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

Abstract

We introduce and study a new class of multi-commodity flow problems where, in addition to demands on flows and capacity constraints on edges in the network, there is an additional requirement that flows be processed by nodes in the network. These problems are motivated by the placement and configuration of so-called middleboxes at nodes in the network so as to perform services on the network traffic: how many middleboxes to run, where to place them and how to direct traffic through them?

We study the problems that arise from jointly optimizing the: (1) placement of middleboxes over a pool of server resources, (2) steering of traffic through a suitable sequence of middleboxes, and (3) routing of the traffic between the servers over efficient network paths. We introduce and study several problems in this class from the exact and approximation point of view.

1 Introduction

1.1 Background 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].

1.2 The General Problem 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.

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.

Our model is a superset of standard multi-commodity flow model[5], that is, if we can solve this problem, we can naturally solve multi-commodity flow problem: simply assigning each vertex with an infinite capacity it becomes an MCF problem. However our problem is also very different from standard multicommodity flow varieties, for example in our model loops can happen, and actually quite often in real settings as flows are "detoured" to be processed.

1.3 Outline of this paper We separate the core of this paper into two sections. In Section 2, we discuss the *processed packet routing* class of problems, in which we discuss how to feasibly route packets in a fixed network while optimizing various objective functions. We also use this section to discuss how to handle the case where the processing alters the size of the flow, affecting how much capacity is required of edges following the processing vertex, and when the act of processing requires multiple steps to be handled by various nodes. In Section 3, we discuss the *processing power allocation* problem, in which the goal is to purchase processing capacity to place at certain nodes so maximize or minimize something.

2 Packet Routing

2.1 The basic problem We begin by introducing the routing problem in the presence of processing demands. 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. The goal is then either to route as much flow as possible, or

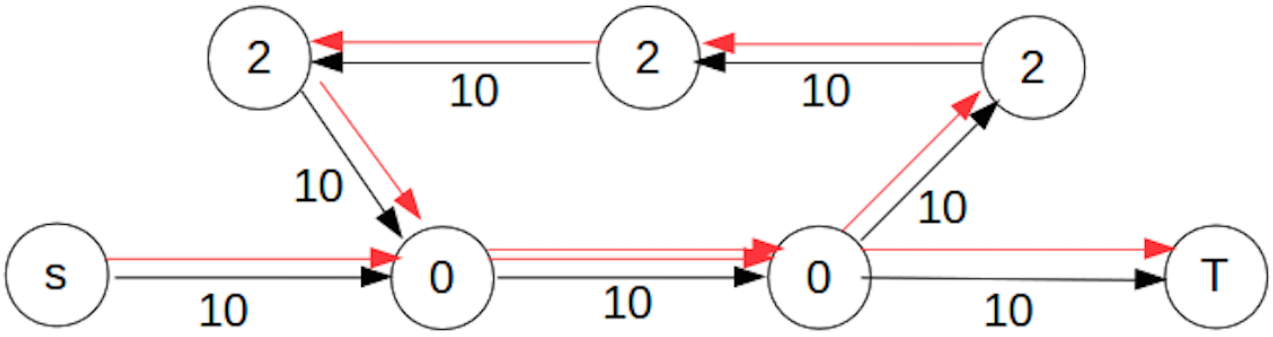

Figure 1: Example above we assume all edges have a capacity of 10, and node processing capacity are 2 and 0 for vertices, we can send maximum flow size 5, and we route flow to a loop for processing and then send to sink, the flow goes through the bottom middle edge twice, and 5 units of processing required can be assigned to the top three nodes where the sum of processing capacity is 6

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.

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 constraints remains an interesting open problem.* *Mention this in the conclusions? - YN Sure this sounds like a hard problem -KZ*

2.1.1 Flow Maximization We begin by showing how to express the maximization version of the problem both as an *edge-based* and as a *walk-based* linear program. While neither of these constructions are particularly difficult, it's not immediate that either is enough to solve the flow problem in polynomial time. In particular, while the walk-based LP requires exponential size, the polynomial-sized edge-based LP may a-priori not correspond to valid routing pattern at all. In subsection A.1, we resolve this problem by showing that the two linear programs are equivalent, and so the edge-based LP inherits the correctness of the walk-based program, ensuring that we can indeed find a valid solution in polynomial time. 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 ??? time.* *This might be worthwhile to state? -YN I think the run time should be $O(Vf)$, however f is the number of rounds, it should be bounded by $\sum_{\pi} f_\pi / \min\{f\}$ -KZ 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?*

To express the walk-based linear program, we require one variable $p^v_{i,\pi}$ for each walk-vertex-demand triplet, representing the total amount of flow from s_i, t_i exactly utilizing walk π and processed at v. The aggregate (s_i, t_i) flow sent along a given walk π is then simply denoted by $p_{i,\pi}$, and the set of all walks is given by P. The linear program is then the standard multicommodity-flow LP augmented with the new processing capacity constraints.

The edge-based formulation can be thought of as sending two flows for each D_i: f_i represents the packets being sent from s_i to t_i and w_i is the processing demand of these packets. While f_i is absorbed (non-conserved) only at the terminals, w_i is absorbed only at the processing vertices. The variables $f_i(e)$ and $w_i(e)$ measure how much of f_i and w_i passes through edge e. We use the notation $\delta^+(v)$ and $\delta^-(v)$ to denote the edges leaving and entering vertex v, respectively. The two linear programs are given below:

Walk-based formulation:

$$\text{MAXIMIZE} \qquad \sum_{i=1}^{|D|} \sum_{\pi \in P} p_{i,\pi}$$

SUBJECT TO

$$p_{i,\pi} = \sum_{v \in \pi} p^v_{i,\pi} \qquad \forall i \in [|D|], \forall \pi \in P$$

$$\sum_{i=1}^{|D|} \sum_{\substack{\pi \in P \\ \pi \ni e}} p_{i,\pi} \leq B(e) \qquad \forall e \in E$$

$$\sum_{i=1}^{|D|} \sum_{\pi \in P} p^v_{i,\pi} \leq C(v) \qquad \forall v \in V$$

$$p^v_{i,\pi} \geq 0 \qquad \forall i \in [|D|], \forall \pi \in P, \forall v \in V$$

Edge-based formulation:

$$\text{MAXIMIZE} \qquad \sum_{i=1}^{|D|} \sum_{e \in \delta^+(s_i)} f_i(e)$$

SUBJECT TO

$$\sum_{e \in \delta^-(v)} f_i(e) = \sum_{e \in \delta^+(v)} f_i(e) \qquad \forall i \in [|D|], \forall v \in V \setminus \{s_i, t_i\}$$

$$p_i(v) = \sum_{e \in \delta^-(v)} w_i(e) - \sum_{e \in \delta^+(v)} w_i(e) \qquad \forall i \in [|D|], \forall v \in V$$

$$\sum_{i=1}^{[D]} f_i(e) \leq B(e) \qquad \forall e \in E$$

$$\sum_{i=1}^{|D|} p_i(v) \leq C(v) \qquad \forall v \in V$$

$$w_i(e) \leq f_i(e) \qquad \forall i \in [D], \forall e \in E$$

$$w_i(e) = f_i(e) \qquad \forall i \in [D], \forall e \in \delta^+(s_i)$$

$$w_i(e) = 0 \qquad \forall i \in [D], \forall e \in \delta^-(t_i)$$

$$w_i(e), p_i(v) \geq 0 \qquad \forall i \in [D], \forall e \in E$$

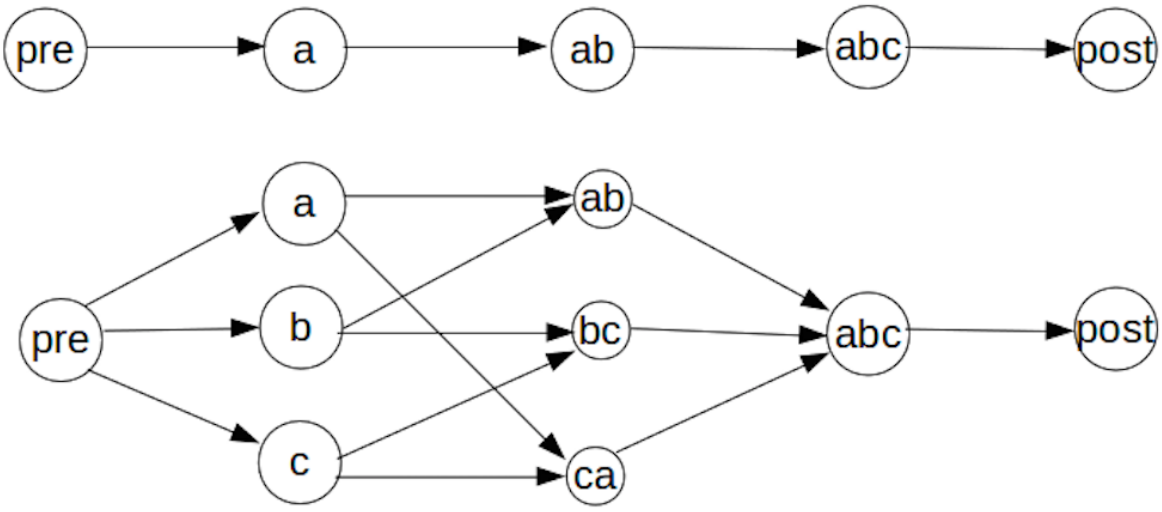

Figure 2: up: all serial tasks, Down: all parallel tasks; a,b,c represents three different tasks and each link represents one dependency

Flow Maximization with Size Changes In some cases, middlebox processing might significantly alter the volume of data in a given microflow. For example, encrypting middleboxes might increase the size of the flow, while compression and transcoding may substantially decrease it [2]. If processing scales the size of the data of flow i by a constant multiplicative factor $r_i \in \mathbb{R}^+$, such effects can be captured by linear programs. To do so, we need to separate the flow unity into two types, preprocessed and postprocessed, which can be represented as w and $f - w$. The increase of postprocessed flow equals the decrease of preprocessed flow. So the flow conservation constraint of the original edge-based LP should be replaced by $p_i(v) \cdot r_i = \sum_{e \in \delta^+(v)} (f_i(e) - w_i(e)) - \sum_{e \in \delta^-(v)} (f_i(e) - w_i(e))$.

2.1.2 Cost Minimization 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, concave 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 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*

As a warmup, let's first consider the special case of linear $c_v(\cdot)$ and $c_e(\cdot)$. A linear program solving this problem can be constructed akin to the above edge-based formulation.

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 upper bound of the number of flow states is exponential to the width of the poset, and linear to the height of poset.

3 Network Design

In this section, we discuss the problem of how to optimally purchase processing capacity so to satisfy a given flow demand. Although this can be modeled in multiple ways, we limit our discussion to the case where each vertex v has a potential processing capacity $\hat{C}$, which can only be utilized if $\hat{C}$ is purchased As in the previous section, this yields two general categories of optimization problems

1. The *minimization* version of the problem, where the goal is to pick the smallest set of vertices such that all flow is routable.

2. The *maximization* version of the problem, where we try to maximize the amount of routable flow while subject to a budget constraint of k. *Throw this into the future directions section. -YN*

We begin with the simple observation that the minimization and maximization versions of this problem inherit the hardness of Set Cover and Max Coverage, respectively.

Theorem 3.1. *It is $NP-$ hard to approximate the minimization problem to within a factor better than $.2267 \cdot \log n$. Further, there is no $(1 - o(1))$-approximation algorithm for the minimization problem unless* **NP** $\subseteq$ **DTIME**$(n^{O(\log \log n)})$. *Citation: https://www.cs.duke.edu/courses/spring07/cps296.2/papers/p634-feige.pdf , http://www.tau.ac.il/ nogaa/PDFS/GPGames.pdf*

Theorem 3.2. *It is $NP-$hard to approximate the maximization problem to within a factor better than $1 - 1/e$. Citation: https://www.cs.duke.edu/courses/spring07/cps296.2/papers/p634-feige.pdf*

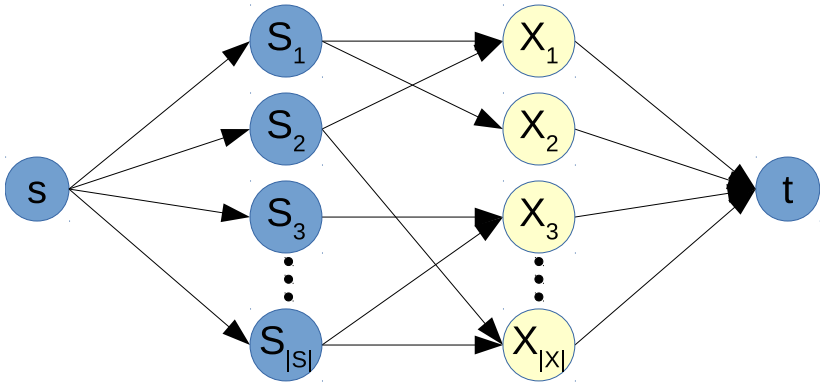

Figure 3: Approximation-preserving reduction from Set Cover and Max Coverage to **the minimization problem** and **the maximization problem** problem. All edges have infinite capacity, blue vertices have 0 potential capcity, yellow vertices have $|X|$ potential capacity.

4 Conclusion

hello

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] Chi, C., Deng, J., and Lim, Y. Compression Proxy Server: Design and Implementation. *USENIX Symposium on Internet Technologies and Systems. 1999.*

[3] 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.

[4] 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.

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

[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 Apppendix

A.1 Proof for equivalence between two LPs section2.1 Proof sketch: we first show that we can compose an *edge-based* solution based on a *walk-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 *walk-based* LP solution, there is an *edge-based* solution.
2. show *Direction B:* If there is an *edge-based* LP solution, there is a *walk-based* solution using walk decomposition.
3. show the formulations for multi-commodity flows are also equivalent via extending the above approach.

A.1.1 *walk-based* solution → *edge-based* solution

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

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

For each vertex v, $w_i(e) = \sum_{v' \in \pi, v' \leq v} p^v_{i,\pi}$ ($v' \leq v$ means e' is topologically at or after e on the walk π).

Flow conservation holds $\sum_{(u,v) \in E} f_i(e) = \sum_{\pi \in P, v \in \pi} p_{i,\pi} = \sum_{(v,w) \in E} f_i(e)$.

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

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

A.1.2 *edge-based* solution → *walk-based* solution

to prove this; we need to show:

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

Setup: For simplicity, we only construct all walks for a flow each time, so notation wise we can remove i. 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.

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 `Walk Construction()`

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

Algorithm 1: Walk Decomposition

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

Proof. First, from Lemma A.2, the walk cannot loop a cycle twice from [Walk 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 walk consists of two DAGs, one is from source to v and one is from v to sink, the walk is a DAG as well.

Second we need to show for a certain walk $\pi; p_\pi > 0$. Since $p_\pi = \min\{f^1(e^a), f^2(e^b), p(v)\}$; 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 $\forall e^b; f^2(e^b) > 0$.

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: $\forall v \in \pi; \sum_{in} f(e) - \sum_{out} f(e) = \sum_{in \neq e_i} f(e) - \sum_{out \neq e_{i+1}} f(e) + [f(e_i) - p_\pi] - [f(e_{i+1})p_\pi] = 0$
A.2d: $\forall e \in \pi; f(e) = f(e) - p_\pi \leq B(e) - p_\pi = B^{new}(e)$

A.2e: $\forall v \in \pi; p(v) - p_\pi \leq C(v) - p_\pi = C^{new}(v)$
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.1.3 Multi-Commodity Flow For MCF, we can use the same approach above. For a graph with K source-sink paired flows, we iterate $i = 1 \dots K$, for each flow we genrate a G' and exhaustively decompose walks 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: $\forall i, \forall v; \sum_{l=i}^{K} p_l(v) - p_i(v) \leq C(v) - p_i(v)$.