A Survey and Implementation of Fast Approximate Maxflow Algorithms

Kaicheng Guo [1] **Zhengyu Zou** [1]

Abstract

We give a survey on several breakthroughs of graph sparsification and oblivious routing scheme subroutines that leads to an approximate maxflow algorithm that runs in $O(m\,\text{polylog}(n))$ time. We give a breakdown of the algorithmic pieces in the previous works of Madry (Madry, 2010), Sherman (Sherman, 2013), Kelner et al. (Kelner et al., 2013), Räcke et al. (Räcke et al., 2014), and Peng (Peng, 2016). We propose a revision of Peng's algorithm that achieves $\tilde{O}(m\log^{31} n)$ runtime guarantee, and a potential acceleration to $\tilde{O}(m\log^{17} n)$. We implement some highlights of the algorithms (see Appendix A, B) and give a qualitative examination on their performances.

1. Introduction

The maxflow problem and its dual, the minimum cut problem, are fundamental combinatorial optimization problem. It involves finding a feasible flow through a flow network that obtains the maximum possible flow rate. It has vast application in airline-scheduling, image segmentation etc. It also led to important tools in algorithm design such as augmenting path, dual algorithm, graph sparsification and electrical flows.

The maximum flow problem is to route as much flow as possible from the source to the sink, in other words find the flow f_{max} with maximum value while obeying the capacities. In section 2, we will show that there is an equivalent problem of routing a single unit of flow from s to t while minimizing the maximum conjestion $\frac{f_e}{c_e}$ on any edge.

The maxflow problem has a long history starting from 1955 when Lester R. Ford, Jr. and Delbert R. Fulkerson (Ford & Fulkerson, 1956) created the first known algorithm, the Ford–Fulkerson algorithm with complexity $O(|E|U)$. Since then deterministic approach via augmenting paths, network simplex, approximated approaches via l_2 relaxation into energy consumption problems (electric network flow) and implications in numerical linear algebra, graph sparsification approaches to accelerate the runtime of approximate maxflow algorithms (Vishnoi, 2012). Then, oblivious routing schemes came into play to deal with minimum congestion of flows on a network. Combinations of sparsification and oblivious routing schemes allow us to embed the problem on a dense graph to a more sparsified graph, which in turn allows us to call existing oblivious routing scheme routines on structurally simplier graphs while only scarificing a $O(\log n)$ factor of approximation accuracy. This recursive sparsify-routing cycle is finally closed by Peng's 2016 paper (Peng, 2016), where he discovered that a recursive approach starting with sparsifying the graph can effectively lead to a running time of $O(m\,\text{polylog}(n))$.

1.1. State-of-the-Art Fast Algorithm

In (Peng, 2016), the author presents improved algorithms for approximating maximum flows by exploiting a recursive approach. It uses the breakthrough on a more efficient algorithm for constructing oblivious routing schemes by (Räcke et al., 2014; Madry, 2010). Also, the algorithm calculate approximate maxflow using (Sherman, 2013). The core idea in the algorithm is to recursively refine the graph representation and its associated flow approximator, leading to progressively more accurate solutions. The recursive approach outlined in Algorithm 1 takes advantage of the fact that each iteration produces a smaller graph that still retains a close approximation to the original graph's flow characteristics. This smaller graph size has two main benefits: it reduces the computational complexity and it limits the error propagation through the recursive calls.

Algorithm 1 Approximate Max-Flow Scheme

1: Produce a graph H with size $m/\text{polylog}(n)$ that can $\text{polylog}(n)$-approximate G.
2: Construct an approximator for H using the Räcke et al. algorithm, making more recursive maximum flow calls.
3: Convert this scheme to one for G, and use it to solve approximate maximum flows.

In section 5, we provide an alternative proof sketch that gives an $O(m\log^{31} n\log^2\log n)$ algorithm. We then point at a potential direction for a faster $O(m\log^{17} n\log\log n)$ runtime guarantee using accelerated gradient descent (Nesterov, 2005).

1.2. Outline of the Paper

We will start with some preliminaries in section 2, where we formally define the maxflow problem in a mathematical context. In section 3, we introduce the idea behind oblivious routing scheme and how to construct them. In section 4, we consider the problem in the context of convex optimization and discuss the runtime guarantees given an oblivious routing scheme. In section 5, we will present the recursive algorithm (Peng, 2016) that achieves $O(m\,\text{polylog}(n))$ overall runtiome. In section 6, we present our implementation on approximate maxflow algorithm given by (Sherman, 2013) with input G and α-congestion approximator R. In Appendix A, we provide pseudocode to many black-box algorithms to implement recursive algorithm in (Peng, 2016). We also refer to our github code in Appendix B.

2. Preliminaries

2.1. Notations and Definitions

We will use lower letters a to denote elements of a set or real numbers, bold lower letters $\mathbf{a}$ to denote vectors, capital letters A to denote sets, and bold capital letters $\mathbf{A}$ to denote matrices. We treat vectors $\mathbf{a} : V \to W$ (henceforth $\mathbf{a} \in W^V$) as mappings from its domain to its codomain, and we denote $w = \mathbf{a}(v)$ to indicate $\mathbf{a} : v \mapsto w$. We let $\mathbf{1}_S$ (we abbreviate $\mathbf{1}$ for $S = V$) to denote the indicator vector of a subset $S \subset V$.

We will assume all graphs of our interests are *undirected and capacitated*, which we denote G by $G = (V, E, \mathbf{c})$, where V is the *vertex* set of G, E is the *edge* set of G that consists of unordered pairs of elements from V, and the *capacity* $\mathbf{c} \in \mathbb{R}^E_{>0}$ is an embedding of the edges onto the positive reals. We will use the capacity vector $\mathbf{u} : \mathbb{N}^E$ in place of $\mathbf{c}$ to denote *integral-capacitated graphs* if the algorithm of our interest requires the capacity to be integers. We let $n := |V|$, $m := |E|$, and $u^* := \max_{e_1,e_2 \in E} \frac{\mathbf{u}(e_1)}{\mathbf{u}(e_2)}$ to denote the capacity ratio of an integral-capacitated graph.

We let $\mathbf{b} \in \mathbb{R}^V$ such that $\mathbf{1}^T\mathbf{b} = 0$ denote the *demand* vector with respect to a graph, $\mathbf{v} \in \mathbb{R}^V$ denote an real embedding of vertices (we will often refer to it as *vertex potential*), $\mathbf{B} \in \mathbb{R}^{m\times n}$ denote the incident matrix of an undirected graph constructed by choosing specific orientations of each edge. We let $\mathbf{f} \in \mathbb{R}^E$ denote a *flow* vector over all edges in G, and we let $C \in \mathbb{R}^{m\times m}$ (and U respectively) denote the diagonal matrix whose diagonal entry are capacities of edges (i.e., $(\mathbf{Cf})(e) = \mathbf{c}(e)\mathbf{f}(e)$). We say a demand is *feasible* if there exists $\mathbf{f} \in \mathbb{R}^E$ such that $\mathbf{B}^T\mathbf{f} = b$ and $\|\mathbf{C}^{-1}\mathbf{f}\|_\infty \le 1$.

We often refer to a vertex cut of a graph. We say that $S \subset V$ is a vertex cut of $G = (V, E, \mathbf{c})$. Given any demand $\mathbf{b}$, we define the *total excess* of S to be $b_S := \sum_{v\in S} \mathbf{b}(v)$, and we define the *capacity of the cut* S to be $c_S := \sum_{e\in\partial S} \mathbf{c}(e)$, where $\partial S := \{e = (u, v) : u \in S, v \in V - S\}$.

Given any two real numbers $x, y \in \mathbb{R}$, we say that x α-approximates y, denoted as $x \approx_\alpha y$, if $x \le y \le \alpha x$.

2.2. Formalizing Maxflow Problem

We now formalize the maximum flow problem with respect to general demand $\mathbf{b}$ where $\mathbf{1}^T\mathbf{b} = 0$. Given $G = (V, E, \mathbf{c})$, a source $s \in V$, and a target $t \in V$, the original maximum s-t flow problem consists of finding a maximal value $f^* \in \mathbb{R}$ such that the demand $f^*\mathbf{d}$ where $\mathbf{d} = \mathbf{1}_{\{s\}} - \mathbf{1}_{\{t\}}$ is feasible as well as its corresponding flow vector $\mathbf{f}^*$. We now present a more general definition that encapsulate the essence of the s-t flow problem. This definition is an adaptation from Sherman that focuses on the congestion of each edge and can be formulated into an l_∞ norm optimization problem:

Definition 2.1. Given $G = (V, E, \mathbf{c})$, and a flow $\mathbf{f} \in \mathbb{R}^E$, the *congestion* of $\mathbf{f}$ on an edge $e \in E$ is given by $\text{cong}(e) := \mathbf{f}(e)/\mathbf{c}(e)$.

For a demand vector $\mathbf{b} \in \mathbb{R}^V$, the *minimum congested flow* is the vector $\mathbf{f}^*$ such that

$$\mathbf{f}^* = \text{argmin}_{\mathbf{f}\in\mathbb{R}^E} \|\mathbf{C}^{-1}\mathbf{f}\|_\infty \quad \text{subject to} \quad \mathbf{B}^T\mathbf{f} = \mathbf{b}.$$

We refer to the value of $\|\mathbf{C}^{-1}\mathbf{f}^*\|_\infty$ as $\text{opt}(\mathbf{b})$. We say that $\mathbf{b}$ is *feasible* if $\text{opt}(\mathbf{b}) \le 1$.

Notice that if we take $\mathbf{b} := \mathbf{1}_{\{s\}} - \mathbf{1}_{\{t\}}$, then $\text{opt}(\mathbf{b}) = 1/f^*$. This is because l_∞ norm is linear and $f^*\mathbf{b}$ is maximally feasible if and only if $\text{opt}(f^*\mathbf{b}) = F^*\text{opt}(\mathbf{b}) = 1$. Therefore, it is enough to find $\mathbf{f}^*$ with respect to $\text{opt}(\mathbf{b})$ and scale it by $1/\text{opt}(\mathbf{b})$ to get the maximal s-t flow. In section 4 we will present two ways to fit this optimization problem to fit an l_∞-norm gradient descent framework.

2.3. Primal-Dual Problem

It is often useful to consider a primal-dual perspective of the general-demand minimum congested flow problem (Sherman, 2013). By the max-flow-min-cut theorem, we know that the minimum congestion required to route a demand $\mathbf{b}$ in G equals the maximal congestion produced by any cut $S \subset V$. We will not formally prove this result, but a high-level understanding is that the congestion bottleneck of our optimal flow $\mathbf{f}^*$ occurs at a specific subset of edges $F \subset E$, so removing these bottleneck edges would give us a cut with maximal congestion. We formally state it below:

Theorem 2.2. *Given $G = (V, E, \mathbf{c})$, and a demand $\mathbf{b}$, the congestion of a cut $S \subset V$ with respect to b is given by b_S/c_S. Then,*

$$\min_{\mathbf{B}^T\mathbf{f}=\mathbf{b}} \|\mathbf{C}^{-1}\mathbf{f}\|_\infty = \max_{\emptyset\neq S\subset V} b_S/c_S.$$

Note that if we can obtain an α-approximation of $\max b_S/v_S$, then we would have an α approximation of opt($\mathbf{b}$). However, this is not sufficient if we want to find the actual flow vector $\mathbf{f}^*$ that routes the minimum congestion given the demand. Therefore, new sparsification techniques on graphs are needed to not only preserve the cut structure but also the flow structures. We will introduce these techniques in section 3.

3. Oblivious Routing Scheme

A meta problem solving trend tells us that problems are easier to solve objects whose structure is very simple. In the context of maxflow, the most structurally-simple object of our interest is a tree. Suppose $G = (V, E, \mathbf{c})$ is a tree. We observe that each edge $e \in E$ induces a unique vertex cut $S_e \subset V$ by removing e from G. Therefore, given a demand $\mathbf{b}$, the congestion over any edge $e \in E$ is given by congestion of its vertex cut $|b_{S_e}/c_{S_e}|$ (note that we can also pick $S'_e = V - S_e$ to be the induced vertex cut, but since $\mathbf{1}^T\mathbf{b} = 0$, we would have $|b_{S'_e}/c_{S'_e}| = |b_{S_e}/c_{S_e}|$). This gives us an $O(n)$ algorithm that returns exactly $\mathbf{f}^*$ when G is a tree. For generalized dense graph, the problem becomes whether we can instead route the demand $\mathbf{b}$ on some tree-like graph, where we can find the exact flow efficiently, and then embed this result back into our original graph G that produce a flow $\tilde{\mathbf{f}}^*$. Such flow $\tilde{\mathbf{f}}^*$ may not route $\mathbf{b}$ with minimal congestion, but it guarantees an α-approximation of opt($\mathbf{b}$). The object of our interest is called an *oblivious routing scheme* for a graph G, which we formally define below:

Definition 3.1. Given $G = (V, E, \mathbf{c})$, an α-competitive oblivious routing scheme is a mapping $R : \mathbb{R}^V \to \mathbb{R}^E$ such that given any demand $\mathbf{b}$, $R(\mathbf{b})$ is a flow on G that routes $\mathbf{b}$ and $\|\mathbf{C}^{-1}R(\mathbf{b})\| \leq \alpha\,\text{opt}(\mathbf{b})$. Here, α is called the *competitive ratio* of R.

A sequence of works (Räcke, 2008; Abraham et al., 2008; Chekuri et al., 2005) focus on proving the existence as well as finding an $O(\log n)$ oblivious routing scheme for general graphs. proves the existence of such mapping, and gives an algorithm that produces it in polynomial time. In this section, we introduces the approaches taken by Madry (Madry, 2010) and Räcke et al. (Räcke et al., 2014), which at the expense of losing some prediction accuracy, produces an oblivious routing scheme in nearly linear time.

3.1. Graph Decomposition using J-Trees

Madry's work involves finding a convex combination of t number of tree-like graphs where any feasible flow $\mathbf{f} \in \mathbb{R}^E$ in the original graph can be effectively routed in each decomposition, and flows in the decompositions can be routed approximately in the original graph. We start by defining embeddable graphs.

Definition 3.2. Given $G = (V, E, \mathbf{c})$ and $\bar{G} = (V, \bar{E}, \bar{\mathbf{c}})$ that share the same vertices, and a collection of demands $\{\mathbf{d}_e\}$, where $\mathbf{d}_e = \mathbf{c}(e)(\mathbf{1}_{\{u\}} - \mathbf{1}_{\{v\}})$ for $e = (u, v) \in E$. We say G is t-embeddable into $\bar{G}$ if there exists a corresponding sequence of flows $\{\mathbf{f}_e\}$, each routing $\mathbf{d}_e$, such that

$$\left\|\bar{\mathbf{C}}^{-1}\sum_{e\in E}|\mathbf{f}_e|\right\|_\infty \leq t.$$

We call this sequence of flows $F := \{\mathbf{f}_e\}$ a *t-embedding* of G into $\bar{G}$. If $t = 1$, we simply call it an *embedding*. Intuitively, this definition implies that all demands $\mathbf{d}_e$ between each edge of G with $\mathbf{c}(e)$ units can be routed concurrently on $\bar{G}$ with congestion t. Then, if we obtain a flow $\mathbf{f}$ on G with congestion α, we are guaranteed a feasibility of a flow $\bar{\mathbf{f}}$ on $\bar{G}$ that routes the same demand with congestion $t\alpha$. Ideally, we want t to be close to 1 to get better approximation results.

Madry finds an effective method to obtain a family of graphs $\mathcal{G}$, which he calls "j-trees" that themselves have simple structures and can be embedded into G efficiently. We first describe the structure of a j-tree:

Definition 3.3. Given $G = (V, E, \mathbf{c})$, a graph $H = (V, E_H, \mathbf{c}_H)$ is a *j-tree* of G if it consists of a subgraph H' of G induced by some vertex set $V'_H \subset V$, where $|V'_H| \leq j$, and a *forest* F on V whose each connected component has exactly one vertex in V'_H. We call H' the *core* of H, and the forest F the *envelope* of H. We let $\mathcal{G}_V[j]$ denote the family of all j-trees on the vertex set V.

Note that to route in a j-tree, we can simply collapse all demands on the envelopes to the vertices that are in the intersection with the core. This is because there is a unique path to route on the forest. Therefore, a j-tree acts as an efficient j-vertex sparsifier for maxflow problems. We state the main result and begin our discussion of the approaches:

Theorem 3.4. *(Madry, 2010) Given $G = (V, E, \mathbf{u})$, and $t \geq 1$, we can find in time $\tilde{O}(tm)$ a collection $\{(\lambda_i, G_i)\}$ of size t where each $G_i \in \mathcal{G}_V[\tilde{O}(m \log U/t)]$ and satisfies the following properties:*

1. $\sum_i \lambda_i = 1$.
2. *G is embeddable into each G_i.*
3. *There exists embeddings F_i of each G_i such that*

$$\left\|\mathbf{U}^{-1}\sum_i\sum_{e\in G_i}|\mathbf{f}_e|\right\|_\infty \leq \tilde{O}(\log n).$$

We call this an $\tilde{O}(\log n)$-quality, $\mathcal{G}_V[\tilde{O}(m \log U/t)]$-decomposition of G, where G is decomposed into a convex

combination of graphs $G_i \in \mathcal{G}_V[\tilde{O}(m \log U/t)]$. Notice that the last property guarantees that if a demand $\mathbf{b}$ can be routed on each G_i with constant maximal congestion, then it can be routed in G with $\tilde{O}(\log n)$ congestion. Madry proposed a fast algorithm with a logarithmic trade-off of $\tilde{O}(\log n)$ quality instead of $O(\log n)$. Also, we remark that Madry's decomposition depends on the capacity ratio U when the algorithm leverages a low average-stretch spanning tree finder using a U-dependant length function on G. However, this factor can be eliminated simply by adding an inverse capacity factor into the length function (see Sherman13). Therefore, we can actually replace $\mathbf{u}$ with $\mathbf{c}$ and obtain a family of $\tilde{O}(m/t)$-trees, where the dependance is only on m. We observe that as we allow t to decrease, we can actually obtain near linear time $\tilde{O}(m)$ in construction, while the size of the core approaches $\tilde{O}(m)$.

To get an $\tilde{O}(\log n)$-quality decomposition family of $\tilde{O}(m/t)$-trees, we have to start from 1-trees, which are just spanning trees of G, plus $\tilde{O}(m/t)$ many bypass edges. Then, we refine our construction by deleting as many bypass edges as possible to obtain a decomposition of graphs that are "almost" $\tilde{O}(m/t)$-trees. Finally, we make slight adjustment to each instance in our decomposition to obtain actuall $\tilde{O}(m/t)$-trees.

3.2. Spanning Trees with Bypass Edges

We start off by considering a family of graphs $\mathcal{H}[j]$, where each $H_{T,F} \in \mathcal{H}[j]$ are constructed by a spanning tree T and an edge subset $F \subset E$ where $|F| \leq j$. We now describe the construction of such $H_{T,F}$ (the graph generated by the spanning tree T and an edge subset $F \subset E_T$). The algorithm is provided in Appendix A.

Express $T = (V, E_t, \mathbf{c}_T)$. Recall that we discussed in the beginning of the section regarding a unique way to embed flows in G into T. For G to be embeddable into T, we want $\mathbf{c}_T(e) = |b_{S_e}/c_{S_e}|$ for all $e \in T$. However, we claim that by making "bypasses" in the tree we can avoid accumulating high capacity in each tree edge. Define the set $E[T](F) := \{e \in E : \partial S_e \cap F \neq \emptyset\}$, which gives us the edges in G that lies in disconnected components of an edge cut $E_T - F$. Finally, we let $H_{T,F} = (V, E_H, \mathbf{c}_H)$ be given by $E_H = E_T \cup E[T](F)$, where

$$\mathbf{c}_H(e) = \begin{cases} \mathbf{c}(e) & e \in E[T](F); \\ \mathbf{c}_T(e) & \text{otherwise.} \end{cases}$$

We want to construct a decomposition of G using t graphs from the family $\mathcal{H}[\tilde{O}(m/t)]$. Let $l : V^2 \to \mathbb{R}$ be a metric on G. Consider the *volume* $l(H)$ given by $l(H) := \sum_{e \in E_H} l(e)\mathbf{u}_H(e)$. Also, consider a function $\gamma_H : E \to \mathbb{R}$ given by $\gamma_H(e) = \mathbf{u}(e)/\mathbf{u}_H(e)$, which is the inverse of the congestion of the flow $\mathbf{f} = \mathbf{u}_H(e)(\mathbf{1}_{\{u\}} - \mathbf{1}_{\{v\}})$, where $e = (u, v)$. Intuitively, this is the inverse of congestion accrues from an identity embedding of H into G. We are particularly interested in the edges where the identity embedding has high congestion; that is, $\mathbf{u}_H(e) \gg \mathbf{u}(e)$. For $H_{T,F}$ to be a good candidate, we want it to contain more edges with high capacity in $H(T, F)$. Consider the following colllection of edges $\kappa(H) := \{e \in E : \gamma_H(e) \leq 2 \min_{e' \in E} \gamma_H(e')\}$. We formalize this in a theorem that is implicitly proved by Räcke (Räcke, 2008):

Theorem 3.5. *If there is an $\alpha \geq \log m$ and a family $\mathcal{G}$ such that for all metric l on G, we can find in $\tilde{O}(m)$ time a subgraph $H \in \mathcal{G}$ such that*

1. *$l(H) \leq \alpha l(G)$;*
2. *G is embeddable into H;*
3. *$|\kappa(H)| \geq 4\alpha m/t$;*

then, we can compute in $\tilde{O}(tm)$ tima a 2α-quality decomposition of G into t many graphs in $\mathcal{G}$.

We leverage this theorem on $\mathcal{H}[\tilde{O}(m/t)]$. The goal is to show that a graph with the above properties can be found in $\tilde{O}(m)$ time. We find such graph using low average-stretch spanning trees. Fixing a metric l, we define the *stretch* of an edge $e = (u, v) \in V$ on a spanning tree T to be the ratio of the distance of its endpoints on T to the distance on G:

$$\text{stretch}_T^l(e) := \frac{d_T^l(u, v)}{d_G^l(u, v)}.$$

We observe that $\text{stretch}_T^l(e) \geq 1$ for all $e \in E$. A low average-stretch spanning tree is a spanning tree whose average distance between two leafs are small. Intuitively, this prevents an edge from accumulating too much congestion from the cut it induces. A theorem from Abraham et al. (Abraham et al., 2008) states that there exists an algorithm that takes in any metric l and returns in $\tilde{O}(m)$ time a spanning tree T such that $\frac{1}{m}\sum_{e \in E} \text{stretch}_T^l(e) = \tilde{O}(\log n)$. This algorithm allows us to bound the volume of a capacitated low-average spanning tree by a factor of $\tilde{O}(\log n)$ (recall that in order for T to be embeddable, its capacity is fixed by the cut congestion).

We summarize our result so far: we are able to find a spanning tree T in $\tilde{O}(m)$ time that satisfies condition 1 and 2 of Theorem 3.5. However, it is possible that T is a bad candidate for $H_{T,F}$, because we can easily have $\kappa(T) = \{e\}$, where all the congestions of the network accumulates on a single edge e. Therefore, our next goal is to find $F \subset E$ such that $H(T, F)$ is the desired graph. In fact, from $1 \geq \gamma_T(e) \geq 1/mU$ we see that there exists at most $\log mU$ many values of $\gamma_T(e)$ where each value differ by a factor of 2 from each other. We will leveage this fact to find $O(m \log U/t)$ edges in F that will give us the

third condition for Theorem 3.5. We sort the edges in T by ascending order of $\gamma_T(e)$, and Madry showed that we need to add at most $\tilde{O}(m \log U/t)$ many edges to F to satisfy condition 3. We provide the algorithm and some intuition in Appendix A.

Finally, by Theorem 3.5, we have obtained in $\tilde{O}(tm)$ time an $\tilde{O}(\log n)$-quality $\mathcal{H}[\tilde{O}(m \log U/t)]$-decomposition of G of size t. We remark that the approximation quality remains at $\tilde{O}(\log n)$ because we only lose a constant factor going from T to $H_{T,F}$. To conclude, we now have a collection $\{(\lambda_i, H_i)\}$ of t graphs from the family $\mathcal{H}[\tilde{O}(m/t)]$, where we know that given a demand $\mathbf{d}$ and if we can route on all H_i's with congestion α, then we can route $\mathbf{b}$ on G with congestion $\tilde{O}(\alpha \log n)$. However, the structure of the family $H[\tilde{O}(m/t)]$ is still not sparse and simple enough: the construction allows too many bypass edges to be added to the graph, so the graph remains dense. We need to convert this decomposition into a better family, which Madry calls "almost-j-trees."

3.3. From Bypass Edges to Almost-J-Tree

Our next goal is to delete as many bypass edges as possible while only incurring constant multiplicative congestion trade-offs. The ultimate goal is to have a family of j-trees, but it requires us to take an intermediate step. Our next step acts like an edge sparsifier.

We call a graph G' an *almost-j-tree* if it is a union of a tree and an arbitrary graph on at most j vertices. Clearly, j-trees are almost-j-trees. Also, the flow structure on $H_{T,F} \in \mathcal{H}[j]$ resembles an almost-j-tree $G'(T, F)$ generated by the same T and F. We find $G'(T, F)$ by projecting edges from $E[T](F)$ (these are exactly the bypass edges when we remove F from T) to T. We state and give a high-level proof sketch of the following lemma:

Lemma 3.6. *For any spanning tree $T = (V, E_T)$ of G, and $F \subset E_T$, we can find in $\tilde{O}(m)$ time an almost-$O(|F|)$-tree $G'_{T,F}$ such that the graph $H_{T,F}$ is embeddabe into $G'_{T,F}$ and $G'_{T,F}$ is efficiently 3-embeddable into $H_{T,F}$.*

Consider an edge $e = (u, v') \in E[T](F)$. Let $P(u, v') \subset T$ be the unique path subgraph in the original spanning tree T that connects u and v'. Presume that we start a walk from u to v' via $P(u, v')$. Let $v^1(e)$ (resp. $v^2(e)$) be the vertex that the walk first (resp. last) intersects endpoints of edges in F (we remark that $(v^1(e), v^2(e))$ is not necessarily an edge in F, because it's possible that $P(u, v')$ intersects with F by multiple edges).

We now project all bypass edges e to $G'_{T,F}$. The edge set of $G'_{T,F}$ consists of untouched edges in the original spanning tree $E_T - F$, and edges $f = (w, w')$ such that there exists an edge $e \in E[T](F)$ where $(w, w') = (v^1(e), v^2(e))$. We denote the set of all such edges f as F', and we let $\text{Proj}(f)$ to be the set of all such $e \in E[T](F)$ with $(w, w') = (v^1(e), v^2(e))$. Now, we set the capacity $\mathbf{c}'$ to be the following:

$$c'(e) = \begin{cases} 2\mathbf{c}_T(e) & e \in E_T \setminus F; \\ \sum_{e' \in \text{Proj}(e)} \mathbf{c}_H(e) & \text{otherwise (if } e \text{ is projected).} \end{cases}$$

We remind the readers that $\mathbf{c}_T(e) = |b_{S_e}/c_{S_e}|$, and $\mathbf{c}_H(e)$ is given by $H_{T,F} = (V, E_H, \mathbf{c}_H)$ from the decomposition. We claim that to find these projections, one only need to first divide $E_T - F$ into connected components and look at which components do the edges $e \in E[T](F)$ connect, so the total running time of computing $\mathbf{c}'$ should be dominated by the running time of computing $H_{T,F}$, which is $\tilde{O}(m)$. We provide the algorithmic details and how to embed $G'_{T,F}$ into $H_{T,F}$ and vice versa in Appendix A.

Finally, we set $G'_{T,F} = (V, (E_T - F) \cup F', \mathbf{c}')$. We remark that for all $f = (w, w') \in F'$, there exists some $f_1, f_2 \in F$ such that $w \in f_1$ and $w' \in f_2$, so the subgraph induced by F' has at most $2|F|$ vertices. Hence, $G'_{T,F}$ is an almost-$2|F|$-tree. This tells us that we can obtain in $\tilde{O}(tm)$ time an $\tilde{O}(\log n)$-quality decomposition $(\{\lambda_i, G'_i\})$ of almost-$\tilde{O}(m/t)$-tree, where λ_i are inherited from the decomposition $(\{\lambda_i, H_i\})$. Notice that the embeddings from G'_i to H_i are constant, so we don't incur more quality loss by performing this conversion.

3.4. From Almost-J-Tree to J-Tree

It remains for us to convert the almost-j-trees to actual j-trees. We first observe that the structure of an almost-j-tree and a j-tree differs only in the sense that a j-tree is a graph on j vertices after pruning then entire envelope, whereas we have no information on the size of an almost-j-tree if we prune its outgoing branches. In worst case scenario, we can pick T to be a path of n-vertices and project edges to the two endpoints of T. The resulting graph is a cycle, which clearly is an almost-1-tree, but it would be considered as an n-tree with the entire cycle being the core. Our goal therefore becomes first choosing a candidate for the core, and then move the bypass edges that are far away from the core closer towards it. We show that we can do this pruning efficiently so that it doesn't incur further runtime.

Lemma 3.7. *Let $G' = (V', E', \mathbf{u}')$ be an almost-j-tree. We can obtain in $\tilde{O}(|E'|)$ time an $O(j)$-tree $\bar{G} = (V' \bar{E}, \bar{\mathbf{u}})$ such that G' is embeddable into $\bar{G}$ and $\bar{G}$ is 3-embeddable into G'.*

We will show in Appendix A that such tree can be computed efficiently by first eliminating all degree-1 vertices in G', and then find a collection of edge-disjoint paths $P_1, \ldots, P_k$ that covers all degree-2 vertices (a high level understanding is that the degree-2 vertices must be a part of some path, which proves existence. The correctness can be proved by

picking a degree-2 vertex as a starting point and doing DFS). One can show that the endpoints of these paths must be vertices of degrees higher than 2. Finally, for each path P_i, we find the edge e_i with the lowest capacity, which are the bottleneck edges in the path, and we relocate them to the endpoints of P_i. The resulting graph (with degree-1 vertices added back) must be a $3j-2$-tree. We provide the full algorithm in Appendix A.

Finally, combining theorem 3.5, lemma 3.6, and lemma 3.7 proves theorem 3.4.

We remark that the approach taken by Mad10 is analogous to first performing an edge sparsification (section 3.2, 3.3) and a subsequence vertex sparsification (section 3.4). Additionally, since the guarantee of the decomposition is a convex combination of graphs, we can treat this combination as assigning a probability distribution to each graph in the collection, so a monte-carlo sampling algorithm can allow us to achieve, under certain probabilistic guarantees, a high-quality j-tree sparsification of the original graph withhout explicitly finding the entire collection. Then, we recurse on the core of these trees to finally narrow down to a small enough graph that we can route the demands exactly. We state the theorem in Mad10 below:

Theorem 3.8. *For any $0 \le l \le 1$, integral $k \ge 1$, and any $G = (V, E, \mathbf{u})$, we can find in $\tilde{O}(m + 2^k n^{2+\frac{1-l}{2^k-1}})$ time a collection of $(2^{k+1}\log n)$ n^l-trees $\{G_i\}_i$ that $(\log^{(1+o(1))k} n)$-preserves the cuts of G with high probability.*

Madry's algorithm outputs an oblivious routing scheme that solves the minimum cut problem. A crucial implication of this result is that an $\tilde{O}(\log n)$-competitive oblivious routing scheme can be constructed efficiently, where the runtime bottleneck appears in theorem 3.5 and the size of t to compensate vertex sparsity for runtime. A series of subsequent maxflow results (She13, KLOS14) leverage this algorithm as a subroutine to compute either an oblivious routing scheme or a congestion approximator. We will discuss one application in section 4.

3.5. Single Hierarchical Tree Decomposition

After Madry 10 and the subsequent works that succesfully accelerated the runtime of maxflow algorithm to nearly linear, Räcke et al (Räcke et al., 2014). discovered an even faster algorithm to get $O(\log^4 n)$-competitive oblivious routing schemes by a single hierarichical tree of total edge size $O(m \log^4 n)$ where G is broken down into clusters, subclusters, and eventually single vertices via recursion. At a high level, Räcke et al. separates a cluster S of the graph G into subclusters $S_1, \ldots, S_k$, with inter-cluster edges F such that for any two edge $f \in F$, we can route demands from f uniformly to all edges $F - \{f\}$ using only intracluster edges and edges in $F - \{f\}$. Such tree decomposition has $\log n$ height, and Räcke et al. finds good candidates of subclusters by performing an approximate maxflow on S. Therefore, his time dependancy is dominated by the runtime of the fastest approximate maxflow routing. We will see in section 5 how Peng (Peng, 2016) combines the approach of Räcke et al. (Räcke et al., 2014) and Sherman's l_∞ (section 4) descent (Sherman, 2013) to yield an $\tilde{O}(m \,\text{polylog}(n))$ approximate maxflow problem.

4. l_∞ Gradient Descent

There has been comprehensive literature on the convergence guarantees of the l_2 gradient descent method on convex objective functions with Lipschitz continuous gradients. A natural question then arises from the perspective of maxflow problem: Can we utilize existing analytical tools on l_2 gradient descent to solve for a seemingly cominatorial problem. Two results from Sherman (Sherman, 2013) and Kelner et al. (Kelner et al., 2013) provide different approaches, yet similar in the spirit of l_∞ minimization, on how to relax maxflow to an unconstrained l_∞ norm gradient descent problem. We refer to Sherman for the approach we presents.

4.1. Softmax Relaxation and Lipschitz Continuity

To connect problem 2.1 to an unconstrained optimization problem where we can apply the gradient descent method, Sherman proposes that if we have an operator $\mathbf{R}$ such that given any demand $\mathbf{b}$ on the graph $G = (V, E, \mathbf{c})$, $\|\mathbf{Rb}\|_\infty \approx_\alpha \text{opt}(\mathbf{b})$, then there is a very effective way to leverage this congestion approximator $\mathbf{R}$ to convert problem 2.1 to minimize the potential function below:

$$\min \|\mathbf{C}^{-1}\mathbf{f}\|_\infty + 2\alpha\|\mathbf{R}(\mathbf{b} - \mathbf{B}^T\mathbf{f})\|_\infty.$$

At a high level, for any flow $\mathbf{f}$ that routes some demand $\tilde{\mathbf{b}}$, the first term of the potential function accounts for the maximum congestion of sending $\mathbf{f}$, and the second term provides an overestimation of $\text{cong}(\mathbf{b} - \tilde{\mathbf{b}})$; that is, the congestion of routing the residual demand. We observe that the optimizer $\mathbf{f}^*$ may not route exactly the demand $\mathbf{b}$, but since the second term overestimates the residual congestion by at least a factor of two, we are actually imposing more penalty on residues than on congestion. In section 4.2, we introduce an iterative algorithm that utilizes this feature of the potential function.

Our focus then becomes finding an optimizer $\mathbf{f}^*$ for this potential function. However, the l_∞ norm is not smooth. In order to use the gradient descent method, we need a smooth alternative that approximate well the behavior of l_∞. We consider the following symmetric softmax function. For all

$\mathbf{x} \in \mathbb{R}^n$

$$\text{lmax}(\mathbf{x}) = \log\left(\sum_{i=1}^{n} e^{x(i)} + e^{-x(i)}\right).$$

We provide the following two properties of lmax (for full proof, see (Kelner et al., 2013)). For all $\mathbf{x}, \mathbf{y} \in \mathbb{R}$:

$$\begin{cases} \|\mathbf{x}\|_\infty \le \text{lmax}(\mathbf{x}) \le \|\mathbf{x}\|_\infty + \log(2n); \\ \|\nabla\text{lmax}(\mathbf{x}) - \nabla\text{lmax}(\mathbf{y})\|_1 \le \|\mathbf{x} - \mathbf{y}\|_\infty. \end{cases}$$

The first inequality tells us that lmax-norm provides an additive $\log 2n$-factor approximation of the l_∞ norm, which is much better than the l_2-norm multiplicative $\sqrt{m}$-factor approximation. The second inequality tells us that the gradient of the symmetric soft max function is Lipschitz continuous with respect to its dual norm (l_1 in this case) and has Lipschitz constant 1. This allows us to develop an l_∞ gradient descent scheme with provable convergence guarantees. We remark that the fastest increasing direction of a vector under l_∞-norm differs from the direction in l_2-norm, which is simply the normalized version of the original vector. Therefore, instead of straightforwardly using $\nabla\text{lmax}(\mathbf{x})$ as our gradient, we use the following fastest increasing direction (Kelner et al., 2013):

$$\nabla\text{lmax}(\mathbf{x})^\#(e) = \text{sgn}(\nabla\text{lmax}(\mathbf{x})(e))\,\|\nabla\text{lmax}(\mathbf{x})\|_1$$

Therefore, our algorithm aims to optimize the following smoothed version of the objective:

$$\Phi(\mathbf{f}) = \text{lmax}(\mathbf{C}^{-1}\mathbf{f}) + \text{lmax}(2\alpha\mathbf{R}(\mathbf{b} - \mathbf{B}^T\mathbf{f})).$$

One can show that the Lipschitz constant of $\Phi(\mathbf{f})$ equals $\frac{1}{1+4\alpha^2}$, where $\mathbf{R}$ is an α-competitive congestion approximator. From the result in section 3.4 and 3.5 we can assume $\alpha = \tilde{O}(\log n)$. In section 6 we implement the algorithm to check for correctness using $\alpha = O(m)$.

The following fact along with theorem 4.2 to ensure that the output of the optimizer of Φ returns an $(1+\varepsilon)$-approximation of $\text{opt}(\mathbf{b})$:

To connect this expediency to approximation of maxflow value, we simultaneously maximize the objective for the dual problem. The high level intuition is that Φ implicitly also maximizes a vertex cut, so when we minimize Φ we also obtain a high-quality vertex cut, which squeeze the optimal answer in between. We formalize it with the following:

Let $\mathbf{v} \in \mathbb{R}^V$ be a vertex embedding. We say that a *threshold cut* of $\mathbf{v}$ is given by a threshold $\mu \in \mathbb{R}$ and a cut $S := \{u \in V : \mathbf{v}(u) < \mu\}$. We make the following claim:

Proposition 4.1. *Given any $\mathbf{v} \in \mathbb{R}^V$ such that $\|\mathbf{C}\mathbf{B}\mathbf{v}\|_1 \le 1$, then there exists a threshold cut S of $\mathbf{v}$ such that $b_S/c_S \ge \mathbf{b}^T\mathbf{v}$.*

We now refer to a theorem and its corresponding algorithm from Sherman (Sherman, 2013) that outputs both a flow and a cut to sandwich the optimal value.

Theorem 4.2. *Given $G = (V, E, \mathbf{c})$. There is an algorithm* ALMOSTROUTE *that, given a demand $\mathbf{b}$ and error parameter $\varepsilon \le 1/2$, performs $\tilde{O}(\alpha^2\varepsilon^{-3}\log^2 n)$ [1] iterations and returns a flow $\mathbf{f}$ and cut S with*

$$\|\mathbf{C}^{-1}\mathbf{f}\|_\infty + 2\alpha\|\mathbf{R}(\mathbf{b} - B^T\mathbf{f})\|_\infty \le (1+\varepsilon)\frac{b_S}{c_S}.$$

Each iteration requires $O(m)$ time plus a multiplication by $\mathbf{R}$ and $\mathbf{R}^T$.

Note that if the output $\mathbf{f}$ routes exactly $\mathbf{b}$, then we would have $\|\mathbf{C}^{-1}\mathbf{f}\|_\infty \le (1+\varepsilon)\frac{b_S}{c_S}$. By proposition 4.1, $\mathbf{f}$ is an $(1+\varepsilon)$-approximation of $\text{opt}(\mathbf{b})$.

4.2. From Additive Error to Multiplicative Error

To leverage the gradient descent method, we use the fact that lmax is a $\log 2n$ additive approximator of l_∞ and a scaling manipulation to convert this additive error into multiplicative error, eventually yielding theorem 4.2. Let $\mathbf{b}$ be the input demand, and let $\mathbf{R}$ be an α-competitive congestion approximator. The algorithm ALMOSTFLOW is given in Appendix A (see algorithm A.3).

At a high level, the algorithm maintains the potential Φ above $O(\log n/\varepsilon)$ to ensure that when it terminates, the implicit additive $\log(2n)$ error in the lmax norm becomes multiplicative $1+\varepsilon$. For example, if $\frac{b_S}{c_S} \ge \Phi(\mathbf{f}) - O(\log n)$, then maintaining $\Phi(\mathbf{f})$ above $O(\log n/\varepsilon)$ gives us $\frac{b_S}{c_S} \ge \Phi(\mathbf{f})(1 - O(\varepsilon))$, which yields roughly $1+\varepsilon$ multiplicative error. Then, since lmax is an overestimation of l_∞-norm, we naturally would have the desired result.

Sherman proceeds to discuss how to obtain an $\tilde{O}(\log n)$-competitive congestion approximator $\mathbf{R}$ from j-trees. However, Peng and Räcke et al. (Peng, 2016; Räcke et al., 2014) later remark that using the hierarchical decomposition tree, one can get an $\tilde{O}(\log n)$-competitive $\mathbf{R}$ by performing maxflow on graph of edge size $o(m \log n)$. This is not a significant speed up by its own, but with the ultrasparsifier by Spielman & Teng (Spielman & Teng, 2014), Peng is able to first sparsify the original graph so that its hierarchical decomposition tree is a graph of roughly $m/2$ edges. We will tough upon the highlights of this recursive approach in section 5.

4.3. Handling Residual Demand

As stated in section 4.1, the output of ALMOSTROUTE may not route exactly demand $\mathbf{b}$. However, this can be resolved

[1] Sherman proposed a better iteration bound $O(\alpha\varepsilon^{-2}\log n)$, which he claimed achievable using the accelerated gradient method by Nesterov (Nesterov, 2005)

by iteratively calling ALMOSTROUTE on the residual demands of the previous iteraton until the residual demand is small enough for us to route with a greedy algorithm. We notice that the $\phi(\mathbf{f})$ penalizes the residual term by a factor of 2, so if we call ALMOSTROUTE $\log m$ times, the excess congestion from routing the residue will be full compensated by the congestion term of all previous iterations. For the algorithm, we refer to Appendix A.

To conclude, Sherman provides an algorithm that runs in $\tilde{O}(m\alpha^2\varepsilon^{-3}\log^2 n)$ time (hiding $\log\alpha$ factors) provided that we have an α-competitive congeston approximator. We know from Räcke et al. that we can obtain an $O(\log n)$-quality congestion approximator by calling maxflow on graphs whose total edge size is $O(m\log^4 n)$.

5. Recursive Approach to $O(m\,\textbf{polylog}(n))$

5.1. Obstacles in Naive Recursive Approach

Previous works (Madry, 2010; Sherman, 2013; Räcke et al., 2014) have provided effective methods in edge and vertex sparsifications of dense graphs G that preserves the flow and cut structures in nearly linear time. Also, in section 4 we proposed an $\tilde{O}(m\alpha\varepsilon^{-3}\log n)$ algorithm that solves maxflow given an α-quality congestion approximator, which can be in turn found using maxflow on a collection of graphs with edge sum $O(m\log^4 n)$. This creates an issue: if we naively recurse on the two algroithms given by Sherman and Räcke et al., the runtime of the first call is unbounded. To be precise, let $\mathcal{T}(m)$ denote the runtime of maxflow with m edges. Then, the recursion gives us

$$\mathcal{T}(m) \le \tilde{O}(m\alpha^2\varepsilon^{-3}\log^2 n) + \mathcal{T}(m\log^4 n),$$

where the first term incurs from runtime of Sherman, and the second incurs from calling maxflow on subclusters of G.

5.2. A Better Logarithmic Dependance

To bound the recursion, we want the second term to look like $\mathcal{T}(m/2)$. To accomplish this, Peng (Peng, 2016) uses an edge sparsifier and a vertex sparsifier (in the spirit of section 3.3 and 3.4) to achieve the $O(\log^4 n)$ multiplicative reduction of $G=(V,E,\mathbf{c})$ to some $G'=(V',E',\mathbf{c}')$ where $|E'|\sim\frac{m}{\log^4 n}|E|$. The runtime of edge sparsifier is given by a result of (Koutis et al., 2014; Abraham & Neiman, 2012):

Theorem 5.1. *There is a routine* ULTRA-SPARSIFY *that takes $G=(V,E,\mathbf{c})$ and parameter $\kappa>1$, returns in $O(m\log n\log\log n)$ time a graph $H=(V,E_H,\mathbf{c}_H)$ with $n-1+O(m\log^2 n\log\log n/\kappa)$ edges such that w.h.p., $c_S\approx_\kappa c'_S$ for all cuts $S\subset V$.*

If we set $\kappa=\Omega(\log^6 n\log\log n)$, then we would have $|E'|\sim\frac{m}{\log^4 n}|E|$. In addition, we sparsify the vertices as in lemma 3.6, whose runtime is dominated by theorem 5.1 but a trade-off of $O(\kappa\alpha)$ quality congestion approximator instead of $O(\alpha)$. Finally, we use $\alpha=\log^4 n$ in section 3.5 and $\varepsilon=1/\Theta(\log^3 n)$. Then, theorem 4.2 guarantees the runtime of computing maxflow to be

$$\begin{aligned}O(\alpha^2\log^2 n\varepsilon^{-3}) &= O(\log^{22} n\log^2\log n\varepsilon^{-3})\\ &= O(\log^{31} n\log^2\log n).\end{aligned}$$

Notice that multiplying by $\mathbf{R}$ and $\mathbf{R}^T$ adds an $O(m\log^6 n)$ overhead, but this is dominated by the runtime of computing a single iteration of maxflow using Sherman and Räcke et al. Therefore, the final runtime is bounded by

$$\mathcal{T}(m)\le O(m\log^{31} n\log^2\log n)+\mathcal{T}(m/2).$$

This is achieved by setting $\varepsilon=1/\Theta(\log^3 n)$. We do remark that our computation achieves a lower logarithmic dependance than Peng's result due to a magical appearing additional $\log^{10} n$ factor in Lemma 3.2 of (Peng, 2016). In addition, we emphasize that if we can improve Sherman's algorithm with accelerated gradient method, we would then have the runtime of computing maxflow to be

$$O(m\alpha\log n\varepsilon^{-2}) = O(m\log^{17} n\log\log n).$$

The total runtime of recursive calls would be bounded by

$$\mathcal{T}(m)=\tilde{O}(m\log^{17} n),$$

where $\tilde{O}$ hides $\log\log n$ factors. We provide the algorithm below

Algorithm 2 APPROXIMATORMAXFLOW (Peng, 2016)

1: Set $\kappa\leftarrow C\log^6 n\log\log n$ for absolute constant C.
2: $G'\leftarrow$ ULTRASPARSIFYANDREDUCE(G,κ)
3: $R_{G'}\leftarrow$ CONGESTIONAPPROXIMATOR(G'), which in turn makes recursive calls to RECURSIVEAPPROXMAXFLOW.
4: $R_G\leftarrow$ CONVERT$(G,G',R_{G'})$
5: Return APPROXIMATORMAXFLOW(G,R_G,ε).

6. Implementation

6.1. Example

Due to the incompleteness of certain steps in the algorithms presented in (Räcke et al., 2014) and (Madry, 2010), we have included only the pseudocode in Appendix A. For a practical implementation, we have chosen to utilize the algorithm described as APPROXIMATOR-MAXFLOW in (Sherman, 2013). The implementation was conducted

in Python using the NetworkX library. The efficacy of the algorithm was evaluated using the following test case:

Example 1.3 (Sherman, 2013) Let T be a maximum weight spanning tree in G, and let R be the $n-1 \times n$ matrix with a row for each edge in T, and

$$(Rb)_e = \frac{b_S}{c_S}$$

where $(S, V \setminus S)$ is the cut in G induced by removing e from T. Then, R is a m-congestion-approximator.

Proposition 6.1. *Define* $A \in \mathbb{R}^{n-1 \times n}$,

$$A_{e,i} = \begin{cases} 1 & \text{if vertex } i \in S_e, \\ 0 & \text{if vertex } i \notin S_e. \end{cases}$$

Then the m-congestion-approximator R in Example 1.3 is $R = C^{-1}A$.

Proof. Given that $R = C^{-1}A$, the congestion on edge e due to the flow vector b is given by the e-th entry of the vector Rb. This can be expanded as follows:

$$\begin{aligned} (Rb)_e &= \sum_i C^{-1}_{e,e} A_{e,i} b \\ &= \sum_i C^{-1}_{e,e} (A_{e,i} b) \\ &= \sum_i C^{-1}_{e,e} b_s \\ &= \frac{b_s}{c_s} \end{aligned}$$

Since $A_{e,i} = 1$ if vertex i is in S_e and 0 otherwise, $A_{e,i}b$ simplifies to b_s and $\sum_i C^{-1}_{e,e} b_s$ simplifies to $\frac{b_s}{c_s}$, where $C_{e,e}$ is the capacity of edge e. Thus, we prove the proposition. □

To test the performance of the APPROXIMATOR-MAXFLOW algorithm, we construct a complete graph with five nodes, assign random capacities to the edges, and generate a demand vector with random values such that the sum of the demands equals zero. See Appendix B) for github link. We then induce a spanning tree from this graph and calculate the R matrix according to the formula given in Example 1.3. The APPROXIMATOR-MAXFLOW algorithm is applied to this scenario to evaluate its effectiveness in approximating the maximum flow. Set $\epsilon = 0.5$, $\alpha = m$, then

```
> approximator_maxflow(G, R, epsilon)
> Original demand: [-2  1 -2 -2  5]
> Demand sent by total flow: [-2.00000001
    0.99999999 -1.99999999 -1.99999999  5.]
> Approximated flow:
[-6.44771304e-01  2.17040920e-02
    2.17042940e-02 -1.39863709e+00
    6.13827026e-01 6.13826709e-01
    -8.72425052e-01  5.84381091e-08
    -1.36446893e+00 -1.36446893e+00]
```

From the result, we see that the demand by routing total flow is equal to the original demand. To see more clearly how the potential change with iterations in the algorithms, we can draw the potential and iteration graph in figure 1. From the graph, we can see that the potential decrease in quadratic shape, which means as iterations goes up, the congestion decrease.

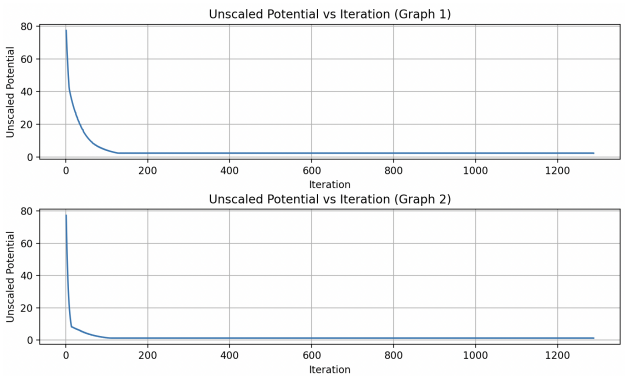

Figure 1. Unscaled potential vs Iteration

6.2. Obstacles

During implementation, one major obstacle we meet is the algorithm does not terminate with $\delta < \frac{\epsilon}{4}$ in algorithm 8. δ converge to some number higher than $\frac{\epsilon}{4}$, as we can see in the following graph

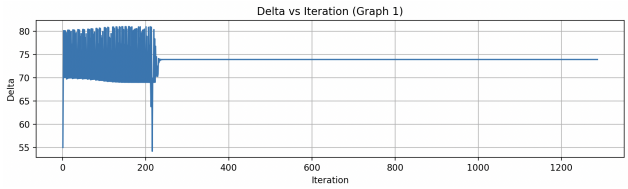

Figure 2. Delta vs Iteration

To resolve this issue, we add another bound to terminate the algorithm besides $\delta < \frac{\epsilon}{4}$. This problem mainly arouse from incompatible of capacity and demand, where the demand can be too large that exceeds the bound of capacity. To resolve this issue, we use the following lemma from (Sherman, 2013):

Lemma 2.5.(Sherman, 2013) *AlmostRoute* terminates after at most $\tilde{O}(\alpha^2\epsilon^{-3}\log n)$ iterations.

Therefore, using this lemma, we manually terminate the algorithm after $\tilde{O}(\alpha^2\epsilon^{-3}\log n)$ iterations even δ does not suffice the terminate requirement $\delta < \frac{\epsilon}{4}$. It is important to note that this solution is a workaround that allows the algorithm to proceed to the subsequent step without fulfilling the original termination condition. Although this may introduce discrepancies, Lemma 2.5 assures that the number of iterations is sufficient for an approximate solution within the desired error bounds.

References

Abraham, I. and Neiman, O. Using petal-decompositions to build a low stretch spanning tree. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pp. 395–406, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312455. doi: 10.1145/2213977.2214015. URL https://doi.org/10.1145/2213977.2214015.

Abraham, I., Bartal, Y., and Neiman, O. Nearly tight low stretch spanning trees. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pp. 781–790, 2008. doi: 10.1109/FOCS.2008.62.

Chekuri, C., Khanna, S., and Shepherd, F. B. Multicommodity flow, well-linked terminals, and routing problems. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pp. 183–192, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139608. doi: 10.1145/1060590.1060618. URL https://doi.org/10.1145/1060590.1060618.

Ford, L. R. and Fulkerson, D. R. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi: 10.4153/CJM-1956-045-5.

Kelner, J. A., Orecchia, L., Lee, Y. T., and Sidford, A. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. *CoRR*, abs/1304.2338, 2013. URL http://arxiv.org/abs/1304.2338.

Koutis, I., Miller, G. L., and Peng, R. Approaching optimality for solving sdd linear systems. *SIAM Journal on Computing*, 43(1):337–354, 2014. doi: 10.1137/110845914. URL https://doi.org/10.1137/110845914.

Madry, A. Fast approximation algorithms for cut-based problems in undirected graphs. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pp. 245–254, Los Alamitos, CA, USA, oct 2010. IEEE Computer Society. doi: 10.1109/FOCS.2010.30. URL https://doi.ieeecomputersociety.org/10.1109/FOCS.2010.30.

Nesterov, Y. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1):127–152, 2005. doi: 10.1007/s10107-004-0552-5. URL https://doi.org/10.1007/s10107-004-0552-5.

Peng, R. Approximate undirected maximum flows in o(mpolylog(n)) time. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pp. 1862–1867, USA, 2016. Society for Industrial and Applied Mathematics. ISBN 9781611974331.

Räcke, H., Shah, C., and Täubig, H. Computing cut-based hierarchical decompositions in almost linear time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pp. 227–238, USA, 2014. Society for Industrial and Applied Mathematics. ISBN 9781611973389.

Räcke, H. Optimal hierarchical decompositions for congestion minimization in networks. pp. 255–264, 05 2008. doi: 10.1145/1374376.1374415.

Sherman, J. Nearly maximum flows in nearly linear time. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pp. 263–269, 2013. doi: 10.1109/FOCS.2013.36.

Spielman, D. A. and Teng, S.-H. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM Journal on Matrix Analysis and Applications*, 35(3):835–885, 2014. doi: 10.1137/090771430. URL https://doi.org/10.1137/090771430.

Vishnoi, N. Lx=b. laplacian solvers and their algorithmic applications. *Foundations and Trends in Theoretical Computer Science*, 8, 01 2012. doi: 10.1561/0400000054.

A. Algorithm Pseudocodes

We are able to fill in some of the gaps of algorithmic details omitted in the work of (Räcke et al., 2014; Madry, 2010; Sherman, 2013; Peng, 2016).

A.1. Algorithm Pseudocode for Madry (Madry, 2010)

Algorithm 3 Compute Low Stretch Spanning Trees

Input: A graph $G = (V, E, u)$, a length function l' on G
Output: A low stretch spanning tree T
[Insert algorithm or method for computing a low stretch spanning tree here]
return T

Algorithm 4 Construct H_l

Input: A graph $G = (V, E, u)$, a length function l over G, a sparsity parameter t
Output: A subgraph H_l of G that satisfies the criteria of Theorem 5.2
$T \leftarrow$ Spanning tree of G with capacities u_T on edges
$G_{\text{bar}} \leftarrow$ Multigraph of G with vertex set V with each edge e in G_{bar} repeats $r(e) = 1 + \lfloor l(e) \cdot u(e) \cdot |E|/l(G) \rfloor$ times
$T_l, \alpha \leftarrow$ LowStretchTrees(G_{bar}, l') ▷ need to leverage ABN algorithm on multigraphs (this l' is normalized l))
$F_j \leftarrow$ Empty dictionary with $\lfloor \log(m) \cdot \max(u) \rfloor$ keys
for each e in $E(T_l)$ **do**
    $j \leftarrow \lfloor \log(m \cdot \max(u) \cdot \gamma_{T_l}(e)) \rfloor$
    $F_j[j]$.add(e)
end for
Sort $E(T_l)$ with respect to $\gamma_{T_l}(e)$
threshold $\leftarrow 4 \cdot (2\alpha + 1) \cdot m \cdot (\log(m) \cdot \max(u) + 1)/t$
$e^* \leftarrow E(T_l)[\lfloor \text{threshold} \rfloor]$
$j^* \leftarrow \lfloor \log(m \cdot \max(u) \cdot \gamma_{T_l}(e^*)) \rfloor$
for $j = 0$ to $j^* + 1$ **do**
    if $|F_j[j]| \geq 4 \cdot (2\alpha + 1) \cdot m/t$ **then**
        $j_{\text{bar}} \leftarrow j + 1$
        break
    end if
end for
$F \leftarrow$ List comprehension collecting $F_j[j]$ for $j \leq j_{\text{bar}}$
return $H(T, F)$

Algorithm 5 Compute t-Sparse Decomposition

Input: A graph $G = (V, E, u)$, a sparsity parameter t
Output: A t-sparse $(2\alpha, G_{\text{fam}})$-decomposition $\{\lambda[i], H[i]\}$
$G_{\text{decomp}} \leftarrow$ Apply Theorem 5.2 with G, t
return G_{decomp}

Algorithm 6 Convert to Almost-J-Tree

Input: a graph $G = (V, E, u)$, a graph H from family H_{fam} represented as $H(T, F)$
Output: an almost-j-tree $G(T, F)$

Create a new graph $G(T, F)$ with the same vertex set as G
Add all edges from $E(T)$ except those in F to $G(T, F)$
Assign a weight of $2u_T(e)$ to each edge in $G(T, F)$ that was in $E(T)$
Let V' be the set of pairs (v_1, v_2) where each v_i is an endpoint of an edge in F
for each pair (v_1, v_2) in V' **do**
    Add an edge between v_1 and v_2 to $G(T, F)$
    Assign a weight to this new edge equal to the sum of weights of edges in F connecting v_1 and v_2
end for
return $G(T, F)$

Algorithm 7 Convert to J-Tree

Input: a graph $G = (V, E, u)$, an almost-j-tree $G' = (V', E', u')$
Output: an $O(j)$-tree

$F_{\text{bar}} \leftarrow$ all degree 1 vertices in G' and their incident edge
G'.remove(F_{bar})
$W \leftarrow$ degree 2 vertices in G'
Identify edge-disjoint paths $p_1, \ldots, p_k$ in G' with endpoints $(v_1[i], v_2[i])$ in $V' \setminus W$
for $i = 1$ to k **do**
    $e_i \leftarrow \text{argmin}_{e \in p_i} u(e)$
    $p'_i \leftarrow p_i \setminus \{e_i\}$
    Assign capacities to each edge $f \in p'_i$ with $u(f) + u(e_i)$
    E_{bar}.add(p'_i), F_{bar}.add($p'_i \times W$)
    E_{bar}.add($(v_1[i], v_2[i])$) with capacity $u(e_i)$
end for
return $G_{\text{bar}} = (V' \setminus F_{\text{bar}}, E_{\text{bar}}, u_{\text{bar}})$

Algorithm 8 Find J-Tree Decomposition

Input: A graph $G = (V, E, u)$, a sparsity parameter t
Output: A convex combination of $O(\frac{m \log(m)}{t})$-trees

$G_{\text{decomp}} \leftarrow$ ComputeHDecomposition(G, t)
for each H in G_{decomp}[Tree] **do**
    $G' \leftarrow$ ConvertToJTree$(G,$ ConvertToAlmostJTree$(G, H))$
end for
return G_{decomp}

A.2. Algorithm Pseudocode for Räcke et al. (Räcke et al., 2014)

Algorithm 9 Partial Fractional Matching

Input: A set of edge candidates F, commodity matrix F_{bar}
Output: A new collection of edges $F \subseteq E[S]$ that satisfies some properties of Lemma 3.1 or Lemma 3.2

$u, \mu_{\text{bar}} \leftarrow$ RandProj(F_{bar}), RandProj(AvgFlow(F_{bar})) ▷ Random projection
$A_s, A_t, \eta \leftarrow$ SeparateST(F, u, μ_{bar})
$V' \leftarrow V' \cup \{s, t\}$
Assign capacity 2 for all edges in G'
Connect s to A_s with capacity 1
Connect t to A_t with capacity $\frac{1}{2}$
$G'_{st} \leftarrow G'$
$f \leftarrow$ ApproximateMaxFlow($G'_{st}, s, t, \frac{1}{\log^3(n)}$)
Scale f so that $f[x[A_s], s] \geq \frac{1}{2}$ becomes 1 ▷ Scaling of flow, feasibility path
$C' \leftarrow$ Decompose into partial fractional matchings ▷ TODO: Specify decomposition method
$C \leftarrow$ Corresponding edges of C' in G
return C

Algorithm 10 PartitionA Method

Input: a cluster $S \subseteq V$
Output: a list of subclusters $Z_1, \ldots, Z_z$

Use Lemma 3.1 and Lemma 3.2 recursively to obtain a flow
return Z

Algorithm 11 SeparateST Method

Input: a set of edge candidates A, projected capacity u, projected average μ_{bar}
Output: a source subset A_s, a target subset A_t, a separation value η ▷ satisfies Lemma 3.3

$L \leftarrow A[u_e < \mu_{\text{bar}}]$
$R \leftarrow A[u_e \geq \mu_{\text{bar}}]$
if $|L| > |R|$ **then**
    Swap L, R
    if $P(L) < \frac{1}{20}P(A)$ **then**
        $\eta \leftarrow \mu_{\text{bar}} - \frac{4P(L)}{|A|}$
        $A_t \leftarrow A[u_e \geq \eta]$
        $R' \leftarrow A[u_e \leq \mu_{\text{bar}} - \frac{6P(L)}{|A|}]$
        return $R'[\text{max } |A|/8 \text{ fraction}], A_t, \eta$
    end if
end if
if $P(L) \geq \frac{1}{20}P(A)$ **then**
    return $L[\text{max } |A|/8 \text{ fraction}], R, \mu_{\text{bar}}$
end if
$\eta \leftarrow \mu_{\text{bar}} + \frac{4P(L)}{|A|}$
$A_t \leftarrow A[u_e \leq \eta]$
$R' \leftarrow A[u_e \geq \mu_{\text{bar}} + \frac{6P(L)}{|A|}]$
return $R'[\text{max } |A|/8 \text{ fraction}], A_t, \eta$

Algorithm 12 Get Subclusters Method

Input: a cluster $S \subseteq V$
Output: a list of subclusters $S_1, \ldots, S_n$ that partition S

$G_S \leftarrow$ Induced subgraph $G[S]$
$B \leftarrow E(S, V \setminus S)$
$Z \leftarrow$ PartitionA(S) ▷ a list of subclusters generated by PartitionA
$L, R \leftarrow$ PartitionB(S, B, Z) ▷ two subclusters generated by PartitionB
$\tilde{S} \leftarrow$ Combine(Z, L, R) ▷ taking the intersection of Z and L, R
return $\tilde{S}$

Algorithm 13 Hierarchical Decomposition

Input: a cluster $S \subseteq V$
Output: a hierarchical tree decomposition T

$T \leftarrow$ Init(T, 0) ▷ initialize a tree with a root node
if $|C| == 1$ **then**
    return T ▷ return the tree if C consists of one vertex
end if
$S \leftarrow$ GetSubclusters(S) ▷ a list of subclusters
for each C in S **do**
    T.AddChildren(HierarchicalDecomposition(C))
end for
return T

A.3. Algorithm Pseudocode for Sherman (Sherman, 2013)

Algorithm 14 AlmostRoute (Sherman, 2013)

Input: demands b, error parameter ε
Output: flow f and potentials induced by $\nabla\phi(f)$

Initialize $f = 0$, scale b so that $2\alpha\|Rb\|_\infty = 16\varepsilon^{-1}\log(n)$.
repeat
  While $\phi(f) < 16\varepsilon^{-1}\log(n)$, scale f and b up by $17/16$.
  Set $\delta \leftarrow \|C\nabla\phi(f)\|_1$.
  if $\delta \geq \varepsilon/4$ **then**
    Set $f \leftarrow f - \frac{\delta}{1+4\alpha^2}\text{sgn}(\nabla\phi(f))_c e_c$
  else
    Terminate and output f together with the potentials induced by $\nabla\phi(f)$,
    after undoing any scaling.
  end if
until termination condition is met

Algorithm 15 ApproximatorMaxflow (Sherman, 2013)

Input: G, initial demand b, T maximal spanning tree of G
Output: $f_1 + \cdots + f_{T+1}$ and S_0

$b_0 \leftarrow b$
$(f_0, S_0) \leftarrow$ AlmostRoute(b_0, ε)
for $i = 1, \ldots, T$ where $T = \log(2m)$ **do**
  $b_i \leftarrow b_{i-1} - Bf_{i-1}$
  $(f_i, S_i) \leftarrow$ AlmostRoute$(b_i, \frac{1}{2})$
end for
$b_{T+1} \leftarrow b_t - Bf_t$
f_{T+1} be a flow routing b_{T+1} in a maximal spanning tree of G.

A.4. Algorithm Pseudocode for Peng (Peng, 2016)

B. Github Code

The implementation of the algorithm can be found in the following GitHub repository:

https://github.com/KevinGuo27/maxflow.git