

# Heuristic Improvements in Multi-commodity Flow Algorithm

Naveen Garg, Shubham Mittal, Pushkar Tripathi

Indian Institute of Technology, Delhi

**Abstract.** We propose heuristic improvements in the procedure to obtain a  $(1+\omega)$ -approximation of the multicommodity flow which satisfies all demands when the cost bound is given as an input parameter. Through a series of improvements we are able to significantly reduce the number of shortest path computations. We also present a new version of the algorithm given in [1] which is amenable to our optimizations. Finally we compare our results against those given in [2] and demonstrate significant improvement in the results. We also propose methods to reduce the running time of the algorithm.

## 1 Introduction

The multicommodity flow algorithm is the problem of routing flow from various sources to their respective sinks in a directed capacitated network. The specification of each commodity consists of the *demand, source and target* for the commodity. The *cost of flow* through an edge is proportional to the flow routed through the edge and *the total cost* of any flow is the cummulation of its costs over all edges. Various versions of this problem have been studied since it was proposed in the 1980s. Among the most common versions is one in which a flow of the lowest cost needs to be found such that it fulfills the demands for all commodities. Another variant tries to find a flow that satisfies the largest fraction of all demands such that its total cost is less than a given upper bound.

The problem may be formulated as a *linear-programming problem* and solved using the interior point method as shown by [3] and [4]. Alternately one may use LP-solvers like *CPLEX* and *SOPLEX* to solve this problem. However these methods are too slow in practice and cannot be used for most real-world examples as shown in [2]. Another avenue of attacking this problem is through combinatorial approximation algorithms. Most combinatorial algorithms for solving this problem rely on ideas presented in [1], [5], [6].

Though development of faster algorithms for this problem remains an area of active research many people have looked at experimental validation of these ideas. The first experimental results for this algorithm were published by [7] and [9]. [2] and [8] give heuristics to improve the running time of procedure and cite extensive experimental results regarding the effectiveness of these heuristics. Since [2] is the most recent and thorough experimental study of this algorithm and it incorporates the new ideas presented in [1] we will compare our results to those given in [2].

There are two variants of the algorithm - the first uses shortest path computations as a basic procedure while the second uses single commodity mincost flow (MCF) computations. We will use the former in this paper. There is a strong case to work on the shortest-path version of the algorithm since it has a slightly better asymptotic complexity than the MCF version of the algorithm as shown in [1]. Further more, due to the relative simplicity of the shortest-path procedure it presents greater scope for improvement.

This paper is organized as follows. Section 2 has a brief description of the combinatorial approximation algorithm used in this paper. In section 3 we present heuristics to reduce

the number of shortest path computations required by the algorithm. In section 4 we explain the testing framework and present our results. Here we also compare our results with those given in [2]. In section 5 we discuss some techniques to speed up the shortest path computation and also provide some experimental results for the same. Finally, we conclude in section 6 with a brief discussion of our results and observations.

## 2 Overview of the Algorithm

In this section, we will first introduce the terminology that is going to be used in the rest of the paper. The algorithm presented here is a variant of the one presented in [1].

### Terminology

Given a directed graph  $G = (V, E)$ ,  $k$  source-target pairs  $(s_i, t_i)$  where  $s_i \in V, t_i \in V$ , functions  $\mathbf{c} : E \rightarrow \mathcal{R}^+ \cup \{0\}$ ,  $\mathbf{b} : E \rightarrow \mathcal{R}^+ \cup \{0\}$ , and  $\mathbf{d} : \{1, 2, \dots, k\} \rightarrow \mathcal{R}^+ \cup \{0\}$ , and a scalar  $B$ , a valid flow that meets all the demands is a function  $\mathbf{f} : E \rightarrow \mathcal{R}^+ \cup \{0\}$  such that

$$f(e) \leq c(e) \quad \forall e \in E \quad (1)$$

$$\sum_{e \in E} f(e) * b(e) \leq B \quad (2)$$

$$\sum_v^{e:(s_i,v)} f(e) - \sum_w^{e:(w,s_i)} f(e) \geq d_i \quad \forall i \in \{1, 2, \dots, k\} \quad (3)$$

$$\sum_v^{e:(t_i,v)} f(e) - \sum_w^{e:(w,t_i)} f(e) \leq -d_i \quad \forall i \in \{1, 2, \dots, k\} \quad (4)$$

$$\sum_v^{e:(u,v)} f(e) - \sum_w^{e:(w,u)} f(e) = 0 \quad \forall u \in V, u \neq s_i, u \neq t_i \quad (5)$$

Here  $B$  is the cost bound,  $\mathbf{c}$  represents the capacity of each edge,  $\mathbf{b}$  is the cost function,  $\mathbf{d}$  gives the demands for each of the  $k$  commodities, and  $\mathbf{f}$  is the flow function. We will use  $d_i$  instead of  $d(i)$  to denote the demand for the  $i^{\text{th}}$  commodity, and use  $s_i$  and  $t_i$  to represent its source and target vertices. We will differentiate between scalars and functions by representing functions as boldface.

To merge cost constraint 2 with capacity constraint 1, we will extend  $E$  to  $\overline{E} := E \uplus \{\bar{e}\}$ , where  $\bar{e}$  is a pseudo edge. We will also extend  $\mathbf{c}$  to  $\overline{E}$  by defining  $c(\bar{e}) = B$  and extend  $\mathbf{f}$  to  $\overline{E}$  by defining  $f(\bar{e}) = \sum_{e \in E} f(e) * b(e)$ . With this, we can merge equation 1 and 2 and simply write

$$f(e) \leq c(e) \quad \forall e \in \overline{E} \quad (6)$$

A  $(1 + \omega)$ -approximate flow  $\mathbf{f}$  is a function  $\mathbf{f} : E \rightarrow \mathcal{R}^+ \cup \{0\}$  which satisfies equations 3, 4 and 5, and satisfies a relaxed version of equation 6, given below

$$f(e) \leq (1 + \omega)c(e) \quad \forall e \in \overline{E} \quad (7)$$

We will now describe our algorithm, which given a cost bound, finds a  $(1 + \omega)$ -approximate if it exists. Using this algorithm, we can find the minimum cost for a given flow network through a binary search on the solution space, which will make  $O(\log \log B_{max} + \log(\frac{1}{\omega}))$  calls to our algorithm, by first binary searching on the exponent of the cost, and then on its mantissa.

## An overview of the Algorithm

Let us first formalize the problem at hand as a linear program

*Minimize*  $B$  : subject to

$$\frac{1}{d_i} \sum_{P:s_i \rightarrow t_i} f_P \geq 1 \quad \forall i \in \{1, 2, \dots, k\} \quad (8)$$

$$\frac{1}{c(e)} \sum_{P:e \in P} f_P \leq 1 \quad \forall e \in \bar{E} \quad (9)$$

$$f_P \geq 0 \quad (10)$$

where  $f_P$  is a variable representing the flow along a path  $P$  between one of the source-target pairs.

We now introduce dual variables  $y(e) : \forall e \in \bar{E}$  and  $z_i : \forall i \in \{1, 2, \dots, k\}$ , multiply equation 8 with  $z_i$ , equation 9 with  $y(e)$  and add to get

$$\sum_{e \in \bar{E}} \frac{y(e)}{c(e)} \sum_{P:e \in P} f_P \leq \sum_{e \in \bar{E}} y(e) \quad (11)$$

$$\sum_{i=1}^k \frac{z_i}{d_i} \sum_{P:s_i \rightarrow t_i} f_P \geq \sum_{i=1}^k z_i \quad (12)$$

Substituting  $\underline{y}(e) \left( = \frac{y(e)}{\sum y(e)} \right)$  as the normalized variables, and separating  $\bar{e}$  from  $\bar{E}$ , from equation 11 we get

$$\begin{aligned} & \sum_{e \in E} \frac{\underline{y}(e)}{c(e)} \sum_{P:e \in P} f_P + \frac{y(\bar{e})}{B} \sum_P f_P \left( \sum_{e \in P} b(e) \right) \leq 1 \\ \Rightarrow & \sum_P f_P \left( \sum_{e \in P} \frac{\underline{y}(e)}{c(e)} + \sum_{e \in P} \frac{y(\bar{e})b(e)}{B} \right) \leq 1 \end{aligned} \quad (13)$$

Similarly using normalized  $z_i$  as  $\underline{z}_i = \frac{z_i}{\sum z_i}$ , we can rewrite equation 12 as

$$\begin{aligned} & \sum_{i=1}^k \frac{\underline{z}_i}{d_i} \sum_{P:s_i \rightarrow t_i} f_P \geq 1 \\ \Rightarrow & \sum_P f_P \cdot \frac{\underline{z}_i}{d_i} \geq 1 \end{aligned} \quad (14)$$

where in equation 14,  $i$  is such that  $P$  is a path from  $s_i$  to  $t_i$ .

From equations 13 and 14, we conclude that there should exist a path  $P$  between a source  $s_i$  and its target  $t_i$  such that

$$\sum_{e \in P} \frac{\underline{y}(e)}{c(e)} + \sum_{e \in P} \frac{y(\bar{e})b(e)}{B} \leq \frac{\underline{z}_i}{d_i} \quad (15)$$

Thus, defining length function  $\mathbf{len} : E \rightarrow \mathcal{R}^+ \cup \{0\}$  as

$$\mathit{len}(e) = \frac{\underline{y}(e)}{c(e)} + \frac{y(\bar{e})b(e)}{B} \quad (16)$$

we get

$$\text{len}(P) \leq \frac{\underline{z}_i}{d_i} \quad (17)$$

### 3 Optimizations to our Algorithm

To improve the overall running time of our implementation, we will try to optimize the algorithm through a number of heuristics, which can be broadly classified into the following two categories.

1. Heuristics that try to reduce the total number of shortest path computations required by the algorithm. For these we will experiment with the parameters of the algorithm and evaluate the results.
2. Heuristics that alter the technique for finding the shortest paths and thus reduce the running time of the implementation. These heuristics will target the fact that in every iteration, a large number of shortest path computations are performed for the same source-target pair, and thus some of the results of the current computation may be reused in the next one.

In this section, we will mainly focus on the heuristics belonging to the first category. Since this algorithm makes lots of calls to the shortest-path subroutine, these heuristics should significantly reduce the overall running time of the algorithm.

#### Rapid Path Utilization : Algorithm A

As noted earlier, in each step we need a path that satisfies equation 17. This need not be the shortest path between the source and the target. In particular, when we find a shortest path and route flow along that path making necessary updates, the path might continue to satisfy equation 17. Thus instead of re-computing the shortest path after every flow routing, we shall continue to route flow along the same path until it violates equation 17.

We will try to get a lower bound on the number of times for which flow  $f$  should be routed along a candidate path  $P$  that satisfies equation 17, before the path becomes saturated (i.e. it violates equation 17). For all edges  $e \in P$ , we know that

$$\underline{y}^{j+1}(e) \approx \frac{\underline{y}^j(e)(1 + \alpha \frac{f}{c(e)})}{1 + \sum_{e \in P} \alpha \left( \underline{y}^j(e) \frac{f}{c(e)} + \underline{y}^j(\bar{e}) \frac{fb(e)}{B} \right)} \quad (18)$$

where  $\underline{y}^j$  is the normalized value of  $\mathbf{y}$  after sending  $j \cdot f$  units of flow along the path. Similarly, for commodity  $i$  for which flow is being sent in this particular iteration

$$\underline{z}_i^{j+1} \approx \frac{\underline{z}_i^j(1 - \alpha \frac{f}{d_i})}{1 - \alpha \frac{f}{d_i}} \quad (19)$$

Recall that for path  $P$ , its length after  $j$  routings of flow  $f$  is  $\sum_{e \in P} \left( \frac{\underline{y}^j(e)}{c(e)} + \underline{y}^j(\bar{e}) \frac{b(e)}{B} \right)$ . We denote this by  $\text{len}^j(P)$ .

Using  $f \leq c(e)$  in equation 18, and summing it over all edges in  $P$  after dividing both sides by  $c(e)$ , we get the following inequality.

$$\begin{aligned} \text{len}^{j+1}(P) &\leq \text{len}^j(P) \left( \frac{1 + \alpha}{1 + \alpha f \cdot \text{len}^j(P)} \right) \\ \left( \frac{1}{1 + \alpha} \right)^{j+1} \cdot \left\{ \frac{1}{\text{len}^0(P)} - f \right\} &\leq \frac{1}{\text{len}^{j+1}(P)} - f \end{aligned} \quad (20)$$

Similarly for equation 19 we get,

$$\frac{1}{z_i^{j+1}} - \frac{f}{d_i} \leq \left( \frac{1}{1 + \alpha \frac{f}{d_i}} \right)^{-(j+1)} \left( \frac{1}{z_i^0} - \frac{f}{d_i} \right) \quad (21)$$

Now we can route flow along the path  $P$  so long as equation 17 is not violated. That is, from equation 15, as long as,

$$\frac{d_i}{z_i^r} \leq \frac{1}{len^r(P)} \quad (22)$$

Using inequalities 20 and 21, we get a lower bound on the number of times flow may be safely routed through the path  $P$ .

$$r = \left\lceil \log \left( \frac{\frac{1}{len^0(P)} - f}{\frac{d_i}{z_i^0} - f} \right) / \log \left( \frac{1 + \alpha}{1 + \alpha f/d_i} \right) \right\rceil \quad (23)$$

We will route flow  $f$  through this path  $r$  times in a single execution, and then update the dual variables accordingly. Let this be the basic version of our algorithm (**Algorithm A**), over which other heuristics will be applied.

### Early Exit : Algorithm B

Recall that for a given approximation factor  $\omega$  and cost bound  $B$ , our algorithm tries to find a flow that meets all the demands and satisfies the capacity and cost constraints within a factor of  $(1 + \omega)$ .

For a flow  $f$ , let us define

$$\lambda_f = \max_{e \in E} \left( \frac{f(e)}{c(e)} \right) \quad (24)$$

The routine thus has to return a flow that meets all the demands and for which  $\lambda_f \leq (1 + \omega)$ . We will use this as a condition to exit from the routine in addition to those that are obtained theoretically in ?? and are used in *Algorithm A*. Let us call this version of the algorithm as **Algorithm B**.

### Ordering of commodities : Algorithm C

During a single phase, all the commodities have to be considered one by one for an iteration of updates. Theoretically, there seems to be no particular ordering of the commodities that might improve the asymptotic running time of the algorithm. However the running time of the implementation might depend on these orderings.

From equations 17 and 23, it is clear that smaller the length of the shortest path  $P_i$  between  $s_i$  and  $t_i$ , and larger the ratio  $\frac{z_i}{d_i}$ , greater is the number of times one can route flow through the shortest path. And thus more work can be done in a single step. Hence a useful parameter that can be used for ordering the commodities is  $\frac{len(P_i)}{z_i/d_i}$ , where  $P_i$  is the shortest path from  $s_i$  to  $t_i$ .

We have experimented with a number of different orderings with different parameters, like  $len(P_i)$ ,  $\frac{z_i}{d_i}$ , etc. but have found the results most encouraging, when we order commodities in increasing order of  $\frac{len(P_i)}{z_i/d_i}$ , at the start of a phase. As a few shortest path computations are required in finding  $len(P_i)$ , we do this ordering periodically, so that atmost 5% overhead is seen in the number of shortest path computations. This will be called **Algorithm C**.

## Demand Scaling : Algorithm D

Based on a similar idea as above, if we start with small demands, the algorithm would converge faster as  $\frac{z_i}{d_i}$  would increase, and thus more flow can be routed through a path in each step. Intuitively as well, if demands are small, one needs to do less work in routing these demands. Hence in this optimization, we start with half the actual demands, and then slowly increase these demands to their actual value. In each step, we will need to distort the flow very slightly, so as to meet the small increase in demands. Thus we do a small amount of work in every step.

An important parameter in this optimization is the number of steps taken while going from  $\frac{d_i}{2}$  to  $d_i$ ,  $\forall i$ . We experimented with a number of different values, and obtained the best results when we made 25 hops.

## Epsilon Walk : Algorithm E

The asymptotic running time of the algorithm depends directly on  $\frac{1}{\omega^2}$ , where  $\omega$  is the approximation factor (which we will refer by  $\epsilon$  from now). In this optimization, instead of finding the flow directly at  $\epsilon = \omega$ , we will find it at a sequence of decreasing  $\epsilon$ , terminating at  $\epsilon = \omega$ . The intuition behind such an idea is as follows:

In the beginning, nothing is known about the flow network to the algorithm. Hence by working at large  $\epsilon$ , the algorithm can quickly converge to a flow that meets all the demands, but violates the capacity and cost constraints by some small factor. And now by running the algorithm at a reduced  $\epsilon$ , the algorithm can make use of the knowledge of the previous computation.

In other words, instead of making  $(\frac{1}{\omega})^2$  shortest path computations while finding the flow directly at  $\epsilon = \omega$ , we will run the algorithm at a series of decreasing  $\epsilon$ , and thus make about  $(\frac{1}{\epsilon_1})^2 + (\frac{1}{\epsilon_2} - \frac{1}{\epsilon_1})^2 + (\frac{1}{\epsilon_3} - \frac{1}{\epsilon_2})^2 \dots + (\frac{1}{\epsilon_n} - \frac{1}{\epsilon_{n-1}})^2$  shortest path computations, where  $\epsilon_1 > \epsilon_2 > \dots > \epsilon_n = \omega$ , which should be an improvement by virtue of the well known fact :  $\sum_i n_i^2 < (\sum_i n_i)^2$ .

We tried using this optimization over **Algorithm D** but the results were not as good as using this without *Demand Scaling* optimization, i.e. directly with **Algorithm C**. Hence we will call this version of the algorithm which uses *Epsilon Walk* over Algorithm C as **Algorithm E**.

## 4 Experimental Results

We tested our implementations on graphs generated by three generators. These families are described below.

1. **NETGEN(n, m, k)**: These graphs are generated by the NETGEN generator [10]. Here  $n$  denotes the number of nodes,  $m$  is the number of edges and  $k$  the number of commodities. The edge capacities were uniformly distributed in the range  $[0, 5000]$ .
2. **GENFLOT(n,m,k,b)**: This generator was developed to produce non-linear test problems. It produces graphs with  $b$  blocks and demand in the range  $[0, 10000]$
3. **GRIDGEN(n,m,k)**: This network generator generates a grid-like network plus a super node. In addition to the arcs connecting the nodes in the grid, there is an arc from each supply node to the super node and from the super node to each demand

node to guarantee feasibility. These arcs have very high costs and very big capacities. The source and sink nodes are selected uniformly in the network, and the demand of each source-sink pair is also assigned by uniform distribution. The arc capacity is in the range [100, 30000].

We implemented the algorithms explained in the previous section and Tables 1, 2 and 3 show the results from our experiments. The first column gives the graph parameters while the subsequent columns show number of calls made to the *shortest-path subroutine* in each of the algorithms. We also implemented the algorithm described in [2] ourselves, as their implementation was unavailable. In [2], they start with any flow that meets all the demands. In our implementation, we chose to begin with a flow that floods the BFS paths between each source and target, with flow equal to the demand of the commodity. The last column in each table gives the number of calls made to the shortest-path subroutine by this implementation.

All tests were carried out at an approximation factor of  $\omega = 0.1$  (i.e. 10 percent error). Smaller approximation factors could not be tested on such large graphs due to the problem of storing the dual variables, that increase exponentially as  $\omega$  decreases. [2] and [8] show some results for smaller approximation factors (3-7 percent error) but these are for small graphs having only a few hundred vertices. To the best of our knowledge no experimental study has been done for graphs with upto 10000 vertices and 200000 edges.

Problem Inst. (n,m,k)	Alg. A	Alg. B	Alg. C	Alg. D	Alg. E		Radzik
					2.0	1.414	
N1: 1000,10000,50	94187	1470	917	1202	592	572	8850
N2: 1500,15000,75	132118	1388	1404	1786	765	1244	14175
N3: 2000,20000,100	155652	8702	8694	7616	3869	2315	20000
N4: 2500,25000,125	283833	25569	23911	14976	7412	6555	24500
N5: 3000,30000,150	247030	7987	7035	4675	1640	1690	30600
N6: 3500,35000,175	437722	22792	18697	12815	7529	6275	36575
N7: 4000,40000,200	418718	10887	10579	7074	1954	8009	41600

**Table 1.** Table of results for graphs from the NETGEN family

Problem Inst. (n,m,k)	Alg. A	Alg. B	Alg. C	Alg. D	Alg. E		Radzik
					2.0	1.414	
G1: 4061,81220,200	527220	51727	55852	32005	14538	13799	61400
G2: 5001,100020,250	823630	136894	131713	64350	24003	27466	78684
G3: 5881,117620,300	$\geq 10^5$	42811	39429	25182	21796	11112	96600
G4: 6901,138020,350	$\geq 10^5$	71425	64911	30257	28836	24659	113774
G5: 8126,162520,400	$\geq 10^5$	109672	92661	59543	42635	30659	131340
G6: 9101,182020,450	$\geq 10^5$	47283	42250	31380	4589	17233	149068
G7: 10151,203020,500	$\geq 10^5$	60007	57820	41707	21404	12313	166992

**Table 2.** Table of results for graphs from the GRIDGEN family

Problem Inst. (n,m,k)	Alg. A	Alg. B	Alg. C	Alg. D	Alg. E		Radzik
					2.0	1.414	
F1: 1000,10000,19	32365	258	243	310	216	212	1577
F2: 2000,20000,39	89112	2149	2637	1331	1030	2104	2496
F3: 4000,40000,79	158310	1486	1498	1906	983	1003	5451
F4: 5000,50000,250	660543	10516	8834	7534	2613	9560	27250
F5: 6000,60000,300	$\geq 10^5$	21749	19187	12341	15775	9934	33900
F6: 8000,80000,400	$\geq 10^5$	138918	129768	73919	35613	30971	58800
F7: 9000,90000,450	$\geq 10^5$	34241	31319	21058	18591	13733	89100

**Table 3.** Table of results for graphs from the GENFLOT family

## 5 Time Optimizations

Since shortest-path procedure is an integral part of the algorithm any improvement in the running time of this procedure would have significant bearing on the overall running time of the algorithm. Various versions of Dijkstra’s algorithm to find the shortest path in a directed graphs are known(see [11]) and among these the double bucket implementation is believed to be the fastest in practice. However most of these algorithms, such as [12], use the discreteness of the edge length to achieve running times that depend on the length of the longest edge. This is not feasible in our context since the range of values that the edge lengths (dual variables) can attain is very large. Further more, through our experiments we found that an error of  $10^{-4}$  percent in the tightness of equation 17 can cascade in to huge errors in the final output of the algorithm. Thus we decided against using numerical algorithms for the shortest-path subroutine.

[13] presents a survey of various combinatorial methods to speed up dynamic shortest path(DSP) algorithms where the edge lengths change randomly. We implemented some new DSP optimizations that exploit the deterministic manner of change of the edge lengths in our algorithm. These are explained below.

### Warm Starting : Optimization A

In the traditional version of the Dijkstra’s algorithm every vertex is assigned an initial potential of infinity. The potential values then fall to their actual values until the termination of the algorithm. However in our algorithm the potentials need not be reset to infinity between successive executions of the shortest-path subroutine for the same source-target pair. One can view the output of the shortest-path subroutine as a directed tree where the parent of every node is its predecessor in the shortest path. The updations of the dual variables increases the edge lengths for edges in a path of this tree. We can bound the final length of the shortest path from the source to any vertex by its distance from the source in this tree. This can be used to reduce the number of heap decrement operations required to find the shortest path.

### Shortcut Methods : Optimization B

It has been noted in section 3 that the basic algorithm relies on finding paths that satisfy equation 17 and these need not be the shortest source-target paths. Flow is routed through a path until it ceases to satisfy equation 17. However we reuse information obtained in the previous invocation of the shortest-path subroutine to look for new paths that satisfy

our constraints. By examining the length of the shortest path to nodes from which edges are incident on the target we try to locate alternate paths that satisfy equation 17. If we fail to locate a path that satisfies equation 17 we continue our search by back-tracking along the current shortest path and analyzing the edges incident on this vertex. Through experimentation we found that back-tracking upto a maximum of five levels in the search path gives the best results. If we still fail to locate an alternate path we invoke the shortest-path subroutine.

The optimizations described above were implemented and tested on 3 Ghz, 2GB RAM, penium IV machines. Table 4 gives the running times for these experiments. We compare the execution time against a version of Algorithm E which does not use these optimizations.

Instance (N, G, F)	NETGEN			GRIDGEN			GENFLOT		
	Alg E	Opt A	Opt B	Alg E	Opt A	Opt B	Alg E	Opt A	Opt B
N1,G1,F1	820	760	660	1.8E5	1.6E5	1.5E5	290	280	250
N2,G2,F2	1870	1720	1460	3.7E5	3.4E5	3.2E5	2730	2670	830
N3,G3,F3	9510	8610	3040	4.8E5	3.6E5	3.5E5	4480	4370	3570
N4,G4,F4	27700	25200	24200	6.5E5	5.8E5	5.6E5	4160	23140	19350
N5,G5,F5	660	8140	6870	1.2E6	1.1E6	1.0E6	1.9E5	1.8E5	28730
N6,G6,F6	43910	40230	42320	1.3E5	1.2E5	1.0E5	6.3E5	6.0E5	5.2E5
N7,G7,F7	11810	10920	9080	8.4E5	6.7E5	6.5E5	5.1E5	3.4E5	3.4E5

**Table 4.** Table of running times in ms for all graphs

We tested our algorithm for very large graphs with upto  $10^6$  edges. Table ?? gives the running times for such graphs.

## 6 Discussion and Conclusion

### References

1. Naveen Garg and Jochen Konemann, "Faster and Simpler Algorithms for Multicommodity Flow and Other Fractional Packing Problems," FOCS, 1998
2. T. Radzik, "Experimental Study of a solution method for multicommodity flow problems," Workshop on Algorithm Engineering and Experiments, 2000.
3. S. Kapoor and P. M. Vaidya, "Fast algorithms for convex programming and multicommodity flows," STOC, CA, USA, 1986.
4. A. P. Kamath and O. Palmon, "Improved interior point algorithms for exact and approximate solution of multicommodity flow problems," SODA, San Francisco, 1995
5. A. V. Goldberg, "A natural randomization strategy for multicommodity flow and related algorithms," Infomation Processing Let., 1992
6. S. Plotkin, D. Shmoys and E. Tardos, "Fast approximation algorithms for fractional packing and covering problems," Math Opp. Res., 1995
7. A. V. Goldberg, A. D. Oldham, S. Plotkin, C. Stein, "An implementation of an Approximation Algorithm for Minimum-Cost Multicommodity Flows," IPCO, 1998.
8. D. Bienstock, "Potential Function Methods for Approximately Solving Linear Programs: Theory and Practice," CORE, Belgium, 2001.
9. T. Leong, P. Shor, C. Stein, "Implementation of combinatorial multicommodity flow algorithm," Network flows and Algorithms DIMACS series in discrete mathematics and theoretical computer science, American MathematicalS, 1993
10. D. Klingman, A. Napier, and J. Stutz, "Netgen: A program for generating large scale capacitated assignment, transportation and minimum cost flow network problems," Management Science, 20:814821, 1974.

11. B. V. Cherkassky, A. V. Goldberg, Tomasz Radzik, "Shortest Path Algorithms: Theory and Experimental Evaluation," SODA, 1994.
12. R. B. Dial, "Algorithm 360: Shortest-path forest with topological ordering," Commun. ACM, NY, USA, 1969
13. L. S. Buriol, M. G. C. Resende and M. Thorup, "Speeding up dynamic shortest path algorithms," AT&T Labs Tech. Report, 2003