Rounding fractional flows in $O(m \log \frac{n^2}{m})$

Donggu Kang
MIT
donggu@mit.edu

James Payor
MIT
payor@mit.edu

Abstract

Algorithms for approximating or algebraically solving network flow problems with integral capacities often produce solutions that assign fractional flow to edges, although it is usually desired to obtain an integral solution. An integral solution with no worse cost always exists, and existing work has shown that such a solution can be found in $O(m \log n)$ time. This paper first presents such an algorithm due to [Peng, 2014]. Then we describe a new algorithm that rounds a solution in $O(n^2)$ time, with simpler implementation and better performance for dense graphs. We then combine the approaches of each algorithm to achieve rounding in $O(m \log \frac{n^2}{m})$, which is strictly faster than previous work.

Contents

1 Introduction

Algorithms for approximating or algebraically solving network flow problems with integral capacities often produce solutions that assign fractional flow to edges. However, it is usually desired to obtain an integral solution from this fractional flow - and it can be shown that such a solution with no worse cost always exists. Many recent papers seeking fast max-flow approximations and fast exact solutions for classes of graphs make use of flow rounding following an initial algebraic approximation, as seen in [Lee et al., 2013] and [Madry, 2013]. Thus, fast algorithms for finding fractional flow solutions are complemented by a fast rounding algorithm.

Existing work has shown that rounding a fractional flow to an integer flow of no worse cost in a costed, integral capacitated graph can be achieved in $O(m \log n)$ time. [Lee et al., 2013] showed such an algorithm relying on random-walk analysis, and [Madry, 2013] gave an algorithm that reduced the general flow rounding problem into the perfect b-matching rounding problem.

Note that there is no difference in the difficulty of rounding flows and circulations. Circulations are a special case of flows, so any algorithm that rounds a flow will also round a circulation. To round a flow given an algorithm for a circulation, we can add a dummy edge $(sink, source)$, with flow value equal to the flow into *sink* and cost $-\infty$, which creates a circulation we can round. The negative infinite cost guarantees the rounding doesn't decrease the *source*-*sink* flow.

Further, such a scheme also lets us round fractional solutions to approximate max-flow problems without decreasing the flow, by taking the costs of the edges to be zero. As such, an algorithm for rounding a min-cost circulation can be applied to round fractional solutions to all of these standard problems.

We first present preliminary theorems and background on required data structures. Then we introduce the algorithm by [Peng, 2014] that cancels cycles using the link-cut tree data structure [Sleator and Tarjan, 1981], and our approach to rounding in $O(n^2)$ time. Finally, we apply our approach in the $O(n^2)$ algorithm on top of link-cut trees of restricted size to achieve an algorithm with $O(m \log \frac{n^2}{m})$ time, performing strictly better than previous approaches in both dense and sparse graphs.

2 Background

2.1 Fractional and integral circulations

A *circulation* over a graph $G = (V, E)$ with *capacities* $c(u, v)$ on each edge (u, v) is an assignment $f(u, v)$ of flow to each $(u, v) \in E$ that satisfies three conditions:

- $\forall v \in V, \sum_{u|(u,v)\in E} f(u, v) = 0$: The net flow into a node is zero.
- $\forall (u, v) \in E, f(u, v) = -f(u, v)$: The assignment of flow is antisymmetric.
- $\forall (u, v) \in E, f(u, v) \leq c(u, v)$: The assignment of flow to each edge does not exceed the capacity of the edge.

We call a circulation *fractional* if it may assign edges (u, v) a non-integral value of flow $f(u, v)$. Conversely, circulation is *integral* if all values of $f(u, v)$ are integral.

2.2 Circulations and fractional cycles

The key to the algorithms presented in this paper is the following observation about *fractional circulations* - that is, circulations that may have non-integral assignments of flow $f(u,v)$.

Theorem 1. *If a circulation f has the property that the subgraph containing all edges with potentially fractional flow forms a forest, then f is an integral circulation.*

Proof. Suppose not. Then there must be a leaf node v in the forest with only one connected edge that has fractional flow. Then the net flow into v cannot be zero as its remaining edges have integral flow, which violates the net flow condition on a circulation. $\square$

This leads to the following corollary.

Corollary 1. *If a circulation f contains no cycles of edges with fractional flow, then f is an integral circulation.*

Proof. The subgraph of edges with fractional flow can only be a forest if there are no cycles. $\square$

So, if we are able to cancel all fractional cycles in a circulation, the remaining circulation must be integral. Each of the algorithms presented in this paper develops methods to efficiently find and cancel fractional cycles in order to round circulations.

2.3 Link-cut trees

[Peng, 2014]'s $O(m \log n)$ algorithm and our $O(m \log \frac{n^2}{m})$ algorithm both utilize the link-cut trees of [Sleator and Tarjan, 1981]. A link-cut tree is a data structure that allows the maintenance of a dynamic forest of rooted trees over a set of nodes, and tree-path operations over the forest. This section details the amortized logarithmic time operations enabled by the data structure that we will require in section 3. The implemenation details (splay trees, preferred paths) and analysis thereof are neglected here, as the original paper can provide further background.

Each operation described below is logarithmic in the maximum link-cut tree size involved - e.g. $Link(u,v)$ runs in $O(\log n_u + \log n_v)$ where n_u and n_v are the size of the trees u and v are contained in. This property is explicitly required when we restrict the size of link-cut trees in section 5, and in other cases guarantees that operations take $O(\log n)$ time in a graph of n nodes.

We make use of the following operations supported by link-cut trees:

- $Link(u,v)$: Create an edge between u and v. Makes the node v a child of u.
- $Cut(u,v)$: Remove the edge between u and v.
- $FindRoot(v)$: Find the root of a tree which v belongs to.
- $LCA(u,v)$: Report the lowest common ancestor of u and v.
- $PathAdd(u,v,c)$: Add a number c to the weight of every edge along the u-v path.
- $PathMin(u,v)$: Report the edge with minimum weight over the u-v path. Break ties by reporting the edge closest to u.
- $PathSum(u,v)$: Report the sum of edge weights over the u-v path.

3 Rounding in $O(m \log n)$

There are several existing works that give algoritms for rounding a fractional flow into an integral flow in $O(m \log n)$. [Peng, 2014] gave a $O(m \log n)$ algorithm for rounding circulations In this section, we present this algorithm. Given a costed graph $G = (V, E)$ with integral capacities, and a fractional circulation over this graph, the algorithm will find an integral circulation of no worse cost over the graph.

This algorithm relies on the insight of corollary 1: it repeatedly cancels fractional cycles, and when none remains the circulation must be integral.

3.1 Preliminaries

First, we will establish the following notation:

- Let $f(u, v)$ be the fractional circulation (as defined in section 2.1) to be rounded.
- Let $f'(u, v)$ be the new circulation the algorithm finds, which will ultimately be integer-valued.
- Let $available(u, v) = \lceil f(u, v) \rceil - f'(u, v)$. For example, if $f(u, v) = f'(u, v) = 0.6$, $available(u, v) = 0.4$ and $available(v, u) = 0.6$, indicating that we can push 0.4 units of flow from u to v or 0.6 units from v to u before the flow on the edge becomes integral.
- Let $cost(u, v)$ be the cost for each unit flow on the edge (u, v).

We will refer to $available(u, v)$ as the *availability* of the edge (u, v). We will require at all times that f' satisfies net flow constraints, as well as $available(u, v) \geq 0$ for all (u, v), and claim that this guarantees that f' is a feasible circulation.

Theorem 2. *If f is a feasible circulation and f' satisfies the net flow constraints, then f' is a feasible circulation if $available(u, v) \geq 0$ for all (u, v).*

Proof. If $available(u, v) \geq 0$ and $available(v, u) \geq 0$, then:

$$\lceil f(u, v) \rceil \geq f'(u, v) \wedge \lceil -f(u, v) \rceil \geq -f'(u, v) \implies \lfloor f(u, v) \rfloor \leq f'(u, v) \leq \lceil f(u, v) \rceil$$

As the graph has integer capacities, any flow value on (u, v) between $\lceil f(u, v) \rceil$ and $\lfloor f(u, v) \rfloor$ must satisfy the capacity constraints given that $f(u, v)$ does. So f' satisfies all capacity and net flow constraints, and is thus a feasible flow. □

3.2 Data Structure

We use a link-cut tree to maintain a forest that consists of only edges (u, v) with fractional flow, $f'(u, v)$. In accordance with the operations given in section 2.3, we use the following operations, each of which runs in $O(\log n)$ time over our graph of n nodes:

- $PushFlow(u, v, x)$: Add x to the flow values of each edge on the u-v path, where x can be negative. This is a *PathUpdate* operation.
- $FindMin(u, v)$: Report the minimum $available(x, y)$ for an edge (x, y) on the u-v path. This is a *PathQuery* operation we can support, as $PushFlow$ simply subtracts from the availability along a path, akin to the situation with $PathMin$ and $PathAdd$ in section 2.3.

- $FindIntegralEdge(u,v)$: Report an edge with integral flow on the u-v path. It may report none. Integral edges are found when $Min(u,v)$ or $Min(v,u)$ is zero, as zero availability implies an integral flow.
- $FindCost(u,v)$: Report the cost per unit flow along the u-v path.

Note that typically queries over the a link-cut tree are undirectional, but $FindMin(u,v)$ and $FindMin(v,u)$ may report different values as $available(x,y) \neq available(y,x)$. We can achieve this by remembering availability in both directions on edges of the tree, aggregating minimums of both values.

3.3 Algorithm

Initially we are given a circulation f over a costed graph $G = (V,E)$, to be rounded. We initialize our new circulation as $f' = f$. We construct a graph without edges $G' = (V, E' = \{\})$, and add edges $(u,v) \in E$ with fractional flow $f'(u,v)$ one by one. If the edge to be added forms a cycle in G' (as (E,H) in figure 1a below), we push flow around the cycle in the direction of a smaller cost until at least one edge becomes integral (figure 1b). The amount we have to push to achieve this is the *minimum availability* along the cycle, as availability measures how close the flow along each edge is to the next integer.

Now, the flow in one of two directions around the cycle will have negative or zero cost, so we never increase the cost of the circulation f'. As we add this flow to f', note that we maintain that f' is a feasible circulation. Pushing flow around a cycle cannot violate any net flow constraints, and theorem 2 applies as we never reduce the availability of an edge below zero.

Finally, we remove integral edges from G' introduced by pushing flow (figure 1c). This guarantees that G' no longer has a cycle, as we remove at least one edge for every cycle we introduce. After processing all edges with fractional flow, G' must still be a forest. It also must be the subgraph of remaining fractional edges in f'. Then by theorem 1, f' is an integral circulation (and G' has no edges remaining).

A more detailed specification of the algorithm is as follows:

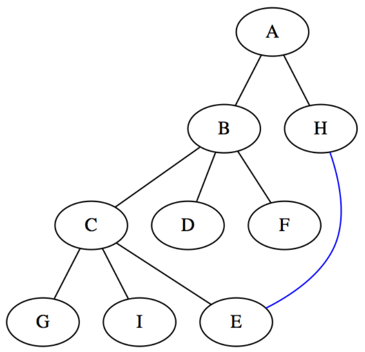
1. Represent $G' = (V,E')$ using link-cut trees, with each node initially it's own tree.
2. For every edge $(u,v) \in E$ with fractional $f'(u,v)$:
   - If $FindRoot(u) \neq FindRoot(v)$: Simply perform $Link(u,v)$ to add the edge to G', as this doesn't introduce a cycle.
   - If $FindRoot(u) = FindRoot(v)$: This means there exists a path P between u and v. The path P together with (u,v) forms a cycle (as in figure 1a). The cost of unit flow through P and the edge (v,u) is $cost(u,v) - c_e$. If this cost isn't positive, we push flow around the cycle until one of edges in the cycle becomes integral without increasing the total cost. The amount of additional flow to make an edge integral is the minimum availability on the cycle, $flowAmount = min(FindMin(u,v), available(v,u))$. (In figure 1b, the minimum availability is 0.2.)

     To update the flow on the path P, we use $PushFlow(u,v,flowAmount)$. This makes at least one edge integral, because at least one edge now has availability zero. To remove such edges, while $FindIntegralEdge(u,v)$ isn't null, we perform $Cut(FindIntegralEdge(u,v))$. Finally, if $f'(u,v)$ is still not integral, we perform $Link(u,v)$ to add it to G'. In figure 1c, the edge with minimum availibility is (B,C) with 0.2, which becomes zero and is then disconnected.

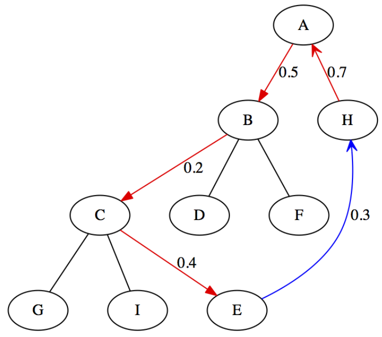
     If the cost of the cycle was positive, we can push flow in the other direction with negative cost as above.

3. After processing all fractional edges, f' is an feasible integral flow with no worse cost than f, by the above the analysis.

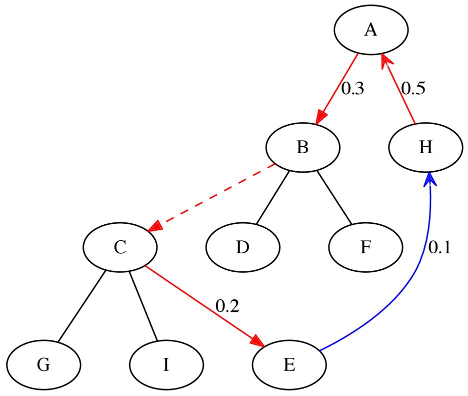
In the above, processing each edge takes $O(1)$ link-cut tree operations, excluding those involving removing integral edges. $FindIntegralEdge$ reports a result at most m times, so the cost of deleting integral edges that are formed is amortized to $O(1)$ per edge. As each operation takes $O(\log n)$, the total running time of the algorithm is $O(m \log n)$.



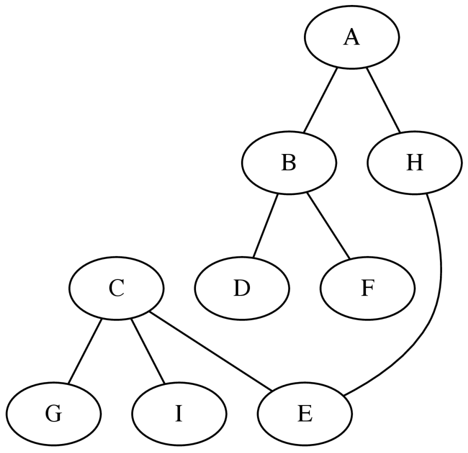
(a) Adding an edge (H,E) forms a cycle.



(b) The minimum availability on the cycle is $available(B,C) = 0.2$.



(c) Push 0.2 units of flow around the cycle, decreasing availabilities. (B,C) becomes integral, thus (B,C) is cut.



(d) Resulting link-cut tree.

Figure 1: A procedure of the $O(m \log n)$ algorithm.

4 Rounding in $O(n^2)$

In [Peng, 2014]'s algorithm, we added edges with fractional flow to the link-cut tree in some arbitrary order, maintaining a forest by cancelling fractional cycles as they occurred and deleting integral edges. Instead, we will process edges from the same node in one batch, cancelling all the cycles introduced at once. This takes $O(n)$ time for each node, achieving a running time of $O(n^2)$.

4.1 Algorithm

We maintain a tree of processed nodes. To process a node x, we consider its fractional edges that are connected to nodes in the tree. If x has more than 1 edge with the tree, then cycles will be formed, which we will cancel in batch. The algorithm that achieves this will be called *Cancel*, which we run on the root. After $Cancel(u)$ we guarantee that the subtree of u has no cycle through x - that is, there is at most one path to x through the subtree.

To perform $Cancel(u)$, first call $Cancel(c)$ for each child c of u. Each call to a child cancels cycles and returns the remaining path to x if exists, or $null$. After that, if more than one of u and its children have paths to x, there will be a cycle consisting of two disjoint paths to x. Let the paths be *pathDown* and *pathUp*, such that *pathDown* is a u-x path, *pathUp* is an x-u path, and $cost(pathDown) + cost(pathUp) \leq 0$. (If the cost is positive, we can switch *pathDown* and *pathUp* to negate the cost.) We will send flow F equal to the minimum availability of *pathDown* and *pathUp* around the cycle - down *pathDown*, and up *pathUp*. This will create at least one edge with availability zero on the cycle, and we remove this edge (which is now integral). Cancelling each cycle removes one path to x. We repeat this until there is only one path left, then return this path.

To send flow around all of the cycles through x in $O(n)$ time, instead of immediately updating the availability of every edge on *pathDown* and *pathUp*, we record F on the edge out of u on *pathDown*, and $-F$ on the edge out of u on *pathUp*, deferring pushing F and $-F$ units along the paths to x. After cancelling all the cycles formed by adding x, we traverse the tree again to push flow along these paths to x in the procedure *PushFlow*.

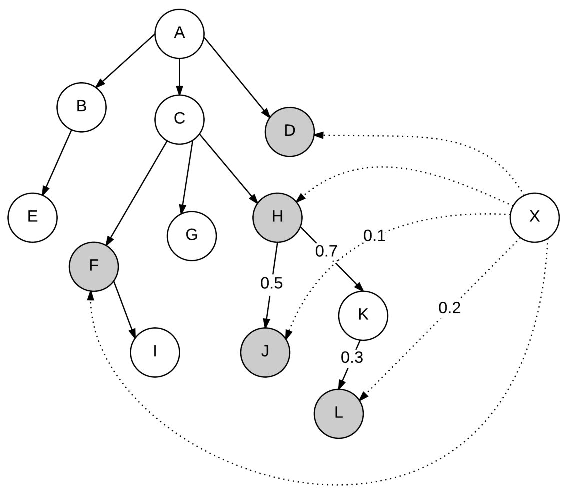
At termination, every fractional edge has been processed and only the edges left in the tree can have fractional flow. Then by theorem 1 we have an integral flow.

4.2 Example

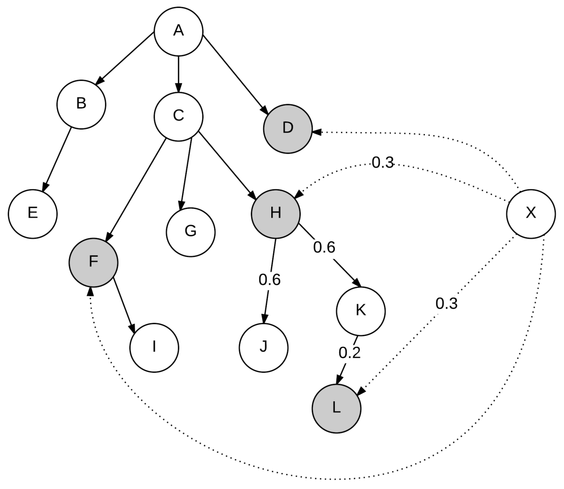
Figure 2 illustrates some of the steps in processing a node X with edges into the existing tree. In figure 2a we are performing $Cancel(H)$. $Cancel(J)$ and $Cancel(K)$ have each returned paths to X, which together form a cycle. The cost for the cycle is negative in the H-K direction, so we let *pathDown* be H-K-L-X and *pathUp* be X-J-H. The minimum availabilities on each path are 0.3 (for (K,L)) and 0.1 (for (X,J)), so we will send 0.1 units of flow around the cycle to make (X,J) integral, then delete (X,J). (Note that in the actual implementation, updates are deferred.)

We now arrive at figure 2b, still in $Cancel(H)$, with two paths to X remaining. Again the cycle cost is negative in the H-K direction, with *pathDown* as before and *pathUp* as X-H. The minimum availability on the paths in 0.2 for (K,L), so we send 0.2 units of flow around the cycle and delete (K,L). After this operation we have figure 2c. $Cancel(H)$ has now cancelled all cycles within the subtree, and returns the remaining path to X, which is the edge (H,X).

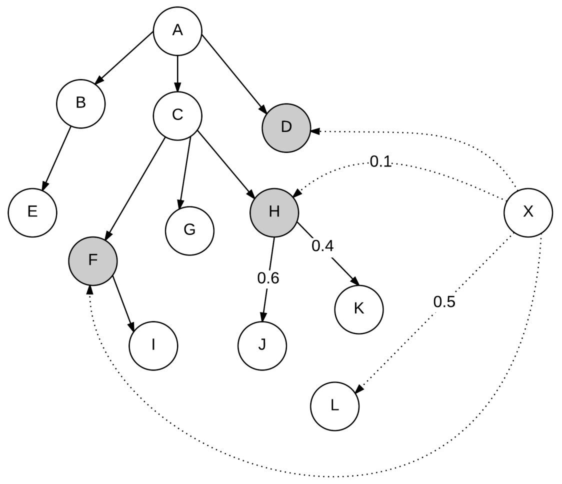
Now, $Cancel(C)$ can begin cancelling cycles, as in figure 2d. Both F and H have returned paths to X, and we cancel that cycle by sending 0.1 units of flow around in the C-F direction, deleting (C,F). This takes us to figure 2e, and $Cancel(C)$ is complete. Note that there is now a zero availability edge (X,H) as we only delete one integral edge for each cycle we cancel.



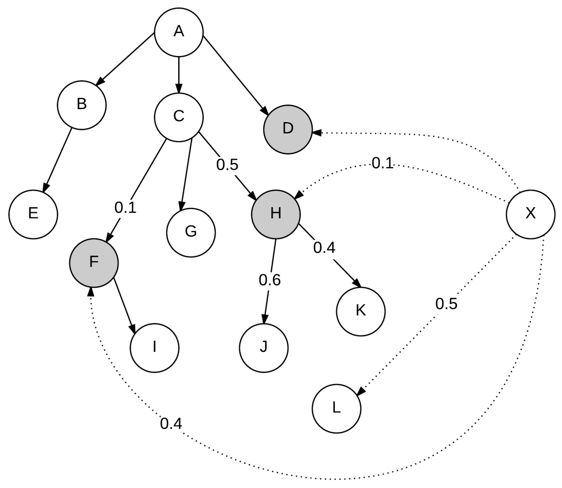
(a) In $Cancel(H)$, cancelling the cycle through J and K.



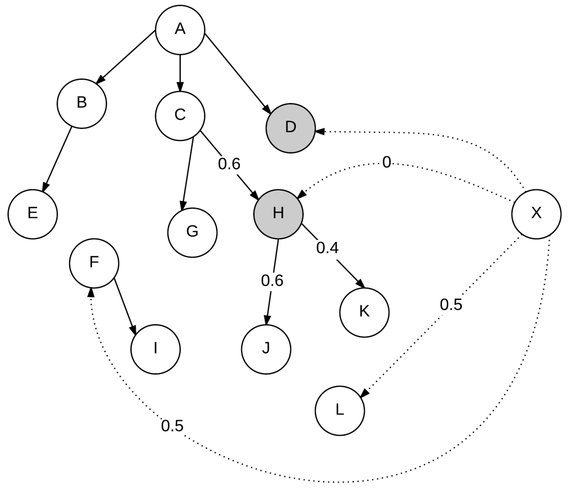
(b) In $Cancel(H)$, cancelling the cycle through L.



(c) The result when $Cancel(H)$ returns.



(d) In $Cancel(C)$, cancelling the cycle through F and H.



(e) The result when $Cancel(C)$ returns.
Note $available(X,H) = 0$.

Figure 2: $O(n^2)$ example

4.3 Pseudocode

The following pseudocode for *RoundFractionalFlow* implements this algorithm. It computes an integral flow f' given the original flow f and flow graph $G = (V,E)$. For every connected component in the graph it maintains a tree (V',E') and incrementally processes nodes connected to the existing tree with *ProcessNode*. *ProcessNode* in turn runs the cycle-cancelling traversal *Cancel* and the flow-pushing traversal 'PushFlow' to adjust f' to cancel fractional cycles with the new node, and restores the tree (V',E') by adding v to V' and removing one integral edge from each cancelled cycle.

Algorithm 1 RoundFractionalFlow implementation.

```
 1: function RoundFractionalFlow($G = (V,E)$, $f$)
 2:     for all $x \in V$ do
 3:         $processed(x) \leftarrow$ false
 4:     $f' \leftarrow f$                                    ▷ Initialise new flow.
 5:     function TraverseGraph($v$)
 6:         if not $processed(v)$ then
 7:             $processed(v) \leftarrow$ true
 8:             $root \leftarrow v$
 9:             for all $(x,v) | (x,v) \in E, available(x,v) \neq 0$ do
10:                 ProcessNode($x$)
11:                 TraverseGraph($x$)
12:     for all $v \in V$ do
13:         $V' = \{\}, E' = \{\}$
14:         TraverseGraph($v$)
      return $f'$
```

In this implementation, we defer pushing flow and deleting edges. The associative array $flowToPush(u,v)$ tracks deferred flow and the list *toDelete* tracks the edges that are to be removed.

Algorithm 2 ProcessNode implementation.

```
 1: function ProcessNode($x$)
 2:     $toDelete \leftarrow \{\}$
 3:     for all $e \in E'$ do
 4:         $flowToPush(e) \leftarrow 0$
 5:     for all $(u,x) | (u,x) \in E, processed(u)$ do
 6:         add $(u,x)$ to $E'$
 7:     Cancel($root$, null , $x$)
 8:     PushFlow($root$, null , $x$, 0)
 9:     for all $e \in toDelete$ do
10:         remove $e$ from $E'$
```

When we return from a node in *Cancel*, we remember the first edge on the remaining path to x (if it exists) with the associative array $edgeToX(u)$. This will be used in *PushFlow* later.

Algorithm 3 Cancel implementation.

```
function CANCEL($u$, $parent$, $x$)
    if $u = x$ then
        return 0, null , $\infty$, null , $\infty$
    curEdge, curCost $\leftarrow$ null , $\infty$
    curDownMin, curDownAvail $\leftarrow$ null , 0
    curUpMin, curUpAvail $\leftarrow$ null , 0
    for all $v | (u,v) \in E', v \neq parent$ do                  ▷ Iterating over all children.
        childCost, childDownMin, childDownAvail,
                          childUpMin, childUpAvail $\leftarrow$ CANCEL($v$, $u$, $x$)
        if childCost = $\infty$ then                              ▷ This indicates no path to $x$.
            continue
        nextCost $\leftarrow$ $cost(u,v)+$ childCost
        if $available(u,v) <$ childDownAvail then
            nextDownMin $\leftarrow$ $(u,v)$
            nextDownAvail $\leftarrow$ $available(u,v)$
        else
            nextDownMin $\leftarrow$ childDownMin
            nextDownAvail $\leftarrow$ childDownAvail
        if $available(v,u) <$ childUpAvail then
            nextUpMin $\leftarrow$ $(v,u)$
            nextUpAvail $\leftarrow$ $available(v,u)$
        else
            nextUpMin $\leftarrow$ childUpMin
            nextUpAvail $\leftarrow$ childUpAvail
        if curCost = null  then
            needToSwichPaths $\leftarrow$ true
        else
            if currentCost - nextCost $< 0$ then        ▷ It's profitable to take flow from the child.
                flowAmount $\leftarrow$ min(curDownAvail, nextUpAvail)
                curDownAvail $-=$ flowAmount
                curUpAvail $+=$ flowAmount
                nextUpAvail $-=$ flowAmount
                nextDownAvail $+=$ flowAmount
                $flowToPush$(curEdge) $+=$ flowAmount
                $flowToPush((u,v))$ $-=$ flowAmount
                if curDownAvail $= 0$ then
                    add curDownMin to $toDelete$
                    needToSwitchPaths $\leftarrow$ true
                else
                    add nextUpMin to $toDelete$
                    needToSwichPaths $\leftarrow$ false
```

```
42:             else                                   ▷ It's profitable to push flow to the child.
43:                 flowAmount ← min(nextDownAvail, curUpAvail)
44:                 curDownAvail += flowAmount
45:                 curUpAvail −= flowAmount
46:                 nextDownAvail −= flowAmount
47:                 nextUpAvail += flowAmount
48:                 $flowToPush$(curEdge) −= flowAmount
49:                 $flowToPush((u,v))$ += flowAmount
50:                 if curUpAvail = 0 then
51:                     add curUpMin to $toDelete$
52:                     needToSwichPaths ← true
53:                 else
54:                     add nextDownMin to $toDelete$
55:                     needToSwichPaths ← false
56:         if needToSwitchPaths then         ▷ Switch current $u$-$x$ path to the path through $v$.
57:             curEdge ← $(u,v)$
58:             curCost ← nextCost
59:             curDownMin, curDownAvail ← nextDownMin, nextDownAvail
60:             curUpMin, curUpAvail ← nextUpMin, nextUpAvail
61:     $edgeToX(u)$ ← curEdge
62:     return curCost, curDownMin, curDownAvail, curUpMin, curUpAvail
```

To implement *PushFlow*, we need to understand how to handle flow coming into the current node u from its parent. This flow is intended for the path to x that the node returned to its parent earlier, so we add it to the flow to be pushed down $edgeToX(u)$. The remainder is straightforward - we add $flowToPush(u,v)$ to $f'(u,v)$ for every child v, then recurse on v with $flowToPush(u,v)$ incoming flow.

Algorithm 4 PushFlow implementation.

```
1: function PushFlow($u$, $parent$, $x$, incomingFlow)
2:     if $u = x$ then
3:         return
4:     if incomingFlow $\neq 0$ then
5:         $flowToPush(edgeToX(u))$ += incomingFlow
6:     for all $v | (u,v) \in E', v \neq parent$ do              ▷ Iterating over all children.
7:         $f'(u,v)$ += $flowToPush(u,v)$
8:         PushFlow($v$, $u$, $x$, $flowToPush(u,v)$)
```

The *Cancel* and *PushFlow* procedure both do work linear in the number of processed nodes and edges between them, for $O(n)$ work it total. As ProcessNode only adds linear overhead, each node is processed in $O(n)$ time, for an overall runtime of $O(n^2)$.

5 Rounding in $O(m \log \frac{n^2}{m})$

Combining the cycle-cancelling via link-cut trees of the $O(m \log n)$ algorithm and the batch processing of the $O(n^2)$ algorithm, it's possible to achieve $O(m \log \frac{n^2}{m})$. We achieve this with a new data structure that allows us to limit the nodes involved in path operations to $O(\frac{n^2}{m})$.

5.1 k-link-cut tree

We introduce a k-link-cut tree as a method of restricting the sizes of involved link-cut trees. The basic idea is to partition a tree into clusters so the size of each cluster is limited to some constant, $2k$. We maintain a link-cut tree over each cluster.

An edge connecting two nodes in different clusters is called a *joint*. Joints are not visible to the link-cut tree operations, so different clusters are considered disconnected for these operations. Figure 3a shows a tree partitioned into 6 clusters, and in this example $FindRoot(F)$ D, not A, because the joint (B,C) is not visible to the link-cut trees. We refer to the tree in which each node is a cluster and the edges are joints as the *primary tree*.

We define operations for the k-link-cut tree below. Operations of link-cut trees are prefixed by "*Old*".

- $FindGlobalRoot(v)$: Report the root of a tree containing v. Joints are considered visible. To perform this, we may have to move up through multiple clusters and joints.
- $FindClusterRoot(v)$: Report the root of a cluster containing v. Joints are considered invisible. This is the same as $OldFindRoot(v)$.
- $Size(v)$: Report the size of v's cluster.
- $Link(u,v)$: If $Size(u)+Size(v) \le 2k$, perform $OldLink(u,v)$, and u and v will belong to the same cluster. If the $Size(u)+Size(v) > 2k$, add a joint (u,v) without merging the clusters.
- $Cut(u,v)$: If (u,v) is a joint, simply remove it. If not, u and v are in the same cluster, and we use $OldCut(u,v)$ to split the cluster in two. If u's cluster has a neighboring cluster such that the sum of their sizes is w such that $size(cluster(uorv)) + size(w) \le 2k$, merge them, and repeat until there is no such neighbor.
- $Min, IntEdge, Cost, Flow(u,v)$: There may be multiple clusters between the node u and v. We simply traverse the clusters and perform the operation on each cluster in $O(\log k)$. The number of clusters is $O(d+n/k)$, which will be proven later. Thus the running time is $O((d+n/k)\log k)$.

All operations try to merge two adjacent clusters whenever their total size is not greater than $2k$. The modified link-cut tree has several interesting properties:

1. Each cluster has at most 2k nodes.
2. For any two adjacent clusters, one of them has more than k nodes.

   Proof. If not, they should have been merged into one. □

3. The number of internal nodes of a primary tree is $O(n/k)$.

Proof. First we decompose the tree into a set of paths with one leaf node per path. This can be achieved by numbering the leaf nodes in some order, and setting the first path to be from the leaf node 1 to the root, and then the $i(>1)$-th path to be from the i-th leaf node to its lowest common ancestor with the previous leaf nodes. Let x_i be the number of internal clusters in the i-th path, and n_i be the total number of nodes in the i-th path. For every two clusters on a path, there are at least k nodes. Thus $n_i \geq \lfloor \frac{x+1}{2} \rfloor k \geq \frac{x}{2}k$. Summing up all the inequalities, we get $\sum x_i \leq \frac{n}{k}$. Thus the number of internal nodes of the primary tree is $O(n/k)$. □

5.2 Algorithm

The algorithm is similar to $O(n^2)$. Suppose we are adding the node v, and the number of edges to the processed nodes is d. Cycles are processed in this order:

Step1 *Process cycles within a cluster.*

Use Peng [2014]'s algorithm. Add edges and make a flow whenever there is a cycle. Every edge takes $O(\log k)$ to process, so this step takes $O(d \log k)$. After this step, clusters may be split into smaller ones. Instead of merging clusters now, later we will lazily merge split clusters with neighbors if their size is not greater than $2k$. After the step1, there is at most one edge between the node v and any cluster.

In Figure 3a, the cluster 3 has three edges from v. Looking at a cycle (V,I,G,V), the minimum edge (G,I) is canceled and results in two split clusters. After the step1, there is only one or no edge from v to each cluster (3b).

Step2 *Process cycles within a tree.*

Use the DFS approach in our $O(n^2)$ algorithm on the primary tree. The number of edges traversed is $O(d+n/k)$ because the number of edges from leaves to their parents is $O(d)$ and edges from internal nodes to their parents is $O(n/k)$ by the property 3. This proves the cost of $Min, IntEdge, Cost, Flow$ operations across multiple clusters is $O((d+n/k)\log k)$.

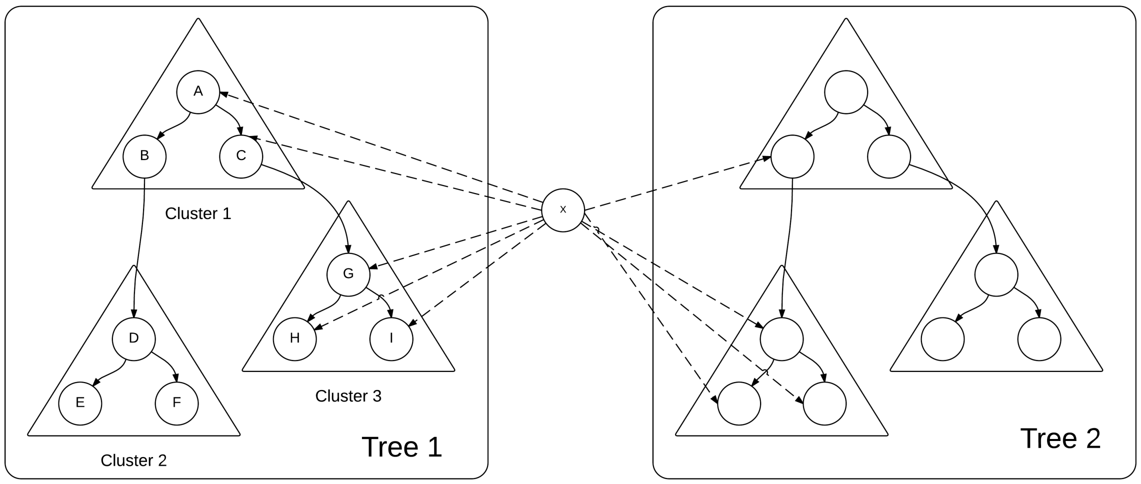
However, because of the lazy merging in the step1, we perform the merges if we find a pair of clusters to be merged while travesing the edges. the actual cost is $O((s+d+n/k)\log k)$ where s is the number of merges performed. Note that we relaxed the invariant that there are $O(n/k)$ internal nodes at any moment, but after s merges, we have $O(n/k)$ remaining edges involved in the traversal.

In Figure 3b, the cluster 3 has three edges from v. Looking at a cycle (V,I,G,V), the minimum edge (G,I) is canceled and results in two split clusters. After the step1, there is only one or no edge from v to each cluster (3b).

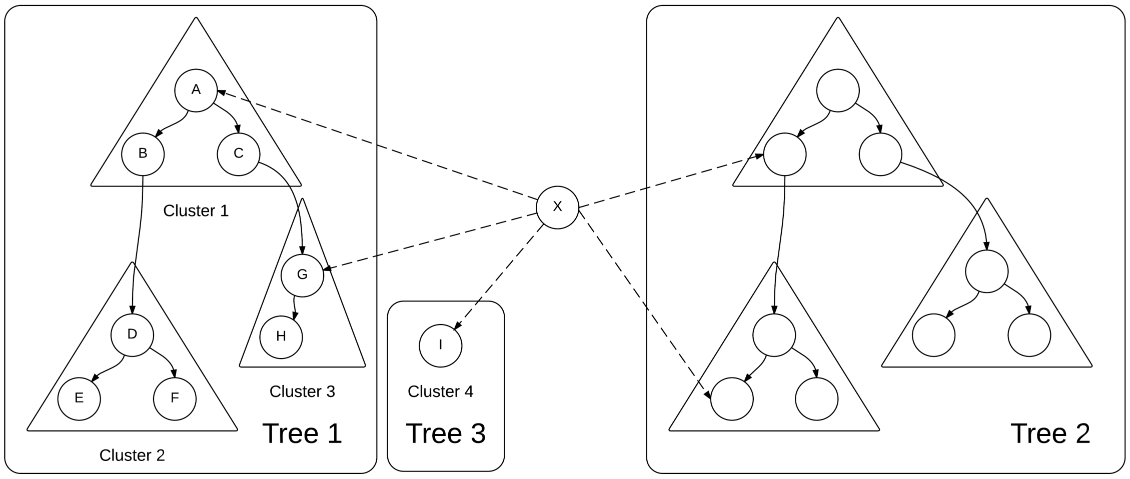
Step3 *Link clusters as needed.*

After Step2, the node v has at most one edge to each tree. There is no cycle. Simply link all trees with v. (For an edge (u,v), $Link(u,v)$). The cost of one linking is $O(\log k)$ so it's $O(d \log k)$ in total.

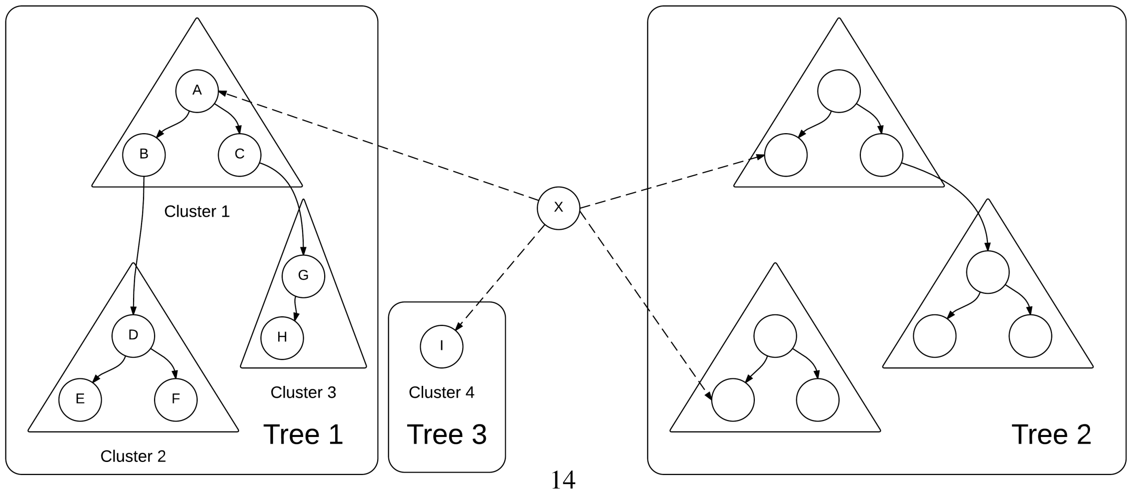
The above steps have $O((s+d+n/k)\log k)$ running time for adding the node v. From $\sum s = O(m), \sum d = O(m), \sum n/k = n^2/k$. the total running time is $O((m+n^2/k)\log k)$. Choosing $k = \frac{n^2}{m}$, we get $O(m \log \frac{n^2}{m})$.



(a) Step 1. Process cycles within each cluster



(b) Step 2. Process cycles within each tree



(c) Step 3. Join components

Figure 3: Steps of $O(m \log(n^2/m))$ algorithms

References

Lee, Y. T., Rao, S., and Srivastava, N. (2013). A New Approach to Computing Maximum Flows using Electrical Flows Categories and Subject Descriptors.

Madry, A. (2013). Navigating central path with electrical flows: From flows to Matchings, and back (Extended Abstract). In *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 253–262.

Peng, R. (2014). Converting Any Circulation to an Integral One of Smaller Cost in O(mlog(n)) time. *Personal Communication*.

Sleator, D. D. and Tarjan, R. E. (1981). A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81*. ACM Press.