

Lisa K. Fleischer · Kevin D. Wayne

## Fast and simple approximation schemes for generalized flow<sup>\*</sup>

Received: June 3, 1999 / Accepted: May 22, 2001

**Abstract.** We present fast and simple fully polynomial-time approximation schemes (FPTAS) for generalized versions of maximum flow, multicommodity flow, minimum cost maximum flow, and minimum cost multicommodity flow. We extend and refine fractional packing frameworks introduced in FPTAS's for traditional multicommodity flow and packing linear programs. Our FPTAS's dominate the previous best known complexity bounds for all of these problems, some by more than a factor of  $n^2$ , where  $n$  is the number of nodes. This is accomplished in part by introducing an efficient method of solving a sequence of generalized shortest path problems. Our generalized multicommodity FPTAS's are now as fast as the best non-generalized ones. We believe our improvements make it practical to solve generalized multicommodity flow problems via combinatorial methods.

### 1. Introduction

Generalized network flow problems generalize traditional network flow problems by specifying a *gain factor*  $\gamma(e) > 0$  for each arc  $e$ . For each unit of flow that enters the arc,  $\gamma(e)$  units exit. For traditional network flows, the gain factor of every arc is one. Generalized flows satisfy capacity constraints and node conservation constraints just like traditional network flows. In this paper we consider the following generalized flow problems.

**Generalized maximum flow:** *Find a generalized flow that maximizes the amount of flow reaching one distinguished node called the sink, given unlimited supply at another distinguished node called the source.*

**Generalized minimum cost maximum flow:** *Given nonnegative arc costs, find a generalized maximum flow of minimum cost.*

**Generalized maximum multicommodity flow:** *Given  $k$  source-sink pairs  $(s_j, t_j)$ , find a generalized flow maximizing the sum over all pairs  $j$  of the flow reaching  $t_j$  from  $s_j$ .*

**Generalized maximum concurrent flow:** *Given  $k$  source-sink pairs  $(s_j, t_j)$  and demands  $d_j$ ,  $1 \leq j \leq k$ , find the maximum  $\lambda$  and a corresponding generalized flow that delivers  $\lambda d_j$  units of flow to  $t_j$  by sending flow from  $s_j$ , for each  $j$ .*

---

L.K. Fleischer: GSIA, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15217. Partially supported by NSF through grants DMS 95-27124, and EIA-0049084. e-mail: lkf@andrew.cmu.edu

K.D. Wayne: Computer Science Department, Princeton University, Princeton, NJ 08544. Research supported in part by ONR through grant AASERT N00014-97-1-0681. e-mail: wayne@cs.princeton.edu

\* An extended abstract of this paper appears in [42].

**Generalized minimum cost concurrent flow:** *Given nonnegative arc costs, find a generalized maximum concurrent flow of minimum cost.*

We also consider a useful common generalization of all of the above problems.

**$k$ -commodity packing problem:** *Given  $A \in \mathbf{R}_{\geq 0}^{m \times n}$ ,  $c \in \mathbf{R}_{> 0}^n$ ,  $b \in \mathbf{R}_{> 0}^m$ ,  $d \in \mathbf{R}_{> 0}^k$ , and  $K \in \mathbf{R}^{k \times n}$  that is block diagonal in which each block consists of exactly one row containing all ones, find a vector  $x \in \mathbf{R}^n$  and scalar  $\lambda$  that is a solution to  $\max\{\lambda \mid Ax \leq b; -Kx + d\lambda \leq 0; x, \lambda \geq 0\}$ .*

Generalized flows have many applications. Gain factors can represent physical transformations of a commodity due to leakage, evaporation, breeding, theft, or interest rates. They can also represent transformations from one commodity into another as a result of manufacturing, scheduling, or currency exchange: converting dollars into euros, raw materials into processed materials into finished products, acres into feed into fattened cattle, crude oil into processed oil, and machine time into completed orders. For more information and applications, see Ahuja et al. [3] or Glover et al. [15].

In this paper, we design fast and simple approximation schemes for the above mentioned versions of generalized flow. Our goal is to find an  $\epsilon$ -approximate solution for any error parameter  $\epsilon > 0$ . For generalized maximum flow and maximum concurrent flow, an  $\epsilon$ -approximate solution is a generalized flow that has value at least  $(1 - \epsilon)$  times the optimal. For versions with costs, an  $\epsilon$ -approximate solution is a generalized flow that has value at least  $(1 - \epsilon)$  times the maximum value and costs at most the optimal cost. For each problem, we develop a *fully polynomial-time approximation scheme (FPTAS)*. A FPTAS is a family of algorithms that finds an  $\epsilon$ -approximate solution for each  $\epsilon > 0$  in time polynomial in the size of the input and  $1/\epsilon$ . Here, the size of the input is specified by the number of nodes  $n$ , the number of arcs  $m$ , the number of commodities  $k$ , and the largest integer  $M$  used to specify any of the capacities, costs, gain factors, and demands. To simplify the run times, we use  $\mathcal{O}(f)$  to denote  $f \log^{\mathcal{O}(1)} m$ .

### 1.1. Previous work

Generalized flow has a rich history. The problem was first studied by Kantorovich [25] and Dantzig [9]. All of our problems can be solved exactly via general purpose linear programming techniques, including simplex, ellipsoid, and interior point methods. Researchers have also designed efficient combinatorial algorithms that exploit the underlying network flow structure of the problem. Goldberg, Plotkin, and Tardos [17] designed the first polynomial-time combinatorial algorithms for generalized maximum flow. Their algorithms were refined and improved upon in [18, 19, 34] with the fastest exact algorithm developed so far by Goldfarb, Jin, and Orlin [19]. For generalized maximum flow, researchers have also developed fast approximation schemes in [8, 32, 37]. The second author [41] proposed the first polynomial combinatorial algorithms for generalized minimum cost flow. There are no known exact polynomial combinatorial algorithms for generalized multicommodity flow.

Our approximation schemes build upon combinatorial approximation schemes for traditional multicommodity flow. Shahrokhi and Matula [35] proposed a FPTAS for the maximum concurrent flow problem with uniform arc capacities. They introduced

a length function which is exponential in the total flow going through that arc. They iteratively route flow along shortest paths with respect to the exponential length function. The method was refined by Klein et al. [26] and extended to handle arbitrary arc capacities by Leighton et al. [27]. Plotkin, Shmoys, and Tardos [31] and Grigoriadis and Khachiyan [20, 21] extended the method further to solve more general fractional packing and covering problems. Goldberg [16] proposed a faster randomized version; Radzik [33] derandomized it. Garg and Könemann [14] simplified the method for packing problems, drawing on ideas from Young [43].

Oldham [30] proposed FPTAS's for a variety of generalized flow problems, using the fractional packing framework of Garg and Könemann [14]. When this framework is applied to traditional network flow, each iteration routes flow along a shortest path with respect to the exponential length function. The fundamental computation is a shortest path problem with nonnegative arc lengths. For generalized flow, the framework requires a subroutine to solve a version of this shortest path problem with gain factors. All efficient combinatorial methods for this subroutine make use of a Bellman-Ford [5, 12, 29] style procedure of Aspvall and Shiloach [4] that tests whether or not some guess on the generalized shortest path value is too big, too small, or just right. The presence of gain factors makes computing shortest paths more complicated and expensive than standard Bellman-Ford. Currently, the best complexity bound for the problem is  $\tilde{O}(mn^2)$  due to [7, 23, 30].

## 1.2. Our contributions

We refine and extend the generalized flow FPTAS's of Oldham [30] and the fractional packing framework of Garg and Könemann [14]. We obtain faster approximation schemes for the following problems: generalized maximum flow, generalized minimum cost maximum flow, generalized maximum multicommodity flow, generalized maximum concurrent flow, and generalized minimum cost concurrent flow. We distinguish between networks with and without flow-generating cycles, as we are able to obtain even faster and simpler algorithms in the latter case. Figure 1 summarizes our run times for each problem, and compares them with previous work.

Oldham's [30] generalized flow FPTAS's rely on the fractional packing framework of Garg and Könemann [14]. The crucial subroutine in [30] is a generalized shortest path computation: it requires  $\tilde{O}(mn^2)$  time using any of the subroutines in [7, 23, 30]. The packing algorithm involves solving a sequence of shortest path problems. Significantly, the shortest path problems all involve the same underlying graph, with only minor perturbations in arc lengths. Until now, researchers did not know how to solve the sequence of shortest path problems any faster than solving each one independently. We show how to reduce the amortized time per generalized shortest path computation to  $\mathcal{O}(mn)$ . As a result, our algorithms are faster than previous ones by a factor of  $n$ . To do this, we first observe that all of the above-mentioned generalized shortest path subroutines rely on a Bellman-Ford style procedure due to Aspvall and Shiloach [4]. We are able to break the  $\Omega(mn^2)$  barrier by extending a technique introduced in [11]: We directly embed the Aspvall-Shiloach procedure in a simple and more practical scaling framework, instead of using one of the generalized shortest path subroutines as a black box.

Exact algorithm for generalized flow	Previous best FPTAS	Our FPTAS
<b>Maximum flow</b> $\tilde{O}(m^3 I)$ [19] $\tilde{O}(m^{1.5} n^2 I)$ [39]	$\tilde{O}(\log \epsilon^{-1} m^2 n)$ [32] $\tilde{O}(\log \epsilon^{-1} m(m + n \log I))$ [34, 37]	$\tilde{O}(\epsilon^{-2} m^2)^\dagger$ $\tilde{O}(\log \epsilon^{-1} m(m + n \log I))$
<b>Minimum cost flow</b> $\tilde{O}(m^{1.5} n^2 I)$ [39]	$\tilde{O}(\log \epsilon^{-1} m^2 n^2)$ [41]	$\tilde{O}(\epsilon^{-2} m^2 J)^\dagger$ $\tilde{O}(\epsilon^{-2} m^2 n J)$
<b>Maximum multicommodity flow</b> $\tilde{O}(k^{2.5} m^{1.5} n^2 I)$ [39] $\tilde{O}((k^{0.5} m^3 + km^{1.5} n^{1.5})(m + I) I)$ [24]	$\tilde{O}(\epsilon^{-2} km^2 n^2)$ [30] $\tilde{O}(\log \epsilon^{-1} (k^{0.5} m^3 + km^{1.5} n^{1.5}) n I)$ [24]	$\tilde{O}(\epsilon^{-2} m^2)^\dagger$ $\tilde{O}(\epsilon^{-2} m^2 n)$
<b>Maximum concurrent flow</b> same as max multicommodity flow	same as max multicommodity flow	$\tilde{O}(\epsilon^{-2} (k + m) m)^\dagger$ $\tilde{O}(\epsilon^{-2} (k + m) m n)$
<b>Minimum cost concurrent flow</b> same as max multicommodity flow	$\tilde{O}(\epsilon^{-2} \log \epsilon^{-1} km^2 n^3 I)$ [30] $\tilde{O}(\log \epsilon^{-1} (k^{0.5} m^3 + km^{1.5} n^{1.5}) n I)$ [24]	$\tilde{O}(m(km + \epsilon^{-2} (k + m) J))^\dagger$ $\tilde{O}(mn(km + \epsilon^{-2} (k + m) J))^\dagger$

**Fig. 1.** Comparison of work on generalized flow problems. Above  $m$  denotes the number of arcs,  $n$  denotes the number of nodes,  $k$  denotes the number of commodities,  $M$  denotes the biggest integer used to represent any of the costs, capacities, or gain factors, and  $1 - \epsilon$  is the approximation ratio.  $I := \log M$ ;  $J := \log \log M + \log \epsilon^{-1}$ .  $\dagger$  denotes run time for networks with no flow-generating cycles.

We show how, in most natural applications, to replace the Bellman-Ford style procedure with a faster and simpler  $\mathcal{O}(m + n \log n)$  Dijkstra-style [10, 13] procedure. The Dijkstra-style procedure works under the practical assumption that the underlying network has no flow-generating cycles. A *flow-generating cycle* is a cycle such that the product of its gain factors exceeds one. Flow-generating cycles represent arbitrage in financial networks and perpetual energy sources in energy networks, and do not appear to occur often in practice. Generalized networks with and without flow-generating cycles are analogous to traditional shortest path networks with and without negative cost cycles. By disallowing flow-generating cycles, we sacrifice a small amount of generality, but this greatly simplifies the underlying combinatorial structure of the problem. As a result, our specialized FPTAS's are much simpler and faster than previous algorithms for the problem. They are faster by roughly a factor of  $n$  over our FPTAS's that work in networks with flow-generating cycles, and by a factor of  $n^2$  or more over the best results prior to this paper.

Using a technique introduced in [11], we extend an improved version of the Garg-Könemann algorithm to solve the generalized maximum multicommodity flow problem. This saves an additional factor of  $k$ , where  $k$  is the number of commodities. This leads to overall improvements over previous results by a factor of  $kn$  for networks with flow-generating cycles and by a factor of roughly  $kn^2$  for networks without flow-generating cycles.

Unlike the single commodity flow problem, the natural formulations of the multicommodity concurrent flow problems are not packing LP's. Thus, the above techniques do not apply directly. To cope with this obstacle, we extend the Garg and Könemann [14] framework to the  $k$ -commodity packing problem. This enables us to achieve the same improvements for generalized multicommodity flow that we obtained for the single commodity versions.

## 2. Preliminaries

### 2.1. Generalized networks

A *generalized network* is a directed graph  $G = (V, E)$  with capacity function  $u : E \rightarrow \mathbf{R}_{>0}$ , gain function  $\gamma : E \rightarrow \mathbf{R}_{>0}$ , and a cost function  $c : E \rightarrow \mathbf{R}$ . Since our goal in this paper is to design approximation algorithms, we assume that all arc costs are nonnegative; otherwise, in order to provide a meaningful approximation guarantee, we would need to determine whether the optimum value is positive or negative, and this appears to be as difficult as solving the problem to optimality.

### 2.2. Lossy networks and networks with no flow-generating cycles

Arbitrary generalized networks allow each gain factor to be any positive number. We refer to a network with no gain factor exceeding one as a *lossy network*. This captures many natural generalized networks (including traditional networks), where flow only “leaks” or is conserved as it is sent through the network. We specifically design some of our algorithms to take advantage of the special structure of lossy networks.

A *flow-generating cycle* is a cycle such that the product of its gain factors exceeds one. Lossy networks cannot have flow-generating cycles, but networks with no flow-generating cycles need not be lossy. It is possible to transform a network with no flow-generating cycles into an equivalent lossy network in  $\mathcal{O}(mn)$  time: Given a network with no flow-generating cycles, introduce an artificial node  $s$ , and connect this node to every other node with a unit gain arc. Compute the highest gain path from  $s$  to every other node, and then delete  $s$ . The highest gain path is well-defined because the network has no flow-generating cycles. The gain of the highest gain path can be computed in  $\mathcal{O}(mn)$  time using the Bellman-Ford shortest path algorithm [5, 12, 29] with lengths  $l = -\log \gamma$  (or by specializing the Bellman-Ford algorithm directly to problem to avoid taking logarithms). Let  $\mu(v)$  be the gain of the highest gain path from  $s$  to  $v$ . Now, replace the gain of each arc  $(v, w)$  with its relabeled gain  $\gamma(v, w)\mu(v)/\mu(w)$ . Since  $\mu(w) \geq \mu(v)\gamma(v, w)$ , the relabeled gain is at most 1. Relabel the capacity as  $u(v, w)/\mu(v)$ , and the cost as  $c(v, w)\mu(v)$ . Then, any generalized flow  $g(v, w)$  in the relabeled network has a unique correspondence with a generalized flow  $g(v, w)\mu(v)$  in the original network. The two flows have the same cost, and each obeys capacity and flow conservation constraints if the other does. This “relabeling” technique plays an important role in many generalized flow algorithms, and was first used by Truemper [38].

### 2.3. Extensions

Our algorithms easily extend to handle other standard generalized flow variants. For example, in the generalized maximum flow problem, we can choose to maximize the net flow sent out of the source, rather than into the sink. We can also handle multiple sources by connecting a new “super-source” to each of the original sources with an infinite (or sufficiently large) capacity arcs of unit gain. Within the packing framework, we can also handle multiple budget constraints, different cost functions for different commodities, and different gain factors for different commodities.

#### 2.4. Packing algorithm

Our algorithms are based on the packing framework of Garg and Könemann [14], which we review in this section.

One important feature of this packing framework, when interpreted in the setting of network flow, is that all computations are performed in the original network, and not in a residual network. Consequently, if the original network has special structure, (e.g., no flow-generating cycles, lossy network) then this property is retained throughout the algorithm and can be repeatedly exploited. In contrast, the residual network does not maintain such properties. The efficiency of our FPTAS's depends critically on the ability to work in the original network, and not in a residual network.

*The packing algorithm for traditional maximum flow.* The Garg-Könemann maximum flow algorithm can be best understood by considering the (exponential-size) path-formulation of the problem. Let  $\mathcal{P}$  denote the set of directed  $s$ - $t$  paths. The variable  $x(P)$  denotes the flow sent on path  $P \in \mathcal{P}$ . The maximum flow problem is to maximize  $\sum_{P \in \mathcal{P}} x(P)$  subject to  $\sum_{P: e \in P} x(P) \leq u(e)$ ,  $\forall e \in E$  and  $x(P) \geq 0$ . The dual LP is  $\min \{ \sum u(e)l(e) \mid \sum_{e \in P} l(e) \geq 1 \forall P \in \mathcal{P}, l(e) \geq 0 \}$ . This is the problem of assigning nonnegative lengths  $l(e)$  for each arc  $e$  so that the length of the shortest  $s$ - $t$  path is at least 1, and  $\sum u(e)l(e)$  is minimized. The length of arc  $e$  can be interpreted as the marginal cost of using up one unit of capacity on arc  $e$ .

Given length function  $l$ , define  $\alpha(l)$  as the length of the shortest  $s$ - $t$  path, and  $D(l) = \sum u(e)l(e)$ . The length function  $l$  that minimizes  $D(l)/\alpha(l)$  is an optimal dual solution after scaling by  $\alpha(l)$ .

The Garg-Könemann maximum flow algorithm starts with the zero flow and lengths  $l_0(e) = \delta/u(e)$ , where  $\delta = \Theta(m^{-1/\epsilon})$ . In iteration  $i$ , it finds the shortest  $s$ - $t$  path  $P$  using lengths  $l_{i-1}$ , and increases the current flow on each arc in this path by the bottleneck capacity  $v := \min\{u(e) \mid e \in P\}$ , i.e., the minimum *original* capacity of any arc in the path. In general, the updated flow will violate one or more arc capacity constraints since the augmentation amount is independent of residual capacities. We may obtain a feasible flow by dividing all flows by the maximum factor of violation of a capacity constraint. The length function is then updated by multiplying the length of each arc  $e \in P$  by  $1 + \epsilon \frac{v}{u(e)}$ . The algorithm terminates at the first iteration  $\tau$  with  $D(l_\tau) \geq 1$ . Garg and Könemann [14] prove that after scaling the final flow so that it is feasible for the primal problem, the resulting solution is within  $1 - 2\epsilon$  of the best dual solution found by the algorithm. Then, linear programming weak duality implies that the scaled flow is a  $2\epsilon$ -approximate maximum flow.

*General LP packing algorithm.* In order to use this algorithm to handle gain and loss factors, it is necessary to consider a more general packing problem, the packing linear program, and extend the algorithm to work in this setting. A *packing LP* is of the form  $\max \{c^T x \mid Ax \leq b, x \geq 0\}$  with all entries of  $m \times n$  matrix  $A$  nonnegative and all entries of vectors  $b$  and  $c$  positive. (Without loss of generality, all rows and columns of  $A$  contain at least one nonzero entry.) Note that the path formulation of the traditional maximum flow problem is a packing LP. The dual LP is  $\min \{b^T y \mid A^T y \geq c, y \geq 0\}$ . Denoting the  $h^{\text{th}}$  column of  $A$  by  $A(h)$ , this can be rewritten as  $\min \{b^T y \mid \frac{A(h)^T y}{c(h)} \geq 1$

$\forall 1 \leq h \leq n, y \geq 0$ ). Linear programming duality asserts that the optimal dual value equals the optimal value of the packing LP, and that any feasible dual solution has value greater than or equal to any feasible solution to the packing LP.

Given dual variable vector  $y$ , define  $\alpha(y) := \min_h \{A(h)^T y / c(h)\}$  and  $D(y) := b^T y$ . The  $y$  that minimizes  $D(y)/\alpha(y)$  also gives the optimal solution to the dual LP after scaling by  $\alpha(y)$ . In this packing setting, the Garg-Könemann algorithm starts with primal solution  $x = 0$ , and an infeasible dual solution,  $y_0(r) = \delta/b(r)$ ,  $1 \leq r \leq n$ , for an appropriately chosen  $\delta = \Theta(m^{-1/\epsilon})$ . At each iteration, it determines the most violated dual constraint, that is, the dual constraint that determines the current value of  $\alpha$ . It constructs a primal feasible solution that complements (in the complementary slackness sense) this dual solution. This primal feasible solution is not taken as the new primal solution, however. Instead, it is *added* to the current primal solution. Thus the new primal solution is likely infeasible, since it can violate the packing constraints. At the very end, the primal solution is scaled to be feasible. Specifically, the algorithm first determines the column  $q$  of  $A$  that minimizes  $A(h)^T y / c(h)$ . For this column  $q$ , it then determines a row  $p$  for which  $b(p)/A(p, q) \leq b(r)/A(r, q) \forall$  rows  $r$ , and increases  $x(q)$  by  $b(p)/A(p, q)$ . (Here and throughout,  $A(r, h)$  denotes the entry in the  $r^{\text{th}}$  row and  $h^{\text{th}}$  column of  $A$ .) The dual solution  $y_{i-1}$  is updated by setting  $y_i(r) = y_{i-1}(r)[1 + \epsilon \frac{b(p)/A(p, q)}{b(r)/A(r, q)}]$ . The algorithm terminates at the first iteration  $\tau$  with  $D(y_\tau) \geq 1$ . By our previous observation,  $y_\tau/\alpha(y_\tau)$  is a feasible solution to the dual packing LP of value  $\beta := D(y_\tau)/\alpha(y_\tau)$ . Let  $z_i$  be the value of primal solution constructed by the algorithm at the end of iteration  $i$ . By the above discussion,  $z_0 = 0$  and  $z_i = z_{i-1} + c(q)(b(p)/A(p, q))$ . Garg and Könemann [14] prove the following sequence of lemmas, using  $\delta = \frac{1+\epsilon}{[(1+\epsilon)m]^{1/\epsilon}}$ .

**Lemma 1 ([14], Claim 3.1).** *There is a feasible solution to the packing LP of value  $\frac{z_\tau}{\log_{1+\epsilon} \frac{1+\epsilon}{\delta}}$ .*

**Lemma 2 ([14], Theorem 3.1).** *The packing algorithm terminates in at most  $\frac{1}{\epsilon}m(1 + \log_{1+\epsilon} m)$  iterations.*

**Lemma 3 ([14], Theorem 3.1).** *Upon termination, the ratio of the primal feasible objective value to the optimal dual solution is at least  $(1 - \epsilon)^2$ .*

Let  $\mathcal{S}(m, n)$  be the run time of a subroutine to find the most violated dual constraint and the corresponding complementary primal feasible solution. Thus, a feasible solution to the packing LP whose objective is at least  $(1 - \epsilon)$  times the optimum can be found in  $\mathcal{O}(\epsilon^{-2}\mathcal{S}(m, n)m \log m)$  time.

## 2.5. Generalized shortest paths

To solve generalized flow problems using the packing framework described above, we will need a subroutine to solve the *generalized shortest path problem*: given an uncapacitated digraph  $G = (V, E)$  with a nonnegative length or cost function  $l: E \rightarrow \mathbf{R}_{\geq 0}$ , a gain function  $\gamma: E \rightarrow \mathbf{R}_{> 0}$ , a source node  $s \in V$  and a sink node  $t \in V$ , the goal is to send flow from  $s$  so that one unit of flow arrives at the  $t$  as cheaply as possible. In

this section we describe how to solve this problem efficiently in general networks, and networks without flow-generating cycles.

Formally, a generalized shortest path is an optimal solution  $g(v, w)$  to the following linear program. This reduces to the traditional shortest path problem if all gain factors are one.

$$\begin{aligned} \min \quad & \sum_{(v,w) \in E} l(v, w)g(v, w) \\ \sum_{(w,v) \in E} \gamma(w, v)g(w, v) - \sum_{(v,w) \in E} g(v, w) = & \begin{cases} 1 & v = t \\ 0 & v \in V \setminus \{s, t\} \end{cases} \\ \forall (v, w) \in E : g(v, w) \geq & 0. \end{aligned}$$

Note that the assumption that all arc lengths are nonnegative is not a restriction for our purposes, since the exponential length function that is used in the fractional packing framework is nonnegative.

*2.5.1. Generalized shortest paths in networks with flow generating cycles.* Optimality conditions for generalized flow problems are well-known [3]. Here, we review the structure of the optimal solution for the generalized shortest path problem. One possibility is that the solution sends flow only along a single simple  $s$ - $t$  path. If there are no flow-generating cycles, this is the only possibility. However, in networks with flow-generating cycles, there is a second possibility – the solution can send flow around a flow-generating cycle and then along a path to  $t$ . By sending one unit of flow around a cycle, more than one unit reaches the first node of the cycle. Thus, flow can be generated at any node of the cycle (typically at some cost), instead of at  $s$ . This combination of a simple flow-generating cycle and a simple path from a node on this cycle to the sink is called a *generalized augmenting path (GAP)*.

Existing polynomial combinatorial methods for solving the generalized shortest path problem with flow-generating cycles are all based on the Bellman-Ford algorithm. Extending this method to generalized flow appears to require additional care and complexity. The best known complexity bound is  $\tilde{O}(mn^2)$  due to Cohen and Megiddo [7] and Hochbaum and Naor [23]. Their algorithms are actually more general; they test the feasibility of a general two-variable-per-inequality linear system. All of these algorithms are based on Procedure 1, which determines whether the generalized shortest path value is bigger than, less than, or equal to a trial value  $L$ . Aspvall and Shiloach [4] give a  $\mathcal{O}(mn)$  time Bellman-Ford style algorithm for this procedure. Their algorithm exploits structure described by Shostak [36].

**Procedure 1 (Aspvall and Shiloach [4]).** *Let  $L^*$  denote the value of the generalized shortest path. Given  $L$ , it can be decided in  $\mathcal{O}(mn)$  time whether  $L = L^*$ ,  $L < L^*$ , or  $L > L^*$ .*

Recently, Oldham [30] proposed an algorithm for directly solving the generalized shortest path problem that matches the  $\tilde{O}(mn^2)$  complexity bound. His algorithm combines the Aspvall-Shiloach procedure with Megiddo's [28] parametric search.

**2.5.2. Generalized shortest paths in networks with no flow generating cycles.** In the case when there are no flow-generating cycles, the optimality conditions described in the previous section imply that the generalized shortest path is a simple  $s$ - $t$  path, since there can be no GAP's. Using the procedure described in Sect. 2.2, we first transform the network into a lossy network. For lossy networks, we now describe a more efficient Dijkstra-like algorithm, similar to that proposed by Charnes and Raike [6], to find a generalized shortest path. The difference between this approach and the approach required in the setting with flow-generating cycles is analogous to the difference between the traditional shortest path problem with and without negative cost arcs. As a result, these faster methods do not extend to networks with flow-generating cycles.

For each node  $v$ , we maintain a distance  $\pi(v)$  in a priority queue. Upon termination,  $\pi(v)$  is the cheapest cost of sending flow from  $s$  so that one unit arrives at  $v$ , given an unlimited and free supply at the source. Our algorithm is identical to Dijkstra's, except in the way the distances are updated. We examine the cheapest node  $v$  and delete it from the priority queue. We update the distances of all its neighbors. Suppose the unit cost of getting flow at  $v$  is  $\pi(v)$ . Then, obtaining one unit of flow at  $v$  and shipping to  $w$  along arc  $(v, w)$  costs  $\pi(v) + l(v, w)$ . But, only  $\gamma(v, w)$  units would then arrive at  $w$ . So, we should scale everything by the gain factor. This leads to updating  $\pi(w)$  with  $\min\{\pi(w), \frac{\pi(v)+l(v,w)}{\gamma(v,w)}\}$ . Using Fibonacci heaps, as in Fredman and Tarjan's [13] implementation of Dijkstra's algorithm, leads to the following theorem.

**Theorem 1.** *The generalized shortest path problem in lossy networks with nonnegative lengths can be solved in  $\mathcal{O}(m + n \log m)$  time.*

### 3. Generalized maximum flow

In this section, we first describe a fast and simple FPTAS for generalized maximum flow in networks with no flow-generating cycles. It runs in  $\tilde{\mathcal{O}}(\epsilon^{-2}m^2)$  time, which, even for this well-studied problem, is an improvement over previous work. For any constant  $\epsilon > 0$ , it is faster by a factor of  $n$  than the previous best known strongly-polynomial bound in [32]. It also dominates the previous best weakly-polynomial bound of  $\tilde{\mathcal{O}}(\log(1/\epsilon)(m^2 + mn \log \log M))$  in [34, 37] for sparse networks or when  $M$  is very large.

Next we discuss how to turn this FPTAS into an exact algorithm using the “gain-scaling” technique in [37]. This leads to a  $\tilde{\mathcal{O}}(\log(1/\epsilon)(m^2 + mn \log \log M))$  approximation scheme, which exactly matches the running time in [34, 37]. In this framework, the run time is proportional to  $\log(1/\epsilon)$  instead of  $\epsilon^{-2}$ , allowing us to solve the generalized maximum flow problem to optimality by using a sufficiently small value of  $\epsilon$ . We note that our gain-scaling algorithm also works in networks with flow-generating cycles, but, as in [17, 34, 37], we first need to perform an  $\tilde{\mathcal{O}}(\min\{m^2n, mn^2 \log M\})$  preprocessing step to “cancel” all flow-generating cycles.

#### 3.1. Generalized max flow packing algorithm

We formulate the generalized maximum flow problem in lossy networks as a packing LP (**P**). In Sect. 2.5.2, we explained that in lossy networks the generalized shortest path

is a simple  $s$ - $t$  path, so we need not worry about GAP's. Let  $\mathcal{P}$  denote the set of all directed  $s$ - $t$  paths. In the generalized flow setting, variable  $x(P)$  represents the amount of flow that reaches  $t$  on path  $P \in \mathcal{P}$ . Note that this does not typically equal the amount of flow leaving  $s$ . There is a capacity constraint for each arc. This is straightforward to model, but we first need some notation since our decision variables only implicitly determine how much flow goes through a given arc. Given an  $s$ - $t$  path  $P = \{e_1, \dots, e_r\}$ , we define  $\gamma_P(e_q) := 1 / \prod_{i=q}^r \gamma(e_i)$ . It is the amount of flow that must be sent into arc  $e_q$  in order to deliver one unit of flow at  $t$  using path  $P$ .

$$\begin{aligned} \max \quad & \sum_P x(P) \\ \forall e \in E : \quad & \sum_{P: e \in P} \gamma_P(e) x(P) \leq u(e) \\ \forall P \in \mathcal{P} : \quad & x(P) \geq 0. \end{aligned} \tag{P}$$

The linear programming dual **(D)** is to find an assignment of nonnegative arc lengths or costs  $l$  so that  $\sum_e u(e)l(e)$  is minimized. The constraints require that the marginal cost (using costs  $l$ ) of getting one unit of flow to reach  $t$ , using any  $s$ - $t$  path, is at least one.

$$\begin{aligned} \min \quad & \sum_e u(e)l(e) \\ \forall P \in \mathcal{P} : \quad & \sum_{e \in P} \gamma_P(e)l(e) \geq 1 \\ \forall e \in E : \quad & l(e) \geq 0. \end{aligned} \tag{D}$$

We interpret the packing algorithm of [14] for **(P)**. The algorithm maintains a length function  $l(e)$  that is exponential in the total flow going through that arc. Initially, we set  $l(e) = \frac{\delta}{u(e)}$ , where  $\delta = \Theta(m^{-1/\epsilon})$  is chosen as in Sect. 2. Although **(D)** has an exponential number of constraints, finding the most violated dual constraint corresponds to finding a *generalized shortest path*  $P$ . Using path  $P$  is the cheapest way to send flow from  $s$  so that one unit arrives at  $t$ . Such a subroutine is described in Sect. 2.5.2. Once path  $P$  is obtained, the algorithm sends as much flow as possible along  $P$ , without violating the capacity constraints in the *original* network. That is, it sends flow from  $s$  along  $P$  so that  $v = \min_{e \in P} \frac{u(e)}{\gamma_P(e)}$  units arrive at  $t$ . We update the length function: for each arc  $e \in P$ , we multiply its length by a factor of  $(1 + \frac{\epsilon v}{u(e)/\gamma_P(e)})$ . The algorithm terminates when the dual objective value  $\sum_e u(e)l(e) \geq 1$ . Lemmas 1, 2, and 3 imply that after  $\mathcal{O}(\epsilon^{-2}m \log m)$  iterations, we obtain an  $\epsilon$ -approximate generalized maximum flow. Combining this with Theorem 1 yields the following theorem.

**Theorem 2.** *An  $\epsilon$ -approximate generalized maximum flow in networks with no flow-generating cycles can be computed in  $\mathcal{O}(\epsilon^{-2}m(m + n \log m) \log n)$  time.*

### 3.2. An exact algorithm

We extend our algorithm to *exactly* solve the generalized maximum flow problem, even in networks with flow-generating cycles. To do this efficiently, we use “error-scaling.” The basic idea is to run the packing algorithm, say with  $\epsilon = 1/2$ , and obtain a  $1/2$ -approximate generalized maximum flow  $g$ . Then, we repeatedly run the packing algorithm in the residual network  $G_g$ , adding the resulting flow to  $g$ . Each iteration captures at least  $1/2$  of the remaining flow possible, so the optimality gap geometrically

decreases to zero. We obtain an  $\epsilon$ -approximate flow after  $\log(1/\epsilon)$  iterations. If  $\epsilon$  is sufficiently small, say  $M^{-3m}$ , then the  $\epsilon$ -approximate flow can be efficiently “rounded” to an optimal flow [17].

The main flaw in this approach is that the residual network may contain flow-generating cycles, even if the original network did not. Recall, our fast generalized shortest path subroutine does not work if there are flow-generating cycles. To overcome this obstacle, before running the packing algorithm, we first *cancel residual flow-generating cycles*, as described in [17]. That is, we repeatedly send flow around flow-generating cycle until (at least) one arc becomes saturated. In the process, excess (but no deficit) is created at one node of the cycle. If the cycles are chosen carefully (e.g., minimum mean cost cycles using costs  $c = -\log \gamma$ ) then all flow-generating cycles can be canceled in polynomial time. Canceling flow-generating cycles appears to be more expensive than canceling negative cost cycles. Goldberg, Plotkin, and Tardos [17] give a  $\tilde{O}(mn^2 \log M)$  time algorithm. After canceling all flow-generating cycles, the resulting residual network may contain nodes with excess. Before applying the original packing algorithm to this network, we add a new unit gain arc from  $s$  to each excess node, make its capacity equal to the node’s excess, and remove the excess from the network. This allows the excess created from canceling flow-generating cycles to be subsequently sent to the sink by the algorithm along source to sink paths.

This approach leads to an  $\tilde{O}(\log(1/\epsilon)mn^2 \log M)$  algorithm. The complexity can be significantly improved using the gain-scaling technique of Tardos and Wayne [37]. We sketch the method here, but the reader is referred to [37] for more details. The bottleneck computation is canceling flow-generating cycles. To reduce this bottleneck, we first round down all of the gain factors in the original network to powers of a certain base  $b > 1$ . In this rounded network, flow-generating cycles can be canceled more efficiently. Moreover, the rounding causes only a modest degradation in the approximation guarantee. In the next iteration, the gain factors are re-rounded to powers of a smaller base in order to refine the approximation guarantee. By slowly rounding the gain factors, the amortized complexity of canceling flow-generating cycles is reduced from  $\tilde{O}(mn^2 \log M)$  to  $\tilde{O}(mn \log \log M)$  per iteration. After  $\mathcal{O}(\log(1/\epsilon))$  iterations, the flow is  $\epsilon$ -optimal.

**Theorem 3.** *The packing algorithm, in conjunction with gain-scaling, computes an  $\epsilon$ -approximate generalized maximum flow in  $\tilde{O}(\log \epsilon^{-1}(m^2 + mn \log \log M))$  time.*

This exactly matches the best known complexity bound of Radzik [34] and Tardos and Wayne [37].

#### 4. Generalized minimum cost flow

In this section, we describe how to extend the FPTAS’s for generalized maximum flow problems to versions with nonnegative arc costs. In Sect. 4.1 we consider the budget constrained version, where the total shipping cost is constrained to be at most some fixed budget. The budget constrained problem arises as a subproblem in finding a maximum flow of minimum cost. In Sect. 4.2 we describe a FPTAS for the minimum cost maximum flow problem in networks with no flow-generating cycles; in Sect. 4.3, we describe a FPTAS for the version with flow-generating cycles.

#### 4.1. Generalized maximum flow with budget constraint

We first consider the generalized maximum flow problem with a budget constraint. Each arc  $e$  has a nonnegative cost  $c(e)$ , representing the unit cost of shipping one unit of flow into  $e$ . Given budget  $B$  we seek a generalized maximum flow  $x$  among all flows that have cost at most  $B$ , i.e.,  $\sum_{P \in \mathcal{P}} \sum_{e \in P} c(e) \gamma_P(e) x(P) \leq B$ . Like the generalized maximum flow problem, this is also a packing LP. The dual LP ( $\mathbf{D}'$ ) for this problem is

$$\begin{aligned} \min \quad & \sum_e u(e) l(e) + B\phi \\ & \sum_{e \in P} \gamma_P(e) [l(e) + c(e)\phi] \geq 1 \quad \forall P \\ & l(e) \geq 0 \quad \forall e \\ & \phi \geq 0. \end{aligned} \tag{\mathbf{D}'}$$

We maintain a dual variable  $l(e)$  for each arc and a dual variable  $\phi$ . Initially we set  $l(e) = \delta/u(e)$  and  $\phi = \delta/B$ . The problem of finding a most violated dual inequality for ( $\mathbf{D}'$ ) is the problem of finding a generalized shortest path  $P$  using length function  $l(e) + c(e)\phi$ . Thus, with slight modifications, the same algorithm we describe in Sect. 2.5 works here. To determine how much additional flow to send on  $P$ , we need to find the row of matrix  $A$  that constrains  $P$  the most. This is either determined as in Sect. 3, or by the budget constraint. That is, we send flow so that by  $v = \min\{\min_{e \in P} \frac{u(e)}{\gamma_P(e)}, \frac{B}{\sum_{e \in P} c(e)\gamma_P(e)}\}$  units arrive at  $t$ . We update the length function exactly as before: for each arc  $e \in P$ , we multiply its length by a factor of  $(1 + \frac{\epsilon v}{u(e)/\gamma_P(e)})$ . We update the dual variable corresponding to the budget constraint by  $\phi_i = \phi_{i-1} (1 + \frac{\epsilon v}{B/\sum_{e \in P} c(e)\gamma_P(e)})$ . There are  $\mathcal{O}(\epsilon^{-2} m \log m)$  iterations by Lemma 2.

In lossy networks, each iteration requires  $\mathcal{O}(m + n \log n)$  time to find a generalized shortest path using Theorem 1. In networks with no flow generating cycles, we must first relabel the network. However, once we relabel it, the network remains lossy for the remainder of the algorithm. The run time for relabeling is dominated by the remainder of the algorithm.

**Theorem 4.** *An  $\epsilon$ -approximate generalized maximum flow of cost at most  $B$  in networks with no flow generating cycles can be computed in  $\mathcal{O}(\epsilon^{-2} m \log m(m + n \log n))$  time.*

Oldham derived a corresponding result for networks with flow-generating cycles using Procedure 1 in conjunction with Megiddo's parametric search [28]. We give an improved theorem and procedure for networks with flow-generating cycles in Sect. 4.3.

**Theorem 5 (Oldham [30]).** *An  $\epsilon$ -approximate generalized maximum flow of cost at most  $B$  in networks with flow-generating cycles can be computed in  $\tilde{\mathcal{O}}(\epsilon^{-2} m \log m(mn^2 \log m))$  time.*

#### 4.2. Generalized minimum cost maximum flow in networks with no flow generating cycles

We describe how to find an approximate generalized minimum cost flow in lossy networks. Recall, if a maximum flow of minimum cost delivers  $U^*$  units of flow at the

sink and has cost  $B^*$ , then an  $\epsilon$ -approximate minimum cost maximum flow is defined to deliver at least  $(1 - \epsilon)U^*$  units of flow at the sink and costs no more than  $B^*$ . To find such a flow, we can binary search for the optimal budget  $B^*$ , and at each step find an  $\epsilon$ -approximate maximum flow (using Theorem 4) that does not exceed the given budget. However, the optimal cost of a generalized flow can be exponentially small,  $\Theta(M^{-n})$ , since the amount of the flow reaching the sink depends on the product of gains of arcs along the path. Therefore, standard binary search could increase the run time by a factor of  $n \log M$ .

To reduce this run time, suppose we could estimate  $B^*$  within a  $(1 + \epsilon)$  factor, say by  $B^* \leq B \leq B^*(1 + \epsilon)$ . Then, we can use  $B$  in the budget constraint. By Theorem 4, we can find a flow of value  $(1 - \epsilon)$  times the optimum among flows that have cost at most  $B$ . This flow might exceed the optimum budget  $B^*$ , so we scale it down by a factor of  $(1 + \epsilon)$ . Now, the scaled flow has value at least  $(1 - \epsilon)/(1 + \epsilon) \geq 1 - 2\epsilon$  and has cost at most  $B^*$ .

To find a suitable approximation to  $B^*$ , we use the geometric-mean binary search technique of Hassin [22] (§4). Given a lower bound  $LB$  and an upper bound  $UB$  on the desired value  $B^*$ , conventional binary search uses the arithmetic mean  $(LB + UB)/2$  and shrinks the difference  $UB - LB$  in half. Our goal is actually to decrease the *ratio* of the two endpoints  $UB/LB$ . Using the geometric mean  $\sqrt{(LB)(UB)}$ , each iteration halves the log of the ratio, i.e., the ratio is “square-rooted.” Thus, after  $\log \log_{1+\epsilon}(UB/LB) = \mathcal{O}(\log(\epsilon^{-1}) + \log \log(UB/LB))$  calls to the budget constrained generalized maximum flow algorithm, the ratio between the search interval endpoints is at most  $(1 + \epsilon)$ . To compute the geometric mean, we need to take square roots. But again, an approximation suffices, and traditional techniques (e.g., Newton’s method) can be used to approximately compute square roots.

**Theorem 6.** *An  $\epsilon$ -approximate generalized minimum cost maximum flow in networks with no flow generating cycles can be computed in  $\tilde{\mathcal{O}}(\epsilon^{-2}m^2(\log \epsilon^{-1} + \log \log M))$  time.*

*Proof.* We begin by transforming the network into a lossy network in  $\mathcal{O}(nm)$  time. In each search iteration, we approximately solve a generalized maximum flow with budget constraint problem. By Theorem 4, this requires  $\mathcal{O}(\epsilon^{-2}m \log m(m + n \log n))$  time per search iteration. There are  $\mathcal{O}(\log \epsilon^{-1} + \log \log(UB/LB))$  geometric-mean binary search iterations. Since each arc has capacity and cost at most  $M$ , the value of the generalized minimum cost maximum flow is at most  $mM^2$ . Since each gain factor is at least  $1/M$ , the value is at least  $1/M^n$ , assuming it is positive. If it is zero, this case can be easily detected and solved using Theorem 2. Thus, the number of search iterations is  $\mathcal{O}(\log \frac{1}{\epsilon} + \log \log m + \log n \log \log M)$ . □

#### 4.3. Networks with flow-generating cycles

We design a faster and simpler FPTAS for finding a generalized maximum flow of minimum cost in networks with flow-generating cycles. The packing algorithm requires a subroutine to find a generalized shortest path in networks with flow-generating cycles.

In Theorem 5, this subroutine is Procedure 1 in conjunction with Megiddo’s parametric search [28]. This is essentially Oldham’s [30] algorithm. Adapting and extending ideas in [11], we propose a simpler alternative to parametric search, and improve the run time by a factor of  $n$ .

The first fact we use is that, in the packing framework, it is not necessary to find the most violated dual constraint (i.e., the exact generalized shortest path). Instead, it suffices to find a nearly most violated dual constraint (i.e., a path of length at most  $(1 + \epsilon)L^*$ , where  $L^*$  is the length of the generalized shortest path). This fact has been used before in other  $\epsilon$ -approximate packing algorithms (see [20, 27, 31]) and is proved for the Garg-Könemann packing algorithm in [11].

To take advantage of this fact, we maintain a lower bound  $L$  on the true generalized shortest path length  $L^*$ . We use  $(1 + \epsilon)L$  as our guess for the shortest path length in Procedure 1. If Procedure 1 determines that the shortest path is greater than  $(1 + \epsilon)L$ , then we can update our lower bound  $L$  to  $(1 + \epsilon)L$ . Otherwise, Procedure 1 outputs a generalized path of length at most  $(1 + \epsilon)L$ . Since  $L$  is a lower bound on the generalized shortest path, the value of the path is within a  $(1 + \epsilon)$  factor of the shortest such path. We use this path in the packing framework. The improvement in the running time (over parametric search) comes from showing that the final value of  $L$  is at most  $(1 + \epsilon)/\delta$  times the starting value of  $L$ , and hence  $L$  increases at most  $\log_{1+\epsilon}((1 + \epsilon)/\delta)$  times.

**Lemma 4.** *The final value of  $L$  is at most  $(1 + \epsilon)/\delta$  times the starting value of  $L$ .*

*Proof.* Initially,  $L$  is determined by some generalized shortest path  $P_0$  using  $l_0 = \delta/u$ . That is,  $L_0 = \min_P \sum_{e \in P} \gamma_P(e)l_0(e)$ , where the minimum is determined by  $P_0$ . At the final iteration  $\tau$ , we have that  $L_\tau \leq \sum_{e \in P} \gamma_P(e)l(e)$  for all  $P \in \mathcal{P}$ . In particular, this inequality must hold for path  $P_0$ .

Since we terminate the algorithm when the dual objective function reaches 1, we have that  $l_\tau(e) \leq (1 + \epsilon)/u(e)$ . Hence the ratio  $l_\tau/l_0$  is at most  $(1 + \epsilon)/\delta$ . Combining this with the conclusion of the preceding paragraph yields the lemma.  $\square$

**Theorem 7.** *An  $\epsilon$ -approximate generalized maximum flow of cost at most  $B$  in networks with flow-generating cycles can be computed in  $\mathcal{O}(\epsilon^{-2}m^2n \log m)$  time.*

*Proof.* We analyze the number of calls to Procedure 1, which is the bottleneck computation. If the generalized shortest path length is within a  $(1 + \epsilon)$  factor of our current lower bound  $L$ , we find a path with a length that is within a  $(1 + \epsilon)$  factor of the shortest path, with just a single call to Procedure 1. Lemma 2 implies that this happens  $\mathcal{O}(\epsilon^{-2}m \log m)$  times. Otherwise, we increase the value of  $L$  by a  $(1 + \epsilon)$  factor. Lemma 4 implies this happens at most  $\mathcal{O}(\epsilon^{-2} \log m)$  times, given our choice of  $\delta$ .  $\square$

Using geometric binary search as in Sect. 4.2, we obtain a FPTAS for finding a generalized maximum flow of minimum cost.

**Theorem 8.** *An  $\epsilon$ -approximate generalized minimum cost maximum flow in networks with flow-generating cycles can be computed in  $\tilde{\mathcal{O}}(\epsilon^{-2}m^2n(\log \epsilon^{-1} + \log \log M))$  time.*

## 5. Generalized maximum multicommodity flow

We present FPTAS's for the generalized maximum multicommodity flow problem, both for networks with and without flow-generating cycles. The problem falls into the packing framework, and a straightforward analysis leads to a  $\tilde{O}(\epsilon^{-2}km^2)$  FPTAS for networks with no flow-generating cycles. Interestingly, the fastest known FPTAS for the non-generalized version of the problem requires  $\tilde{O}(\epsilon^{-2}m^2)$  time [11]. After presenting the straightforward analysis, we combine the ideas in [11] with Lemma 4 to exactly match the run time of the non-generalized FPTAS. We also use the same idea to obtain a fast FPTAS in networks with flow-generating cycles.

The path formulation of the generalized maximum multicommodity flow problem is also a packing LP. Let  $\mathcal{P}$  denote the set of all directed paths from  $s_i$  to  $t_i$  for all commodities  $1 \leq i \leq k$ .

$$\begin{aligned} \max \quad & \sum_{P \in \mathcal{P}} x(P) \\ \forall e \in E : \quad & \sum_{P: e \in P} \gamma_P(e)x(P) \leq u(e) \\ \forall P \in \mathcal{P} : \quad & x(P) \geq 0. \end{aligned}$$

For the single commodity case, finding a most violated dual constraint corresponds to finding a generalized shortest path from  $s$  to  $t$ . For the multicommodity case, we need to find the generalized shortest path from  $s_i$  to  $t_i$  among all commodities  $1 \leq i \leq k$ . This can be accomplished in lossy networks or networks with flow-generating cycles using  $k$  single commodity generalized shortest path computations. The number of iterations of the packing algorithm remains  $\mathcal{O}(\epsilon^{-2}m \log m)$  by Lemma 2.

For lossy networks, the subroutine to find the most violated dual constraint in one iteration consists of  $k$  of our modified Dijkstra computations. Depending on the value of  $k$ , this can be done more efficiently. It suffices to perform an all-pairs generalized shortest path computation, e.g., with only  $n$  Dijkstra computations instead of  $k$ .

**Theorem 9.** *An  $\epsilon$ -approximate solution to the generalized maximum multicommodity flow problem in networks with no flow-generating cycles can be computed in  $\mathcal{O}(\epsilon^{-2} \min\{k, n\} m \log m(m + n \log n))$  time.*

We improve upon this theorem by incorporating the essential idea in [11]: avoid computing a shortest path for each commodity in each iteration, by sticking with a single commodity as long as the shortest path for that commodity is at most  $(1 + \epsilon)L$ . When this no longer holds, the algorithm moves on to the next commodity. In this manner, the commodities are cycled through once per update to  $L$ . Thus there is only one shortest path computation per iteration, plus  $k$  shortest path computations per update to  $L$ . By grouping commodities by common source, this  $k$  can be replaced with  $n$ .

**Theorem 10.** *An  $\epsilon$ -approximate solution to the generalized maximum multicommodity flow problem in lossy networks can be computed in  $\mathcal{O}(\epsilon^{-2}m \log m(m + n \log n))$  time.*

We note that all our multicommodity formulation can easily accommodate distinct gain factors for distinct commodities. In this case, we need to perform a separate generalized shortest path computation for each commodity using the same costs generated by

the exponential length function, but different gain factors. This would prevent grouping of commodities by common source node.

For flow-generating cycles, we combine the above ideas with Procedure 1 embedded in the scaling framework described in Sect. 4.3. Using a similar analysis, we obtain the following theorem.

**Theorem 11.** *An  $\epsilon$ -approximate solution to the generalized maximum multicommodity flow problem in networks with flow-generating cycles can be computed in  $\mathcal{O}(\epsilon^{-2}m^2n \log m)$  time.*

## 6. Generalized concurrent flow

Unlike the single commodity flow problem, the natural formulations of the traditional multicommodity concurrent flow problems are not packing LP's. For these problems, Garg and Könemann modify their approximate maximum flow algorithm to handle multiple commodities [14]. This modification assumes  $A$  is a 0-1 matrix. We show that the algorithm for the packing LP can be modified to extend to a packing LP with multiple commodities in a similar fashion, and thus we can provide approximation algorithms for generalized concurrent flow problems.

### 6.1. LP formulation of the generalized concurrent flow problem

We use the path formulation of the generalized maximum concurrent flow problem. The LP for the generalized maximum concurrent flow problem and its dual are given below. As with the maximum multicommodity flow problem, this formulation easily accommodates distinct gain/loss factors for distinct commodities. For simplicity, we assume the gain function is the same for all commodities.

$$\begin{array}{ll}
 \text{(P'')} & \text{(D'')} \\
 \max \lambda & \min \sum_e u(e)l(e) \\
 \forall e : \sum_{P:e \in P} \gamma_P(e)x(P) \leq u(e) & \forall P (P \in \mathcal{P}_j) : \sum_{e \in P} \gamma_P(e)l(e) \geq z_j \\
 \forall j : \sum_{P \in \mathcal{P}_j} x(P) - \lambda d_j \geq 0 & \sum_j d_j z_j \geq 1 \\
 \forall P : x(P) \geq 0 & \forall e : l(e) \geq 0 \\
 \lambda \geq 0. & z_j \geq 0.
 \end{array}$$

This path formulation is a special case of the  $k$ -commodity packing problem described next.

### 6.2. The $k$ -commodity packing problem

The  $k$ -commodity packing problem is a somewhat more general form of the packing LP. We develop a FPTAS for the problem and will use it in the next two sections to solve

generalized multicommodity flow problems. The  $k$ -packing problem is:  $\max\{\lambda \mid Ax \leq b; -Kx + d\lambda \leq 0; x, \lambda \geq 0\}$  where  $A \in \mathbf{R}_{\geq 0}^{m \times n}$ ,  $c \in \mathbf{R}_{> 0}^n$ ,  $b \in \mathbf{R}_{> 0}^m$ ,  $d \in \mathbf{R}_{> 0}^k$ , and  $K$  is a  $k \times n$  block diagonal matrix in which each block consists of exactly one row containing all ones. The variables corresponding to columns with non-zero entries in block  $j$  are referred to as *commodity  $j$  variables*, and this set of columns is denoted by  $C_j$ . The dual of the  $k$ -commodity packing problem is  $\min\{b^T y \mid A^T y - K^T z \geq 0; d^T z \geq 1; y, z \geq 0\}$ .

It is straightforward to see that the generalized concurrent flow problem is a special case of  $k$ -commodity packing: in this case,  $A$  is the arc-path incidence matrix weighted by the arc gains,  $b$  is the arc capacity vector,  $K(j, P) = 1$  if  $P$  is a path for commodity  $j$  and 0 otherwise, and  $d$  is the vector of commodity demands. The  $k$ -commodity packing problem is more general since it allows matrices  $A$  whose support is more general than that of an incidence matrix. We start by describing and analyzing an FPTAS for the  $k$ -commodity packing problem. In Sect. 6.3, we then explain how this specializes in the case of generalized concurrent flows.

**6.2.1. The algorithm.** Our FPTAS for the  $k$ -commodity packing problem works in phases, and each phase consists of  $k$  iterations. In the  $j^{\text{th}}$  iteration of the  $i^{\text{th}}$  phase, we increase the total value of commodity  $j$  variables by  $d_j$ . Each variable has a corresponding *increment amount* that is unchanged throughout the algorithm. The increment amount of variable  $x(h)$  equals  $\min_r\{b(r)/A(r, h)\}$ . Each iteration consists of a sequence of steps. In any one step, a commodity  $j$  variable is increased by the minimum of its increment amount and  $d'_j$ , where  $d'_j$  is the difference between  $d_j$  and the total amount that commodity  $j$  variables have been increased so far in this iteration. The resulting primal solution  $x$  is likely not feasible. We can make  $x$  feasible by scaling it by the largest factor of violation of an inequality in the system  $Ax \leq b$ . If  $k = 1$ , then the algorithm we describe here is essentially the packing algorithm described in Sect. 2.4.

The algorithm starts with  $x \equiv 0$  and with dual variables  $y_0(r) = \delta/b(r)$ ,  $1 \leq r \leq m$ . Given any  $y$ , we may find a  $z$  and scale both  $y$  and  $z$  so that we obtain a feasible solution to the dual in the following manner: We set  $z(j) = \min_{h \in C_j} A(h)^T y$ ,  $1 \leq j \leq k$ , so that  $A^T y - K^T z \geq 0$  is always satisfied. Then, we scale  $y$  and  $z$  by  $d^T z$ , so that the last dual inequality is also satisfied. Thus, we explicitly maintain  $y$  and not  $z$  below.

Any one step looks a lot like the packing algorithm described in Sect. 2.4. In step  $s$  for commodity  $j$ , we find a most violated dual constraint  $q$  among the constraints corresponding to primal commodity  $j$  variables. We next determine the increment amount of primal variable  $x(q)$ , and set the actual increment  $v$  to be the minimum of this and the remaining demand  $d'_j$ . After incrementing  $x(q)$  by  $v$ ,  $y$  is updated by setting  $y(r) = y(r)(1 + \epsilon \frac{v}{b(r)/A(r, q)})$ . This algorithm is summarized in Fig. 2.

Let  $\alpha(y, z) = \sum_j d_j z(j)$  and  $D(y) = b^T y$ . The algorithm stops as soon as  $D(y) \geq 1$ . We extend the analysis in [14] to show that this algorithm leads to approximately optimal solutions for the  $k$ -commodity packing problem.

**6.2.2. Approximation guarantee.** Let  $t$  be the final phase of the algorithm. In the first  $t - 1$  phases, the algorithm increases commodity  $j$  variables by a total of  $(t - 1)d(j)$  units, possibly violating the packing constraints. After scaling the final  $x$  to obey the packing constraints, let  $\lambda$  the maximum value such that  $Kx \geq d\lambda$ .

```

K-CommodityPacking ( $A, b, K$ )
Input : matrices:  $m \times n$   $A$ ,  $m \times 1$   $b$ ,  $k \times n$   $K$ 
Output : primal and dual (infeasible) solutions  $x$  and  $y$ 

Initialize  $y(r) = \delta/b(r) \forall r$ ,  $D(y) = m\delta$ ,  $x \equiv 0$ .
while  $b^T y < 1$  do
  for  $j = 1$  to  $k$  do
     $d'_j = d_j$ 
    while  $d'_j > 0$  and  $b^T y < 1$ 
       $q \leftarrow \operatorname{argmin}_{h \in C_j} A(h)^T y$ 
       $p \leftarrow \operatorname{argmin}_r b(r)/A(r, q)$ 
       $v = \min\{d'_j, b(p)/A(p, q)\}$  /* increment amount */
       $x(q) \leftarrow x(q) + v$  /* primal update */
       $d'_j \leftarrow d'_j - v$  /* remaining demand of commodity  $j$  */
       $y(r) \leftarrow y(r)(1 + \epsilon \frac{v}{b(r)/A(r, q)})$ ,  $\forall r$  /* dual update */
    If  $b^T y \geq 1$ , stop. Return  $(x, y)$  scaled to be feasible.

```

**Fig. 2.**  $k$ -commodity packing algorithm

**Lemma 5.**  $\lambda \geq \frac{t-1}{\log_{1+\epsilon} \frac{1}{\delta}}$ .

*Proof.* If the primal solution is not feasible, it is because there is a violated packing constraint  $\sum_h A(r, h)x(h)/b(r) \leq 1$ . When the algorithm increases  $x(q)$  by  $b(p)/A(p, q)$ , the left hand side of this constraint increases by  $v := \frac{A(r, q)b(p)}{b(r)A(p, q)}$ . At the same time, dual variable  $y(r)$  is multiplied by  $1 + \epsilon v$ . By the choice of  $p$ ,  $v \leq 1$  and thus each increase in the left hand side of the  $r^{\text{th}}$  constraint by 1 causes  $y(r)$  to be multiplied by at least  $1 + \epsilon$ . Since  $D(t-1) < 1$ ,  $y_{t-1}(r) < 1/b(r)$ . Since  $y_0(r) = \delta/b(r)$ , the value of the left hand side of the  $r^{\text{th}}$  constraint after  $t-1$  phases is at most  $\log_{1+\epsilon} \frac{1}{\delta}$ . This holds for all primal constraints, and hence scaling the primal solution obtained after  $t-1$  phases by  $\log_{1+\epsilon} \frac{1}{\delta}$  satisfies the packing constraints. Scaling  $\lambda$  by the same value maintains the validity of the commodity constraints.  $\square$

**Lemma 6.** For  $\delta = (m/(1-\epsilon))^{-1/\epsilon}$ , the ratio of the optimal dual solution to the primal feasible solutions obtained by the algorithm is  $\theta \leq (1-\epsilon)^{-3}$ .

*Proof.* Let  $y_{ij}^s$  and  $z_{ij}^s$  be the dual variable setting at the end of the  $s^{\text{th}}$  step in the  $j^{\text{th}}$  iteration of the  $i^{\text{th}}$  phase. Let  $v_{ij}^s$  denote the increment amount in this step. Iteration  $j$  ends at the step  $\pi$  when  $d'_j = 0$ . We will let  $y_{ij}$  and  $z_{ij}$  denote the value of  $y$  and  $z$  at the start of iteration  $j+1$ ,  $D(i) := D(y_{ik}^\pi)$  and  $\alpha(i) = \alpha(y_{ik}^\pi, z_{ik}^\pi)$ . At the end step  $s$  of iteration  $j$  we have

$$\begin{aligned}
 D(y_{ij}^s) &= \sum_r b(r)y_{ij}^s(r) = \sum_r b(r)y_{ij}^{s-1}(r) + \epsilon v_{ij}^s \sum_r a_{rq}y_{ij}^{s-1}(r) \\
 &= D(y_{ij}^{s-1}) + \epsilon v_{ij}^s A_q^T y_{ij}^{s-1} \\
 &= D(y_{ij}^{s-1}) + \epsilon v_{ij}^s z_{ij}^{s-1}(j).
 \end{aligned}$$

Note that  $y$  is monotone increasing throughout the algorithm. This implies that  $z$  is also. We have

$$D(y_{ij}^s) \leq D(y_{ij}^{s-1}) + \epsilon v_{ij}^s z_{ij}^\pi(j)$$

Using the fact that  $\sum_s v_{ij}^s = d(j)$ ,

$$D(y_{ij}^\pi) = D(y_{ij}) \leq D(y_{i,j-1}) + \epsilon d(j) z_{ij}^\pi(j).$$

Summing over all iterations in a phase, we have

$$D(y_{ik}) \leq D(y_{i0}) + \epsilon \alpha(y_{ik}, z_{ik}),$$

or, rewriting,

$$D(i) \leq D(i-1) + \epsilon \alpha(i).$$

Let  $\beta$  be the optimal dual value. Thus  $\beta \leq \frac{D(i)}{\alpha(i)}$  which is the value of the dual feasible solution corresponding to  $y_{ik}/\alpha(i)$ . As in [14], we start with the assumption that  $\beta \geq 1$ . We remove this assumption later. Thus,

$$D(i) \leq \frac{D(i-1)}{1 - \epsilon/\beta}.$$

Since  $D(0) = m\delta$  and  $\beta \geq 1$ , for  $i \geq 1$

$$D(i) \leq \frac{m\delta}{(1 - \epsilon/\beta)^i} = \frac{m\delta}{1 - \epsilon/\beta} \left(1 + \frac{\epsilon}{\beta - \epsilon}\right)^{i-1} \leq \frac{m\delta}{1 - \epsilon/\beta} e^{\frac{\epsilon(i-1)}{\beta(1-\epsilon)}}.$$

The algorithm stops at the first phase  $t$  for which  $D(t) \geq 1$ . Thus

$$1 \leq D(t) \leq \frac{m\delta}{1 - \epsilon/\beta} e^{\frac{\epsilon(t-1)}{\beta(1-\epsilon)}},$$

and

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

The ratio of the values of the optimal dual solution to the primal feasible solution obtained is  $\theta := \frac{\beta}{t-1} \log_{1+\epsilon} 1/\delta$ . Setting  $\delta = (m/(1-\epsilon))^{-1/\epsilon}$  yields the result.  $\square$

**6.2.3. Complexity.** We now discuss how to remove assumptions on  $\beta$ , and analyze the run time of this algorithm. Our discussion extends the arguments in [14], which uses ideas in [31].

By weak duality,  $1 \leq \theta = \frac{\beta}{t-1} \log_{1+\epsilon} 1/\delta$ . This implies that the number of phases,  $t$ , is at most  $1 + \beta \log_{1+\epsilon} 1/\delta = 1 + \frac{\beta}{\epsilon} \log_{1+\epsilon} \frac{m}{1-\epsilon}$ . Thus the running time depends on  $\beta$ .

Let  $\zeta_j$  be the maximum sum of  $j$ -commodity variables that satisfy all constraints  $Ax \leq b$  (e.g., when all other variables are zero). Let  $\zeta = \min_j \zeta_j/d_j$ . Then  $\zeta/k \leq \beta \leq \zeta$ , and these upper and lower bounds on  $\beta$  differ by at most a factor of  $k$ . We scale  $d$  so that this lower bound equals 1. Now  $1 \leq \beta \leq k$ .

We run the algorithm, and if it does not stop after  $T := 2\frac{1}{\epsilon} \log_{1+\epsilon} \frac{m}{1-\epsilon}$  phases, then  $\beta > 2$ . We then multiply demands by 2, so that  $\beta$  is halved, and still at least 1. We continue the algorithm, and again double demands if it does not stop after  $T$  phases. After repeating this at most  $\log k$  times, the algorithm stops. The total number of phases is  $T \log k$ . As noted in [14,31], we can reduce the number of phases further by first computing a 1/2-approximation (within a factor of 2) to our problem, using this scheme. This takes  $\mathcal{O}(\log k \log m)$  phases. We get a value  $\hat{\beta}$  such that  $\beta \leq \hat{\beta} \leq 2\beta$ . Thus with at most  $T$  additional phases, we obtain an  $\epsilon$ -approximate solution. Using the fact that there are at most  $k$  iterations per phase, we have the following lemma.

**Lemma 7.** *The total number of iterations required by the  $k$ -commodity packing algorithm is at most  $2k \log m(\log k + \epsilon^{-2})$ .*

It remains to bound the number of steps. For each step except the last step in an iteration, the algorithm increases some dual variable by a multiplicative factor of  $1 + \epsilon$  (i.e., variable  $y(p)$ ). Since each variable  $y(r)$  has initial value  $\delta/b(r)$  and value at most  $\frac{1}{b(r)}$  before the final step of the algorithm (since  $D(t-1) < 1$ ), the number of steps in the entire algorithm exceeds the number of iterations by at most  $m \log_{1+\epsilon} \frac{1}{\delta} = m \log_{1+\epsilon} \frac{m}{1-\epsilon}$ .

Let  $\mathcal{S}(m, n)$  be the time required to find  $p$  and  $q$  in one step of one iteration.

**Theorem 12.** *Given  $\zeta_j$ , a  $(1 + \epsilon)$ -approximate solution to the  $k$ -commodity packing problem can be obtained in  $\tilde{\mathcal{O}}(\epsilon^{-2} \mathcal{S}(m, n)(k + m))$  time.*

We don't actually need the exact values of  $\zeta_j$ , since they are just used to get an estimate on  $\beta$ . In fact, if we are willing to lose  $\log$  factors in the run time, it suffices that these estimates be within a factor polynomial in  $m$  of  $\beta$ . That is, if our estimate  $\hat{\zeta}_j$  is  $\geq \frac{1}{m} \zeta_j$ , then we have upper and lower bounds on  $\beta$  that differ by a factor of at most  $mk$ .

One way to get an approximate value of  $\zeta_j$  is to use the approximate generalized maximum flow algorithm described in Sects. 2.4 and 3.1 for the single commodity packing LP. Doing this separately for each commodity requires  $\tilde{\mathcal{O}}(m^2)$  time per commodity in lossy networks, for a total of  $\tilde{\mathcal{O}}(km^2)$  time. In networks with flow-generating cycles, this requires a total of  $\tilde{\mathcal{O}}(km \min\{(m + n \log \log M), mn\})$  time, using the FPTAS in one of [32,34,37].

### 6.3. Generalized maximum concurrent flows

The algorithm for the generalized maximum concurrent flows works in phases, and each phase consists of  $k$  iterations. In the  $j^{\text{th}}$  iteration of the  $i^{\text{th}}$  phase, we send flow from  $s_j$  so that  $d_j$  units of flow arrive at  $t_j$ . Each iteration consists of a sequence of steps. In any one step, flow is routed along a single  $(s_j, t_j)$  path. The amount of flow sent along this

path is determined by the minimum of the capacity of this path in the *original* graph, and the remaining unsatisfied demand at this iteration.

This problem is a  $k$ -commodity packing problem, and hence the analysis of the algorithm in the previous section applies. The subroutine to find a most violated primal constraint is a generalized shortest path problem using length function  $l$ .

To get estimates of  $\zeta_j$  for each commodity  $1 \leq j \leq k$ , we could use our approximate generalized maximum flow algorithm. However, we would like something faster, because otherwise this computation will be the bottleneck computation of our algorithm. As before, it suffices to get a solution within a factor polynomial in  $m$  of  $\zeta_j$ .

**Lemma 8.** *In lossy networks, estimates  $\hat{\zeta}_j$ ,  $j = 1, \dots, k$  for which  $\hat{\zeta}_j \geq \zeta_j/m$ , can be computed for all  $j$  in  $\mathcal{O}(\min\{k, n\}(m + n \log n))$  time.*

*Proof.* When there are no flow-generating cycles, a maximum generalized flow can be decomposed into at most  $m$   $s$ - $t$  path flows, e.g., see [17]. The path in this decomposition that delivers the most flow to  $t$  thus generates at least a  $1/m$  fraction of the maximum possible. The  $s$ - $t$  path that maximizes the amount of flow reaching  $t$  over all such paths is at least as good. This path is a *generalized maximum capacity path*. The standard maximum capacity path problem can be solved by modifying Dijkstra's algorithm to update node labels not with the shortest path distance, but with the maximum bottleneck capacity. The generalized maximum capacity path problem can be solved by modifying our definition of bottleneck capacity. Let  $\pi(v)$  denote the capacity of the maximum capacity path reaching  $v$ . Then the capacity of a path reaching  $w$  through node  $v$  is  $\min\{u(v, w), \pi(v)\}\gamma(v, w)$ . Thus  $\pi(w) \leq \min\{u(v, w), \pi(v)\}\gamma(v, w)$ . Since we are concerned with lossy networks,  $\gamma(e) \leq 1 \forall e$ , and thus  $\pi(w) = \max_v \min\{u(v, w), \pi(v)\}\gamma(v, w)$ . Starting with  $\pi(s) = \infty$ , we use this as our update rule in a modified Dijkstra's algorithm to compute the generalized maximum capacity  $s$ - $t$  path. Such a method was described in [40]. We set  $\zeta_j = \pi(t_j)$ . The maximum capacity path for all possible  $s$ - $t$  pairs can be determined with at most  $\min\{k, n\}$  such computations. □

Finding a most violated dual constraint corresponds to a generalized shortest path computation. By Theorem 1  $\mathcal{S}(m, n) = \mathcal{O}(m + n \log n)$  for lossy networks. Combining this with Lemma 8 and Theorem 12 we obtain the following theorem.

**Theorem 13.** *There exists a FPTAS for the generalized concurrent multicommodity flow problem in networks with no flow generating cycles that requires  $\mathcal{O}(\epsilon^{-2}(m + n \log n)(m + k))$  time.*

This matches the run time of one of the asymptotically fastest FPTAS for traditional maximum concurrent flow [11].

To obtain an algorithm for networks with flow-generating cycles, we need to determine  $\hat{\zeta}_j$  for  $1 \leq j \leq k$  and describe the generalized shortest path algorithm. To obtain  $\hat{\zeta}_j$ , we use our approximation algorithm for the single commodity generalized flow problem. To compute generalized shortest paths, we embed Procedure 1 in a scaling framework as described in Sect. 4.3. In this setting, we maintain a lower bound  $L_j$  for each commodity  $j$ . A straightforward analysis yields the following theorem.

**Theorem 14.** *There exists a FPTAS for the generalized concurrent multicommodity flow problem in networks with flow-generating cycles that requires  $\tilde{O}(\epsilon^{-2}(k+m)mn)$  time.*

#### 6.4. Generalized minimum cost concurrent flow

As with the single commodity problem, we can add a budget constraint to the multiple commodity problem, and find a generalized concurrent flow that satisfies the budget constraint and satisfies at least a  $(1 - \epsilon)$  fraction of the maximum demand possible. This is because a budget constraint is a packing constraint, and hence the resulting LP is a  $k$ -commodity packing LP. Since we have a different variable for each commodity-path pair, this budget constraint can easily incorporate different costs for different commodities.

The subroutine to find a most violated primal constraint is, as with the generalized maximum flow with a budget constraint, the generalized shortest path problem using length function  $l(e) + \phi c(e)$ , where  $c$  is the cost vector and  $\phi$  is the dual variable for the budget constraint. This can be easily adapted to multiple budget constraints with corresponding dual variables  $\phi_i$  and cost vectors  $c_i$  using length function  $l + \sum_i \phi_i c_i$ .

To get estimates on  $\zeta_j$  for all  $1 \leq j \leq k$ , as needed to delimit  $\beta$ , it suffices to compute  $\mathcal{O}(m)$ -approximations to the  $\min\{n^2, k\}$  generalized maximum flow with budget problems. We use the algorithms discussed in Sect. 4, with a constant value for  $\epsilon$ .

To find a  $\epsilon$ -approximate generalized maximum concurrent flow of cost no more than the minimum cost generalized maximum concurrent flow, we can use geometric-mean binary search as discussed in Sect. 4.2. In the following theorems, the first expression in the run time comes from finding  $\hat{\zeta}_j$ , and the second expression is the time needed to solve the scaled problem obtained with the bounds given by the  $\hat{\zeta}_j$ .

**Theorem 15.** *There exists a FPTAS for the generalized minimum cost concurrent flow problem in lossy networks that requires  $\tilde{O}(km^2 + \epsilon^{-2}m(k+m)(\log \epsilon^{-1} + \log \log M))$  time.*

**Theorem 16.** *There exists a FPTAS for the generalized minimum cost concurrent flow problem in networks with flow-generating cycles that requires  $\tilde{O}(km^2n + \epsilon^{-2}mn(k+m)(\log \epsilon^{-1} + \log \log M))$  time.*

*Acknowledgements.* This research was inspired by a conversation the authors had with Jeffrey Oldham at INFORMS Montreal in May, 1998. We are grateful to Jeffrey for providing us with a preprint of [30] and pointing out the reference [6]. We are also grateful to the associate editor and the anonymous referees for their useful comments.

## References

1. Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms, 1995
2. Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, 1999
3. Ahuja, R.K., Magnanti, T.L., Orlin, J.B. (1993): Network Flows: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs, NJ
4. Aspvall, B., Shiloach, Y. (1980): A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. SIAM J. Comput. **9**, 827–845

5. Bellman, R.E. (1958): On a routing problem. *Quart. Appl. Math.* **16**, 87–90
6. Charnes, A., Raike, W.M. (1966): One-pass algorithms for some generalized network flow problems. *Oper. Res.* **14**, 914–924
7. Cohen, E., Megiddo, N. (1994): Improved algorithms for linear inequalities with two variables per inequality. *SIAM J. Comput.* **23**, 1313–1347
8. Cohen, E., Megiddo, N. (1994): New algorithms for generalized network flows. *Math. Program.* **64**, 325–336
9. Dantzig, G.B. (1962): *Linear programming and extensions*. Princeton University Press, Princeton, NJ
10. Dijkstra, E.W. (1959): A note on two problems in connexion with graphs. *Numer. Math.* **1**, 269–271
11. Fleischer, L.K. (2000): Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.* **13**(4), 505–520 (electronic)
12. Ford, L.R., Fulkerson, D.R. (1962): *Flows in Networks*. Princeton University Press, Princeton, NJ
13. Fredman, M.L., Tarjan, R.E. (1987): Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* **34**, 596–615
14. Garg, N., Könemann, J. (1998): Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In: 39th Annual IEEE Symposium on Foundations of Computer Science, pp. 300–309
15. Glover, F., Klingman, D., Phillips, N. (1990): Netform modeling and applications. *Interfaces* **20**, 7–27
16. Goldberg, A.V. (1992): A natural randomization strategy for multicommodity flow and related algorithms. *Information Processing Letters* **42**, 249–256
17. Goldberg, A.V., Plotkin, S.A., Tardos, É. (1991): Combinatorial algorithms for the generalized circulation problem. *Math. Oper. Res.* **16**, 351–379
18. Goldfarb, D., Jin, Z. (1996): A faster combinatorial algorithm for the generalized circulation problem. *Math. Oper. Res.* **21**, 529–539
19. Goldfarb, D., Jin, Z., Orlin, J.B. (1997): Polynomial-time highest gain augmenting path algorithms for the generalized circulation problem. *Math. Oper. Res.* **22**, 793–802
20. Grigoriadis, M.D., Khachiyan, L.G. (1994): Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM J. Optim.* **4**, 86–107
21. Grigoriadis, M.D., Khachiyan, L.G. (1996): Coordination complexity of parallel price-directive decomposition. *Math. Oper. Res.* **21**, 321–340
22. Hassin, R. (1992): Approximation schemes for the restricted shortest path problem. *Math. Oper. Res.* **17**, 36–42
23. Hochbaum, D.S., Naor, J. (1994): Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM J. Comput.* **23-6**, 1179–1192
24. Kamath, A., Palmon, O. (1995): Improved interior point algorithms for exact and approximate solution of multicommodity flow problems. In: *ACM/SIAM [1]*, pp. 502–511
25. Kantorovich, L.V. (1960): *Mathematical methods in the organization and planning of production*. Publication House of the Leningrad State University, p. 68, 1939. Translated in: *Manage. Sci.* **6**, 366–422
26. Klein, P., Plotkin, S., Stein, C., Tardos, É. (1994): Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM J. Comput.* **23**, 466–487
27. Leighton, T., Makedon, F., Plotkin, S., Stein, C., Tardos, É., Tragoudas, S. (1995): Fast approximation algorithms for multicommodity flow problems. *J. Comput. System Sci.* **50**, 228–243
28. Megiddo, N. (1983): Applying parallel computation algorithms in the design of serial algorithms. *J. ACM* **30**, 852–865
29. Moore, E.F. (1959): The shortest path through a maze. In: *International Symposium on the Theory of Switching*, pp. 285–292
30. Oldham, J.D. (1999): Combinatorial approximation algorithms for generalized flow problems. In: *ACM/SIAM [2]*
31. Plotkin, S.A., Shmoys, D., Tardos, É. (1995): Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.* **20**, 257–301
32. Radzik, T. (1993): Approximate generalized circulation. Technical Report 93-2, Cornell Computational Optimization Project, Cornell University
33. Radzik, T. (1997): Fast deterministic approximation for the multicommodity flow problem. *Math. Program.* **78**, 43–58
34. Radzik, T. (1998): Faster algorithms for the generalized network flow problem. *Math. Oper. Res.* **23**, 69–100
35. Shahrokhi, F., Matula, D.W. (1990): The maximum concurrent flow problem. *J. ACM* **37**, 318–334
36. Shostak, R. (1981): Deciding linear inequalities by computing loop residues. *J. ACM* **28**, 769–779
37. Tardos, É., Wayne, K.D. (1998): Simple generalized maximum flow algorithms. In: 6th International Integer Programming and Combinatorial Optimization Conference, pp. 310–324

38. Truemper, K. (1977): On max flows with gains and pure min-cost flows. *SIAM J. Appl. Math.* **32**, 450–456
39. Vaidya, P.M. (1989): Speeding up linear programming using fast matrix multiplication. In: 30th Annual IEEE Symposium on Foundations of Computer Science, pp. 332–337
40. Wayne, K.D. (1999): Generalized Maximum Flow Algorithms. PhD thesis, Department of Operations Research and Industrial Engineering, Cornell University
41. Wayne, K.D. (1999): A polynomial combinatorial algorithm for generalized minimum cost flow. In: Proceedings of the 31th Annual ACM Symposium on Theory of Computing
42. Wayne, K.D., Fleischer, L. (1999): Faster approximation algorithms for generalized flow. In: ACM/SIAM [2]
43. Young, N. (1995): Randomized rounding without solving the linear program. In: ACM/SIAM [1], pp. 170–178