# Faster Approximation Schemes for Fractional Multicommodity Flow Problems [*]

George Karakostas [†]
Department of Computing and Software
McMaster University
1280 Main St. W.
Hamilton, Ontario L8S 4K1, Canada
karakos@mcmaster.ca
fax: +1 905-524-0340

**Abstract**

We present fully polynomial approximation schemes for concurrent multicommodity flow problems that run in time of minimum possible dependency on the number of commodities $k$. We show that by modifying the algorithms by Garg & Könemann [7] and Fleischer [5] we can reduce their running time on a graph with $n$ vertices and $m$ edges from $\tilde{O}(\varepsilon^{-2}(m^2+km))$ to $\tilde{O}(\varepsilon^{-2}m^2)$ for an *implicit* representation of the output, or $\tilde{O}(\varepsilon^{-2}(m^2+kn))$ for an *explicit* representation, where $\tilde{O}(f)$ denotes a quantity that is $O(f \log^{O(1)} m)$. The implicit representation consists of a set of trees rooted at sources (there can be more than one tree per source), and with sinks as their leaves, together with flow values for the flow directed from the source to the sinks in a particular tree. Given this implicit representation, the approximate value of the concurrent flow is known, but if we want the explicit flow per commodity per edge, we would have to combine all these trees together, and the cost of doing so may be prohibitive. In case we want to calculate explicitly the solution flow, we modify our schemes so that they run in time poly-logarithmic in $nk$ ($n$ is the number of nodes in the network). This is within a poly-logarithmic factor of the trivial lower bound of time $\Omega(nk)$ needed to explicitly write down a multicommodity flow of $k$ commodities in a network of $n$ nodes. Therefore our schemes are within a poly-logarithmic factor of the minimum possible dependency of the running time on the number of commodities $k$.

## 1 Introduction

Flow problems are at the core of many optimization problems, cf. the book by Ahuja et al. [1]. Usually the input to such a problem consists of a directed network $G=(V,E)$, an edge utilization (or capacity) function $u: E \rightarrow \mathbb{R}^+$ and a source-sink pair $(s,t)$ (single commodity flow) or a set of $k$ source-sink pairs $(s_i,t_i),\ 1 \le i \le k$ (multicommodity flow). In general we want to calculate flows $f_i$ from $s_i$ to $t_i$ that would optimize an objective function, subject to the constraint that the sum of flows through an edge cannot exceed the capacity of the edge, possibly together with other constraints. The simplest multicommodity flow problem is the *maximum multicommodity flow* problem: the objective is the maximization of the total flow (throughput) $\sum_{i=1}^{k} f_i$ subject to the capacity constraints. A more complex variation called the *maximum concurrent flow* problem is the following: besides the above,

[*]An extended abstract of this paper appeared in the 13th Annual ACM/SIAM Symposium on Discrete Algorithms (SODA), 2002 [12].

[†]Research supported by an NSERC Discovery grant.

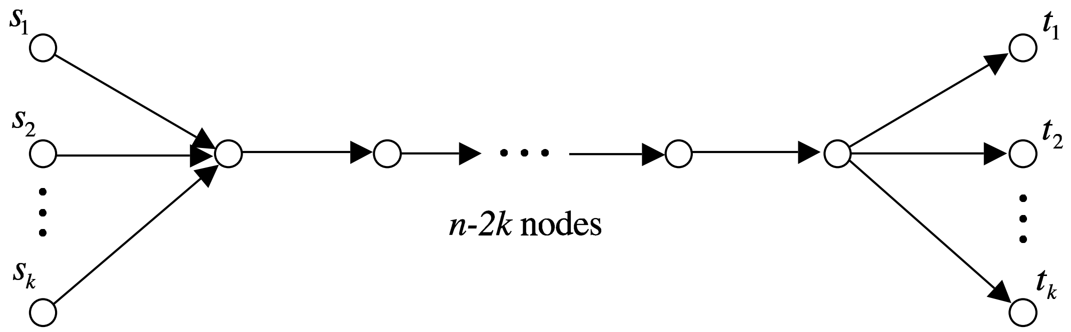

Figure 1: A simple example of $k$ commodities, each with a flow path of length $\Theta(n)$.

we are also given a set of $k$ *demands* $d_i$, one for each commodity pair $(s_i, t_i)$, and we are asked to satisfy the maximum fraction $\lambda$ of all demands, i.e. $\max \lambda$, subject to $f_i \geq \lambda d_i$, $\forall i$. The maximum concurrent flow problem and some other problems related to it are the focus of this work. Note that we can assume that at most one commodity corresponds to a single source-sink pair of nodes, since we can combine commodities with the same origin and destination into a single commodity with demand equal to the summation of the individual demands. If a fraction $\lambda$ of the demand of this aggregated commodity is satisfied, then a fraction $\lambda$ of the demand of each one of its constituent commodities is satisfied as well.

The maximum concurrent flow problem has recently attracted the attention of researchers working on approximation algorithms. The maximum concurrent flow problem is the dual of the LP relaxation of the *sparsest cut* problem. Finding the sparsest cut of a graph with weighted edges means finding a cut $S$ that minimizes the ratio of the weight of the edges cut to the total demand that is separated by the cut. There are many approximation algorithms for the sparsest cut based on rounding the solution of its LP relaxation [17], [4]. Thus any algorithm for solving the maximum concurrent flow problem provides also a solution to this LP and is the starting point for these approximation algorithms (more on the sparsest cut problem and other problems related to it and to the maximum concurrent flow can be found in the survey by Shmoys [21]). Most recently the major breakthrough by Arora, Rao, and Vazirani [3] achieved an approximation ratio of $O(\sqrt{\log n})$ by rounding its SDP relaxation, with its running time improved to $O(n^2)$ by Arora et al. [2].

Although the problems considered in this paper can be solved optimally in polynomial time (since they can be formulated as linear programs), in many applications the need to solve them quickly is greater than the need for optimality. Hence for the last decade there has been a series of papers on the design of *fully polynomial time approximation schemes (FPTAS)* for multicommodity flow problems. An approximation scheme for a maximization problem is an algorithm that, given an instance of the problem and any positive number $\varepsilon$, computes a solution within a factor $(1-\varepsilon)$ of the optimal. If the running time of an approximation scheme is *poly(input size, 1/ε)* then the scheme is *fully polynomial-time.*

A line of research (cf. [20] [14] [16] [8] [18] [19] [13] [10]) based on Langrangian relaxation and linear programming decomposition techniques has led to ever decreasing running times for multicommodity flow problems. All these algorithms compute an initial flow, and then try to redistribute flow from more congested paths to less congested paths. In a departure from this general theme, Young [22] described a randomized algorithm which builds a solution from scratch, and does not reroute flow. At every step Young's algorithm solves a shortest path problem (on the dual LP), instead of a maximum single commodity flow problem in earlier algorithms.

Garg and Könemann [7] followed a similar approach. However, the simple and elegant analysis of their algorithm allowed them to make a greater advance at each step: while Young's procedure pushes a unit flow at each step along a (shortest) path, the one by Garg and Könemann pushes enough flow so as to saturate the minimum capacity edge of the path. Their results generalize to *fractional packing problems*, and yield an algorithm whose running time does not depend on the *width* of the problem (i.e. the maximum extent to which any constraint can be violated). Width-free bounds for general fractional packing, which specialize also to multicommodity flows, were previously obtained by Grigoriadis and Khachiyan [9] via logarithmic potential Langrangian approximation schemes. Fleischer [5] noticed that for the maximum multicommodity flow problem, we can use *approximate* shortest paths, in order to avoid recalculations of shortest paths for commodities with a common source. By making the greatest possible advance for every *source* (instead for every *commodity*), she reduces the dependence of the running time on $k$ to a mere logarithmic factor. Unfortunately, her techniques for the maximum multicommodity flow do not apply to the maximum concurrent flow. Nevertheless, using different techniques (which we also use and are described in what follows) she reduces the dependence of the running time of the algorithm in [7] on $k$ by replacing a maximum single commodity flow computation with a shortest path one.

In this paper, we build on the work of Garg and Könemann [7] and Fleischer [5] to minimize the dependence of the maximum concurrent flow on $k$. We also base our algorithm on the iterative update of a dual solution, while we distinguish between an *implicit* and an *explicit* representation of the corresponding primal solution. The first representation arises in case we are interested in computing quickly a good approximation of the optimum $\lambda$ and not in explicitly producing a flow that achieves this $\lambda$ in the form of a list of flow value per commodity for every edge. For this case we match Fleischer's running time for the maximum multicommodity flow. The implicit representation consists of a set of trees rooted at sources (there can be more than one tree per source), and with sinks as their leaves, together with flow values for the flow directed from the source to the sinks in a particular tree. We base our improvement on the observation that the analysis by Garg and Könemann still holds if at every step we try to push flow from a single source along the whole *tree* produced by Dijkstra's algorithm. This has the effect of Fleischer's techniques on the maximum multicommodity flow, and combined with the other improvements in [5] produces better algorithms for the maximum concurrent flow and related problems.

In most applications though, it is important to output an explicit description of a flow that achieves the near-optimal $\lambda$. The implicit representation described above is problematic, since extra work is needed in order to compute explicitly the amount of flow passing through each edge and for each commodity from such a representation. We call the explicit enumeration of the flow of each commodity that goes through each edge the *explicit* representation of this flow. Simple examples of networks with $n$ nodes and $k$ commodities can be used to show that this explicit representation can be of size $\Omega(nk)$. One such example is shown in Figure 1, where each commodity is routed through a path of length $\Theta(n)$. This gives a trivial lower bound of $\Omega(nk)$ on the dependency of the running time of any algorithm that produces an explicit representation on $k$. In all previous work, no extra effort was dedicated to the output of an explicit representation of the computed flow, because the dependence of all previous algorithms on $k$ was big enough to account for the explicit computation and output of the flow. Our aim in this work is more ambitious: we want to develop an algorithm that produces explicitly (in the sense above) the near-optimum flow in time that depends *minimally* on the number of commodities, i.e., it has a dependency of $O(nk)$ on $k$. In the main result of this work, we manage to achieve this goal (within a poly-logarithmic factor), for any constant $\varepsilon > 0$, by a more careful distribution of flow through the shortest path tree used in the implicit case.

Therefore our schemes achieve a nearly (within poly-logarithmic factors) minimal dependence on the number of commodities $k$ for both the implicit and explicit representation cases. Our results are summarized in Table 1. Our improvement in the running time becomes apparent in many applications

| Problem | Previous best | This work (implicit) | This work (explicit) |
|---|---|---|---|
| max concurrent flow | $\tilde{O}(\varepsilon^{-2}m(m+k))$ [5] | $\tilde{O}(\varepsilon^{-2}m^2)$ | $\tilde{O}(\varepsilon^{-2}(m^2+kn))$ |
| min cost concurrent flow | $\tilde{O}(\varepsilon^{-2}m(m+k)\log M)$ [5] | $\tilde{O}(\varepsilon^{-2}m^2\log M)$ | $\tilde{O}(\varepsilon^{-2}(m^2\log M+kn))$ |
| max concurrent flow (lossy) | $\tilde{O}(\varepsilon^{-2}m(k+m))$ [6] | $\tilde{O}(\varepsilon^{-2}m^2)$ | $\tilde{O}(\varepsilon^{-2}(m^2+kn))$ |
| min cost concurrent flow (lossy) | $\tilde{O}(km^2+\varepsilon^{-2}m(k+m)S)$ [6] | $\tilde{O}(m^2(k+\varepsilon^{-2}S))$ | $\tilde{O}(km^2+\varepsilon^{-2}(m^2S+kn))$ |

Table 1: Comparison of previous algorithms to our results. $\tilde{O}(f)$ denotes a quantity that is $O(f\log^{O(1)} m)$, $M$ is the largest integer that is used to specify capacities, costs or demands, and $S$ is the geometric binary search factor $(\log\varepsilon^{-1}+\log\log M)$.

(e.g. routing over a telephone network) where the network topology is itself sparse (for example $m=O(n)$) but there are demands for almost every pair of nodes (i.e. $k=\Omega(n^2)$).

A generalization of the flow problems described above involves the introduction of one more network parameter, the *gain factor* $\gamma : E \rightarrow \mathbb{R}^+$. An edge $e$ with gain factor $\gamma(e) > 0$ allows a fraction $\gamma(e)$ of the flow that enters $e$ to exit. Hence, in case $\gamma(e) > 1$, $\gamma(e) = 1$ or $\gamma(e) < 1$, the edge increases, conserves or 'leaks' the flow that passes through it. A network with a gain factor $\gamma$ is a *generalized network* and the obvious extensions of flow problems to such networks are *generalized flow* problems. In the special case of $\gamma(e) \leq 1$ for all edges, the network is called *lossy.* Our analysis applies directly to lossy networks, with the appropriate (more general) definition of a shortest path. Fleischer and Wayne [6] describes how a modification of Dijkstra's algorithm can be used to solve the generalized shortest path problem in the case of lossy networks. By adapting our algorithms to this framework very much in the same way the Fleischer's results [5] were adapted to generalized networks in [6], we get the results shown in Table 1.

## 2 Notation

In what follows, $G$ will represent the underlying network graph for the flow computations, $n$ is its number of nodes and $m$ is its number of edges. We will assume that $G$ is connected, so $m=\Omega(n)$. We use the notation $\tilde{O}(f)$ to denote a quantity that is $O(f\log^{O(1)} m)$.

## 3 Maximum concurrent flow: the implicit representation case

In this section we present an algorithm that computes a near-optimal value of $\lambda$ in time that is independent of the number of commodities $k$. Although the algorithm doesn't produce an enumeration of the amount of flow that crosses every edge for every commodity, it is the basis for the algorithm that will achieve exactly that, and will be presented in the next section.

The statement of the problem is the following: We are given a directed graph $G=(V,E)$ with edge capacities $u : E \rightarrow \mathbb{R}^+$ and $k$ commodities with $(s_i, t_i)$ being the *(source,sink)* pair for commodity $i$. For each commodity $i$ we are also given a demand $d(i) > 0$. We want to find the largest $\lambda$ such that there is a multicommodity flow which routes at least $\lambda d(i)$ units of commodity $i$ for all commodities $i$. Since we are interested in fractional flows, this $\lambda$ can be expressed as the solution of a linear program:

Let $\mathcal{P}_i$ be the set of paths between $s_i$ and $t_i$ in $G$, and let $\mathcal{P} := \cup_i \mathcal{P}_i$. Variable $x(P)$ denotes the amount of flow sent along path $P$, for every $P \in \mathcal{P}$. Then the LP path formulation of the maximum concurrent multicommodity flow problem is the following:

$$\begin{array}{rcll} \text{maximize} & \lambda & \text{s.t.} & \\ \sum_{P:e\in P} x(P) & \leq & u(e) & \forall e \in E \\ \sum_{P\in\mathcal{P}_i} x(P) & \geq & \lambda d(i) & \forall i \\ x(P) & \geq & 0 & \forall P \end{array}$$

Note that this LP formulation is of exponential size, but we will never need to explicitly write it down.

The dual LP has a variable $l(e)$ for each capacity constraint of the primal and a variable $z(i)$ for every commodity demand constraint:

$$\begin{array}{rcll} \text{minimize} & & \sum_{e\in E} u(e)l(e) & \text{s.t.} \\ \sum_{e\in P} l(e) & \geq & z(i) & \forall i, \forall P \in \mathcal{P}_i \\ \sum_{i=1}^{k} d(i)z(i) & \geq & 1 & \\ l(e) & \geq & 0 & \forall e \\ z(i) & \geq & 0 & \forall i \end{array}$$

Let $D(l) := \sum_e u(e)l(e)$ be the quantity minimized by the dual. Define $\text{dist}_i(l)$ as the distance of the shortest path from $s_i$ to $t_i$ in $G$ under the length function $l$. Let $\alpha(l) := \sum_i d(i)\text{dist}_i(l)$. As observed by Garg and Könemann [7], minimizing $D(l)$ under the dual constraints is equivalent to computing lengths $l(e)$ for the edges that minimize $D(l)/\alpha(l)$. Let $\beta := \min_l D(l)/\alpha(l)$.

The algorithm by Garg and Könemann [7] runs in time $\tilde{O}(\varepsilon^{-2}m(m+k))$, provided $\beta \geq 1$. If $\beta < 1$, there is a standard procedure for scaling the problem (see [5] for a description of this procedure) so that $\beta \geq 1$, using $k$ maximum bottleneck path computations[1] that run in time $O(\max\{n,k\}m)$. We use this scaling procedure as well, but we improve the running time for the case $\beta \geq 1$ to $\tilde{O}(\varepsilon^{-2}m^2)$. For the moment we will assume that $\beta \geq 1$. For completeness, we will describe later this scaling procedure that removes this assumption.

## 3.1 The case $\beta \geq 1$

We modify the algorithm in [7] (also used in [5]). In order to demonstrate exactly where our modifications happen, and for completeness reasons, we give the whole analysis, although many parts of it are a repetition of their analysis. The algorithm proceeds in phases and each phase is composed of $|\mathcal{S}|$ iterations, with $\mathcal{S}$ being the set of *sources* (notice that this is *not* necessarily the number of *commodities* $k$). In iteration $j$ of the $i^{\text{th}}$ phase we consider all the commodities with the same source $s_j$. Let $c_1, c_2, \ldots, c_r$ be these commodities. We route $d(c_q), q = 1, \ldots, r$ units of commodity $c_q$ in a series of steps. Throughout its running, the algorithm maintains a length function $l(\cdot)$ that gives the length $l(e)$ for every edge $e \in E$. Let $l_{i,j,s}$ be the length function at the end of the $s^{\text{th}}$ step. At every step we compute the shortest path tree from $s_j$ to the sinks of commodities $c_q, q = 1, \ldots, r$. Let $P^{c_q}_{i,j,s}$ be the path in this tree from $s_j$ to the sink of commodity $c_q$, i.e. $P^{c_q}_{i,j,s}$ has length $\text{dist}_{c_q}(l_{i,j,s-1})$. It is crucial to our improvement that the shortest path tree for all commodities with a common source $s_j$ can be computed with only *one* call to Dijkstra's algorithm. Let $d^{c_q}_{i,j,s} > 0$ be the amount of commodity $c_q$ that has not been routed yet at step $s$ (obviously $d^{c_q}_{i,j,0} = d(c_q)$). Notice that we consider only commodities with *strictly* positive remaining demand, and we ignore commodities whose demand has

[1]The scaling procedure used in [7] solves $k$ maximum flow problems. This turns out to be the bottleneck of their algorithm, causing a running time that matches previously known concurrent flow algorithms. This bottleneck was improved by Fleischer [5]. We improve the algorithm by Garg and Könemann [7] itself.

already been completely routed. Then at step $s$ we route $f^{c_q}_{i,j,s} = d^{c_q}_{i,j,s-1}/\sigma$ units of each commodity $c_q$ along path $P^{c_q}_{i,j,s}$, where $\sigma$ is a scaling factor that ensures we don't push through an edge flow greater than its capacity. After this we set $d^{c_q}_{i,j,s} := d^{c_q}_{i,j,s-1} - f^{c_q}_{i,j,s}$, and for every edge $e$ on these paths we set

$$\begin{aligned} l_{i,j,s}(e) &:= l_{i,j,s-1}(e)(1 + \varepsilon \cdot \frac{\text{total new flow through } e}{u(e)}) \\ &= l_{i,j,s-1}(e)(1 + \varepsilon \cdot \frac{\sum_{c_q : e \in P^{c_q}_{i,j,s}} f^{c_q}_{i,j,s}}{u(e)}). \end{aligned}$$

Note that for every saturated edge $e$ its length $l(e)$ increases by a factor of $1 + \varepsilon$, and that in each iteration, during each step except possibly the last one, at least one edge is saturated, i.e., gets $u(e)$ units of flow. After the last step $s$ $d^{c_q}_{i,j,s} = 0$ for all commodities $c_q$. The whole procedure ends as soon as $D(l_{i,j,s}) \geq 1$, and produces a length function $l$ and a corresponding multicommodity flow value vector $x(q), q = 1 \dots k$ which is infeasible for the primal LP. However, by applying the analysis of [7] we prove that scaling down $x$ will produce a close to optimal *feasible* solution.

Initially $l_{1,1,0}(e) = \delta/u(e)$ for all edges $e$, where $\delta$ is a parameter to be defined later. The algorithm is described in Figure 2. For brevity we use $P_{c_q}$ to denote $P^{c_q}_{i,j,s}$, the shortest path for commodity $c_q$ in the $s^{\text{th}}$ step of the $j^{\text{th}}$ iteration in the $i^{\text{th}}$ phase.

After step $s$ of the $j^{\text{th}}$ iteration of the $i^{\text{th}}$ phase

$$\begin{aligned} D(l_{i,j,s}) &= D(l_{i,j,s-1}) + \varepsilon \cdot \sum_{e \in P_{c_1} \cup \ldots \cup P_{c_r}} l_{i,j,s-1}(e) \sum_{q : e \in P_{c_q}} f^{c_q}_{i,j,s} \\ &= D(l_{i,j,s-1}) + \varepsilon \cdot \sum_{e \in P_{c_1} \cup \ldots \cup P_{c_r}} \sum_{q : e \in P_{c_q}} l_{i,j,s-1}(e) \cdot f^{c_q}_{i,j,s} \\ &= D(l_{i,j,s-1}) + \varepsilon \cdot \sum_{q=1}^{r} f^{c_q}_{i,j,s} \cdot \sum_{e \in P_{c_q}} l_{i,j,s-1}(e) \\ &= D(l_{i,j,s-1}) + \varepsilon \cdot \sum_{q=1}^{r} f^{c_q}_{i,j,s} \cdot \mathrm{dist}_{c_q}(l_{i,j,s-1}) \end{aligned}$$

where the third equality is a simple rearrangement of terms, and the fourth equality holds because $P_{c_q}$ is by definition the shortest path for commodity $c_q$ under the length function of the previous step. Using the same arguments for the iterations, we can show that

$$D(l_{i,1,0}) \leq D(l_{i-1,1,0}) + \varepsilon \cdot \alpha(l_{i,1,0}).$$

This recursive relation together with $D(l_{0,1,0}) = m\delta$ and the fact that $\beta = \min_l \frac{D(l)}{\alpha(l)}$ imply that

$$\beta \leq \frac{\varepsilon(t-1)}{(1-\varepsilon)\ln \frac{1-\varepsilon}{m\delta}}. \tag{1}$$

**Lemma 1** *If $\beta \geq 1$ the algorithm terminates after at most $t := \lceil \beta \log_{1+\varepsilon} \frac{1+\varepsilon}{\delta} \rceil$ phases.*

**Proof:** In order to bound the primal solution $\lambda$ produced by the algorithm, Garg and Könemann [7] prove the following claim. We repeat its proof here in order to make clear why $\sigma$ is needed in Figure 2:

**Claim 1** $\lambda > \frac{t-1}{\log_{1+\varepsilon} \frac{1+\varepsilon}{\delta}}$.

**Input :** Graph $G = (V, E)$, capacities $u(e)$, pairs $(s_i, t_i)$ with demands $d(i), 1 \leq i \leq k$, accuracy $\varepsilon$.

**Output:** Primal solution $x$ (implicit representation), $\lambda$.

```
Initialize $l(e) := \delta/u(e), \forall e, \quad x(P) := 0, \forall P$.
**while** $D(l) < 1$ **do**
    **for** $i = 1$ **to** $|\mathcal{S}|$ **do**
        $d'(c_q) := d(c_q), q = 1, \ldots, r$
        **while** $D(l) < 1$ and $d'(c_q) > 0$ for some $q$ **do**
            $P_{c_q} :=$ shortest path in $\mathcal{P}_{c_q}$ using $l$, $q = 1, \ldots, r$ with $d'(c_q) > 0$
            $f_{c_q} := d'(c_q),\ q = 1, \ldots, r$ with $d'(c_q) > 0$
            $\sigma := \max\{1, \max_{e \in P_{c_1} \cup \ldots \cup P_{c_r}} \{\frac{\sum_{c_q : e \in P_{c_q}} f_{c_q}}{u(e)}\}\}$
            $f_{c_q} := f_{c_q}/\sigma$
            $d'(c_q) := d'(c_q) - f_{c_q}$
            $x(P_{c_q}) := x(P_{c_q}) + f_{c_q}$   $\Big\}\ q = 1, \ldots, r$ with $d'(c_q) > 0$
            $l(e) := l(e)(1 + \varepsilon \cdot \frac{\sum_{c_q : e \in P_{c_q}} f_{c_q}}{u(e)}),\ \forall e \in P_{c_1} \cup \ldots \cup P_{c_r}$
        **end while**     /* end of step */
    **end for**           /* end of iteration */
**end while**             /* end of phase */
$x(P) := x(P)/\log_{1+\varepsilon} \frac{1+\varepsilon}{\delta},\ \forall P$
$\lambda := \min_i \frac{\sum_{P \in \mathcal{P}_i} x(P)}{d(i)}$
**Output** $x, \lambda$
```

Figure 2: Maximum concurrent flow FPTAS for $\beta \geq 1$.

**Proof:** Since after each phase we route $d(i)$ units of commodity $i$, after $t-1$ phases we have routed $(t-1)d(i)$ units of flow. This flow may be infeasible (it may violate capacity constraints), hence in the end we scale it down to make it feasible. Since we scale down the flow through a path by $\sigma$ in every step so that no edge is overflowed, for every $u(e)$ units of flow routed through edge $e$ we increase the length $l(e)$ by a factor of at least $1+\varepsilon$. At the beginning $l_0(e) = \delta/u(e)$ and by the end of phase $t-1$ we have $l_{t-1}(e) < 1/u(e)$ (since $D(l_{t-1}) < 1$). Hence at the end of the algorithm $l_t(e) < (1+\varepsilon)/u(e)$ (each step of the last phase increases $l(e)$ by at most a factor of $1+\varepsilon$, and the algorithm stops as soon as $D(l_{t-1}) \geq 1$). Therefore the total amount of flow through $e$ in all $t$ phases is strictly less than $\log_{1+\varepsilon} \frac{(1+\varepsilon)/u(e)}{\delta/u(e)} = \log_{1+\varepsilon} \frac{1+\varepsilon}{\delta}$ times its capacity. So by scaling the flow by $\log_{1+\varepsilon} \frac{1+\varepsilon}{\delta}$ we get the claimed *feasible* primal solution.

□

This implies that

$$1 \leq \frac{\beta}{\lambda} < \frac{\beta}{t-1} \log_{1+\varepsilon} \frac{1+\varepsilon}{\delta} \tag{2}$$

which, in turn, implies the lemma.

□

From (2) and (1) we have

$$\frac{\beta}{\lambda} < \frac{\varepsilon \log_{1+\varepsilon} \frac{1+\varepsilon}{\delta}}{(1-\varepsilon)\ln \frac{1-\varepsilon}{m\delta}} = \frac{\varepsilon}{(1-\varepsilon)\ln(1+\varepsilon)} \cdot \frac{\ln \frac{1+\varepsilon}{\delta}}{\ln \frac{1-\varepsilon}{m\delta}}$$

By setting

$$\delta := \frac{1}{(1+\varepsilon)^{\frac{1-\varepsilon}{\varepsilon}}} \cdot \left(\frac{1-\varepsilon}{m}\right)^{\frac{1}{\varepsilon}} \tag{3}$$

the dual-primal solution ratio becomes less than $(1-\varepsilon)^{-3}$ and we can pick $\varepsilon$ so that this ratio is less than $1+w$ for any $w > 0$. Hence we have proven the following

**Lemma 2** *The proposed algorithm for the concurrent multicommodity flow problem is an approximation scheme, provided $\beta \geq 1$.*

Next we deal with the case $\beta < 1$ and prove that this approximation scheme runs in polynomial time.

## 3.2 The case $\beta < 1$

For $\delta$ as in (3) the number of phases cannot be more than $t = \lceil \frac{\beta}{\varepsilon} \log_{1+\varepsilon} \frac{(1+\varepsilon)m}{1-\varepsilon} \rceil$, provided that $\beta \geq 1$. Garg and Könemann in [7] observe that if $\zeta(i)$ is the maximum flow of commodity $i$ that can be routed in $G$ when no other commodity is routed, the value $\zeta = \min_i\{\zeta(i)/d(i)\}$ is an upper bound of the optimal solution (at best, all maximum *single* commodity flows for all commodities can be routed simultaneously). The solution that routes a fraction of $1/k$ of each flow $\zeta(i)$ is a feasible solution, so $\zeta/k$ is a lower bound of the optimal solution. Hence $\zeta/k \leq \beta \leq \zeta$, so by multiplying the initial demands by $\zeta/k$ we ensure that $\beta \geq 1$. But this computation requires $k$ maximum single commodity flow computations, and becomes the bottleneck of the algorithm in [7]. Fleischer [5] reduced this time by observing that an $m$-factor approximation of the $\zeta(i)$'s is enough for our purposes. Since any flow can be decomposed into at most $m$ path flows, the flow along a maximum capacity path is an $m$-approximation to $\zeta(i)$. Such a path can be computed easily in $O(n \log n + m)$ time by using a modified version of Dijkstra's algorithm that updates its node labels using the maximum capacities computed so far (instead of edge costs). In fact *all* these paths/estimates can be computed in $O(min\{n,k\}m \log n)$ time, if we group the commodities by their common source node.

The procedure above can make $\beta$ as big as $km$, thus increasing the running time of the algorithm (recall that the number of phases $t$ depends on $\beta$). Garg and Könemann [7] propose the following solution to reducing the dependence on $\beta$: if the algorithm does not stop after $T := 2\lceil \frac{1}{\varepsilon} \log_{1+\varepsilon} \frac{(1+\varepsilon)m}{1-\varepsilon} \rceil$ phases, we know that $\beta \geq 2$. We double the demands of all commodities, so that $\beta$ is halved and still $\beta \geq 1$, and continue running the algorithm for another $T$ phases. If the algorithm doesn't stop we double again the demands and continue doing this until the algorithm ends. Since every time $\beta$ is halved, this doubling can happen at most $\log(km)$ times, the total number of phases is at most $T \log(km)$. This number can be further reduced by using a standard technique of Plotkin et al. [18] that is used in [7], [5]. First we can compute a 2-approximate solution $\hat{\beta}$ using the scheme above. This takes $O(\log k \log m)$ phases. Since $\beta \leq \hat{\beta} \leq 2\beta$, if we multiply all the demands by $\hat{\beta}/2$, only at most $T$ additional phases are needed to get an $\varepsilon$-approximation because $1 \leq \beta \leq 2$.

## 3.3 The running time

The number of iterations in each phase is bounded by the number of sources, which is at most $\min\{n,k\}$. Since we need $O(\log m \log k)$ phases to reduce $\beta$ to less than 2, and at most $T = O(\varepsilon^{-2}\log m)$ additional phases to get an $\varepsilon$-approximation, the total number of iterations is $O(\min\{n,k\}\log m(\log k + \varepsilon^{-2}))$. In order to compute the total running time, we need to calculate the number of steps in the algorithm. At every step other than the last step in an iteration, the length of an edge increases by a factor of at least $1+\varepsilon$. At the beginning each edge has length $\delta/u(e)$ and at the end its legth is at most $(1+\varepsilon)/u(e)$. So the number of steps exceeds the number of iterations by at most $m\log_{1+\varepsilon}\frac{1+\varepsilon}{\delta} = O(\varepsilon^{-2}m\log m)$. Note that this estimate holds for both the first part of the algorithm that reduces $\beta$ to $\beta \le 2$ and the second part that computes the final solution. Hence the total number of steps is at most $\tilde{O}(\varepsilon^{-2}(m + \min\{n,k\})) = \tilde{O}(\varepsilon^{-2}m)$.

**Lemma 3** *Each step involves* one *run of Dijkstra's algorithm that takes time $O(n\log n + m)$ using Fibonacci heaps, and other computations that take at most $O(m)$ time, for a total of $O(m + n\log n)$ time per step.*

**Proof:** The only part of the inner while-loop of Figure 2 whose running time requires some explaining is the calculation of quantities

$$F(e) := \sum_{c_q : e\in P_{c_q}} f_{c_q}, \ \forall e \in P_{c_1} \cup \ldots \cup P_{c_r}$$

where $P_{c_q}$ is the current shortest path for commodity $c_q$, $q = 1,\ldots,r$. Note that all these paths start from the same source and they form a shortest path tree. It will be easier for the exposition to assume that the sinks of the commodities $c_q$ are the leaves of this tree (if a sink is at an internal node of the shortest path tree, we can connect this sink to this node via a new artificial edge of infinite capacity). For each one of these commodities we are routing $f_{c_q}$ units of flow through the tree path $P_{c_q}$. The calculation of $F(e)$ is done as follows:

**Step 0:** Set $F(e) := f_{c_q}$ for all edges $e$ that connect the sink of $c_q$ to the tree.

**Step 1:** Repeat the following steps until all the tree nodes have been processed:

- Pick a tree node $v$ which is connected to its children $u_1,\ldots,u_d$ via edges $e_1 = (v,u_1),\ldots,e_d = (v,u_d)$ and such that all $F(e_i)$, $i = 1,\ldots,d$ have already been calculated. Set $G(v) := \sum_{i=1}^{d} F(e_i)$.
- Let $e = (w,v)$ be the edge that connects $v$ to its tree ancestor $w$. Set $F(e) := G(v)$.

It is clear that the running time of this procedure is $O(n)$, and once we have the $F(e)$'s, we can calculate easily $c$ and $l(e)$ for all the tree edges in time $O(n)$.

□

Putting all these together with the scaling procedure we get an algorithm that runs in time $\tilde{O}(\varepsilon^{-2}m^2)$.

**Theorem 1** *The algorithm described above is a FPTAS for the maximum concurrent multicommodity flow problem, that runs in $\tilde{O}(\varepsilon^{-2}m^2)$ time.*

### 3.4 Implicit flow representation

The algorithm in Figure 2 calculates an approximation of the *value* $\lambda$ of a maximum concurrent flow. With the same algorithm and in the same running time we can also compute an *implicit* representation of a flow that achieves this value: all we have to do is to store the shortest path tree computed in every step, together with the amount of flow routed through each path in this tree (the $x(P_{c_q})$'s in Figure 2). The problem with this representation is that it may produce many more paths than needed. This can be avoided by a more sophisticated choice of paths used to route flow, as described in the next section.

## 4 Maximum concurrent flow: the explicit case

In most applications it is important to output an explicit description of a flow that achieves the near-optimal $\lambda$. The implicit representation described above is problematic, since extra work is needed in order to compute explicitly the amount of flow passing through each edge and for each commodity from such a representation. Let $x_e(q)$ be the amount of flow of commodity $q \in \{1, \ldots, k\}$ that passes through edge $e \in E$. We would like our algorithm to output the explicit enumeration of $x_e(q)$, $\forall q, e$. As we show in the introduction, for very simple examples of networks with $n$ nodes and $k$ commodities this explicit representation can be of size $\Omega(nk)$. This gives a trivial lower bound of $\Omega(nk)$ on the dependency of the running time of any algorithm that produces an explicit representation on $k$. In this section we present the main result of this work: we achieve a running time that depends minimally (within a poly-logarithmic factor) on the number of commodities, for any constant $\varepsilon > 0$. We do that by a more careful distribution of flow through the shortest path tree used in the implicit case.

We will alter the algorithm of Figure 2 so that it computes *explicitly* a flow that achieves an almost optimal value for $\lambda$ in time whose dependence on the number of commodities $k$ is at most $O(nk)$ (times poly-logarithmic factors). The crucial observation is that as long as in every step we route enough flow to either saturate an edge or so that no more flow remains to be routed by the current iteration (this will happen in the last step of this iteration), our analysis of correctness and running time remains the same. We will saturate an edge (if possible) by routing the *whole* remaining flow for a commodity until we get to a commodity whose remaining flow can be routed only partially and saturates an edge. We charge the flow paths of commodities routed as a whole to the commodity itself, and the flow paths of commodities routed partially to the saturated edge. Therefore every iteration $j$ is charged for *exactly* as many paths as the commodities with source $s_j$, which means that every phase is charged for *exactly* $k$ flow paths (of at most $n$ edges each). Also every saturation of an edge is charged for *exactly* one path (and increases the length of this edge by a factor of at least $1 + \varepsilon$, from the previous analysis). The modified algorithm is shown in Figure 3.

### 4.1 Correctness

Note that the only change in the algorithm is the way flow is distributed to paths during a step. The correctness analysis carries over to this modified algorithm exactly as is in Section 3. The main differences are in the output and the running time.

### 4.2 Running time

The modifications do not change the estimates for the number of phases and iterations within each phase. Therefore there are at most $O(\log m(\log k + \varepsilon^{-2}))$ phases. This results in at most $O(\min\{n, k\} \log m(\log k + \varepsilon^{-2}))$ iterations ($\min\{n, k\}$ iterations per phase). As was the case before

**Input :** Graph $G = (V, E)$, capacities $u(e)$, pairs $(s_i, t_i)$ with demands $d(i), 1 \leq i \leq k$, accuracy $\varepsilon$.

**Output:** Explicit solution $x$, $\lambda$.

```
Initialize $l(e) := \delta/u(e),\ x_e(q) := 0,\ \ \forall e \in E, q = 1, \dots, k$.
while $D(l) < 1$ do
    for $i = 1$ to $|\mathcal{S}|$ do
        $d'(c_q) := d(c_q), q = 1, \dots, r$
        while $D(l) < 1$ and $d'(c_q) > 0$ for some $q$ do
            $P_{c_q}$ := shortest path in $\mathcal{P}_{c_q}$ using $l$, $q = 1, \dots, r$ with $d'(c_q) > 0$
            $u'(e)$ := $u(e),\ \forall e \in P_{c_1} \cup \dots \cup P_{c_r}$
            for $q = 1$ to $r$ do
                $c := \min_{e \in P_{c_q}} u'(e)$
                if $d'(c_q) \leq c$ then
                    $\left.\begin{array}{lll} x_e(c_q) & := & x_e(c_q) + d'(c_q) \\ u'(e) & := & u'(e) - d'(c_q) \end{array}\right\} \forall e \in P_{c_q}$
                    $f_{c_q}$ := $d'(c_q)$
                    $d'(c_q)$ := $0$
                else
                    $x_e(c_q)$ := $x_e(c_q) + c,\ \forall e \in P_{c_q}$
                    $f_{c_q}$ := $c$
                    $d'(c_q)$ := $d'(c_q) - c$
                    break        /* out of the inner for-loop */
            end for
            $l(e) := l(e)(1 + \varepsilon \cdot \frac{\sum_{c_q : e \in P_{c_q}} f_{c_q}}{u(e)}),\ \forall e \in P_{c_1} \cup \dots \cup P_{c_r}$
        end while    /* end of step */
    end for    /*end of iteration */
end while    /* end of phase */
$x_e(q) := x_e(q) / \log_{1+\varepsilon} \frac{1+\varepsilon}{\delta},\ \forall e \in E, q = 1, \dots, k$
$\lambda := \min_i \frac{\text{flow}(s_i,t_i)}{d(i)}$
Output $\lambda, x$
```

Figure 3: Modified maximum concurrent flow FPTAS for $\beta \geq 1$ that produces an explicit flow.

the modification, each step will saturate an edge, unless it is the last step of an iteration. Therefore, the number of steps will exceed the number of iterations by the maximum possible edge saturations. Each such saturation increases the length function of the saturated edge by a factor at least $1 + \varepsilon$. Just as before, the total number of steps is $\tilde{O}(\varepsilon^{-2} m)$. In every step we route as many *whole* remaining demands of commodities considered in the current iteration as possible, until an edge is saturated (if all commodities are wholly routed, the current iteration finishes). Hence the commodities routed during the current step are finished with (for the current iteration) except possibly the last one that may have some demand left to be routed in a subsequent step. We charge the path update cost (the cost of an iteration of the inner for-loop) to the commodity itself if the whole remaining demand for this commodity was routed through this path, and we charge the path cost to the saturated edge if only a part of the remaining demand for this commodity was routed through this path (resulting to the saturation of the edge). The path update cost is at most $O(n)$. How many such updates are there? Note that in a phase a commodity demand can be partially routed many times but can be wholly routed only once (this would be the last time we deal with this commodity in the current phase). Hence there are at most as many *whole* routings in the current phase as there are commodities ($k$), for a total of $O((\#\text{ of phases})k) = O(k \log m(\log k + \varepsilon^{-2}))$. The total number of *partial* routings is at most the number of edge saturations ($O(\varepsilon^{-2} m \log m)$ from Section 3.3). Therefore, in total we have at most $O(\varepsilon^{-2}(m+k)\log m + k \log k \log m) = \tilde{O}(\varepsilon^{-2}(m+k))$ path updates, for a total cost of $\tilde{O}(\varepsilon^{-2}(m+k)n)$.

The cost of the rest of the operations (Dijkstra's algorithm etc.) can be estimated as in Section 3.3. There are two points that have to be made:

- The update of the length $l(e)$ for every edge $e$ in the modified algorithm presented in Figure 3 can be implemented by keeping track of the flow that was routed through every edge during one step. We can use one variable $f(e)$ for every edge $e$, that is updated every time the edge is in a flow path in the inner for-loop (at the beginning of the current step we set $f(e) := 0$). Then, the update of the length will be as follows:

$$l(e) := l(e)(1 + \varepsilon \frac{f(e)}{u(e)})$$

- The last two steps of the algorithm scale down the solution so that it is feasible, and compute $\lambda$. Since we know the scaling factor even before the algorithm starts executing, we can scale down the amount by which the variable $x_e(q)$ is increased in the inner for-loop. If we do this then the second to last step of scaling $x$ is not needed. Also it is very easy to keep track of the total flow routed so far for every commodity, and this can be used to calculate $\lambda$.

By distributing the calculations as just described, the running time of all operations outside the inner for-loop is at most $\tilde{O}(\varepsilon^{-2} m^2)$. From the previous paragraph we know that the total cost of the inner for-loop is $\tilde{O}(\varepsilon^{-2}(m+k)n)$. Therefore the total running time is $\tilde{O}(\varepsilon^{-2}(m^2 + kn))$. Note that the commodity dependent part of the running time bound comes exclusively from the explicit calculation of the flow that achieves value $\lambda$.

## 5 Minimum cost concurrent flow

A cost version of the maximum concurrent flow problem is the following: in addition to demands and capacities, we are also given a budget $B$ and a cost function $c : E \rightarrow \mathbb{R}^+$ for a unit of flow that passes through an edge, and we are asked to maximize the concurrent flow without violating the budget constraint. It is clear that if we have a FPTAS for this problem, we have a FPTAS for the minimum cost concurrent flow, by just using binary search to compute the minimum budget that satisfies all the demands (i.e. $\lambda \geq 1$). This search incurs a factor of at most $\log M$, where $M$ is the biggest number used to specify capacities, demands or costs. Note that we can have different costs for different commodities, since there is a different primal variable for every commodity-path pair.

The analysis is almost identical to the analysis for the maximum concurrent flow above and in [7]. The dual has one extra variable $\phi$ for the budget constraint. The algorithm is the same as in the maximum concurrent flow case, but instead of $l$ the length function we use is $l + c \cdot \phi$. Initially $\phi = \delta/B$. The flow sent through at every step is the flow calculated above but scaled down (if necessary) so that its total cost is not more than $B$ (in the explicit case, we route commodities until either an edge becomes saturated, or their cost reaches $B$, or all commodities are routed). Lengths $l$ are updated as before, and the new $\phi$ becomes $\phi := \phi(1 + \varepsilon \cdot \frac{\text{cost of flow sent}}{B})$. For the analysis we need only to observe that at every step either the length $l$ of an edge or $\phi$ increases by a factor of at least $1 + \varepsilon$ (and, in the explicit case, we charge the path cost of partially routed commodities to the saturated edge or the 'saturated cost' respectively). Also $\phi$ can be at most $(1 + \varepsilon)/B$ at the end of the algorithm (the algorithm ends when $D(l, \phi) \geq 1$ and $D(l, \phi) := \sum_e l(e)u(e) + \phi B$). For the case $\beta < 1$, we apply Fleisher's procedure [5] to approximate the solution to maximum-flow-with-budget problems: perform a binary search on the $m$ capacities, in order to find the maximum possible capacity of a path that meets the budget; the latter check can be done in every binary search step in time $O(m \log m)$ by running a shortest path computation.

The analysis above together with the analysis in [7] shows the following:

**Theorem 2** *There is a FPTAS that computes the minimum cost concurrent flow in time $\tilde{O}(\varepsilon^{-2} m^2 \log M)$ in the implicit case, and $\tilde{O}(\varepsilon^{-2}(m^2 \log M + kn))$ in the explicit case.*

Note that in the explicit case the binary search factor $\log M$ doesn't affect the path update part of the cost $\tilde{O}(\varepsilon^{-2} kn)$. This is due to the fact that we need to run the explicit version of the maximum concurrent flow with a budget constraint algorithm only in the last step of the binary search (i.e., when we know that a binary search step is the last, we repeat it using the explicit representation version of the algorithm).

## 6 Concurrent flow for lossy networks

A generalization of the flow problems discussed so far involves the introduction of one more network parameter, the *gain factor* $\gamma : E \to \mathbb{R}^+$. An edge $e$ with gain factor $\gamma(e) > 0$ allows a fraction $\gamma(e)$ of the flow that enters $e$ to exit. Hence, in case $\gamma(e) > 1$, $\gamma(e) = 1$ or $\gamma(e) < 1$, the edge increases, conserves or 'leaks' the flow that passes through it. A network with a gain factor $\gamma$ is a *generalized network* and the obvious extensions of flow problems to such networks are *generalized flow* problems. In the special case of $\gamma(e) \leq 1$ for all edges, the network is called *lossy.*

The analysis above applies directly to lossy networks, with the appropriate (more general) definition of a shortest path. Fleischer and Wayne [6] describe how a modification of Dijkstra's algorithm can be used to solve the generalized shortest path problem in the case of lossy networks in time $O(m + n \log m)$ (Theorem 1 in [6]). They also provide a generalization of the method to produce estimates $\zeta(i), i = 1, \ldots, k$ for the case $\beta < 1$ that runs in $O(\min\{k, n\}(m + n \log m))$ time (Lemma 8 in [6]). By introducing these modifications to our algorithm for the maximum concurrent flow problem we get the following

**Theorem 3** *There is a FPTAS that computes the maximum concurrent flow for lossy networks implicitly in time $\tilde{O}(\varepsilon^{-2} m^2)$ and explicitly in time $\tilde{O}(\varepsilon^{-2}(m^2 + nk))$.*

For the minimum cost concurrent flow problem, Fleischer and Wayne [6] use their approximation algorithms for generalized maximum flow problems (Section 4 in [6]) to compute $O(m)$ approximations to $\min\{n^2, k\}$ generalized maximum-flow-with-budget problems and produce estimates $\zeta(i), i = 1, \ldots, k$. The running time bound of computing these estimates in the first part of the algorithm appears as the first expression in the bounds of the following theorem, and the running time bound for the second (scaled) part appears as the second expression:

**Theorem 4** *There is a FPTAS that computes the minimum cost concurrent flow for lossy networks in time* $\tilde{O}(km^2 + \varepsilon^{-2}m^2(\log \varepsilon^{-1} + \log\log M))$ *in the implicit and* $\tilde{O}(km^2 + \varepsilon^{-2}(m^2(\log \varepsilon^{-1} + \log\log M) + kn))$ *in the explicit case.*

The $(\log \varepsilon^{-1} + \log\log M)$ factor in Theorem 4 is due to the fact that instead of simple binary search, we can use the geometric-mean binary search technique of Hassin [11].

## 7 Open problems

The main problem with the running time we can achieve is the quadratic dependence on $1/\varepsilon$. This becomes a problem when $\varepsilon$ becomes very small. Klein and Young [15] give evidence that this may be a lower bound for a general class of approximation schemes based on the *Dantzig-Wolfe model* (our schemes as well as the schemes of [7] and [5] follow this model). But they leave open whether one can achieve subquadratic dependence on $1/\varepsilon$ by a possible increase of the dependence of the running time on $m$. This question can be answered in the affirmative if we consider interior-point methods. In these methods though, the running time dependence on the other parameters of the problem $(n, m, k)$ is higher that the dependence achieved by the Langrangian approximation line of research. It would be of great practical and theoretical importance to achieve the dependence on $\varepsilon$ that interior-point methods achieve but with the small dependence on $n, m, k$ Langrangian approximation schemes achieve.

## 8 Acknowledgements

I am grateful to Michael Grigoriadis for a very extensive review of this manuscript and a plethora of extremely useful comments. Many thanks to Tami Carpenter for initiating this research. Many thanks also to Takashi Takabatake, Muralidharan Kodialam and other people that brought a mistake in the initial algorithm of [12] into our attention. I'm also grateful to Tomasz Radzik for pointing out the example in Figure 1, and to Cliff Stein and Phil Klein for many helpful remarks on [12] and for bringing [15] to my attention.

I am also indebted to two anonymous reviewers whose detailed comments greatly improved a previous version of this manuscript.

## References

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network flows: theory, algorithms, and applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] S. Arora, E. Hazan, and S. Kale. $O(\sqrt{\log n})$ approximation to SPARSEST CUT in $\tilde{O}(n^2)$ time. In *Proceedings 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*, pp. 238–247, 2004.

[3] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. In *Proceedings 36th ACM STOC*, pp. 222–231, 2004.

[4] Y. Aumann and Y. Rabani. An $O(logk)$ approximate min-cut max-flow theorem and approximation algorithm. *SIAM Journal on Computing*, **27**:1, pp. 291-301, 1998.

[5] L. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal on Discrete Mathematics*, **13**, 2000.

[6] L. Fleischer and K.D. Wayne. Fast and simple approximation schemes for generalized flow. *Mathematical Programming Ser. A*, 91:2, pp. 215–238, 2002.

[7] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *Proceedings 39th IEEE FOCS*, pp. 300-309, November 1998.

[8] M. D. Grigoriadis and L. G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. SIAM Journal on Optimization, **4**:86-107, 1994.

[9] M. D. Grigoriadis and L. G. Khachiyan. Coordination complexity of parallel price-directive decomposition. *Mathematics of Operations Research* 21:2, pp. 321-340, 1996.

[10] M. D. Grigoriadis and L. G. Khachiyan. Approximate minimum-cost multicommodity flows in $\tilde{O}(knm/\epsilon^2)$ time. *Mathematical Programming*, **75**:477-482, 1996.

[11] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17:36-42, 1992.

[12] G. Karakostas Faster approximation schemes for fractional multicommodity flow problems. In *Proceedings of 13th ACM/SIAM SODA*, pp. 166-173, 2002.

[13] D. Karger and S. Plotkin. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. In *Proceedings of 27th ACM STOC*, pp. 18-25, 1995.

[14] P. Klein, S. Plotkin, C. Stein, and É. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, **23**:466-487, 1994.

[15] P. Klein and N. Young. On the Number of Iterations for Dantzig-Wolfe Optimization and Packing-Covering Approximation Algorithms. In *Proceedings of 7th Integer Programming and Combinatorial Optimization Conference (IPCO)*, LNCS 1610, p. 320, 1999.

[16] T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast approximation schemes for multicommodity flow problems. *JCSS* **50**:228-243, 1995.

[17] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, *15*:215-246, 1995.

[18] S. Plotkin, D. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, **20**:257-301, 1995.

[19] T. Radzik. Fast deterministic approximation for the multicommodity flow problem. In *Proceedings of the 6th ACM/SIAM SODA*, pp. 486-492, 1995.

[20] F. Shahrokhi and D.W. Matula. The maximum concurrent flow problem. *JACM*, **37**:318-334, 1990.

[21] D. Shmoys. *Cut problems and their application to divide-and-conquer*, Chapter 5 in D.S. Hochbaum (Ed.) *Approximation algorithms for NP-hard problems*, PWS Publishing Company, Boston, 1997.

[22] N. Young. Randomized routing without solving the linear program. In *Proceedings of the 6th ACM/SIAM SODA*, pp. 170-178, 1995.