k \log n)$. After the call to CONCURRENT ϵ decreases by a factor of two each iteration, it follows that the total expected time required for all iterations is $O(m \log n \log \epsilon^{-1})$ plus twice the time for the last call to REDUCE. During the last call to REDUCE, $\epsilon^{-1} = O(\sqrt{k/\log n})$, so the time required for all iterations is $O(km \log n \log k + k^{3/2}(m + n \log n)/\sqrt{\log n})$. This time dominates the time required by CONCURRENT since $w_{\min} = \Omega(\log n)$ implies $k = \Omega(\log n)$. ∎

Acknowledgments

We are grateful to Andrew Goldberg, Tom Leighton, Satish Rao, David Shmoys, and Pravin Vaidya for helpful discussions.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[2] R. Dial. Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM*, 12:632–633, 1969.

[3] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

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

[5] M. D. Hansen. Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 604–610. IEEE, October 1989.

[6] 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.

[7] P. Klein, A. Agrawal, R. Ravi, and S. Rao. Approximation through multicommodity flow. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 726–737, 1990.

[8] P. Klein and C. Stein. Leighton-Rao might be practical: a faster approximation algorithm for uniform concurrent flow. Unpublished manuscript, November 1989.

[9] P. Klein, C. Stein, and É. Tardos. Leighton-Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 310–321, May 1990.

[10] F. T. Leighton, November 1989. Private communication.

[11] 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.

[12] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 10–18, 1986.

[13] P. Raghavan and C. D. Thompson. Provably good routing in graphs: regular arrays. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 79–87, 1985.

[14] R. Ravi, A. Agrawal, and P. Klein. Ordering problems approximated: single-processor scheduling and interval graph completion. In *Proceedings of the 1991 ICALP Conference*, pages 751–762, 1991.

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

[16] D D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[17] E. Tardos. Improved approximation algorithm for concurrent multi-commodity flows. Technical Report 872, School of Operations Research and Industrial Engineering, Cornell University, October 1989.

[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.