

Deterministic Almost-Linear-Time Gomory-Hu Trees

Amir Abboud*	Rasmus Kyng ^{†‡}	Jason Li
Weizmann Institute of Science	ETH Zurich	Carnegie Mellon University
amir.abboud@weizmann.ac.il	kyng@inf.ethz.ch	jmli@cs.cmu.edu
Debmalya Panigrahi [§]	Maximilian Probst Gutenberg [‡]	Thatchaphol Saranurak [¶]
Duke University	ETH Zurich	University of Michigan
debmalya@cs.duke.edu	maximilian.probst@inf.ethz.ch	thsa@umich.edu
Weixuan Yuan [‡]	Wuwei Yuan	
ETH Zurich	ETH Zurich	
weyuan@inf.ethz.ch	wuyuan@ethz.ch	

Abstract

Given an m -edge, undirected, weighted graph $G = (V, E, w)$, a Gomory-Hu tree T (Gomory and Hu, 1961) is a tree over the vertex set V such that all-pairs mincuts in G are preserved exactly in T .

In this article, we give the first almost-optimal $m^{1+o(1)}$ -time *deterministic* algorithm for constructing a Gomory-Hu tree. Prior to our work, the best deterministic algorithm for this problem dated back to the original algorithm of Gomory and Hu that runs in $nm^{1+o(1)}$ time (using current maxflow algorithms). In fact, this is the first almost-linear time deterministic algorithm for even simpler problems, such as finding the k -edge-connected components of a graph.

Our new result hinges on two separate and novel components that each introduce a distinct set of de-randomization tools of independent interest:

- a deterministic reduction from the all-pairs mincuts problem to the single-source mincuts problem incurring only subpolynomial overhead, and
- a deterministic almost-linear time algorithm for the single-source mincuts problem.

*This work is part of the project CONJEXITY that has received funding from the European Research Council (ERC) under the European Union’s Horizon Europe research and innovation programme (grant agreement No. 101078482). Supported by an Alon scholarship and a research grant from the Center for New Scientists at the Weizmann Institute of Science. Part of this work was done while visiting INSAIT, Sofia University “St. Kliment Ohridski”.

[†]The research leading to these results has received funding from the starting grant “A New Paradigm for Flow and Cut Algorithms” (no. TMSGI2 218022) of the Swiss National Science Foundation.

[‡]The research leading to these results has received funding from grant no. 200021 204787 of the Swiss National Science Foundation.

[§]Supported in part by NSF grants CCF-1955703 and CCF-2329230. Part of this work was done when the author was visiting the Simons Institute for Theory of Computing and Google Research.

[¶]Supported by NSF Grant CCF-2238138. Partially funded by the Ministry of Education and Science of Bulgaria’s support for INSAIT, Sofia University “St. Kliment Ohridski” as part of the Bulgarian National Roadmap for Research Infrastructure.

Contents

1	Introduction	1
1.1	Related Work	3
2	Overview	5
2.1	Computing Gomory-Hu Trees via SSMC (see Section 6)	5
2.2	SSMC (see Section 5)	7
2.3	Guide Trees (see Section 4)	8
3	Preliminaries	10
4	Guide Trees	12
4.1	Vertex Sparsifier for Steiner Mincuts	13
4.2	Implementation of the Augmented Dynamic Tree Data Structure	24
4.3	Guide trees	29
5	Single-Source Mincuts via Guide Trees	30
5.1	Partial Single-Source Mincuts	31
5.2	Single-Source Mincuts	34
6	Gomory-Hu Tree via Single-Source Mincuts	35
6.1	Expander Decompositions and τ -Connectivity	35
6.2	Minimal Gomory-Hu trees	37
6.3	Detecting the largest τ -Connected Component	40
6.4	Decomposition of $U \setminus \mathcal{C}_{G,U,\tau}(r)$	44
6.5	Decomposition of $\mathcal{C}_{G,U,\tau}(r)$	49
6.6	The Gomory-Hu Tree Algorithm	52
7	References	56
A	Expander Decomposition	61

1 Introduction

For a pair of vertices s, t in a graph G , the (s, t) edge connectivity $\lambda_G(s, t)$ is the value of a minimum (s, t) -cut, or equivalently, that of a maximum (s, t) -flow. Finding the (s, t) edge connectivity for all pairs of vertices in a graph is a fundamental problem. In a seminal result in 1961, Gomory and Hu solved this problem on undirected (weighted) graphs by proposing a tree data structure defined on the vertices of the graph that encodes an (s, t) -mincut for each vertex pair s, t in the graph [GH61]. This is called a Gomory-Hu tree (GHTREE) or cut tree of the graph. Since their introduction, GHTREES have been widely studied in combinatorial optimization [Elm64, HS83, GH86, Has88, Gus90, Ben95, HM95, Har98, Har01, GT01] and have found applications in diverse domains such as image segmentation, computer networks, and optimization (see textbooks such as [AMO93, CCPS97, Sch03] for further discussions on GHTREES).

The classic algorithm by Gomory and Hu from the 1960s is a deterministic reduction to (s, t) -mincut and (using current max-flow algorithms) achieves runtime $nm^{1+o(1)}$ by a divide-and-conquer algorithm of recursion depth up to $n - 1$, that reduces the problem to solving (s, t) -mincut problems on graphs of size $O(m)$ on each level. During the last years, a long line of work [AKT21b, AKT21a, LPS21, Zha21, AKT22, AKL⁺22, ALPS23] has developed a powerful framework that follows this reduction but enforces $\tilde{O}(1)$ recursion depth, resulting in an almost-optimal $m^{1+o(1)}$ time algorithm.

However, all of the above results rely for their improvement of the recursion depth on the insight that for each graph $G = (V, E, w)$, there exists a *pivot* $r \in V$, such that a set of vertex-disjoint (v, r) -mincuts S_v exists, such that for some constant $C > 1$, each cut is of size at most $|V|/C$ and the set of vertices not in these cuts is of size at most $|V|/C$. Since, in fact, at least a $\Omega(1)$ -fraction of the vertices are pivots, it then suffices to sample such a vertex uniformly at random. Once a pivot r is sampled, they then show, using advanced machinery, how to compute disjoint sets of (v, r) -mincuts with the properties described above, again heavily relying on randomization. Notably, the recent results breaking through the Gomory-Hu bound for weighted graphs [AKL⁺22, ALPS23] are based on a sophisticated extension of Karger's tree-packing technique [Kar00] that is infamously difficult to derandomize; in particular, one needs to sample a so-called *guide tree* from a packing of Steiner trees.

The unfortunate consequence is that the state-of-the-art *deterministic* algorithm for the all-pairs mincuts problem remains the one given by Gomory and Hu more than 60 years ago! In fact, even for the simpler (and extensively studied) problem of finding the k -connected components of a graph (which is trivial given a GHTREE), an almost-linear-time deterministic algorithm is not known unless $k = o(\log n)$.

Our Contribution. In this article, we close the gap between the randomized and the deterministic setting (up to subpolynomial factors). We give the first *deterministic* almost-optimal algorithm to all-pairs mincut.

Theorem 1.1. *Given an undirected, weighted graph $G = (V, E, w)$ with polynomially-bounded weights, there is a deterministic algorithm that computes a GHTREE T for G in $m^{1+o(1)}$ time.*

This positively answers the first open question in [ALPS23]. As a corollary, this yields the first deterministic almost-linear time algorithm to compute k -connected components.

Theorem 1.2. *Given an undirected, weighted graph $G = (V, E, w)$ with polynomially-bounded weights and positive integer $k \geq 0$, there is a deterministic algorithm that computes the k -connected components of G in $m^{1+o(1)}$ time.*

Our contribution can be seen as part of a larger push in graph algorithms to understand the deterministic complexity of various fundamental problems. In particular, the global mincut problem, a special case of GHTREE, has been a central problem in this quest, and the deterministic algorithms developed for it [KT15, LP20, Li21, HLRW24] have revealed deep structural properties about graphs that have since inspired various breakthroughs [AKT21b, LNP⁺21, ALPS23]. Seeing that the almost-linear-time (randomized) GHTREE algorithms employ sophisticated augmentations of techniques from the global mincut literature, it is important to investigate whether they can be derandomized as well. We believe that the insights used to obtain Theorem 1.1 further deepen our understanding of the cut structure of graphs, and hope that they can inspire further advancements in the area.

Technical Contribution. We believe that the following two key technical contributions towards proving Theorem 1.1 are of particular interest:

1. To resolve the aforementioned challenge of deterministically finding one of the $\Omega(n)$ “pivots”, we give the structural insight that for the largest integer $k \geq 0$, such that the largest k -connected component C is of size at least $\frac{3}{4}|V|$, we have that *every* vertex in C is a suitable pivot. This yields a radical new approach towards detecting a pivot $r \in V$. It remains to compute k and C as above. We find k by binary search and C by a novel technique that is inspired by the expander-based de-randomizations for global mincut from [KT15, LP20].
2. The second major challenge is that of sampling a guide tree from a Steiner tree packing which, in turn, is computed by a Multiplicative Weight Updates (MWU) framework. A key difficulty is that the support size of the packing vastly exceeds m , which means that the $m^{1+o(1)}$ -time algorithm of [ALPS23] had to operate on an *implicit* packing, as in the Randomized MWU framework [BGS22]. At the end of the algorithm, [ALPS23] samples few trees from the packing, constructs them explicitly via the implicit representation and forms guide trees which can then be used to extract single-source all-mincuts.

Our algorithm instead uses the recent data structure [KMG24], and we show that by augmenting dynamic tree structure [ST81] in a highly non-trivial way, the algorithm can deterministically maintain an implicit representation that is much more convenient to work with. In fact, we show that we can extract all necessary information and build a *vertex sparsifier* using the entire cut information of the packing! We then show that on the *vertex sparsifier*, we can afford to compute a small collection of guide trees that deterministically satisfy the same guarantees that the randomized sample of the packing satisfied with high probability.

We believe that the *vertex sparsifier* and our novel algorithm for its construction are of broad independent interest.

Remarkably, substituting our new algorithm for computing guide trees into the (randomized) algorithm of [ALPS23] gives an almost-linear-time solution for GHTREE that is more modular and could be considered much simpler (see further discussion below). This makes progress on the final open question in [ALPS23] regarding a simple reduction from all-pairs to single-pair for max-flow.

We give a more detailed overview of our deterministic, almost-linear time Gomory-Hu tree algorithm and the new techniques involved in Section 2.

1.1 Related Work

Global MinCut. The objective in the global mincut problem is to return a (s, t) -mincut of smallest value over all $s, t \in V$. Since such a mincut can be extracted from a GHTREE straightforwardly, the global mincut problem is a special case of GHTREES.

For more than a decade, global mincut has served as a prominent example for showing the power of randomness in graph algorithms: a basic problem that has beautiful near-linear time randomized algorithms, namely the Karger-Stein algorithm via edge contractions [KS93] and Karger’s tree packing algorithm [Kar00], beating the best-known deterministic algorithms by a factor of n . A celebrated 2015 paper by Kawarabayashi and Thorup [KT15] gave an $\tilde{O}(m)$ deterministic algorithm to the problem for *unweighted, simple* graphs, while introducing novel expander-based arguments to the literature on mincuts. Later on, the quest for a similar result for weighted graphs [LP20, Li21, HLRW24] has resulted in the invention of other influential techniques such as the *Isolating Cuts* framework [LP20]¹. Notably, both the expander-based arguments and the Isolating Cuts framework were key ingredients in the recent (randomized) results on GHTREE, further supporting the claim that one often discovers valuable insights when derandomizing algorithms.

As mentioned before, the most recent GHTREE algorithms [AKL⁺22, ALPS23] can be seen as augmentations of Karger’s tree packing technique from global mincut to single-source mincuts. Given how productive the derandomization of Karger’s algorithm has been, it is a natural to hope that our work will also have further impact.

Maxflow/Mincut. Another special case of GHTREES is to compute the (s, t) -mincut value $\lambda_G(s, t)$ for a fixed pair $s, t \in V$. It is well-known that this problem is dual to the (s, t) -maxflow problem asking for a flow that sends as many units of a commodity from s to t as possible.

While the problem of finding maxflows/mincuts is one of the most well-studied graph problems, with many seminal results commonly taught in undergraduate- and graduate-level courses. An almost-linear time algorithm has only recently been obtained in 2022 [CKL⁺22]. The algorithm computes maxflows via an interior point method (IPM) that reduces the problem to solving the so-called min-ratio cycle problem on a dynamic graphs. They then present an efficient data structure that solves the min-ratio cycle problem on dynamic graphs generated by the IPM.

In [vBCP⁺23], the algorithm was derandomized by showing that a deterministic data structure can be given that solves the min-ratio cycle problem on dynamic graphs generated by the IPM. Further research has recently achieved a deterministic min-ratio cycle data structure that works even in general dynamic graphs [CKL⁺24]. Besides making proofs more modular, this allowed them to extend the maxflow algorithm to work even in graphs undergoing edge *insertions*. Similarly, a second novel deterministic maxflow algorithm was discovered to obtain an almost-linear total update time in graphs undergoing edge *deletions* [vBCK⁺24]. Thus, again in the context of flow algorithms, the interplay between determinism and broader algorithmic goals has been very fruitful.

Notably, all algorithms for GHTREE reduce to solving maxflow problems, and so the times stated in the introductory section are implicitly using the recent progress for the maxflow problem. In fact, only since the first deterministic algorithm in [vBCP⁺23] was given, the time required to compute GHTREE via the classic algorithm from Gomory and Hu decreased to $nm^{1+o(1)}$.

¹Independently discovered by [AKT21b].

k -Connected Components. In the k -connected component problem, the objective is to find the coarsest possible partition \mathcal{C} of the vertex set V of graph G such that each partition set $C \in \mathcal{C}$ and $u, v \in C$ has $\lambda_G(u, v) \geq k$, or alternatively, that there is a set of at least k edge-disjoint paths between u and v in G . This problem can be solved by constructing a Gomory-Hu tree and identifying the k -edge-connected components in the tree by removing edges of weight less than k .

For $k = 1$, the problem is trivial: simply use BFS or DFS to report the connected components of the graph in $O(m + n)$ time. But, somewhat surprisingly, the problem already becomes non-trivial when $k = 2$. Starting with the seminal work of Tarjan [Tar72] in 1972, there has been a large body of literature obtaining fast algorithms for $k \geq 2$ [HT73a, Pat71, HT73b, NI92, Tsi07, Tsi09].² In fact, the first linear-time algorithm for $k = 4$ was obtained as recently as 2021 [NRSS21, GIK21] and that for $k = 5$ in 2024 [Kos24]. Finally, the case of arbitrary k was addressed a few months back in the work of Korhonen [Kor24], who gave an $O(m) + k^{O(k^2)}n$ -time algorithm for the problem. Clearly, this algorithm is inefficient when k is super-constant, even for $k = O(\log n)$. In parallel to this line of work, Bhalgat *et al.* [BHKP07] (see also [HKP07]) gave an $\tilde{O}(mk)$ -time algorithm for this problem, based on what they called a *partial* Gomory-Hu tree. Their result has the advantage that it can be applied for super-constant values of k , but the algorithm is randomized. In summary, we note that in spite of much research on the problem over many decades, there is no deterministic algorithm for finding the k -edge-connected components of a graph that runs in (almost) linear time, for general k .

Previous Gomory-Hu tree algorithms. Prior to the aforementioned line of work, near-linear time GHTREE algorithms were known for restricted graph families such as planar graphs [HM94, ACZ98, BSW15], surface-embedded graphs [BENW16], and bounded treewidth graphs [ACZ98]; these works did not use randomization at all. The randomized pivot selection idea was introduced by Bhalgat *et al.* [BHKP07] inside their $O(mn)$ -time algorithm for unweighted graphs. Subsequently, Abboud, Krauthgamer, and Trabelsi [AKT20] capitalized on it to give a general reduction from GHTREE to single-source min-cuts, which was used in all subsequent developments. As explained below, in this work, we make this reduction deterministic. To our knowledge, the only previous result along these lines was by Abboud, Karuthgamer, and Trabelsi [AKT21a], who put forth the *dynamic pivot* technique and used it to derandomize their quadratic-time algorithm for simple graphs. Rather than using structural insights to lead the algorithm to a good pivot, their method is based on an efficient algorithm for switching from one pivot to another (when the former is revealed as being bad). Unfortunately, their technique does not seem general enough to be compatible with the breakthroughs for *weighted* graphs; i.e. its applicability depends on the properties of the algorithm for single-source min-cuts. Moreover, it seems to inherently incur a quadratic factor and is therefore insufficient for getting almost-linear time bounds. Finally, we would like to remark that the randomized pivot idea is also used in a recent paper aiming to make the recent developments more simple and practical [Kol22]. (We refer to the survey [Pan16] for more background on GHTREE and Table 1 for the state-of-the-art GHTREE algorithms in different settings.)

²Technically, the algorithms in [Tar72] and [HT73a] return k -vertex-connected components for $k = 2, 3$. But, k -edge-connected components can also be obtained via these algorithms (e.g., see [GI91]).

Reference	Restriction	Det./Rand.	Time
Gomory and Hu [GH61]	Weighted	Det.	$O(n) \cdot T_{maxflow}(n, m)$
Abboud, Krauthgamer, and Trabelsi [AKT21a]	Simple unweighted	Det.	$n^{2+o(1)}$
Abboud, Krauthgamer, Li, Panigrahi, Saranurak, and Trabelsi [AKL ⁺ 22]	Unweighted	Rand.	$m^{1+o(1)}$
Abboud, Li, Panigrahi, and Saranurak [ALPS23]	Weighted	Rand.	$n^{1+o(1)} + \tilde{O}(1) \cdot T_{maxflow}(n, m)$
This work	Weighted	Det.	$m^{1+o(1)}$

Table 1: The state-of-the-art algorithms for GHTREE. $T_{maxflow}(n, m)$ denotes the running time to compute (s, t) -maxflow in a graph with n vertices and m edges, where in sparse graphs [CKL⁺22] gives $T_{maxflow}(n, m) = m^{1+o(1)}$, while in moderately dense graphs [vdBLL⁺21] gives $T_{maxflow}(n, m) = \tilde{O}(m + n^{1.5})$.

2 Overview

In this overview, we give a top-down description of our deterministic algorithm to compute a Gomory-Hu tree in a weighted graph in almost linear deterministic time. We start by giving a reduction from Gomory-Hu trees to solving the problem of *single-source mincuts* (SSMC) in Section 2.1. We then give a reduction that reduces the SSMC problem to the problem of finding *guide trees* in Section 2.2. Finally, we show how to construct guide trees efficiently in Section 2.3. The technical part of this article is organized oppositely, i.e. in a bottom-up fashion.

2.1 Computing Gomory-Hu Trees via SSMC (see Section 6)

As in all previous algorithms for Gomory-Hu trees, we work with a terminal version where we compute the Gomory-Hu tree of G only over a set of terminals $U \subseteq V$. This is convenient for computing a Gomory-Hu tree recursively (where artificial vertices are added).

Definition 2.1 (Gomory-Hu U -Steiner Tree). *Given a graph $G = (V, E, w)$ and a set of terminals $U \subseteq V$, the Gomory-Hu U -Steiner tree is a weighted tree T on the vertices U , together with a function $f : V \mapsto U$ such that:*

- for all $s, t \in U$, consider any minimum-weight edge (u, v) on the unique st -path in T . Let U' be the vertices of the connected component of $T \setminus (u, v)$ containing s . Then, $f^{-1}(U') \subseteq V$ is an (s, t) -mincut, and its value is $w_T(u, v)$.

The original algorithm by Gomory and Hu. We next describe how to compute a U -Steiner Gomory-Hu tree in deterministic almost linear time. We follow the basic divide-and-conquer already employed in the original paper by Gomory and Hu [GH61]: first for vertices $s, t \in U$, we find an (s, t) -mincut $S \subseteq V$ (we use one side of the cut to identify it). Let G_S be the graph obtained from G contracting $V \setminus S$ into an artificial vertex. Then, find the Gomory-Hu $(U \cap S)$ -Steiner tree T_S on G_S . Equivalently, for $T = V \setminus S$, find a Gomory-Hu $(U \cap T)$ -Steiner tree T_T on G_T . The U -Steiner

Gomory-Hu tree is then obtained from joining T_S and T_T by an (s, t) edge with value equal to the (s, t) -mincut value $\delta_G(S) = \lambda_G(s, t)$. Correctness of this algorithm follows from the insight that mincuts are *nesting*, that is for any $x, y \in U$, we can find an (x, y) -mincut X such that:

- if $x, y \in S$, then $X \subseteq S$, and
- if $x, y \in T$, then $X \subseteq T$, and
- if $x \in S, y \in T$ (or vice versa), then either $X \subseteq S$ or $X \subseteq T$. In fact, either $X = S$ or X is also an (s, x) -mincut or a (y, t) -mincut.

In the above algorithm, it can be ensured that the graphs at the same recursion level have a total of $O(m)$ edges. Thus, for recursion depth d , the algorithm runs in time $d \cdot m^{1+o(1)}$ using that an (s, t) -mincut can be found in deterministic time $m^{1+o(1)}$ on an m -edge graph. Unfortunately, the recursion depth can only be bounded by $O(n)$, which only yields an $nm^{1+o(1)}$ time algorithm.

Gomory-Hu trees in almost linear time. While a first guess towards improving the algorithm above is to search for an (s, t) -mincut that is relatively balanced to improve the recursion depth, there are graphs that do not have any such mincuts (for example the star graph).

However, in [BHKP07, AKT20] it was observed that for some constant $C > 1$, one can find a *pivot* $r \in U$, together with a set \mathcal{S} consisting of disjoint (v, r) -mincuts S_v such that each S_v contains at most $|U|/C$ terminals, and $V \setminus \cup_{v \in \mathcal{S}} S_v$ also only contains at most $|U|/C$ terminals. The disjointness of the sets $S_v \in \mathcal{S}$ implies that one can simulate multiple steps of the original Gomory-Hu algorithm in sequence using \mathcal{S} at cost $O(m)$, and the recursive subproblems at this point have decreased by a constant fraction in size.

In this article, we give the first almost linear time deterministic algorithm to find such a pivot r and family of disjoint (v, r) -mincuts as summarized above. By our discussion so far, this implies Theorem 1.1.

Informal Lemma 2.1 (Decomposition Lemma (see Lemma 6.28)). *There is an algorithm returns a pivot $r \in U$ and a set \mathcal{S} in deterministic $m^{1+o(1)}$ time such that*

1. *every triple $(S_v, v, r) \in \mathcal{S}$ has $v \in U \setminus \{r\}$ and S_v is a (v, r) -mincut,*
2. *the cuts S_v in \mathcal{S} are disjoint,*
3. *and for every cut S_v , we have $|S_v \cap U| = (1 - \Omega(1))|U|$, and the remaining cut $V_{rest} = V \setminus \bigcup_{(S_v, v, r)} S_v$ has $|V_{rest} \cap U| = (1 - \Omega(1))|U|$.*

The decomposition lemma. For the rest of this section, we sketch the ideas required to obtain Informal Lemma 2.1. We start by discussing how to find a pivot vertex $r \in U$. Note that it is easy to construct graphs with vertex r such that any family of disjoint (v, r) -mincuts \mathcal{S} cannot satisfy the guarantees in Informal Lemma 2.1: consider the graph obtained from the union of K_{n-1} and additional vertex r attached with a single edge to some arbitrary vertex in K_{n-1} ; let all vertices be terminals; clearly, $\{r\}$ and $V(K_{n-1})$ are the only (v, r) -mincuts for any $v \in V$, but these cuts are not balanced.

In [BHKP07, AKT20], it was observed that an $\Omega(1/C)$ -fraction of the vertices of U in any graph are suitable pivots, allowing them to simply sample for a pivot uniformly at random among the terminals U .

Since we cannot exploit the same trick here, we instead exploit a new structural insight that was not observed so far: let τ^* be the largest positive integer such that some τ^* -connected component C of G that has $|C \cap U| \geq \frac{3}{4}|U|$. Then, it turns out, that each vertex $r \in C_U = C \cap U$ is a suitable pivot (see Claim 6.24).

This reduces the problem of detecting a pivot, to finding the threshold τ^* and the τ^* -connected component C . To solve this problem, we binary search over values τ to find τ^* . To this end, we design an algorithm that finds, for $\tau \leq \tau^*$ where C'' is the τ -connected component with $|C'' \cap U| \geq \frac{3}{4}|U|$, a set X of subpolynomial size such that $X \cap C''_U \neq \emptyset$. Then, we take each vertex $w \in X$, find all values $\lambda_G(w, v)$ for $v \in V$ using an SSMC data structure with source w and compute the τ -connected component $C_w = \{v \in V \mid \lambda_G(v, w) \geq \tau\}$ containing w . We can then explicitly check if $|C_w \cap U| \geq \frac{3}{4}|U|$ which implies $C_w = C$. We show in the next section that the SSMC problem can be solved in deterministic almost linear time. Hence, computing the (w, v) -mincut values $\lambda_G(w, v)$ can be done in almost linear time for all vertices in the subpolynomially-sized set X .

We further show that X can be identified deterministically by initializing the set of active terminals A by U , and while A is not subpolynomially-sized, iteratively an expander decomposition w.r.t. A is computed and in each cluster X of the expander decomposition, we delete half the vertices in $X \cap A$ from A . This vastly generalizes the techniques from [LP20], where it was shown that this strategy succeeds when τ is the value of the global mincut.

Finally, we pick an arbitrary vertex $r \in C_U$, and show that the pivot detection algorithm from above can be used with the isolating cut lemma from [LP20] to compute a set of disjoint (v, r) -mincuts \mathcal{S} , as described in Informal Lemma 2.1.

2.2 SSMC (see Section 5)

As mentioned above, we also give the first deterministic almost linear time algorithm for SSMC.

Theorem 2.2 (Single Source Mincuts). *Given an undirected weighted graph $G = (V, E, w)$ and a source terminal s , we can compute $\lambda(s, t)$ for all vertices $t \neq s$ in deterministic time $m^{1+o(1)}$.*

In this article, we follow closely the randomized algorithm by [Zha21] (which is heavily inspired by [AKL⁺22], which is in turn a clever generalization of Karger's algorithm [Kar00]). This algorithm reduces the SSMC problem to computing a set of guide trees. Let us briefly define guide trees.

Definition 2.3 (Guide trees). *For graph $G = (V, E, w)$, we say a cut $A \subseteq V$ is k -respected by a tree T (with $V(T) \subseteq V(G)$) if there are at most k edges in T with exactly one endpoint in A .*

For terminals $U \subseteq V$ with source $s \in U$, a collection of trees \mathcal{T} is called k -respecting set of guide trees, if for every $t \in U \setminus \{s\}$ with $\lambda_G(s, t) \leq 1.1\lambda_G(U)$ ³, some tree $T \in \mathcal{T}$, k -respects some (s, t) -mincut.

The main idea of the reduction is to recursively construct a $(k - 1)$ -respecting set of guide trees \mathcal{T}' from a k -respecting set \mathcal{T} (or extract the (s, t) -mincut value by rather direct computations). While this reduction is randomized in [Zha21], we show that it can be de-randomized using standard techniques. Since the blow-up factor in each such step is constant, we can only allow for the initial collection of guide trees to be k -respecting for k being a constant.

Showing that such a collection of guide trees can be computed deterministically in almost linear time poses a central challenge (especially for weighted graphs!) that is addressed in the next section.

³ $\lambda_G(U) = \min_{x, y \in U} \lambda_G(x, y)$.

2.3 Guide Trees (see Section 4)

Finally, we sketch the ideas behind our last and technically most involved component: a deterministic algorithm to compute guide trees.

Theorem 2.4. *Given an undirected weighted graph $G = (V, E, w)$, a terminal set $U \subseteq V$, and a source terminal s , we can compute in deterministic time $m^{1+o(1)}$ a 16-respecting set of guide trees \mathcal{T} with size $|\mathcal{T}| = n^{o(1)}$. Moreover, all trees in \mathcal{T} have vertex set U .*

A Review of randomized Techniques. The central contribution of [AKL⁺22] was to define and show the existence of good guide trees (and to construct them in randomized almost linear time). They show that:

- There is a fractional \mathcal{P} of U -Steiner trees T_1, T_2, \dots, T_k , that are trees that span the terminal set but not necessarily the entire vertex set of G , such that sampling $O(\log n)$ trees (proportional to their fractional values), yields a collection of $O(1)$ -respecting guide trees.
- An (approximate) packing \mathcal{P} can be found by using the multiplicative weights update (MWU) framework which reduces the problem to $m^{1+o(1)}$ calls to find the U -Steiner tree T in G of minimum length w.r.t. a length function ℓ (given by the MWU) that is monotonically increasing in each coordinate, and undergoes $m^{1+o(1)}$ updates.
- That the minimum-length U -Steiner tree problem can be solved via a decremental single-source shortest paths (SSSP) data structure by dynamizing Mehlhorn's algorithm [Meh88].

However, the above reduction glosses over many technical complications that one has to overcome, especially of a weighted graph: even an approximate fractional packing \mathcal{P} has to consist of at least $\Omega(m)$ trees which implies that the trees cannot be stored explicitly (for $U = V$, the support size would be $\Omega(mn)$). Naturally, one would hope that instead the trees T_1, T_2, \dots, T_k that are packed as such that T_i and T_{i+1} differ only in $m^{o(1)}$ edges on average such that the sequence of changes $\Delta_2 = T_2 \setminus T_1, \Delta_3 = T_3 \setminus T_2, \dots, \Delta_k = T_k \setminus T_{k-1}$ can be written down explicitly.

But instead, [ALPS23] shows that it can determine if an edge e is used in T_i and report it with some pre-defined probability p_e . Setting this probability reasonably small, they show that it suffices to report to the MWU the edge e if sampled and report that it was used roughly p_e times⁴. Since they only need to subsample $O(\log n)$ trees for the guide tree collection \mathcal{T} in the end, they then use a lot of query time to construct the (implicitly) sampled trees explicitly.

Note that our description above is still an extreme over-simplification, the data structure to implement the MWU algorithm requires over 50+ pages in [ALPS23]. A particular detail is that for technical reasons, the packing is for subgraphs H_i instead of trees T_i . This is also the case for our algorithm, we will, however, ignore this technical detail henceforth.

Computing the packing deterministically. In this article, we show that indeed, the tree T_1 and the difference sequence $\Delta_2, \Delta_3, \dots, \Delta_k$ can essentially be outputted explicitly!

To achieve this goal, we show that using the decremental SSSP data structure from [KMG24] in-lieu of the data structure [BGS22] that was used in [ALPS23] crucially simplifies the problem due to the way that the SSSP tree is maintained internally.

⁴See also [BGS22] for the details of this technique.

This yields the following sub-problem to be addressed: while the SSSP tree M is explicitly outputted by the data structure and consists of edges in G , we could hope to glean the difference sequence from the difference sequence of the shortest paths tree M .

However, Mehlhorn's reduction [Meh88] adapted by [AKL⁺22] does not show M to be the next tree added to the fractional packing but instead only M^U , that is, the U -Steiner subtree of M . We show, however, that for dynamic forest M undergoing Δ updates, we can isolate M^U using only $\tilde{O}(\Delta)$ updates.

Informal Lemma 2.2 (Theorem 4.11). *There is a deterministic algorithm \mathcal{M} that, given a forest M undergoing batches of edge insertions/deletions, and a terminal set $U \subseteq V(M)$ satisfying U is connected at all times, maintains a forest $R \subseteq M$ containing M^U as a connected component such that each edge update to M results in $O(\log n)$ edge updates to R and takes $O(\log^2 n)$ time.*

In fact, as sketched in the next paragraph, we can implement Informal Lemma 2.2 by extending relatively standard techniques for dynamic tree data structures.

Maintaining the U -Steiner Subtree in a forest via augmented dynamic trees. Clearly, we cannot explicitly maintain the subtree M^U in a separate data structure as an edge insertion or deletion to M may lead to $\Omega(n)$ edge insertions or deletions to M^U : if $U = \{s, t\}$ and M consists of s, t and P_n , then M could toggle between an edge (s, t) existing, and s being attached to the first vertex, and t to the last vertex on the path, changing M^U by $\Omega(n)$ edges every constant many updates to M .

In our implementation, we follow the approach of [ST81] to decompose M . Following their nomenclature, we refer to edges that are in $R \subseteq M$ as *solid* and edges in $M \setminus R$ as dashed. Now consider processing edge updates to M (from the batch of updates, so U might be temporarily disconnected).

For an edge (u, v) deletion, it suffices to find the last edge on the intersection of all paths from u (respectively v) to every terminal u' (v') in the subtree of u (v) (after removing the edge (u, v) from M) and to make it dashed. For an edge insertion (u, v) , where (u, v) now appears on a $u'v'$ -path in M for u' (v') being a terminal in the subtree of u (v), we need to convert each edge on the $u'v'$ -path to a solid edge. Potentially, there are many dashed edges on such a path. However, by using arguments similar to the analysis in [ST81], we can show that this operation turns at most $O(\log n)$ from dashed to solid on average.

Extracting a vertex sparsifier from the packing (via Euler tour trees). Ideally, we would next use the computed packing \mathcal{P} , implicitly represented by the difference sequence computed above, directly to extract few guide trees.

While it is not clear how to do so directly, we achieve this by first extracting a vertex sparsifier from the packing, and then showing that a set of guide trees can be found in the vertex sparsifier. The guarantees of our vertex sparsifier are summarized in the informal lemma below.

Informal Lemma 2.3 (Theorem 4.1). *There is an algorithm that, given graph $G = (V, E, w)$ and terminal set $U \subseteq V$, constructs a graph $G' = (U, E_{G'}, w_{G'})$ in $m^{1+o(1)}$ time such that,*

1. *Let A be a cut in G that crosses U . If $w_G(\partial_G A) = O(\lambda(U))$, then $A \cap U$ is an approximate (i.e. up to a constant) global mincut of G' .*

2. G' has at most $m^{1+o(1)}$ edges.

Consider first the following (inefficient) algorithm to extract a vertex sparsifier given the trees T_1, T_2, \dots, T_k from the packing (associated with values $\lambda_1, \lambda_2, \dots, \lambda_k$):

1. for each tree T_i , find an Euler tour of T_i generated as follows: root the tree at an arbitrary vertex r , run a DFS from the root, and write down the ID of a vertex whenever it is visited in the DFS. This generates a sequence of vertex visits $r, v_1, v_2, \dots, v_x, r$ where v_i might be equal to v_j even for $i \neq j$. The Euler tour is the non-simple cycle $C_i = (r, v_1), (v_1, v_2), \dots, (v_{x-1}, v_x), (v_x, r)$. Finally, recursively contract on this cycle every edge that is not incident to two terminals in U . One can then show that C_i (after scaling by $\lambda_i/2$) is a vertex sparsifier of T_i .
2. Let G' be the direct sum of cycles C_i .

Naturally, the algorithm above is not sufficient for our purpose as it requires going over the supports of T_1, T_2, \dots, T_k which might be too large which causes large runtime, and does not guarantee the size bound on the number of edges in G' .

However, we maintain the forest R as in Informal Lemma 2.2 in a dynamic Euler tour data structure described in [HK95]. This data structure makes it explicit that as the underlying tree is undergoing Δ updates, its corresponding Euler tour only undergoes $O(\Delta)$ updates. This allows us to merge the step 1 across trees and bundles multi-edges in G' implicitly to reduce the support of G' to the desired $m^{1+o(1)}$.

Guide Trees via Vertex Sparsifiers Finally, let G' be a vertex sparsifier returned by Informal Lemma 2.3. It remains to extract the set of guide trees from G' . We will rely on a modified version of Gabow's tree packing algorithm ([Kar00], [Gab91]). Let H be an unweighted graph with m_H edges and global edge connectivity λ_H . Applying Gabow's algorithm on H returns a feasible tree packing on H with size λ_H and packing value $\lambda_H/2$ in deterministic time $\tilde{O}(\lambda_H m_H)$. By the first property of G' in Informal Lemma 2.3, we can prove that a tree packing of G' with value $\Omega(\lambda_{G'})$ is our desired guide tree set via a straightforward argument (see the end of Section 4). However, we cannot simply apply Gabow's algorithm to G' since G' is weighted and $\lambda_{G'}$ may not be subpolynomial. To address this issue, we apply the algorithm in Theorem 1.3 of [Li21] to G' to construct a skeleton graph H on U , which preserves a property similar to the first point in Informal Lemma 2.3 (Theorem 4.22) with a desirable edge connectivity $\lambda_H = n^{o(1)}$. Therefore, applying Gabow's tree packing algorithm on H yields $n^{o(1)}$ guide trees defined in Definition 2.3.

3 Preliminaries

Graphs. We denote graphs as tuples $G = (V, E)$ where V is the vertex set, E is the edge set. In this article all graphs are undirected. For a given graph G , we denote by n_G the number of vertices in G , by m_G the number of edges.

For any subset $A \subset V$, we use $G[A]$ to denote the induced graph and $E[A]$ to denote the set of edges with both endpoints in A , i.e. $E[A] = \{\{u, v\} \in E, u, v \in A\}$. For any subsets $A, B \subset V$, we denote by $E(A, B)$ the set of edges in E with one endpoint in A and the other in B . We define the shorthand $\partial A = E(A, A^c)$.

Given $G = (V, E)$, we reserve w_G to refer to the weight function of G that maps edges in G to integers in $[1, W_G]$ for some integer W_G polynomially-bounded in m . We reserve ℓ_G to denote the length function of G mapping edges to integers in $[1, L_G]$. For any set of edges $E' \subseteq E$, we extend w_G (respectively ℓ_G) to $w_G(E') = \sum_{e \in E'} w_G(e)$ ($\ell_G(E') = \sum_{e \in E'} \ell_G(e)$), and sometimes write $G = (V, E, w_G)$ (respectively $G = (V, E, \ell_G)$) to stress that the graph is endowed with a weight (length) function.

For $G = (V, E)$ endowed with length function ℓ_G , we define the distance $dist_G(u, v)$ to be the minimum length of any uv -path.

We often omit the subscript G when it is clear from the context.

Contractions. For a graph $G = (V, E, w)$ and vertex set $S \subseteq V$, we denote by G/S , the graph obtained from G by contracting all vertices in S into a single super-vertex. We extend this notion to any collection $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ of vertex-disjoint subsets of V , we denote by G/\mathcal{S} the graph $((G/S_1)/S_2) \dots /S_k$, i.e. the graph obtained from G by contracting the vertices in each set in \mathcal{S} into a single vertex.

Partition Sets. For any partition \mathcal{X} of some ground set X , we denote by $\mathcal{X}(x)$ the partition set in \mathcal{X} that contains the item $x \in X$.

Cuts and Connectivity. For $G = (V, E, w)$ and cut $S \subseteq V$, we denote by $\partial_G(S)$ the set of edges in G with exactly one endpoint in S . We call $w(\partial_G(S))$ the value of the cut and often use the shorthand $\delta_G(S) = w(\partial_G(S))$. When the context is clear, we often write $\partial_G S$ instead of $\partial_G(S)$ and $\delta_G S$ instead of $\delta_G(S)$.

We say S is an (A, B) -cut for $A, B \subseteq V$ if either $A \subseteq S$ and $B \cap S = \emptyset$ or $B \subseteq S$ and $A \cap S = \emptyset$. We denote by $\lambda_G(A, B)$ the connectivity of A and B that is the minimum value over all (A, B) -cuts. We say that an (A, B) cut is an (A, B) -mincut if the value of the cut is minimal, i.e. equals $\lambda_G(A, B)$. We say that an (A, B) -mincut S is vertex-minimal if $A \subseteq S$ and there is no other (A, B) -mincut $A \subseteq S'$ with $|S'| \leq |S|$. When A, B are singleton sets, i.e. $A = \{a\}, B = \{b\}$, we often simply write (a, b) -cut, (a, b) -mincut and vertex-minimal (a, b) -mincut instead of (A, B) -cut, (A, B) -mincut and vertex-minimal (A, B) -mincut. Note that while (A, B) -mincuts are symmetric in A, B , minimal (A, B) -mincuts are not!

For a set of vertices $U \subseteq V$, we write $\lambda_G(U) = \min_{s, t \in U} \lambda_G(s, t)$, and we write $\lambda_G = \lambda_G(V)$.

Connected Components. Given graph $G = (V, E, w)$, we reserve $\mathcal{C}_{G, \tau}$ to denote the τ -connected components of G , i.e. for every $X, Y \in \mathcal{C}_{G, \tau}, X \neq Y$, we have for all $u, v \in X, x \in Y$ that $\lambda_G(u, v) \geq \tau$ and $\lambda_G(u, x) < \tau$. For $U \subseteq V$, we denote by $\mathcal{C}_{G, U, \tau}$ the τ -connected components of G w.r.t. U , i.e. $\mathcal{C}_{G, U, \tau} = \{C \cap U \mid C \in \mathcal{C}_{G, \tau}\}$.

Submodularity/ Posi-Modularity of Cuts. Given a finite set S , a function $f : 2^S \mapsto \mathbb{R}$ is *submodular* if for all $A, B \subseteq S$, $f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$. Alternatively, f is submodular if for all $A \subseteq B \subseteq S$ and $s \in S$, $f(A + s) - f(A) \geq f(B + s) - f(B)$. If f is further symmetric, i.e. for all $A \subseteq S$, $f(A) = f(S \setminus A)$, then it is *posi-modular* which implies that $f(A) + f(B) \geq f(A \setminus B) + f(B \setminus A)$. It is well-known (and easy to prove) that the value of cuts (i.e. δ_G) is a posi-modular function.

Connectivity Facts. We state some facts about connectivity that relate to Gomory-Hu trees for general use throughout the paper. These facts are standard and essentially follow from submodularity and pos-modularity.

Fact 3.1. [see [GH61]] For graph $G = (V, E, w)$ and any vertices $s, t \in V$, let S be an (s, t) -mincut. For any vertices $u, v \in S$, there is a (u, v) -mincut U with $U \subseteq S$.

Fact 3.2. For graph $G = (V, E, w)$ and disjoint sets $A, B \subseteq V$, the minimal (A, B) -mincut is unique.

We henceforth denote by $M_{G,A,B}$, the unique minimal (A, B) -mincut in G ; by $M_{G,U,A,B}$ we denote $M_{G,A,B} \cap U$. We define $m_{G,A,B} = |M_{G,A,B}|$ and $m_{G,U,A,B} = |M_{G,U,A,B}|$. Again, we overload notation to allow vertex a in-lieu of A if A is a singleton set containing only a ; analogously for B .

Fact 3.3. For graph $G = (V, E, w)$, and fixed vertex $r \in V$, the minimal (v, r) -mincuts $M_{G,v,r}$ for all $v \in V \setminus \{r\}$, form a laminar family, i.e. for any two such cuts, either one is contained in the other, or their intersection is empty. Similarly, for any set $U \subseteq V$, the sets $M_{G,U,v,r}$ for all $v \in V \setminus \{r\}$ form a laminar family.

Isolating Cuts Lemma. The following Lemma is stated as given in [AKL⁺22] but was first introduced by [LP20] (an alternative proof is given in [AKT21b]). While all of these articles state the Lemma as a reduction to maximum flow on graphs of total size $\tilde{O}(m)$, we directly state the implied bound when using the deterministic almost-linear time maximum flow algorithm given in [vBCP⁺23].

Lemma 3.4 (Isolating Cuts Lemma). *There is an algorithm COMPUTEISOLATINGCUTS that takes as input a graph $G = (V, E, w)$ and a collection \mathcal{U} of disjoint terminal sets $U_1, U_2, \dots, U_h \subseteq V$. The algorithm then computes a collection of disjoint cuts S_1, S_2, \dots, S_h where for each $1 \leq i \leq h$, S_i is a vertex-minimal $(U_i, \cup_{j \neq i} U_j)$ -mincut. The algorithm is deterministic and runs in time $m^{1+o(1)}$.*

Hit-And-Miss Theorems. We use the following hit-and-miss theorem that is a special case of a theorem from [CP24].

Theorem 3.5 (see Definition 4 and Theorem 3.1 in [CP24]). *For any positive integers N, a, b with $a \geq b$ and $b = O(1)$, there is an algorithm CONSTRUCTHITANDMISSFAMILY($[N], a, b$) that constructs a family \mathcal{H} of hit-and-miss hash functions such that for any $A \subseteq [N]$ with $|A| \leq a$ and $B \subseteq [N] \setminus A$ with $|B| \leq b$, there is some function $h \in \mathcal{H}$ such that $h(x) = 0$ for all $x \in A$ and $h(y) = 1$ for all $y \in B$. The algorithm constructs \mathcal{H} to be of size $(a \cdot \log(N))^{O(b)}$ in deterministic time $N \cdot (a \cdot \log(N))^{O(b)}$. Moreover, when $a = O(1)$, the size of \mathcal{H} is $O(\log N)$.*

4 Guide Trees

In this section, we give the first almost-linear time deterministic construction of guide trees. The formal guarantees of our result are summarized in the statement below.

Theorem 2.4. Given an undirected weighted graph $G = (V, E, w)$, a terminal set $U \subseteq V$, and a source terminal s , we can compute in deterministic time $m^{1+o(1)}$ a 16-respecting set of guide trees \mathcal{T} with size $|\mathcal{T}| = n^{o(1)}$. Moreover, all trees in \mathcal{T} have vertex set U .

4.1 Vertex Sparsifier for Steiner Mincuts

To compute guide trees efficiently, we give the first almost-linear time deterministic construction of vertex sparsifiers for U -Steiner mincuts (that preserve cuts close in value to the mincut of smallest value separating any terminals in U).

Theorem 4.1. *There is an algorithm that, given graph $G = (V, E, w)$ and terminal set $U \subseteq V$, constructs a graph $G' = (U, E_{G'}, w_{G'})$ in $m^{1+o(1)} \log W$ time such that,*

1. $\lambda_{G'} \geq \lambda_G(U)/4.1$;
2. For any cut (A, B) of G , $w_{G'}(A \cap U, B \cap U) \leq w_G(A, B)$;
3. G' has at most $m^{1+o(1)} \log W$ edges.

Steiner-Subgraph Packings. We obtain the vertex sparsifiers by packing U -steiner subgraphs. We define packings of such subgraphs below.

Definition 4.2 (c.f. Definition 3.1 in [AKL⁺22]). *Let $G = (V, E, w)$ be an undirected weighted graph with a set of terminals $U \subseteq V$. A subgraph H of G is said to be a U -Steiner subgraph (or simply a Steiner subgraph if the terminal set U is unambiguous from the context) if all the terminals are connected in H .*

Definition 4.3 (c.f. Definition 3.2. in [AKL⁺22]). *A U -Steiner-subgraph packing \mathcal{P} is a collection of U -Steiner subgraphs H_1, \dots, H_k , where each subgraph H_i is assigned a value $\text{val}(H_i) > 0$. If all $\text{val}(H_i)$ are integral, we say that \mathcal{P} is an integral packing. Throughout, a packing is assumed to be fractional (which means that it does not have to be integral), unless specified otherwise. The value of the packing \mathcal{P} is the total value of all its Steiner subgraphs, denoted $\text{val}(\mathcal{P}) = \sum_{H \in \mathcal{P}} \text{val}(H)$. We say that \mathcal{P} is feasible if $\forall e \in E, \sum_{H \in \mathcal{P}: e \in H} \text{val}(H) \leq w(e)$.*

We next state a natural LP to pack U -Steiner-subgraphs and its dual program. Here \mathcal{H} denotes the set of all U -Steiner subgraphs of G . For the dual program, we define $\ell(H) = \sum_{e \in H} \ell(e)$.

$\begin{aligned} (\mathbf{P}_{LP}) \\ \max \quad & \sum_{H \in \mathcal{H}} \text{val}(H) \\ \text{s.t.} \quad & \sum_{H \in \mathcal{H}: e \in H} \text{val}(H) \leq w(e) \quad \forall e \in E \\ & \text{val}(H) \geq 0 \quad \forall H \in \mathcal{H} \end{aligned}$	$\begin{aligned} (\mathbf{D}_{LP}) \\ \min \quad & \sum_{e \in E} w(e) \ell(e) \\ \text{s.t.} \quad & \ell(H) \geq 1 \quad \forall H \in \mathcal{H} \\ & \ell(e) \geq 0 \quad \forall e \in E \end{aligned}$
---	---

The dual LP can be interpreted as minimizing over length functions ℓ to find a non-negative function that assigns each U -Steiner subgraph of G total lengths at least 1 while minimizing the weighted length over all edges.

Interpreting w as a capacity and val as a flow, the primal LP formulations is strongly reminiscent of the st -max-flow formulation as being a set of st -paths with val units of flow sent along each path such that the capacity on each edge is respected. For this formulation, at the cost of an $(1 + \epsilon)$ -approximation, the well-known Garg-K  nnemann MWU framework reduces the st -max flow problem to a sequence of relaxed dual programs that amount to dynamically computing st -shortest paths under an increasing length function. This sequence of problems has recently been shown to be solvable via modern shortest paths data structures near-linear time. Thus, yielding approximate st -max flow (see [BGS22]).

Algorithm 1: A $(\gamma + O(\epsilon))$ -approximation algorithm for Steiner subgraph packing

Input : Undirected edge-weighted graph $G = (V, E, w)$, terminal set $U \subseteq V$, and accuracy parameter $0 < \epsilon < 1$, termination threshold $\tau \in [(1 - \epsilon), 1]$.

Output: Feasible U -Steiner subgraph packing \mathcal{P}

```

1  $\mathcal{P} \leftarrow \emptyset$  and  $\delta \leftarrow (2m)^{-1/\epsilon}$ 
2 foreach  $e \in E$  do  $\ell(e) \leftarrow \delta/w(e)$ ;
3 while  $\sum_{e \in E} w(e)\ell(e) < \tau$  do
4    $H \leftarrow$  a  $\gamma$ -approximate minimum  $\ell$ -length  $U$ -Steiner subgraph
5    $v \leftarrow \min\{w(e) : e \in E(H)\}/\alpha$ 
6   add  $H$  with  $\text{val}(H) \leftarrow v$  into the packing  $\mathcal{P}$ 
7   foreach  $e \in E(H)$  do  $\ell(e) \leftarrow \ell(e) \cdot \text{EXP}(\frac{\epsilon v}{w(e)})$ ;
8 return a scaled-down packing  $\mathcal{P}'$ , where  $\text{val}(H) \leftarrow \text{val}(H)/\log_{1+\epsilon}(\frac{1+\epsilon}{\delta})$  for all  $H \in \mathcal{P}$ 

```

[AKL⁺22] exploits this similarity and shows a similar reduction to a sequence of relaxed dual solves. In particular, they suggest the following algorithm.

Here, we use the notion of γ -approximate minimum ℓ -length U -Steiner subgraph H which is defined to be a graph $H \subseteq G$ with length $\ell(H) \leq \gamma \cdot \min_{H' \in \mathcal{H}} \ell(H')$. We use the following result from [AKL⁺22].

Lemma 4.4 (c.f. Lemma 3.3 and 3.10 in [AKL⁺22]). *The scaled-down packing \mathcal{P}' that is returned by Algorithm 1 for any $\alpha \geq 1$ is feasible and satisfies $\text{val}(\mathcal{P}') \geq \lambda(U)/2(\gamma + \Theta(\epsilon))$ (for any input $\tau \in [1 - \epsilon, 1]$).*

Our statement of Algorithm 1 and Lemma 4.4 slightly differs from the statements in [AKL⁺22] in the following ways:

- In [AKL⁺22] the algorithm uses $\tau = 1$, however, it is straightforward to show that any value $\tau \in [(1 - \epsilon), 1]$ suffices to achieve the lower bound on the packing value of \mathcal{P}' and does not affect feasibility of the solution.
- We increase variables $\ell(e)$ by factor $\text{EXP}(\frac{\epsilon v}{w(e)})$ instead of $(1 + \frac{\epsilon v}{w(e)})$. This allows us to characterize the length function $\ell(e)$ after many increases v_1, v_2, \dots, v_k by $\ell(e) = \text{EXP}(\epsilon \cdot \sum_{i=1}^k v_i)$.

The change affects the analysis of the MWU only very slightly since $1 + x \approx e^x$ for x being small (in our case, we always have $x \leq \epsilon$). See [BGS22] for a similar MWU analysis with the adapted length function update rule.

- For technical reasons, we scaled down the amount of value v added in each iteration by a factor α . We require this change as we allow for the rest of the section for U -Steiner subgraphs H to be multi-graphs, i.e. to select edges of G multiple times, where edge $e' \in E(G)$ occurs in subgraph H up to α times.

This ensures that every update of an edge e' that occurs $k \leq \alpha$ times in H increases the length $\ell(e')$ by $\text{EXP}(\frac{\epsilon \cdot k \cdot v}{w(e')}) \leq \text{EXP}(\frac{\epsilon \cdot \alpha \cdot w(e')/\alpha}{w(e')}) = \text{EXP}(\epsilon)$ per iteration of the outer while-loop. Thus the exponent increases is bounded by at most ϵ as previously promised, which is important to ensure correctness of the MWU analysis which exploits that $1 + 2x \geq e^x$ for $x \leq 1$. Naturally,

decreasing v by α in every iteration results in slower convergence of the algorithm, however, as in our application α is of subpolynomial size, this slowdown is acceptable.

It remains to implement Algorithm 1 efficiently and for small γ and to extract a vertex sparsifier from a packing \mathcal{P} .

An Efficient Implementation of Algorithm 1. Algorithm 1 requires for efficient implementation a data structure that maintains a γ -approximate minimum ℓ -length U -Steiner subgraph where ℓ is point-wise monotonically increasing over time.

One of the key observations in [AKL⁺22] is that one can reduce the problem of maintaining a $(2 + O(\tau))$ -approximation of the minimum ℓ -length U -Steiner subgraph to the problem of maintaining $(1 + \tau)$ -approximate single-source shortest-paths (SSSP) in a graph undergoing edge deletions. They then use a data structure for this problem by Bernstein, Probst Gutenberg and Saranurak [BGS22] which maintains an approximate SSSP tree in a graph H that is not a subgraph of G but can be embedded into G (although with very high congestion). We use the following data structure by Kyng, Meierhans and Probst Gutenberg [KMG24] in-lieu which achieves almost the same guarantees but additionally, maintains the SSSP tree T in a forest graph F that embeds with low congestion α into the original graph G w.r.t. ℓ .⁵ This is crucial to obtain an efficient deterministic implementation of Algorithm 1 as the convergence of the MWU algorithm now scales linearly in α . We now give formal definitions of embeddings of low congestion and state the theorem from [KMG24] that we use in our algorithm.

Definition 4.5 (Graph Embedding). *Given two graphs H, G , and a vertex map $\Pi_{V(H) \rightarrow V(G)}$ that maps every vertex in H to a vertex in G , we say that a map $\Pi_{H \rightarrow G}$ is a graph embedding of H into G if it maps each $e = (u, v) \in H$ to a xy -path $\Pi_{H \rightarrow G}(e)$ in G for $x = \Pi_{V(H) \rightarrow V(G)}(u)$, and $y = \Pi_{V(H) \rightarrow V(G)}(v)$.*

Definition 4.6 (Edge Congestion of Paths and Embeddings). *Given a set of paths P_1, P_2, \dots, P_k in graph G , we define the edge congestion induced by a collection of paths for an edge $e \in E(G)$ by*

$$\text{econg}(\{P_i\}_{i \in [1, k]}, e) = \sum_{i \in [1, k]} \sum_{e' \in P_i} \mathbf{1}[e = e'].$$

We define the edge congestion by $\text{econg}(\{P_i\}_{i \in [1, k]}) = \max_{e \in E} \text{econg}(\{P_i\}_{i \in [1, k]}, e)$. We define the congestion of a graph embedding $\Pi_{H \rightarrow G}$ by $\text{econg}(\Pi_{H \rightarrow G}) = \text{econg}(\text{im}(\Pi_{H \rightarrow G}))$.

Theorem 4.7 (c.f. Theorem 5.1 in [KMG24]). *Given graph $G = (V, E, \ell)$ with $L \leq n^4$ that undergoes a sequence of Δ edge length increases, a dedicated source vertex $s \in V$, and an accuracy parameter $\epsilon = \Omega(1/\text{polylog}(m))$. Then, there is an algorithm that maintains a forest F along with injective vertex maps $\Pi_{V(G) \rightarrow V(F)}, \Pi_{V(F) \rightarrow V(G)}$ (such that $\Pi_{V(F) \rightarrow V(G)} \circ \Pi_{V(G) \rightarrow V(F)} = \text{id}_{V(G)}$) and an embedding $\Pi_{F \rightarrow G}$ that maps each edge in F to a single edge in G such that, for some $\gamma_{SSSP} = e^{O(\log^{83/84} m \log \log m)}$, at any time:*

1. *for every $v \in V$, if $\text{dist}_G(s, v) < \infty$ ⁶, then $\Pi_{F \rightarrow G}(\pi_F(\Pi_{V(G) \rightarrow V(F)}(s), \Pi_{V(G) \rightarrow V(F)}(v))) \leq (1 + \epsilon) \text{dist}_G(s, v)$, i.e. the unique path between the two nodes in F (denoted by $\pi_F(\cdot, \cdot)$) that vertices s and v are mapped to has length at most $(1 + \epsilon) \text{dist}_G(s, v)$, and*

⁵Another way of thinking about this is that F has up to $\alpha = m^{o(1)}$ copies of the vertices and edges in G that are patched together such that each path in F maps to a path in G .

⁶In [KMG24], this condition is stated as $\text{dist}_G(s, v) \leq n^5$ but for $L \leq n^4$, these conditions are equivalent.

2. $\text{econg}(\Pi_{F \rightarrow G}) \leq \gamma_{SSSP}$.

The algorithm maintains F and the associated maps $\Pi_{V(G) \rightarrow V(F)}$, $\Pi_{V(F) \rightarrow V(G)}$ and $\Pi_{F \rightarrow G}$ explicitly (in fact, $\Pi_{V(G) \rightarrow V(F)}$ is constant) and the total number of changes to F and these maps is at most $m \cdot \gamma_{SSSP}$. The algorithm runs in time $O(\Delta) + m \cdot \gamma_{SSSP}$.

Remark 4.8. The algorithm can, within the same asymptotic time, explicitly maintain for each vertex v with $\text{dist}_G(s, v) \leq \infty$, an approximate distance estimate $\hat{d}(v)$ such that at all times $\text{dist}_G(s, v) \leq \hat{d}(v) \leq (1 + \epsilon)^2 \text{dist}_G(s, v)$ and otherwise sets $\hat{d}(v) = \infty$.

Note that while F is a forest graph, the only relevant tree in F is the tree T_s that contains the source vertex s as all vertices $v \in V$ with $\text{dist}_G(s, v) < \infty$ are required to have their image in the same tree T_s by the guarantees stated in 1 of Theorem 4.7. Since the map $\Pi_{V(G) \rightarrow V(F)}$ is constant, we henceforth identify the vertices V with the vertices in $\Pi_{V(G) \rightarrow V(F)}(V)$, i.e. we often use v to represent both the vertex $v \in V(G)$ and the vertex $\Pi_{V(G) \rightarrow V(F)}(v)$. Analogously, we often write $w(e)$ for edge $e \in F$ in-lieu of $w(\Pi_{F \rightarrow G}(e))$.

We now give the algorithmic description of the MWU implementation using the data structure from Theorem 4.7. While Theorem 4.7 assumes all edge weights to be in $[1, n^4]$, our description applies the data structure to a graph with weights in $\{0\} \cup [1, m^{O(\log(n)/\epsilon)}]$ for some constant $\epsilon > 0$. Towards the end of the section, we show that this limitation of the data structure can be remedied without increasing the runtime asymptotically by a relatively straightforward reduction. We thus ignore this technicality until then.

Definition 4.9. For graph G and length function $\tilde{\ell}$, we denote by $G_{\tilde{\ell}} = (V \cup \{s^*\}, E \cup \{(s^*, u) \mid u \in U\}, \tilde{\ell})$ the graph G with an additional dummy source s^* and a zero length edge from s^* to each vertex $u \in U$.

Throughout our implementation, we will maintain a length function $\tilde{\ell}$ to approximate the 'true' length function ℓ , which is not maintained by our algorithm but used in our theoretical analysis. With the length function $\tilde{\ell}$, we also maintain:

- a data structure \mathcal{D} as in Theorem 4.7 (without the restriction on lengths, as we discuss above) on the graph $G_{\tilde{\ell}}$ undergoing edge length increases. We let henceforth F denote the forest (undergoing edge/vertex insertions and deletions) maintained by the data structure \mathcal{D} and $\hat{d}(v)$ the distance estimate for vertex $v \in V$.
- an auxiliary graph $A = (\hat{V}, \hat{E})$ and length function $\hat{\ell}$ defined as follows:
 - \hat{V} consists of the vertex set $V(F)$ except for the dummy vertex s^* and all of its copies, i.e. the set $V(F) \setminus \Pi_{V(F) \rightarrow V(G_{\tilde{\ell}})}^{-1}(s^*)$,
 - \hat{E} consists of two types of edges (associated with lengths $\hat{\ell}$):
 - * each edge e in the induced graph $F[\hat{V}]$ is added to \hat{E} and assigned length 0 under $\hat{\ell}$, and
 - * for each edge $e = (u, v) \in E$, we add an edge $\hat{e} = (\Pi_{V(G) \rightarrow V(F)}(u), \Pi_{V(G) \rightarrow V(F)}(v))$ with length $\hat{\ell}(\hat{e}) = \hat{d}(u) + \tilde{\ell}(e) + \hat{d}(v)$.

We let $\Pi_{A \rightarrow G}$ denote the graph embedding that maps each edge e in A that originated from the induced graph $F[\hat{V}]$ to $\Pi_{F \rightarrow G_{\tilde{\ell}}}(e)$ and every edge \hat{e} that originated from an edge $e \in E$ back to the edge e .

- a minimum spanning forest (MSF) M of A maintained by running the algorithm of Holm et al. [HDLT01] on the fully-dynamic graph A .⁷

The auxiliary graph plays a similar role to the helper graph defined in [AKL⁺22]. We show below that the minimum spanning forest of A approximates the minimum U -Steiner subgraph of G w.r.t. ℓ . Note that since F contains multiple copies of each edge in G , and since we maintain the MST on a graph containing F , we might have each edge in G occurring multiple times in M (however, at most γ_{SSSP} times). The claim below shows that a subgraph M^U of M exists that when mapped back to G , i.e. $\ell(\Pi_{A \rightarrow G}(M^U))$ forms a good approximation of the minimum ℓ -length U -Steiner subgraph. In fact, even if one maps M^U point-wise, which means that copies of edges e in G , are accounted for with cost $\ell(e)$ times the number of occurrences in M^U , the length remains competitive.

Claim 4.10. *For F, A, M defined above, let M^U be the graph obtained from M by deleting every edge that is not on a path between two vertices in $\Pi_{V(G_{\tilde{\ell}}) \rightarrow V(F)}(U)$. If for all edges $e \in E$, we have $\frac{1}{1+\epsilon}\ell(e) \leq \tilde{\ell}(e) \leq (1+\epsilon)\ell(e)$ for some $\epsilon \in (0, 1)$, then $\Pi_{A \rightarrow G}(M^U)$ is a $(2 + \Theta(\epsilon))$ -approximate minimum ℓ -length U -Steiner subgraph. Moreover, $\sum_{e \in M^U} \ell(\Pi_{A \rightarrow G}(e))$ is no more than $(2 + \Theta(\epsilon))$ times the cost of the minimum ℓ -length U -Steiner subgraph.*

Proof. We first note that it suffices to prove the results for $\tilde{\ell}$ as it is a $(1 + \epsilon)$ -approximation of ℓ . Throughout the proof, we denote by dist_G the distance function on G w.r.t. $\tilde{\ell}$ and denote by OPT the length of the minimum U -Steiner subgraph of G w.r.t. $\tilde{\ell}$.

We start by constructing a spanning forest of A with length no more than $(2 + \Theta(\epsilon))\text{OPT}$. Let V_F , U_F , and s_F be the images of V , U , and s^* under the vertex map $\Pi_{V(G_{\tilde{\ell}}) \rightarrow V(F)}$, respectively. Let T_s be the tree in F that contains s_F and we consider s_F as the root of T_s . By Theorem 4.7, all vertices in V_F are connected to s_F in F , so $V_F \subseteq T_s$.

We further claim that U_F is exactly the set of children of s_F . Indeed, we have

- (1) By construction of $G_{\tilde{\ell}}$, the edge connecting s^* and any vertex in U has length 0. Therefore, by Theorem 4.7, any u_F in U_F is connected to s_F by a path in F , whose image in G has zero length. As edges in E have positive lengths, the path $\pi_F(s_F, u_F)$ must be a single edge. So vertices in U_F are children of s_F in F .
- (2) By Theorem 4.7, there cannot be an edge in F connecting s_F and $\Pi_{V(G) \rightarrow V(F)}(x)$ for $x \in V \setminus U$, because $\Pi_{F \rightarrow G}$ maps each edge to an edge in G , by Theorem 4.7.

So the children set of s_F in T_s is U_F . By construction, A can thus be decomposed into (1) subtrees of T_s , each subtree rooted at a vertex $u_F \in U_F$ and (2) trees other than T_s in F . Since the lengths in A of edges in these trees are set to 0, any two vertices in the same tree are at distance 0 in A .

With such a partition, we can easily extend any U_F -Steiner tree in A to a spanning forest by adding zero-length edges originating from $F[\hat{V}]$. It then remains to construct a U_F -Steiner tree in A with length no more than $(2 + \Theta(\epsilon))\text{OPT}$. Let $G^*[U] = (U, E_{G^*[U]}, \ell_{G^*[U]})$ be the transitive closure of U in G , i.e., for any vertices $u, v \in U$, there is an edge $(u, v) \in E_{G^*[U]}$ with length $\text{dist}_G(u, v)$. By Lemma 4.2 in [AKL⁺22], the MST of $G^*[U]$ has length at most 2OPT (This lemma can be proven by simply replacing edges in the MST of $G^*[U]$ by their corresponding shortest paths in G). We will prove the following:

⁷Technically speaking, the algorithm in [HDLT01] is only described for maintaining a minimum spanning tree and for fully-dynamic graphs undergoing only edge updates, not vertex updates. Both of these limitations can however be lifted without an asymptotic increase in running time by standard techniques.

- For any bipartition (U', U'') of U and any $u \in U', v \in U''$, there exists a path P in A connecting $\Pi_{V(G) \rightarrow V(F)}(U')$ and $\Pi_{V(G) \rightarrow V(F)}(U'')$ with length $(1 + \Theta(\epsilon)) \text{dist}_G(u, v)$. Moreover, $P \cap U_F$ only contains the two endpoints of P .

To see why this statement leads to a U_F -Steiner tree in A with our desired length, we start with $T = \text{MST}_{G^*[U]}$ and let $e = (u, v)$ be an arbitrary edge in T . Removing e from T results in a bipartition of U . Let P be one such path that connects the two components with length $(1 + \Theta(\epsilon)) \text{dist}_G(u, v)$. Suppose the two endpoints are $\Pi_{V(G) \rightarrow V(F)}(u')$, $\Pi_{V(G) \rightarrow V(F)}(v')$. We then update the T by replacing the edge e with a new edge (u', v') . By construction, the new T is still a spanning tree of $G^*[U]$. We can repeat until we replace all original edges from MST_H with new edges. As T is always a spanning tree of $G^*[U]$ throughout the process, the collection of corresponding paths forms a U_F -Steiner tree in A with length no more than $(1 + \Theta(\epsilon))\ell_{G^*[U]}(\text{MST}_{G^*[U]}) \leq (2 + \Theta(\epsilon))\text{OPT}$. We then prove the bullet point. Consider $u \in U', v \in U''$, let $(u = t_1, t_2, \dots, t_r = v)$ be a shortest path connecting u and v in G . Let u_F, v_F and $t_{j,F}$ be the images of u, v and t_j under the vertex map $\Pi_{V(G) \rightarrow V(F)}$, respectively. We denote the root of the subtree (of T_s) containing $t_{j,F}$ by $u_{j,F}$. Let $u_j = \Pi_{V(F) \rightarrow V(G_\ell)}(u_{j,F}) \in U$, then there exists some $1 \leq j < r$ such that $u_j \in U', u_{j+1} \in U''$. Let $P_1 = \pi_F(u_{j,F}, t_{j,F})$ and $P_2 = \pi_F(t_{j+1,F}, u_{j+1,F})$ be paths in F ; let $e' = (t_j, t_{j+1})$ be the edge in G and \hat{e}' be its induced edge in A . Then $P_1 - \hat{e}' - P_2$ is a path in A connecting U' and U'' , and its length is

$$\begin{aligned}\hat{\ell}(e') &= \hat{d}(t_{j,F}) + \tilde{\ell}(e') + \hat{d}(t_{j+1,F}) \\ &\leq (1 + \epsilon)^2 (\text{dist}_{G_\ell}(s^*, t_j) + \tilde{\ell}(e') + \text{dist}_{G_\ell}(s^*, t_{j+1})) \\ &= (1 + \epsilon)^2 (\text{dist}_{G_\ell}(u, t_j) + \tilde{\ell}(e') + \text{dist}_{G_\ell}(v, t_{j+1})) \\ &= (1 + \epsilon)^2 \text{dist}_{G_\ell}(u, v)\end{aligned}$$

This concludes the construction of a spanning forest of A with length no more than $(2 + \Theta(\epsilon))\text{OPT}$.

However, M might not be the same as our construction. Note that M^U has the same length as M (otherwise we can extend M^U to a spanning forest without increasing its length, leading to a spanning forest with length smaller than M). So by minimality of M , we have

$$\hat{\ell}(M^U) = \hat{\ell}(M) \leq (2 + \Theta(\epsilon))\text{OPT}.$$

It remains to prove that mapping M^U to G does not increase the length too much, i.e.,

$$\sum_{e \in M^U} \tilde{\ell}(\Pi_{A \rightarrow G}(e)) \leq (1 + \Theta(\epsilon))\hat{\ell}(M^U).$$

We do this by decomposing $E(M^U)$ into edges originating from E and paths in $F[\hat{V}]$. Recall our partition of A . As M^U is obtained by deleting edges in M not on paths connecting vertices in U , it does not contain vertices in trees other than T_s . Let \hat{E}_M be the edges in M^U that connect distinct subtrees of T_s . Note that \hat{E}_M is exactly the set of edges with non-zero lengths (i.e., edges originating from edges in E), since vertices inside any subtree can be connected by zero-length paths in A . We then observe that the edge set \hat{E}_M forms a spanning tree of U_F if we contract each subtree of T_s to its root; otherwise, we can replace a redundant edge in \hat{E}_M with existing edges and zero-length paths in subtrees, hence decreasing the length of M^U , leading to a contradiction. M^U thus consists of

- (1) $|U| - 1$ edges in \hat{E}_M connecting different subtrees;
- (2) for each edge $\hat{e} = (x_F, y_F)$ connecting two subtrees rooted at u_F and v_F , two paths $\pi_A(x_F, u_F)$ and $\pi_A(y_F, v_F)$ in $F[\hat{V}]$.

Let E_M be the edge set in G inducing \hat{E}_M in A and \mathcal{Q} be the set of such paths. We then have

$$\sum_{\hat{e} \in M^U} \tilde{\ell}(\Pi_{A \rightarrow G}(\hat{e})) = \sum_{e \in E_M} \tilde{\ell}(e) + \sum_{P \in \mathcal{Q}} \tilde{\ell}(\Pi_{A \rightarrow G}(P)).$$

On the other hand, we have

$$\begin{aligned} \hat{\ell}(M^U) &= \sum_{\hat{e} \in \hat{E}_M} \hat{\ell}(\hat{e}) = \sum_{e \in E_M} \left(\tilde{\ell}(e) + \sum_{x \in e} \hat{d}(x) \right) \geq \sum_{e \in E_M} \left(\tilde{\ell}(e) + \sum_{x \in e} \text{dist}_{G_{\tilde{\ell}}}(s^*, x) \right) \\ &\geq \sum_{e \in E_M} \tilde{\ell}(e) + \frac{1}{1+\epsilon} \sum_{P \in \mathcal{Q}} \tilde{\ell}(\Pi_{A \rightarrow G}(P)) \geq \frac{1}{1+\epsilon} \left(\sum_{e \in E_M} \tilde{\ell}(e) + \sum_{P \in \mathcal{Q}} \tilde{\ell}(\Pi_{A \rightarrow G}(P)) \right), \end{aligned}$$

where the two inequalities follow from Theorem 4.7. Combining the above lines gives us the desired inequality

$$\sum_{e \in M^U} \tilde{\ell}(\Pi_{A \rightarrow G}(e)) \leq (1 + \Theta(\epsilon)) \hat{\ell}(M^U).$$

□

Unfortunately, we cannot output M^U as defined in Claim 4.10 explicitly in each query as the total support of all such graphs might be too big. Instead, we use an implicit representation. In particular, we store M in a dynamic forest data structure with augmented functionality that we henceforth denote by \mathcal{M} .

Theorem 4.11. *Given graph $A = (\hat{V}, \hat{E})$ endowed with weight function $w_A : \hat{E} \mapsto \mathbb{R}^+$, length function $\ell_A : \hat{E} \mapsto \mathbb{R}^+$, congestion threshold function $\tau_A : \hat{E} \mapsto \mathbb{R}^+$, and subgraph $M \subseteq A$, both undergoing batches of edge insertions/deletions constrained such that the updates ensure that $M \subseteq A$ and remains a forest at all times. Let terminal set $U \subseteq \hat{V}$ satisfying U is connected at all times and denote by M^U again the subgraph of M with edge $e \in M$ being in M^U iff it is on a path between two terminals in U . There is a data structure \mathcal{M} that processes any edge update to M or A , maintains a field $\Delta \text{val}(\hat{e})$ for each edge \hat{e} in graph A , and can additionally implement the following operations:*

- RETURNSTEINERTREELLENGTH(): Returns $\ell_A(M^U)$.
- RETURNSTEINERTREEMINWEIGHT(): Returns $\min_{\hat{e} \in M^U} w(\hat{e})$.
- ADDVALUEONSTEINERTREE(Δ): Sets $\Delta \text{val}(\hat{e}) \leftarrow \Delta \text{val}(\hat{e}) + \Delta$ for each $\hat{e} \in M^U$.
- GETCONGESTEDEDGE(): Returns an edge \hat{e} in A with $\Delta \text{val}(\hat{e}) \geq \tau(\hat{e})$ if there is any.
- GETVALUE(\hat{e}): Returns $\Delta \text{val}(\hat{e})$.
- RESETVALUE(\hat{e}): Sets $\Delta \text{val}(\hat{e}) \leftarrow 0$.

For \hat{V}, \hat{E} and the update sequence being of size $\text{poly}(m)$, the data structure requires $O(|\hat{V}| + |\hat{E}|)$ initialization time and any processing of an update to A or M or any other data structure operation takes time $O(\log^2 m)$.

Remark 4.12. In fact, we are proving that we can explicitly output a graph R undergoing batches of edge insertions/deletions such that at all times $R \subseteq M$ and M^U is contained in R as a connected component. Further, if M undergoes k edge updates, then the total number of edge updates to R across all batches is only $O(k \log m)$.

We show how to implement \mathcal{M} by augmenting the classic dynamic forest data structures of Sleator and Tarjan [ST81] in Section 4.2 and henceforth assume the result where we give each edge $\hat{e} \in A$ weight $w_A(\hat{e}) = w(\Pi_{A \rightarrow G}(\hat{e}))$, length $\ell_A(\hat{e}) = \tilde{\ell}(\Pi_{A \rightarrow G}(\hat{e}))$ and set the threshold $\tau(\hat{e}) = w_A(\hat{e})/(4 \cdot \gamma_{SSSP})$. The terminal set is $\Pi_{V(G_{\tilde{\ell}}) \rightarrow V(F)}(U)$.

Given all the different data structures, we are now ready to give an efficient implementation of Algorithm 1. The pseudocode is given in Algorithm 2.

Algorithm 2: A $(\gamma + O(\epsilon))$ -approximation algorithm for Steiner subgraph packing

Input : Undirected edge-weighted graph $G = (V, E, w)$, terminal set $U \subseteq V$, and accuracy parameter $0 < \epsilon < 1$

Output: Implicitly-Represented feasible U -Steiner subgraph packing \mathcal{P}

```

1  $\mathcal{P} \leftarrow \emptyset$  and  $\delta \leftarrow (2m)^{-1/\epsilon}$ 
2  $\alpha \leftarrow 4\gamma_{SSSP}$ 
3 foreach  $e \in E$  do  $\tilde{\ell}(e) \leftarrow \delta/w(e)$ ;
4 Initialize graph  $G_{\tilde{\ell}}$ , data structure  $\mathcal{D}$  on  $G_{\tilde{\ell}}$  yielding forest  $R$ , auxiliary graph  $A$  from  $R$ , and a dynamic forest data structure  $\mathcal{M}$  yielding MSF  $M$ .
5 while  $\sum_{e \in E} w(e)\tilde{\ell}(e) < 1$  do
6    $v \leftarrow \mathcal{M}.\text{RETURNSTEINERTREEMINWEIGHT}() / \alpha$ .
7   Let  $t$  be the number of updates to the graph  $A$  since its initialization
    /* Implicitly add  $P = \Pi_{A \rightarrow G}(M^U)$  for  $M$  after  $t$  updates to the underlying graph  $A$  where issued, with  $\text{val}(P) = v$ . */
8    $\mathcal{P} \leftarrow \mathcal{P} \cup \{(t, v)\}$ .
9    $\mathcal{M}.\text{ADDVALUEONSTEINERTREE}(v)$ .
10  while  $\mathcal{M}.\text{GETCONGESTEDEDGE}()$  returns an edge  $e$  do
11     $e' \leftarrow \Pi_{A \rightarrow G}(e)$ .
12     $x \leftarrow 0$ .
13    foreach  $e'' \in \Pi_{A \rightarrow G}^{-1}(e')$  do
14       $| x \leftarrow x + \mathcal{M}.\text{GETVALUE}(e'')$ ;  $\mathcal{M}.\text{RESETVALUE}(e'')$ .
15       $\tilde{\ell}(e') \leftarrow \tilde{\ell}(e') \cdot \text{EXP}(\frac{\epsilon \cdot x}{w(e')})$ .
16    Update  $G_{\tilde{\ell}}$ , data structure  $\mathcal{D}$ , forest  $R$ , auxiliary graph  $\hat{G}$ , the MSF  $M$ , and a dynamic forest data structure  $\mathcal{M}$  on  $M$ .
17 return  $M$  and a scaled-down (implicitly represented) packing

$$\mathcal{P}' = \left\{ \left( t, v / \left( \log \left( \frac{(1+\epsilon)^2}{\delta} \right) / \epsilon \right) \right) \mid (t, v) \in \mathcal{P} \right\}$$


```

Analyzing the Implementation of Algorithm 1. We finish this section by proving that our implementation yields a $(4 + \Theta(\epsilon))$ -approximation of $\lambda(U)$, is efficient and that the implicit representation is useful to extract a vertex sparsifier.

Claim 4.13. *The implicit packing \mathcal{P}' produced by the implementation above of Algorithm 1 is feasible and its value attains at least a $1/(4 + \Theta(\epsilon))$ -fraction of the value $\lambda(U)$ for $\epsilon \in (0, 1)$.*

Proof. Let $\{M_j^U\}_{j=1}^k$ be the sequence of all M^U produced by Algorithm 2. For each edge e in E , we define $\ell(e, 0) = \delta/w(e)$ and for $j > 0$,

$$\ell(e, j) = \ell(e, j - 1) \cdot \exp\left(\frac{k_j v_j \epsilon}{\alpha w(e)}\right),$$

where $k_j = |\Pi_{A \rightarrow G}^{-1}(e) \cap M_j^U|$ denotes the number of copies of e in M_j^U , and $v_j = \min_{e' \in \Pi_{A \rightarrow G}(M_j^U)} w(e')$. We denote by $\tilde{\ell}(\cdot, j)$ the current length function $\tilde{\ell}$ in Algorithm 2 after (implicitly) producing M_j^U and executing Lines 9-16.

Observe that the only difference between $\tilde{\ell}$ and ℓ is that we store the values on the edges of the graph A and “flush” into $\tilde{\ell}(\Pi_{A \rightarrow G}(\hat{e}))$ when we detect that the accumulated value on \hat{e} is too large (larger than $w(\Pi_{A \rightarrow G}(\hat{e}))/(4\gamma_{SSSP})$), while we update all $\ell(e)$ each time after we produce a new M_j^U . Therefore, $\tilde{\ell}(e, j)$ is equal to $\ell(e, j)$ if e is updated in Line 15 of Algorithm 2. Moreover, we have $\tilde{\ell}(e, j) \leq \ell(e, j)$ for any e, j . We then prove that $\ell(e, j)/\tilde{\ell}(e, j) \leq 1 + \epsilon$. In fact, we do not update $\tilde{\ell}(e)$ after the algorithm produces M_j^U if and only if $\Pi_{A \rightarrow G}^{-1}(e)$ does not return any edge that is returned by any call of $\mathcal{M}.\text{GETCONGESTEDEDGE}()$ in the same outer while-loop iteration.

Thus, after any outer while-loop iteration, the amount of total value of the edges in $\Pi_{A \rightarrow G}^{-1}(e)$ is strictly smaller than

$$|\Pi_{A \rightarrow G}^{-1}(e)| \cdot \frac{w(e)}{4\gamma_{SSSP}} \leq \frac{w(e)}{4},$$

where we use the low congestion property in Theorem 4.7. Since this is also true at the beginning of every outer while-loop iteration, and since the maximum increase in value throughout an iteration by adding the current M^U as a packing is increased by at most $v \leq w(e)/\alpha$ and again e ’s image is at most γ_{SSSP} times in M^U , the total amount of value x across all edges in $\Pi_{A \rightarrow G}^{-1}(e)$ is at most $\frac{w(e)}{2}$ at any point in time.

Thus, the difference between $\tilde{\ell}$ and ℓ , is at most

$$\frac{\ell(e, j)}{\tilde{\ell}(e, j)} = \exp\left(\frac{\epsilon x}{w(e)}\right) < \exp\left(\frac{\epsilon}{2}\right) < 1 + \epsilon.$$

Therefore, $\tilde{\ell}$ is a $(1 + \epsilon)$ -approximation of ℓ . By Claim 4.10, $\sum_{e \in M_j^U} \ell(\Pi_{A \rightarrow G}(e), j - 1)$ is upper bounded by $(2 + \Theta(\epsilon))$ times the cost of the minimum $\ell(\cdot, j - 1)$ -length U -Steiner subgraph.

The reasoning above also tells us that the technical condition introduced by the MWU analysis underlying Lemma 4.4 is satisfied which requires that no increase in length on a coordinate e should exceed e^ϵ . We can therefore conclude by Lemma 4.4 that the final packing outputted by the algorithm is feasible and attains $1/(4 + \Theta(\epsilon)) \cdot \lambda(U)$. \square

Claim 4.14. *The above implementation of Algorithm 1 takes for any constant $\epsilon > 0$, deterministic time $m^{1+o(1)} \log W$.*

Proof. We first show that in Algorithm 2:

1. The while-loop in Line 5–Line 16 terminates in $m^{1+o(1)} \log W$ steps.
 2. Throughout the while-loop in Line 5–Line 16, the total number of times that Line 14 is executed is $m^{1+o(1)} \log W$.
1. Recall our definition of ℓ in Claim 4.13. For each iteration step, ℓ is updated for edges in $\Pi_{A \rightarrow G}(M_U)$. Moreover, for the minimum-weight edge $e \in E$ with some image $\Pi_{A \rightarrow G}^{-1}(e)$ in M^U , the length $\ell(e)$ is increased by factor $\exp(\epsilon/\alpha)$. For any edge $e \in E$, its initial length (w.r.t. ℓ) is $\delta/w(e)$ and once its length exceeds $1 + \epsilon$, we have that $\tilde{\ell}(e) > 1$ from the arguments in Claim 4.13 which implies that the outer while-loop condition is no longer met, and thus the outer while-loop terminates. Thus, we have for each edge $e \in E$, at most $\alpha \cdot \log \left(\frac{w(e)(1+\epsilon)}{\delta} \right) / \epsilon + 1 = m^{o(1)} \log W$ iterations where e is the minimum-weight edge in M^U . Since there are m edges, the claim follows.
 2. By Theorem 4.7, the size of $\Pi_{A \rightarrow G}^{-1}(e)$ is at most $\gamma_{SSSP} = m^{o(1)}$ for any $e \in E$. It then suffices to show the total number of edges returned by $\mathcal{M}.\text{GETCONGESTEDEDGE}()$ is $m^{1+o(1)}$. Let e be an edge returned by $\mathcal{M}.\text{GETCONGESTEDEDGE}()$ in Line 10 and let $e' = \Pi_{A \rightarrow G}(e)$ be its image in G . Then $\tilde{\ell}(e')$ is increased by at least $\exp(\epsilon/4\gamma_{SSSP})$ in Line 15. For any edge $e' \in E$, its initial length (w.r.t. $\tilde{\ell}$) is $\delta/w(e)$ and it is no longer updated once its length exceeds 1 (as otherwise the outer while-loop terminates). Thus, it is updated for at most $O \left(\log \left(\frac{w(e)}{\delta} \right) \cdot 4\gamma_{SSSP}/\epsilon \right) = m^{o(1)}$ times throughout the algorithm. As each edge returned by $\mathcal{M}.\text{GETCONGESTEDEDGE}()$ leads to an increase of the length $\tilde{\ell}$ for the pre-image of the edge in G by factor $e^{\epsilon/\alpha}$, the total number of such edges is at most $m^{1+o(1)} \log W$.

Given these two properties, we can conclude from Theorem 4.7 that the total time required by the data structure is $m^{1+o(1)} \log W$. It is not hard to see that the remaining operations are subsumed by this time bound. \square

Let us also address the shortcoming of Theorem 4.7 that allows only for lengths in $[1, n^4]$. However, since we are looking for a minimum length U -Steiner Subgraph w.r.t. $\tilde{\ell}$, and $\tilde{\ell}$ is monotonically increasing over time, it is easy to see that the minimum U -Steiner Subgraph is of monotonically increasing total length. The way we use the SSSP data structure, if currently, the minimum U -Steiner Subgraph has total length X , then we can still find an approximate solution on the graph where all edge weights are rounded up to the nearest multiple of $\epsilon \cdot X/m$ and edges of weight more than X are omitted from the graph. By appropriately scaling the lengths, it is not hard to see that they can be mapped to range $[1, n^3]$. Finally, we use the operation $\text{RETURNSTEINERTREELLENGTH}()$ from the data structure \mathcal{M} (see Theorem 4.11) to get a $4 + \Theta(\epsilon)$ approximation of X in polylogarithmic time, in each iteration of the outer while-loop. Thus, whenever we detect that the value of X has at least doubled since initialization of the current data structure, we can rebuild the data structure with newly rounded edges/ omitted edges as described above. This causes an additional $\log_2 Wn$ number of rebuilds of the data structure. By carefully analyzing with the bounds above, the additional time required is however only $m^{1+o(1)} \log W$ and thus already subsumed in the runtime that we obtained in Claim 4.14.

Extracting a Vertex Sparsifier from the Packing. Using dynamic tree structures, we can further extract a vertex sparsifier of G . We will use a slightly modified version of the Euler tour

data structure introduced in [HK95]. An Euler tour of a tree T with n' vertices is a cycle of $2n' - 2$ vertex symbols, obtained from running a depth-first search (DFS) from an arbitrary vertex as the root, and record each vertex u visited (including repetitive visits, but except for the last visit of the root) as an occurrence o_u . Each edge is visited twice (once in each direction) and each vertex of degree d is recorded d times. Given a forest with the Euler tours of trees maintained in binary search trees, adding/removing an edge can be done in $O(\log n')$ time using standard binary search tree techniques.

We will extract a vertex sparsifier of G based on the sequence of $\mathcal{M}.\text{ADDVALUEONSTEINERTREE}(\Delta)$ operations and edge updates to R generated by Theorem 4.11 according to Remark 4.12 obtained from running Algorithm 2. We maintain the Euler tours E' of trees in R in a binary search tree data structure \mathcal{E} . Let C be cycles obtained from repeatedly contracting on E' every edge that is not incident to two occurrences of terminals in U . We also maintain C in a binary search tree data structure \mathcal{C} . For every edge $(o_u, o_v) \in C$, we maintain a value $\Delta \text{val}'(o_u, o_v)$ in \mathcal{C} . We add that each edge insertion/deletion to R incurs $O(1)$ edge insertions/deletions to E' and C . Let G' be an initially empty graph on U .

We give the algorithm in Algorithm 3.

Algorithm 3: Extraction of vertex sparsifier from the Steiner subgraph packing

```

1 Initialize  $\mathcal{E}$  and  $\mathcal{C}$  defined above, and  $G'$  as an empty graph on  $U$ .
2 Run Algorithm 2 to obtain a sequence of  $\mathcal{M}.\text{ADDVALUEONSTEINERTREE}(\Delta)$  operations
   and edge updates to graph  $R$  generated by Theorem 4.11 according to Remark 4.12.
3 foreach operation  $op$  received do
4   | if  $op$  is an operation to  $R$  then
5     |   Perform the operation  $op$ .
6     |   Whenever some edge  $(o_u, o_v)$  is removed from  $C$ , flush the edge  $(o_u, o_v)$  in  $C$  by
        |   adding an edge  $(\Pi_{A \rightarrow G}(u), \Pi_{A \rightarrow G}(v))$  with weight  $\Delta \text{val}'(o_u, o_v)$  to  $G'$  and setting
        |    $\Delta \text{val}'(o_u, o_v) \leftarrow 0$ .
7   | else if  $op$  is  $\mathcal{M}.\text{ADDVALUEONSTEINERTREE}(\Delta)$  then
8     |   Add  $\Delta/2$  to  $\Delta \text{val}'(o_u, o_v)$  for every edge  $(o_u, o_v)$  in  $C$ .
9   | Flush all edges in  $C$  to  $G'$ .
10  | Scale down the weights of all edges in  $G'$  by a factor of  $\log\left(\frac{(1+\epsilon)^2}{\delta}\right)/\epsilon$ .
11 return  $G'$ .

```

Claim 4.15. *The number of edges in G' is $m^{1+o(1)} \log W$ and Algorithm 3 takes deterministic time $m^{1+o(1)} \log W$.*

Proof. First, since Algorithm 2 takes time $m^{1+o(1)} \log W$ by Claim 4.14, there are $m^{1+o(1)} \log W$ operations to R .

Each edge update to R may cut the Euler tours at at most 2 places and hence add at most 2 edges to G' . Also, Algorithm 3 adds one edge to G' for each remaining edge in R after all operations are performed. So the number of edges in G' is at most 3 times the number of operations to R , which is $m^{1+o(1)} \log W$.

Flushing an edge to G' and each operation to R and the Euler tour data structures \mathcal{E} and \mathcal{C} takes $O(\log n)$ time. So the running time of the above operations takes $m^{1+o(1)} \log W$ time. Plus,

flushing all the edges in C to G' takes time at most $O(\log n)$ times the number of edges in G' , which is also $m^{1+o(1)} \log W$. Hence Algorithm 3 has running time $m^{1+o(1)} \log W$. \square

Proof of Theorem 4.1. We choose a sufficiently small $\epsilon > 0$ such that, in Claim 4.13, the value of the feasible implicit packing \mathcal{P}' is at least $\lambda(U)/4.1$.

We run Algorithm 3 to construct a vertex sparsifier G' on U as the algorithm for Theorem 4.1. The running time of the algorithm is $m^{1+o(1)} \log W$ by Claim 4.14 and Claim 4.15. The number of edges in G' is $m^{1+o(1)} \log W$ by Claim 4.15.

Then we prove the two cut properties of G' . It suffice to show that the properties (1) and (2) hold if Algorithm 3 flushes all the edges in the cycle C to G' after each $\mathcal{M}.\text{ADDVALUEONSTEINERTREE}$ operation.

To prove property (1), for each cut (A, B) of G' and each U -Steiner subgraph H_i in the packing \mathcal{P}' , since Line 9 of Algorithm 2 invokes $\mathcal{M}.\text{ADDVALUEONSTEINERTREE}(v_i)$ where $v_i = \text{val}(H_i) \log\left(\frac{(1+\epsilon)^2}{\delta}\right)/\epsilon$ and thus Algorithm 3 adds a cycle containing all terminals to G' , there exist 2 paths in the cycle between adjacent terminal occurrences such that each path crosses the cut (A, B) odd times that corresponds to an edge added to G' crossing the cut (A, B) and has weight $\text{val}(H_i)/2$ after scaling down. Hence $\lambda_{G'} \geq \sum_{H \in \mathcal{P}'} \text{val}(H) \geq \lambda_G(U)/4.1$.

For (2), consider any cut (A, B) of G . For each U -Steiner subgraph H_i in the packing \mathcal{P}' and the corresponding cycle C_i added to G' by Algorithm 3, for each edge in H_i crossing the cut (A, B) , C_i goes across the cut (A, B) 2 times and hence Algorithm 3 adds at most 2 edges to G' , each with weight $\text{val}(H_i)/2$ and crosses the cut $(A \cap U, B \cap U)$. By the definition of a feasible packing, $w_{G'}(A \cap U, B \cap U) \leq \sum_{H \in \mathcal{P}'} \sum_{e \in H, e \in E(A, B)} \text{val}(H) \leq \sum_{e \in E(A, B)} w(e) = w_G(A, B)$. \square

4.2 Implementation of the Augmented Dynamic Tree Data Structure

In this section, we prove that the augmented dynamic tree data structure can be implemented efficiently. We restate the theorem for convenience.

Theorem 4.11. *Given graph $A = (\hat{V}, \hat{E})$ endowed with weight function $w_A : \hat{E} \mapsto \mathbb{R}^+$, length function $\ell_A : \hat{E} \mapsto \mathbb{R}^+$, congestion threshold function $\tau_A : \hat{E} \mapsto \mathbb{R}^+$, and subgraph $M \subseteq A$, both undergoing batches of edge insertions/deletions constrained such that the updates ensure that $M \subseteq A$ and remains a forest at all times. Let terminal set $U \subseteq \hat{V}$ satisfying U is connected at all times and denote by M^U again the subgraph of M with edge $e \in M$ being in M^U iff it is on a path between two terminals in U . There is a data structure \mathcal{M} that processes any edge update to M or A , maintains a field $\Delta \text{val}(\hat{e})$ for each edge \hat{e} in graph A , and can additionally implement the following operations:*

- `RETURNSTEINERTREELLENGTH()`: Returns $\ell_A(M^U)$.
- `RETURNSTEINERTREEMINWEIGHT()`: Returns $\min_{\hat{e} \in M^U} w(\hat{e})$.
- `ADDVALUEONSTEINERTREE(Δ)`: Sets $\Delta \text{val}(\hat{e}) \leftarrow \Delta \text{val}(\hat{e}) + \Delta$ for each $\hat{e} \in M^U$.
- `GETCONGESTEDEDGE()`: Returns an edge \hat{e} in A with $\Delta \text{val}(\hat{e}) \geq \tau(\hat{e})$ if there is any.
- `GETVALUE(\hat{e})`: Returns $\Delta \text{val}(\hat{e})$.
- `RESETVALUE(\hat{e})`: Sets $\Delta \text{val}(\hat{e}) \leftarrow 0$.

For \hat{V}, \hat{E} and the update sequence being of size $\text{poly}(m)$, the data structure requires $O(|\hat{V}| + |\hat{E}|)$ initialization time and any processing of an update to A or M or any other data structure operation takes time $O(\log^2 m)$.

Dealing with $A \setminus M$. Since the edges in $A \setminus M$ are not affected by $\mathcal{M}.\text{RETURNSTEINERTREELLENGTH}$, $\mathcal{M}.\text{RETURNSTEINERTREEMINWEIGHT}$ and $\mathcal{M}.\text{ADDVALUEONSTEINERTREE}$, we can store them with their Δval in a set so that we can get an edge $e \in A \setminus M$ with $\Delta \text{val}(e) \geq \tau(e)$ as well as add/remove an edge in $O(\log n)$ time.

Hence, in the following, we will focus on storing the edges in M and maintaining their Δval and ignore the edges in $A \setminus M$.

Representing M and graph $M^U \subseteq R \subseteq M$. We first note that though U is guaranteed to be connected between the batches of edge updates, U can be temporarily disconnected within each batch of updates. To support operations on M^U , we maintain a subgraph $R \supseteq M^U$ of M on the same vertex set $V(R) = V(M)$ that satisfies the invariant Property 4.16 using a dynamic forest data structure \mathcal{R} . We also maintain M in a dynamic forest data structure \mathcal{S} . When we refer to an edge, we are by default referring to an edge in M .

Property 4.16. *Each connected component of R either is a connected component of M^U , or is a path that does not contain any terminal vertex.*

Every edge e in \mathcal{R} has a value $\Delta \text{val}_R(e)$ and every edge e in \mathcal{S} has a value $\Delta \text{val}_M(e)$. For each edge $e \in R$, we maintain $\Delta \text{val}_R(e) = \Delta \text{val}(e)$ and set $\Delta \text{val}_M(e) = 0$. For the rest of the edges $e \in M \setminus R$, we let $\Delta \text{val}_M(e) = \Delta \text{val}(e)$. When an edge e is added to or removed from R , we update $\Delta \text{val}_R(e)$ and $\Delta \text{val}_M(e)$ correspondingly.

We adapt the concept of solid and dashed edges from [ST81], and they are not related to the implementation of the underlying dynamic forest data structures in our context. We say an edge e is solid if $e \in R$ and dashed otherwise. We call each connected component in R a solid tree. For each edge, we mark whether it is solid in \mathcal{S} . When we say we convert an edge e to solid, we mean to mark e as solid in \mathcal{S} and add e to R . Similarly, converting an edge e to dashed means to mark e as dashed in \mathcal{S} and remove e from R . We will only convert one edge to solid or dashed at a time.

Each tree in M is viewed as a rooted tree at some vertex. We maintain the root of the solid tree containing u as $\mathcal{S}.\text{ROOT}(u)$. The root of a solid tree containing u can be changed to u by invoking the $\mathcal{S}.\text{EVERT}(u)$ operation without changing the properties of the edges (e.g. whether they are solid or dashed), unlike in [ST81] changing the root of a tree using EXPOSE may convert some solid edges to dashed and vice versa.

The operations on \mathcal{S} and \mathcal{R} below are standard and are straightforward to implement using the techniques from [ST81]. Each of the following operations can be implemented in $O(\log n)$ time.

\mathcal{S} supports LINK, CUT and the following operations:

- **GLOBALLCA(u):** Returns the lowest common ancestor of all terminals in the tree containing u with respect to the root $\text{ROOT}(u)$, or NULL if there is no terminal.
- **MARKSOLID(u, v):** Marks the edge (u, v) as solid. This operation does not add the edge to R .
- **MARKDASHED(u, v):** Marks the edge (u, v) as dashed. This operation does not remove the edge from R .

- $\text{EVERT}(u)$: Sets the root of the tree containing u to u .
- $\text{ROOT}(u)$: Returns the root of the tree containing u .
- $\text{PARENT}(u)$: Returns the parent of u with respect to the root $\mathcal{S}.\text{ROOT}(u)$ if it exists, or NULL otherwise.
- $\text{HIGHEST}(u)$: Returns the last vertex on the u -to- $\mathcal{S}.\text{ROOT}(u)$ path that is reachable from u by only using solid edges.
- $\text{GETVALUE}(e)$: Returns $\Delta \text{val}_M(e)$.
- $\text{RESETVALUE}(e)$: Sets $\Delta \text{val}_M(e) \leftarrow 0$.
- $\text{GETCONGESTEDEDGE}()$: Returns an edge e in M with $\Delta \text{val}_M(e) \geq \tau(e)$.

Besides, \mathcal{R} supports LINK , CUT and the following operations:

- $\text{GETSOMEINCIDENTEDGES}(u)$: Returns the set of edges incident to u in R . If there are more than 10 edges incident to u , returns arbitrary 10 edges of them.
- $\text{GETVALUE}(e)$: Returns $\Delta \text{val}_R(e)$.
- $\text{RESETVALUE}(e)$: Sets $\Delta \text{val}_R(e) \leftarrow 0$.
- $\text{GETCONGESTEDEDGE}()$: Returns an edge e in R with $\Delta \text{val}_R(e) \geq \tau(e)$.

\mathcal{R} also supports the following operations whenever M^U exactly characterizes one of the connected components in R :

- $\text{RETURNSTEINERTREELLENGTH}()$: Returns $\ell_A(M^U)$.
- $\text{RETURNSTEINERTREEMINWEIGHT}()$: Returns $\min_{e \in M^U} w(e)$.
- $\text{ADDVALUEONSTEINERTREE}(\Delta)$: Sets $\Delta \text{val}_R(e) \leftarrow \Delta \text{val}_R(e) + \Delta$ for every edge e in M^U .

Implementation of \mathcal{M} . We implement each operation of \mathcal{M} as follows:

- $\mathcal{M}.\text{RETURNSTEINERTREELLENGTH}()$: Returns $\mathcal{R}.\text{RETURNSTEINERTREELLENGTH}()$.
- $\mathcal{M}.\text{RETURNSTEINERTREEMINWEIGHT}()$: Returns $\mathcal{R}.\text{RETURNSTEINERTREEMINWEIGHT}()$.
- $\mathcal{M}.\text{ADDVALUEONSTEINERTREE}(\Delta)$: Invokes $\mathcal{R}.\text{ADDVALUEONSTEINERTREE}(\Delta)$.
- $\mathcal{M}.\text{GETCONGESTEDEDGE}()$: If either $\mathcal{S}.\text{GETCONGESTEDEDGE}()$ or $\mathcal{R}.\text{GETCONGESTEDEDGE}()$ returns an edge, return the edge.
- $\mathcal{M}.\text{GETVALUE}(e)$: Returns $\mathcal{R}.\text{GETVALUE}(e)$ if $e \in R$ or $\mathcal{S}.\text{GETVALUE}(e)$ otherwise.
- $\mathcal{M}.\text{RESETVALUE}(e)$: Invokes $\mathcal{R}.\text{RESETVALUE}(e)$ if $e \in R$ or $\mathcal{S}.\text{RESETVALUE}(e)$ otherwise.

We also adapt the operations EXPOSE and SPlice operation from [ST81]. For an edge $(u, \mathcal{S}.\text{PARENT}(u))$, we say that the edge leaves u and enters $\mathcal{S}.\text{PARENT}(u)$. We implement the operations $\mathcal{M}.\text{SPlice}$, $\mathcal{M}.\text{EXPOSE}$, $\mathcal{M}.\text{LINK}$ and $\mathcal{M}.\text{CUT}$ as Algorithm 4, Algorithm 5, Algorithm 6 and Algorithm 7, respectively.

Algorithm 4: $\mathcal{M}.\text{SPLICE}(u)$

```
1  $v \leftarrow \mathcal{S}.\text{PARENT}(u).$ 
2 if  $\mathcal{S}.\text{HIGHEST}(v) \neq \mathcal{S}.\text{ROOT}(v)$  then
3    $E_1 \leftarrow \mathcal{R}.\text{GETSOMEINCIDENTEDGES}(v).$ 
4   Convert the solid edges in  $E_1$  that enters  $v$  to dashed.
5    $E_2 \leftarrow \mathcal{R}.\text{GETSOMEINCIDENTEDGES}(\mathcal{S}.\text{HIGHEST}(v)).$ 
6   Convert the other solid edges in  $E_2$  that enters  $\mathcal{S}.\text{HIGHEST}(v)$  to dashed except for the
      edge on  $v$ -to- $\mathcal{S}.\text{HIGHEST}(v)$  tree path.
7 Convert the dashed edge  $(u, v)$  to solid.
```

Algorithm 5: $\mathcal{M}.\text{EXPOSE}(u)$

```
1 while  $\mathcal{S}.\text{HIGHEST}(u) \neq \mathcal{S}.\text{ROOT}(u)$  do
2   |  $\mathcal{M}.\text{SPLICE}(\mathcal{S}.\text{HIGHEST}(u)).$ 
```

Algorithm 6: $\mathcal{M}.\text{LINK}(u, v)$

```
1  $\mathcal{S}.\text{EVERT}(u).$ 
2  $u' \leftarrow \mathcal{S}.\text{GLOBALLCA}(u).$ 
3  $\mathcal{S}.\text{EVERT}(v).$ 
4  $v' \leftarrow \mathcal{S}.\text{GLOBALLCA}(v).$ 
5  $\mathcal{S}.\text{LINK}(u, v).$ 
6 if  $u' \neq \text{NULL}$  and  $v' \neq \text{NULL}$  then
7   |  $\mathcal{S}.\text{EVERT}(u').$ 
8   |  $\mathcal{S}.\text{EXPOSE}(v').$ 
```

Algorithm 7: $\mathcal{M}.\text{CUT}(u, v)$

```
1 if  $(u, v) \in R$  then
2   | Convert the edge  $(u, v)$  to dashed.
3    $\mathcal{S}.\text{CUT}(u, v).$ 
4    $\mathcal{S}.\text{EVERT}(u).$ 
5    $u' \leftarrow \mathcal{S}.\text{GLOBALLCA}(u).$ 
6    $\mathcal{S}.\text{EVERT}(v).$ 
7    $v' \leftarrow \mathcal{S}.\text{GLOBALLCA}(v).$ 
8 if  $u' \neq \text{NULL}$  and  $v' \neq \text{NULL}$  then
9   | if  $u' \neq u$  then
10    |   |  $\mathcal{S}.\text{EVERT}(u).$ 
11    |   | Convert the edge  $(u', \mathcal{S}.\text{PARENT}(u'))$  to dashed.
12   | if  $v' \neq v$  then
13    |   |  $\mathcal{S}.\text{EVERT}(v).$ 
14    |   | Convert the edge  $(v', \mathcal{S}.\text{PARENT}(v'))$  to dashed.
```

Analysis. We now prove the correctness of our implementation.

Claim 4.17. *The invariant Property 4.16 holds after $\mathcal{M}.\text{LINK}$ (Algorithm 6) is called.*

Proof. If either u' or v' is NULL, then not both connected components containing u and v contained terminals before adding (u, v) to M . In this case, both M^U and R remain unchanged.

Otherwise, only the u' -to- v' path is added to M^U , and EXPOSE adds this path to R . Then we show that every edge incident to this path are removed from R , except for the edges incident to u' or v' , because u' and v' were in M^U . By Property 4.16, every solid tree that did not contain any terminal must be a path. Let v'' be the vertex v in Line 1. If $\mathcal{S}.\text{HIGHEST}(v'') \neq \mathcal{S}.\text{ROOT}(v'')$, every such path P containing v'' must start at some vertex u_1 , went through v'' and $\mathcal{S}.\text{HIGHEST}(v'')$ and ended at some vertex u_2 where P intersects the u -to- $\mathcal{S}.\text{ROOT}(u)$ path is exactly the v'' -to- $\mathcal{S}.\text{HIGHEST}(v'')$ path. Hence we only need to remove at most one edge incident to v'' and at most one edge incident to $\mathcal{S}.\text{HIGHEST}(v'')$ from R . Also, there were at most 2 edges incident to each of v'' and $\mathcal{S}.\text{HIGHEST}(v'')$, so $\mathcal{R}.\text{GETSOMEINCIDENTEDGES}$ would return all of them. Finally, the solid tree containing u_1 is still a path after LINK if u_1 is not on the u -to- $\mathcal{S}.\text{ROOT}(u)$ path, and the same for u_2 . \square

Claim 4.18. *The invariant Property 4.16 holds after $\mathcal{M}.\text{CUT}$ (Algorithm 7) is called.*

Proof. If either u' or v' is NULL, then the edge (u, v) were not in M^U and we only need to remove (u, v) from R if it was in R . Then the solid tree containing u before CUT must be a path, and can only be split into paths after removing the edge (u, v) .

Otherwise, the edge (u, v) was in M^U . Then we only need to remove the u' -to- v' path from R . So CUT removes the first and the last edge on the u' -to- v' path and the edge (u, v) from R and splits the rest of the path into paths. \square

For the running time, we denote $\text{SIZE}(v)$ as the size of the subtree in M rooted at v with respect to the root $\mathcal{S}.\text{ROOT}(v)$. We say a edge $(v, \mathcal{S}.\text{PARENT}(v))$ is heavy if $2 \cdot \text{SIZE}(v) > \text{SIZE}(\mathcal{S}.\text{PARENT}(v))$, or light otherwise. Let k be the total number of operations of LINK and CUT. Observe that the number of light edges on the path from any vertex to its root is small.

Observation 4.19. *For every vertex v , the number of light edges on the v -to- $\mathcal{S}.\text{ROOT}(v)$ path is $O(\log n)$.*

Claim 4.20. *There are at most $O(k \log n)$ splices.*

Proof. We say a SPLICE is light if the edge $(u, \mathcal{S}.\text{PARENT}(u))$ is light, and heavy otherwise. In addition, we say a SPLICE is special if the edge on the u -to- $\mathcal{S}.\text{ROOT}(u)$ path entering $\mathcal{S}.\text{HIGHEST}(\mathcal{S}.\text{PARENT}(u))$ is light, and normal otherwise. Let $\#hs$ be the number of heavy solid edges in the tree and $\#\text{normal_heavy_splices}, \#\text{light_splices}, \#\text{links}, \#\text{cuts}, \#\text{eversions}, \#\text{exposes}$ be the numbers of normal heavy splices, light splices, links, cuts, eversions and exposes, respectively.

One EXPOSE operation causes at most $O(\log n)$ light splices and at most $O(\log n)$ special heavy splices. Each normal heavy splice increases $\#hs$ by 1, each special heavy splice does not decrease $\#hs$, and each light splice decreases $\#hs$ by at most $O(1)$.

One $\mathcal{M}.\text{LINK}$, apart from invoking other functions, only increases the subtree size of u , so at most one solid edge incident to u is converted from heavy to light. Hence $\#hs$ is decreased by at most one.

One \mathcal{M} .CUT, apart from invoking other functions, only decreases the subtree size of u , so no edge is converted from heavy to light. Since we remove the edge (u, v) and it may be heavy and solid, $\#hs$ is decreased by at most one.

For EVERT, changing the root from the original root r to the new root u may convert at most $O(\log n)$ heavy edges from solid to dashed. Since only the subtree sizes of the vertices on the tree path from r to u are changed, only the heavy edges along the path or incident to the path may be converted from solid to dashed. The path contains at most $O(\log n)$ light edges after the eversion, and there are at most $O(\log n)$ heavy edges incident to the path before the eversion since each of such heavy edge corresponds to a light edge on the path, so one EVERT may decrease $\#hs$ by at most $O(\log n)$.

Therefore, $\#\text{links} - \#\text{cuts} \geq \#hs \geq \#\text{normal_heavy_splices} - \#\text{light_splices} - O(\log n) \cdot (\#\text{links} + \#\text{cuts} + \#\text{eversions})$, so $\#\text{normal_heavy_splices} \leq \#\text{light_splices} + O(\log n) \cdot (\#\text{links} + \#\text{cuts} + \#\text{eversions}) = O(k \log n)$.

Hence, the total number of splices is $O(k \log n)$. \square

Claim 4.21. *Each operation of \mathcal{M} takes $O(\log^2 n)$ time.*

Proof. There are $O(k \log n)$ splices by Claim 4.20. Each SPLICE adds one edge to R , so the numbers of edges ever added to R and ever removed from R are both $O(k \log n)$. Each SPLICE takes $O(\log n)$ time. Hence SPLICE has a running time of $O(k \log^2 n)$ over all invocations.

For the rest of the operations, it is clear that each operation takes time $O(\log n)$. Therefore, the running time is $O(\log^2 n)$ per operation. \square

4.3 Guide trees

In this section, we construct for given graph G a 16-respecting set of guide trees of size $n^{o(1)}$. Let U be the set of terminals, and $s \in U$ a dedicated source. Given the Steiner vertex sparsifiers from last section, we closely follow the approach in [Li21] to extract the guide trees:

- we first compute vertex sparsifier G' of G ,
- then apply Li's algorithm [Li21] on G' to produce a skeleton graph H with $m_{G'}^{1+o(1)}$ edges that of global edge connectivity $\lambda_H = n^{o(1)}$, such that for any $t \in U \setminus \{s\}$ and any (s, t) -mincut S in G , $S \cap U$ is still an approximate (up to a constant factor) mincut in H , and
- finally apply Gabow's tree packing algorithm [Gab91] on H to obtain an approximate maximum tree packing on H with size $\lambda_H = n^{o(1)}$. A similar analysis to Lemma 2.3 in [Kar00] shows the tree packing is a desired set of guide trees.

We start by stating Li's result to produce a skeleton graph.

Theorem 4.22 (c.f. Theorem 1.5 in [Li21]). *For any weighted input graph $G = (V, E, w)$, and constants $c > 1$, $\epsilon \in (0, 1]$, we can compute in deterministic time $\epsilon^{-4}n^{o(1)}m$ an unweighted (multi)graph H with $m^{1+o(1)}$ edges and some weight $W = \epsilon^4\lambda_G/n^{o(1)}$ such that*

1. *For any cut $\emptyset \subsetneq S \subsetneq U$ with cut size no more than $c\lambda_G$ of G , we have $W \cdot |\partial_H S| \leq (1 + \epsilon) \cdot w(\partial_G S)$.*
2. *For any cut $\emptyset \subsetneq S \subsetneq U$ of G , we have $W \cdot |\partial_H S| \geq (1 - \epsilon)\lambda_G$,*

where $\lambda_G = \min_{s,t \in V} \lambda_G(s,t)$ is the edge connectivity of G .

We remark that Li's original statement only states the upper bound for global mincuts of G . However, Li has shown in the proof that for all “unbalanced cuts” S , including any cut S with a cut size $O(\lambda_G)$, it holds that $|w(\partial_G S) - W \cdot |\partial_H S|| \leq \epsilon w(\partial_G S)$. The modified version Theorem 4.22 then follows.

We denote by G' the vertex sparsifier obtained from applying the algorithm in Theorem 4.1 to G, U and s . Then, we apply Theorem 4.22 to graph G' with $c = 5$ and $\epsilon = 0.2$ and denote the resulting graph by H . Recall that a tree packing is a subgraph packing where all subgraphs are trees. We then refer to Section 4.1 in [Kar00] (a modified version of Gabow's tree packing algorithm for directed graphs [Gab91]) to construct a feasible tree packing of H with value $\lambda_H/2$. And we claim that the tree packing is our desired guide tree set in Theorem 2.4.

Theorem 4.23 (c.f. Section 4.1 in [Kar00]). *We can compute a feasible tree packing \mathcal{T} of H with value $\lambda_H/2$ and size $|\mathcal{T}| = \lambda_H$ in deterministic time $\tilde{O}(m_H \lambda_H)$.*

Proof of Theorem 2.4. By construction, H is a graph defined on U , so a tree packing on H consists of trees defined on U . By Theorem 4.23, the size of \mathcal{T} is λ_H , which is upper bounded by $6\lambda_{G'}/W = n^{o(1)}$, according to Theorem 4.22. Regarding the running time, it remains to bound the running time of the tree packing algorithm in Theorem 4.23. We have $m_H = m^{1+o(1)}$ by Theorem 4.1 and Theorem 4.22, so the running time in Theorem 4.23 is $\tilde{O}(m_H \lambda_H) = m^{1+o(1)}$.

It remains to prove that \mathcal{T} is a set of guide trees. Let $t \in U \setminus \{s\}$ satisfy $\lambda_G(s, t) \leq 1.1\lambda_G(U)$ and let $(S, V \setminus S)$ be a (s, t) -mincut in G . It follows from Theorem 4.1 that

$$w_{G'}(S \cap U, U \setminus S) \leq w_G(S, V \setminus S) = \lambda_G(s, t) \leq 1.1\lambda_G(U) \leq 1.1 \cdot 4.1\lambda_{G'} < 5\lambda_{G'}.$$

Let $S' = S \cap U$. As T is defined on U , it suffices to find some $T \in \mathcal{T}$ 16-respects S' . By Theorem 4.22 with $\epsilon = 0.2$, we have $|\partial_H S'| \leq 6\lambda_{G'}/W$ and $\lambda_H \geq 0.8\lambda_{G'}/W$, so $|\partial_H S'| \leq 8\lambda_H$. For $T \in \mathcal{T}$, we denote by val_T the value of T and denote by x_T the number of edges crossing $(S', U \setminus S')$. By feasibility, we have $\sum_{T \in \mathcal{T}} \text{val}_T x_T \leq |\partial_H S'| \leq 8\lambda_H$. Moreover, the value of the tree packing \mathcal{T} is $\sum_{T \in \mathcal{T}} \text{val}_T = \lambda_H/2$, so the weighted average of $\{x_T\}_{T \in \mathcal{T}}$ is

$$\frac{\sum_{T \in \mathcal{T}} \text{val}_T x_T}{\sum_{T \in \mathcal{T}} \text{val}_T} \leq 16.$$

Therefore, there exists at least one $T \in \mathcal{T}$ with $x_T \leq 16$, concluding our proof. \square

5 Single-Source Mincuts via Guide Trees

In this section, we give a deterministic almost-linear time algorithm for single-source mincuts, which proves Theorem 2.2. We propose an algorithm (Theorem 5.2) for computing mincuts given one guide tree that de-randomizes the algorithm in [AKL⁺22]. The general idea follows from [AKL⁺22] and [Zha21]. A combination of the guide trees algorithm (Theorem 2.4) and Theorem 5.2 then immediately leads to the following partial version of SSMC, by taking the minimum of all returned functions $\tilde{\lambda}$.

Theorem 5.1 (Partial Singe Source Mincuts). *Given an undirected weighted graph $G = (V, E, w)$, a set of terminals $U \subseteq V$, and a source terminal s , we can compute in deterministic $m^{1+o(1)}$ time a function $\tilde{\lambda} : U \setminus \{s\} \rightarrow \mathbb{N} \cup \{+\infty\}$ such that $\tilde{\lambda}(t) \geq \lambda(s, t)$ for any $t \in U \setminus \{s\}$ and that $\tilde{\lambda}(t) = \lambda(s, t)$ if $\lambda(s, t) \leq 1.1\lambda(U)$.*

In Section 5.2, we remove the restriction $\lambda(s, t) \leq 1.1\lambda(U)$ in our proposed algorithm, which proves Theorem 2.2.

5.1 Partial Single-Source Mincuts

In this section, we provide a deterministic algorithm for the partial SSMC problem, hence proving Theorem 5.1. The major step is to compute the mincuts for vertices whose mincuts are k -respected by a given tree, which we summarize in Theorem 5.2.

Theorem 5.2. *Let $k \geq 2$ be a fixed integer, $G = (V, E, w)$ be an undirected weighted graph, and $s \in V$ be a vertex. Given a tree T defined on some subset of V such that $s \in V(T)$, we can compute in deterministic $m^{1+o(1)}$ time a function $\tilde{\lambda} : V(T) \setminus \{s\} \rightarrow \mathbb{N} \cup \{+\infty\}$ such that $\tilde{\lambda}(t) \geq \lambda(s, t)$ for any $t \neq s$ in T and that $\tilde{\lambda}(t) = \lambda(s, t)$ if T k -respects some (s, t) -mincut.*

Our algorithm follows the ideas of the randomized algorithm in Section 2 of [AKL⁺22] and the randomized algorithm in Section 2 of [Zha21]. The main tools for de-randomization are the Isolating Cuts Lemma (Lemma 3.4) and the following corollary of the Hit-And-Miss Theorem (Theorem 3.5).

Lemma 5.3. *Let X be a set with N elements. We can construct in deterministic time $\tilde{O}(N)$ a family of sets \mathcal{F} such that for any two distinct elements $x, y \in X$, there exists some set $A \in \mathcal{F}$ satisfying $x \in A$ and $y \notin A$. Moreover, the size of the set family is $O(\log N)$.*

Proof. We may assume $X = [N]$. Let \mathcal{H} be the function class constructed by Theorem 3.5. Let $\mathcal{F} = \{h^{-1}(i) : h \in \mathcal{H}, i \in [2]\}$. For any $x, y \in X$, let $h \in \mathcal{H}$ be a function satisfying $h(x) \neq h(y)$, then we have either $x \in h^{-1}(1), y \notin h^{-1}(1)$ or $x \in h^{-1}(2), y \notin h^{-1}(2)$. \square

The Deterministic SSMC Algorithm We are now ready to present the deterministic SSMC algorithm given a guide tree. Recall that the input is a graph $G = (V, E, w)$, a positive integer k , a vertex $s \in V$, and a tree T containing s . The goal is to output a function $\tilde{\lambda} : V(T) \setminus \{s\} \rightarrow \mathbb{N}$ such that for any $t \in V(T) \setminus \{s\}$, $\tilde{\lambda}(t) \geq \lambda(s, t)$ and if T k -respects some (s, t) -mincut, then we have $\tilde{\lambda}(t) = \lambda(s, t)$.

For a vertex t and a number x , we formally write $\text{UPDATE}(t, x) : \tilde{\lambda}(t) \leftarrow \min\{\tilde{\lambda}(t), x\}$. We denote our main algorithm by (G, T, s, k) , which consists of the following Steps 1-5. We also need a sub-algorithm consisting of Steps 1, 2, 3*, 4, 5*, which we denote by $(G, T, s, k)^*$. The sub-algorithm is used to tackle the case that s is a leaf of T .

1. Set $\tilde{\lambda}(t) = \infty$ for $t \in V(T)$. Directly compute $\lambda(s, t)$ for $t \in V(T)$ when $|V(T)| < 100$.
2. Find the *centroid* of T (denoted by c) such that each subtree has at most $|V(T)|/2$ vertices. If $s \neq c$, $\text{UPDATE}(c, \lambda(s, c))$ by running the deterministic (s, c) -maxflow algorithm from [vdBCP⁺23]. Root T at c , let T_0 be the subtree containing s (if it exists), and let $\{T_i\}_{i=1}^l$ denote other subtrees. Apply the isolating cuts algorithm to $\{V(T_0) \setminus \{s\}, V(T_1), \dots, V(T_l), \{s\}, \{c\}\}$. Let W_i denote the resulting cut containing $V(T_i)$ (or $V(T_0) \setminus \{s\}$ when $i = 0$).
3. Contract $\cup_{i=1}^l W_i$ to one vertex (labeled as c') and denote G, T after contraction by $G_0, T_0^{(1)}$, then call $(G_0, T_0^{(1)}, s, k)$. For each $i > 0$, contract $V \setminus W_i$ to one vertex (labeled as s) and denote G, T after contraction by $G_i, T_i^{(1)}$, then call $(G_i, T_i^{(1)}, s, k)$.

- 3*. Contract $\cup_{i=1}^l W_i$ to one vertex (labeled as c') and denote G, T after contraction by $G_0, T_0^{(1)}$, then call $(G_0, T_0^{(1)}, s, k)^*$.
4. Apply Lemma 5.3 to $\{T_1, \dots, T_l\}$ and denote the resulting set family by \mathcal{F} . For each $A \in \mathcal{F}$, we delete subtrees in A from T and obtain a new tree. We denote the set of all such resulting trees by \mathcal{T} . Recursively call $(G, T^{(2)}, s, k - 1)$ for each $T^{(2)} \in \mathcal{T}$.
5. Execute this step only if $s \neq c$. Let $T^{(3)}$ be the tree obtained by removing $V(T_0) \setminus \{s\}$ from T and adding the edge (s, c) . Call $(G, T^{(3)}, s, k - 1)$ and $(G, T^{(3)}, s, k)^*$.
- 5*. Execute this step only if $s \neq c$. Let $T^{(3)}$ be the tree obtained by removing $V(T_0) \setminus \{s\}$ from T and adding the edge (s, c) . Let z be one maximizer of current $\tilde{\lambda}$ over $\cup_{i=1}^l V(T_i)$. Let T' be the subtree of T that contains z (for convenience we may assume $T' = T_1$). Contract $\cup_{i \geq 0, i \neq 1} W_i$ to the centroid c and denote G, T after contraction by $G', T^{(4)}$. Call $(G, T^{(3)}, s, k - 1)$ and $(G', T^{(4)}, s, k)^*$.

For vertices $s, t \in V(T)$, let $\lambda_{G,T,k}(s, t)$ denote the minimum cut value over all (s, t) -cuts that are k -respected by T . When s is a leaf of T (i.e., $\deg_T(s) = 1$), let $\eta_{G,T,k}(s, t)$ denote the minimum cut value over all (s, t) -cuts that are k -respected by T and crossed by the edge incident to s in T . We set $\lambda_{G,T,k}(s, t)$ and $\eta_{G,T,k}(s, t)$ to be ∞ if such cuts do not exist. The following claim shows the correctness of our algorithm.

Claim 5.4. *Let $k \geq 2$ be a fixed integer, $G = (V, E, w)$ be an undirected weighted graph, and $s \in V$ be a vertex. Let T be a tree defined on some subset of V such that $s \in V(T)$. We have*

1. *The output of (G, T, s, k) satisfies $\lambda(s, t) \leq \tilde{\lambda}(t) \leq \lambda_{G,T,k}(s, t)$ for $t \in V(T) \setminus \{s\}$.*
2. *When $\deg_T(s) = 1$, the output of $(G, T, s, k)^*$ satisfies $\lambda(s, t) \leq \tilde{\lambda}(t) \leq \eta_{G,T,k}(s, t)$ for $t \in V(T) \setminus \{s\}$.*

Proof. Let $r = |V(T)|$. We will prove by induction on r, k . The base case ($r < 100, k \geq 1$) is completed by Step 1. Note that each recursion always strictly decreases either r or k except from $(G, T^{(3)}, s, k)^*$ in Step 5, which moves from the main algorithm to the sub-algorithm. It then suffices to prove the following induction step:

- (a) If the main algorithm succeeds for inputs with $k' < k, r' \leq r$ or $k' \leq k, r' < r$, and the sub-algorithm succeeds for inputs with $k' \leq k, r' \leq r$, then (G, T, s, k) succeeds.
- (b) If both algorithms succeed for inputs with $k' < k, r' \leq r$ or $k' \leq k, r' < r$, then $(G, T, s, k)^*$ succeeds.

So for the induction step, we may assume both algorithms succeed for inputs with $k' < k, r' \leq r$ or $k' \leq k$.

Lower bound in (a) and (b): Throughout the algorithm, the only way we modify the graph is contraction, which can only increase $\lambda(s, t)$. So the lower bound $\tilde{\lambda}(t) \geq \lambda(s, t)$ trivially holds.

Upper bound in (a): Following our above analysis about the induction step, we further assume the sub-algorithm succeeds when $k' = k, r' = r$ in this upper bound proof.

Let $t \in V(T)$, we may assume $\lambda_{G,T,k}(s, t) < \infty$. Let $(A, V \setminus A)$ be a (s, t) -cut that is k -respected by T , such that $w(\partial A) = \lambda_{G,T,k}(s, t)$. When $t = c$, Step 2 correctly outputs $\lambda(s, t)$. We may therefore assume $t \neq c$. Suppose T_i contains t . Let C denote the set of edges in T crossing A , which we call cut edges. Since the path connecting s, t in T is cut by A , at least one edge in the path is in C ,

which is either incident to T_0 or T_i . We can then divide the proof into three cases according to the locations of edges in C :

Case 1: All cut edges are incident to one subtree (can only be T_0 or T_i).

Case 2: There exists at least one cut edge incident to a subtree that is neither T_0 nor T_i .

Case 3: $T_0 \neq T_i$. There exist and only exist cut edges incident to T_0 and cut edges incident to T_i .

For Case 1, if all cut edges are only incident to T_0 , then $\cup_{i=1}^l V(T_i)$ must be in the same side of the cut $(A, V \setminus A)$. We may assume $\cup_{i=1}^l V(T_i) \subseteq A$. Note that the modified cut $(\cup_{i=1}^l W_i \cup A, V \setminus (\cup_{i=1}^l W_i \cup A))$ is also k -respected by T since vertices in $V(T)$ remain intact compared with $(A, V \setminus A)$. By submodularity, $(\cup_{i=1}^l W_i \cup A, V \setminus (\cup_{i=1}^l W_i \cup A))$ is also minimal among all (s, t) -cuts k -respected by T . So the contraction does not affect the cut value and hence $\lambda_{G_0, T_0^{(1)}, k}(s, c') = \lambda_{G, T, k}(s, t)$. Similarly, if all cut edges are only incident to T_i and $T_i \neq T_0$, we have $\lambda_{G_i, T_i^{(1)}, k}(s, t) = \lambda_{G, T, k}(s, t)$. Therefore, Step 3 outputs correctly in this case.

For Case 2, suppose some cut edge is incident to a subtree T_j that is neither T_0 nor T_i . By Lemma 5.3, there exists a tree $T^{(2)}$ in the tree family computed in Step 4 such that $T^{(2)}$ (rooted at c) has T_i as its subtree but does not contain T_j . Therefore, $T^{(2)}$ $(k-1)$ -respects the cut A . Step 4 then succeeds for this case.

It remains to tackle Case 3. Since t is not in T_0 , the tree $T^{(3)}$ in Step 5 still contains t . When s, c are in the same side of $(A, V \setminus A)$, all original cut edges incident to T_0 are removed, and the newly added edge (s, c) is not a cut edge in $T^{(3)}$, so $T^{(3)}$ contains at most $k-1$ cut edges and $(G, T^{(3)}, s, k-1)$ works for t . When s, c are not in the same side, s has an edge in $T^{(3)}$ and the edge (s, c) is cut by A , so $\eta_{G, T^{(3)}, k}(s, t) = \lambda_{G, T^{(3)}, k}(s, t)$ and the sub-algorithm $(G, T^{(3)}, s, k)^*$ works.

Upper bound in (b): Let $t \in V(T)$. Again, we may assume $\eta_{G, T, k}(s, t) < \infty$ and $t \neq c$. Let $(A, V \setminus A)$ be a (s, t) -cut that attains $\eta_{G, T, k}(s, t)$ and satisfies the corresponding conditions. Let (s, p) be the only edge connecting s in T , then the edge is cut by A . We still consider the same three cases as in the proof for (a). For the first case, all edges must be incident to T_0 , so Step 3*, the partial version of Step 3, suffices. The argument for Case 2 remains exactly the same as the argument for the upper bound in (a), so we omit it here.

For case 3, if there exists more than one edge incident to T_0 , then $T^{(3)}$ contain at most $k-1$ cut edges and $(G, T^{(3)}, s, k-1)$ works for t . In the rest of the proof, we assume the only cut edge in T incident to T_0 is (s, p) . We further assume that $\tilde{\lambda}(t) > w(\partial A) = \eta_{G, T, k}(s, t)$ at the current state (otherwise we are done). We claim that $i = 1$ in Step 5*. As the cut edges in T are either the edge (s, p) or incident to T_i , we have $\{c\}, V(T_0) \setminus \{s\}, V(T_j) \subseteq V \setminus A$, for $j \neq i$. Assume $i \neq 1$, then $z \in V(T_1) \subseteq V \setminus A$. So A is a (s, z) -cut that is k -respected by T and cuts (s, p) , which implies $\eta_{G, T, k}(s, z) \leq w(\partial A) < \tilde{\lambda}(t)$. On the other hand, $T_1 \neq T_i$ implies that $\tilde{\lambda}(z) \leq \eta_{G, T, k}(s, z)$ has been achieved in Step 4, contradicting the fact that z is the maximizer of current $\tilde{\lambda}$ over $\cup_{j=1}^l V(T_j)$. Therefore, we have $i = 1$. Since $\{c\}, V(T_0) \setminus \{s\}, \cup_{j=2}^l V(T_j)$ are contained in $V \setminus A$, the cut $(A \setminus (\cup_{j \geq 0, j \neq i} W_j \cup \{c\}), (V \setminus A) \cup (\cup_{j \geq 0, j \neq i} W_j \cup \{c\}))$ is still minimal over all (s, t) cuts k -respected by T and crossed by (s, p) . Thus, the contraction in Step 5* does not affect the value $\eta_{G, T, k}(s, t)$ and $(G', T^{(4)}, s, k)^*$ succeeds. \square

For time analysis, we note that the de-randomization step (Step 4) recursively calls the main algorithm $O(\log n)$ times with parameter no larger than $(m, n, r, k-1)$, which is the same as its randomized counterpart in [Zha21]. The other modifications of the algorithm in [Zha21] do not

affect the time analysis. Therefore, our recursive formula and analysis are exactly the same as Section 2.4 of [Zha21], so we omit it here.

Combining the above algorithm with the guide trees algorithm in Section 4 immediately gives the PARTIALSSMC algorithm.

Proof of Theorem 5.1. Let \mathcal{T} be the tree set obtained by applying Theorem 2.4 to (G, U, s) . For each $T \in \mathcal{T}$, apply Theorem 5.2 to T and denote the resulting function by $\tilde{\lambda}_T$. Let

$$\tilde{\lambda}(t) = \min_{T \in \mathcal{T}} \tilde{\lambda}_T(t) \text{ for } t \in U \setminus \{s\}.$$

By Theorem 5.2, we have $\tilde{\lambda}(t) \geq \lambda(s, t)$. Moreover, suppose $t \in U \setminus \{s\}$ satisfies $\lambda(s, t) \leq 1.1\lambda(U)$, then there exists some tree $T \in \mathcal{T}$ 16-respecting some (s, t) -mincut in G by Theorem 2.4. Therefore, $\tilde{\lambda}(t) \leq \tilde{\lambda}_T(t) = \lambda(s, t)$ by Theorem 5.2. Since the size of \mathcal{T} is $n^{o(1)}$, the overall time is still $m^{1+o(1)}$. \square

5.2 Single-Source Mincuts

We conclude Section 5 by removing the restriction in our partial SSMC algorithm, which leads to an unconditional SSMC algorithm.

Algorithm 8: Single-source mincuts algorithm

```

Input : Undirected edge-weighted graph  $G = (V, E, w)$ , source terminal  $s \in V$ 
Output:  $\{\lambda(s, t) : t \in V \setminus \{s\}\}$ 

1  $U \leftarrow V$ .
2 for  $i = 1, \dots, \lceil \log_{1.1}(nW) \rceil$  do
3    $\tilde{\lambda} \leftarrow \text{PARTIALSSMC}(G, U, s)$ .
4    $\lambda_{\min} \leftarrow \min_{t \in U \setminus \{s\}} \tilde{\lambda}(t)$ .
5    $A \leftarrow \{t \in U \setminus \{s\} : \tilde{\lambda}(t) \leq 1.1\lambda_{\min}\}$ .
6   foreach  $t \in A$  do  $\hat{\lambda}(t) = \tilde{\lambda}(t)$ ;
7    $U \leftarrow U \setminus A$ .
8   if  $U = \{s\}$  then break;
9 return  $\{\hat{\lambda}(t) : t \in V \setminus \{s\}\}$ 

```

Proof of Theorem 2.2. We need to show that Line 7 in Algorithm 8 always assign the correct value to $\hat{\lambda}(t)$ (i.e., $\lambda(s, t)$), and that U contains only s after the for loop. Note that $\lambda(U) = \min_{t \in U \setminus \{s\}} \lambda(s, t)$, since the mincut that cuts U is also a (s, t) -mincut for any vertex $t \in U$ lying in the different cut side than s . In any of the iteration step, let $t^* \in U \setminus \{s\}$ be some vertex satisfying $\lambda(s, t^*) = \lambda(U)$. We then have $\lambda_{\min} \leq \tilde{\lambda}(t^*) = \lambda(s, t^*) = \lambda(U)$ by Theorem 5.1. On the other hand, we have $\lambda_{\min} \geq \lambda(U)$ since $\tilde{\lambda}(t) \geq \lambda(s, t)$ for any $t \in U \setminus \{s\}$. Therefore, $\lambda_{\min} = \lambda(U)$. For any $t \in A$, we have $\lambda(s, t) \leq \tilde{\lambda}(t) \leq 1.1\lambda_{\min} = 1.1\lambda(U)$, so $\tilde{\lambda}(t) = \lambda(s, t)$ by Theorem 5.1. Thus, all assignments in Line 7 are correct.

Due to the above argument, we have $A = \{t \in U \setminus \{s\} : \lambda(s, t) \leq 1.1\lambda(U)\}$, so any vertex t in $U \setminus A$ other than s satisfies $\lambda(s, t) > 1.1\lambda(U)$ as long as $\{s\} \subsetneq U \setminus A$. Therefore, $\lambda(U \setminus A) > 1.1\lambda(U)$,

i.e., $\lambda(U)$ increases by at least a 1.1 factor after each iteration step. As $\max_{t \in V \setminus \{s\}} \lambda(s, t) \leq nW$, we have $U = \{s\}$ after the for loop.

For time analysis, we mark that the input graph to PARTIALSSMC is always G and we only have $O(\log(nW))$ calls to PARTIALSSMC, so the overall running time is still $m^{1+o(1)}$. \square

6 Gomory-Hu Tree via Single-Source Mincuts

In this section, we propose our deterministic algorithm for Gomory-Hu tree, hence proving Lemma 6.1, which immediately implies our main theorem Theorem 1.1, for $U = V$.

Lemma 6.1. *There is an algorithm COMPUTEGHSTEINERTREE(G, U) taking as inputs graph $G = (V, E, w)$ and set $U \subseteq V$ that returns a Gomory-Hu U -Steiner tree (T, f) . The algorithm runs in time $m^{1+o(1)}$.*

The rest of this section is organized as follows. In Section 6.1, we introduce our main tool, expander decompositions, and prove several important insights. In Section 6.2, we introduce the concept of rooted minimal Gomory-Hu Steiner tree and prove some of its properties. We present and analyze our deterministic Gomory-Hu algorithm in Section 6.3–Section 6.6. In Section 6.3, we propose the algorithm for detecting the desired τ^* -connected component. We then propose the algorithms for the case where $|U \setminus C|$ is large and the case where $|U \setminus C|$ is small, respectively, in Section 6.4 and Section 6.5. Finally, we put the detection phase and decomposition phase together in Section 6.6 to get the final Gomory-Hu tree algorithm.

6.1 Expander Decompositions and τ -Connectivity

We first present a strengthening of classic expander decompositions that is alluded to in [LRW25] and that can be derived by applying techniques from [SW19] to the deterministic expander decomposition in [LS21].

Definition 6.2. *Given $G = (V, E, w)$, $U \subseteq V$ and parameter $\phi > 0$, we say for any cut $X \subseteq V$ that it is ϕ -expanding with respect to U if $w(\partial X) \geq \phi \cdot \min\{|X \cap U|, |U \setminus X|\}$. We say G is a ϕ -expander with respect to U if every cut $X \subseteq V$ is ϕ -expanding with respect to U .*

We say a partition \mathcal{X} of V is a ϕ -expander decomposition w.r.t. U if for some $\gamma_{\text{expDecomp}} = e^{O(\log^{4/5}(m) \log \log(Wm))}$ (where W is the largest weight in G):

- for every $X \in \mathcal{X}$, $G[X]$ is ϕ -expander w.r.t. $X \cap U$, and
- for the flow problem where each vertex $v \in V$ for $v \in X \in \mathcal{X}$ has $w(E(\{v\}, \cup_{Y \in \mathcal{X}, Y \neq X} Y))$ units of source mass and each vertex $u \in U$ has $\phi \cdot \gamma_{\text{expDecomp}}$ units of sink capacity and vertices $v \in V \setminus U$ have no sink capacity, there exists a feasible flow f on the network G where each edge e has capacity $w(e) \cdot \gamma_{\text{expDecomp}}$.

Theorem 6.3. *There is a deterministic algorithm that, given graph $G = (V, E, w)$, $U \subseteq V$ and parameter $\phi > 0$, computes a ϕ -expander decomposition w.r.t. U in time $m^{1+o(1)}$.*

We defer the proof to Appendix A.

Note the following facts about expander decompositions (along with their relatively straightforward proofs).

Fact 6.4. Given graph $G = (V, E, w)$, $U \subseteq V$, τ and $\psi > 0$. Let \mathcal{X} be any $\psi \cdot \tau$ -expander decomposition w.r.t. U . The following properties hold:

1. Let $E_{\text{crossing}} = \cup_{X \in \mathcal{X}} \partial_G X$ denote the edges crossing clusters, then $|E_{\text{crossing}}| \leq \psi \cdot \tau \cdot \gamma_{\text{expDecomp}} \cdot |U|/2$.
2. $|\{X \in \mathcal{X} | w(\partial X) \geq \tau\}| \leq \gamma_{\text{expDecomp}} \cdot \psi \cdot |U|$, i.e. the number of clusters in \mathcal{X} that do not form a cut of value less than τ is bounded by $\gamma_{\text{expDecomp}} \cdot \psi \cdot |U|$.
3. For any $X, Y \subseteq V$, we define the crossing w.r.t. U as $\text{CROSSING}_U(X, Y) = \min\{|(X \cap Y) \cap U|, |(X \cap U) \setminus Y|\}$. For \mathcal{X} , we then have

$$\text{CROSSING}_U(Y, \mathcal{X}) = \sum_{X \in \mathcal{X}} \text{CROSSING}_U(Y, X) \leq w(\partial_G Y) / (\tau \cdot \psi). \quad (1)$$

4. For any $Y \subseteq V$ where $Y \cap U$ is τ -connected, the number of clusters in \mathcal{X} that intersect Y , i.e. $|\{X \in \mathcal{X} | X \cap Y \cap U \neq \emptyset\}|$, is at most $w(\partial_G Y) / (\tau \cdot \psi) + \max\{1, \gamma_{\text{expDecomp}} \cdot \psi \cdot |Y \cap U| + \gamma_{\text{expDecomp}} \cdot w(\partial_G Y) / \tau\}$.

Proof. We prove the facts one by one:

1. Each edge $e \in E_{\text{crossing}}$ contributes $w(e)$ source mass to each of its endpoints and thus the total source mass in the flow problem described in Definition 6.2 is $2 \cdot w(E_{\text{crossing}})$. Since the total amount of sink capacity is at most $\psi \cdot \tau \cdot \gamma_{\text{expDecomp}} \cdot |U|$, we have by the existence of a feasible flow that $2 \cdot w(E_{\text{crossing}}) \leq \psi \cdot \tau \cdot \gamma_{\text{expDecomp}} \cdot |U|$.
2. Let $E_{\text{crossing}} = \cup_{X \in \mathcal{X}} \partial_G X$. Let ℓ be the number of sets $X \in \mathcal{X}$ with $w(\partial X) \geq \tau$. Since \mathcal{X} is a partition of V and thus at most two clusters $X, Y \in \mathcal{X}$ are incident to the same edge, we have that $w(E_{\text{crossing}}) \geq \ell \cdot \tau/2$.

Thus, we have that

$$\ell \leq \frac{\gamma_{\text{expDecomp}} \cdot \psi \cdot \tau \cdot |U|/2}{\tau/2} = \gamma_{\text{expDecomp}} \cdot \psi \cdot |U|.$$

3. Clearly, $X \cap Y$ and $X \setminus Y$ partition X . For $X \in \mathcal{X}$, we further have that $G[X]$ is a $(\psi \cdot \tau)$ -expander w.r.t. U and thus

$$\min\{|(X \cap Y) \cap U|, |(X \cap U) \setminus Y|\} \cdot (\psi \cdot \tau) \leq w(E(X \cap Y, X \setminus Y)).$$

Taking summation on both sides, we have

$$\psi \cdot \tau \cdot \text{CROSSING}_U(Y, \mathcal{X}) \leq \sum_{X \in \mathcal{X}} w(E(X \cap Y, X \setminus Y)) \leq w(\partial_G Y),$$

where the last inequality follows from the fact that \mathcal{X} is a partition of V . Dividing by $\psi \cdot \tau$ on both sides yields the desired result.

4. Note that the number of crossing clusters X of Y , i.e. clusters where $X \cap Y \cap U$ and $(X \cap U) \setminus Y$ are both non-empty, is upper bounded by $\text{CROSSING}_U(X, Y)$. Thus, by Property 3, it suffices to bound the number of clusters X that intersect with $Y \cap U$ but have $(X \cap U) \setminus Y = \emptyset$ by $\gamma_{\text{expDecomp}} \cdot (\psi \cdot |Y \cap U| + w(\partial_G Y)/\tau)$.

In the case where some cluster X has $X \cap U = Y \cap U$, we can trivially bound this number by one. We thus henceforth assume that all such sets X are proper subsets of Y w.r.t. U , i.e. that $X \cap U \subsetneq Y \cap U$.

Finally, to bound the number of such clusters, we use the flow problem in Definition 6.2. For each such cluster X being a proper subset of Y w.r.t. U , since $Y \cap U$ is τ -connected, we have that $w(\partial_G X) \geq \tau$. Since all edges in ∂X are incident to Y , each such cluster X adds at least τ units of source mass on vertices in Y . Thus, for ℓ such clusters, we have a total of at least $\ell \cdot \tau$ source mass placed on vertices in Y .

On the other hand, since each edge e receives capacity $w(e) \cdot \gamma_{\text{expDecomp}}$ in the flow network, we have that at most $w(\partial_G Y) \cdot \gamma_{\text{expDecomp}}$ units of source flow are routed to sinks outside of Y by the max-flow min-cut theorem. Since the flow is feasible, we thus have that the remaining $\ell \cdot \tau - w(\partial_G Y) \cdot \gamma_{\text{expDecomp}}$ units of flow are routed to vertices in Y . But the sink capacity of vertices in Y is at most $\psi \cdot \tau \cdot \gamma_{\text{expDecomp}} \cdot |Y \cap U|$. Thus, $\ell \cdot \tau - w(\partial_G Y) \cdot \gamma_{\text{expDecomp}} \leq \psi \cdot \tau \cdot \gamma_{\text{expDecomp}} \cdot |Y \cap U|$. Thus, $\ell \leq \gamma_{\text{expDecomp}} \cdot \psi \cdot |Y \cap U| + \gamma_{\text{expDecomp}} \cdot w(\partial_G Y)/\tau$, as desired.

□

6.2 Minimal Gomory-Hu trees

For graph $G = (V, E, w)$, set of terminals $U \subseteq V$ and vertices $r, v \in U$, let $M_{G,v,r}$ denote the v -side vertex-minimal (v, r) -mincut and let $m_{G,U,r}(v) = |M_{G,v,r} \cap U|$. Recall that we denote by $\mathcal{C}_{G,U,\tau}$ the τ -connected components of G w.r.t. U .

Definition 6.5 (Rooted minimal Gomory-Hu Steiner tree). *Given a graph $G = (V, E, w)$ and a set of terminals $U \subseteq V$, a rooted minimal Gomory-Hu U -Steiner tree is a Gomory-Hu U -Steiner tree on U , rooted at some vertex $r \in U$, with the following additional property:*

- For all $t \in U \setminus \{r\}$, consider the minimum-weight edge (u, v) on the unique rt -path in T ; if there are multiple minimum weight edges, let (u, v) denote the one that is closest to t . Let U' be the vertices of the connected component of $T \setminus (u, v)$ containing t . Then, $f^{-1}(U') \subseteq V$ is a minimal (t, r) -mincut, i.e. $f^{-1}(U') = M_{G,t,r}$, and its value is $w_T(u, v)$.

We note that the rooted minimal U -Steiner Gomory-Hu tree always exists for any $r \in U \subseteq V$.

Fact 6.6 (c.f. Theorem A.8 in [AKL⁺22]). *For any graph $G = (V, E, w)$, $U \subseteq V$ and vertex $r \in U$, there exists a minimal Gomory-Hu U -Steiner tree rooted in r .*

We henceforth denote fix one minimal Gomory-Hu U -Steiner tree rooted in r and denote it by $T_{G,U,r}$ along with function $f_{G,U,r}$, or simply $T_{U,r}$ along with function $f_{U,r}$ if G is clear from context. If $U = V$, we simply write T_r along with function f_r to denote the minimal Gomory-Hu Tree rooted in r .

Definition 6.7. For any graph $G = (V, E, w)$, $U \subseteq V$ and vertex $r \in U$, for every vertex $v \in U$, we denote by $c_{G,U,r}(v)$ the vertex in $T_{G,U,r}$ such that the subtree rooted at $c_{G,U,r}(v)$ consists exactly of the vertices of the minimal (v, r) -mincut $M_{G,v,r}$, i.e. $f_{G,U,r}^{-1}(V(T_{G,U,r}[c_{G,U,r}(v)]))$ is the minimal (v, r) -mincut.

Fact 6.8. For any graph $G = (V, E, w)$, $U \subseteq V$ and vertices $r, v \in U$, for any set $A \subseteq U$ with $A \cap M_{G,v,r} = \{v\}$ and $r \in A \setminus \{v\}$, the algorithm COMPUTEISOLATINGCUTS on A from Lemma 3.4 returns the minimal (v, r) -mincut $M_{G,v,r}$ (together in a collection of various cuts).

Proof. From Lemma 3.4, we have that COMPUTEISOLATINGCUTS(A) returns (among other sets) a set S that is a minimal $(v, A \setminus \{v\})$ -mincut. Since $M_{G,v,r}$ contains v but no other vertex in A , we have that it is a $(v, A \setminus \{v\})$ -mincut.

Since every other $(v, A \setminus \{v\})$ cut S also has $v \in A$ and $r \notin S$, the $(v, A \setminus \{v\})$ -mincut value cannot be smaller than $w_G(\partial_G M_{G,v,r})$. Again, since $v \in S, r \notin S$ (since it is in $A \setminus \{v\}$), there cannot be a $(v, A \setminus \{v\})$ -mincut S with $|S| < |M_{G,v,r}|$ as this would contradict that $M_{G,v,r}$ is a minimal (v, r) -mincut.

The fact then follows from submodularity. \square

Fact 6.9. For any graph $G = (V, E, w)$, $U \subseteq V$ and vertex $r \in U$, we have

1. for any threshold $\tau > 0$, $x \in \mathcal{C}_{G,U,\tau}(r)$, then the x -to- r path P in $T_{G,U,r}$ does not cross the cut $\partial_{T_{G,U,r}} \mathcal{C}_{G,U,\tau}(r)$. Further, every edge on P is of weight at least τ .
2. for any threshold $\tau > 0$, $x \notin \mathcal{C}_{G,U,\tau}(r)$, then the x -to- r path in $T_{G,U,r}$ contains exactly one edge e in $\partial_{T_{G,U,r}} \mathcal{C}_{G,U,\tau}(r)$. Further, e is of weight less than τ .
3. for any $x \in U \setminus \{r\}$ and vertex w on the x -to- r path in $T_{G,U,r}$, we have $\lambda_G(v, r) \leq \lambda_G(w, r)$.
4. for any $x \in U \setminus \{r\}$ and vertex w on the x -to- r path in $T_{G,U,r}$, we have that $c_{G,U,r}(w)$ is on the $c_{G,U,r}(x)$ -to- r path.
5. for any $x \in U \setminus \{r\}$ and vertex w on the x -to- r path in $T_{G,U,r}$, we have $M_{G,x,r} \subseteq M_{G,w,r}$, hence $m_{G,U,r}(x) \leq m_{G,U,r}(w)$.
6. for any $x, w \in U \setminus \{r\}$, we have either $M_{G,x,r} \subseteq M_{G,w,r}$, $M_{G,w,r} \subseteq M_{G,x,r}$, or $M_{G,x,r} \cap M_{G,w,r} = \emptyset$.
7. $w(\partial_G f_{G,U,r}^{-1}(\mathcal{C}_{G,U,\tau}(r))) \leq w_{T_{G,U,r}}(\partial_{T_{G,U,r}} \mathcal{C}_{G,U,\tau}(r))$.
8. for any $U' \subseteq U$ with $r \in U'$, $T_{G,U',r}$ can be obtained from $T_{G,U,r}$ via a sequence of edge contractions.

Proof. Let us prove the items one-by-one:

1. Let $y \in P$ be the last vertex in $U \setminus \mathcal{C}_{G,U,\tau}(r)$. Such a vertex must exist as otherwise the path is only supported on vertices in $\mathcal{C}_{G,U,\tau}(r)$ and does not cross the cut $\partial_{T_{G,U,r}} \mathcal{C}_{G,U,\tau}(r)$. But note that since y is not inside $\mathcal{C}_{G,U,\tau}(r)$, we have that $\lambda_{T_{G,U,r}}(r, y) = \lambda_G(r, y) < \tau$. Thus, the segment of P from y to r contains at least one edge of weight less than τ . But this implies that the smallest edge weight on P is smaller τ , and thus $\lambda_G(r, x) = \lambda_{T_{G,U,r}}(r, x) < \tau$ which yields the desired contradiction as x is in the same τ -connected component as r .

2. Let y be the vertex closest on P to x that is inside of $\mathcal{C}_{G,U,\tau}(r)$. By the above item, the segment $P[y, r]$ does not contain a vertex outside $\mathcal{C}_{G,U,\tau}(r)$. Let (x', y) be the edge preceding the segment $P[y, r]$ on P . Since $x' \notin \mathcal{C}_{G,U,\tau}(r)$ by minimality of y , we have that $\lambda_{T_{G,U,r}}(r, x') = \lambda_G(r, x') < \tau$. But again from the previous item, we have that all edges but the edge (x', y) on the segment $P[x', r] = (x', y) \circ P[y, r]$ have weight at least τ , so the weight of (x', y) has to be less than τ , as desired.
3. Clearly, we have $\lambda_G(v, r) = \lambda_{T_{G,U,r}}(v, r) = \min_{e \in T_{G,U,r}[v, r]} w_{T_{G,U,r}}(e) \leq \min_{e \in T_{G,U,r}[w, r]} w_{T_{G,U,r}}(e) = \lambda_{T_{G,U,r}}(w, r) = \lambda_G(w, r)$.
4. For every $v \in V \setminus \{r\}$, the cut vertex $c_{G,U,r}(v)$ is the closest endpoint among all minimum-weight edges on the v -to- r path. This implies the claim immediately.
5. We have $M_{G,x,r} = f_{G,U,r}^{-1}(T_{G,U,r}[c_{G,U,r}(x)]) \subseteq f_{G,U,r}^{-1}(T_{G,U,r}[c_{G,U,r}(w)]) = M_{G,w,r}$.
6. Note that $M_{G,v,r} = f_{G,U,r}^{-1}(T_{G,U,r}[c_{G,U,r}(v)])$ for any $v \in U$. Thus, when $c_{G,U,r}(x)$ is a descendent of $c_{G,U,r}(w)$, we have $M_{G,x,r} \subseteq M_{G,w,r}$, and vice versa. On the other hand, if neither of x, w is the descendent of the other, we have $T_{G,U,r}[c_{G,U,r}(x)] \cap T_{G,U,r}[c_{G,U,r}(w)] = \emptyset$ by the property of minimal mincuts, hence $M_{G,x,r} \cap M_{G,w,r} = \emptyset$.
7. Let A be all endpoints in $\partial_{T_{G,U,r}} \mathcal{C}_{G,U,\tau}(r)$ that are not in $\mathcal{C}_{G,U,\tau}(r)$. Note that for each $u \in A$ with associated endpoint v in $\mathcal{C}_{G,U,\tau}(r)$ contributes $w_{T_{G,U,r}}(u, v)$ to the cut value $w_{T_{G,U,r}}(\partial_{T_{G,U,r}} \mathcal{C}_{G,U,\tau}(r))$. By Properties 1 and 2 (and Definition 2.1), we further have $w_{T_{G,U,r}}(u, v) = \lambda_G(u, v)$.

On the other hand, we have that u is in the subtree $T_{G,U,r}[c_{G,U,r}(u)]$ in $T_{G,U,r} \setminus \{u, v\}$ by Property 2 and r is not as the v -to- r path does not contain an edge from $\partial_{T_{G,U,r}} \mathcal{C}_{G,U,\tau}(r)$ by Property 1. This implies by Definition 6.5 that $f_{G,U,r}^{-1}(V(T_{G,U,r}[c_{G,U,r}(u)])) = M_{G,u,r}$.

It remains to observe that the sets $V(T_{G,U,r}[c_{G,U,r}(u)])$ over all $u \in U$ and $\mathcal{C}_{G,U,\tau}(r)$ partition U and thus the sets $f_{G,U,r}^{-1}(V(T_{G,U,r}[c_{G,U,r}(u)]))$ over all $u \in U$ and $f_{G,U,r}^{-1}(\mathcal{C}_{G,U,\tau}(r))$ partition V .

We conclude that

$$\begin{aligned}
w_{T_{G,U,r}}(\partial_{T_{G,U,r}} \mathcal{C}_{G,U,\tau}(r)) &= \sum_{(u,v) \in \partial \mathcal{C}_{G,U,\tau}(r), u \in A} w_{T_{G,U,r}}(u, v) \\
&= \sum_{u \in A} \lambda_G(u, v) \\
&= \sum_{u \in A} w_G(\partial_G M_{G,u,r}) \\
&= \sum_{u \in A} w_G(\partial_G f_{G,U,r}^{-1}(V(T_{G,U,r}[c_{G,U,r}(u)]))) \\
&\geq \sum_{u \in A} w_G(E_G(f_{G,U,r}^{-1}(V(T_{G,U,r}[c_{G,U,r}(u)])), f_{G,U,r}^{-1}(\mathcal{C}_{G,U,\tau}(r)))) \\
&= w_G(\partial f_{G,U,r}^{-1}(\mathcal{C}_{G,U,\tau}(r))).
\end{aligned}$$

8. By induction, it suffices to prove this claim for $U' = U \setminus \{x\}$ for some vertex $x \in U \setminus \{r\}$. Let (x, y) be the edge incident to x in $T_{G,U,r}$ of largest weight (if multiple such edges exist, take

the one closest to r , if multiple such edges exist, pick an arbitrary such edge). We claim that contracting edge (x, y) and identifying the super-vertex $\{x, y\}$ with y yields $T_{G, U', r}$.

Observe first that for all $v \in U' \setminus \{r\}$, the v -to- r path never has e as the minimum weight edge on the path closest to v . To see this, it is clear that if x is not on the v -to- r path then the claim trivially holds. On the hand, since $x \notin U'$, x is strictly contained on any v -to- r path that contains x , and thus there are two edges incident to x , (z_1, x) and (x, z_2) where we let (z_1, x) be the edge that appears closer to v than (x, z_2) . If (z_1, x) induces the minimal (v, r) -mincut, then its weight is at most equal to the weight of (x, z_2) . But since (z_1, x) is closer to v , this implies that $z_1 \neq y$. On the other hand, if (x, z_2) induces the minimal (v, r) -mincut, then the weight of (x, z_2) is strictly smaller than the weight of (z_1, x) and so $z_2 \neq y$. In either case, the minimal (v, r) -mincut is not induced by (x, y) . But this implies that even after contracting (x, y) , all such minimal (v, r) -mincuts are preserved. Since the obtained tree after contraction is over the vertex set U' , and since minimal (v, r) -mincuts are unique, it follows that the resulting tree is $T_{G, U', r}$, as desired.

□

Theorem 6.10 (Tree-Path Decomposition Lemma, see [ST81]). *Given an undirected n -vertex tree $T = (V, E)$ and a vertex $r \in V$. There is a collection of vertex-disjoint paths $\mathcal{P} = \{P_1, P_2, \dots, P_k\}$ such that*

- $V(P_1), V(P_2), \dots, V(P_k)$ partitions the vertex set V , and
- each path P_i is a sub-path of a v -to- r path $T[v, r]$ for some $v \in V$, and
- for every $v \in V$, the v -to- r path $T[v, r]$ intersects with at most $C_{\text{pathDecomp}} \cdot \log n$ paths from \mathcal{P} for some universal constant $C_{\text{pathDecomp}} > 0$.

6.3 Detecting the largest τ -Connected Component

Lemma 6.11. *Given graph $G = (V, E, w)$, set $U \subseteq V$, and connectivity threshold $\tau > 0$. There is an $m^{1+o(1)}$ time algorithm $\text{DETECTCC}(G, U, \tau)$ that returns set $C \subseteq V$ such that:*

- if the largest τ -connected component w.r.t. U has size at least $\frac{3}{4}|U|$, then C is exactly the largest τ -connected component.
- otherwise, if there is no τ -connected component w.r.t. U that has size at least $\frac{3}{4}|U|$, then $C = \emptyset$.

Algorithm. We present our main algorithm DETECTCC in this section below. The algorithm first finds a set $A \subseteq U$ with size $n^{o(1)}$ that contains at least one vertex in the largest τ -connected component w.r.t. U (Line 2–Line 12). Applying SSMC to each vertex in A then recovers C when $|C| \geq \frac{3}{4}|U|$. To find such a subset A , we start with the active set $A = U$ and iteratively remove at least $\frac{|A|}{n^{o(1)}}$ vertices from A until A is small enough. Concretely, in each iteration, we first detect some cuts that do not cross C via isolating cuts (foreach-loop). If the total size of such cuts is already large enough, we remove such cuts from A and move to the next iteration. Otherwise, we halve vertices in A via an expander decomposition on A . We will prove that whenever we execute the halving step, we also approximately halve $A \cap C$, which guarantees that $A \cap C \neq \emptyset$ after the while-loop.

Algorithm 9: DETECTCC(G, U, τ)

```

1  $A \leftarrow U; \psi \leftarrow 1/(\gamma_{expDecomp} \cdot 100 \log_2(n)); L \leftarrow \frac{100 \log_2(n)}{\psi}.$ 
2 while  $|A| > 1/\psi$  do
3    $\mathcal{S} \leftarrow \text{REMOVELEAVES}(G, A, L).$ 
4    $A' \leftarrow \emptyset.$ 
5   foreach  $S \in \mathcal{S}$  where  $w(\partial S) < \tau$  and  $|S \cap U| < \frac{3}{4}|U|$  do
6      $A' \leftarrow A' \cup (A \cap S).$ 
7     /* If few leaves were identified, it is safe to do a halving step */
8     if  $|A'| < \frac{\psi}{100 \log_2(n)} \cdot |A|$  then
9        $\mathcal{X} \leftarrow \text{COMPUTEXPANDERDECOMP}(G, \psi \cdot \tau, A).$ 
10      foreach  $X \in \mathcal{X}$  do
11        Remove  $\lceil |A \cap X|/2 \rceil$  vertices in  $A \cap X$  from the set  $A$ .
12      else
13         $A \leftarrow A \setminus A'$ 
14      /* Using the SSMC data structure Theorem 2.2, we can detect  $\mathcal{C}_{G,U,\tau}(r)$  for
15      every  $r$ . */
16    if  $\exists r \in A, |\mathcal{C}_{G,U,\tau}(r)| \geq \frac{3}{4}|U|$  then
17      return  $\mathcal{C}_{G,U,\tau}(r).$ 
18  return  $\emptyset.$ 

```

Algorithm 10: REMOVELEAVES(G, A, L)

```

1  $\mathcal{H} \leftarrow \text{CONSTRUCTHITANDMISSFAMILY}(A, L + 1, 2)./* \text{ See Theorem 3.5.}$ 
2  $\mathcal{S} \leftarrow \emptyset.$ 
3 foreach  $h \in \mathcal{H}$  do
4    $A_h \leftarrow \{a \in A \mid h(a) = 1\}.$ 
5    $\{S_v\}_{v \in A_h} \leftarrow \text{COMPUTISOLATINGCUTS}(\{\{v\} \mid v \in A_h\}). /* \text{ See Lemma 3.4.}$ 
6    $\mathcal{S} \leftarrow \mathcal{S} \cup \{S_v\}_{v \in A_h}.$ 
7 return  $\mathcal{S}$ 

```

Correctness Analysis. We can assume henceforth that the largest τ -connected component C in G w.r.t. U indeed has size at least $\frac{3}{4}|U|$ as otherwise, the guarantees on the output of the algorithm are vacuously true from the while-condition. We start by showing that when the else-statement is executed, the number of terminals in C remains untouched.

Claim 6.12. *If the largest τ -connected component C in G w.r.t. U has size $|C| \geq \frac{3}{4}|U|$, then whenever the algorithm executes the else-statement, it cannot remove a vertex from $A \cap C$.*

Proof. Note first that any while-loop iteration that does not execute the if-statement (but only the else-statement) cannot decrease the number of vertices in $A \cap C$ since it only removes vertices in the foreach-loop starting in Line 5 which in turn only removes vertices in cuts S of value less than τ that contain less than $\frac{3}{4}|U|$ vertices in U . Thus, if such a cut S would intersect C , it cannot contain all of C since $|C| \geq \frac{3}{4}|U|$ which in turn implies that the cut value has to be at least τ by the definition of τ -connected components. \square

It remains to prove that in case the if-statement is executed, we stay very close to halving the number of vertices in $C \cap A$.

Claim 6.13. *Consider any execution of Algorithm 10 with parameters G, A, L . Let \mathcal{S} be the set the algorithm returns. Then, for any $r \in A$, we have for every vertex $a \in A$ where $|M_{G,a,r} \cap A| \leq L$, the cut $M_{G,a,r}$ in \mathcal{S} .*

Proof. By assumption, $|(M_{G,a,r} \cap A) \cup \{r\}| \leq L + 1$. Thus, by Theorem 3.5, there is a function $h \in \mathcal{H}$ such that $h(r) = h(a) = 1$ and $h(x) = 0$ for all vertices $x \in M_{G,a,r} \cap A \setminus \{a, r\}$. The claim then follows from Fact 6.8. \square

Claim 6.14. *Consider any time in the algorithm at which the if-condition in Line 7 holds. Then, if $C \cap A \neq \emptyset$, we have $w(\partial_{T_{G,A,r}}(C \cap A)) < \tau \cdot \frac{2\psi}{100 \log_2(n)} \cdot |A|$.*

Proof. Let us fix an arbitrary root $r \in C \cap A$. We partition the vertices in $A \setminus C$ into sets $A_{\leq L} = \{v \in A \setminus C \mid |V(T_{G,A,r}[c_{G,A,r}(v)])| \leq L\}$ and $A_{>L} = \{v \in A \setminus C \mid |V(T_{G,A,r}[c_{G,A,r}(v)])| > L\}$.

We next show that

$$|\partial_{T_{G,A,r}}(C \cap A)| \leq |A|/L + |A_{\leq L}|. \quad (2)$$

Note first that by Fact 6.9 after removing edges $\partial_{T_{G,A,r}}(C \cap A)$ from $T_{G,A,r}$, the vertices in $C \cap A$ are still in one component. Thus, we have that each endpoint $b \in A \setminus C$ of an edge $(b, b') \in \partial_{T_{G,A,r}}(C \cap A)$ is incident to exactly one such edge and thus the subtrees $T_{G,A,r}[b]$ are (vertex-)disjoint. Further, if $|V(T_{G,A,r}[b])| \leq L$, then $b \in A_{\leq L}$ since again by Fact 6.9 all edges on the b' -to- r path have weight at least τ and the edge (b, b') has weight less than τ since $\lambda_G(r, b) < \tau$ from $b \in A \setminus C$. It follows that for each edge in $\partial_{T_{G,A,r}}(C \cap A)$, there are either at least L distinct vertices in the subtree, or one distinct vertex from $A_{\leq L}$. This yields Equation 2.

Next, we upper bound $A_{\leq L}$. From Claim 6.13, we have that for each vertex $v \in A_{\leq L}$, the minimal (v, r) -mincut S_v is contained in \mathcal{S} . Further, since $v \notin C$, we have that $w(\partial_G S_v) < \tau$ and $|S_v \cap U| \leq |A| - |C| \leq |U| - |C| < \frac{3}{4}|U|$. Thus, for each vertex $v \in A_{\leq L}$, we have that v is added to A' or, put differently, $A_{\leq L} \subseteq A'$. By the if-condition, we thus have $|A_{\leq L}| \leq |A'| < \frac{\psi}{100 \log_2(n)} |A|$.

Combining the two inequalities yields

$$|\partial_{T_{G,A,r}}(C \cap A)| < \frac{\psi}{100 \log_2(n)} |A| + \frac{\psi}{100 \log_2(n)} |A| = \frac{2\psi}{100 \log_2(n)} |A|.$$

The claim finally follows by observing that by Fact 6.9, every edge in $\partial_{T_{G,A,r}}(C \cap A)$ has weight less than τ . \square

Claim 6.15. *Consider any execution of the if-statement starting in Line 7. Let C' be the largest τ -connected component C' in G w.r.t. A . Then, if initially $|C'| \geq \frac{1}{2}|A|$, the if-statement removes at most $(\frac{1}{2} + \frac{1}{10\log n})|C'| + 1$ vertices from C' .*

Proof. We can now analyze the decrease in the number of vertices in $A \cap C'$ caused by the halving of the number of vertices in each expander component. Consider any cluster $X \in \mathcal{X}$. If X does not intersect with C' , then it is not relevant to the process. Otherwise, it is not hard to see that the number of vertices in $X \cap C'$ that are removed are at most

$$\frac{1}{2}|C' \cap X| + \text{CROSSING}_A(C', X) + 1,$$

where we add 1 to account for the rounding and $\text{CROSSING}_A(C', X)$ is defined in Fact 6.4. Summing over all sets $X \in \mathcal{X}$, we thus get that the total number of terminals removed from C' is at most

$$\begin{aligned} & \sum_{X \in \mathcal{X}, X \cap C' \neq \emptyset} \frac{1}{2}|C' \cap X| + \text{CROSSING}_A(C', X) + 1 \\ & \leq |C'|/2 + \text{CROSSING}_A(C', \mathcal{X}) + |\{X \in \mathcal{X}, X \cap C'\}| \\ & \leq |C'|/2 + \text{CROSSING}_A(f_{G,A,r}^{-1}(C'), \mathcal{X}) + |\{X \in \mathcal{X}, X \cap C'\}| \\ & \leq |C'|/2 + \tau \cdot \frac{2\psi}{100\log_2(n)} \cdot |A|/(\tau \cdot \psi) + |\{X \in \mathcal{X}, X \cap C'\}| \\ & \leq |C'|/2 + \frac{8}{100\log_2(n)} \cdot |C'| + \gamma_{expDecomp} \cdot \psi \cdot |A| + 1 \\ & < (1/2 + 1/(10\log_2(n)))|C'| + 1 \end{aligned}$$

where we first use that \mathcal{X} properly partitions C' and that overlap between any two sets contributes non-negatively. Then, we use that $A \cap f_{G,A,r}^{-1}(C') = A \cap C'$ since $f_{G,A,r}^{-1}$ works as the identity function on the set A . We then upper bound $\text{CROSSING}_A(f_{G,A,r}^{-1}(C'), \mathcal{X})$ by using that

- $\text{CROSSING}_A(f_{G,A,r}^{-1}(C'), \mathcal{X}) \leq w(\partial f_{G,A,r}^{-1}(C'))/(\tau \cdot \psi)$ by the second property from Fact 6.4,
- $w(\partial_G f_{G,A,r}^{-1}(C')) \leq w(\partial_{T_{G,A,r}} C')$ by Property 7 of Fact 6.9, and
- $w(\partial_{T_{G,A,r}}(C')) < \tau \cdot \frac{2\psi}{100\log_2(n)} \cdot |A|$ by Claim 6.14 where we have $A \cap C'$ by assumption and where we again use that $A \cap f_{G,A,r}^{-1}(C') = A \cap C'$.

Finally, we use $|C'| \geq |A|/2$ in the second term and simplify, and use that every cluster X that crosses C' has value at least τ which implies that the number of such clusters is $\gamma_{expDecomp} \cdot \psi \cdot |A|$ by the first property of Fact 6.4, and there is at most one cluster $X \in \mathcal{X}$ that does not cross C' and intersects it, namely a cluster X with $C' \subseteq X$. \square

The above claim shows that the if-condition in Line 7 does not remove too many vertices in $C \cap A$. We then conclude the correctness of the algorithm with the following claim.

Claim 6.16. *If the largest τ -connected component C in G w.r.t. U has size $|C| \geq \frac{3}{4}|U|$, then the algorithm correctly outputs set C . Otherwise, it outputs \emptyset .*

Proof. From Claim 6.12, we have that in this case, while-loop iteration that enters the else-statement cannot decrease the number of active terminals A in C .

Next, note that every execution of the if-statement at least halves the number of active terminals A . Thus, after $R = \lceil \log_2(\psi|U|) \rceil$ if-statement executions, we have that A is of size at most $|U|/2^R \leq \psi|U|$. We next prove by induction that after the i -th execution of the if-statement for any $1 \leq i \leq R$, $|C \cap A| \geq \frac{3}{4} \cdot (\frac{1}{2} - \frac{1}{5\log n})^i |U|$.

Observe that for $1 \leq i+1 \leq R$, we have either from the initial assumption or by induction, that before the $(i+1)$ -th if-statement is executed, we have $|C \cap A| \geq \frac{3}{4} \cdot 2^{-i} \cdot (1 - \frac{2}{5\log n})^i |U| \geq \frac{3}{4} \cdot 2^{-i} \cdot e^{-R/(5\log n)} |U| \geq \frac{3}{4} \cdot 2^{-i} \cdot e^{-1/5} |U| > 2^{-(i+1)} \cdot |U| > 10 \log(n)$ using $e^x \leq 1 + 2x$ for $x \leq 1$, and that for n sufficiently large, we have both $R \leq \log n$ and $R+1 \leq \log(\sqrt{\psi}|U|)$ and $\sqrt{\psi} < 1/(10 \log n)$. At the same time, since each execution at least halves the number of vertices in A , and else-statement executions monotonically decrease the set A , we further also have $|A| \leq 2^{-i} |U|$. Thus, before any such $(i+1)$ -th iteration, we have $|C \cap A| \geq |A|/2$.

We therefore can use Claim 6.15 yielding that the size of $C \cap A$ decreases by at most

$$\left(\frac{1}{2} + \frac{1}{10 \log(n)} \right) |C \cap A| + 1 \leq \left(\frac{1}{2} + \frac{1}{10 \log(n)} \right) |C \cap A| + \frac{1}{10 \log n} \cdot |C \cap A| = \left(\frac{1}{2} + \frac{1}{5 \log n} \right) |C \cap A|.$$

Thus, the number of remaining vertices is at least a $(\frac{1}{2} - \frac{1}{5\log n})$ -fraction, as desired.

It follows that after the R -th execution of the if-statement, we have $A \cap C \neq \emptyset$ and $|A| \leq 1/\psi$. As the latter satisfies the while-loop condition, we thus have that the algorithm returns A with $A \cap C \neq \emptyset$, as desired. \square

Runtime Analysis. Finally, we turn to the runtime analysis.

Claim 6.17. *Algorithm 9 runs for at most $\tilde{O}(1/\psi)$ iterations.*

Proof. In each iteration, either the set A decreases in size by a factor $\tilde{\Omega}(1/\psi)$ or it is halved, i.e. A always decreases by at least a $\tilde{\Omega}(1/\psi)$ -factor. Thus, after $\tilde{O}(1/\psi)$ iterations A is of size less than 1 which forces the while-loop (and thus the algorithm) to terminate. \square

Claim 6.18. *Each iteration of Algorithm 9 takes time $m^{1+o(1)}/\psi^{O(1)}$.*

Proof. Each iteration in Algorithm 9 calls COMPUTEXPANDERDECOMP and Algorithm 10 once. By Theorem 6.3, COMPUTEXPANDERDECOMP takes time $m^{1+o(1)}$. For Algorithm 10, we note that $L = \tilde{O}(1/\psi)$. It then follows from Theorem 3.5 that CONSTRUCTHITANDMISSFAMILY takes time $\tilde{O}(m^{1+o(1)}L^{O(1)}) = m^{1+o(1)}/\psi^{O(1)}$ and the size of \mathcal{H} is $m^{o(1)}/\psi^{O(1)}$. Since Algorithm 10 calls COMPUTEISOLATINGCUTS for each function in \mathcal{H} and each call of COMPUTEISOLATINGCUTS takes time $m^{1+o(1)}$. The total time cost of Algorithm 10 is $m^{1+o(1)}/\psi^{O(1)}$. \square

6.4 Decomposition of $U \setminus \mathcal{C}_{G,U,\tau}(r)$

Lemma 6.11 allows us to find our desired threshold cut τ by a binary search. In Section 6.4 and Section 6.5, we assume that we have a pivot r and a threshold τ such that $\mathcal{C}_{G,U,\tau}(r) \geq \frac{3}{4}|U|$ and $\max_{x \in U} |\mathcal{C}_{G,U,\tau+1}(x)| < \frac{3}{4}|U|$. For simplicity, we will write $C = \mathcal{C}_{G,U,\tau}(r)$ throughout the

two subsections. In this section, we propose an algorithm to find isolating cut families such that $\text{polylog}(n)$ calls of the algorithm suffice to cover $U \setminus C$.

We then present our algorithm for decomposition $U \setminus \mathcal{C}_{G,U,\tau}$ below.

Algorithm 11: DECOMPFIRSTSTEP(G, A, r, τ)

```

1  $\psi \leftarrow 1/(\gamma_{expDecomp} \cdot 20 \log_2(n)); L \leftarrow 1000 \log_2(nW)/\psi; \mathcal{S} \leftarrow \emptyset.$ 
/* Each path size reduction needs to be tailored to the mincut value that
separates intervals */
2 foreach  $\tau' \in \{2^0, 2^1, \dots, 2^{\lceil \log \tau \rceil}\}$  do
3    $A' \leftarrow A.$ 
   /* For each "sampling threshold" try the isolating cut lemma. */
4   for  $k = 1, \dots, \log_2 n$  do
5      $\mathcal{S} \leftarrow \mathcal{S} \cup \text{REMOVELEAFFIRSTSTEP}(A', r, \tau, L).$ 
6      $\mathcal{X} \leftarrow \text{COMPUTEXPANDERDECOMP}(G, \psi \cdot \tau', A').$ 
7     foreach  $X \in \mathcal{X}$  do
8       | Remove  $\lceil |A'|/2 \rceil$  vertices from  $A' \cap X$  from the set  $A'$ .
   /* For all cuts that were pruned off, remove the contained terminals from
the active vertex set. */
9 return  $\mathcal{S}$ 

```

Algorithm 12: REMOVELEAFFIRSTSTEP(A', r, τ, L)

```

1  $\mathcal{S}' \leftarrow \emptyset.$ 
2  $\mathcal{H} \leftarrow \text{CONSTRUCTHITANDMISSFAMILY}(A', L, 2)./* \text{ See Theorem 3.5.}$ 
3 foreach  $h \in \mathcal{H}$  do
4    $A_h \leftarrow \{a \in A' \mid h(a) = 1\} \cup \{r\}.$ 
5    $\{S_v\}_{v \in A_h} \leftarrow \text{COMPUTEISOLATINGCUTS}(\{\{v\} \mid v \in A_h\}). /* \text{ See Lemma 3.4.}$ 
   /* If the cut is an  $(v, r)$ -mincut, then add it to  $\mathcal{S}'$ 
6   foreach  $v \in A_h \setminus \{r\}$  where  $w(\partial S_v) < \tau$  and  $\lambda_G(r, v) = w(\partial S_v)$  do
7     |  $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{(S_v, v, r)\}.$ 
8 return  $\mathcal{S}'$ 

```

Correctness of Algorithm 11. To understand and analyze the algorithm, we introduce a distance function on $T_{G,U,r}$ based on its path decomposition in Theorem 6.10.

Definition 6.19 (Distance in GH-tree). *Let $T_{G,U,r}$ be the rooted minimal Gomory-Hu U-Steiner tree rooted at r (as described in Definition 6.5). Let \mathcal{P} be a tree-path decomposition of the tree $T_{G,U,r}$ obtained from Theorem 6.10. For any vertex $u \in U$, we let $d_{\mathcal{P}}(u)$ denote the number of paths in \mathcal{P} that intersect the $T_{G,U,r}[c_{G,U,r}(u), r]$ path in at least one vertex.*

By Theorem 6.10, the distance function is bounded by $O(\log n)$. Towards the end of this section, we will show each call of Algorithm 11 returns a collection \mathcal{S} that covers half of vertices in A with the largest distance. Therefore, for any $A \subseteq U \setminus C$, $\log_2 n$ calls of Algorithm 11 removes all vertices in A with the largest distance.

We first note that our algorithm can find an isolating mincut family for $v \in U \setminus C$ if $|M_{G,v,r} \cap A'| \leq L$.

Claim 6.20. *An invocation of REMOVELEAFFIRSTSTEP(A', r, τ, L) returns in deterministic time $m^{1+o(1)} \cdot L$ a set \mathcal{S}' such that*

- *every triple $(S, v, r) \in \mathcal{S}'$ has $v \in A'$ and S is the minimal (v, r) -mincut, and*
- *for every vertex $v \in A'$ with $\lambda_G(v, r) < \tau$ and $|M_{G,v,r} \cap A'| \leq L$, then the triple $(M_{G,v,r}, v, r)$ is present in \mathcal{S}' .*

Proof. The first result follows by condition $\lambda(r, v) = w(\partial_G S_v)$ in Algorithm 14. For the second result, by Theorem 3.5, there exists some A_h in the outer foreach-loop such that $r \in A_h$ and $|A_h \cap M_{G,v,r}| = \{v\}$, hence applying isolating mincut algorithm to A_h recovers $M_{G,v,r}$. \square

The following technical lemma establishes the effectiveness of the halving technique in Algorithm 11 and Algorithm 13.

Lemma 6.21. *Let $A_0, Y \subseteq V$ be two vertex sets, τ' be a positive integer and let $\psi = 1/(20 \log n \cdot \gamma_{expDecomp})$. We recursive define A_{i+1} as follows. Let $\mathcal{X}_i \leftarrow \text{COMPUTEXPANDERDECOMP}(G, \psi \cdot \tau', A_i)$, then remove any $\lceil |A_i \cap X|/2 \rceil$ vertices from $A_i \cap X$ from the set A_i for each $X \in \mathcal{X}_i$ and denote the set of remaining vertices in A_i by A_{i+1} . Then*

1. Upper bound: $|Y \cap A_k| \leq 2^{-k} \cdot |Y \cap A_0| + 2 \cdot w(\partial_G Y)/(\tau' \cdot \psi)$ for $k > 0$.
2. Lower bound: When $Y \cap A_0$ is τ -connected and $k \leq \log_2 \left(\frac{|Y \cap A_0|}{160 \log_2(n) \cdot \max\{1, w(\partial_G Y)/(\tau' \cdot \psi)\}} \right)$, we have $|Y \cap A_k| \geq \left(\frac{1}{2} - \frac{1}{10 \log n} \right)^k |Y \cap A_0|$.

Proof. Upper bound: Note that we remove at least $|A_i \cap X \cap Y|/2 + \text{CROSSING}_{A_i}(Y, X)$ from $A_i \cap X \cap Y$ for each $X \in \mathcal{X}_i$. So the total decrease in the number of vertices from A_i to A_{i+1} is at least $|Y \cap A_i|/2 - \text{CROSSING}_{A_i}(Y, \mathcal{X}_i) \geq |Y \cap A_i|/2 - w(\partial_G Y)/(\tau' \cdot \psi)$. We then have $|Y \cap A_{i+1}| \leq |Y \cap A_i|/2 + w(\partial_G Y)/(\tau' \cdot \psi)$. Directly computing the recursion gives

$$|Y \cap A_k| \leq 2^{-k} \cdot |Y \cap A_0| + \sum_{i=1}^k 2^{1-i} w(\partial_G Y)/(\tau' \cdot \psi) \leq 2^{-k} \cdot |Y \cap A_0| + 2w(\partial_G Y)/(\tau' \cdot \psi).$$

Lower bound: We prove by induction. Suppose we have

$$|Y \cap A_i| \geq \left(\frac{1}{2} - \frac{1}{10 \log n} \right)^i |Y \cap A_0|.$$

for some $i < k$. By assumption on k , we further have

$$\begin{aligned} |Y \cap A_i| &\geq 2^{-\log_2 \left(\frac{|Y \cap A_0|}{160 \log_2(n) \cdot w(\partial_G Y)/(\tau' \cdot \psi)} \right)} \left(1 - \frac{1}{5 \log n} \right)^{\log_2(n)} \cdot |Y \cap A_0| \\ &> 80 \log_2(n) \cdot w(\partial_G Y)/(\tau' \cdot \psi). \end{aligned}$$

Similar to the upper bound case, we remove at most $|A_i \cap X \cap Y|/2 + \text{CROSSING}_{A_i}(Y, X) + 1$ from $A_i \cap X \cap Y$ for each $X \in \mathcal{X}_i$, where the one extra vertex is due to the rounding. So the total decrease in the number of vertices from A_i to A_{i+1} is at most

$$\begin{aligned}
& \frac{|Y \cap A_i|}{2} + \text{CROSSING}_{A_i}(Y, \mathcal{X}_i) + |\{X \in \mathcal{X}_i : X \cap Y \neq \emptyset\}| \\
& \leq \frac{|Y \cap A_i|}{2} + \frac{w(\partial_G Y)}{\tau' \cdot \psi} + \frac{w(\partial_G Y)}{\tau' \cdot \psi} + \gamma_{expDecomp} \cdot \psi \cdot |Y \cap A_i| + \frac{\gamma_{expDecomp} \cdot w(\partial_G Y)}{\tau'} \\
& \leq |Y \cap A_i|/2 + \frac{4w(\partial_G Y)}{\tau' \cdot \psi} + \gamma_{expDecomp} \cdot \psi \cdot |Y \cap A_i| \\
& \leq |Y \cap A_i|/2 + 4 \cdot \frac{|Y \cap A_i|}{80 \log_2(n)} + \gamma_{expDecomp} \cdot \psi \cdot |Y \cap A_i| \\
& = \left(\frac{1}{2} + \frac{1}{20 \log_2(n)} + \gamma_{expDecomp} \cdot \psi \right) |Y \cap A_i| \\
& = \left(\frac{1}{2} + \frac{1}{10 \log_2(n)} \right) |Y \cap A_i|.
\end{aligned}$$

It then follows that $|Y \cap A_{i+1}| \leq \left(\frac{1}{2} - \frac{1}{10 \log n} \right) |Y \cap A_i| \leq \left(\frac{1}{2} - \frac{1}{10 \log n} \right)^k |Y \cap A_0|$. \square

We finally claim that calling Algorithm 11 on $A \subseteq U \setminus C$ can always cover at least half vertices in A with the largest distance.

Claim 6.22. *An invocation of DECOMPFIRSTSTEP(G, A, r, τ), for $G = (V, E, w)$ and $A \subseteq V \setminus C$, returns in deterministic time $m^{1+o(1)}$ a set \mathcal{S} such that*

- every triple $(S, v, r) \in \mathcal{S}$ has $v \in A$ and S is the minimal (v, r) -mincut, and
- letting $A_{max} = \{a \in A \mid d_P(a) = \max_{a' \in A} d_P(a')\}$ be the set of vertices in A at maximum distance from r , we have

$$|\cup_{(S, v, r) \in \mathcal{S}} S \cap A_{max}| \geq |A_{max}|/2$$

i.e. DECOMPFIRSTSTEP(G, A, r, τ) covers at least a half of A_{max} .

Proof. The first claim is immediate from Claim 6.20. Let $d_{max} = \max_{a' \in A} d_P(a')$. For $P \in \mathcal{P}$, define $A_P := \{a \in A_{max} \mid c_{G, A \cup \{r\}, r}(a) \in V(P)\}$. Since \mathcal{P} decomposes $T_{G, U, r}$, the sets $\{A_P\}_{P \in \mathcal{P}}$ is a partition of A_{max} . It then suffices to prove $|\cup_{(S, v, r) \in \mathcal{S}} S \cap A_P| \geq |A_P|/2$ for $P \in \mathcal{P}$. The case $A_P = \emptyset$ is trivial, so let $P \in \mathcal{P}$ be a path such that $A_P \neq \emptyset$ in the rest part of the proof.

We call a vertex $v \in T_{G, A \cup \{r\}, r}$ a cut vertex if $v = c_{G, A \cup \{r\}, r}(u)$ for some vertex $u \in U$. We decreasingly order all cut vertices in P by their depth in the rooted tree $T_{G, A \cup \{r\}, r}$ and denote them by $\{p_1, \dots, p_l\}$. Let $V_i := T_{G, A \cup \{r\}, r}[c_{G, A \cup \{r\}, r}(p_i)]$ for $i = 1, \dots, l$ and let $A_0 = \emptyset$. Then by Fact 6.9(4), we have $\emptyset = V_0 \subsetneq V_1 \subsetneq \dots \subsetneq V_l = A_P$. Let

$$i_0 := \min\{i \leq l : |V_i| \geq |A_P|/2\} - 1.$$

Let

$$i_k := \max\{i_0 < i \leq l : \lambda(r, p_i) \leq 2^k\} \quad \text{for } k = 1, \dots, \lceil \log_2 \tau \rceil.$$

By Fact 6.9(3), $\lambda(p_i, r)$ increases with i , so we have $i_0 < i_1 \leq \dots \leq i_{\lceil \log_2 \tau \rceil}$. Since $V_{i_{\lceil \log_2 \tau \rceil}} = V_l = A_P$, the set family $\{V_{i_k} \setminus V_{i_{k-1}} : k \geq 1\}$ forms a partition of $A_P \setminus V_{i_0}$, whose size is at least $|A_P|/2$,

by definition of i_0 . Therefore, there exists some i_k such that $|V_{i_k} \setminus V_{i_{k-1}}| \geq |A_P \setminus V_{i_0}|/\lceil \log_2 \tau \rceil > |A_P|/(2 \log_2(nW))$. Note that vertices in $V_{i_k} \setminus V_{i_{k-1}}$ have cut vertices in $\{p_{i_{k-1}+1}, \dots, p_{i_k}\}$. It then follows from Fact 6.9(4) that for any $v \in V_{i_k} \setminus V_{i_{k-1}}$, we have $M_{G,v,r} \supseteq V_{i_{k-1}+1} \supseteq V_{i_0+1}$, whose size is at least $|A_P|/2$ by definition of i_0 . Let $\tau' = 2^{k-1}$ in the rest of the proof. Consider the the foreach-loop with τ' in Algorithm 11, we denote by A_j the set A' after the j -th iteration in the for-loop. By Claim 6.20, it suffices to find some $j \leq \log_2 n$ such that

$$|(V_{i_k} \setminus V_{i_{k-1}}) \cap A_j| > 0 \text{ and } |V_{i_k} \cap A_j| \leq L.$$

Let $Y = f_{G,A \cup \{r\},r}^{-1}(V_{i_k} \setminus V_{i_{k-1}})$ and $Z = f_{G,A \cup \{r\},r}^{-1}(V_{i_k})$. We then need to show, equivalently, that

$$|Y \cap A_j| > 0 \text{ and } |Z \cap A_j| \leq L.$$

When $|Z \cap A_0| \leq L$, we are done. We then assume $|Z \cap A_0| > L$. We will apply the two bounds in Lemma 6.21 to Z and Y respectively. We start by bounding $w(\partial_G Z)$ and $w(\partial_G Y)$ and verifying τ -connectivity of $Y \cap A_0$.

- By definition of V_{i_k} , we have $w(\partial_G Z) = \lambda(r, p_{i_k}) \leq 2\tau'$.
- Since $Y = f_{G,A \cup \{r\},r}^{-1}(V_{i_k} \setminus V_{i_{k-1}}) = f_{G,A \cup \{r\},r}^{-1}(V_{i_k}) \setminus f_{G,A \cup \{r\},r}^{-1}(V_{i_{k-1}})$, we have

$$w(\partial_G Y) \leq w(\partial_G f_{G,A \cup \{r\},r}^{-1}(V_{i_k})) + w(\partial_G f_{G,A \cup \{r\},r}^{-1}(V_{i_{k-1}})) \leq 2\tau' + \tau' = 3\tau'.$$

- For any two vertices in $Y \cap A_0 = V_{i_k} \setminus V_{i_{k-1}}$, we have $\lambda(x, r), \lambda(y, r) \geq \tau'$, it then follows by transitivity of λ that $\lambda(x, y) \geq \tau$, so $Y \cap A_0$ is τ -connected.

Applying Lemma 6.21 with $j = \lfloor \log_2 \left(\frac{|Y \cap A_0|}{160 \log_2(n) \cdot (3/\psi)} \right) \rfloor$ to Z and Y then gives us

$$\begin{aligned} |Z \cap A_j| &\leq \frac{160 \log_2 n \cdot (3/\psi)}{|Y \cap A_0|} \cdot |Z \cap A_0| + \frac{2w(\partial_G Z)}{\psi \cdot \tau'} \leq \frac{960 \log_2 n \cdot \log_2(nW)}{\psi} + \frac{4}{\psi} \leq L \\ |Y \cap A_j| &\geq \frac{160 \log_2 n \cdot (3/\psi)}{|Y \cap A_0|} \cdot \left(1 - \frac{1}{5 \log n}\right)^{\log_2 n} \cdot |Y \cap A_0| \geq \frac{240 \log_2 n}{\psi} > 0, \end{aligned}$$

concluding the proof. \square

Runtime of Algorithm 11. Again, bounding the runtime of the algorithm is relatively straightforward.

Claim 6.23. *Algorithm 11 takes time $m^{1+o(1)}$.*

Proof. It suffices to show each inner iteration takes time $m^{1+o(1)}$. Note that each iteration in Algorithm 11 calls COMPUTEXPANDERDECOMP and Algorithm 12 once. By Theorem 6.3, COMPUTEXPANDERDECOMP takes time $m^{1+o(1)}$. For Algorithm 12, we note that $L = \tilde{O}(1/\psi)$. It then follows from Theorem 3.5 that CONSTRUCTHITANDMISSFAMILY takes time $\tilde{O}(m^{1+o(1)}L^{O(1)}) = m^{1+o(1)}/\psi^{O(1)}$ and the size of \mathcal{H} is $m^{o(1)}/\psi^{O(1)}$. Since Algorithm 12 calls COMPUTEISOLATINGCUTS and SSMC once for each function in \mathcal{H} , and each call of COMPUTEISOLATINGCUTS and SSMC takes time $m^{1+o(1)}$. The total time cost of Algorithm 12 is $m^{1+o(1)}/\psi^{O(1)}$. \square

6.5 Decomposition of $\mathcal{C}_{G,U,\tau}(r)$

Let $B = \{v \in C \setminus \mathcal{C}_{G,U,\tau+1}(r) : m_{G,U,r}(v) \leq \frac{15}{16}|U|\}$. We claim that the size of B is $\Theta(|U|)$ when C is large enough, justifying our choice of r as an arbitrary vertex in C .

Claim 6.24. *Suppose $|C| \geq \frac{15}{16}|U|$ and the size of the largest $(\tau + 1)$ -connected component of G w.r.t. U is smaller than $\frac{3}{4}|U|$, then $|B| \geq \frac{1}{8}|U|$.*

Proof. Denote the complement of B in $C \setminus \mathcal{C}_{G,U,\tau+1}(r)$ by D . If $D = \emptyset$, we have

$$|B| = |C \setminus \mathcal{C}_{G,U,\tau+1}(r)| \geq \frac{15}{16}|U| - \frac{3}{4}|U| \geq \frac{1}{8}|U|.$$

We may then assume $D \neq \emptyset$. By Fact 6.9(6), for any two vertices $x, y \in D$, we have either $M_{G,x,r} \subseteq M_{G,y,r}$, $M_{G,x,r} \subseteq M_{G,y,r}$, or $M_{G,x,r} \cap M_{G,y,r} = \emptyset$. However, the last case can never happen since $|M_{G,x,r} \cap U|, |M_{G,y,r} \cap U| \geq \frac{15}{16}|U|$. So the collection $\{M_{G,x,r} : x \in D\}$ is nesting. Let $M_{G,v,r}$ be a minimal set in the collection and let $v' = c_{G,U,r}(v)$, then $M_{G,v,r} = f_{G,U,r}^{-1}(T_{G,U,r}[v'])$. We claim $M_{G,v,r} \cap C \setminus \mathcal{C}_{G,U,\tau+1}(v') \subseteq B$, hence

$$|B| \geq |M_{G,v,r} \cap C \setminus \mathcal{C}_{G,U,\tau+1}(v')| \geq |M_{G,v,r} \cap U| - |U \setminus C| - |\mathcal{C}_{G,U,\tau+1}(v')| \geq \frac{15}{16}|U| - \frac{1}{16}|U| - \frac{3}{4}|U| = \frac{1}{8}|U|.$$

It remains to prove $M_{G,v,r} \cap \mathcal{C}_{G,U,\tau}(r) \setminus \mathcal{C}_{G,U,\tau+1}(v') \subseteq B$. Let $x \in M_{G,v,r} \cap \mathcal{C}_{G,U,\tau}(r) \setminus \mathcal{C}_{G,U,\tau+1}(v')$. It follows from Fact 6.9(6) that $x \in M_{G,x,r} \cap U \subseteq M_{G,v,r} \cap U = T_{G,U,r}[v']$. Since $x \notin \mathcal{C}_{G,U,\tau+1}(v')$, we have $\lambda_G(x, v) = \tau$, so there exists an edge e with weight τ on the path connecting x, v' in $T_{G,U,r}$. Cutting this edge and taking the x -side yields a subtree of $T_{G,U,r}[v']$, whose preimage under $f_{G,U,r}$ is a (x, v') -mincut with weight τ . Let W denote the x -side of this cut. Since v' is the root of the tree $T_{G,U,r}[v']$, the edge e cannot be on the path $T_{G,U,r}[v, r]$, so $r \in V \setminus W$ is on the same side with v' . Since $x \in C \setminus \mathcal{C}_{G,U,\tau+1}(r)$, we have $\lambda_G(x, r) = \tau$. Therefore, W is also an (x, r) -mincut. By minimality of $M_{G,x,r}$, we have

$$M_{G,x,r} \cap U \subseteq W \cap U \subsetneq M_{G,v,r} \cap U.$$

Recall that $M_{G,v,r}$ is the minimal set in $\{M_{G,y,r} : y \in D\}$, we consequently have $x \notin D$, i.e., $x \in B$. \square

Let \mathcal{P} be the path decomposition of $T_{G,U,r}$ as in Theorem 6.10 and let $P \in \mathcal{P}$ be any path in this decomposition. Consider $x, y \in P$ with y closer to r . By Fact 6.9(1), if $x \in \mathcal{C}_{G,U,\tau+1}(r)$, we have $y \in \mathcal{C}_{G,U,\tau+1}(r)$. By Fact 6.9(5), if $m_{G,U,r}(x) > \frac{15}{16}|U|$, we have $m_{G,U,r}(y) > \frac{15}{16}|U|$. Therefore, if one vertex $x \in P = T_{G,U,r}[r', r'']$ (where r' is closer to r) is in $C \setminus B$, then $T_{G,U,r}[r', x] \subseteq C \setminus B$. For any path P crossing $C \setminus B$, we can then split it into two path P', P'' such that $P' \cap (C \setminus B) = \emptyset$ and $P'' \subseteq C \setminus B$. Let \mathcal{P}' denote the path set obtained by splitting all paths in \mathcal{P} crossing $C \setminus B$ and then removing all paths in $C \setminus B$. Then \mathcal{P}' forms a partition of $U \setminus (C \setminus B)$, and it still satisfies the second properties in Theorem 6.10. We can then define the distance function $d_{\mathcal{P}'}$ for vertices in B , as Definition 6.19. Note that we have $d_{\mathcal{P}'}(v) \leq d_{\mathcal{P}}(v) \leq C_{pathDecomp} \cdot \log n$ by construction.

The following two results are similar to Claim 6.20 and Claim 6.22, which show calling Algorithm 13 to A always cover at least half vertices in $B \cap A$ at largest distance.

Claim 6.25. *An invocation of REMOVELEAFSECONDSTEP(A', r, τ, L) returns in deterministic time $m^{1+o(1)} \cdot L$ a set \mathcal{S}' such that*

Algorithm 13: DECOMPSECONDSTEP(G, A, r, τ)

```

1  $\psi \leftarrow 1/(\gamma_{expDecomp} \cdot 20 \log_2(n)); L \leftarrow 1000 \log_2^2(nW)/\psi; \mathcal{S} \leftarrow \emptyset.$ 
2 for  $k = 1, \dots, \log_2 n$  do
3    $\mathcal{S} \leftarrow \mathcal{S} \cup \text{REMOVELEAFSECONDSTEP}(A, r, \tau, L).$ 
4    $\mathcal{X} \leftarrow \text{COMPUTEXPANDERDECOMP}(G, \psi \cdot \tau, A).$ 
5   foreach  $X \in \mathcal{X}$  do
6     | Remove  $\lceil |A \cap X|/2 \rceil$  vertices from  $A \cap X$  from the set  $A$ .
/* For all cuts that were pruned off, remove the contained terminals from
the active vertex set. */
```

7 **return** \mathcal{S}

Algorithm 14: REMOVELEAFSECONDSTEP(A', r, τ, L)

```

1  $\mathcal{S}' \leftarrow \emptyset.$ 
2  $\mathcal{H} \leftarrow \text{CONSTRUCTHITANDMISSFAMILY}(A', L, 2)./* \text{ See Theorem 3.5.} */$ 
3 foreach  $h \in \mathcal{H}$  do
4    $A_h \leftarrow \{a \in A' \mid h(a) = 1\} \cup \{r\}.$ 
5    $\{S_v\}_{v \in A_h} \leftarrow \text{COMPUTEISOLATINGCUTS}(\{\{v\} \mid v \in A_h\}). /* \text{ See Lemma 3.4.} */$ 
/* If the cut is an  $(v, r)$ -mincut with small size, then add it to  $\mathcal{S}'$  */
6   foreach  $v \in A_h \setminus \{r\}$  where  $w(\partial S_v) = \tau$  and  $|S_v \cap U| \leq \frac{15}{16}|U|$  do
7     |  $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{(S_v, v, r)\}.$ 
8 return  $\mathcal{S}'$ 


---



```

- every triple $(S, v, r) \in \mathcal{S}'$ has $v \in B \cap A'$ and S is the minimal (v, r) -mincut, and
- for every vertex $v \in B \cap A'$ and $|M_{G, v, r} \cap A'| \leq L$, then the triple $(M_{G, v, r}, v, r)$ is present in \mathcal{S}' .

Proof. We mark that $v \in B$ implies $w(\partial_G S_v) = \tau$ and $|S_v| \leq \frac{15}{16}|U|$. The remaining proof is the same as Claim 6.20. \square

Claim 6.26. An invocation of DECOMPSECONDSTEP(G, A, r, τ), for $G = (V, E, w)$ and $A \subseteq C$, returns in deterministic time $m^{1+o(1)}$ a set \mathcal{S} such that

- every triple $(S, v, r) \in \mathcal{S}$ has $v \in B \cap A$ and S is the minimal (v, r) -mincut, and
- let $A_{max} = \{a \in B \cap A \mid d_{\mathcal{P}'}(a) = \max_{a' \in B \cap A} d_{\mathcal{P}'}(a)\}$ be the set of vertices in $B \cap A$ at maximum distance from r , we have

$$|\cup_{(S, v, r) \in \mathcal{S}} S \cap A_{max}| \geq |A_{max}|/2$$

i.e. at least a constant fraction of vertices from A_{max} are removed by the cuts in \mathcal{S} .

Proof. The first claim follows immediately from Claim 6.25. Let $d_{max} = \max_{a' \in B \cap A} d_{\mathcal{P}'}(a')$. For $P \in \mathcal{P}'$, define $A_P := \{a \in A_{max} \mid c_{G, A \cup \{r\}, r}(a) \in V(P)\}$. Similar to the proof of Claim 6.22, it suffices to prove $|\cup_{(S, v, r) \in \mathcal{S}} S \cap A_P| \geq |A_P|/2$ for $P \in \mathcal{P}'$. Let $P \in \mathcal{P}'$ be a path such that $A_P \neq \emptyset$. Note that $A_P \subseteq B$ since all paths in \mathcal{P}' do not intersect $C \setminus B$.

Decreasingly order all cut vertices in P by their depth in the rooted tree $T_{G,A\cup\{r\},r}$ and denote them by $\{p_1, \dots, p_l\}$. Let $V_i := T_{G,A\cup\{r\},r}[c_{G,A\cup\{r\},r}(p_i)]$ for $i = 1, \dots, l$ and let $V_0 = \emptyset$. Then by Fact 6.9(4), we have $\emptyset = V_0 \subsetneq V_1 \subsetneq \dots \subsetneq V_l = A_P$. Let

$$i_0 := \min\{i \leq l : |V_i| \geq |A_P|/2\} - 1.$$

It then follows that $|V_{i_0+1}| \geq |A_P|/2$ and for any vertex $v \in V_P \setminus V_{i_0}$, we have $M_{G,v,r} \supseteq V_{i_0+1}$. We denote by A_j the set A after the j -th iteration in the for-loop. By Claim 6.25, it suffices to find some $j \leq \log_2 n$ such that

$$|(V_P \setminus V_{i_0}) \cap A_j| > 0 \text{ and } |V_P \cap A_j| \leq L.$$

Let $Y = f_{G,A\cup\{r\},r}^{-1}(V_P \setminus V_{i_0})$ and $Z = f_{G,A\cup\{r\},r}^{-1}(V_P)$. We then need to show, equivalently, that

$$|Y \cap A_j| > 0 \text{ and } |Z \cap A_j| \leq L.$$

When $|Z \cap A_0| \leq L$, we are done. We then assume $|Z \cap A_0| > L$. We will apply the two bounds in Lemma 6.21 to Z and Y respectively. Similar to the proof of Claim 6.22, we have

- $w(\partial_G Z) = \lambda(r, p_l) = \tau$;
- $w(\partial_G Y) \leq w(\partial_G f_{G,A\cup\{r\},r}^{-1}(V_P)) + w(\partial_G f_{G,A\cup\{r\},r}^{-1}(V_{i_0})) = 2\tau$;
- τ -connectivity of $Y \cap A$ follows from $A \subseteq C$.

Applying Lemma 6.21 with $j = \lfloor \log_2 \left(\frac{|Y \cap A_0|}{160 \log_2(n) \cdot (2/\psi)} \right) \rfloor$ to Z and Y then gives us

$$\begin{aligned} |Z \cap A_j| &\leq \frac{160 \log_2 n \cdot (2/\psi)}{|Y \cap A_0|} \cdot |Z \cap A_0| + \frac{2w(\partial_G Z)}{\psi \cdot \tau} \leq \frac{320 \log_2 n}{\psi} + \frac{2}{\psi} \leq L \\ |Y \cap A_j| &\geq \frac{160 \log_2 n \cdot (2/\psi)}{|Y \cap A_0|} \cdot \left(1 - \frac{1}{5 \log n}\right)^{\log_2 n} \cdot |Y \cap A_0| \geq \frac{160 \log_2 n}{\psi} > 0, \end{aligned}$$

concluding the proof. \square

Runtime of Algorithm 13.

Claim 6.27. *Algorithm 13 takes time $m^{1+o(1)}$.*

Proof. It suffices to show each inner iteration takes time $m^{1+o(1)}$. Note that each iteration in Algorithm 13 calls COMPUTEXPANDERDECOMP and Algorithm 14 once. By Theorem 6.3, COMPUTEXPANDERDECOMP takes time $m^{1+o(1)}$. For Algorithm 14, we note that $L = \tilde{O}(1/\psi)$. It then follows from Theorem 3.5 that CONSTRUCTHITANDMISSFAMILY takes time $\tilde{O}(m^{1+o(1)}L^{O(1)}) = m^{1+o(1)}/\psi^{O(1)}$ and the size of \mathcal{H} is $m^{o(1)}/\psi^{O(1)}$. Since Algorithm 14 calls COMPUTEISOLATINGCUTS and SSMC once for each function in \mathcal{H} , and each call of COMPUTEISOLATINGCUTS and SSMC takes time $m^{1+o(1)}$. The total time cost of Algorithm 14 is $m^{1+o(1)}/\psi^{O(1)}$. \square

6.6 The Gomory-Hu Tree Algorithm

We conclude by presenting the final Gomory-Hu tree algorithm using the previous steps. We first give an algorithm GHTREESTEP (Algorithm 15) to compute a balanced isolating mincut partition using the results from the detection phase and the decomposition phase. Then, we use the deterministic steps in the GHTREE algorithm (Algorithm 16) in [AKL⁺22] to recursively construct the Gomory-Hu Steiner tree. Though [AKL⁺22] uses a randomized algorithm to select a pivot and compute the isolating mincut family, their recursive algorithm deterministically builds the Gomory-Hu Steiner tree and hence we can reuse part of their algorithm and analysis.

Algorithm 15: GHTREESTEP(G, U)

```

1 Use binary search on  $\tau$  to find the largest  $\tau$  such that the largest  $\tau$ -connected component
   w.r.t.  $U$  computed by DETECTCC( $G, U, \tau$ ) has size at least  $\frac{3}{4}|U|$ . /* This means the
   largest  $(\tau + 1)$ -connected component w.r.t.  $U$  has size  $< \frac{3}{4}|U|$ . */
2  $C \leftarrow \text{DETECTCC}(G, U, \tau)$ .
3  $r \leftarrow$  an arbitrary vertex in  $C$ .
4  $\mathcal{S}_{temp} \leftarrow \emptyset$ .
5 if  $|C| < \frac{15}{16}|U|$  then
6    $A_1 \leftarrow U \setminus C$ .
7   for  $i = 1, 2, \dots, C_{pathDecomp} \cdot (\log_2(n) + 1)^2$  do
8      $\mathcal{S}_1 \leftarrow \text{DECOMPFIRSTSTEP}(G, A_1, r, \tau)$ .
9      $\mathcal{S}_{temp} \leftarrow \mathcal{S}_{temp} \cup \mathcal{S}_1$ .
10     $A_1 \leftarrow A_1 \setminus \bigcup_{(S, v, r) \in \mathcal{S}_1} S$ .
11 else
12    $A_2 \leftarrow C \setminus \{r\}$ .
13   for  $i = 1, 2, \dots, C_{pathDecomp} \cdot (\log_2(n) + 1)^2$  do
14      $\mathcal{S}_2 \leftarrow \text{DECOMPSECONDSTEP}(G, A_2, r, \tau)$ .
15      $\mathcal{S}_{temp} \leftarrow \mathcal{S}_{temp} \cup \mathcal{S}_2$ .
16      $A_2 \leftarrow A_2 \setminus \bigcup_{(S, v, r) \in \mathcal{S}_2} S$ .
17    $\mathcal{S} \leftarrow \emptyset$ .
18    $R \leftarrow \emptyset$ .
19   foreach  $(S, v, r)$  in  $\mathcal{S}_{temp}$  in the descending order of  $|S|$  do
20     if  $S \cap R = \emptyset$  then
21        $\mathcal{S} \leftarrow \mathcal{S} \cup \{(S, v, r)\}$ .
22        $R \leftarrow R \cup S$ .
23 return  $r$  and  $\mathcal{S}$ .

```

Lemma 6.28. GHTREESTEP(G, U) returns a pivot r and a set \mathcal{S} such that

1. every triple $(S, v, r) \in \mathcal{S}$ has $v \in U \setminus \{r\}$ and S is the minimal (v, r) -mincut,
2. the mincuts S in \mathcal{S} are disjoint,
3. for every $(S, v, r) \in \mathcal{S}$, we have

$$|S \cap U| = (1 - \Omega(1))|U|,$$

and

4. we have that

$$\sum_{(S,v,r) \in \mathcal{S}} |S \cap U| = \Omega(|U|).$$

Proof. For Item 1, if the if-condition in Line 5 holds, then every triple $(S, v, r) \in \mathcal{S}_1$ has $v \in U \setminus C$ and S is the minimal (v, r) -mincut by Claim 6.22. So every triple $(S, v, r) \in \mathcal{S}$ is a triple in some \mathcal{S}_1 and thus has $v \in U \setminus C \subseteq U \setminus \{r\}$ and S is the minimal (v, r) -mincut. Otherwise, every triple $(S, v, r) \in \mathcal{S}_2$ has $v \in C \setminus \{r\}$ and S is the minimal (v, r) -mincut by Claim 6.26. So every triple $(S, v, r) \in \mathcal{S}$ is a triple in some \mathcal{S}_2 and thus has $v \in C \setminus \{r\} \subseteq U \setminus \{r\}$ and S is the minimal (v, r) -mincut.

For Item 2, the set R maintains the union of all mincuts in \mathcal{S} , so we will not add a triple (S, v, r) to \mathcal{S} if S intersects any of the mincuts in \mathcal{S} .

For Item 3, if the if-condition in Line 5 holds, every triple $(S, v, r) \in \mathcal{S}$ is a triple returned by Claim 6.22 and has $v \in A_1 = U \setminus C$ and S is the minimal (v, r) -mincut. Since C is a τ -connected component w.r.t. U , $r \in C$ and $v \in U \setminus C$, r and v are not τ -connected, so $S \cap U \subseteq U \setminus C$. Hence $|S \cap U| \leq |U \setminus C| \leq \frac{1}{4}|U|$. Otherwise, every triple $(S, v, r) \in \mathcal{S}$ is a triple returned by Claim 6.26 and has $v \in A_2 \cap B$ and S is the minimal (v, r) -mincut, so $|S \cap U| \leq \frac{15}{16}|U|$ by the definition of B .

For Item 4, we first show that

$$|\cup_{(S,v,r) \in \mathcal{S}_{temp}} S \cap U| = \Omega(|U|).$$

If the if-condition in Line 5 holds, since

$$|\cup_{(S,v,r) \in \mathcal{S}} S \cap A_{max}| \geq |A_{max}|/2$$

by Claim 6.22 and we remove the vertices contained by the mincuts in \mathcal{S} from A_1 after every iteration, after every $\log_2(n) + 1$ iterations, all the vertices in the previous A_{max} are removed from A_1 and hence $\max_{a \in A_1} d_{\mathcal{P}}(a)$ is decreased by at least 1. As $\max_{a \in A_1} d_{\mathcal{P}}(a) \leq C_{pathDecomp} \cdot \log_2 n$ initially, we have $A_1 = \emptyset$ after $C_{pathDecomp} \cdot (\log_2(n) + 1)^2$ iterations. This means every vertex in $U \setminus C$ is contained in some mincut S in some triple in \mathcal{S}_{temp} , where $|U \setminus C| \geq \frac{1}{16}|U|$. Otherwise, the proof is similar, except that we show every vertex in $(C \setminus \{r\}) \cap B = B$ is contained in some mincut S in some triple in \mathcal{S}_{temp} where $|B| \geq \frac{1}{8}|U|$.

Then we show that

$$|\cup_{(S,v,r) \in \mathcal{S}} S \cap U| = |\cup_{(S,v,r) \in \mathcal{S}_{temp}} S \cap U| = \Omega(|U|).$$

Every triple $(S, v, r) \in \mathcal{S}_{temp}$ has $v \in U \setminus \{r\}$ and S is the (v, r) -mincut by a proof similar to that of Item 1. By Fact 6.9(6), any $(S_1, v_1, r), (S_2, v_2, r) \in \mathcal{S}_{temp}$ satisfy either $S_1 \subseteq S_2$, $S_2 \subseteq S_1$, or $S_1 \cap S_2 = \emptyset$. If the if-condition in Line 20 holds, then there is some $(S', v', r) \in \mathcal{S}$ satisfying $S' \cap S \neq \emptyset$, which implies $S \subseteq S'$ since $|S| \leq |S'|$. Therefore

$$|\cup_{(S,v,r) \in \mathcal{S}} S \cap U| = |\cup_{(S,v,r) \in \mathcal{S}_{temp}} S \cap U| = \Omega(|U|).$$

The proof is completed by using Item 2 that the mincuts S 's in \mathcal{S} are disjoint. \square

Claim 6.29. $\text{GHTREESTEP}(G, U)$ takes deterministic time $m^{1+o(1)}$.

Proof. Each invocation of DETECTCC , DECOMPFIRSTSTEP or DECOMPSECONDSTEP takes time $m^{1+o(1)}$, and there are a total of $O(\log^2 n)$ invocations. The sum of sizes of mincuts in \mathcal{S}_1 or \mathcal{S}_2 is also $m^{1+o(1)}$. Hence the running time of GHTREESTEP is $m^{1+o(1)}$. \square

Let $V_S = \{v : (S_v, v, r) \in \mathcal{S}\}$.

Algorithm 16: $\text{GHTREE}(G, U)$ (c.f. Algorithm 4 in [AKL⁺22])

```

1 Call  $\text{GHTREESTEP}(G, U)$  to obtain  $r$  and  $\mathcal{S}$ .
2 foreach  $(S_v, v, r) \in \mathcal{S}$  do
3   Let  $G_v$  be the graph  $G$  with vertices  $V \setminus S_v$  contracted to a single vertex  $x_v$ .
4    $U_v \leftarrow S_v \cap U$  /*  $S_v$  are disjoint
5   if  $|U_v| > 1$  then
6     | Let  $(T_v, f_v) \leftarrow \text{GHTREE}(G_v, U_v)$ .
7 Let  $G_{large}$  be the graph  $G$  with (disjoint) vertex sets  $S_v$  contracted to single vertices  $y_v$  for
   $v \in V_S$ .
8  $U_{large} \leftarrow U \setminus \cup_{(S_v, v, r) \in \mathcal{S}} S_v$ .
9 if  $|U_{large}| > 1$  then
10  | Let  $(T_{large}, f_{large}) \leftarrow \text{GHTREE}(G_{large}, U_{large})$ .
11 Combine  $(T_{large}, f_{large})$  and  $\{(T_v, f_v) : v \in V_S\}$  into  $(T, f)$  using  $\text{COMBINE}$ .
12 return  $(T, f)$ .
```

Algorithm 17: $\text{COMBINE}((T_{large}, f_{large}), \{(T_v, f_v) : v \in V_S\})$ (c.f. Algorithm 5 in
 [AKL⁺22])

```

1 Construct  $T$  by starting with  $T_{large} \cup \bigcup T_v$  and, for each  $v \in V_S$ , adding an edge between
   $f_v(x_v) \in U_v$  and  $f_{large}(y_v) \in U_{large}$  with weight  $w_G(\partial_G S_v)$ .
2 Construct  $f : U \mapsto V$  by setting  $f(v') = f_{large}(v')$  for  $v' \in U_{large}$  and  $f(v') = f_v(v')$  for
   $v' \in U_v$ .
3 return  $(T, f)$ .
```

Lemma 6.30 (c.f. Lemma A.11 in [AKL⁺22]). *If every triple $(S_v, v, r) \in \mathcal{S}$ returned by $\text{GHTREESTEP}(G, U)$ has S_v is the minimal (v, r) -mincut, and the mincuts S_v 's are disjoint, then $\text{GHTREE}(G, U)$ outputs a Gomory-Hu Steiner tree.*

Claim 6.31. $\text{GHTREE}(G, U)$ outputs a Gomory-Hu Steiner Tree.

Lemma 6.28 and Lemma 6.30 together prove Claim 6.31.

Lemma 6.32 (c.f. Lemma A.15 in [AKL⁺22]). *If GHTREE has d recursion levels, then the total numbers of vertices and edges over all recursive instances are $O(dn)$ and $O(dm + dn \log n)$, respectively.*

Claim 6.33. $\text{GHTREE}(G, U)$ takes deterministic time $m^{1+o(1)}$.

Proof. As each recursive branch has a $(1 - \Omega(1))$ fraction of U by Lemma 6.28, GHTREE has maximum recursion depth $O(\log n)$. By Lemma 6.32, the total number of edges over all instances is $\tilde{O}(m)$. By Claim 6.29, an invocation on GHTREESTEP(G, U) with $|E(G)| = m_*$ takes deterministic time $m_*^{1+o(1)}$. Hence GHTREE(G, U) takes deterministic time $m^{1+o(1)}$. \square

7 References

- [ACZ98] Srinivasa Rao Arikati, Shiva Chaudhuri, and Christos D. Zaroliagis. All-pairs min-cut in sparse networks. *J. Algorithms*, 29(1):82–110, 1998. 4
- [AKL⁺22] Amir Abboud, Robert Krauthgamer, Jason Li, Debmalya Panigrahi, Thatchaphol Saranurak, and Ohad Trabelsi. Breaking the cubic barrier for all-pairs max-flow: Gomory-hu tree in nearly quadratic time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 884–895. IEEE, 2022. 1, 3, 5, 7, 8, 9, 12, 13, 14, 15, 17, 30, 31, 37, 52, 54
- [AKT20] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Cut-equivalent trees are optimal for min-cut queries. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*. IEEE Computer Society, 2020. 4, 6
- [AKT21a] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. APMF < apsp? gomory-hu tree for unweighted graphs in almost-quadratic time. In *Foundations of Computer Science (FOCS), 2021 IEEE 62nd Annual Symposium on*, 2021. 1, 4, 5
- [AKT21b] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Subcubic algorithms for gomory-hu tree in unweighted graphs. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC ’21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1725–1737. ACM, 2021. 1, 2, 3, 12
- [AKT22] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Friendly cut sparsifiers and faster gomory-hu trees. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 3630–3649. SIAM, 2022. 1
- [ALPS23] Amir Abboud, Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. All-pairs max-flow is no harder than single-pair max-flow: Gomory-hu trees in almost-linear time. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2204–2212. IEEE, 2023. 1, 2, 3, 5, 8
- [AMO93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice Hall, 1993. 1
- [Ben95] András A. Benczúr. Counterexamples for directed and node capacitated cut-trees. *SIAM J. Comput.*, 24(3):505–510, 1995. 1
- [BENW16] Glencora Borradaile, David Eppstein, Amir Nayyeri, and Christian Wulff-Nilsen. All-pairs minimum cuts in near-linear time for surface-embedded graphs. In Sándor P. Fekete and Anna Lubiw, editors, *32nd International Symposium on Computational Geometry, SoCG 2016, June 14-18, 2016, Boston, MA, USA*, volume 51 of *LIPICS*, pages 22:1–22:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. 4
- [BGS22] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. In

- 2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1000–1008. IEEE, 2022. [2](#), [8](#), [13](#), [14](#), [15](#)
- [BHKP07] Anand Bhalgat, Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. An $\tilde{O}(mn)$ gomory-hu tree construction algorithm for unweighted graphs. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 605–614. ACM, 2007. [4](#), [6](#)
- [BSW15] Glencora Borradaile, Piotr Sankowski, and Christian Wulff-Nilsen. Min st -cut oracle for planar graphs with near-linear preprocessing time. *ACM Trans. Algorithms*, 11(3):16:1–16:29, 2015. [4](#)
- [CCPS97] W.J. Cook, W.H. Cunningham, W.R. Pulleybank, and A. Schrijver. *Combinatorial Optimization*. Wiley, 1997. [1](#)
- [CKL⁺22] Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623. IEEE, 2022. [3](#), [5](#)
- [CKL⁺24] Li Chen, Rasmus Kyng, Yang P Liu, Simon Meierhans, and Maximilian Probst Gutenberg. Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, $s-t$ shortest path, and minimum-cost flow. *STOC’24*, 2024. [3](#)
- [CP24] Karthik Cs and Merav Parter. Deterministic replacement path covering. *ACM Transactions on Algorithms*, 20(4):1–35, 2024. [12](#)
- [Elm64] Salah E. Elmaghraby. Sensitivity analysis of multiterminal flow networks. *Operations Research*, 12(5):680–688, 1964. [1](#)
- [Gab91] Harold N Gabow. A matroid approach to finding edge connectivity and packing arborescences. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 112–122, 1991. [10](#), [29](#), [30](#)
- [GH61] Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961. [1](#), [5](#), [12](#)
- [GH86] Frieda Granot and Refael Hassin. Multi-terminal maximum flows in node-capacitated networks. *Discret. Appl. Math.*, 13(2-3):157–163, 1986. [1](#)
- [GI91] Zvi Galil and Giuseppe F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, 1991. [4](#)
- [GIK21] Loukas Georgiadis, Giuseppe F. Italiano, and Evangelos Kosinas. Computing the 4-edge-connected components of a graph in linear time. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPICS*, pages 47:1–47:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. [4](#)

- [GT01] Andrew V. Goldberg and Kostas Tsoukatos. Cut tree algorithms: An experimental study. *J. Algorithms*, 38(1):51–83, 2001. [1](#)
- [Gus90] Dan Gusfield. Very simple methods for all pairs network flow analysis. *SIAM J. Comput.*, 19(1):143–155, 1990. [1](#)
- [Har98] David Hartvigsen. The planar multiterminal cut problem. *Discret. Appl. Math.*, 85(3):203–222, 1998. [1](#)
- [Har01] David Hartvigsen. Compact representations of cuts. *SIAM J. Discret. Math.*, 14(1):49–66, 2001. [1](#)
- [Has88] Refael Hassin. Solution bases of multiterminal cut problems. *Mathematics of Operations Research*, 13(4):535–542, 1988. [1](#)
- [HDLT01] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001. [17](#)
- [HK95] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In Frank Thomson Leighton and Allan Borodin, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 519–527. ACM, 1995. [10](#), [23](#)
- [HKP07] Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Efficient algorithms for computing all low s - t edge connectivities and related problems. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 127–136. SIAM, 2007. [4](#)
- [HLRW24] Monika Henzinger, Jason Li, Satish Rao, and Di Wang. Deterministic near-linear time minimum cut in weighted graphs. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3089–3139. SIAM, 2024. [2](#), [3](#)
- [HM94] David Hartvigsen and Russell Mardon. The all-pairs min cut problem and the minimum cycle basis problem on planar graphs. *SIAM J. Discret. Math.*, 7(3):403–418, 1994. [4](#)
- [HM95] David Hartvigsen and François Margot. Multiterminal flows and cuts. *Oper. Res. Lett.*, 17(5):201–204, 1995. [1](#)
- [HS83] T. C. Hu and M. T. Shing. Multiterminal flows in outerplanar networks. *J. Algorithms*, 4(3):241–261, 1983. [1](#)
- [HT73a] John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973. [4](#)
- [HT73b] John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973. [4](#)

- [Kar00] David R Karger. Minimum cuts in near-linear time. *Journal of the ACM (JACM)*, 47(1):46–76, 2000. 1, 3, 7, 10, 29, 30
- [KMG24] Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. A dynamic shortest paths toolbox: Low-congestion vertex sparsifiers and their applications. *STOC’24*, 2024. 2, 8, 15
- [Kol22] Vladimir Kolmogorov. Orderedcuts: A new approach for computing gomory-hu tree. *CoRR*, abs/2208.02000, 2022. 4
- [Kor24] Tuukka Korhonen. Linear-time algorithms for k-edge-connected components, k-lean tree decompositions, and more. *CoRR*, abs/2411.02658, 2024. 4
- [Kos24] Evangelos Kosinas. Computing the 5-edge-connected components in linear time. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 1887–2119. SIAM, 2024. 4
- [KS93] David R Karger and Clifford Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 757–765, 1993. 3
- [KT15] Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic global minimum cut of a simple graph in near-linear time. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 665–674. ACM, 2015. 2, 3
- [Li21] Jason Li. Deterministic mincut in almost-linear time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 384–395, 2021. 2, 3, 10, 29
- [LNP⁺21] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrrachai Yingcharoenthawornchai. Vertex connectivity in poly-logarithmic max-flows. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 317–329, 2021. 2
- [LP20] Jason Li and Debmalya Panigrahi. Deterministic min-cut in poly-logarithmic max-flows. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 85–92. IEEE, 2020. 2, 3, 7, 12, 61
- [LPS21] Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. A nearly optimal all-pairs min-cuts algorithm in simple graphs. In *Foundations of Computer Science (FOCS), 2021 IEEE 62nd Annual Symposium on*, 2021. 1
- [LRW25] Jason Li, Satish Rao, and Di Wang. Congestion-approximators from the bottom up. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2111–2131. SIAM, 2025. 35
- [LS21] Jason Li and Thatchaphol Saranurak. Deterministic weighted expander decomposition in almost-linear time. *arXiv preprint arXiv:2106.01567*, 2021. 35, 61

- [Meh88] Kurt Mehlhorn. A faster approximation algorithm for the steiner problem in graphs. *Information Processing Letters*, 27(3):125–128, 1988. 8, 9
- [NI92] Hiroshi Nagamochi and Toshihide Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics*, 9:163–180, 1992. 4
- [NRSS21] Wojciech Nadara, Mateusz Radecki, Marcin Smulewicz, and Marek Sokolowski. Determining 4-edge-connected components in linear time. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPICS*, pages 71:1–71:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 4
- [Pan16] Debmalya Panigrahi. Gomory-hu trees. In *Encyclopedia of Algorithms*, pages 858–861. 2016. 4
- [Pat71] Keith Paton. An algorithm for the blocks and cutnodes of a graph. *Commun. ACM*, 14(7):468–475, 1971. 4
- [Sch03] A. Schrijver. *Combinatorial Optimization*. Springer, 2003. 1
- [ST81] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, 1981. 2, 9, 20, 25, 26, 40
- [SW19] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2616–2635. SIAM, 2019. 35
- [Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. 4
- [Tsi07] Yung H. Tsin. A simple 3-edge-connected component algorithm. *Theory Comput. Syst.*, 40(2):125–142, 2007. 4
- [Tsi09] Yung H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *J. Discrete Algorithms*, 7(1):130–146, 2009. 4
- [vdBCK⁺24] Jan van den Brand, Li Chen, Rasmus Kyng, Yang P Liu, Simon Meierhans, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-linear time algorithms for decremental graphs: Min-cost flow and more via duality. *to appear at FOCS’24*, 2024. 3
- [vdBCP⁺23] Jan van den Brand, Li Chen, Richard Peng, Rasmus Kyng, Yang P Liu, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 503–514. IEEE, 2023. 3, 12, 31, 62

- [vdBLL⁺21] Jan van den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and l1-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 859–869, 2021. 5
- [Zha21] Tianyi Zhang. Gomory-hu trees in quadratic time. *arXiv preprint arXiv:2112.01042*, 2021. 1, 7, 30, 31, 33, 34

A Expander Decomposition

In this section, we prove Theorem 6.3. The algorithm closely follows Appendix A of [LP20], which proves an expander decomposition with similar (but incomparable) guarantees. To be consistent with their notation, we prove the more general statement using the concept of expanders w.r.t. a vertex weighting \mathbf{d} .

Definition A.1 $((\phi, \mathbf{d})\text{-expander})$. Consider a weighted graph $G = (V, E, w)$ and a vector $\mathbf{d} \in \mathbb{R}_{\geq 0}^V$ of nonnegative entries on the vertices (the “demands”). The graph G is a (ϕ, \mathbf{d}) -expander if for all subsets $S \subseteq V$,

$$\frac{w(\partial S)}{\min\{\mathbf{d}(S), \mathbf{d}(V \setminus S)\}} \geq \phi.$$

Theorem A.2. Given a weighted graph $G = (V, E, w)$ and a demand vector $\mathbf{d} \in \mathbb{R}_{\geq 0}^V$ with entries in the range $\{0, 1, 2, \dots, \text{poly}(nW)\}$, there is a deterministic algorithm running in $\tilde{O}(m \cdot (mW)^\epsilon \log W)$ time that computes a partition \mathcal{X} of V such that

- for every $X \in \mathcal{X}$, $G[X]$ is $(\phi, \mathbf{d}|_X)$ -expander, and
- consider the flow problem where each vertex $v \in V$ for $v \in X \in \mathcal{X}$ has $w(E(\{v\}, V \setminus X))$ units of source mass and each vertex $u \in V$ has $\phi \cdot \gamma_{\text{expDecomp}} \cdot \mathbf{d}(v)$ units of sink capacity. Then, there exists a feasible flow f on the network G where each edge e has capacity $w(e) \cdot \gamma_{\text{expDecomp}}$.

Theorem 6.3 follows from Theorem A.2 with the weighting $\mathbf{d}(v) = 1$ for $v \in U$ and $\mathbf{d}(v) = 0$ for $v \notin U$. For the rest of this section, we prove Theorem A.2, closely following Appendix A of [LP20].

We use Corollary 2.5 of [LS21] as a black box.

Theorem A.3 (Corollary 2.5 of [LS21]). Fix any constant $\epsilon > 0$ and any parameter $\phi > 0$. Given a weighted graph $G = (V, E, w)$ and a demand vector $\mathbf{d} \in \mathbb{R}_{\geq 0}^V$ with entries in the range $\{0, 1, 2, \dots, \text{poly}(nW)\}$, there is a deterministic algorithm running in $\tilde{O}(m \cdot (mW)^\epsilon \log W)$ time that partitions V into subsets V_1, \dots, V_ℓ such that

1. Each graph $G[V_i]$ is a $(\phi, \mathbf{d}|_{V_i})$ -expander.
2. The total weight $w(E(V_1, \dots, V_\ell))$ of inter-cluster edges is $(\log n)^{O(1/\epsilon^4)} \phi \mathbf{d}(V)$.

We now prove Theorem A.2. Apply Theorem A.3 with $\epsilon = (\log n)^{-1/5}$ to obtain a partition V_1, \dots, V_ℓ with $w(E(V_1, \dots, V_\ell)) \leq \alpha \phi \mathbf{d}(V)$ for some $\alpha = (\log n)^{O(1/\epsilon^4)} \log W \leq \gamma_{\text{expDecomp}}/12$ in time $O(m \cdot \gamma_{\text{expDecomp}})$. We group the clusters V_i into a bipartition (A, B) as follows. If there is a (unique) V_i with $\mathbf{d}(V_i) > \mathbf{d}(V)/2$, then let $B = V_i$ and $A = V \setminus B$. Otherwise, start with

$A = \emptyset$ and greedily add clusters V_i into A until $\mathbf{d}(A) \geq \mathbf{d}(V)/4$. Since the last cluster added has $\mathbf{d}(V_i) \leq \mathbf{d}(V)/4$, we also have $\mathbf{d}(A) \leq 3\mathbf{d}(V)/4$. Let $B = V \setminus A$ so that $\mathbf{d}(B) \geq \mathbf{d}(V)/4$ as well.

We construct the following flow instance. Start with the graph $G[B]$, add a source vertex s , and for each vertex $v \in B$ with $E(\{v\}, V \setminus B) \neq \emptyset$, add an edge (s, v) of weight $\frac{1}{12\alpha}w(E(\{v\}, V \setminus B))$. Also, add a new vertex t , and for each vertex $v \in B$ with $\mathbf{d}(v) > 0$, add an edge (v, t) of weight $\frac{\phi}{2}\mathbf{d}(v)$. Let this graph be H , compute an $s-t$ max-flow/min-cut on graph H , and let f be the max-flow and $S \subseteq B \cup \{s\}$ be the min-cut ($s \in S, t \notin S$). Let $B' = B \setminus S$ and $A' = V \setminus B'$, forming a new partition (A', B') .

Claim A.4. *We have the following guarantees:*

1. $\mathbf{d}(B \setminus B') \leq \mathbf{d}(V)/6$.
2. If $G[B]$ is a (ϕ, \mathbf{d}) -expander, then $G[B']$ is a $(\phi/6, \mathbf{d}|_{B'})$ -expander.
3. For each $X' \in \{A', B'\}$, the following is true. Consider the flow problem where each vertex $v \in X'$ has $w(E(\{v\}, V \setminus X'))$ units of source mass and each vertex $u \in U$ has $\frac{\phi}{2}\mathbf{d}(v)$ units of sink capacity. Then, there exists a feasible flow f on the network G where each edge e has capacity $w(e)$.

If $B = V_i$ for some cluster V_i , then make a recursive call on $G[A']$. Otherwise, make recursive calls on both $G[A']$ and $G[B']$. In the former case, by Claim A.4, the graph $G[B']$ is a $(\phi/6, \mathbf{d}|_{B'})$ -expander. In both cases, we have $\mathbf{d}(A') \leq \mathbf{d}(A) + \mathbf{d}(V)/6 \leq 3\mathbf{d}(V)/4 + \mathbf{d}(V)/6$ and $\mathbf{d}(B') \leq \mathbf{d}(B) \leq 3\mathbf{d}(V)/4$, so the recursion depth is $O(\log(nW))$.

Let \mathcal{X} be the final partition output by the algorithm. Each edge cut by the final partition must belong to the cut of some (A', B') partition over the course of the algorithm, so for each vertex $v \in V$ in a final cluster $X \in \mathcal{X}$, the value $w(E(\{v\}, V \setminus X))$ is at most the sum of $w(E(\{v\}, V \setminus X'))$ over all sides $X' \ni v$ of some partition (A', B') encountered in the recursive algorithm. Consider summing the appropriate flows from Claim A.4 over all instances, forming a single flow. Using the fact that the recursion depth is $O(\log(nW))$ and instances are disjoint across any level, this flow satisfies the following: there is $O(\log(nW)) \cdot w(e)$ flow along each edge e , each vertex is the sink of $O(\log(nW)) \cdot \phi$ flow, and each vertex is the source is at least $w(E(\{v\}, V \setminus X))$ flow. Finally, remove enough flow until each vertex is the source is exactly $w(E(\{v\}, V \setminus X))$ flow, fulfilling the condition of Theorem A.2. The total running time is dominated by the calls to Theorem A.3 and max-flow, which take $m^{1+o(1)}$ time [vdBCP⁺23].

It remains to prove Claim A.4. For the first statement $\mathbf{d}(B \setminus B') \leq \mathbf{d}(V)/6$, observe that for each vertex $v \in B \setminus B'$ with $\mathbf{d}(v) > 0$, the edge (v, t) of weight $\frac{\phi}{2}\mathbf{d}(v)$ is cut. Therefore, $\frac{\phi}{2}\mathbf{d}(B \setminus B') \leq w(\partial_H S)$, and the min-cut is at most the degree cut

$$w(\partial_H \{s\}) = \sum_{v \in V} \frac{1}{12\alpha}w(E(\{v\}, V \setminus B)) \leq \frac{1}{12\alpha} \cdot w(E(V_1, \dots, V_\ell)) \leq \frac{\phi}{12}\mathbf{d}(V),$$

so we conclude that $\mathbf{d}(B \setminus B') \leq \mathbf{d}(V)/6$, as desired. For the second statement, suppose for contradiction that $G[B']$ is not a $(\phi/6, \mathbf{d}|_{B'})$ -expander. Then, there is a cut $U \subseteq B'$ with $w(\partial_{G[B']} U) \leq \frac{\phi}{6}\mathbf{d}|_{B'}(U)$. Since $G[B]$ is a $(\phi, \mathbf{d}|_{B'})$ -expander, $w(\partial_{G[B]} U) \geq \phi\mathbf{d}|_{B'}(U) = \phi\mathbf{d}(U)$. Taking the difference of the two inequalities gives $w(E(U, B \setminus B')) = w(\partial_{G[B']} U) - w(\partial_G U) \geq \frac{5\phi}{6}\mathbf{d}(U)$.

By max-flow/min-cut duality, the $s-t$ max-flow f sends full capacity of flow along each edge $e \in \partial S$ in the direction from S to $V \setminus S$. The edges $e \in \partial S$ can be partitioned into three types: the

edges e adjacent to s , the edges in $G[B]$, and the edges adjacent to t . Consider the edges of the first two types with (exactly) one endpoint in U . These edges have total capacity

$$\frac{1}{12\alpha}w(E(U, V \setminus B)) + w(E(U, B \setminus B')) = \frac{1}{12\alpha}w(E(U, V \setminus B)) + \frac{5\phi}{6}\mathbf{d}(U). \quad (3)$$

Flow f sends full capacity from S to U , and this flow must eventually reach t . It can escape U in two ways: through edges in $G[B']$ and through edges adjacent to t . The total capacity of these edges is

$$w(\partial_{G[B']} U) + \frac{\phi}{2}\mathbf{d}(U) \leq \frac{\phi}{6}\mathbf{d}(U) + \frac{\phi}{2}\mathbf{d}(U) = \frac{2\phi}{3}\mathbf{d}(U). \quad (4)$$

Using that $w(E(U, V \setminus B')) = w(E(U, V \setminus B)) + w(E(U, B \setminus B'))$ and comparing term by term, we conclude that expression (3) is strictly larger than expression (4), which means the flow entering U cannot completely escape U , a contradiction.

For the third statement, recall that by max-flow/min-cut duality, the $s-t$ max-flow f sends full capacity of flow along each edge $e \in \partial S$ in the direction from S to $V \setminus S$. Suppose first that $X = B'$, and consider the restriction of the flow to the subgraph $G[B']$ of H . Each vertex $v \in B'$ is the new source of exactly $\frac{1}{12\alpha}w(\{v\}, V \setminus B')$ flow, since the full capacity flow from $V \setminus B'$ to v was removed by the restriction. Each vertex $v \in B'$ is the new sink of at most $\frac{\phi}{2}\mathbf{d}(v)$ flow, since the removed edge (v, t) has weight $\frac{\phi}{2}\mathbf{d}(v)$. This flow scaled up by $12\alpha \leq \gamma_{expDecomp}$ immediately satisfies the third statement, and the case $X = A'$ is analogous with the flow reversed.