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 [7].

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 above problem–flow with in-network processing 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 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 of many connections 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 node computational capacity. A feasible flow pattern satisfies: (1) the sum of flows on each edge is bounded by the edge capacity, (2) the sum of in-network processing done at each vertex is bounded by the vertex capacity, and (3) the processing done at all vertices for a flow is equal to the flow size.

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 variants. For example, a flow might be required to pass through the same edge or vertex multiple times before reaching its destination, a phenomenon that occurs often in practice when flows get "detoured" for processing.[7]

1.3 Outline of this paper In Section 2, we introduce 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. Our main result here is that given a network with edge capacities, vertex processing capacities and flow demands, we give an LP based algorithm to find a multi-commodity flow with processing assigned to vertices so as to optimize several natural objective functions: maximum flow, sum of congestions, etc. As various in-network processing may alter flow size, such as during transcoding, compression, or encryption, we show how our solution can be adapted to handle dynamically changing flow sizes. We also discuss the case when multiple processing steps are required before a packet reaches its destination, as may arise in onion routing or while monitoring processed traffic. In Section 3, we discuss the PROCESSING POWER ALLOCATION class of problems, in which the goal is to purchase processing capacity in a network (the capacities of edges are fixed) so to maximize the total amount of traffic it can carry. We show an $O(\log(n)/\delta^2)$ approximation for processing power cost and an associated multi-commodity flow that satisfies $(1-\delta)$ fraction of the demands and satisfies all edge capacities. We show that the problem is hard to approximate better than a logarithmic factor, even if the demand requirements are relaxed. We also show that minimizing the processing power cost for a version of the problem with indifference routing is LABEL COVER hard.

2 Packet Routing with In-network Processing

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 $C : V \to [0, \infty)$, 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 to satisfy all flow demand subject to appropriate congestion-minimization objective function. 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.

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 is particularly difficult, it is not obvious 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.2, 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.

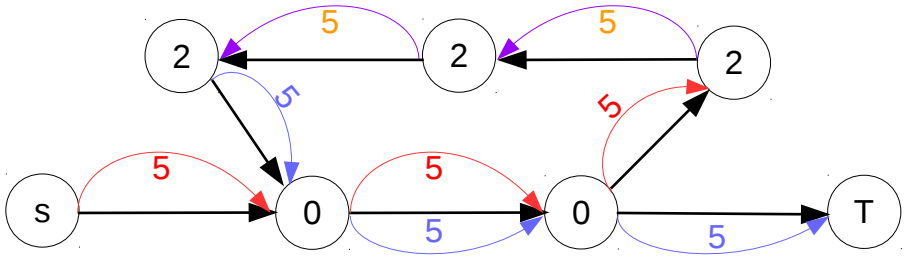

Figure 1: The edge capacity is 10 for all edges and the node capacities are denoted in each node. Here, we can send maximum flow size 5, by routing it along the red arcs, have it processed at the nodes at the top, and then sent to T along the blue arcs. The capacity of the bottom middle edge forms the bottleneck here, as all flow must pass through it twice before reaching T.

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 by decomposing it into its composing s_i, t_i walks in $O(|V| \cdot |E| \cdot |D|)$ time.*

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:

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

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:

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

SUBJECT TO

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

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

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

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

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

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

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

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

Flow Maximization with Size Changes In some cases, middlebox processing might significantly alter the volume of data in a given connection. For example, encrypting middleboxes might increase

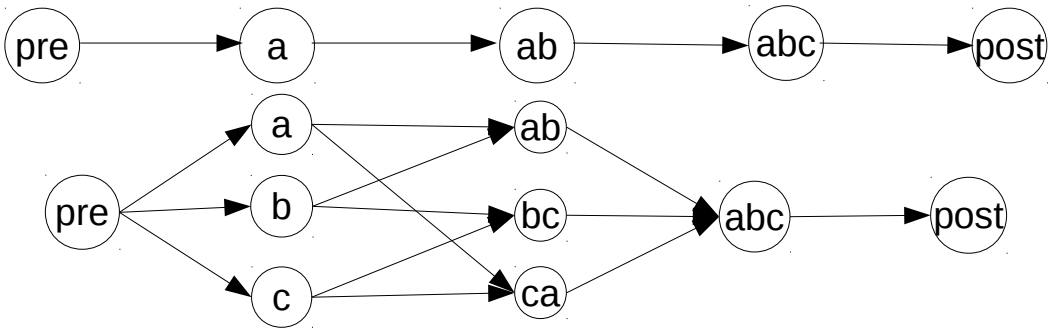

Figure 2: Up: Tasks may have strong dependencies and require sequential processing. Down: Tasks may be processible in any order.

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 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, convex congestion measures $c_v : [0, \infty) \to [0, \infty)$ and $c_e : [0, \infty) \to [0, \infty)$. In this model, a vertex v with processing capacity $C(v)$ is assigned a congestion $c_v = c_v(\frac{\sum_{f \in F} f_v}{C(v)})$ and each edge is assigned congestion $c_e = c_e(\frac{\sum_{f \in F} f_e}{B(e)})$, where f_v and f_e are the amount of processing and amount of flow that flow path f assigns to vertices v and e, respectively. Each *flow*, 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 flow encounter, or, equivalently, to minimize $\sum_{e \in E} c_e f_e + \sum_{v \in V} c_v f_v$.

As a warmup, let's first consider the special case of constant $c_v(\cdot)$ and $c_e(\cdot)$. With these simple functions in place, we can write our objective function as MINIMIZE $\sum_v c_v(\frac{\sum_{f \in F} f_v}{C(v)}) \sum_{f \in F} f_v + \sum_e c_e(\frac{\sum_{f \in F} f_e}{B(e)}) \sum_{f \in F} f_e = \sum_v \hat{c}_v \sum_{f \in F} f_v + \sum_e \hat{c}_e \sum_{f \in F} f_e$ for some constants $\hat{c}_e$ and $\hat{c}_v$. Adding in the equality $\sum_{e \in \delta^+(S_i)} f_e = D_i$ for each S_i and copying the rest of the edge-based linear program from Section 2.1.1 completes the linear program.

The more interesting case is monotone convex functions with bounded second derivative. Consider an arbitrary edge $e = (u, v)$ with monotone, convex cost function $c_e(\cdot)$ whose second derivative is bounded by some K. Dividing e into q edges $e_1, e_2, \cdots, e_q$ from u to v with capacity $B(e)/q$ and constant cost functions $c_{e_i}(\cdot) = c_e\left(\frac{i-.5}{qB(e)}\right)$, we get an edge set which would collectively lower bound the cost incurred by the congestion at e. Standard bounds on the errors of Riemman sums show that this error is bounded by $O\left(\frac{K}{q^2}\right)$. Thus, selecting $q = \frac{\sqrt{K}}{|E|\epsilon}$, the sum of all error terms is bounded by $O(\epsilon)$, giving an additive $O(\epsilon)$ approximation. Thus, complicated functions can be simulated by a collection of constant-congestion edges. A similar technique involving replacing vertices with independent sets lets us replace their cost functions with a collection of constant functions. Thus, if the maximum second derivative $\hat{K}$ over all c_e and c_v is $O(\text{poly}(n))$, this technique gives an additive FPTAS for computing the minimum-cost routing solution.

2.2 Multiple Types of in-Network Processing as a DAG Sometimes packets need to be processed in multiple, distinct stages. For example, onion routing requires the data to visit a number of intermediaries, each with its own decryption key, before reaching its ultimate destination. Further, it might be the case that certain nodes are fit for only certain types of computations, some of which may contain interdependencies, such as decrypting of files after they pass through a firewall, or encypting a file after it's compressed. Thus, it is natural to attempt to generalize the above formulation into one that can handle multiple processing nodes.

One way to model this type of problem is via DAGs. For each s_i, t_i pair, we require a DAG G_i on vertices $T \subseteq V$. We then require the s_i, t_i flow visit and be processed by nodes in G_i such that processing of f_i at a vertex v succeeding u in G_i is only done after u completed its processing of f_i. While in the dependency routing version of the problem we require *all* vertices in the DAG eventually process f_i, the indifference routing version simply requires f_i to fully traverse and receive processing at each of the vertices on one maximal path in G_i. Note that indifference routing fully captures the original routing problem from Section 2.1 when all intermediary nodes of the DAG form an antichain.

Interestingly, we can encode both indifference routing and dependency routing into the above edge-based linear program, though the latter may require an exponential number of new constraints. To do so, each vertex v needs to be given T different processing capacities $C_1(v) \cdots C_T(v)$, one for each task. Indifference routing can then be implemented by replacing each $w_i(e)$ with a collection of $w_i^t(e)$ measuring how much processing of task t the (s_i, t_i) flow along e yet demands and adding in simple inequalities ensuring that the flow processed by v is no more than the sum of all flows processed on v's immediate predecessors in G_i. The straightforward approach to ensuring feasibility in the dependency routing case, however, requires flows to fully identify their processing history to ensure that multiple fractionally-processed flows don't get merged and counted as a fully processed flow by the LP. We then ensure that any processing done by vertex u only be done on flow that identifies itself as having been processed at each of u's immediate predecessors. Although the naive encoding requires up to 2^T new flows be created for each (s_i, t_i) pair, it is sufficient to decompose T into a chains and store the progress of the processing along each chain using at most $\log T$ bits per chain. A simple application of Dilworth's Theorem[3] allows us to bound the size of this encoding by $2^{A(G_i)} \log |T| = |T| 2^{A(G_i)}$, where $A(G_i)$ is the size of the largest antichain in G_i. This gives a significant advantage when the interdependency poset is close to a chain.

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 (Min Middlebox Purchase), where the goal is to pick the smallest set of vertices such that all flow is routable.

2. The *maximization* version of the problem (Max Middlebox Purchase), where we try to maximize the amount of routable flow while subject to a budget constraint of k.

Formally, the input to Min Middlebox Purchase is a graph $G = (V, E)$ with nonnegative costs q_v on its vertices, a potential processing capacity $C : V \to [0, \infty)$, and a collection of (s_i, t_i) pairs with demands R_i. The goal is to select a set $T \subseteq V$ of vertices such that all demands are satisfied. Max Middlebox Purchase is given the same collection of inputs along with a budget integer k, and the goal is to route as much of the demand as possible.

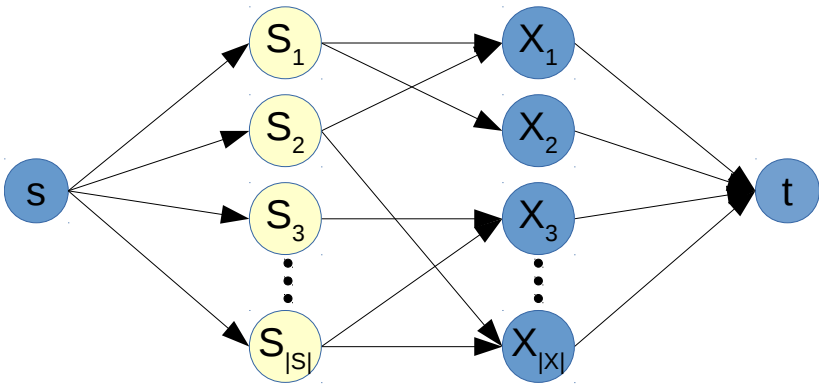

Figure 3: Approximation-preserving reduction from Set Cover and Max k-Coverage to Min Middlebox Purchase and Max Middlebox Purchase. All edges have infinite capacity, blue vertices have 0 potential capcity, yellow vertices have $|X|$ potential capacity. All purchase costs are 1.

We begin with the simple observation that Min Middlebox Purchase and Max Middlebox Purchase inherit the hardness of Set Cover and Max k-Coverage, respectively.

3.1 Hardness of Network Design

Theorem 3.1. *It is* **NP**-hard *to approximate* Min Middlebox Purchase *to within a factor better than* $.2267 \log n$. *Further, there is no* $(1-\epsilon)\log n$*-approximation algorithm for* Min Middlebox Purchase *unless* $\mathbf{NP} \subseteq \mathbf{DTIME}(n^{O(\log \log n)})$. *These results hold even when the algorithm is allowed to satisfy a mere* ϵ*-fraction of each demand yet is compared to an exactly-satisfying optimal solution.*

Theorem 3.2. *It is* **NP**-hard *to approximate* Max Middlebox Purchase *to within a factor better than* $1 - 1/e$.

The reduction for both Theorem 3.1 and Theorem 3.2 is provided in Figure 3. In both cases, we create one vertex S_i for each S in the set system $\mathcal{S}$, and one vertex X_j for each $\mathcal{X}$. Each vertex S_i is connected to vertex X_j for each $X_j \in S_i$. Each S_i is given $|\mathcal{X}|$ potential capacity, while all other vertices are given 0 capacity. Finally, all edges from X_i to t are given capacity 1, and all other edges are assigned unlimited capacity. Thus, the total amount of flow that can be sent from s to t is exactly the number of vertices X_j connected to s through a purchased vertex S_i. It is thus easy to verify that solutions to Max Middlebox Purchase and Min Middlebox Purchase on this graph exactly correspond to solutions to Max k-Coverage and Set Cover, respectively, which, together with known hardness bounds of those two problems[1][4], implies Theorem 3.1 and Theorem 3.2. Finally, by removing vertex T and setting one unit of flow demand from S to each of the X_i, we note that any solution satisfying each demand pair some ϵ amount is able to satisfy it at equality, as any fractional solution in the above constructions can be rounded up without any loss of feasibility, proving the last statement of Theorem 3.1.

It is tempting to state that the amount of routable flow is submodular in the collection of purchased vertices, which would imply that simple greedy algorithms would match the bounds of Theorem 3.1 and Theorem 3.2. As shown in Figure 4, this natural supposition happens to be false, as there exist configurations where the natural greedy algorithm gets stuck at an infeasible solution.

3.2 A bicriterion approximation algorithm We describe a modification of the walk based LP formulation with additional variables x_v corresponding to whether or not processing capacity at vertex

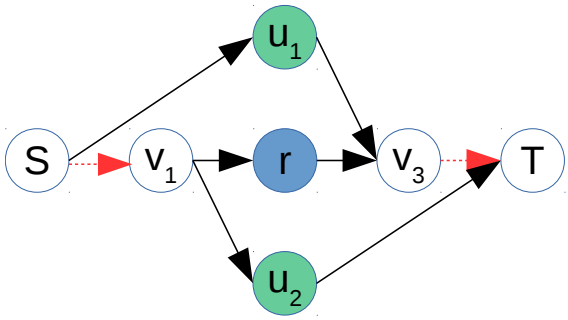

Figure 4: Example graph where vertex purchasing is not submodular. White vertices have no processing potential, colored vertices have 1 potential unit of processing. Solid black edges have capacity 2 while dashed red edges have capacity 1. If the only purchased vertex is r, no single additional purchase can increase the routable flow at all, yet buying both u_1 and u_2 simultaneously increases it to 2.

v has been purchased. We further give a polynomial sized edge-based LP formulation with flow variables $f_i^{1,v}(e)$ and $f_i^{2,v}(e)$ for each commodity i, each vertex $v \in V$ and each edge $e \in E$. The variables $f_i^{1,v}(e)$ correspond to the (processed) commodity i flow that has been processed by vertex v: these variables describe a flow from v to t_i. The variables $f_i^{2,v}(e)$ correspond to the (unprocessed) commodity i flow that will be processed by vertex v: these variables describe a flow from s_i to v.

Walk-based formulation:

$$\text{MINIMIZE} \quad \sum_{v \in V} q_v x_v$$

SUBJECT TO

$$x_v \leq 1 \qquad \forall v \in V$$

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

$$\sum_{\pi \in P} p_{i,\pi} \geq R_i \qquad \forall i \in [|D|]$$

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

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

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

$$\sum_{\pi \in P} p_{i,\pi}^v \leq R_i x_v \qquad \forall i \in [|D|], v \in V,$$

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

$$x_v \geq 0 \qquad \forall v \in V$$

Edge-based formulation:

$$\text{MINIMIZE} \quad \sum_{v \in V} q_v x_v$$

SUBJECT TO

$$x_v \leq 1 \qquad \forall v \in V$$

$$\sum_{e \in \delta^-(u)} f_i^{j,v}(e) = \sum_{e \in \delta^+(u)} f_i^{j,v}(e) \qquad \forall i \in [|D|], j \in \{1,2\}, v \in V, \forall u \in V \setminus \{s_i, t_i, v\}$$

$$\sum_{e \in \delta^-(v)} f_i^{2,v}(e) = \sum_{e \in \delta^+(v)} f_i^{1,v}(e) \qquad \forall i \in [|D|], v \in V,$$

$$\sum_{v \in V} \sum_{e \in \delta^+(s_i)} f_i^{2,v}(e) \geq R_i \qquad \forall i \in [|D|]$$

$$\sum_{i=1}^{|D|} \sum_{v \in V} (f_i^{1,v}(e) + f_i^{2,v}(e)) \leq B(e) \qquad \forall e \in E$$

$$\sum_{i=1}^{|D|} \sum_{e \in \delta^-(v)} f_i^{2,v}(e) \leq C(v) x_v \qquad \forall v \in V$$

$$\sum_{i=1}^{|D|} (f_i^{1,v}(e) + f_i^{2,v}(e)) \leq B(e) x_v \qquad \forall e \in E, v \in V$$

$$\sum_{e \in \delta^+(s_i)} f_i^{2,v}(e) \leq R_i x_v \qquad \forall i \in [|D|], v \in V$$

$$f_i^{2,v}(e) = 0 \qquad \forall i \in [|D|], v \in V, e \in \delta^-(s_i)$$

$$f_i^{1,v}(e) = 0 \qquad \forall i \in [|D|], v \in V, e \in \delta^+(t_i)$$

$$p_i^{1,v}(e), p_i^{2,v}(e), x_v \geq 0 \qquad \forall i \in [|D|], v \in V, e \in E$$

Given an optimal solution to this LP, we pick vertices to install processing capacity on by randomized rounding: pick vertex v with probability x_v. if x_v is picked, then all flows processed by v are rounded up in the following way: $\hat{F}_i^{j,v}(e) = f_i^{j,v}(e)/x_v$ for all $i \in [|D|], j \in \{1,2\}, e \in E$. If v is not picked, then all flows processed by v are set to zero, i.e. $\hat{F}_i^{j,v}(e) = 0$.

By design, $E[\hat{F}_i^{j,v}(e)] = f_i^{j,v}(e)$. In the solution produced by the rounding algorithm, the total flow through edge e is $\sum_{v \in V} \sum_{i=1}^{|D|} ((\hat{F}_i^{1,v}(e) + \hat{F}_i^{2,v}(e))$. This is a random variable whose expectation is at most $B(e)$, and is the sum of independent random variables, one for each vertex v. The constraints of the LP ensure that if v is selected, then the total processing done by vertex v is at most $C(v)$. Further, the total contribution of vertex v to the flow on edge e does not exceed the capacity $B(e)$, i.e. $\sum_{i=1}^{|D|} (\hat{F}_i^{1,v}(e) + \hat{F}_i^{2,v}(e)) \leq B(e)$. Also, the total contribution of vertex v to the commodity i flow is at most R_i, i.e. $\sum_{e \in \delta^+(s_i)} \hat{F}_i^{2,v}(e) \leq R_i$.

We repeat this randomized rounding process $t = O(\log(n)/\epsilon^2)$ times. Let $g^k(e)$ denote the total flow along edge e, and h_i^k denote the total amount of commodity i flow in the solution produced by the kth round of the randomized rounding process. The following lemma follows easily by Chernoff-Hoeffding bounds:

LEMMA 3.1.

$$Pr\left[\sum_{k=1}^{t} g^k(e) \geq (1+\epsilon)t \cdot B(e)\right] \leq e^{-t\epsilon^2/3} \qquad \forall e \in E \tag{3.5}$$

$$Pr\left[\sum_{k=1}^{t} h_i^k \leq (1-\epsilon)t \cdot R_i\right] \leq e^{-t\epsilon^2/2} \qquad \forall i \in [|D|] \tag{3.6}$$

We set $t = O(\log(n)/\epsilon^2)$ so that the above probabilities are at most $1/n^3$ for each edge $e \in E$ and each commodity i. With high probability, none of the associated events occurs. The final solution is constructed as follows: A vertex is purchased if it is selected in any of the t rounds of randomized rounding. Thus the expected cost of the solution is at most $t = O(\log(n)/\epsilon^2)$ times the LP optimum. We consider the superposition of all flows produced by the t solutions and scale down the sum by $t(1+\epsilon)$. This ensures that the capacity constraints are satisfied. Note that the vertex processing constraints are also satisfied by the scaled solution. The total amount of commodity i flow is at least $\frac{1-\epsilon}{1+\epsilon} R_i \geq (1-2\epsilon)R_i$. Hence we get the following result:

THEOREM 3.1. *There is a polynomial time randomized algorithm that satisfies all flow requirements upto factor $1-\delta$ and produces a solution that respects all capacities, with expected cost bounded by $O(\log(n)/\delta^2)$ times the optimal cost.*

3.3 Purchasing Processing Power for Indifferent Flows We can generalize the MIN MIDDLEBOX PURCHASE problem to incorporate indifference routing as described in Section 2.2. Unlike the results of the previous section, the following theorem shows that MIN INDIFFERENCE MIDDLEBOX PURCHASE problem is LABEL COVER-Hard, and thus is unlikely to admit any polylogarithmic approximation algorithm.

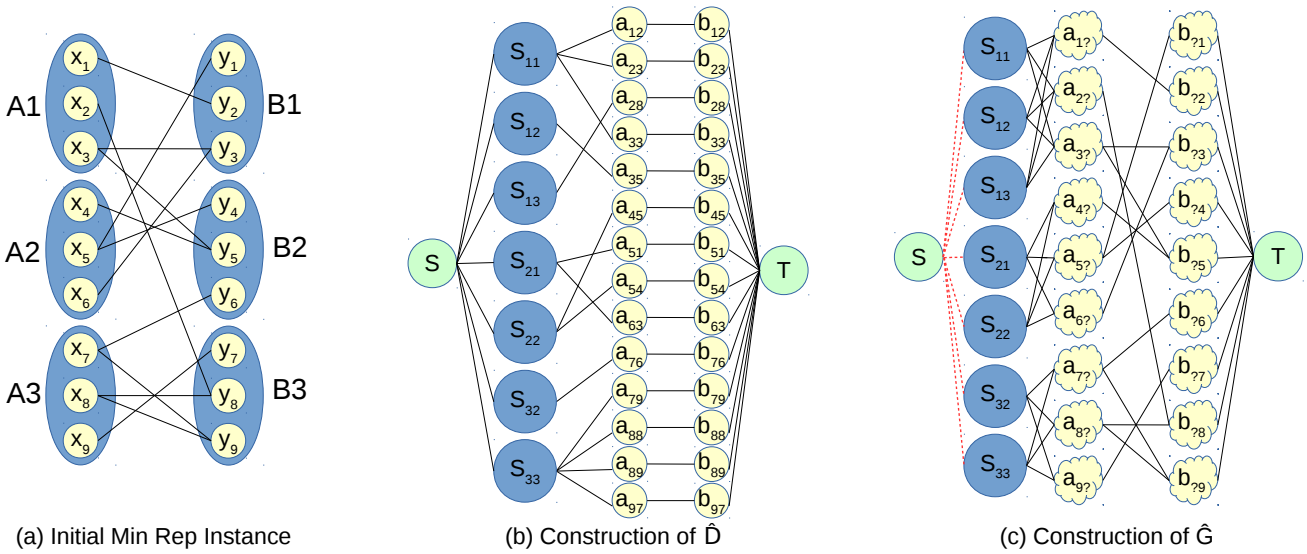

Figure 5: (a) A sample Min Rep instance from which we derive $\hat{D}$ and $\hat{G}$. (b) The construction of $\hat{D}$. Edges are directed left-to-right. (c) The construction of $\hat{G}$. Each cloud is actually a clique of all vertices in $\hat{D}$ whose name can be attained by replacing the question mark with a number. All drawn edges are directed left-to-right. Solid black edges have capacity ∞, while dotted red edges have capacity 1. All vertices are free except those in the clouds, which have a cost of 1 each.

Theorem 3.2. *For every $\epsilon > 0$, there is no polynomial-time algorithm approximating the single-commodity* Min Indifference Middlebox Purchase *problem to within an $O(2^{\log^{(1-\epsilon)} n})$ factor unless $NP \subseteq$ **DTIME**$(n^{\text{polylog}\, n})$.*

We achieve this hardness result by reducing from Min Rep, defined in [6]. Given a bipartite graph G with partitions A and B partitioned into subsets $A_1, A_2, \cdots$ and $B_1, B_2, \cdots$, respectively, we hope to construct a Min Middlebox Purchase instance with graph $\hat{G}$, indifference routing DAG $\hat{D}$, and vertex cost function $q : V \to \mathbb{R}^+$ whose feasible solutions can be mapped into feasible Min Rep solutions on G with the same cost.

We begin with the description of $\hat{D}$. For each hyperedge (A_i, B_j), we construct a source node S_{ij}. For each edge (a_p, b_q) in G, we add in two vertices a_{pq} and b_{pq} connected by an edge. We then connect S_{ij} to a_{pq} if both $a_p \in A_i$ and $b_q \in B_j$. Finally, we construct source and sink vertices S and T, with S connecting to each S_i and each b_{pq} connecting to T. Thus, traversing this DAG $\hat{D}$ from S to T can be thought of as first committing to a hyperedge, and then visiting its endpoints.

Our construction of $\hat{G}$ begins by replacing each $a_p \in G$ (resp. $b_q \in G$) with a clique of all vertices $a_{p\_}$ (resp. $a_{\_q}$), with all edges having unbounded capacity. For each edge (a_p, b_q) in G, we connect one arbitrarily-chosen vertex from clique $a_{p\_}$ to one arbitrarily-chosen vertex of clique $b_{\_q}$ via an infinite-capacity edge. Now we connect S_{ij} to some arbitrarily-chosen vertex from each clique $a_{p\_}$ for which $a_p \in A_i$. Finally, S is connected to S_{ij} with unit capacity edges, and one vertex from each $b_{\_q}$ clique is connected to T with infinite capacity edges. Every vertex has unbounded potential processing capacity, The costs of each vertex in the cliques $a_{p\_}$ and $b_{\_q}$ is 1, and the rest of the vertices are free.

We claim that there's a direct correspondence between valid solutions to the label-cover instance and solutions to the problem on $\hat{G}$ and $\hat{D}$ where R flow is anticipated from S to T. In particular, any solution to the Min Rep instance can be transformed into a solution to the constructed instance by purchasing one vertex from each clique corresponding to vertices chosen in the Min Rep solution. For each hyperedge, one unit of flow can be routed from S to the S_{ij} corresponding to the hyperedge, through some vertices of the cliques forming selected endpoints of the hyperedge, and finally to T (recall

that we may not begin and end in the correct vertices within the clique, but the infinite capacities allow us to traverse to the correct one without issue). Thus, any Min Rep solution may be transformed into a flow solution of equivalent cost. Conversely, any solution routing R units of flow must send one unit to each of the various S_{ij} vertices, to some $a_{p?}$ where a_p is one of the endpoints of the hyperedge corresponding to S_{ij}, over to some b_q, and to T. To create our solution to the label-cover instance, we can thus select exactly the a_i and b_j vertices corresponding to cliques containing selected $a_{p?}$ and $b_{?q}$ vertices. Since we will only select one vertex from each clique in an optimal solution, any flow solution may be transformed into a Min Rep solution of equal cost. Therefore, the optimal achievable values for the provided Min Rep instance and our constructed flow instance coincide, meaning that the flow problem shares any inapproximability of Min Rep.

References

[1] Alon, N., Moshkovitz, D., and Safra, S. Algorithmic construction of sets for k-restrictions. *ACM Transactions on Algorithms (TALG) 2*, 2 (2006), 153–177.

[2] Chi, C.-H., Deng, J., and Lim, Y.-H. Compression Proxy Server: Design and Implementation. In *USENIX Symposium on Internet Technologies and Systems* (1999).

[3] Dilworth, R. P. A decomposition theorem for partially ordered sets. *Annals of Mathematics* (1950), 161–166.

[4] Feige, U. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM) 45*, 4 (1998), 634–652.

[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] Kortsarz, G. On the hardness of approximating spanners. *Algorithmica 30*, 3 (2001), 432–450.

[7] 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.

A Appendix

A.1 Proof for equivalence between two LPs subsection 2.1 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: $w_i(e) = \sum_{v'\in\pi, v'\le v} p^v_{i,\pi} \le \sum_{v\in\pi} p^v_{i,\pi} = f_i(e)$, 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.

LEMMA A.3. (WALK CONSTRUCTION) *algorithm 1 can always generate a walk with non-zero flow from source to sink if there exists any v where $p(v) > 0$, and the algorithm can converge at $O(V^2)$*

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

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

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 $\forall e^b; f^2(e^b) > 0$.

For a single flow, after each iteration, we either take out one edge or one vertex, and the runtime for each iteration is $O(|V|)$ for traversal. As we iterate through $O(|V|)$ vertices and $O(|E|)$ for edges so the runtime total will be $O(|E| + |V|) \cdot |V|) = O(|V| \cdot |E|)$.

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: $\forall i, \forall v; \sum_{l=i}^{K} p_l(v) - p_i(v) \leq C(v) - p_i(v)$.

A.2 Proof for formulation about flow size change in subsection 2.1

Proof. The algorithm of constructing walks for edge-based LP w.r.t. flow size change is very similar to the approach above. However we need to "restore" the processed flow to "raw" flow size before constructing the walks. In particular, using the f^1, f^2 notation to represent pre and post processed flows, we restore f^2 to $\hat{f}^2$ with the multiplicative factor r, $\forall e, \hat{f}^2(e) = f^2(e)/r$. Everything is the same as previous algorithm after this step. Once we get f^1 and $\hat{f}^2$, we simply convert back to f^2 via $f^2 = \hat{f}^2 \cdot r$.