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].

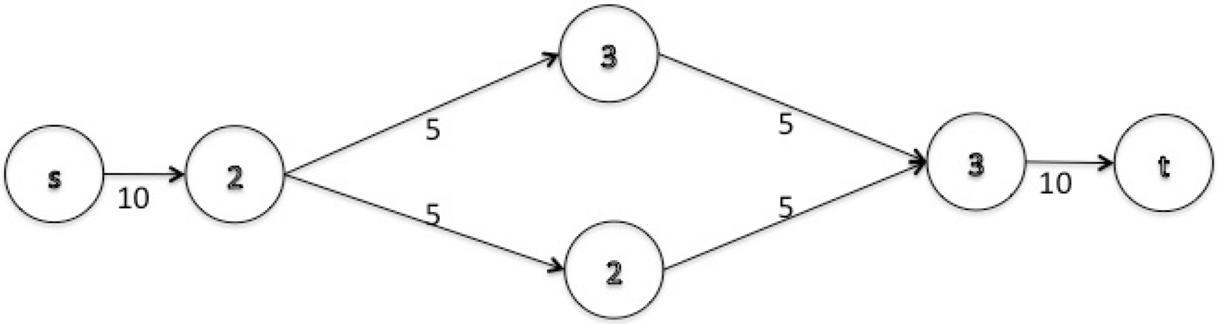

Figure 1: **Example for the Graph Model: given the edge and vertex constraints,** {5,10} **are edge capacities and {2,3} are vertex capacities, the maximum flow from source to sink in the example is 10, in this example all edges and vertices will be saturated.**

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 first solve the routing and steering problem for the above graph model: given all edge capacities $B(e)$ and vertex capacities $C(v)$ and a flow demand pattern $\mathbf{D}$, the optimization problem asks for a routing pattern with minimum congestion, the congestion can be interpreted in utilization. Like Max-flow model, we have capacitated edge constraints. However, our graph model has an extra constraint: vertex processing capacities, and it throttles the flow in a different manner: since a single flow is divisible and can be handled at different vertices, the flow is not constrained by a single node's processing power, but depending on all the nodes in the path(s). So a feasible flow pattern satisfies: (i) the edge capacities, (ii) the vertex capacities, and (iii) the relations between flow demand and processing demand. Without loss of generality, we assume the flow demand equals processing demand.

Path-based formulation is built directly on the definition of the problem, and it takes paths as abstractions. We can also formulate an edge-based linear programming model. We can prove that they are equivalent(Proof see appendix).

Notations: $f(U)$ is a convex function with a high penalty for high utilization. $B(e), C(v)$ are edge and vertex capacities, D_i is the demand for flow i. For path-based formulation, $p(\pi, e)$ represents the processing work done at v where $e \equiv (u, v)$, and P_i is a set of path for flow i with D_i. For edge-based solution, $p_i(v)$ is the processing work done at v for flow i, and $w_i(e)$ is the process work demand at edge e for flow i. Utilizations for edge and vertex are: $U(e) = \frac{\sum\limits_i \sum\limits_{\pi \in P_i : e \in \pi} f(\pi)}{B(e)} = \frac{\sum\limits_i f_i(e)}{B(e)}$ and $U(v) = \frac{\sum\limits_i \sum\limits_{\pi \in P_i} \sum\limits_{e \in \pi} p(\pi,e)}{C(v)} = \frac{\sum\limits_i p_i(v)}{C(v)}$.

Path-based formulation:

$$\text{Minimize:} \sum_e f(U(e)) + \sum_v f(U(v)) \tag{2.1a}$$

Subject to:

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

$$\forall i; \sum_{\pi \in P_i} f(\pi) = D_i \tag{2.1c}$$

$$\forall \pi, \forall e; p(\pi, e) \geq 0 \tag{2.1d}$$

Edge-based formulation:

$$\text{Minimize:} \sum_e f(U(e)) + \sum_v f(U(v)) \tag{2.2a}$$

Subject to:

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

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

$$\forall i, \forall (s - v); \sum_v f_i(e) = D_i \tag{2.2d}$$

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

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

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

Understand edge-based formulation: 2.2b is the same flow conservation, and 2.2c, g $p_i(v)$ shows the flow demand should be decreasing, 2.2d, e, f ensures the flow demand and process demand relations.

Flow Size Changes after Processing: the basic model shows that in-network processing does not affect the traffic size. However the change of flow size after processing is a common case 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 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 flow size change only applies the gaining factor to part of the flow to be processed at the vertex.

The change can be easily captured in the formulation, if there is a size change, at each vertex, we have $\sum\limits_{in} w(e) - \sum\limits_{out} w(e) = p(v)$ and $\sum\limits_{out} (f(e) - w(e)) - \sum\limits_{in} (f(e) - w(e)) = p(v) * r$, assume r is a positive gaining factor. If $r = 1$, we have exactly the flow conservation $\sum\limits_{out} f(e) - \sum\limits_{in} f(e) = 0$.

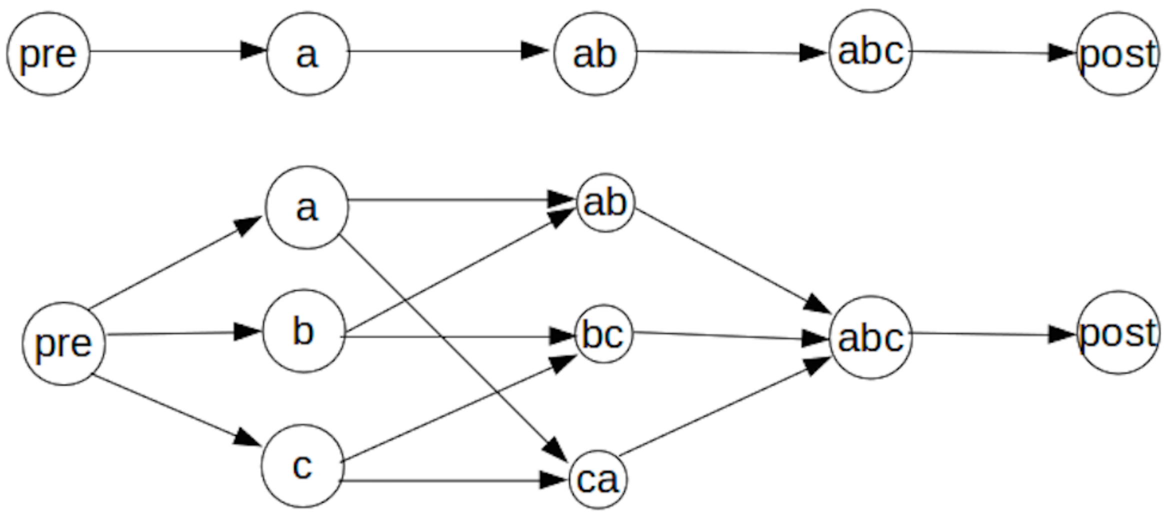

Figure 2: Up: serial tasks, Down: parallel tasks; a,b,c represents three different tasks and each link represents one dependency

2.2 Multiple tasks as a DAG In practice, a flow usually needs several different processes before sink. In the above formulation we assume there is only one task, or each vertex can run all types of flow processing where we can bundle all tasks into one task. Nevertheless if we have specialized hardware or two types of processing are preferred to be separated 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. We can satisfy the new requirement by modifying the optimization formulation. We consider two corner cases where there are N serial or N parallel tasks, and then we extend this to a general case where N tasks have a DAG relation.

For N serial tasks, we have $C^n(v); n \in \{1 \dots N\}$ for N different processing capacities. In optimization formulation 2.2, we can think of two different types of flows, pre-processed and post-processed, and in this model we have $N+1$ types of flows: pre-n-post-$(n-1)$ processed where $n \in \{1 \dots N\}$ and fully processed flows. 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, $p_i^n = \sum\limits_{in} w_i^n(e) - \sum\limits_{out} w_i^n(e)$. 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 sequence. The complexity of this formulation increases in a linear relation to the number of tasks N.

For N parallel tasks, again we have $C^n(v); n \in \{1 \dots N\}$ for N different processing capacities. Unlike serial relation, the number of types of flow grows exponentially, at most there can be 2^N types of flows. We have $O(2^N)$ inequalities represents the dependencies and $O(N)$ inequalities represents processing capacity relations.

To generalize, if there are tasks with a DAG relation where each directed link (p, q) represents a dependency of q to p, there is an inequality relation $w_i^q(e) \geq w_i^p(e)$ for each link (p, q). For any k parallel tasks, we need to build $O(2^k)$ branches and establish the relation using the method above.

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 edge based LP for section2.1

Here we use a max flow formulation for a single source and single sink flow to show the equivalence of formulation, we can simply extend this to min-utilization function based formulation for multi-source, multi-sink flows.

Rewrite the two optimization problem:

Path-based formulation:

$$\text{Max: } \sum_{pi \in P} f(\pi) \tag{A.1a}$$

Subject to:

$$\forall \pi; \sum_{e \in \pi} p(\pi, e) = f(\pi) \tag{A.1b}$$

$$\forall e; \sum_{\pi \in P : e \in \pi} f(\pi) \leq B(e) \tag{A.1c}$$

$$\forall v; \sum_{\pi \in P} \sum_{e \equiv (u,v) \in \pi} p(\pi, e) \leq C(v) \tag{A.1d}$$

$$\forall \pi, \forall e; p(\pi, e) \geq 0 \tag{A.1e}$$

Edge-based formulation:

$$\text{Max: } \sum_{e=(s,v)} f(e) \tag{A.2a}$$

Subject to:

$$\forall v \neq s, t; \sum_{in} f(e) = \sum_{out} f(e) \tag{A.2b}$$

$$\forall v; p(v) = \sum_{in} w(e) - \sum_{out} w(e) \tag{A.2c}$$

$$\forall e; f(e) \leq B(e) \tag{A.2d}$$

$$\forall v; p(v) \leq C(v) \tag{A.2e}$$

$$\forall (s - v); w(e) = f(e) \tag{A.2f}$$

$$\forall (v - t); w(e) = 0 \tag{A.2g}$$

$$\forall e; w(e), f(e) - w(e), p(v) \geq 0 \tag{A.2h}$$

From the formulation we show that.

- *Direction A:* If there is a path-based LP solution, we have an edge-based solution.
- *Direction B:* If there is an edge-based LP solution, we have a path-based solution.

For *Direction A:*

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

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

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

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

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

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

For *Direction B*.

That is, if we have a solution which tells us that if we can assign certain amount of flow and processing demand at each edge, we are able to construct paths with a certain amount of flow and corresponding processing demand and process workload at every/some node along the path in a certain way.

Setup: A directed graph $G(V, E)$. We have a solution of from edge-based LP, with f(e) is the flow for each edge and w(e) is workload demand at that edge. We also have processing work at each node p(v) and it is simply $p(v) = \sum\limits_{e\in E_{u,v}} w_(e) - \sum\limits_{e\in E_{v,w}} w_(e)$.

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. Note we might have cycles (or even two flows with opposite directions if the two vertices are connected bidirectionally) at the same edge. First we run algorithm [Path Construction] to get one path to allocate flow. For each flow path, we run flow allocation and update the graph, we exhaustively do it until we place all flow and workload demand, this step is essentially captured in algorithm [Flow Placement].

We prove two aspects for this algorithm:

1. there is $p(v) > 0$, we can always find a path with non-zero flow
2. constraints are held for the reduced graph

Here we introduce an variable ρ for each edge e where $\rho_e = \frac{w(e)}{f(e)}$. We divide flow into two types, processed and unprocessed f^1 and f^2; $f^1 = w$ and $f^2 = f - w$, we can translate $\rho = \frac{f^1}{f^1+f^2}$.

Lemma A.1. *If there is a cycle in the path composition, we can achieve in the cycle there are two different edges where one has $min(f^2) = 0$ and one has $min(f^1) = 0$.*

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

1. 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.

2. 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 fully unprocessed and get back the flow fully processed.

Lemma A.2. *If there is a cycle, we always have at least one edge with $\rho = 1$ and one edge with $\rho = 0$.*

Proof. Here since $f^2 = 0$ for at least one edge based on Lemma A.1, we have $\rho = 1$ for the edge. The same way to get $\rho_{out} = 0$ since $f^1 = 0$ for at least one edge.

```
Data: $G(V, E)$, $w(e)$, $f(e)$ for $\forall e \in E$ and $p(v)$ for $\forall v \in V$
Result: path $\pi$
pick a node v with $p(v) > 0$;
//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 $\rho_{out} \equiv \frac{w(e_{out})}{f(e_{out})}$;
```

Algorithm 1: Path Construction

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 $\rho < 1$. There may be cycles in the traversal, however since the same edge is never picked twice so there is no loop. 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 $\exists e; w(\pi, e) > 0$. Since $\delta = min(p(v), p(e_i), f(\pi) - p(\pi, e_{i+1}))$; at node v where $p(v) > 0$; we have $p(e_i) > 0$ because $[\rho_{in} = \frac{p(e_{in})}{f(e_{in})}] > \rho_{out} \geq 0$. If $\delta = 0$ we have $f(\pi) - p(\pi, e_{i+1}) = 0$, which lead to $p(\pi, e_{i+1}) > 0$, otherwise $\delta > 0; p(\pi, e_i) = [\delta + p(\pi, e_{i+1})] > 0$.

Finally since $p(\pi, e)$ is using backward greedy algorithm, our algorithm by design conserve the flow and ensures workload demand is decreasing.

Lemma A.4. (Flow Placement) *Algorithm 2 conserves all the constraints for the reduced graph.*

Proof. we show that all the constraints are satisfied:

for A.2b: $\forall v \in \pi; \sum\limits_{in} f(e) - \sum\limits_{out} f(e) = \sum\limits_{in \neq e_i} f(e) - \sum\limits_{out \neq e_{i+1}} f(e) + [f(e_i) - f(\pi)] - [f(e_{i+1}) - f(\pi)] = 0$

A.2d: $\forall e \in \pi; f(e) = f(e) - f(\pi) \leq B(e) - f(\pi) = B^{new}(e)$

A.2e: $\forall v \in \pi; p(v) - \delta \leq C(v) - \delta = C^{new}(v)$

A.2f and A.2g are ensured by greedy algorithm, $p(\pi, e_1) = f(\pi)$ and $p(\pi, e_k) = 0$.

A.2h constraints are satisfied by numerical relations.

So this proves *Direction B*, and therefore the two formulations are equivalent.

```
**Data**: $G(V, E)$, $w(e)$, $f(e)$ for $\forall e \in E$ and $p(v)$ for $\forall v \in V$
**Result**: $f(\pi)$, $p(\pi, e)$ (in which $e \in \pi$)
**while** $\exists v; p(v) > 0$ **do**
    find a path $\pi =< e_1, \ldots, e_k >$ from [Path Construction];
    $f(\pi) = minf((e_i),\ i \in< 1, \ldots, k >$;
    $p(\pi) = 0$;
    $p(\pi, e_k) = 0$;
    **for** $i \leftarrow (k-1)$ **to** 1 **do**
        (u,v)=$e_i$;
        //$\delta_i$: workload processed at node $i$;
        $\delta_i = min(p(v), w(e_i), f(\pi) - p(\pi))$;
        //Update workload;
        $p(\pi, e_i) = \delta_i$;
        $p(\pi) = p(\pi) + \delta_i$;
        $w(e_i) = w(e_i) - p(\pi)$;
        $p(v) = p(v) - \delta_i$;
        $C(v) = C(v) - \delta_i$;
    **end**
    $f(\pi) = p(\pi)$
    //Update flow;
    **for** $i \leftarrow 1$ **to** $k$ **do**
        $f(e_i) = f(e_i) - f(\pi)$;
        $B(e_i) = B(e_i) - f(\pi)$;
    **end**
**end**
```

Algorithm 2: Flow Placement