Multi-Commodity Flow with In-Network Processing

Moses Charikar, Yonatan Naamad, Jennifer Rexford, and X. Kelvin Zou

Department of Computer Science, Princeton University
{moses,ynaamad,jrex,xuanz}@cs.princeton.edu

1 Introduction

In addition to delivering data efficiently, today's computer networks often perform services on the traffic in flight to enhance security, privacy, or performance, or provide new features. Network administrators frequently install so-called "middleboxes" such as firewalls, network address translators, server load balancers, Web caches, video transcoders, and devices that compress or encrypt the traffic. In fact, many networks have as many middleboxes as they do underlying routers or switches. Often a single conversation, or *connection*, must traverse multiple middleboxes, and different connections may go through different sequences of middleboxes. For example, while Web traffic may go through a firewall followed by a server load balancer, video traffic may simply go through a transcoder. In some cases, the traffic volume is so high that an organization needs to run multiple instances of the same middlebox to keep up with the demand. Deciding how many middleboxes to run, where to place them, and how to direct traffic through them is a major challenge facing network administrators.

Until recently, each middlebox was a dedicated appliance, consisting of both software and hardware. Administrators tended to install these appliances at critical locations that naturally see most of the traffic, such as the gateway connecting a campus or company to the rest of the Internet. A network could easily have a long chain of these appliances at one location, forcing all connections to traverse every appliance—whether they need all of the services or not. In addition, placing middleboxes only at the gateway does not serve the organization's many *internal* connections, unless the internal traffic is routed circuitously through the gateway. Over the last few years, middleboxes are increasingly *virtualized*, with the software service separate from the physical hardware. Middleboxes now run as virtual machines that can easily spin up (or down) on any physical server, as needed. This has led to a growing interest in good algorithms that optimize the (i) *placement* of middleboxes over a pool of server resources, (ii) *steering* of traffic through a suitable sequence of middleboxes based on a high-level policy, and (iii) *routing* of the traffic between the servers over efficient network paths [6].

Rather than solving these three optimization problems separately, we introduce—and solve—a joint optimization problem. Since server resources are fungible, we argue that each compute node could subdivide its resources arbitrarily across any of the middlebox functions, as needed. That is, the *placement* problem is more naturally a question of what fraction of each node's computational (or memory) resources to allocate to each middlebox function. Similarly, each connection can have its middlebox processing performed on any node, or set of nodes, that have sufficient resources. That is, the *steering* problem is more naturally a question of how to decide which nodes should devote a share of its processing resources to a particular portion of the traffic. Hence, the joint optimization problem ultimately devolves to a new kind of *routing* problem, where we must compute paths through the network based on both the bandwidth and processing requirements of the traffic between each source-sink pair. That is, a flow from source to sink must be allocated (i) a certain amount of bandwidth on every link in its path and (ii) a total amount of computation across all of the nodes in its path.

1.1 The Problem We can abstract the flow in-network process problem in the following way: there is a flow demand with multi-sources and multi-sinks, and each flow requires a certain amount of in-network (MBox) processing. The in-network processing required for a flow is proportional to the flow size and without losing generality, we assume one unit of flow requires one unit of processing. For a flow from a source to a sink, we assume it is an aggregate flow so the routing and in-network processing for a flow are both divisible. In this model there are two types of constraints: edge capacity and vertex capacity, which represents bandwidth and MBox processing capacity. A feasible flow pattern satisfies: the sum of flows on each edge is bounded by the edge capacity, the sum of in-network process done at each vertex is bounded by the vertex capacity, and the processing done at all vertices for a flow should be equal to flow size and the processing has to be carried out by the flow.

Network design problems involve finding a network with a minimum cost that satisfies various properties, often related to routing and flow feasibility. The network design problem in this model is: for a multi-source and multi-sink flow demand, and a fixed amount of budget to install the processing capacities in the vertices, what is the optimal way to allocate them such that the flow pattern satisfies all the constraints.

Our model is a superset of multi-commodity flow model[4], that is, if we can solve this problem, we can naturally solve multi-commodity flow problem: simply assign each vertex with an infinite capacity it becomes an MCF problem. Many classic MCF design properties can be extended to this problem, such as MC-BB(multi-commodity buy-at-bulk) problem[1, 2, 3].

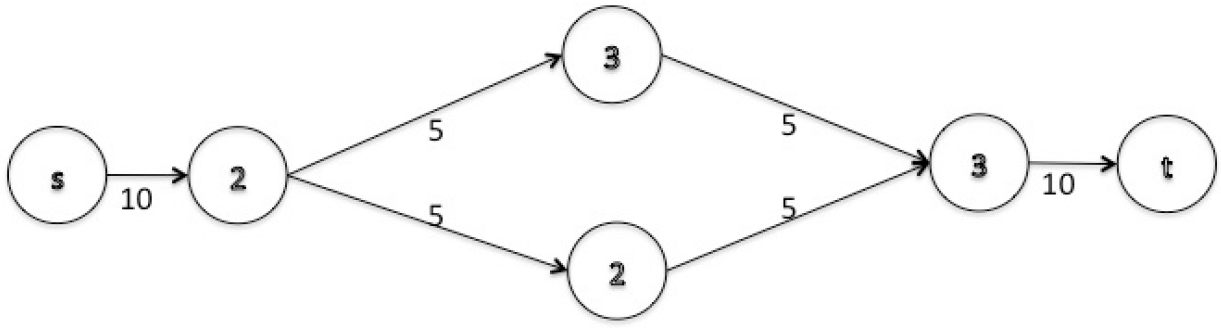

Figure 1: **Example for the Graph Model: given the edge and vertex constraints,** {5,10} **are edge capacities and {2,3} are vertex capacities, the maximum flow from source to sink in the example is 10, in this example all edges and vertices will be saturated.**

1.2 Notations To capture both the capacitated vertices and edges, we use the following notations: a directed graph $G(V, E)$ where each edge $e \in E$ has an edge bandwidth $B(e)$, and each vertex $v \in V$ has a processing capacity $C(v)$.

1.3 The results in this paper

2 Flow Steering and Processing Allocation Problem

2.1 The basic problem We begin by introducing the routing and steering problem for the above graph model. In this problem, we are given a directed graph $G = (V, E)$ along with edge capacities $B : E \to \mathbb{R}^+$, vertex capacities $C : V \to \mathbb{R}^+$, and a collection of demanded integer flows $D = \{(s_1, t_1, k_1), (s_2, t_2, k_2), \cdots\} \subseteq V \times V \times \mathbb{R}^+$. While the edge capacities are used in a manner

entirely analogous to its uses in standard multicommodity flow problems, we also require that each unit of flow undergo one unit of processing at an intermediate vertex. In particular, while edge capacities limit the *total* amount of flow that may pass through an edge, vertex capacities only bottleneck the amount of processing that may be done at a given vertex, regardless of the total amount of flow that uses the vertex as an intermediate node. We are then asked to route the flows with minimum congestion, the congestion can be interpreted in utilization. *I don't quite understand this. -YN You can ignore this, already covered in your text -KZ* The goal is then either to route as much flow as possible, or to satisfy all flow demand subject to appropriate congestion-minimization constraints. Though ignoring vertex capacity constraints reduces our class of problems to those of the standard Multicommodity Flow variety, the introduction of these constraints forms a new class of problems that (to our knowledge) has not yet been studied in the literature. Without loss of generality, we assume the flow demand equals processing demand. *I think this should be moved elsewhere, along with a brief sentence explaining why it's w.l.o.g. -YN I think you already covered this part in previous text, KZ* Throughout this paper, we make the practice inspired assumption that each unit of flow is in fact an aggregate of many small *microflows*, and thus all flows are splittable. What can be said about this problem subject to unsplittable flow demand remains an interesting open question.

2.1.1 Flow Maximization In **??**, we show how to express the maximization version of the problem both as an *edge-based* and as a *path-based* linear program. While neither of these constructions are particularly difficult, it's not immediate that either LP is itself sufficient for solving the problem of identifying the actual routing problem in polynomial time, as the flow-based LP does not identify the individual paths while the path-based LP may end up being of exponential size. In **??**, we solve this problem by showing how to construct actual routing patterns from the solution to the edge-based LP. We summarize this result in the following theorem.

THEOREM 2.1. *There exists a polynomial-sized linear program solving the Maximum Processed Flow problem. Further, the full routing pattern can be extracted from the LP solution in polynomial time. Should we bother ensuring we don't have bit complexity issues with real numbers? -YN KZ: I think the proof takes exponential time, but full routing pattern is contained in the LP solution, in particular, at each node, we take all the incoming flows, divide them into processed and unprocessed, and process part of the unprocessed flow, and then dispense the unprocessed and processed flow based on the flow routing decision $w(e)$ and $f(e)$, but however this approach somehow depends on flow size t and Idk anyway we can get away with it? Seems like if processing of flow is already linear to the flow size, so it may be okay?*

Notations:
For a *path-based* solution, P_i: a set of paths π for flow i; $f(\pi)$: flow size for each path π and $p(\pi, v)$: the processing work done at v on path π.
For an *edge-based* solution, $f_i(e)$: flow size for flow i at edge e, $p_i(v)$: the processing work done at v for flow i, and $w_i(e)$ is the process work demand at edge e for flow i.

The general linear programming solutions for a multi-commodity flow with in-network processing demand is the following:

Linear programming solution should be totally correct, but you can double check

Path-based formulation:

$$\text{MAX:} \sum_{i=1}^{K} \sum_{\pi \in P_i} f(\pi) \tag{2.1a}$$

Subject to:

$$\forall i, \forall \pi \in P_i; \sum_{v \in \pi} p(\pi, v) = f(\pi) \tag{2.1b}$$

$$\forall i, \forall e; \sum_{i} \sum_{\pi \in P_i : e \in \pi} f(\pi) \leq B(e) \tag{2.1c}$$

$$\forall i, \forall v; \sum_{i} \sum_{\pi \in P_i : v \in \pi} p(\pi, v) \leq C(v) \tag{2.1d}$$

$$\forall i, \forall \pi \in P_i, \forall v; p(\pi, v) \geq 0 \tag{2.1e}$$

Edge-based formulation:

$$\text{MAX:} \sum_{i=1}^{K} \sum_{e=(s,v)} f_i(e) \tag{2.2a}$$

Subject to:

$$\forall i, \forall v \neq s, t; \sum_{in} f_i(e) = \sum_{out} f_i(e) \tag{2.2b}$$

$$\forall i, \forall v; p_i(v) = \sum_{in} w_i(e) - \sum_{out} w_i(e) \tag{2.2c}$$

$$\forall i, \forall e; \sum_{i} f_i(e) \leq B(e) \tag{2.2d}$$

$$\forall i, \forall v; \sum_{i} p_i(v) \leq C(v) \tag{2.2e}$$

$$\forall i, \forall (s \to v); w_i(e) = f_i(e) \tag{2.2f}$$

$$\forall i, \forall (v \to t); w_i(e) = 0 \tag{2.2g}$$

$$\forall i, \forall e; w_i(e), f_i(e) - w_i(e), p_i(v) \geq 0 \tag{2.2h}$$

Not sure this makes sense to you or not The *path-based* solution takes paths as an abstraction, and it captures the properties in an explicit way; the *edge-based* solution, on the other hand can compute the max flow problem in a polynomial time. We further prove that they are equivalent and we can have an path decomposition algorithm to extract the information from *edge-based* solution to formulate the paths. The complexity of this algorithm is similar to Ford-Fulkerson, since we peel off Δf at each step.

Flow Size Changes after Processing: the basic max-flow model shows that in-network processing does not affect the traffic volume. However the change of flow size after processing is a common scene in networking, for example, encryption increases the flow size while compression and transcoding decrease the traffic size[5]. We can capture this aspect and integrate into the optimization formulation. In a generalized minimum cost circulation problem [7], there is a gaining factor. We apply the same idea, but one key difference in our model is that our model only applies the gaining factor to part of the flow to be processed at a vertex.

If for flow i there is a size change factor $r_i \in \mathbb{R}^+$, at each vertex, we have $\sum_{in} w_i(e) - \sum_{out} w_i(e) = p_i(v)$ and $\sum_{out}(f_i(e) - w_i(e)) - \sum_{in}(f_i(e) - w_i(e)) = p_i(v) * r_i$, . If $r_i = 1$, we have exactly the flow conservation $\sum_{out} f_i(e) - \sum_{in} f_i(e) = 0$.

2.1.2 Cost Minimization

I have a little trouble of telling the difference here, can you elaborate more about the convex function part in this section-KZ The minimization version of our problem, however, allows for nonlinear (and, in principle, non-convex) objective functions. In this paper, we deal with the *minimum congestion?* model, parameterized by two monotone, convex congestion measures $c_v : \mathbb{R}^+ \to \mathbb{R}^+$ and $c_e : \mathbb{R}^+ \to \mathbb{R}^+$. In this model, a vertex v with processing capacity $C(v)$ is assigned a congestion $c_v(\sum_{f \in F} f_v)$ and each edge is assigned congestion $c_e(\sum_{f \in F} f_e)$, where f_v and f_e are the amount of processing and amount of microflow that flow path f assigns to vertices v and e, respectively. Each *microflow*, in turn, is penalized according to the total amount of congestion it encounters among the edges it takes as well as its processing vertex. The goal is to feasibly route all requested units of flow while minimizing the sum of the penalties the various microflow encounter. *Since we can turn*

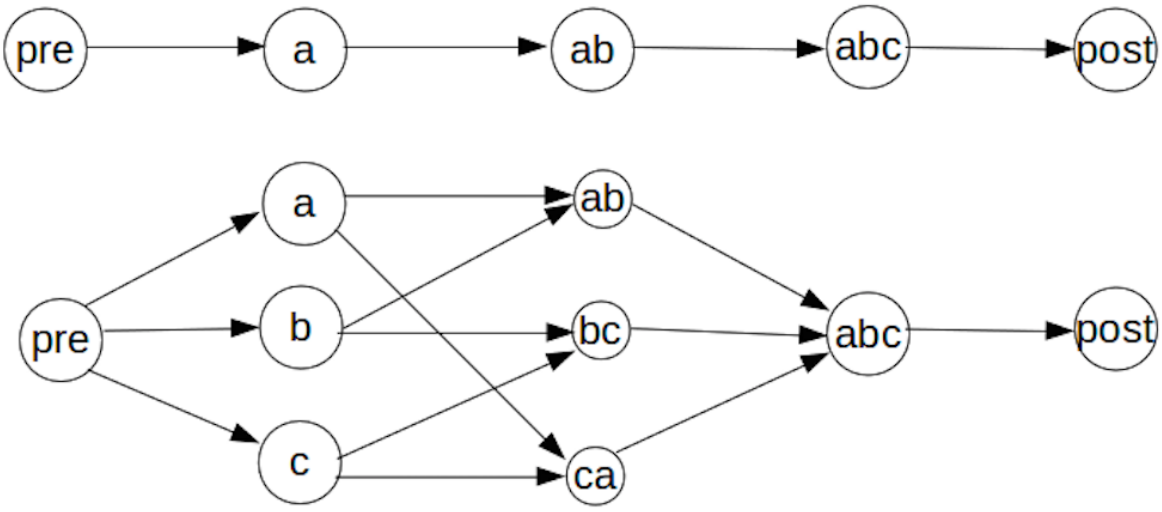

Figure 2: up: all serial tasks, Down: all parallel tasks; a,b,c represents three different tasks and each link represents one dependency

this problem into a maximization problem with linear objectives with only an ϵ loss in approximation, shouldn't just about any convex combination of penalties in the objective function be solvable using convex optimization techniques? This captures the minmax problem too, I think... -YN Min max is captured by LP already, so yeah this should capture it -KZ

In terms of optimization formulation, it is very similar to (2.2) *edge-based* solution in section 2.1, however linear constraints (2.2d) and (2.2e) should be removed since they are already captured in the objective function.

2.2 Multiple tasks as a DAG In the above formulation we assume either there is only one task, or each vertex can run all types of flow processing tasks where we can bundle all tasks into one task. Nevertheless if we have specialized hardware for different types of tasks or two types of tasks are preferred at different vertices, we need to revisit our formulation. We consider two common types of task relationships: serial and parallel. Serial tasks must be handled in a certain order while parallel tasks can happen in any order. A set of tasks may require a policy of a mixture of serial and parallel, e.g., a flow requires three different types of processing a, b and c, and a is before b and c while b and c can happen in either order.

We can extract any arbitrary task topology as a partial ordered set (poset). We show two simplest cases where all N tasks are serial or parallel, then we generalize the solution to any arbitrary order via chain-antichain sets.

For N tasks, we have $C^n(v); n \in \{1 \dots N\}$ for N different processing capacities. In optimization formulation 2.2 f and w can be interpreted as two different flow states, pre-processed with volume w and post-processed $f - w$. In *serial model* we have $N + 1$ flow states: pre-n-post-$(n-1)$ processed where $n \in \{1 \dots N+1\}$. We can extend formulation 2.2 by adding $N-1$ process demands, in particular for 2.2c we extend to N different types of processing, $p_i^n = \sum_{in} w_i^n(e) - \sum_{out} w_i^n(e)$. Besides the same capacity and flow relation constraints, we also need $w_i^n(e) \geq w_i^{n-1}(e)$ where it reflects the order. The complexity of this formulation increases in a linear relation to the number of tasks N. In *paralle model*

we have 2^N processing states and $O(2^N)$ numbers of inequalities for transitioning flow states such as from preprocessed states for cerain tasks to postprocessed states; and $O(N)$ number of inequalities for the vertex capacity. In general; for a topology with a maximum number of A antichains and L is the maximum length of a chain, then we need maximum $A \lg L$ bits to presents the states, so we have $2^{A \lg L} = L * 2^A$ number of states, in other words, the number of flow states is at worst exponential to the width of the poset, and linear to the height of poset.

3 Design Problem

References

[1] Awerbuch, B., and Azar, Y. Buy-at-bulk network design. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science* (1997), pp. 542–547.

[2] Charikar, M., and Karagiozova, A. On non-uniform multicommodity buy-at-bulk network design. In *Proc. of ACM STOC* (2005), ACM Press, pp. 176–182.

[3] Chekuri, C., Hajiaghayi, M. T., Kortsarz, G., and Salavatipour, M. R. Approximation algorithms for node-weighted buy-at-bulk network design. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2007), SODA '07, Society for Industrial and Applied Mathematics, pp. 1265–1274.

[4] Ford, L. R., and Fulkerson, D. R. A suggested computation for maximal multi-commodity network flows. *Management Science 5*, 1 (1958), 97–101.

[5] Mogul, J. C., Douglis, F., Feldmann, A., and Krishnamurthy, B. Potential benefits of delta encoding and data compression for http. *SIGCOMM Comput. Commun. Rev. 27*, 4 (Oct. 1997), 181–194.

[6] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (2013), SIGCOMM '13, ACM, pp. 27–38.

[7] Wayne, K. D. A polynomial combinatorial algorithm for generalized minimum cost flow. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing* (1999), STOC '99, ACM, pp. 11–18.

A Proof for equivalence between two LPs section2.1

Proof sketch: we first show that we can compose an *edge-based* solution based on a *path-based* solution and vice versa for a single flow, and then show that we can iteratively place multi-commodity flows.

1. show *Direction A:* If there is a *path-based* LP solution, there is an *edge-based* solution.
2. show *Direction B:* If there is an *edge-based* LP solution, there is a *path-based* solution using path decomposition.
3. show the formulations for multi-commodity flows are also equivalent via extending the above approach.

A.1 *path-based* solution → *edge-based* solution

Proof. we show that we can easily convert path-based solution to edge-based solution and all the constraints in edge-based formulation hold.

For each edge e, $f(e) = \sum_{\pi \in P: e \in \pi} f(\pi)$.

For each vertex v, $w(e) = \sum_{\pi \in P: e' \in \pi, e' \leq e} p(\pi, e')$ ($e' \leq e$ means e' is topologically at or after e on the path π).(A.2c)

Flow conservation holds $\sum_{(u,v) \in E} f(e) = \sum_{\pi \in P, v \in \pi} f(\pi) = \sum_{(v,w) \in E} f(e)$.(A.2b)

Constraints in terms of $B(e), C(v)$ also hold. (A2d,2e)

Relations between $w(e), f(e)$ also hold: $w(e) = \sum_{\pi \in P: e' \in \pi, e' \leq e} p(\pi, e') \leq \sum_{\pi \in P: e \in \pi} f(\pi) = f(e)$, and $w(s,v) = f(s,v)$ and $w(v,t) = 0$ are special cases.(A.2f,2g,2h)

A.2 *edge-based* solution → *path-based* solution To prove this; we need to show:

1. We can always construct a path if there is some residual flow left in the graph.
2. All constraints holds for the updated residual graph.

Setup: A directed graph $G(V, E)$ with an *edge-based* LP solution, where $f(e)$ is the flow for each edge, $w(e)$ is workload demand at the same edge and $p(v)$ process work done at each vertex v. Build a new graph G': all vertices V, and for $\forall e \in E$, if $f(e) > 0$, we put a direct edge e in the graph. To help proof, divide a flow into two states, processed and unprocessed f^1 and f^2; in terms of flow volume $f^1 = w$ and $f^2 = f - w$.

LEMMA A.1. *loops for flows f^1 and f^2 respectively can be cancelled via flow cancellation without any side effect.*

Proof. it is similar to flow cancellation in a simple graph model:

(i) for e=(u,v) whereas $min(f^1) > 0$, we can simply cancel the unprocessed flow demand by small amount ϵ, and it does not affect the outcome of the flow outside the loop, while we can reduce the flow load and workload demand in the loop without side effect.

(ii) for e=(u,v) whereas $min(f^2) > 0$, we can cancel the processed flow demand by small amount ϵ, and this does not affect the outcome of the flow outside of the loop while we can reduce the flow load in the loop without side effect.

The intuition behind this is that loop exists due to that some flow needs to borrow some processing capacity from some node(s), so it would "detour" a flow in an unprocessed state and get back the flow in a processed state.

Introduce an intermediate variable ρ for each edge e where $\rho_e = \frac{w(e)}{f(e)} = \frac{f^1}{f^1+f^2}$. Run flow loop cancellation for f^1 and f^2 respectively in G'.

Note: after loop cancellation we may still have loops for f as a unitiy.

LEMMA A.2. *ρ has the following property: if there is a cycle for unity flow f, there is always at least one edge with $\rho = 1$ and one edge with $\rho = 0$.*

Proof. This can be easily inferred from Lemma A.1.

LEMMA A.3. (PATH CONSTRUCTION) *Algorithm 1 can always generate path with non-zero flow from source to sink if there exists any v where $p(v) > 0$.*

Proof. First, from Lemma A.2, the path cannot loop a cycle twice from [Path Construction]. Since downstream traversal keeps picking $\min \rho$ while upstreaming traversal keeps picking $\max \rho$, so we never pick the same edge twice. Since $p(v) > 0$ so at the same node there must be one upstream edge with $\rho > 0$ and downstream edge with $\rho < 1$. Since the same edge is never picked twice so there is no loop in terms of f^1 and f^2. The path consists of two DAGs, one is from source to v and one is from v to sink, the path is a DAG as well.

Second we need to show for a certain path π; $f(\pi) > 0$. Since $f(\pi) = \min\{f^1(e^a), f^2(e^b), p(v)\}$; at node v where $p(v) > 0$, so we have $f^1(e_{in}) > 0$ and $f^2(e_{out}) > 0$ at vertex v. Since we only pick $\max\{\rho\}$ for upstream traversal, so for $\forall e^a$; $f^1(e^a) > 0$. The same reason we have $\forall e^b$; $f^2(e^b) > 0$.

Data: $G'(V,E)$, $w(e)$, $f(e)$ for $\forall e \in E$ and $p(v)$ for $\forall v \in V$
Result: $f(\pi)$, $p(\pi, v)$ where $v \in \pi$
Algorithm `Path Construction()`

//Construct path from $s \to v$ and $v \to t$
From v run backward traversal, pick an incoming directed edge with $max(\rho_{in})$ where $\rho_{in} \equiv \frac{w(e_{in})}{f(e_{in})}$
From v run forward traversal, pick an outgoing directed edge with $min(\rho_{out})$ where $\rho_{out} \equiv \frac{w(e_{out})}{f(e_{out})}$
return π

Algorithm `Flow Placement()`

while $\exists v; p(v) > 0$ **do**
//path representation $\pi \equiv < v_1, \ldots, v_k > \equiv < e_1, \ldots, e_{k-1} >$
$\pi =$ `Path Construction()`
$f(\pi) = \min\{f^1(e^a), f^2(e^b), p(v)\},\ e^a \in < e_1, \ldots, u \to v >, e^b \in < v \to w, \ldots, e_{k-1} >$
$p(\pi, v) = f(\pi)$
for $u \in \pi$ *and* $u \neq v$ **do**
$p(\pi, u) = 0$
end
$C(v) = C(v) - f(\pi)$
$p(v) = p(v) - f(\pi)$
for $i \leftarrow 1$ **to** $k-1$ **do**
$f(e_i) = f(e_i) - f(\pi)$
$B(e_i) = B(e_i) - f(\pi)$
end
end

Algorithm 1: Path Decomposition

LEMMA A.4. (FLOW PLACEMENT) *Algorithm 1 conserves all the constraints for the reduced graph.*

Proof. we show that all the constraints are satisfied:
for A.2b: $\forall v \in \pi; \sum_{in} f(e) - \sum_{out} f(e) = \sum_{in \neq e_i} f(e) - \sum_{out \neq e_{i+1}} f(e) + [f(e_i) - f(\pi)] - [f(e_{i+1}) - f(\pi)] = 0$
A.2d: $\forall e \in \pi; f(e) = f(e) - f(\pi) \leq B(e) - f(\pi) = B^{new}(e)$
A.2e: $\forall v \in \pi; p(v) - f(\pi) \leq C(v) - f(\pi) = C^{new}(v)$
A.2f and A.2g are ensured by the algorithm, since $v \neq s$ and $v \neq t$.
A.2h constraints are satisfied by numerical relations.

A.3 Multi-Commodity Flow For MCF, we can use the same approach above. For a graph with K source-sink paired flows, we iterate $i = 1 \ldots K$, for each flow we genrate a G' and exhaustively decompose paths for f_i and it is easy to see that all the constraints still hold after flow i has been removed. In particular, we have : A.2b, A.2c, A.2f and A.2g hold for all the flows left after one flow is removed; A.2d: $\forall i, \forall e; \sum_{l=i}^{K} f_l(e) - f_i \leq B(e) - f_i(e)$; A.2e: $\forall i, \forall v; \sum_{l=i}^{K} p_l(v) - p_i(v) \leq C(v) - p_i(v)$.