Minimum Cuts in Directed Graphs via Partial Sparsification

Ruoxu Cen
Department of Computer Science
Duke University
ruoxu.cen@duke.edu

Jason Li
Simons Institute for Theory of Computing
UC Berkeley
jmli@cs.cmu.edu

Danupon Nanongkai
Department of Computer Science
University of Copenhagen
danupon@gmail.com

Debmalya Panigrahi
Department of Computer Science
Duke University
debmalya@cs.duke.edu

Thatchaphol Saranurak
Computer Science & Engineering Division
University of Michigan Ann Arbor
thsa@umich.edu

Kent Quanrud
Department of Computer Science
Purdue University
krq@purdue.edu

2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS) | 978-1-6654-2055-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/FOCS52979.2021.00113

***Abstract*—We give an algorithm to find a minimum cut in an edge-weighted directed graph with n vertices and m edges in $\tilde{O}(n \cdot \max\{m^{2/3}, n\})$ time. This improves on the 30 year old bound of $\tilde{O}(nm)$ obtained by Hao and Orlin for this problem. Using similar techniques, we also obtain $\tilde{O}(n^2/\epsilon^2)$-time $(1+\epsilon)$-approximation algorithms for both the minimum edge and minimum vertex cuts in directed graphs, for any fixed ϵ. Before our work, no $(1+\epsilon)$-approximation algorithm better than the exact runtime of $\tilde{O}(nm)$ is known for either problem.**

Our algorithms follow a two-step template. In the first step, we employ a *partial sparsification* of the input graph to preserve a critical subset of cut values approximately. In the second step, we design algorithms to find the (edge/vertex) mincut among the preserved cuts from the first step. For edge mincut, we give a new reduction to $\tilde{O}(\min\{n/m^{1/3}, \sqrt{n}\})$ calls of *any* maxflow subroutine, via packing arborescences in the sparsifier. For vertex mincut, we develop new local flow algorithms to identify small unbalanced cuts in the sparsified graph.

I. Introduction

The *minimum cut* (or mincut) problem is one of the most widely studied problems in graph algorithms. In (edge-)weighted[1] directed graphs (or *digraphs*), a mincut is a bipartition of the vertices into two non-empty sets $(S, V \setminus S)$ so that the total weight of edges from S to $V \setminus S$ is minimized. This problem can be solved by solving the *s-mincut* problem (also called rooted mincut), where for a given *root* vertex s, we want to find the minimum weight cut $(S, V \setminus S)$ such that $s \in S$. (We call such cuts minimum s-cuts or s-mincuts.) This is because the mincut can be computed as the minimum between two s-mincuts for an arbitrary vertex s: one with the original edge directions in the input digraph, and the other with the edge directions reversed.

This paper combines, and improves on, two independent manuscripts by Quanrud [26] and the other authors [2].

[1]We assume throughout that edge/vertex weights are polynomially bounded integers.

A simple algorithm for s-mincut (and thus mincut) on an m-edge, n-vertex digraph is to use $n-1$ maxflow calls to obtain the minimum $s-t$ cut for every vertex $t \neq s$ in the graph, and return the minimum among these. A beautiful result of Hao and Orlin [16] showed that these maxflow calls can be amortized to match the running time of a single maxflow call, provided one uses the *push-relabel maxflow algorithm* [15]. This leads to an overall running time of $\tilde{O}(mn)$. Since their work, better maxflow algorithms have been designed, but the amortization does not work for these algorithms. As a consequence, the Hao-Orlin bound remains the best known for the directed mincut problem almost 30 years after their work.

A. Our Results

In this paper, we can solve the s-mincut—thus the directed mincut problem—by essentially reducing it to $O(\sqrt{n})$ maxflow calls. At first glance, this is worse than the Hao-Orlin algorithm that only uses a single maxflow call. But crucially, while the Hao-Orlin algorithm is restricted to a specific maxflow subroutine and therefore cannot take advantage of faster, more recent maxflow algorithms, our new algorithm treats the maxflow subroutine as a black box, thereby allowing the use of *any* maxflow algorithm. Using state of the art maxflow algorithms that run in $\tilde{O}(m + n^{3/2})$ time [28], this already improves on the Hao-Orlin bound. Using some additional ideas, we further reduce to $O(\min\{n/m^{1/3}, \sqrt{n}\})$ maxflow calls, which yields our eventual running time of $\tilde{O}(nm^{2/3} + n^2)$:

Theorem I.1. *There is a randomized Monte Carlo algorithm that solves s-mincut (and therefore directed mincut) whp in $\tilde{O}(nm^{2/3} + n^2)$ time on an n-vertex, m-edge (edge-weighted) directed graph.*

In fact, our reduction in general implies a running time bound of $\tilde{O}(\min\{mk, nk^2\} + \frac{n}{k} \cdot F(m,n))$, where k is

2575-8454/21/$31.00 ©2021 IEEE
DOI 10.1109/FOCS52979.2021.00113

a parameter that we can choose and $F(m,n)$ is the time complexity of maxflow (see Theorem II.1).

Our techniques also yield fast approximations for the mincut problem in directed graphs. In particular, for any $\epsilon \in (0,1)$, we can find a $(1+\epsilon)$-approximate mincut in $\tilde{O}(n^2/\epsilon^2)$ time:

Theorem I.2. *For any $\epsilon \in (0,1)$, there is a randomized Monte Carlo algorithm that finds a $(1+\epsilon)$-approximate s-mincut (and therefore directed mincut) whp in $\tilde{O}(n^2/\epsilon^2)$ time on an n-vertex (edge-weighted) directed graph.*

Finally, we consider vertex-weighted digraphs. A *vertex cut* in a digraph is defined as a tri-partition of vertices into sets (L,X,R) such that there is no edge from L to R. (In other words, removing the vertices in X results in a digraph where the directed cut (L,R) is empty.) A *minimum* vertex cut (or vertex mincut) is a vertex cut (L,X,R) that minimizes the sum of weights of vertices in X. We give an algorithm to find a $(1+\epsilon)$-approximate vertex mincut in $\tilde{O}(n^2/\epsilon^2)$ time:

Theorem I.3. *For any $\epsilon \in (0,1)$, there is a randomized Monte Carlo algorithm that finds a $(1+\epsilon)$-approximate minimum vertex s-cut and the minimum global vertex cut whp in $\tilde{O}(n^2/\epsilon^2)$ time on an n-vertex (vertex-weighted) directed graph.*

To the best of our knowledge, before this work, the fastest algorithms for $(1+\epsilon)$-approximation of mincuts in edge or vertex weighted directed graphs were the respective exact algorithms themselves, which obtained a running time of $\tilde{O}(mn)$ [16], [17].Our approximation results establish a separation between the best exact and $(1+\epsilon)$-approximation algorithms for both edge and vertex mincut problems in directed graphs.

Our results are the first to break the $O(mn)$ barrier for directed mincut problems in general, weighted digraphs. For all values of m except when $m = n^{1+o(1)}$, this is immediate from the above theorems. If $m = n^{1+o(1)}$, we can also break the $O(mn)$ barrier by employing the recent $\tilde{O}\big(m^{1.5-1/328}\big)$-time max-flow algorithm of [12] to obtain $O\big(mn^{1-\Omega(1)}\big)$-time algorithms for all problems in Theorems I.1 to I.3.

Related Work: The directed (edge) mincut problem has been studied over several decades. Early work focused on unweighted graphs [6], [27] eventually resulting in an $O(mn)$-time algorithm due to Mansour and Schieber [21]. This was matched (up to log factors) in weighted graphs by Hao and Orlin [16], whose result we improve in this paper. For unweighted graphs (and graphs with small integer weights), the current record is a recent $\tilde{O}(n^2)$-time algorithm due to Chekuri and Quanrud [3]. A similar story has unfolded for directed vertex mincuts. Early work again focused on unweighted graphs [25], [6], [4], [11] until the work of Henzinger, Rao, and Gabow [17] who obtained an $\tilde{O}(mn)$-time algorithm for weighted graphs. The current best for directed vertex mincut in unweighted graphs is an $\tilde{O}(mn^{11/12+o(1)})$-time algorithm due to Li *et al.* [19]. Faster algorithms are known when the mincut size is small and for $(1+\epsilon)$-approximations in unweighted digraphs [24], [8], [3].

B. Our Techniques

Our results are obtained by solving the s-mincut problem. Let us consider the edge-weighted case. Gabow [9] obtained a running time of $\tilde{O}(m\lambda)$ for this problem (assuming integer weights), where λ is the size of an s-mincut. He did so via *arborescense packing*: Define an s-arborescense to be any spanning tree rooted at s with edges pointing toward the leaves. In $\tilde{O}(m\lambda)$ time, Gabow's algorithm computes λ s-arborescenses such that an edge e of weight $w(e)$ is contained in at most $w(e)$ arborescenses (this is called arborescense packing).[2] Gabow's algorithm is at least as fast as that of Hao and Orlin for unweighted simple graphs (since $\lambda \leq n-1$), but can be much worse for weighted (or multi) graphs. Nevertheless, Karger [18] gave an interesting approach to use arborescence packing for the mincut problem even with edge weights, *but only in undirected graphs*. Karger's algorithm had three main steps:

(a) sparsify the input graph G to H by random sampling of edges to reduce the mincut value in H to $O(\log n)$ while guaranteeing that the mincut in G is a $(1+\epsilon)$-approximate mincut in H,
(b) pack $O(\log n)$ s-arborescences[3] in the sparsifier H, and
(c) find the minimum weight cut among those that have at most two edges in an arboresence using a dynamic program.

The last step is sufficient because the duality between cuts and arborescences ensures that an s-mincut, which is now a $(1+\epsilon)$-approximate mincut after sparsification, has at most two edges in at least one s-arborescence.[4] Karger implements all these steps in $\tilde{O}(m)$ time to obtain an $\tilde{O}(m)$-time mincut algorithm in undirected graphs.

Unfortunately, steps (a) and (c) in Karger's scheme are not valid in a directed graph. To begin with, directed

[2]Gabow actually constructs a directionless spanning tree packing, which is a relaxation of an arborescence packing, but we ignore this technical detail here since it is not relevant to our eventual algorithm.

[3]Since Karger's algorithm considered undirected graphs, the arborescences are simply spanning trees.

[4]The duality implies that if the undirected mincut is λ, then we can pack λ spanning trees where every edge appears in at most two arborescences.

graphs do not admit sparsifiers similar to Karger's sparsifier: while Karger's sparsifier approximately preserves all cuts in an undirected graph (after some scaling), it is well known that a sparsifier with a similar property does not exist in directed graphs (see, e.g., [1]). This is mainly because we cannot bound the number of mincuts and approximate mincuts in directed graphs while we can do so in undirected graphs.

Partial Sparsification: Since it is impossible for a sparsifier to preserve all cuts in directed graphs, it is natural to try to preserve a *partial* subset of cuts. Suppose we were guaranteed that the s-mincut $(S, V \setminus S)$ (recall that $s \in S$) is unbalanced in the sense that $|V \setminus S| \leq k$ for some parameter k that we will fix later. Let us randomly sample edges to scale down the value of the mincut to $\tilde{O}(k)$. In an undirected graph, as long as $k = \Omega(\log n)$, all the cuts would converge to their expected values. This is not true in digraphs, but crucially, *all the unbalanced cuts converge to their expected values* since there are only $n^{\tilde{O}(k)}$ of them. However, it is possible that some balanced cut is (misleadingly) the new mincut of the sampled graph, having been scaled down disproportionately by the random sampling. So, we *overlay* this sampled graph with a star rooted at s, and show that this sufficiently increases the values of all balanced cuts, while only distorting the unbalanced cuts slightly. After this overlay, we can claim that we now have a digraph where $(S, V \setminus S)$ is a $(1+\epsilon)$-approximate mincut. We use one additional idea here. We show that in the sparsifier, every vertex in $V \setminus S$ has only $\tilde{O}(k)$ incoming edges (note that each edge can be weighted)—$\tilde{O}(k)$ edges across the cut from S and $\leq k$ edges from within $V \setminus S$. By contracting all vertices with unweighted in-degree $> \tilde{O}(k)$ into s, we reduce the number of edges in the digraph to $\tilde{O}(nk)$.

But, what if our premise that the s-mincut is unbalanced does not hold? This case is actually simple. We use a uniform random sample of $\tilde{O}(\frac{n}{k})$ vertices, and find $s-t$ mincuts for all vertices t from the sample, using $\tilde{O}(\frac{n}{k})$ maxflow calls. It is easy to see that whp, the sample *hits* $V \setminus S$, and hence, the minimum weighted cut among these $s-t$ mincuts will reveal the s-mincut of the graph.

Let us, therefore, return to the case where the s-mincut is unbalanced. Recall that we have already sparsified the graph to one that has only $\tilde{O}(nk)$ edges, and where the mincut has weight $\tilde{O}(k)$. The next step is to create a maximum packing of edge-disjoint s-arborescences. Because the graph is weighted, instead of using Gabow's algorithm described above, we create a (fractional) packing using a multiplicative weights update procedure (e.g., [30]). By duality,[5] these arborescences have the following property: if we sample $O(\log n)$ random s-arborescences, then whp there will be at least one arborescence T such that there is exactly one edge in T that goes from S to $V \setminus S$. In this case, we say that the cut $(S, V \setminus S)$ 1-*respects* the arborescence T.

1-respecting cut algorithm: Our final task, therefore, is the following: given an arborescence, find the minimum weight cut in the original graph among all those that 1-respect the arborescence T. At first sight, this may look similar to part (c) of Karger's algorithm which can be solved by a dynamic program or other techniques (e.g. [23], [13], [14], [20]). However, these techniques relied on the fact that if the s-mincut $(S, V \setminus S)$ has a single edge e in T, then S and $V \setminus S$ would be contiguous in an s-arborescence T (i.e. removing e from T gives S and $V \setminus S$ as the two connected components). This is not true for a directed graph: While an s-arborescence will contain exactly one edge from S to $V \setminus S$, it could contain an arbitrary number of edges in the opposite direction from $V \setminus S$ to S, thereby only guaranteeing the contiguity of $V \setminus S$ but not S.

One of the main contributions of this paper is to provide an algorithm to solve the above problem using $O(\log n)$ maxflow computations. The main idea is to use a centroid-based recursive decomposition of the arborescence, where in each step, we use a set of maxflow calls that can be amortized on the original graph. The minimum cut returned by all these maxflow calls is eventually returned as the s-mincut of the graph.

Approximation Algorithms: The ideas above also lead to a quadratic time approximation algorithm for edge mincuts. At a high level, if we execute each (s,t)-max flow in the sparsifier instead of in the input graph (both in the unbalanced and balanced settings, with care), then we obtain a $\tilde{O}(n^2/\epsilon^2)$ time $(1+\epsilon)$-approximation algorithm instead.

A similar approach can be taken for approximate vertex mincuts. Our partial sparsification technique reduces the graph to $\tilde{O}(nk/\epsilon^2)$ edges while maintaining the vertex s-mincut (with some additional adjustments for vertex mincuts). For large k, we similarly run (s,t)-max flow between $\tilde{O}(n/k)$ pairs of vertices on the sparsifier. For small k, we design a new local cut algorithm from $\tilde{O}(n/k)$ seeds each of which takes $\tilde{O}(k^3/\epsilon^2)$ time. This local algorithm is inspired by local algorithms for unweighted graphs [8], and speeds up the running time by a factor of k by leveraging the structure

[5]By duality, we have that if the directed mincut is λ, then we can pack λ arborescenses where each edge appears in at most one arborescense. Note that this is different from the case of undirected graphs where each edge appears in at most two arborescences.

of our sparsifiers (beyond sparsity). We finally obtain $\tilde{O}(n^2/\epsilon^2)$ running time by balancing the two cases and calling the max flow algorithm by [28].

Summary: To summarize, our algorithms distinguish between balanced and unbalanced mincuts, solving the former using maxflows on randomly sampled terminals. For unbalanced mincuts, we follow a two-step template. In the first step, we employ partial sparsification to preserve the values of unbalanced cuts approximately, while suppressing balanced cuts using an overlay. In the second step, we design algorithms to find the (edge/vertex) mincut among unbalanced cuts. For edge mincut, the sparsifier allows one to quickly obtain an arborescence that 1-respects the directed mincut. From this arboresence, we obtain the exact minimum cut in $O(\log n)$ max flows via a centroid-based recursive decomposition. For vertex mincut, we develop new local flow algorithms to identify small unbalanced cuts in the sparsified graph.

II. Minimum Cut Algorithms in Edge-weighted Directed Graphs

Given a directed graph $G = (V, E)$ with non-negative edge weights w and a fixed root vertex s, we consider the problem of finding an s-mincut. An *s-arborescence* is a directed spanning tree rooted at s such that all edges are directed away from s. For simplicity, we assume that all edge weights w are integers and are polynomially bounded. We denote $\overline{U} = V \setminus U$. Let $\partial^+(U)$ be the set of edges from U to $\overline{U}$, $\partial^-(U)$ be the set of edges from $\overline{U}$ to U, and let $\delta^+(U)$ and $\delta^-(U)$ be the weight of the cut, i.e., $\delta^+(U) = \sum_{e\in\partial^+(U)} w(e), \delta^-(U) = \sum_{e\in\partial^-(U)} w(e)$. Our goal is to compute the minimum cut (S^*, T^*) where $s \in S^* = \arg\min_{s\in U\subset V} \delta^+(U)$ and $T^* = \overline{S^*}$. Let $F(m, n)$ denote the time complexity of s-t maximum flow on a digraph with n vertices and m edges. The current record for this bound is $F(m, n) = \tilde{O}(m + n^{3/2})$ [28]. We emphasize that our directed mincut algorithm uses maxflow subroutines in a black box manner and therefore, any maxflow algorithm suffices. Correspondingly, we express our running times in terms of $F(m, n)$.

Theorem II.1. *There is a Monte Carlo algorithm that finds a minimum s-cut whp in $\tilde{O}(\min\{mk, nk^2\} + F(m, n)\frac{n}{k})$ time, where k is a parameter and $F(m, n)$ is the time complexity of s-t maximum flow.*

This section is devoted to prove Theorem II.1. If we set $k = m^{1/3}+n^{1/2}$ and use the $\tilde{O}(m+n^{3/2})$ max-flow algorithm, the time complexity becomes $\tilde{O}(nm^{2/3} + n^2)$, which establishes Theorem I.1. If we assume an hypothetical $\tilde{O}(m)$-time max-flow algorithm, then our result becomes $\tilde{O}(\min\{nm^{2/3}, mn^{1/2}\})$ for $k = \min\{m^{1/3}, n^{1/2}\}$.

We obtain Theorem II.1 via a new s-mincut algorithm. The algorithm considers the following two cases, computing a s-cut for each case and returning the minimum as its final output. The cases are split on $|T^*|$ by a threshold $k > 0$.

1) The first case aims to compute the correct mincut in the event that $|T^*| > k$. In this case, if we randomly sample $t \in V$, then with probability at least $1/k$, $t \in T^*$. Then T^* can be obtained via the maxflow from s to t. Repeating the sampling $O(\frac{n}{k}\log n)$ times, we obtain the minimum s-cut whp. The total running time for this case is $O(F(m, n)\frac{n}{k}\log n)$.
2) The second case is for the the event that $|T^*| \le k$. Let λ denote the value of the minimum rooted cut. By enumerating $O(\log n)$ powers of 2, we can obtain an estimate $\tilde{\lambda}$ such that $\lambda \le \tilde{\lambda} \le 2\lambda$. For each value of $\tilde{\lambda}$, we apply Lemma II.5 to sparsify the graph in the following manner. First, Lemma II.5 returns a set of vertices $V_0 \subseteq V$ such that $s \in V_0$ and $T^* \subseteq V_0$ whp. In particular, one can safely contract any vertex $v \in V \setminus V_0$ into s without affecting the minimum s-cut. We contract G accordingly and, overloading notation, let G denote the contracted graph with vertex set V_0 henceforth. Second, Lemma II.5 returns a graph $G_0 = (V_0, E_0)$ in which T^* still induces an $(1+\epsilon)$-approximate s-mincut, but the weight of the cut is now reduced to $O(k\log(n))$. We note that G_0 is not necessarily a subgraph of G. We then invoke Lemma II.6 from Section II-B to fractionally pack an approximately maximum amount of $O(k\log n)$ s-arboresences in G_0 in $\tilde{O}\big(m + \min\{mk, nk^2\}\big)$ time. In a random sample of $O(\log n)$ s-arboresences from this packing, one of them will 1-respect the s-mincut in G (for appropriate $\tilde{\lambda}$) whp:

   Definition II.2. A directed s-cut $(S, V \setminus S)$ *k-respects* an s-arborescence if there are at most k edges in the arborescence from S to $V \setminus S$.

   Finally, for each of the $O(\log n)$ s-arborescences, the algorithm computes the minimum s-cut that 1-respects each arborescence. This algorithm is described in Algorithm 1 and proved in Theorem II.7 from Section II-C. It runs in $O((F(m, n) + m)\cdot \log n)$ time for each of the $O(\log n)$ arborescences.

Combining both cases, the total running time becomes $\tilde{O}(\min\{mk, nk^2\}+F(m, n)\frac{n}{k})$, which establishes Theorem II.1.

Fast approximations: The exact algorithm described above can be modified to produce a randomized $\tilde{O}\big(n^2/\epsilon^2\big)$-time approximation algorithm that computes a $(1+\epsilon)$-approximate minimum s-cut (hence also the

global cut). With logarithmic overhead, we can obtain a parameter k such that $k/2 \leq |T^*| \leq k$, where $\partial^-(T^*)$ is the minimum s-cut. We then follow the same steps as in the exact algorithm, except whenever we compute the max-flow, we compute it in the sparsifier produced by Lemma II.5 instead. Since the sparsifier has at most $\tilde{O}(nk/\epsilon^2)$ edges, we obtain a running time of the form $\tilde{O}\big(\min\{nk^2/\epsilon^2\} + F(nk/\epsilon^2, n)(n/k)\big)$. For $F(m,n) = \tilde{O}\big(m + n^{1.5}\big)$, this gives a running time of $\tilde{O}\big(n^2/\epsilon^2\big)$.

Theorem II.3. *For $\epsilon \in (0,1)$, an $(1+\epsilon)$-approximate minimum s-cut (hence global minimum cut) can be computed in $\tilde{O}(n^2/\epsilon^2)$ time whp.*

We remark that $\tilde{O}\big(n^2/\epsilon^2\big)$ can also be obtained by using a local connectivity algorithm similar to the approach for vertex mincuts in Section III, instead of via an arboresence packing. See [26] for details.

Organization: The rest of this section is organized as follows. The following subsections present each step of the algorithm described above. First, we establish the partial sparsification subroutine in Section II-A. Next, in Section II-B, we obtain an arborescence packing from which sampling yields an arborescence that is 1-respected by the mincut. Finally, in Section II-C, we describe the algorithm to retrieve the mincut among those that 1-respect a given arborescence.

A. Partial Sparsification

This section aims to reduce mincut value to $\tilde{O}(k)$ and edge size to $\min\{m, O(nk\log(n)/\epsilon^2\})$ while keeping $\partial^+(S^*)$ a $(1+\epsilon)$-approximate s-mincut for a constant $\epsilon > 0$ that we will fix later. Our algorithm in this stage has three steps. First, we use random sampling to discretize and scale down the expected value of all cuts such that the expected value of the mincut $\delta^+(S^*)$ becomes $\tilde{O}(k)$. We also claim that $\partial^+(S^*)$ remains an approximate mincut *among all unbalanced cuts* by using standard concentration inequalities. However, since the number of balanced cuts far exceeds that of unbalanced cuts, it might be the case that some balanced cut has now become much smaller in weight than all the unbalanced cuts. This would violate the requirement that $\partial^+(S^*)$ should be an approximate mincut in this new graph. This is where we need our second step, where we overlay a star on the sampled graph to raise the values of all balanced s-cuts above the expected value of $\partial^+(S^*)$ while only increasing the value of $\partial^+(S^*)$ by a small factor. The third step leverages the fact that, after scaling, the minimum edge weight is 1, and the minimum cut is $O\big(k\log(n)/\epsilon^2\big)$. It follows that any vertex with in-degree at least a constant factor greater than $k\log(n)/\epsilon^2$ cannot be in the sink component, and can be safely contracted into the root without affecting the s-mincut.

The first two steps described above are implemented in the next lemma, whose proof is deferred to the full version.

Lemma II.4. *Let $G = (V,E)$ be a directed graph with positive edge weights. Let $s \in V$ be a fixed root vertex. Let $\epsilon \in (0,1)$, let $\lambda > 0$, and let $k \in \mathbb{N}$ be given parameters. Suppose there is an s-mincut of the form $\partial^-(T^*)$, where $s \notin T^*$, $\lambda/2 \leq \delta^-_G(T^*) \leq 2\lambda$ and $|T^*| \leq k$. In randomized nearly linear time, one can compute a randomized directed and reweighted subgraph $G_0 = (V, E_0)$, where $V_0 \subseteq V$ and $s \in V_0$ with the following properties.*

(i) G_0 has integral edge weights and the minimum s-cut has weight at most $O\big(k\log(n)/\epsilon^2\big)$.

(ii) $\partial^-_{G_0}(T^*)$ is a $(1+\epsilon)$-approximate minimum s-cut in G_0.

(iii) Every α-approximate minimum s-cut in G_0 induces an $(1+\epsilon)\alpha$-approximate minimum s-cut in G.

The preceding lemma importantly allows us to reduce the weight of the minimum cut to roughly k, where k is the number of the vertices in the sink component. However it has not actually reduced the size in the graph, in terms of the number of edges. This is accomplished by our third step, which we formalize in the following lemma that reduces the number of edges to $\tilde{O}(nk)$.

Lemma II.5. *Let $G = (V,E)$ be a directed graph with positive edge weights. Let $s \in V$ be a fixed root vertex. Let $\epsilon \in (0,1)$, $\lambda > 0$, and $k \in \mathbb{N}$ be given parameters. Suppose there is a minimum s-cut is of the form $\partial^-(T^*)$, where $\lambda/2 \leq \delta^-_G(T^*) \leq 2\lambda$ and $|T^*| \leq k$.*

In randomized nearly linear time, one can compute a randomized directed and edge-weighted graph $G_0 = (V_0, E_0)$, where $V_0 \subseteq V$ and $s \in V_0$.

(i) G_0 has integral edge weights and the s-mincut in G_0 has weight at most $O\big(k\log(n)/\epsilon^2\big)$.

(ii) G_0 has at most $|E_0| = \min\big\{m, O\big(nk\log(n)/\epsilon^2\big)\big\}$ edges.

(iii) G_0 is a subgraph of the graph obtained by contracting $V \setminus V_0$ into s in G.

(iv) We have $T^* \subseteq V_0$, and $\partial^-_{G_0}(T^*)$ is an $(1+\epsilon)$-approximate minimum s-cut in G_0.

(v) Every $(1+\epsilon)$-approximate minimum s-cut in G_0 induces an $(1+\epsilon)^2$-approximate minimum s-cut in G.

Proof: Consider the reweighted subgraph produced by Lemma II.4, which we denote by $G_1 = (V, E_1)$. We claim that every vertex $v \in T^*$ has *unweighted* in-degree at most $O\big(k\log(n)/\epsilon^2\big)$. Indeed, at most $k-1$ of these edges are from other vertices in T, and the

remaining edges must be in $\partial^-_{G_1}(T^*)$. But $\partial^-_{G_1}(T^*)$ has at most $\delta^-_{G_1}(T^*) = O\big(k\log(n)/\epsilon^2\big)$ edges by properties (i) and (ii) of Lemma II.4.

Let $G_0 = (V_0, E_0)$ be the graph obtained from G_1 by contracting all vertices with unweighted in-degree $\geq ck\log(n)/\epsilon^2$ for a sufficiently large constant c that excludes all vertices in T^*. It is easy to see that G_0 satisfies the claimed properties, particularly as contractions into s do not decrease the s-mincuts, and $\partial^-_{G_0}(T^*)$ is preserved exactly as in G_1. ■

B. Finding a 1-respecting Arborescence

In this section, we assume that there is an unbalanced s-mincut and show how to obtain an s-arborescence that 1-respects the s-mincut. More formally, we prove the following:

Lemma II.6. *Given weighted digraph G and a fixed root vertex s, suppose the sink side of an s-mincut T^* has at most k vertices. In $O(m\log n + \min\{mk\log^2 n, nk^2\log^3 n\})$ time, we can find $O(\log n)$ s-arborescences on vertex set $V_0 \supset T^*$, such that whp an s-mincut 1-respects at least one of them.*

The idea of this lemma is as follows. First, we apply Lemma II.5 to our graph G and obtain the graph G_0. Whp, a minimum s-cut $\partial^-(T^*)$ in G corresponds to a $(1+\epsilon)$-approximate minimum s-cut in G_0. It remains to find an arborescence in G_0 that 1-respects $\partial^-(T^*)$. To do this, we employ a multiplicative weight update (MWU) framework. The algorithm begins by setting all edge weights to be uniform (say, weight 1). Then, we repeat the following for $O(k\log(n)/\epsilon^2)$ rounds: in each round, we find a minimum weight arborescence in $\tilde{O}(m)$ time and multiplicatively increase the weight of every edge in the arborescence. Using the fact that there is no duality gap between arborescence packing and mincut [5], [9], a standard MWU analysis implies that these arborescences that we found, after scaling, form a $(1+\epsilon)$-approximately optimal fractional arborescence packing. So our arborescence crosses T^* at most $(1+O(\epsilon)) < 2$ times on average. Thus, if we sample $O(\log n)$ arborescences from this set, one of them will 1-respect T^* whp.[6] To obtain the running time bound, we note that each iteration of the MWU framework requires us to find the minimum cost s-arborescence, for which an $\tilde{O}(m)$-time algorithm is known [10].

Since the argument above is a standard application of the MWU framework, we defer the detailed proof to the full version.

[6]This should be compared with Karger's mincut algorithm in the undirected case, where there is a factor 2 gap, and hence Karger can only guarantee a 2-respecting tree in the undirected case.

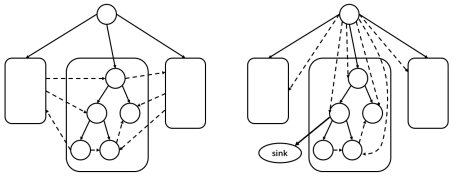

Figure 1. Construction of auxiliary graph G_i in Algorithm 1. Solid lines represent the arborescence T. Dashed lines are other edges in the graph. Rectangles are sets formed by the first level of centroid decomposition. Left: The original graph. Right: The part of G_1 solving the case that the mincut separates root and the centroid of the middle subtree.

C. Mincut Given 1-respecting Arborescence

We propose an algorithm (Algorithm 1) that uses $O(\log n)$ maxflow subroutines to find the minimum s-cut that 1-respects a given s-arborescence. The result is formally stated in Theorem II.7.

Theorem II.7. *Consider a directed graph $G = (V, E)$ with polynomially bounded edge weights $w_e > 0$. Let $s \in V$ be a fixed root vertex and $S \ni s$ be the source side of a fixed s-mincut. Given an s-arborescence T with $|T \cap \partial^+(S)| = 1$, Algorithm 1 outputs a s-mincut of G in time $O((F(m,n)+m)\cdot\log n)$.*

We first give some intuition for Algorithm 1. Because $s \in S$, if we could find a vertex $t \in \overline{S}$, then computing the s-t mincut using one maxflow call would yield a global mincut of G. However, we cannot afford to run one maxflow between s and every other vertex in G. Instead, we carefully partition the vertices into $\ell = O(\log n)$ sets $(C_i)_{i=1}^{\ell}$. We show that for each C_i, we can modify the graph appropriately so that it allows us to (roughly speaking) compute the maximum flow between s and every vertex $c \in C_i$ using one maxflow call.

More specifically, Algorithm 1 has two stages. In the first stage, we compute a *centroid decomposition* of T. Recall that a centroid of T is a vertex whose removal disconnects T into subtrees with at most $n/2$ vertices. This process is done recursively, starting with the root s of T. We let P_1 denote the subtrees resulting from the removal of s from T. In each subsequent step i, we compute the set C_i of the centroids of the subtrees in P_i. We then remove the centroids and add the resulting subtrees to P_{i+1}. This process continues until no vertices remain.

In the second stage, for each layer i, we construct a directed graph G_i and perform one maxflow computation on G_i. The maxflow computation on G_i would yield candidate cuts for every vertex in C_i, and after computing the appropriate maximum flow across every layer, we output the minimum candidate cut as the minimum cut of G. The details are presented in Algorithm 1.

Algorithm 1: Finding an s-mincut.

Input : An arborescence T rooted at $s \in S$ such that S 1-respects T.

1 // Stage I: Build centroid decomposition.
2 Let $C_0 = \{s\}$, $P_1 =$ the set of subtrees obtained by removing s from T, and $i = 1$.
3 **while** $P_i \neq \varnothing$ **do**
4 Initialize C_i (the centroids of P_i) and P_{i+1} as empty sets.
5 **for** *each subtree* $U \in P_i$ **do**
6 Compute the centroid u of U and add it to C_i.
7 Add all subtrees generated by removing u from U to P_{i+1}.
8 Set $\ell = i$ and iterate $i = i + 1$.
9 // Stage II: Calculate integrated maximum flow for each layer.
10 **for** $i = 1$ **to** ℓ **do**
11 Construct a digraph $G_i = (V \cup \{t_i\}, E_1 \cup E_2 \cup E_3)$ as follows (see Figure 1):
12 1) Add edges $E_1 = E \cap \cup_{U \in P_i}(U \times U)$ with capacity equal to their original weight.
13 2) Add edges $E_2 = \{(s, v) : (u, v) \in E \setminus E_1\}$ with capacity of (s, v) equal to the original weight of (u, v).
14 3) Add edges $E_3 = \{(u, t_i) : u \in C_i\}$ with infinite capacity.
15 Compute the maximum s-t_i flow f_i^* in G_i.
16 For each component $U \in P_i$ with centroid u, the value of f_i^* on edge (u, t_i) is a candidate cut value, and the nodes in U that can reach u in the residue graph is a candidate for $\overline{S}$.
17 Return the smallest candidate cut value and the corresponding $(S, \overline{S})$ as an s-mincut.

We first state two technical lemmas that we will use to prove Theorem II.7.

Lemma II.8. *Recall that P_i is the set of subtrees in layer i and C_i contains the centroid of each subtree in P_i. If $C_j \subseteq S$ for every $0 \leq j < i$, then $\overline{S}$ is contained in exactly one subtree in P_i, and consequently, at most one vertex $u \in C_i$ can be in $\overline{S}$.*

Lemma II.9. *Let G_i be the graph constructed in Step 11 of Algorithm 1. Let f_i^* be a maximum s-t_i flow on G_i as in Step 15. For any $U \in P_i$ with centroid u, the amount of flow f_i^* puts on edge (u, t_i) is equal to the value of the minimum cut from $\overline{U}$ to u.*

We defer the proofs of Lemmas II.8 and II.9, and first use them to prove Theorem II.7.

Proof of Theorem II.7: We first prove the correctness of Algorithm 1.

Because $C_0 = \{s\}$ and $s \in S$, and the C_i's form a disjoint partition of V, there must be a layer i such that for the first time, we have a centroid $u \in C_i$ that belongs to $\overline{S}$. By Lemma II.8, we know that $\overline{S}$ must be contained in exactly one subtree $U \in P_i$, and hence u must be the centroid of U. In summary, we have $u \in \overline{S}$ and $\overline{S} \subseteq U$.

Consider the graph G_i constructed for layer i. By Lemma II.9, based on the flow f_i^* puts on the edge (u, t_i), we can recover the value of the minimum cut from $\overline{U}$ to u. Because $\overline{S} \subseteq U$ (or equivalently $\overline{U} \subseteq S$) and $u \in \overline{S}$, the cut $(S, \overline{S})$ is one possible cut that separates $\overline{U}$ and u. Therefore, the flow that f_i^* puts on the edge (u, t_i) is equal to the s-mincut value in G.

In addition, the candidate cut value for any other centroid u' of a subtree $U' \in P_i$ must be at least the mincut value between s and u'. This is because the additional restriction that the cut has to separate $\overline{U'}$ from u' can only make the mincut value larger, and the value of this cut in G_i is equal to the value of the same cut in G. Therefore, the minimum candidate cut value in all ℓ layers must be equal to the s-mincut value of G.

Now we analyze the running time of Algorithm 1. We can find the centroid of an n-node tree in time $O(n)$ (see e.g., [22]). The total number of layers $\ell = O(\log n)$ because removing the centroids reduces the size of the subtrees by at least a factor of 2. Thus, the running time of Stage I of Algorithm 1 is $O(n \log n)$. In Stage II, we can construct each G_i in $O(m)$ time and every G_i has $O(m)$ edges. Since there are $O(\log n)$ layers and the maximum flow computations take a total of $O(MF(m, n) \cdot \log n)$ time, the overall runtime is $O(n \log n + (MF(m, n) + m) \log n) = O((MF(m, n) + m) \log n)$. ∎

Before proving Lemmas II.8 and II.9 we first prove the following lemma.

Lemma II.10. *If x and y are vertices in $\overline{S}$, then every vertex on the (undirected) path from x to y in the arborescence T also belongs to $\overline{S}$.*

Proof: Consider the lowest common ancestor z of x and y. Because there is a directed path from z to x and a directed path from z to y, we must have $z \in \overline{S}$. Otherwise, there are at least two edges in T that go from S to $\overline{S}$.

Because $s \in S$ and $z \in \overline{S}$, there is already an edge in T (on the path from s to z) that goes from S to $\overline{S}$. Consequently, all other edges in T cannot go from S to $\overline{S}$, which means the entire path from z to x (and similarly z to y) must be in $\overline{S}$. ∎

Recall that Lemma II.8 states that if all the centroids in previous layers are in S, then $\overline{S}$ is contained in exactly one subtree U in the current layer i.

Proof of Lemma II.8: For contradiction, suppose that there exist distinct subtrees U_1 and U_2 in P_i and vertices $x, y \in \overline{S}$ such that $x \in U_1$ and $y \in U_2$.

By Lemma II.10, any vertex on the (undirected) path from x to y also belongs to $\overline{S}$. Consider the first time that x and y are separated into different subtrees. This must have happened because some vertex on the path from x to y is removed. However, the set of vertices removed at this point of the algorithm is precisely $\bigcup_{0 \le j < i} C_j$, but our hypothesis assumes that none of them are in $\overline{S}$. This leads to a contradiction and therefore $\overline{S}$ is contained in exactly one subtree of P_i.

It follows immediately that at most one centroid $u \in C_i$ can be in $\overline{S}$. ∎

Next we prove Lemma II.9, which states that the maximum flow between s and t_i in the modified graph G_i allows one to simultaneously compute a candidate mincut value for each vertex $u \in C_i$.

Proof of Lemma II.9: First observe that the maxflow computation from s to t_i in G_i can be viewed as multiple independent maxflow computations. The reason is that, for any two subtrees $U_1, U_2 \in P_i$, there are only edges that go from s into U_1 and from U_1 to t_i in G_i (similarly for U_2), but there are no edges that go between U_1 and U_2.

The above observation allows us to focus on one subtree $U \in P_i$. Consider the procedure that we produce G_i from G in Steps 12 to 14 of Algorithm 1. The edges with both ends in U are intact (the edge set E_1). If we contract all vertices out of U into s, then all edges that enter U would start from s, which is precisely the effect of removing cross-subtree edges and adding the edges in E_2. One final infinity-capacity edge $(u, t_i) \in E_3$ connects the centroid of U to the super sink t_i.

Therefore, the maximum s-t_i flow f_i^* computes the maximum flow between $\overline{U}$ and $u \in U$ simultaneously for all $U \in P_i$, whose value is reflected on the edge (u, t_i). It follows from the maxflow mincut theorem that the flow on edge (u, t_i) is equal to the mincut value between $\overline{U}$ and u in G (i.e., the minimum value $w(A, \overline{A})$ among all $A \subset V$ with $\overline{U} \subseteq A$ and $u \in \overline{A}$). ∎

III. Minimum Cut Algorithms in Vertex-Weighted Directed Graphs

In this section we present the approximation algorithm for the minimum rooted and global vertex cut. Similar to Section II, the main focus is on rooted cuts, and the algorithm is presented in three main parts. All three parts are parameterized by values $\kappa > 0$ and $k \in \mathbb{N}$ that, in principle, are meant to be constant factor estimates for the weight and the number of vertices in the sink component of the minimum rooted vertex cut. The first part, in Section III-A, presents the sparsification lemma that reduces the number of edges to roughly nk and the rooted mincut to roughly k in a graph with integer weights. This sparsifier is used in the remaining two parts. The second part, in Section III-B, gives a roughly nk^2 time approximation algorithm for the minimum rooted cut via a new local flow algorithm. The third part, in Section III-C, gives a roughly $n^2 + n^{2.5}/k$ time approximation algorithm via sampling and (s,t)-flow (as with minimum edge cuts before). Finally, in Section III-D, we balance terms to obtain the claimed running time for rooted cut. The rooted vertex mincut algorithm then leads to a global vertex mincut algorithm via an argument due to [17] (with some modifications).

A. Partial Sparsification

The first part is a sparsification lemma that preserves rooted vertex cuts where the number of vertices in the sink component is below some given parameter. It is similar in spirit to Lemma II.5, but with some necessary changes as we are now preserving the vertex mincut rather than edge mincut. We give a brief overview of the algorithm, highlighting in particular the differences from the partial edge cut sparsifier. The proof and algorithmic details are deferred to the full version.

At a high level, the following sparsifier for vertex cuts randomly samples the *vertex* weights so that the weights are integral, and the weight of the minimum vertex cut becomes $O\big(k \log(n)/\epsilon^2\big)$. Similar to the partial edge sparsifier, this rounding is calibrated to preserve s-cuts with (roughly) k or fewer vertices in the sink components. To pad the weight of vertex s-cuts with large sink components, we add an weighted auxiliary vertex on a short directed path between s and each vertex (as opposed to just adding an edge from s, as we did for edge cuts). If a sampled weight of a vertex v is 0, we cannot simply drop the vertex from the graph (in the way we can drop weight 0 edges) since the vertex may be in the sink component of the min r-cut. Instead we remove all outgoing edges from v. Also, when we detect that a vertex v cannot be in the sink component (by a similar counting argument as before), rather than contract v into s (which may effect the min vertex s-cut), we replace all of the incoming edges to v with a single edge from s. The culmination of these modifications is a similar net effect as for edge cuts: a graph with $O\big(nk \log(n)/\epsilon^2\big)$ that preserves the sink component of the minimum vertex s-cut. That said, the following bounds are more detailed than the bounds for preserving the edge cut in Lemma II.5. These additional properties play a critical role in the customized local flow algorithms presented later.

In the following, let $N^+(v \mid G)$ denote the set of out-neighbors of v in the graph G. We omit G and simply write $N^+(v)$ when G can be inferred from the context.

Lemma III.1. *Let $G = (V, E)$ be a directed graph with positive vertex weights. Let $s \in V$ be a fixed vertex. Let $k, \kappa > 0$ be given parameters. Let $V' = V \setminus (\{s\} \cup N^+(s))$. In randomized linear time, one can compute a randomized directed and vertex-weighted graph $G_0 = (V_0, E_0)$, and a scaling factor $\tau > 0$, with the following properties.*

(i) $s \in V_0$.

(ii) *Let $V_0' = V_0 \setminus (\{s\} \cup N^+(s \mid G_0))$. We have $V_0' = V'$.*

(iii) *G_0 has integer vertex weights between 0 and $O\big(k \log(n)/\epsilon^2\big)$.*

(iv) *Every vertex $v \in V_0$ has at most $O\big(k \log(n)/\epsilon^2\big)$ incoming edges.*

(v) *Every vertex v with weight 0 has no outgoing edges.*

(vi) *With high probability, for all $S \subseteq V'$, the weight of the vertex in-cut induced by S in G_0 (up to scaling by τ) is at least the minimum of the $(1-\epsilon)$ times the weight of the induced vertex in-cut in G or $c\kappa$ (for any desired constant $c > 1$), and at most $(1+\epsilon)$ times its weight in G plus $\epsilon\kappa|S|/k$.*

(vii) *With high probability, for all $S \subseteq V'$ such that $|S| \leq k$ and the weight of the induced vertex in-cut is $\leq O(\kappa)$, we have $S \subseteq V_0'$. (That is, S is still the sink component of an s-cut in G_0.)*

In particular, if the minimum vertex s-cut has weight $\Theta(\kappa)$, and the sink component of a minimum vertex s-cut has at most k vertices, then with high probability G_0 preserves the minimum vertex s-cut up to a $(1 + O(\epsilon))$-multiplicative factor.

As stated above, the proof is deferred to the full version.

B. Rooted vertex mincut for small sink components

This section presents an approximation algorithm for rooted vertex mincut for the particular setting where the sink component is small. In particular, we are given an upper bound k on the number of vertices in the sink component, and want to obtain running times of the form $n\,\mathrm{poly}(k)$. When a similar situation arose previously for small integer capacities in [3], [3] modified a local algorithm from [8] which works well for unweighted graphs. Here, while Lemma III.1 produces relatively sparse graphs with integral vertex capacities, the vertex capacities imply imply that the algorithm from [3], [8] would take roughly nk^3/ϵ^5 time. This section develops an alternative algorithm that is inspired by these local algorithms for (global and rooted) vertex cuts, but reduces the dependency on k to k^2. Compared to [8], [3], the algorithm here is designed to take full advantage of the properties of the graph produced by Lemma III.1. These modifications have some tangible benefits. First, it improves the dependency on k and ϵ. Second, the local subroutine here is deterministic whereas before they were randomized. Third and last, as suggested by the better running time and the determinism, the version presented here is arguably simpler and more direct than the previous algorithms (for this setting).

Lemma III.2. *Let $G = (V, E)$ be a directed graph with positive vertex weights. Let $r \in V$ be a fixed root vertex. Let $\epsilon \in (0, 1)$, $\kappa > 0$ and $k \in \mathbb{N}$ be given parameters. There is a randomized linear time Monte Carlo algorithm that, with high probability, produces a deterministic data structure that supports the following query.*

For $t \in V' \stackrel{\text{def}}{=} V \setminus (\{s\} \cup N^+(s))$, let $\kappa_{t,k}$ denote the weight of the minimum (s,t)-vertex cut such that the sink component has at most k vertices. Given $t \in V'$, deterministically in $O\big(k^3 \log^2(n)/\epsilon^4\big)$ time, the data structure either (a) returns the sink component of a minimum (s,t)-vertex cut of weight at most $(1+\epsilon)\kappa_{t,k}$, or (b) declares that $\kappa_{t,k} > \kappa$.

Proof: Given s, κ, k, and ϵ, let $\epsilon' = c\epsilon$ for a sufficiently small constant $c > 0$. We first apply Lemma III.1 to G with root s and parameters κ, k, and ϵ'. This produces a vertex capacitated graph $G_0 = (V_0, E_0)$ with $V \subset V_0$. We highlight the features that we leverage. All new vertices (in $V_0 \setminus V$) are in $N^+(s \mid G_0)$; that is, V' equals $V_0' \stackrel{\text{def}}{=} V_0 \setminus (\{s\} \cup N^+(s \mid G_0))$. Put alternatively, none of the new vertices is in the sink component of any s-cut. The vertex weights are integers between 0 and $O\big(k \log(n)/\epsilon^2\big)$. Every vertex has unweighted in-degree at most $O\big(k \log(n)/\epsilon^2\big)$. Every vertex with weight 0 has no outgoing edges.

With high probability, we have the following guarantees on the vertex s-cuts of G_0. The vertex weights in G_0 are scaled so that a weight of κ in G corresponds to weight $O\big(k \log(n)/\epsilon^2\big)$ in G_0. Modulo scaling, every vertex s-cut in G_0 has weight no less than the minimum of its weight in G and 2κ. Additionally, modulo scaling, for every vertex s-cut in G with capacity at most κ and at most k vertices in the sink component, the corresponding vertex cut in G_0 has weight at most a $c_0\epsilon\kappa$ additive factor larger than in G, for any desired constant $c_0 > 0$. We consider the algorithm to fail if the cuts are not preserved in the sense described above.

Given $t \in V$, the data structure will search for a small (s,t)-cut in G_0 via a customized, edge-capacitated flow algorithm. This algorithm may or may not return the sink component of (s,t)-cut. If the search does return

a sink component, and the corresponding vertex in-cut in G_0 has weight that, upon rescaling back to the scale of the input graph G, is at most $(1+\epsilon/2)\kappa$, the data structure returns it. Otherwise the data structure indicates that $\kappa_{t,k} > \kappa$.

Proceeding with the flow algorithm, let G_{rev} be the reverse of G_0, and let G_{split} be the standard "split-graph" of G_{rev} modeling vertex capacities with edge capacities. We recall that the split graph splits each vertex v into an auxiliary "in-vertex" v^- and an auxiliary "out-vertex" v^+. For each v there is a new edge (v^-, v^+) with capacity equal to the vertex capacity of v. Each edge (u, v) is replaced with an edge (u^+, v^-) with capacity[7] equal to the vertex capacity of u. Every (s,t)-vertex cut in G_0 maps to a (t^+, s^-)-edge cut in G_{rev} with the same capacity. Any (t^+, s^-)-edge capacitated cut maps to a (s,t)-vertex cut in G_0 (with negligible overhead in the running time). Now, recall that for each $v \in V'$, the sparsification procedures introduces an auxiliary path (s, a_v, s^-) where a_v is was given weight $\Theta(\epsilon\kappa/k)$. It is convenient to replace the corresponding auxiliary path (v^+, a_v^-, a_v^+, s^-) in G_{rev} with a single edge (v^+, s^-) with capacity equal to the weight of a_v. This does not effective the weight of the minimum (t^+, s^-)-edge cut for any $t \in V'$. This adjustment can be easily made within the allotted preprocessing time.

In this graph, given $t \in V'$, we run a specialization of the Ford-Fulkerson algorithm [7] that either computes a minimum (t^+, s^-)-cut or concludes that the minimum (t^+, s^-)-cut is at least $O\big(k\log(n)/\epsilon^2\big)$ (which corresponds to $O(\kappa)$ in G) after $O\big(k\log(n)/\epsilon^2\big)$ iterations. Observe that since every vertex initially has unweighted out-degree at most $O\big(k\log(n)/\epsilon^2\big)$ in G_{rev} (reversing the upper bound on the unweighted in-degrees in G_0), and the flow algorithm updates the residual graph along at most $O\big(k\log(n)/\epsilon^2\big)$ paths before terminating, the maximum unweighted out-degree over all vertices never exceeds $O\big(k\log(n)/\epsilon^2\big)$. We specialize the Ford-Fulkerson framework to take advantage of the auxiliary (v^+, s^-) edges. Call an out-vertex v^+ *saturated* if the auxiliary edge (v^+, s^-) is saturated; that is, if (v^+, s^-) is not in the residual graph. Call an in-vertex v^- *saturated* if the edge (v^-, v^+) is saturated and v^+ is not saturated. (A vertex v^+ or v^- is called *unsaturated* if it is not saturated.) We modify the search for an augmenting path to effectively end when we first visit an unsaturated vertex v^+ or an unsaturated v^-. If we visit an unsaturated v^-, then we automatically complete a path to s^- via v^+. If we find an unsaturated v^+, then we automatically complete a path to s^- via the edge (v^+, s^-). It remains to bound the running time of this search. We first bound the number of saturated v^+'s.

[7]Usually, this edge is set to capacity ∞, but either the weight of u or the weight of v are also valid.

Claim 1. There are at most $O(k/\epsilon)$ saturated v^+'s.

Indeed, each saturated v^+ implies $O(\log(n)/\epsilon)$ units of flow along (v^+, s^-), and the flow is bounded above $O\big(k\log(n)/\epsilon^2\big)$.

Note that Claim 1 also implies there are at most $O(k/\epsilon)$ v^-'s such that v^+ is saturated. The next claim bounds the total out-degree of saturated v^-'s.

Claim 2. The sum of out-degrees of saturated v^-'s is at most the amount of flow routed to s^-.

Indeed, the out-degree of a v^- in the residual graph is bounded above by the amount of flow through (v^-, v^+), since initially (v^-, v^+) is the only outgoing edge from v^-. Recall that if v^- is saturated, then by definition v^+ is unsaturated. As long as v^+ is unsaturated, each unit of flow through (v^-, v^+) goes directly to s^- via the edge (v^+, s^-), and can be charged to the total flow.

We now apply the above two claims to bound the total running time for each search, as follows.

Claim 3. Every (modified) search for an augmenting path traverses at most $O\big(k^2\log(n)/\epsilon^2\big)$ edges.

We first observe that every vertex visited in the search, except the unsaturated vertex terminating the search, is either (a) a saturated v^-, (b) a saturated v^+, or (c) an unsaturated v^- such that v^+ is saturated. We will upper bound the number of edges traversed in each iteration based on the type of vertex at the initial point of that edge. First, the amount of time spent exploring edges leaving (a) a saturated v^- is, by Claim 2, at most the size of the flow at that point, which is at most $O\big(k\log(n)/\epsilon^2\big)$. Second, consider the time spent traversing edges leaving either (b) a saturated v^+ or (c) an unsaturated v^- such that v^+ is saturated. By Claim 1, there are at most $O(k/\epsilon)$ such vertices, and each has out-degree at most $O\big(k\log(n)/\epsilon^2\big)$. Thus we spend $O\big(k^2\log(n)/\epsilon^2\big)$ time traversing such edges. All together, we obtain an upper bound of $O\big(k^2\log(n)/\epsilon^2\big)$ total edges per search.

Claim 3 also bounds the running time for each iteration. The algorithm runs for at most $O\big(k\log(n)/\epsilon^2\big)$ iterations before either finding an (t^+, s^-)-cut or concluding that the weight of the minimum (t^+, s^-)-cut, rescaled to the input scale of G, is at least a constant factor greater than κ. The total running time follows. ■

We now present the overall algorithm for finding vertex s-cuts with small sink components. The algorithm combines Lemma III.2 with randomly sampling for a vertex t in the sink component of an approximately minimum s-cut. In the following, we let $\deg^+(s)$ denote the *unweighted* out-degree of s in G.

Lemma III.3. *Let $G = (V, E)$ be a directed graph with positive vertex weights. Let $s \in V$ be a fixed root vertex. Let $\epsilon \in (0,1)$, $\kappa > 0$ and $k \in \mathbb{N}$ be given parameters. There is a randomized algorithm that*

runs in $O\big(m + (n - \deg^+(s))k^2 \log^3(n)/\epsilon^4\big)$ time and has the following guarantee. If there is a vertex s-cut of capacity at most κ and where the sink component has at most k vertices, then with high probability, the algorithm returns a vertex (s,t)-cut of capacity at most $(1+\epsilon)\kappa$.

Proof: Let T^* be the sink component of the minimum vertex s-cut subject to $|T^*| \leq k$. Assume the capacity of the vertex in-cut of T^* is at most κ (since otherwise the algorithm makes no guarantees). Let $V' = V \setminus (\{s\} \cup N^+(s))$ and note that $|V'| = n - 1 - \deg^+(s)$.

Suppose we had a factor-2 overestimate $\ell \in [|T^*|, 2|T^*|]$ of the number of vertices in T^*. We apply Lemma III.2 with upper bounds κ on the size of the cut and ℓ on the number of vertices in the sink component, which returns a data structure that, with high probability, is correct for all queries. Let us assume the data structure is correct (and otherwise the algorithm fails). We randomly sample $O\big((n - \deg^+(s)) \log(n)/\ell\big)$ vertices from V'. For each sampled vertex t, we query the data structure from Lemma III.2. Observe that if $t \in T^*$, then the query for t returns an s-cut with capacity at most $(1+\epsilon)\kappa$. With high probability we sample at least one vertex from T^*, which produces the desired s-cut. By Lemma III.2, the total running time to serve all queries is $O\big(m + (n - \deg^+(s))\ell^2 \log^3(n)/\epsilon^4\big)$.

A factor-2 overestimate ℓ can be obtained by enumerating powers of 2 between 1 and $2k$, and the running time is dominated by the maximum choice of ℓ. ∎

C. Rooted vertex mincut for large sink components

The third and final part (before the overall algorithm) is an approximation for the rooted vertex cut that is well-suited for large sink components.

Lemma III.4. *Let $G = (V, E)$ be a directed graph with positive vertex weights. Let $s \in V$ be a fixed root vertex. Let $\epsilon \in (0,1)$, $\kappa > 0$, and $k \in \mathbb{N}$ be given parameters. There is a randomized algorithm that runs in $\tilde{O}\big(m + (n - \deg^+(s))\big(n/\epsilon^2 + n^{1.5}/k\big)\big)$ time and has the following guarantee. If there is a vertex s-cut of capacity at most κ and where the sink component has at most k vertices, then with high probability, the algorithm returns a vertex (s,t)-cut of capacity at most $(1+\epsilon)\kappa$.*

Proof: Let T^* be the sink component of the minimum s-cut subject to $|T^*| \leq k$. We assume the capacity of the s-cut induced by T^* is at most κ. (Otherwise the output is not well-defined.) Let $V' = V \setminus (\{s\} \cup N^+(s))$ and note that $|V'| < n - \deg^+(s)$.

We apply Lemma III.1 to produce the graph G_0. Lemma III.1 succeeds with high probability and for the rest of the proof we assume it was successful. (Otherwise the algorithm fails.) We sample $O\big(\big(n - \deg^+(s)\big) \log(n)/k\big)$ vertices $t \in V'$. For each sampled t, we compute the minimum (s,t)-vertex cut in G_0. With high probability, some t will be drawn from the sink component of the true minimum s-cut, in which case the minimum (s,t)-cut in G_0 gives an $(1+\epsilon)$-approximate s-cut in G (by Lemma III.1). We use the $\tilde{O}\big(m + n^{1.5}\big)$ time vertex-capacitated flow algorithm [29]. By Lemma III.1, we have $m = O\big(nk \log(n)/\epsilon^2\big)$. This gives the total running time. ∎

D. Approximating the rooted and global vertex mincut

We now combine the two parameterized approximation algorithms for rooted vertex mincut to give the following overall algorithm for rooted vertex mincut. This establishes one part of I.3 concerning approximate rooted vertex cuts. Due to space constraints, the proof is deferred to the full version.

Theorem III.5. *Let $\epsilon \in (0,1)$, let $G = (V, E)$ be a directed graph with polynomially bounded vertex weights, and let $s \in V$ be a fixed root. A $(1+\epsilon)$-approximate minimum vertex s-cut can be computed with high probability in $\tilde{O}\big(m + n(n - \deg^+(s))/\epsilon^2\big)$ randomized time.*

Next we use the algorithm for rooted vertex mincut to obtain an algorithm for global vertex mincut and establish Corollary III.6. [17] showed that running times of the form $(n - \deg^+(s))T$ for rooted mincut from a root s imply a randomized nT expected time algorithm for global vertex mincut. Theorem III.5 gives a $\tilde{O}\big(m + n(n - \deg^+(s))/\epsilon^2\big)$ running time, so some modifications have to be made to address the additional $\tilde{O}(m)$ additive factor. This establishes the remaining part of I.3. The proof is deferred to the full version.

Corollary III.6. *For all $\epsilon \in (0,1)$, a $(1+\epsilon)$-approximate minimum weight global vertex cut in a directed graph with polynomially bounded vertex weights can be computed with high probability in $\tilde{O}\big(n^2/\epsilon^2\big)$ expected time.*

Acknowledgement: This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 715672. Nanongkai was also partially supported by the Swedish Research Council (Reg. No. 2019-05622). Cen and Panigrahi were supported in part by NSF Awards CCF-1750140 (CAREER) and CCF-1955703, and ARO award W911NF2110230. Quanrud was supported in part by NSF grant CCF-2129816. Cen and Panigrahi would like to thank Yu Cheng and Kevin Sun for helpful discussions at the initial stages of this project. Quanrud thanks Chandra Chekuri for helpful discussions and feedback.

References

[1] R. Cen, Y. Cheng, D. Panigrahi, and K. Sun. Sparsification of balanced directed graphs. *ICALP*, 2021. 3

[2] R. Cen, J. Li, D. Nanongkai, D. Panigrahi, and T. Saranurak. Minimum cuts in directed graphs via $\sqrt{n}$ max-flows. *CoRR*, abs/2104.07898, 2021. 1

[3] C. Chekuri and K. Quanrud. Faster algorithms for rooted connectivity in directed graphs. *CoRR*, abs/2104.07205, 2021. To appear in ICALP, 2021. 2, 9

[4] J. Cheriyan and J. H. Reif. Directed s–t numberings, rubber bands, and testing digraph k-vertex connectivity. *Comb.*, 14(4):435–451, 1994. 2

[5] J. Edmonds. Edge-disjoint branchings. *Combinatorial algorithms*, 1973. 6

[6] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975. 2

[7] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. 10

[8] S. Forster, D. Nanongkai, L. Yang, T. Saranurak, and S. Yingchareonthawornchai. Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms. In S. Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2046–2065. SIAM, 2020. 2, 3, 9

[9] H. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, 1995. 2, 6

[10] H. Gabow, Z. Galil, T. Spencer, and R. Tarjan. Efficient algorithms for finding minimum spanning tree in undirected and directed graphs. *Combinatorica*, 6:109–122, 06 1986. 6

[11] Z. Galil. Finding the vertex connectivity of graphs. *SIAM J. Comput.*, 9(1):197–199, 1980. 2

[12] Y. Gao, Y. P. Liu, and R. Peng. Fully dynamic electrical flows: Sparse maxflow faster than goldberg-rao. *CoRR*, abs/2101.07233, 2021. 2

[13] P. Gawrychowski, S. Mozes, and O. Weimann. Minimum cut in o(m $\log^2$ n) time. In *ICALP*, volume 168 of *LIPIcs*, pages 57:1–57:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. 3

[14] P. Gawrychowski, S. Mozes, and O. Weimann. A note on a recent algorithm for minimum cut. In *SOSA*, pages 74–79. SIAM, 2021. 3

[15] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988. 1

[16] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms*, 17(3):424–446, 1994. 1, 2

[17] M. R. Henzinger, S. Rao, and H. N. Gabow. Computing vertex connectivity: New bounds from old techniques. *J. Algorithms*, 34(2):222–250, 2000. 2, 8, 11

[18] D. R. Karger. Minimum cuts in near-linear time. *Journal of the ACM (JACM)*, 47(1):46–76, 2000. 2

[19] J. Li, D. Nanongkai, D. Panigrahi, T. Saranurak, and S. Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows, 2021. 2

[20] A. López-Martínez, S. Mukhopadhyay, and D. Nanongkai. Work-optimal parallel minimum cuts for non-sparse graphs. *SPAA*, 2021. 3

[21] Y. Mansour and B. Schieber. Finding the edge connectivity of directed graphs. *Journal of Algorithms*, 10(1):76–85, 1989. 2

[22] N. Megiddo, A. Tamir, E. Zemel, and R. Chandrasekaran. An $o(n \log^2 n)$ algorithm for the k th longest path in a tree with applications to location problems. *SIAM Journal on Computing*, 10, 05 1981. 7

[23] S. Mukhopadhyay and D. Nanongkai. Weighted min-cut: sequential, cut-query, and streaming algorithms. In *STOC*, pages 496–509. ACM, 2020. 3

[24] D. Nanongkai, T. Saranurak, and S. Yingchareonthawornchai. Breaking quadratic time for small vertex connectivity and an approximation scheme. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 241–252, 2019. 2

[25] V. D. Podderyugin. An algorithm for finding the edge connectivity of graphs. *Vopr. Kibern.*, 2:136, 1973. 2

[26] K. Quanrud. Fast approximations for rooted connectivity in weighted directed graphs. *CoRR*, abs/2104.06933, 2021. 1, 5

[27] C.-P. Schnorr. Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM Journal on Computing*, 8(2):265–274, 1979. 2

[28] J. van den Brand, Y. T. Lee, Y. P. Liu, T. Saranurak, A. Sidford, Z. Song, and D. Wang. Minimum cost flows, mdps, and ℓ_1-regression in nearly linear time for dense instances. *CoRR*, abs/2101.05719, 2021. 1, 4

[29] J. van den Brand, Y. T. Lee, D. Nanongkai, R. Peng, T. Saranurak, A. Sidford, Z. Song, and D. Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 919–930. IEEE, 2020. 11

[30] N. E. Young. Randomized rounding without solving the linear program. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, page 170–178, USA, 1995. 3