How to Split a Flow ?

Tzvika Hartman*, Avinatan Hassidim*, Haim Kaplan*†, Danny Raz*‡, Michal Segalov*
*Google, Inc. Israel R&D Center
†Tel Aviv University
‡Technion, Israel
{tzvika, avinatan, haimk, razdan, msegalov}@google.com

***Abstract*—Many practically deployed flow algorithms produce the output as a set of values associated with the network links. However, to actually deploy a flow in a network we often need to represent it as a set of paths between the source and destination nodes.**

In this paper we consider the problem of decomposing a flow into a small number of paths. We show that there is some fixed constant $\beta > 1$ such that it is NP-hard to find a decomposition in which the number of paths is larger than the optimal by a factor of at most β. Furthermore, this holds even if arcs are associated only with three different flow values. We also show that straightforward greedy algorithms for the problem can produce much larger decompositions than the optimal one, on certain well tailored inputs. On the positive side we present a new approximation algorithm that decomposes all but an ϵ-fraction of the flow into at most $O(1/\epsilon^2)$ times the smallest possible number of paths.

We compare the decompositions produced by these algorithms on real production networks and on synthetically generated data. Our results indicate that the dependency of the decomposition size on the fraction of flow covered is exponential. Hence, covering the last few percent of the flow may be costly, so if the application allows, it may be a good idea to decompose most but not all the flow. The experiments also reveal the fact that while for realistic data the greedy approach works very well, our novel algorithm which has a provable worst case guarantee, typically produces only slightly larger decompositions.

I. Introduction

Often when tackling network design and routing problems, we obtain a flow (or a multicommodity flow) between a source-sink pair (or pairs) by running a maximum flow algorithm, a minimum cost flow algorithm, or solving a linear program that captures the problem.[1] These algorithms represent and output a flow by associating a flow value with each arc. The value of each arc is at most the arc capacity, and in each node the total incoming flow equals the total outgoing flow, except for the source and the sink. See Figure 1 for an example.

To deploy a flow in a network, say a traffic engineered MPLS network [7], [15], or an open-flow network [13], we often need to represent the flow as the union of paths from the source to the sink, such that each path is coupled with the amount of flow that it carries. In general, there may be many ways to represent a flow as a union of source-sink paths (see Figure 1). While all these representations route the same flow

[1]Linear programming or even convex programming is typically the method of choice for more complicated multicommodity flow and network design problems.

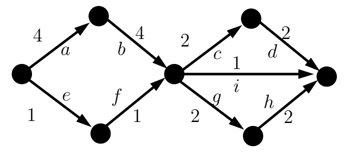

Fig. 1. A flow. We can decompose this flow into 4 flow-paths e, f, g, h of value 1, a, b, g, h of value 1, a, b, i of value 1, and a, b, c, d of value 2. A smaller decomposition consists of the paths e, f, i of value 1, a, b, g, h of value 2, a, b, c, d of value 2.

they may differ substantially in the number of paths used, their latency, and in other parameters. Our focus in this paper is on algorithms that decompose a flow into a minimum number of paths. (If we are given a multicommodity flow, we just apply the algorithm to each flow separately.)

Minimizing the number of paths is desirable for many reasons. When engineering traffic in a network each path consumes entries in tables of the routers that it goes through. Each path also requires periodic software maintenance to check its available bandwidth and other quality parameters [16], [4]. Therefore, reducing the number of paths saves resources and makes network management more efficient, both for automatic software and network operators.

Evidently, the number of paths is not the only important quality criteria of the decomposition. Other parameters are also important. Among them are the maximum and average latency of the paths, and the maximum number of paths going through a single link or a single node. Optimizing more than one parameter is not naturally well defined, and thereby more complicated. We focus on the number of paths as our main objective criterion, but we do address other criteria in our experimental study.

Flow algorithms use the notion of *residual graph* which contains all nonsaturated arcs on which we can push more flow. Initially the residual is identical to the original graph. An augmenting path based algorithm finds a directed path from the source to the sink in the residual graph (called an *augmenting path*), pushes the maximum possible flow along it and updates the residual graph. It stops when there is no directed path from the source to the sink in the residual graph. Different algorithms choose different augmenting paths, and their running time is affected by the number of paths they use.

Why not take the paths produced by such an algorithm as our decomposition? There are two problems with this naive approach. First, the augmenting paths used by these algorithms are paths in the residual graph which contains arcs that are not in the flow, therefore these paths may not be contained in the original flow. See Figure 2 for such an example. Second, the number of these augmenting paths may be large, since even the fastest augmenting path algorithms use a quadratic (in the size of the graph) number of paths.

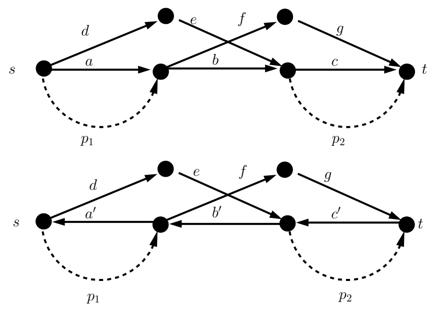

Fig. 2. In this flow f the capacities of all the arcs are 1. The dashed edges represent long paths. Assume that we run an augmenting path maximum flow algorithm, like, for example the algorithm of Edmonds and Karp on $G(f)$. The first augmenting path can be a, b, c. (Augmenting path algorithm augment flow on shortest residual paths.) After pushing one unit of flow on a, b, c we obtain the residual graph at the bottom of the figure. The next augmenting path is d, e, b', f, g which uses the arc b' which is opposite to b and is not in the original flow. The last augmenting path would be the concatenation of p_1, b and p_2.

A. Our Results

We consider two natural greedy algorithms for the problem, also studied by [18]. One picks the widest flow-path, removes it and recurses, we call it *greedy width*. The other picks a shortest (by latency) path, removes it and recurse, we call it *greedy length*.

We show that in the worst case these greedy algorithms produce a large decomposition compared to the optimal one. Specifically, the number of paths which they produce could be as large as $\Omega(\sqrt{m})OPT$ where m is the number of arcs and OPT is the size of the optimal decomposition. Furthermore, this bad performance occurs even if we want to decompose only a constant fraction of the flow.

We give a new algorithm similar to the greedy algorithms mentioned above, and prove that for any ϵ it decomposes $(1-\epsilon)$ fraction of the flow into at most $O(\frac{OPT}{\epsilon^2})$ paths. In particular, for an appropriate choice of parameters, it can decompose $1/3$ of the flow into at most OPT paths (see Section V). We consider two versions of this algorithm, one which we call the *bicriteria width* algorithm which is similar to the greedy width algorithm and the other which we call the *bicriteria length* algorithm which is similar to the greedy length algorithm.

It is known that the problem of decomposing a flow into the minimum number of paths is strongly NP-hard by a reduction from 3-partition [18]. But notice that the 3-partition problem is easy if all the flow values on the arcs are powers of 2, or if there are constantly many different values. So the reduction of [18] does not indicate what is the complexity of our problem in these cases, which are common in real networks.

We show that even when there are only three different flow values on the arcs (and even if these values are only 1, 2, or 4) then the problem is NP-hard. Furthermore, our reduction also shows that it is hard to approximate the problem better than some fixed constant, i.e. there is some fixed constant β such that no algorithm can guarantee a decomposition with less than $\beta \cdot OPT$ paths in polynomial time unless $P = NP$.

In contrast, we show that if there are only two flow values x and y, such that x divides y, then there is a simple algorithm that finds an optimal decomposition.

We implemented the greedy algorithms and our new approximation algorithms and compared their performance on data extracted from Google's backbone network and on synthetically generated layered networks, representing intra data center networks.

For the Google network, we used a dataset consisting of a few hundred demands observed at a particular time period. For each such a demand, we generated a fractional multicommodity flow using an LP solver. We decomposed various fractions of each of the flows using these algorithms and compared the size of the decomposition they produce and the average and maximum latency of the paths. Our findings are as follows:

1) The greedy width algorithm produces the most compact decompositions on the data we used, with the bicriteria width algorithm lagging behind by a few percent. This indicates that flows on which the greedy width algorithm decomposes badly are unlikely to occur in real data. The greedy length and the bicriteria length algorithms perform 10-20% worse than their width counterparts. To eliminate the risk of getting a bad decomposition while using the greedy width algorithm it may make sense to run both the greedy width algorithm and our new bicirteria width algorithm and select the smaller decomposition.
2) The maximum and average latency of the paths produced by the greedy width and the bicriteria width algorithms were comparable. Somewhat counter-intuitively the maximum and average latency of the greedy length and the bicriteria length algorithms were slightly larger. This is a side affect of the fact that the decomposition produced by these algorithms were considerably larger This shows that on the data we tested, optimizing the number of paths does not sacrifice latency substantially.
3) The number of paths required increases exponentially with the fraction of the flow we decompose (again on the tested data sets).

B. Related work

Vatinlen et al. considered exactly the same problem as we do, i.e. minimizing the number of paths. They gave an example showing that counter-intuitively, by eliminating cycles from the flow, the size of the optimal solution can increase. They defined a decomposition to be *saturating* if we can obtain it by taking a source-sink path, pushing as much flow as we can along it, removing it from the flow and repeating this process. In particular the greedy width and the greedy length algorithms that we defined above produce saturating decompositions.

Vatinlen et al. show that there may not be a saturating optimal solution. They also show that the size of a saturating decomposition is at most $m - n + 2$ where m is the number of arcs and n is the number of nodes,[2] and consists of $O(n)$ more paths than OPT.

Vatinlen et al. also defined and implemented the greedy width and the greedy length algorithms. They compared their performance on random networks and showed that for large enough networks with relatively large average degree the greedy width is slightly better ($< 3\%$), and they both were close to the upper bound of $m - n + 2$.

Hendel and Kubiak [9] in an unpublished manuscript consider a similar problem in which the input flow is acyclic, there is a nonnegative cost associated with each arc, and the goal is to find a decomposition in which the maximum cost of a path is minimum. The networking interpretation of this objective is to minimize the maximum latency of a path in the decomposition. They give several results classifying the complexity of this problem (for details see their manuscript).

The problem considered by Hendel and Kubiak is easier if the lengths of all arcs are the same. In this case we want to minimize the number of hops of the path with the largest number of hops in the decomposition. One can construct a polynomial algorithm for this problem using linear programming. We write an LP for the maximum possible flow on paths of at most k hops. This LP has a variable for each path of at most k hops so it may have exponentially many variables. The dual of this LP has a constraint per path of at most k hops which requires that the sum of the dual variables associated with the arcs on the path is at least 1. There is a polynomial separation oracle for this dual so we can solve it using the ellipsoid algorithm [10], [8]. To find the smallest k such that all the given flow can be covered by paths with at most k hops we perform binary search on k solving an LP as above in each iteration of the search.

Another possible objective is to minimize the average weighted latency of the paths in the decomposition (where the weight of each path is the amount of flow it covers). However, it is not hard to see that all the decompositions of a given flow have the same average weighted latency. This average equals to the sum over the arcs of the latency of the arc times its flow value.

Minimizing the nonweighted average latency and the maximum number of paths through each vertex are also interesting objectives which we do not consider in this paper and as far as we know have not been addressed.

A problem related to ours is the maximum unsplittable flow problem introduced by Kleinberg [11]. In this problem we try to find a flow which can be decomposed into a small number of paths. In contrast, recall, that in our setting the flow is **given as the input** and we cannot change it. Kleinberg studied a single-source version of the problem where we are given a single source and many terminals each with demand associated with it. The problem is to decide whether the demand to each terminal can be routed on a single path so that capacity constraints are satisfied. Kleinberg gave approximation algorithms for some optimization versions of this problem. Many variations of the problem were introduced since Kleinberg's work, including unsplittable multicommodity flow, variants in which we allow more than one path per commodity, and varying optimization criteria. (for recent work see [12], [2] and the references there)

Mirrokni et al. [14] studied a network planning problem in which one goal is to minimize the number of paths in multi-path routing setting such as an MPLS network. They formulate the problem as a multicommodity flow problem and derive from the formulation that an optimal solution can be decomposed into at most $m+k$ paths, where k is the number of commodities and m is the number of routes. They further try to reduce the number of routes by using an integer programming formulations. They show that their approach is effective on various simulated topologies.

II. Preliminaries

The input to our algorithm is an st-flow f represented as a graph $G(f)$ with two special vertices s and t called the source and the sink, respectively. Each arc e has a nonnegative flow-value $f(e)$ associated with it. For every vertex v other than s and t the sum of the flow-values on the arcs incoming to v equals the sum of the flow values on the arcs outgoing from v. The total flow on arcs outgoing from s minus the total flow on arcs incoming to s is the *value of* f. This is equal to the total flow incoming to t minus the total flow outgoing from t.

A *flow-path* in f is a path p from s to t in $G(f)$ together with a value which is smaller than the flow-value on all the arcs along p.

A *decomposition of an st-flow* f is a collection of flow-paths with the following property: If we take the union of these paths, and associate with each arc e in this union the sum of the values of the paths containing e we get an st-flow f'. The flow f' is contained in f and its value is equal to the value of f.[3] Note that the difference between f and f' is a collection of flow cycles and if f is acyclic then f must be equal to f'.

Our goal is to find a decomposition of f with the smallest possible number of paths. We point out that removing cycles from f may change the size of the smallest solution as shown in [18].

Let G be a graph with capacity $c(e)$ associated with each arc e and two special vertices s and t. An st-flow in G is an st-flow f such that $G(f)$ is a subgraph of G and for each arc e, $f(e) \leq c(e)$. A *maximum st-flow in* G in an st-flow in G with maximum value.

[2]In fact they showed this for a more general family of independent decompositions, where the incidence vectors of the paths, in the space where each coordinate is an arc, are independent.

[3]A flow f' is contained in f if each arc in f' is also in f and its flow-value in f' is smaller than its flow-value in f.

To simplify the presentation, we will use flow instead of st-flow in the rest of the paper.

We will use the following basic lemma. The proof is straight forward and omitted.

Lemma II.1. *Let G be a graph in which all capacities are multiples of x, and let f be a maximum flow in G of value F. Then F is a multiple of x and we can decompose f into exactly F/x paths each routing x flow. Furthermore, any collection of paths each routing x flow can be completed to such a decomposition of the flow.*

III. Two Value Capacities

A simpler version of the flow decomposition problem is when all the capacities in G have one of two possible values, x and y, such that x divides y, i.e. $y = kx$ for some integer $k > 1$.

We can find an optimal decomposition in this case with the following algorithm. We first form the subgraph G_y of G (which is not necessarily a flow) induced by all arcs of capacity y. We find a maximum flow F_y from s to t in G_y. We decompose F_y into paths of value y, and add these paths into our decomposition. This is possible by Lemma II.1. Then we subtract F_y from G and get a flow G'. Notice that since x divides y all capacities in G' are multiples of x. So using Lemma II.1 again we decompose G' into paths of flow value x and add these paths into our decomposition. We now prove that this algorithm indeed find the optimal solution.

Theorem III.1. *The algorithm described above produces an optimal solution.*

Proof: Let F_1 be the part of F that OPT routes on paths of value $> x$. Note that each such path routes at most y flow. So $F - F_1$ of the flow OPT routes on paths of value $< x$. It follows that

$$OPT \geq \frac{F_1}{y} + \frac{F - F_1}{x} \tag{1}$$

To minimize the right hand side of Equation (1) we want to have F_1 as large as possible. However since all paths of value $> x$ that OPT uses must be in the subgraph induced by the arcs of flow value y, we get that F_1 cannot be larger than the maximum flow in that subgraph. By the definition of the algorithm this maximum flow equals to $\#(y) \cdot y$, where $\#(y)$ is the number of paths of value y used by the algorithm. Substituting this into equation 1 we get that

$$OPT \geq \frac{F_1}{y} + \frac{F - F_1}{x} \geq$$
$$\frac{\#(y) \cdot y}{y} + \frac{F - \#(y) \cdot y}{x} \quad = \#(y) + \#(x) = ALG$$

which concludes the proof. ■

IV. Greedy Approaches

It is easy to verify that when we take a flow-path and subtract it from the flow the result is still a flow. Based on this fact one can come up with many greedy algorithms to decompose a flow. Each such algorithm finds the best flow-path according to some criteria, add it to the decomposition, removes it from the flow and continue decomposing the remaining flow.

Maybe the most natural among these greedy algorithms are the *greedy length* algorithm and the *greedy width* algorithm. Let $G(f)$ be the graph of the current flow f. (Note that $G(f)$ changes as we subtract flow-paths that we accumulate in our decomposition.) In the greedy length we find a shortest path from s to the t in $G(f)$, according to some latency measure on the arcs such as RTT (Round Trip Time). We route the maximum possible flow along this path, so that when we remove this path, at least one arc is completely removed from the flow.

The greedy width finds the "thickest" path from s to t in $G(f)$, which is a path that can carry more flow than any other path from s and t. Again, we route the maximum possible flow along this path, so that at least one arc is completely removed from the flow when we add the path to the decomposition.

We implement greedy length and the greedy width by running a version of Dijskstra's single source shortest path algorithm in each iteration [5]. For the greedy length we run the standard version of Dijskstra's algorithm using RTTs as the weights of the arcs. With the greedy width we need a modified version of Dijskstra for the bottleneck shortest path problem. Here the weight of each arc is the flow that it carries in $G(f)$. The modified version of Dijskstra for the bottleneck shortest path problem uses the width of the thickest flow-path from s to v as the key of v in the heap (rather than the length of this path in the standard version). Note that asymptotically faster algorithms for the bottleneck shortest path problem are known [6] and also for the standard single source shortest path problem when we assume integer weights [17]. These algorithms are more complex and may be practical only for very large graphs.

Both greedy approaches remove at least one arc from $G(f)$ for each path they pick, so they will stop after at most m iterations (where m is the number of arcs in the flow we started out with). Furthermore in the decomposition they produce we have at most m paths.

A. The worst case performance of the greedy algorithms

The next intriguing question that one should understand before using these greedy algorithms is how large could be the decomposition which they produce compared to the optimal one. We note that the greedy length may produce a decomposition with paths of rather small latency and we will consider this in Section VII. But our main objective in this paper is to minimize the number of flow-paths so we first consider the performance of the greedy algorithms with respect to the number of paths which they produce.

Lets look at the example shown in Figure 3. In this flow there are only 3 possible flow values on the arcs which are x, $2x$ and 1. There are $k+1$ arcs of flow $2x$, namely $a_i \to a_{i+1}$, for $1 \leq i \leq k-1$, $s \to a_1$, and $a_k \to t$, that together form a path from s to t. There are $k/2$ arcs from s to every node

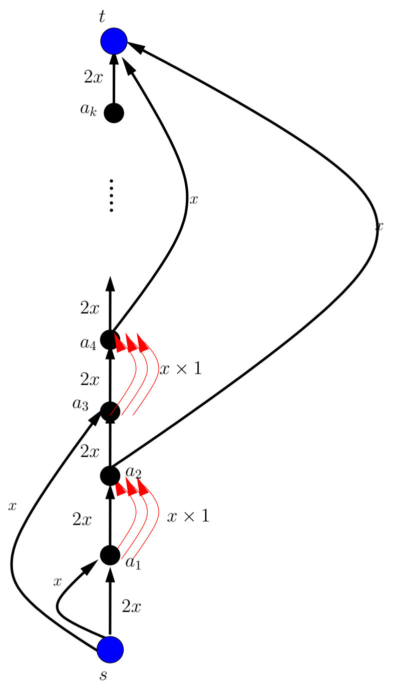

Fig. 3. The performance of the greedy algorithm is bad on this example

a_{2i-1}, and $k/2$ arcs going from a_{2i} to t. Finally, there are x parallel arcs of flow 1 going from a_{2i-1} to a_{2i}. One can easily verify that this is indeed a legal flow of value $kx/2 + 2x$.

The minimal number of paths this flow can be decomposed to is $k/2+x+1$. This can be done by routing x units of flow on the path $s \to a_1 \to \ldots \to t$ which is half the maximum possible flow we can send along this path; x units of flow on each one of the $k/2$ paths $s \to a_{2i-1} \to a_{2i} \to t$ using the remaining x units of flow of the arcs $a_{2i-1} \to a_{2i}$ with flow $2x$; and finally, x paths $s \to a_1 \to a_2 \ldots \to t$ using the parallel arcs between a_{2i-1} and a_{2i}, each carrying 1 unit of flow. Altogether, we get $k/2 + x + 1$ paths.

The greedy width will use $kx/2 + 1$ paths. It will make a mistake and route $2x$ units of flow on the path $s \to a_1 \to a_2 \ldots \to t$. Now, it will have to route the remaining $kx/2$ units of flow on paths of the form $s \to a_{2i} \to a_{2i+1} \to t$ each carrying only one unit of flow. So routing the remaining flow will take $kx/2$ paths.

Now if we set x to some value larger than say, $k/2$, we get that the greedy width uses $\Omega(k) \cdot OPT$ paths to decompose the flow. The resulting flow has $m = \Theta(k^2)$ arcs (and vertices if we eliminate the parallel arcs by subdividing them).

Furthermore, with this rather large value of x we also get that most of the flow is routed by the greedy width on paths of value 1. So even if we compare the number of paths that greedy width needs to carry only a constant fraction of the flow for any fixed constant then we get that it uses $\Omega(k) \cdot OPT$ paths. The following theorem summarizes this result.

Theorem IV.1. *There are flows G of m arcs on which the approximation ratio of greedy width is $\Omega(\sqrt{m})$ even if greedy width is required to decompose only a constant fraction of the flow, for any fixed constant.*

If the latency of the arcs of flow-value $2x$ is small relative to the latency of the other arcs then the greedy length would also pick the path of width $2x$ first and thereby produce a large decomposition of this example. We have a different example in which the latency of all arcs is the same but the greedy length fails in a similar fashion. So we also have the following theorem.

Theorem IV.2. *There are flows G of m arcs on which the approximation ratio of greedy length is $\Omega(\sqrt{m})$ even if greedy length is required to decompose only a constant fraction of the flow.*

V. Bi-criteria Approximation

Let G_t be the subgraph of G containing all arcs of capacity at least t. Let $t_{2/3}$ be the maximum value of t such that the value of the maximum flow from s to t in G_t is at least $2F/3$. To simplify notation we refer to $G_{t_{2/3}}$ as $G_{2/3}$.

We round down all the capacities in $G_{2/3}$ to the largest possible multiple of $t_{2/3}$ and call the resulting graph $G'_{2/3}$. We have the following lemma.

Lemma V.1. *The value of the maximum flow from s to t in $G'_{2/3}$ is at least $F/3$.*

Proof: If we scale down the capacities in $G_{2/3}$ by a factor of 2 then we obtain a flow of value $F/3$ in the resulting graph simply by scaling down the flow of value $2F/3$ that we have in $G_{2/3}$ by its definition.

Consider an arc e in $G_{2/3}$. Let $c(e)$ be the capacity of e in $G_{2/3}$ and let $c'(e)$ be the capacity of e in $G'_{2/3}$. We claim that $c'(e) \geq c(e)/2$. This would imply that we can route the flow of value $F/3$ that we routed in $G_{2/3}$ with capacities divided by 2 also in $G'_{2/3}$.

From the definition of $G_{2/3}$ we know that there is some integer $k \geq 1$ such that $kt_{2/3} \leq c(e) \leq (k+1)t_{2/3}$. Dividing the upper bound by two we obtain that $c(e)/2 \leq (k+1)t_{2/3}/2$. Since for every $k \geq 1$, $(k+1)/2 \leq k$ we have that $c(e)/2 \leq kt_{2/3}$. But by the definition of $G'_{2/3}$, $c'(e) = kt_{2/3}$, so we get that $c(e)/2 \leq c'(e)$, and the claim follows. ■

In $G'_{2/3}$ all capacities are multiples of $t_{2/3}$ so by Lemma II.1 the maximum flow F' in $G'_{2/3}$ is also a multiple of $t_{2/3}$ and can be decomposed into $F'/t_{2/3}$ paths each routing $t_{2/3}$ units of flow. By Lemma V.1, $F' \geq F/3$ so $\lceil F/3t_{2/3} \rceil$ paths of the decomposition of F' route at least $F/3$ units of flow. So we get the following theorem.

Theorem V.2. *Given a flow f of value F the algorithm which we described decomposes f into at most OPT paths that together route at least $F/3$ units of flow.*

Proof: By the definition of $G_{2/3}$ we know that in the optimal decomposition of f at least $F/3$ units of flow are routed using paths each carrying at most $t_{2/3}$ units of flow. Therefore $OPT \geq \lceil F/3t_{2/3} \rceil$. Since our algorithm uses at most $\lceil F/3t_{2/3} \rceil$ paths the theorem follows. ■

We can generalize this result as follows.

Let $t_{1-\epsilon}$ be the maximum value of t such that the value of the maximum flow from s to t in G_t is at least $(1-\epsilon)F$. To simplify notation we refer to $G_{t_{1-\epsilon}}$ as $G_{1-\epsilon}$.

We introduce a new constant $0 \le \delta \le 1$ and round down all the capacities in $G_{1-\epsilon}$ to the largest possible multiple of $\delta t_{1-\epsilon}$ and call the resulting graph $G^{\delta}_{1-\epsilon}$. The following lemma generalizes Lemma V.1 (which is the case where $\delta = 1$ and $\epsilon = 1/3$).

Lemma V.3. *The value of the maximum flow from s to t in $G^{\delta}_{1-\epsilon}$ is at least $\frac{1-\epsilon}{1+\delta}F$.*

The proof follows the same lines of V.1 and is omitted.

In $G^{\delta}_{1-\epsilon}$ all capacities are multiples of $\delta t_{1-\epsilon}$ so by Lemma II.1 the maximum flow F' in $G^{\delta}_{1-\epsilon}$ is also a multiple of $\delta t_{1-\epsilon}$. By Lemma II.1 we can decompose F' into $F'/\delta t_{1-\epsilon}$ paths each routing $\delta t_{1-\epsilon}$ flow. Furthermore, any set of paths each routing $\delta t_{1-\epsilon}$ flow can be completed by additional paths each routing $\delta t_{1-\epsilon}$ flow so that all-together we route F' flow. By Lemma V.3, $F' \ge \frac{1-\epsilon}{1+\delta}F$ so by using $\lceil \frac{(1-\epsilon)F}{(1+\delta)\delta t_{1-\epsilon}} \rceil$ paths of the decomposition of F' we route at least $\frac{(1-\epsilon)F}{(1+\delta)}$ flow. The following theorem summarizes the properties of our algorithm.

Lemma V.4. *In $G^{\delta}_{1-\epsilon}$ we can route $\frac{1-\epsilon}{1+\delta}F$ flow in $\lceil \frac{(1-\epsilon)F}{(1+\delta)\delta t_{1-\epsilon}} \rceil$ paths each routing $\delta t_{1-\epsilon}$. Furthermore, starting with any set of paths in $G^{\delta}_{1-\epsilon}$ each routing $\delta t_{1-\epsilon}$ flow we can find additional paths each routing $\delta t_{1-\epsilon}$ flow so that all paths together route $\frac{1-\epsilon}{1+\delta}F$ flow.*

The following theorem follows from Lemma V.4.

Theorem V.5. *Given a flow f of value F we can decompose f into at most $\lceil \frac{1+\epsilon}{(1+\delta)\epsilon\delta} \rceil \cdot OPT$ paths. Each path routes $\delta t_{1-\epsilon}$ units of flow and together they route at least $\frac{(1-\epsilon)F}{1+\delta}$ units of flow.*

Proof: By the definition of $G_{1-\epsilon}$ we know that in the optimal decomposition of f at least ϵF units of flow are routed using paths each carrying at most $t_{1-\epsilon}$ units of flow. Therefore $OPT \ge \lceil \epsilon F/t_{1-\epsilon} \rceil$. By Lemma V.4 our algorithm uses at most $\lceil \frac{(1-\epsilon)F}{(1+\delta)\delta t_{1-\epsilon}} \rceil$ paths therefore the ratio between the number of paths that we use and the optimal number of paths is

$$\frac{\lceil \frac{(1-\epsilon)F}{(1+\delta)\delta t_{1-\epsilon}} \rceil}{\lceil \epsilon F/t_{1-\epsilon} \rceil} = \frac{\lceil \frac{(1-\epsilon)\epsilon F}{(1+\delta)\epsilon\delta t_{1-\epsilon}} \rceil}{\lceil \epsilon F/t_{1-\epsilon} \rceil} \le$$

$$\le \frac{\left\lceil \frac{(1-\epsilon)}{(1+\delta)\delta\epsilon} \right\rceil \cdot \lceil \frac{\epsilon F}{t_{1-\epsilon}} \rceil}{\lceil \epsilon F/t_{1-\epsilon} \rceil} = \left\lceil \frac{(1-\epsilon)}{(1+\delta)\delta\epsilon} \right\rceil$$

as required. ■

VI. Hardness Results

In this section we prove the following strong hardness result.

Theorem VI.1. *Let F be a flow such that on each arc the flow value is either 1, 2 or 4, and let k be an integer. Then it is NP-complete to decide if there exists a decomposition of F into at most k paths.*

Proof: For a variable x let $o(x)$ be the number of occurrences of the literal x, let $o'(x)$ be the number of occurrences of $\overline{x}$, and let $o_m(x) = \max\{o(x), o'(x)\}$.

Given an instance Φ of 3SAT with a set Z of N variables and a set C of M clauses we construct a flow F with $n = M + 2 + \sum_{x\in Z}(3 + 4o_m(x))$ vertices and $m = 4M + \sum_{x\in Z}(15o_m(x) + 12)$ arcs such that there is a decomposition of F into $\sum_{x\in vars} 5 + 3o_m(x)$ paths if and only if Φ is satisfiable.

For each variable x we construct the gadget shown in Figure 4. We have two vertices $s(x)$ and $t(x)$ which we connect with two parallel paths each containing $o_m(x)$ pairs of nodes $u_i(x)$ and $u'_i(x)$, $1 \le i \le o_m(x)$ in one path and $w_i(x)$ and $w'_i(x)$, $1 \le i \le o_m(x)$ in the other path. There is an arc from $s(x)$ to $u_1(x)$, from $u_i(x)$ to $u'_i(x)$ and from $u'_i(x)$ to $u_{i+1}(x)$ for every $1 \le i \le o_m(x) - 1$, and from $u'_{o_m(x)}(x)$ to $t(x)$. Similarly, there is an arc from $s(x)$ to $w_1(x)$, from $w_i(x)$ to $w'_i(x)$ and from $w'_i(x)$ to $w_{i+1}(x)$ for every $1 \le i \le o_m(x)-1$, and from $w'_M(x)$ to $t(x)$.

The source s is connected to all vertices $s(x_j)$, for each variable x_j, $1 \le j \le N$, and all vertices $t(x_j)$, $1 \le j \le N$ are connected to t. The flow on all arcs we specified so far is 4, and these would be the only arcs with flow 4 in F.

In addition we have four arcs with flow 1 from s to $s(x_j)$, $1 \le j \le N$, and from $t(x_j)$, $1 \le j \le N$ to t. We also have a pair of parallel arcs with flow 1 from $u_i(x)$ to $u'_i(x)$ and from $w_i(x)$ to $w'_i(x)$ for $1 \le i \le o_m(x)$.

For each variable x we have an additional vertex $a(x)$. We connect s to $a(x)$ with $o_m(x)$ arcs of flow 2 and $2o_m(x)$ arcs of flow 1. We also connect $a(x)$ to $u_i(x)$ and $w_i(x)$ for $1 \le i \le o_m(x)$ with an arc of flow 2.

For each clause c we have a vertex which we also call c. Each such vertex c is connected to t by four arcs, two of flow 1 and two of flow 2.

Last we have to specify the connection between the variable gadget and the clause vertices. We connect each vertex $u'_i(x)$ for $1 \le i \le o(x)$ to a clause vertex corresponding to a clause containing x. We make these connections such that each of these clause vertices is connected to a single vertex $u'_i(x)$. Similarly, we connect each vertex $w'_i(x)$ for $1 \le i \le o'(x)$ to a clause vertex corresponding to a clause containing $\overline{x}$, such that each of these clause vertices is connected to a single vertex $w'_i(x)$. If $o(x) < o_m(x)$ we connect each vertex $u'_i(x)$ for $o(x) + 1 \le i \le o_m(x)$. If $o'(x) < o_m(x)$ we connect each vertex $w'_i(x)$ for $o'(x) + 1 \le i \le o_m(x)$ to t. All arcs defined in this paragraph have flow 2.

We now show that Φ is satisfiable if and only if we can decompose F to $\sum_{x\in vars}(5 + 3o_m(x))$ paths.

We first assume that Φ is satisfiable and show a decomposition of F into $\sum_{x\in vars}(5 + 3o_m(x))$ paths. Consider an assignment that satisfies Φ.

Each path in our decomposition saturates an edge from s. The number of these arcs is exactly $\sum_{x\in vars} 5 + 3o_m(x)$. For a variable x which equals 1 there would be a pair of paths of value 1 to each clause vertex containing the literal x, and a path of value 2 to each clause vertex containing the literal $\overline{x}$.

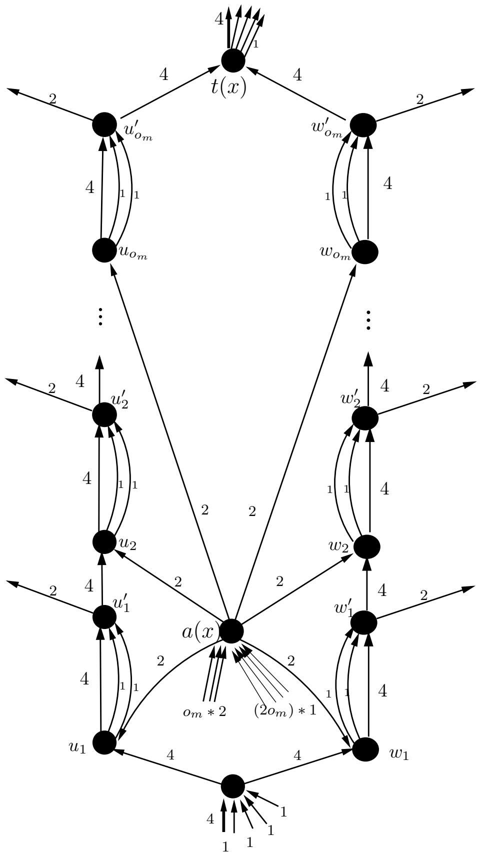

Fig. 4. A variable gadget in the reduction proving Theorem VI.1. Arc incoming to $s(x)$ and $a(x)$ are outgoing of the global source s. Arcs outgoing of the nodes u'_i and w'_i, either enter a clause gadget or enter the sink t. Arcs outgoing of $t(x)$ enter the sink t.

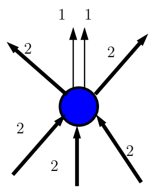

Fig. 5. A clause gadget in the reduction of Theorem VI.1. Incoming arcs arrive from the variable gadgets. Outgoing arcs enter the sink t.

For a variable x which equals 0 there would be a pair of paths of value 1 to each clause vertex containing the literal $\overline{x}$, and a path of value 2 to each clause vertex containing the literal x. Since the assignment is satisfiable each clause has at least one literal which is 1 and therefore at least one pair of paths of value 1 entering it. Therefore we can route these paths through the clause vertices without splitting them.

For each variable x if $x = 1$ we take a single path of value 4 from s to $s(x)$ to $u_i(x)$ and $u'_i(x)$, $1 \leq i \leq o_m(x)$ to $t(x)$, and four paths of value 1 from s to $s(x)$ to $w_i(x)$ and $w'_i(x)$, $1 \leq i \leq o_m(x)$ to $t(x)$. Two of these paths use the arcs from $w_i(x)$ to $w'_i(x)$ of flow 1 and the other two use the arc of flow 4. If $x = 0$ we take a single path of value 4 from s to $s(x)$ to $w_i(x)$ and $w'_i(x)$, $1 \leq i \leq o_m(x)$ to $t(x)$, and four paths of value 1 from s to $s(x)$ to $u_i(x)$ and $u'_i(x)$, $1 \leq i \leq o_m(x)$ to $t(x)$. Two of these paths use the arcs from $u_i(x)$ to $u'_i(x)$ of flow 1 and the other two use the arc of flow 4.

If $x = 1$ we take $2o(x)$ paths of value 1 from s to $a(x)$ to $u_i(x)$ and $u'_i(x)$, and then to a clause vertex containing the literal x, for $1 \leq i \leq o(x)$. If $o(x) < o_m(x)$ then we take $2(o_m(x) - o(x))$ paths of value 1 from $a(x)$ to $u_i(x)$ and $u'_i(x)$, and then to t for $o(x) + 1 \leq i \leq o_m(x)$. We also take $o'(x)$ paths of value 2 from s to $a(x)$ to $w_i(x)$ and $w'_i(x)$, and then to a clause vertex containing $\overline{x}$, for every $1 \leq i \leq o'(x)$. If $o'(x) < o_m(x)$ then we take $o_m(x) - o'(x)$ paths of value 2 from s to $a(x)$ to $w_i(x)$ and $w'_i(x)$, and then to t for $o'(x)+1 \leq i \leq o_m(x)$. Each of these paths uses an arc from $w_i(x)$ to $w'_i(x)$ of flow 4, and if the path goes through a clause vertex (which must correspond to a clause containing $\overline{x}$) it uses an arc of flow 2 outgoing from this clause vertex.

Symmetrically, if $x = 0$ we take $o(x)$ paths of value 2 from s to $a(x)$ to $u_i(x)$ and $u'_i(x)$, and then to a clause containing x, for $1 \leq i \leq o(x)$. If $o(x) < o_m(x)$ we take $o_m(x) - o(x)$ additional paths of value 2 from s to $a(x)$ to $u_i(x)$ and $u'_i(x)$, and then to t, for $o(x) \leq i \leq o_m(x)$. Each of these paths ships two units of flow on an arc from $u_i(x)$ to $u'_i(x)$ of flow 4, and if the path goes through a clause vertex (which must correspond to a clause containing x) it uses an arc of flow 2 outgoing from this clause vertex. We also take $2o'(x)$ paths of value 1 from s to $a(x)$ to $w_i(x)$ and $w'_i(x)$, and then to a clause vertex containing $\overline{x}$ for $1 \leq i \leq o'(x)$. If $o'(x) < o_m(x)$ we take $2(o_m(x) - o(x))$ paths of value 1 from $a(x)$ to $w_i(x)$ and $w'_i(x)$ to t, for $o'(x) \leq i \leq o'_m(x)$.

To show that this decomposition is well defined we have to argue that each path of value 2 arriving to a clause vertex can use an arc of flow 2 outgoing from this vertex. Since the assignment satisfies Φ every clause has at least one literal which equals 1. This literal corresponds to two paths of value 1 arriving to the clause vertex. Therefore, the corresponding clause vertex has at most two paths of value 2 reaching it each using one of the arcs of flow 2 outgoing from the vertex.

We omit the other direction in this version due to lack of space. ∎

A variation of the proof of Theorem VI.1 using a reduction from a variant of MAX-3-SAT that is hard to approximate gives the following stronger result.

Theorem VI.2. *Let F be a flow such that on each arc the flow value is either* 1, 2 *or* 4. *Then there is an $\epsilon > 0$ such that it is NP-hard to find a decomposition of F of size at most $(1 + \epsilon)OPT$.*

VII. Experimental Results

We conducted experiments on two types of networks. The first one is the Google backbone production network. The second one is a synthetically generated layered network that resembles the topology inside a data center (which is a big cluster of interconnected machines).

Google's backbone network is one of the largest networks in the world today, recently mentioned as the second largest

ISP backbone in terms of overall traffic in Arbors list of the top 10 carriers of Internet traffic[4].

Our first set of experiments were conducted on this network using a real topology and the actual set of pop-level high priority demands. We used a linear program (LP) solver with the objective of maximizing overall throughput to generate several hundred flows between the source-destination pairs. This models a traffic engineering setup in an MPLS network (as many large backbone networks are) where an ISP is seeking to optimize network utilization.

The LP solver outputs for each commodity the amount of flow through each arc in the network. As discussed before, we need to decompose these flows into distinct paths which then can be set as LSPs (Labeled Switched Paths) in the relevant routers. Routers have constraints on the number of LSPs which they can support, both due to table size restrictions in the router hardware, as well as CPU load when updating the tables as paths change.

We decomposed the flows obtained from the LP into paths using 4 algorithms: Greedy width, greedy length, bicriteria length and bicriteria width. Specifically, we decomposed $(1-\epsilon)$ of each flow separately for ϵ ranging from 0.5 to 0.001, and compared the aggregated (sum of all flows) number of paths used by each algorithm to decompose all the flows.

Our results show that on this input, greedy width finds the minimum number of paths amongst the algorithms compared. Bicriteria width achieves similar results. Bicriteria length lags behind and the greedy length finds almost twice as many paths, depending on ϵ. See the left plot in Figure 7.

For all the algorithms, the number of paths increases exponentially with the fraction of flow that they cover. To route 70% of the flows, greedy width uses an average of 3.34 paths. To cover 90% it needs an average of 5.8 paths, and to cover 99.99% of the flows almost 12 paths are used on the average per demand. The bicriteria algorithms behave similarly. The number of paths used by the greedy length also increase with the fraction of flow covered, but less rapidly. This can be due to the fact that this is the only algorithm that ignores the width of the paths completely (bicriteria length decomposes a subgraph of the original flow graph consisting of wide arcs).

The maximum and average latency of the decompositions found by each of the algorithms are depicted in Figure 6 . We found that the latencies for all algorithms are comparable. When looking at the average latencies, the greedy length and bicriteria length perform slightly worse than the greedy width and bicriteria width. This seems counter intuitive, but can be explained by the fact that the number of paths found by the greedy width and bicriteria width was smaller than the number of paths found by the greedy length and bicriteria length.

The second network we tested models an intra cluster topology. The topology inside a data-center is typically organized as a tree-like layered network to allows one to can get a large scale communication network. Common architectures today consist of several layers of switches forming a tree-like topology [1], where the number of the layers depends on the size of the cluster.

The *edge-tier* consists of the machines themselves, each node (leaf) in this layer corresponds to a rack with a few dozens machines. The next layer is an *aggregation layer* consisting of switches where each rack is connected to a constant number d_1 of switches in the aggregation layer and each switch of the aggregation layer is connected to a constant number d_2 of racks. Typically $d_1 \ll d_2$, so the number of switches in the aggregation layer is d_1/d_2 fraction of the number of racks. There may be several such layers and the last layer is the *core* or the root layer. It consists of a similar number of switches as the aggregation layers, and interconnects switches in the previous layer. Data-centers in the same geographic location are further connected using a similar tree-like layered network.

Several routing strategies have been suggested for data centers including a traffic engineering approach based on open flow [3]. The topology of the data center forces each flow from a leaf node in one data-center to a leaf node in the same cluster to look as a layered network as well. Each layer of the flow consists of the switches that the flow goes through either on the way up from the source leaf, or on its way down to the destination leaf.

To test the decomposition algorithms in this setting, we generated a network that consists of 10 layers, each containing 5 switches. Two adjacent layers are connected as a full bipartite graph. All arcs between adjacent layers have the same latency. To generate a specific flow between 2 racks we combined 100 paths that connect the source and the destination each having a uniformly distributed flow value into a single flow. This ensures us that an optimal decomposition uses at most 100 paths.

The results for this scenario were similar to the results for the Google backbone network. The right plot in Figure 7 shows the average number of paths found by the different algorithms for 10 random layered networks as above. Again, the greedy width found the smallest number of paths and bicriteria width performed slightly worse. Greedy length and bicriteria length performed 3 to 6 times worse than greedy width, depending on ϵ. The bad behavior of greedy length and bicriteria length in this case is due to the fact that all the source-destination paths have the same latency. This causes greedy length and bicriteria length to effectively choose paths arbitrarily when decomposing this flow.

For greedy width and bicriteria width, similarly to the Google backbone network, we see an exponential increase in the number of paths with the fraction of the flow routed. This increase is less rapid and closer to linear for the greedy length and bicriteria length, which is again a side effect of the arbitrary choice of paths of these algorithms when decomposing this flow.

We also notice that the number of paths found by the bicriteria algorithms does not necessarily monotonically increase as a function of the amount of flow routed. For instance, the bicriteria width algorithm routes 85% of the flow using 40

[4]See http://asert.arbornetworks.com/2010/03/how-big-is-google/.

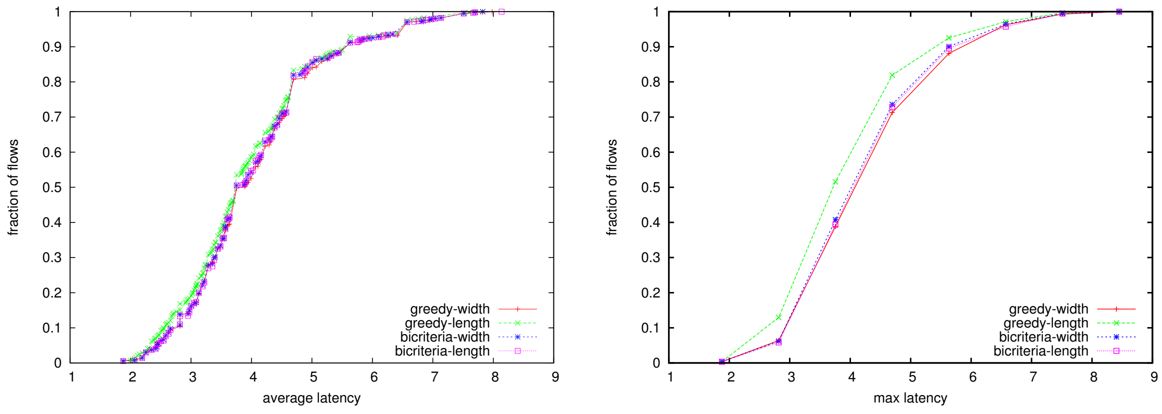

Fig. 6. Cumulative distribution of maximum and average latency of the paths that cover 70% of the flows in the Google network.

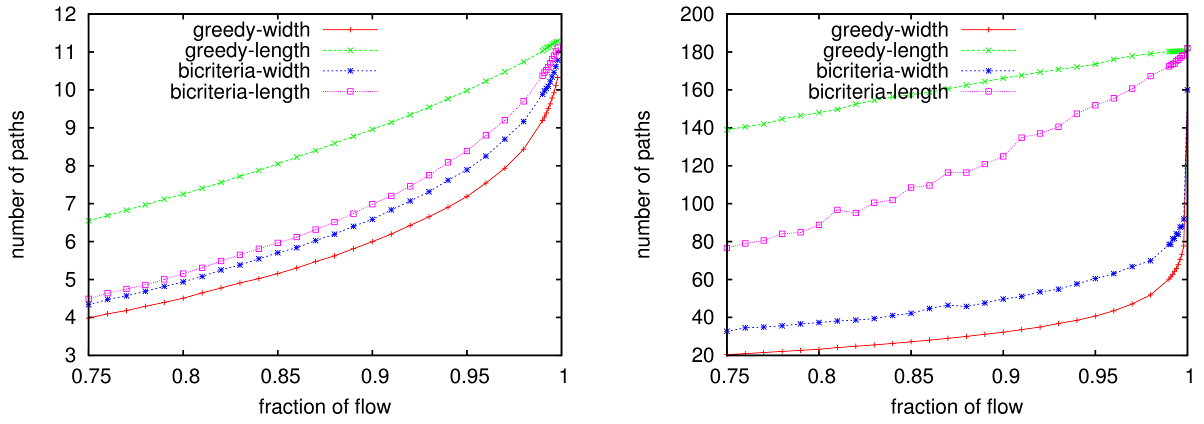

Fig. 7. The number of paths in the decomposition for different values of ϵ. The left figure is for an aggregation over few hundred commodities in the Google backbone network. The right figure is for a graph consisting of 10 layers with 5 vertices in each, representing intra data center network. For each flow there is a decomposition with at most 100 paths.

paths, but routes 86% of the flow with less than 40 paths. This is due to the fact that when increasing the amount of flow that we have to cover we increase the subgraph that the bicreteria algorithms pick. A larger subgraph contains more paths and maybe counter-intuitively may have a smaller decomposition.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[2] G. Baier, E. Köhler, and M. Skutella, "The k-splittable flow problem," *Algorithmica*, vol. 42, pp. 231–248, 2005.

[3] T. Benson, A. Anand, A. Akella, and M. Zhang, "The case for fine-grained traffic engineering in data centers," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking (INM/WREN'10)*. USENIX Association, 2010.

[4] *MPLS Traffic Engineering (TE) – Automatic Bandwidth Adjustment for TE Tunnels*, Cisco.

[5] E. W. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[6] H. N. Gabow and R. E. Tarjan, "Algorithms for two bottleneck optimization problems," *J. Algorithms*, vol. 9, pp. 411–417, September 1988.

[7] L. D. Ghein, *MPLS Fundamentals*. Cisco Press, 2006.

[8] M. Grötschel, L. Lovász, and A. Schrijver, "The ellipsoid method and its consequences in combinatorial optimization," *Combinatorica*, vol. 1, no. 2, pp. 169–197, 1981.

[9] Y. Hendel and W. Kubiak, "Decomposition of flow into paths to minimize their length."

[10] L. Khachiyan, "A polynomial time algorithm in linear programming," *Soviet Math. Dokl.*, vol. 20, pp. 191–195, 1979.

[11] J. M. Kleinberg, "Single-source unsplittable flow," in *FOCS*, 1996, pp. 68–77.

[12] S. G. Kolliopoulos, "Edge-disjoint paths and unsplittable flow," in *Handbook of Approximation Algorithms and Metaheuristics*, ser. Chapman and Hall/CRC, T. F. Gonzalez, Ed., 2007.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, March 2008.

[14] V. S. Mirrokni, M. Thottan, H. Uzunalioglu, and S. Paul, "A simple polynomial time framework for reduced-path decomposition in multipath routing," in *INFOCOM*, 2004.

[15] E. Osborne and A. Simha, *Traffic Engineering with MPLS*. Pearson Education, 2002.

[16] A. Premji, *Using MPLS Auto-bandwidth in MPLS Networks*, Juniper Networks, Sunnyvale, CA 94089 USA.

[17] M. Thorup, "Integer priority queues with decrease key in constant time and the single source shortest paths problem," *J. Comput. Syst. Sci.*, vol. 69, no. 3, pp. 330–353, 2004.

[18] B. Vatinlen, F. Chauvet, P. Chrétienne, and P. Mahey, "Simple bounds and greedy algorithms for decomposing a flow into a minimal set of paths," *European Journal of Operational Research*, vol. 185, pp. 1390–1401, 2008.